

FUZE⁴

 **NINTENDO
SWITCH™**

**FUZE⁴ Nintendo Switch
Programmer's Reference Guide
V 0.2.11**

By David Silvera & Colin Bodley

CODING & COMPUTING WORKSHOPS

for Schools, Academies, Colleges, Universities, Computing Clubs, Holiday Camps & Special Events

www.fuze.co.uk

To contact FUZE call +44 (0) 1844 239 432
(UK 09:30 am to 17:00 pm weekdays)
or email contact@fuze.co.uk

Published in the United Kingdom ©2019 by FUZE Technologies Ltd. FUZE, FUZE⁴, FUZE logos, designs, documentation and associated materials are Copyright FUZE Technologies Ltd. FUZE is a UK registered trademark [UK00002655290].

No part of this document may be copied, reproduced and or distributed without written consent from FUZE Technologies Ltd. Nintendo and Nintendo Switch are copyright of Nintendo. Any other brand names are the copyright of their respective owners. All rights reserved.

FUZE⁴ Nintendo Switch is developed by FUZE Technologies Ltd in the UK by;

Jon Silvera - Project manager and CEO

Luke Mulcahy - Lead programmer

Will Tice – Programmer and Graphic Artist

Mike Green – Dev supervisor and programmer

David Silvera – Help content author, Sound engineer & lead Tutor

Kat Deak – 3D Graphic Artist

Colin Bodley - Technical consultant & Help content author

Martin White - Technical consultant

Nic Baxter – Sales Guy

Ben Taylor / Lizzie Botelle – Product testing and FUZE helpers

Join the FUZE community – share your projects, get tips and
help from the experts and help others

www.fuzearena.com

Getting started

Introduction	5
Using FUZE	10
Code Editor	12
Map Editor	15
Image Editor	25
Keyboard Shortcuts	37

Keywords	39
-----------------	-----------

Operators

About Operators	84
-----------------	----

Command reference	120
--------------------------	------------

2D Graphics	121
3D Graphics	307
Arithmetic	363
Binary	412
File Handling	420
Input	427
Screen Display	437
Sound and Music	449
Text Handling	477
Time and Date	502

Quick reference guide	517
------------------------------	------------

About	511
--------------	------------

Tutorials

Loops	529
Variables	532
If Then Statements	535
Screen	538
Arrays	547
Using Controls	550
For Loops	556
Functions	560
And Or Not	564
Variables Extended	569
Drawing Images	572
Structures	578
Making Music	584
Vectors	596
Sprites	601
Sprite Game	606
3D Simple Shapes	618
3D Simple Lighting	625
3D Simple Rotation	629
3D Camera Movement	633
Game – Introduction	639
Game – The Background	639
Game – The Level	648
Game – The Character	656
Game – Movement	664
Game – Animation	670
Game – Items	677
Game – Enemies	684
Game - Customise	692

GETTING STARTED

GETTING STARTED

Epilepsy Warning

This program allows users to create flashing images.

Some people may experience a seizure when exposed to certain visual images, including flashing lights or visual patterns.

The risk of photosensitive epileptic seizures may be reduced by taking the following precautions:

Use in a well-lit room.

Do not use if you are drowsy or fatigued.

View with greater distance from the screen so that it fills less of one's field of vision.

Should you experience any sensations of light-headedness, dizziness, nausea, altered vision, eye or face twitching, jerking or shaking of arms or legs, disorientation, confusion, or momentary loss of awareness, then please immediately stop playing and consult a doctor.

Introduction

Hello! Welcome to **FUZE⁴ Nintendo Switch**. Congratulations on your awesome purchase!

If you're new to coding and not sure where to start, you've come to the right place! In this introduction we'll be covering a few important things to bear in mind whilst using **FUZE**.

If you are an experienced programmer already, we recommend that you jump right in and flex your coding muscles! You can find a detailed description of every function, keyword and operator in the Command Reference.

Coding is an incredible skill. Every electronic device, video game and piece of software you've ever used all run on code. The world would be a very different place without it.

Before we dive in to some details about the software let's get a few things out of the way.

Learning to code is like learning a super power. You can find it difficult to get things right at first, you will make many mistakes, you might even think your new powers aren't working for you. Just like with everything else in life, practice makes perfect.

Keep honing your skills, figuring out problems and completing projects. Soon you'll be a coding super hero!

Improvement takes time, so don't be disheartened if you get things wrong! Every error (or bugs as we call them) you fix makes you a better programmer.

FUZE⁴ Nintendo Switch gives you a handheld environment to create anything you want. But where to begin?

Well, here are a few important things to keep in mind.

Computer Languages

You may have heard that computers are very complicated things. This is true to an extent, but one also say that in some ways they are very simple and logical.

A computer **understands two things**. *On*, and *Off*. *True* and *False*. *1* and *0*.

A computer's brain is called the **CPU**. It is made of lots and lots of switches that are Simply *on* or *off*.

Where it gets complicated is that there really are lots of them. Billions and billions actually.

Whenever you use a computer to play a game, message your friends, surf the net or do your homework, what you are doing is *changing billions and billions of switches* incredibly quickly.

How does your computer manage to do all these things using *just 1's and 0's*? A very clever chap called Gottfried Leibniz invented something called the **Binary Number System**, and it is vital to all computers. We'll talk about **binary** in more detail later.

Now, it would be quite boring and incredibly difficult to write a whole program using 1's and 0's, so some incredibly clever people developed **computer languages** to "speak" to the computer in a way which makes more sense to us.

There are **many computer languages** out there in the world. Thousands and thousands of them. They're very different to each other and are designed for different reasons.

The **language** you will be using here is called **FUZE**!

FUZE is similar to **languages** you might use if you study Computer Science at school, or even those used by professional computer programmers. However, with FUZE we have put great emphasis on making things simple and intuitive to learn and use.

Formatting

When writing code we can lay it out in lots of different ways, we call this **formatting**.

Here's an example of some messy code below:

```
1. loop
2.           ink(fuzepink)
3. print("HELLO" )
4.
5.           update(           )
5.     sleep( 1)
6.
7.     repeat
```

We have lots of random spaces and blank lines for no reason, **but**, and it's important to say this, the code above will work *just fine*.

The program is a small loop, we are printing the word "HELLO" in a nice pink colour, waiting for 1 second then repeating the loop.

The same code formatted differently will work *exactly the same*. See the example below:

```
1. loop ink(fuzepink) print("HELLO") update() sleep(1) repeat
```

It doesn't matter to FUZE whether your program is written on just one line, but as your programs get bigger and more complex it will make finding bugs and editing your code much more difficult!

How we format our code will have a big effect on how we read it, so learning at the start how to format our code correctly will help us greatly in the future.

Compare the previous examples with the formatting of the code below:

```
1. loop
2.     ink(fuzepink)
3.     print("HELLO")
4.     update()
5.     sleep(1)
6.     repeat
```

Doesn't that look neater, some might say more logical?

Now we can easily see where our loop begins and ends, and everything in between is **tabbed**.

Remember: this code above will **run** in *exactly the same way* as the previous examples. The only difference is that it is easier to read and edit.

The projects and tutorials in FUZE are all **formatted** this way. You don't have to copy us, in fact you might want to make up your own style of **formatting** that works for you!

Syntax

Here's a strange word which you may or may not know!

Syntax describes the **structure of statements** in a computer language. Certain statements must be structured in a particular way if we want them to work. Take the example below:

```
1. prin t("see the mistake?")
```

Can you spot the mistake we made?

We've made a crucial **syntax** error. We put a space between the "prin" and the "t" in the word "print". FUZE will read this as **two** separate statements.

If we correct this **syntax** error, we get:

```
1. print("see the mistake?")
```

Now FUZE knows exactly what to do and we'll get no error.

Let's see another very similar example:

```
1. print("what about now?")
```

This one might be trickier to see, but we've made another **syntax** error.

Can you spot it?

We're missing a " before the last bracket! Without it, FUZE will be confused by what you want to print.

Certain statements must be laid out in a particular way, and this is often to do with punctuation. *Always* double check the placement of your commas, speech marks and brackets.

Functions and Keywords

During the tutorials you'll be seeing a few words again and again.

Something you'll need to know for the upcoming projects is what we mean by a **function**. We'll go into more detail further into the tutorials, but for now, take a look at the line of code below:

```
1. print("Print is a function.")
```

When we want to print words on the screen, we use the `print()` **function**.

See the brackets after the word `print`? This is how you know we're using a **function**! In **FUZE**, **functions** will also appear in a light blue colour.

Functions usually need some information in the brackets to work. For example, the `print()` **function** needs something in the brackets to print on the screen.

Here's another:

```
1. ink(green)
```

This one is called the `ink()` **function**. We use it to change the colour of text on the screen. This time, we put a colour in the brackets!

When you see the name of a **function** in the FUZE tutorials, it will have brackets after it, like this: `print()`

This is to make it as clear as possible that **functions** *always* have brackets after them.

Keywords are a little different. They do not use brackets, and in **FUZE⁴ Nintendo Switch** they appear as a red/pink colour. Take a look below:

```
1. loop
2. repeat
```

The two lines above are an empty **loop**. `loop` and `repeat` are **keywords**.

Keep your eyes peeled in the upcoming projects to see clearly which parts are **functions** and which parts are **keywords**.

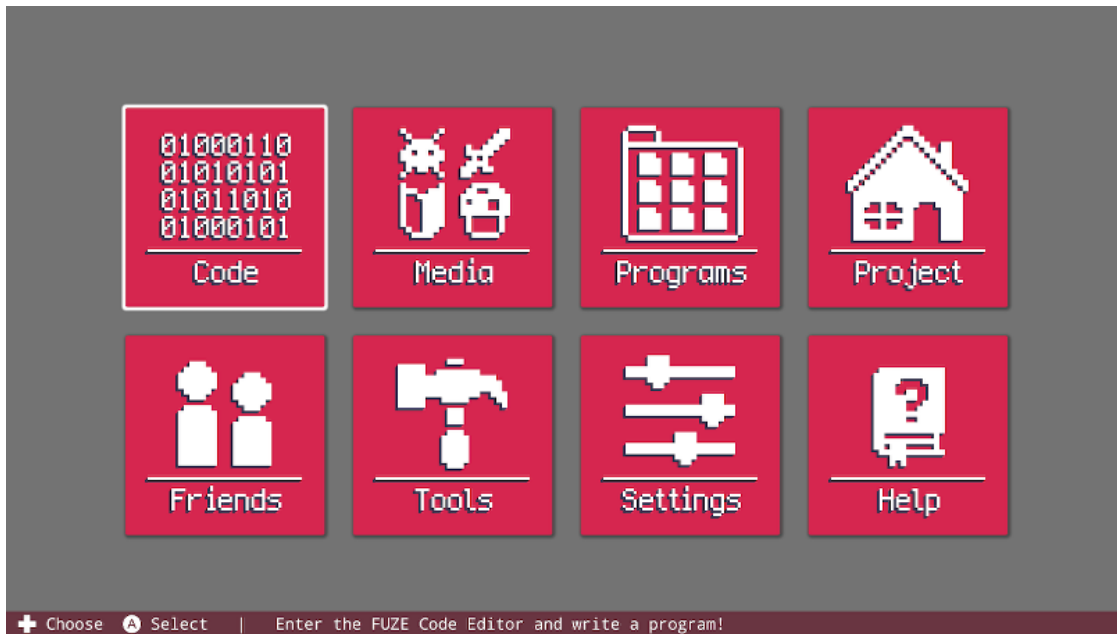
Let's get started!

Well done for reading all the above information. You are now fully equipped to get started with the tutorial projects and begin your journey to coding mastery! Looking forward to seeing you in the tutorials, let's go and have fun coding!

GETTING STARTED

Using FUZE

When you first load **FUZE⁴ Nintendo Switch** you will be greeted by the main menu screen:



From here you can access all of the tools and features FUZE has to offer.

Command Bar

Before we take a look at each menu item, take a closer look at the bottom of your screen:



This helpful little bar lives at the bottom of every screen in **FUZE⁴ Nintendo Switch**. It's called the **Command Bar**, and it tells you all of the controls you currently have access to.

Be sure to check the **Command Bar** for guidance if you're stuck with the controls!

Code

With the selection cursor on the 'Code' button, press the A button on your Joy-Con controller (Enter key on USB keyboard) to be taken to the Code Editor.

The Code Editor is where you'll be spending most of your time with FUZE. This is the place where we will actually write a program! From the Code Editor, you can also access the Media Browser to load assets, or check out the tutorials.

There are lots of controls to get used to in the Code Editor - Keep your eye on the **command bar** at the bottom of screen!

Media

Pressing A on the 'Media' button will take you to the **FUZE⁴ Nintendo Switch** Media Browser.

From here you can browse the vast collection of visual and audio assets at your disposal. Perhaps you'll stumble across something awesome for your next game idea!

Programs

Selecting the 'Programs' button from the Main Menu will take you to the existing programs in **FUZE⁴ Nintendo Switch**. From here you can start a new project, load one of the included demo programs or share or load one of your own projects. You'll have to make some first of course!

Project

The **FUZE⁴ Nintendo Switch** Project Menu is where we'll find information on the currently loaded project. You can edit your project details, manually save and begin new projects here.

Friends

Selecting the 'Friends' option from the Main Menu will take you to a page displaying all of your Nintendo Switch friends. If they have **FUZE⁴ Nintendo Switch** themselves, you can download their shared projects from this screen.

Tools

Want to make your own level using the assets? Perhaps you want to create your own assets to use! Either way, the Tools menu is the place you need.

Pressing the 'Tools' button will take you to a selection between the Map or Image Editor. Use the Map Editor to create level maps from the assets in **FUZE⁴ Nintendo Switch**. Use the Image Editor to create your own assets!

NOTE: Maps and images created using the map editor or image editor are saved **into a specific project**. When you open either the map or image editor you will be prompted to select a project first.

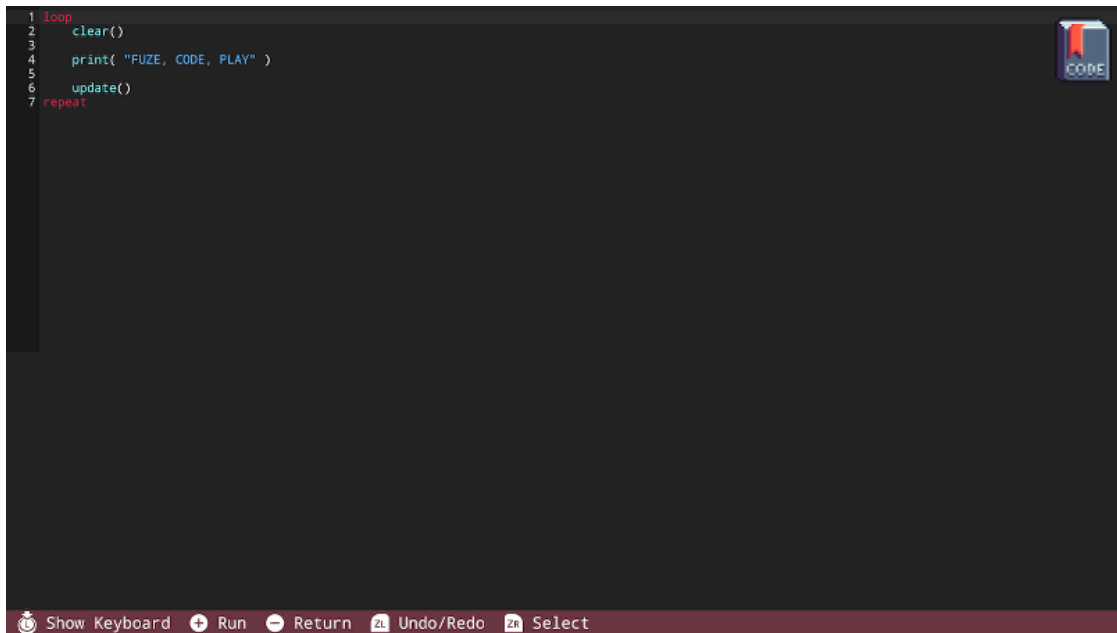
Settings

As you might imagine, the 'Settings' button on the Main Menu will take you to the **Fuze⁴ Nintendo Switch** settings page. From here you can adjust FUZE to your heart's content, changing the appearance and behaviour of the application.

CODE EDITOR

Code Editor

To enter the code editor select the 'Code' button from the Main Menu, it looks like this:



```
1 loop
2   clear()
3
4   print( "FUZE, CODE, PLAY" )
5
6   update()
7 repeat
```

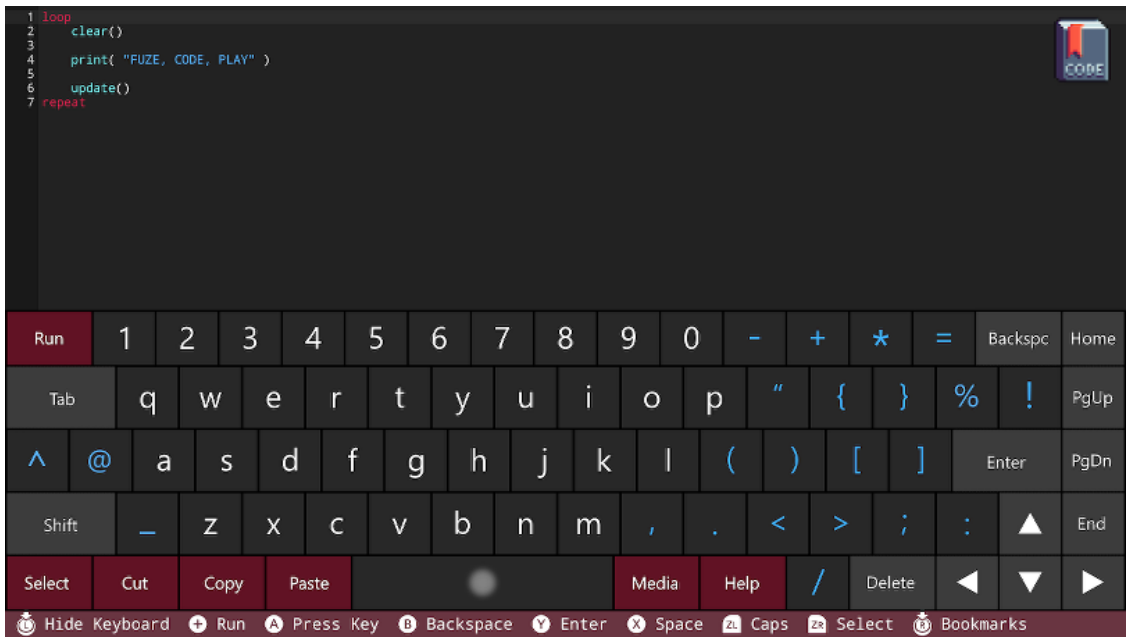
The screenshot shows a code editor window with a dark background. The code is displayed in a light color. At the bottom of the window, there is a command bar with several buttons: 'Show Keyboard', 'Run', 'Return', 'Undo/Redo', and 'Select'. A small 'CODE' icon is visible in the top right corner of the editor window.

Take a look at the **command bar** at the bottom of the screen. Here you can find button prompts to help you navigate the editor.

Press the + button to **run** a program. Press the - button to return to the Main Menu.

On-Screen Keyboard

Press in the left control stick to open the keyboard:

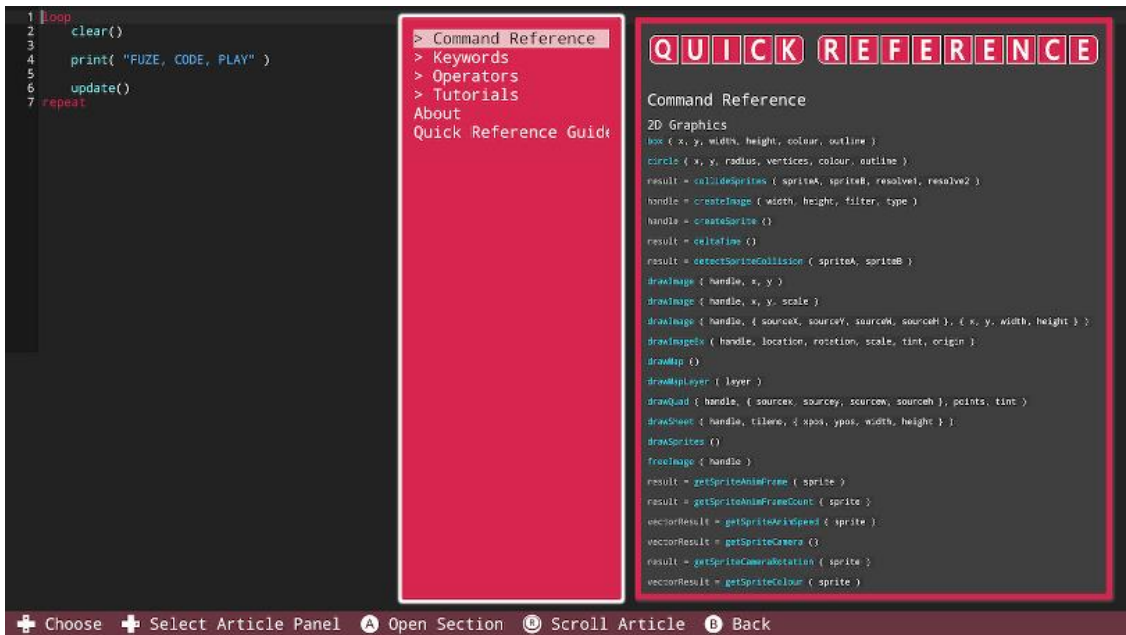


Check the **command bar** at the bottom of the screen and you will see it has changed to display the controls for the on-screen keyboard. Make sure to refer to the **command bar** if you're not sure how to control FUZE!

With the keyboard on screen, moving the left control stick will allow you to select keys to press. Use the A button to press a key.

Pressing the 'Media' button on the keyboard is a fast and easy way to access the Media Browser if you're looking for assets to use.

Similarly, pressing the 'Help' button on the keyboard will open the in-editor Help menu:



When the in-editor Help menu is open, notice the **command bar** has changed to display the controls for the Help Menu. From here you can navigate to any page you want to view, and view it with your code on screen.

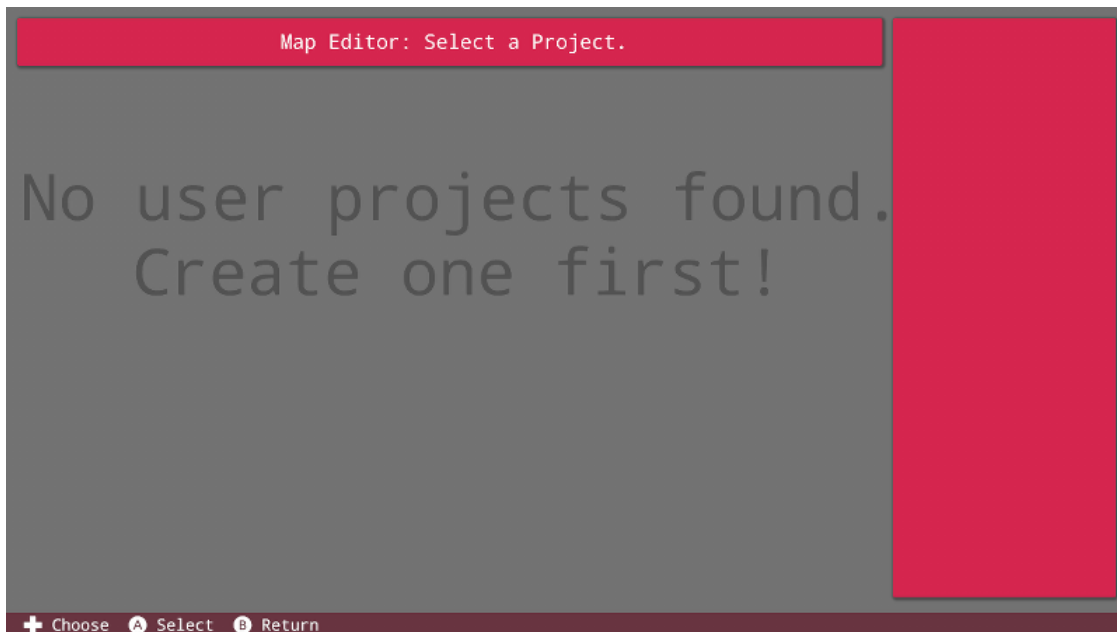
If you want to enter or edit code with the Help Menu on screen, click in the left control stick to open the keyboard.

MAP EDITOR

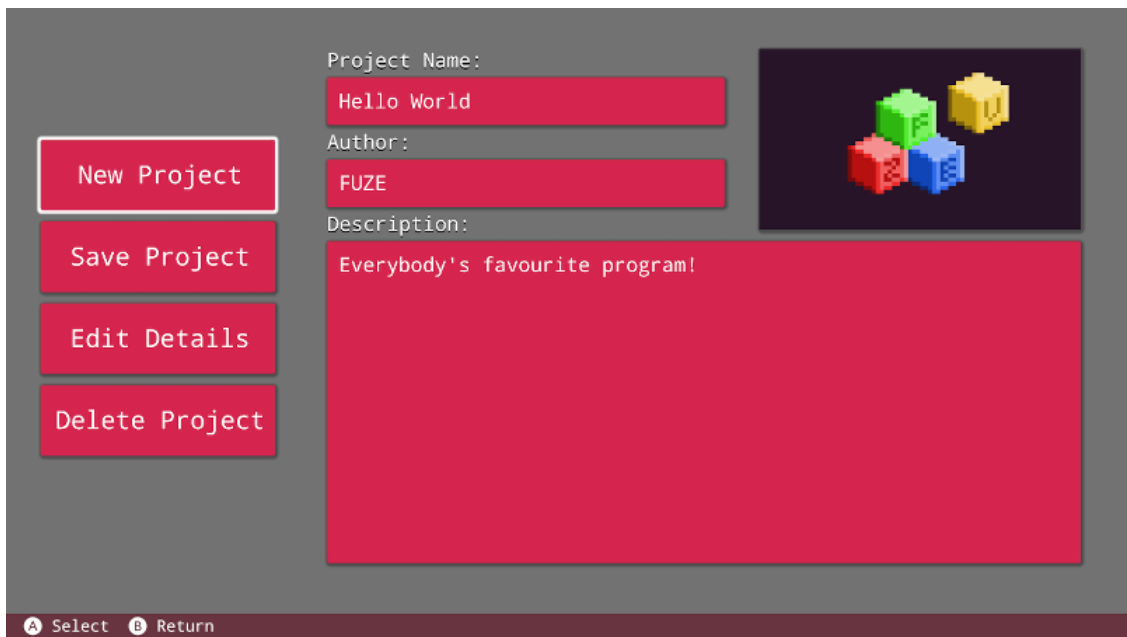
Map Editor

<>

The FUZE⁴ Nintendo Switch map editor is designed to make it easier to create your own maps. Access the Map Editor by going to the Tools icon from the Main Menu, then select 'Map Editor' and if it's your first time using FUZE you'll see the screen shown below:



As we have no projects in our user save data, there is nothing for us to do. Let's create a project so we can store a map. Return to the Main Menu and click the 'Project' button. You'll be taken to this screen shown below:



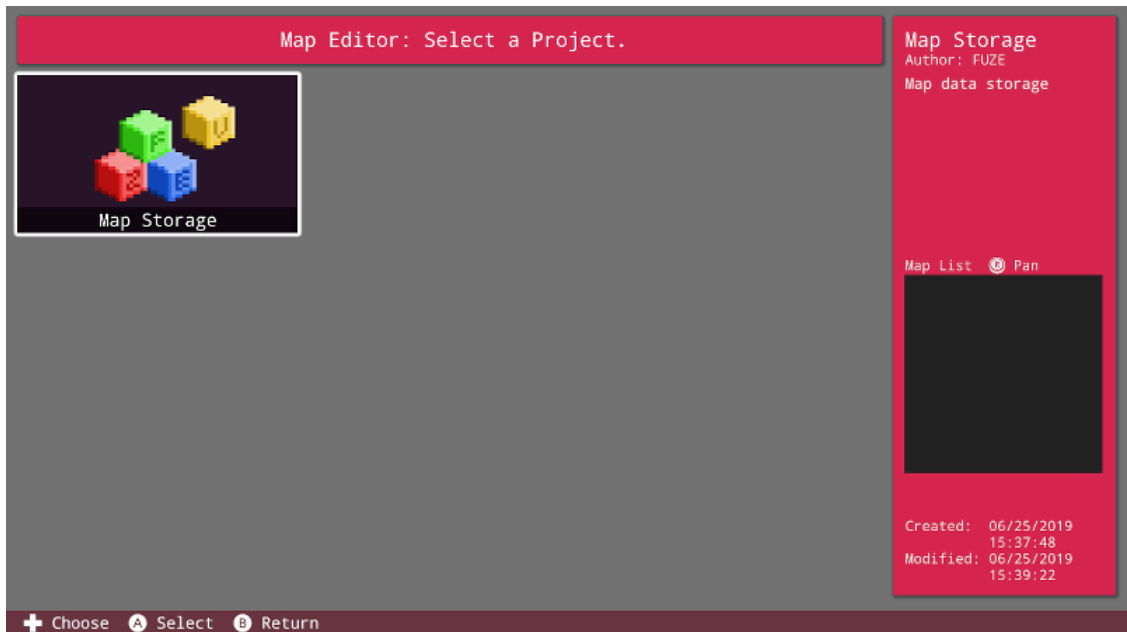
what we have here are the project details for the default **FUZE** program, 'Hello World'.

Select the 'New Project' button on the left to create a new project file in our user save data. This will allow us to start creating maps for that project.

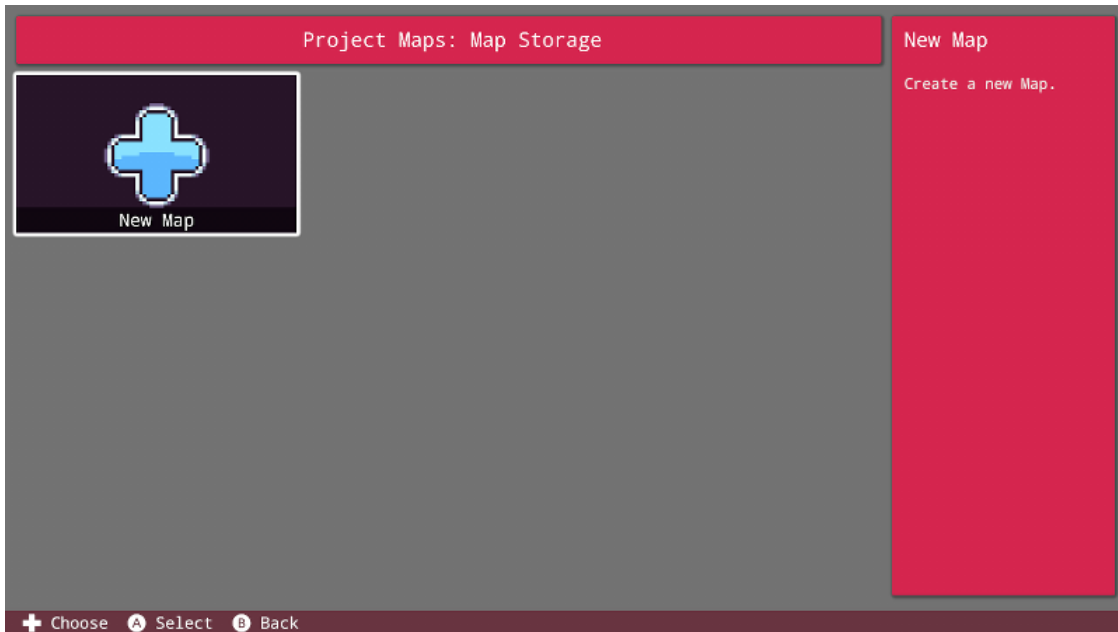
Enter the title, author and description for your first project.

It is a good idea to create a project in which to store all of your map data going forward. This will be a handy way of knowing where all of your maps are for future reference. Why not call it something like 'Map Storage' or 'Atlas'?

Once you've created the project you'll be taken to Code Editor. Return to the Main Menu using the minus button on the Joy-Con controller, then click the 'Tools' icon followed by 'Map Editor':



As you can see, we can now see our newly created project. Click the project icon and you'll be taken to the next window:

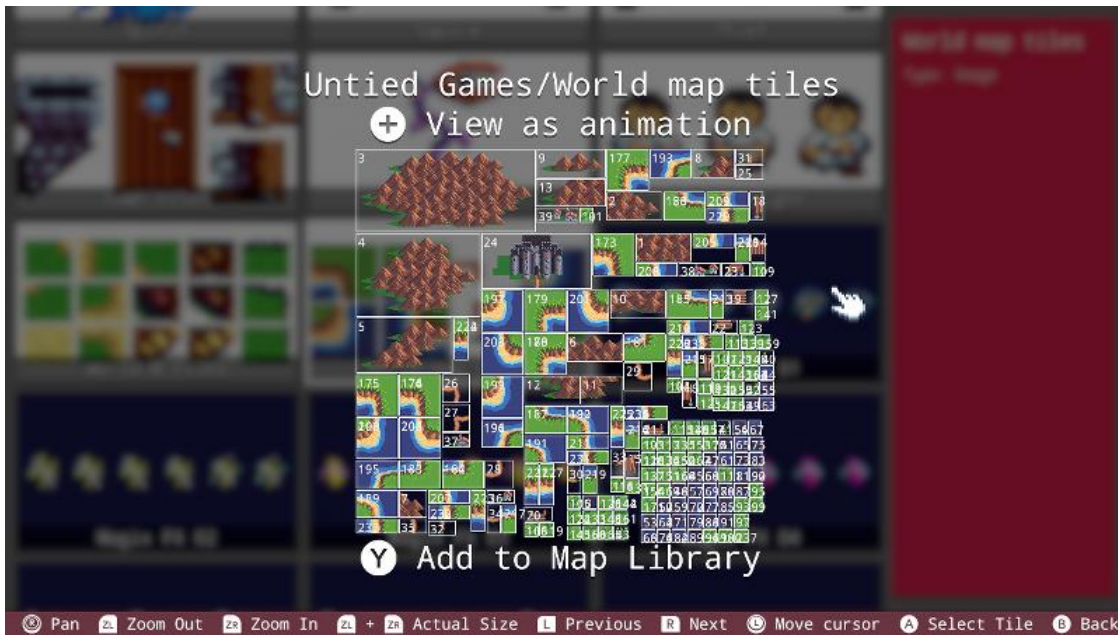


This window is where we can see all the maps stored within this project. Since we don't have any yet, let's click the 'New Map' button to get started!

You will be prompted to enter a name for your map, then press the plus button to confirm.

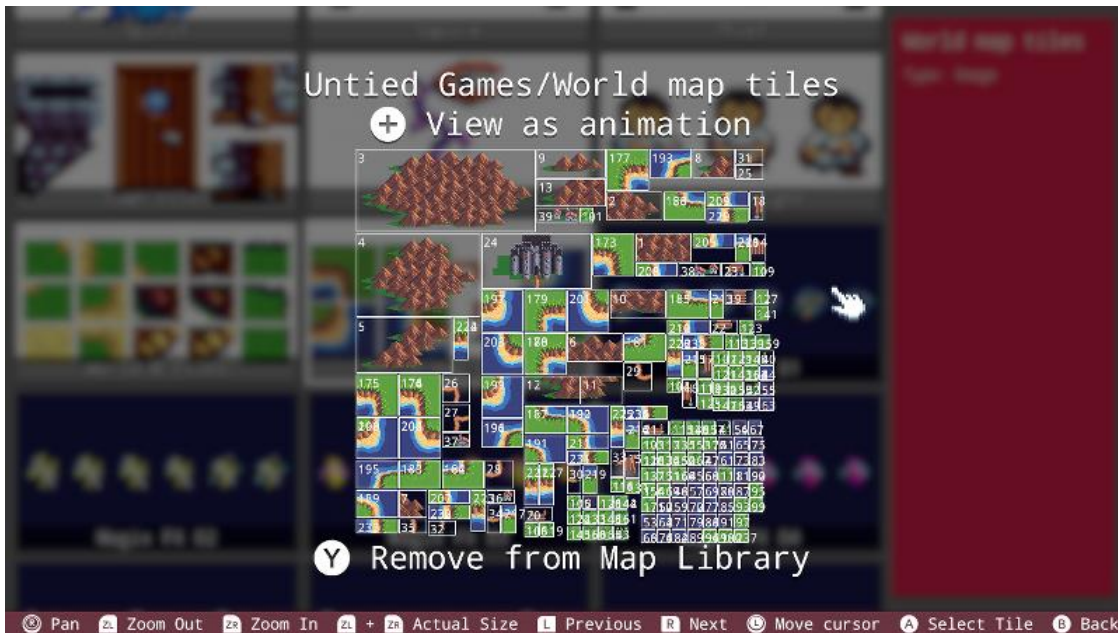


Name your map and press the plus button. You will see a message reading "Choose assets to use in your map!". Pressing okay will take you to the **FUZE** Media Browser to select some assets. For this example, we'll be using some assets by the very talented 'Untied Games'. Select the 'Untied Games' artist icon in the media browser:

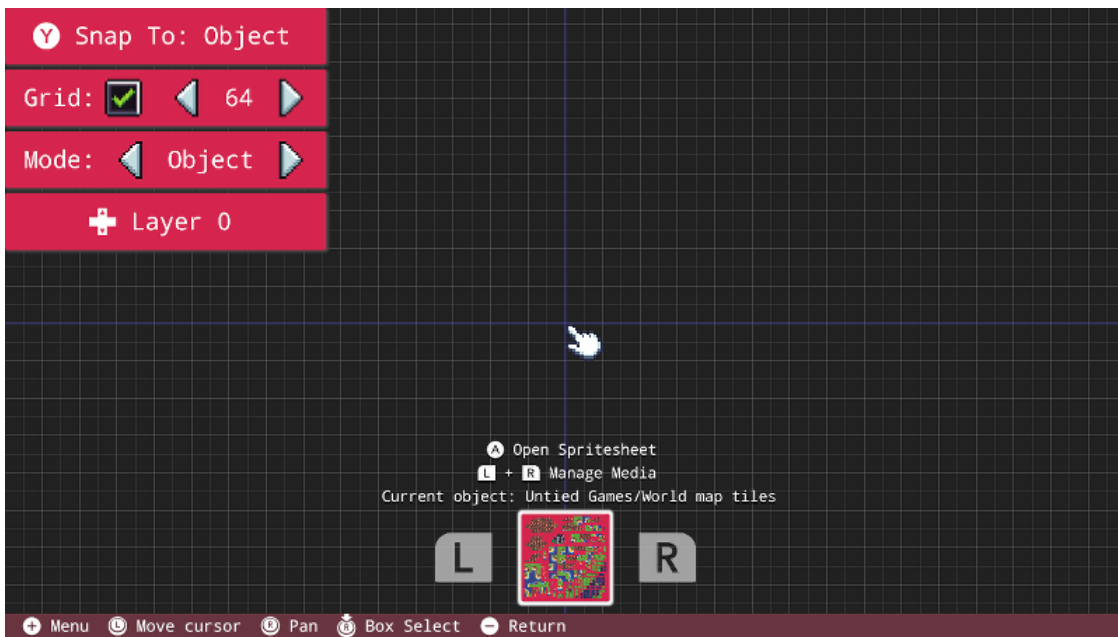


Scroll down and you'll find the tilesheet called "Untied Games/World map tiles". Open this tilesheet by pressing the A button, then press the Y button to add this asset to your map library as shown at the bottom of the screen.

Once the Y button is pressed, notice that the text at the bottom of the screen now reads differently, we can press the Y button once more to remove this asset from our Map Library.

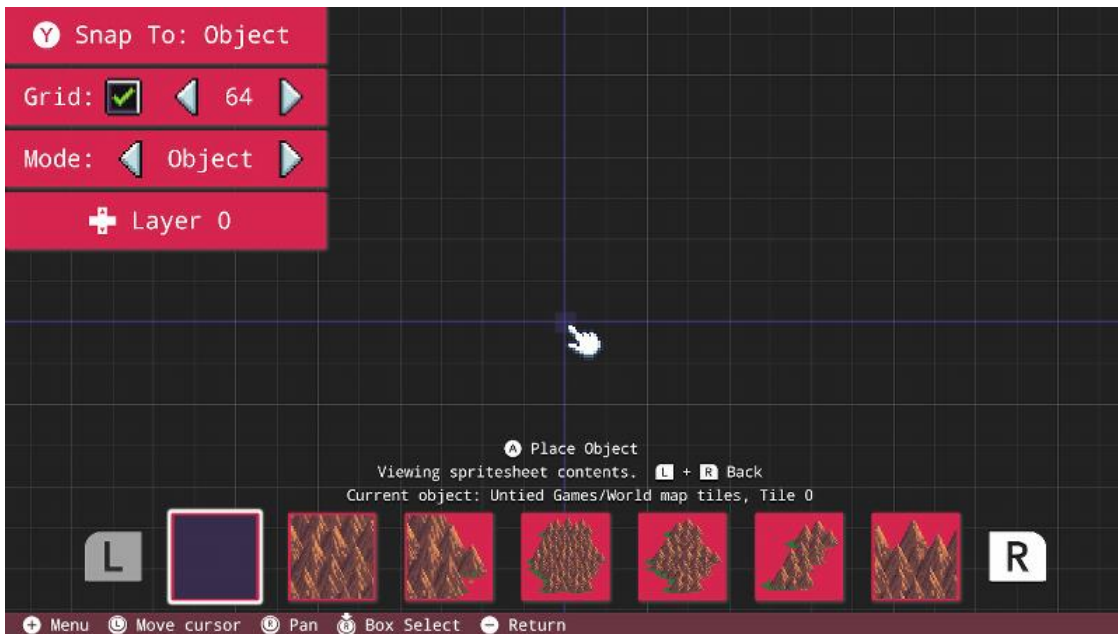


We now have the option to go back into the Media Browser and to keep adding assets to use in our map design. For now, we'll stick with this tilesheet. Press the B button to go back, then press the plus button to enter the map editor:



Here we are! Now we're in the Map Editor, take a look at the bottom of our screen. We have two options. We can add more assets to our library by pressing the L and R buttons at the same time. We can also open the currently selected spritesheet to begin drawing with those assets.

Since we only have one tilesheet loaded into our library, we can only see one option at the bottom of the screen. If we had more assets loaded into the Map Library, these would appear here. Let's open the tilesheet to begin building. Press the A button to open the tilesheet:



Before we start placing tiles, it's important to take a look at the control options in the top left corner of the screen. Pressing the Y button will change the way the tiles snap together. Snapping to 'Object' will allow you to place tiles perfectly next to each other. Snapping to 'Grid' will force the tiles into strict grid positions, whilst snapping to 'None' allows for totally free placement.

The 'Grid' option just beneath is the size of your grid squares. A higher number will allow for more precise grid placement.

The 'Mode' option just beneath that allows you to change the editing mode. By default, we are in Object mode. This means we are simply placing tiles. By pressing the arrows on this box we can change our edit mode to Collision. This will become useful when we actually have a map! So let's make one.

The last option to touch on here is just beneath the Mode selection. By using the up and down directional buttons, you can change the layer upon which you place tiles. This is great for building complex maps.

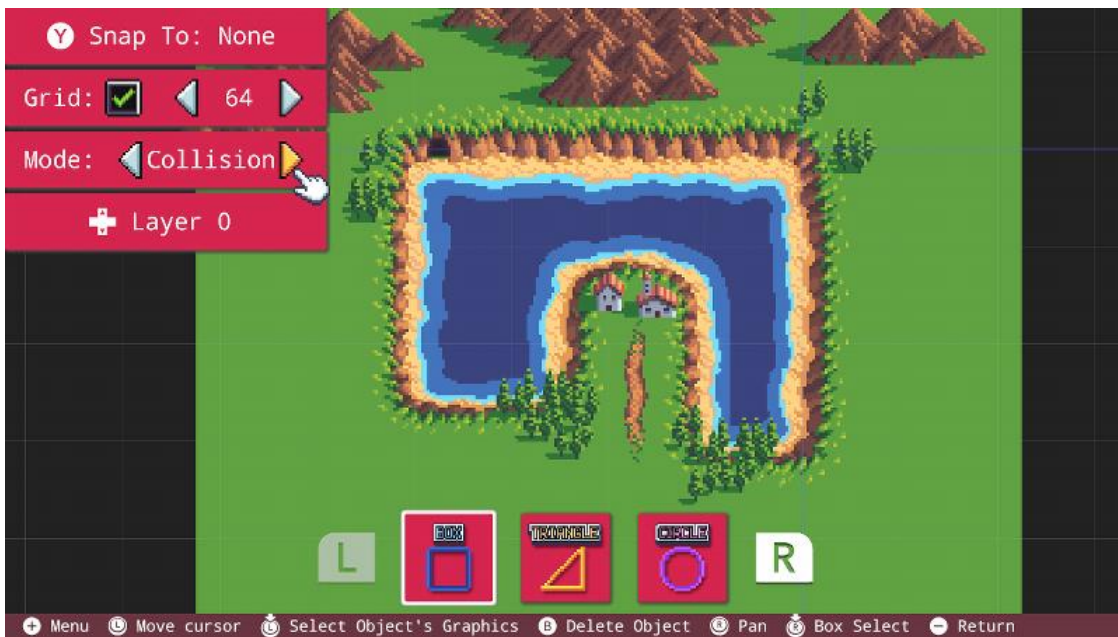
Once the tilesheet is opened we will be able to see all of the individual tiles along the bottom of the screen. Use the L and R buttons to navigate to the tiles you want.

If you place a tile and would like to delete it, simply move the cursor highlight over the tile and press the B button.



Here we have a small world map section we could potentially use. Let's add some collision data using the Collision edit mode.

Move your cursor to the arrows displayed on the top left boxes and press the A button to change to Collision mode:



Notice the bottom of the screen has changed now we're in Collision mode. Now we can place boundary boxes in the areas we want to behave in certain ways. For example, we might make it so that the player cannot enter the water. To achieve this easily, we can simply draw collision boxes around the areas:



It's very useful in Collision mode to use the 'Object' snapping mode. Move the cursor to the area you want your collision to begin, then press the A button to start placing a box.

Moving the cursor around will adjust the position of your box. Use the direction buttons to fine-tune the size of your collision box precisely.

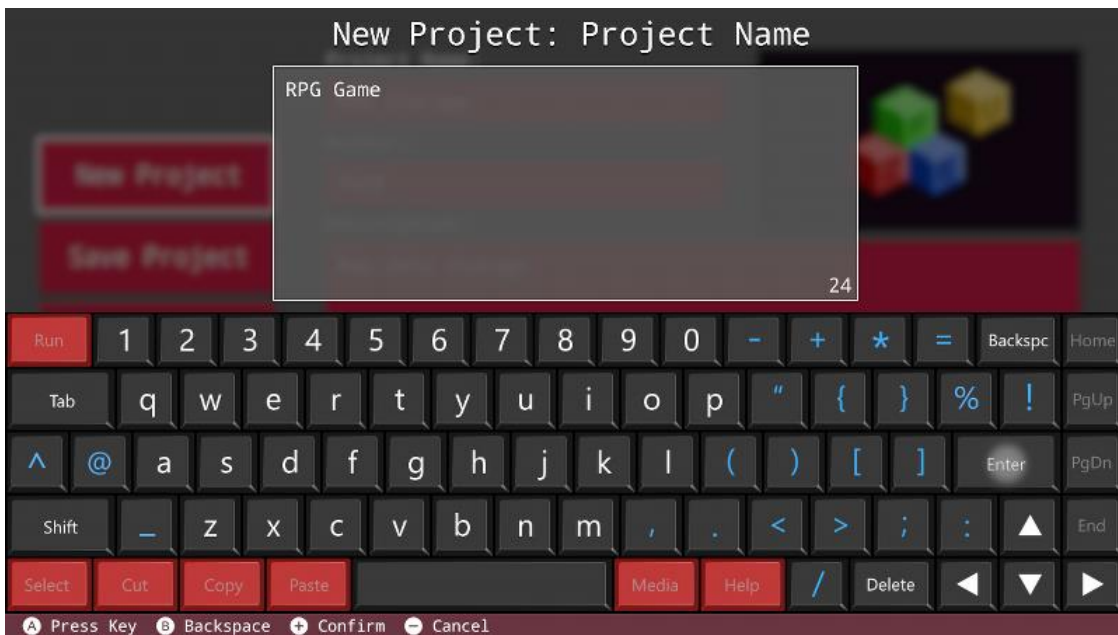


It's a great idea to overlap collision boxes to avoid any strange behaviour with the corners. Now that we've got our collision data set, we're ready to go with this map!

To save your map, simply press the minus button to return to the Main Menu. You'll see the **FUZE** save icon appear in the top right.

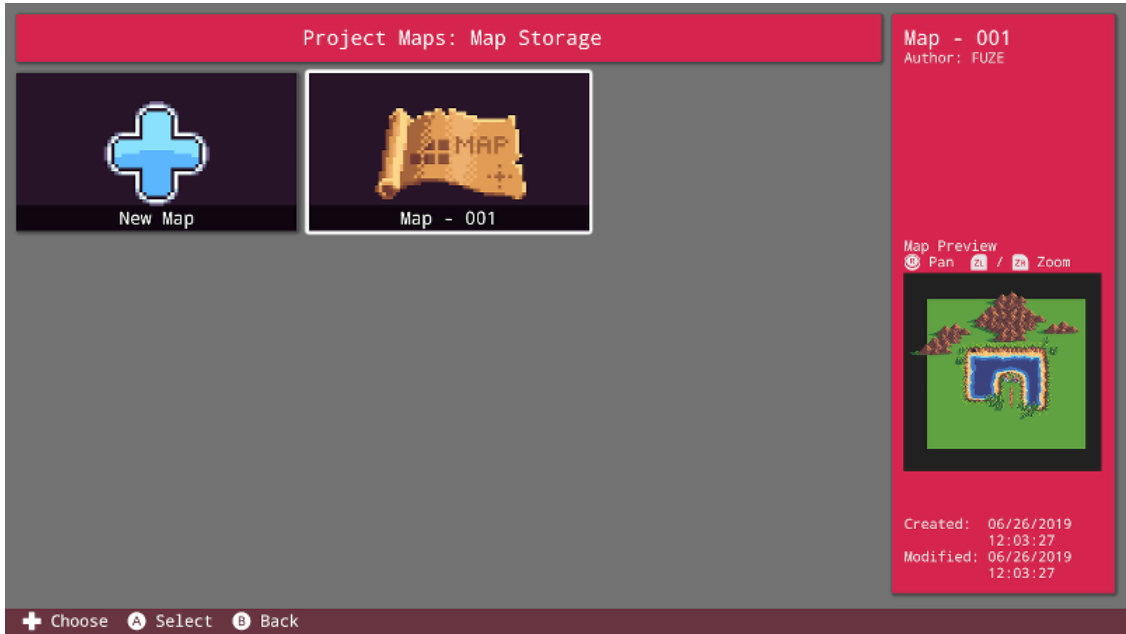
Copying a Map to a New Project

Let's get this map loaded into a project. Since we're using the current project file as map storage, we will want to copy this map into a new project. From the Main Menu, select the 'Project' icon and create a new project:

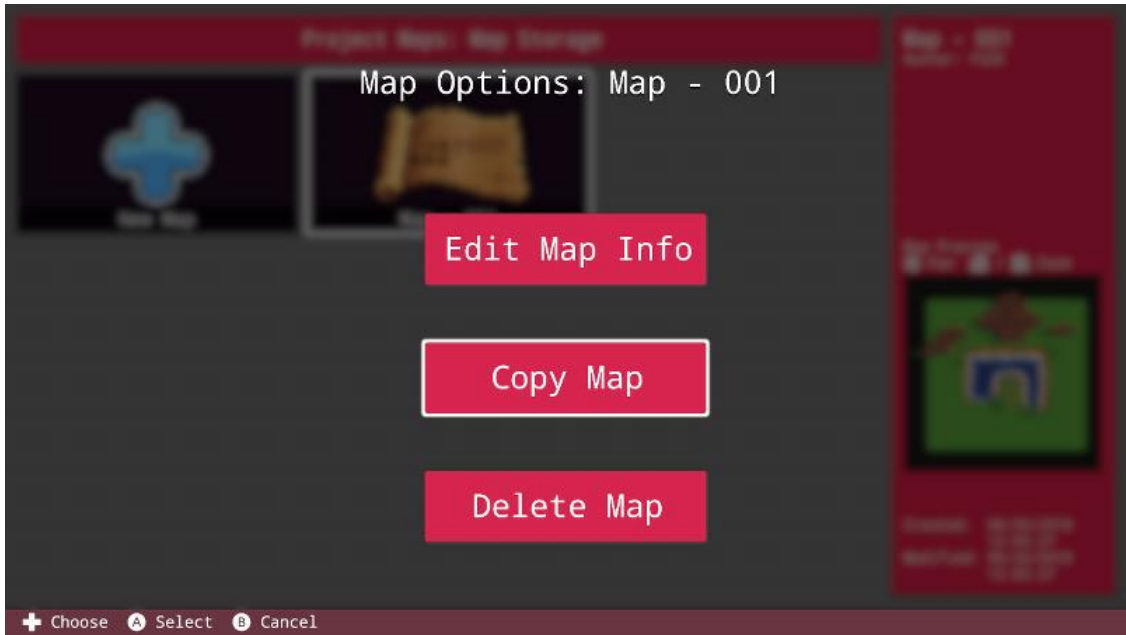


Once our project is created, return to the Main Menu and select the 'Tools' icon and then 'Map Editor'

Select the 'Map Storage' project to see our map:

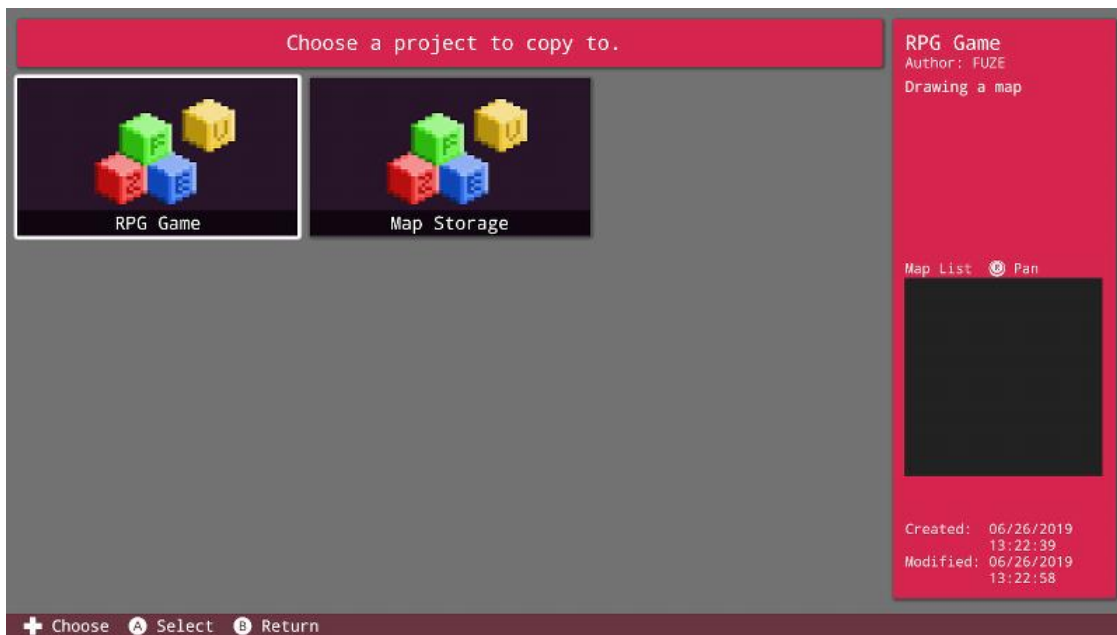


Press the X button to view the options for our map. You'll see a menu with three options. Select 'Copy Map':



You will see two options. Select 'Copy To Another Project'.

You will now see the list of projects we can copy our map to. We want to copy the map to our newly created project:



Select the project to copy to, then you'll be prompted to create a new name for the copied map. Once you're finished, return to the Main Menu.

Select the 'Programs' icon and open our new project with the copied map. This will take us to the code editor.

The first order of business is to load the map into memory. We use the `loadMap()` function to do this:

```

1. loadMap( "Map - 001" )
2.
3. setSpriteCamera( gWidth() / 2, gHeight() / 2, 2 )
4.
5. loop
6.   clear()
7.   centreSpriteCamera( 0, 0 )
8.   drawMap()
9.   update()
10. repeat

```

This program above will simply load our map and draw it at the centre of the screen with a zoom of 2.

Check out the map commands in the Reference Guide, they're linked just underneath. Using the variety of 2D graphic controls, with **FUZE** you can do just about anything with your map from here. Check out the map demos in the FUZE Projects too for more inspiration and how to use your collision data!

Map Commands

[collideMap\(\)](#), [detectMapCollision\(\)](#), [drawMap\(\)](#), [drawMapLayer\(\)](#), [loadMap\(\)](#), [unloadMap\(\)](#)

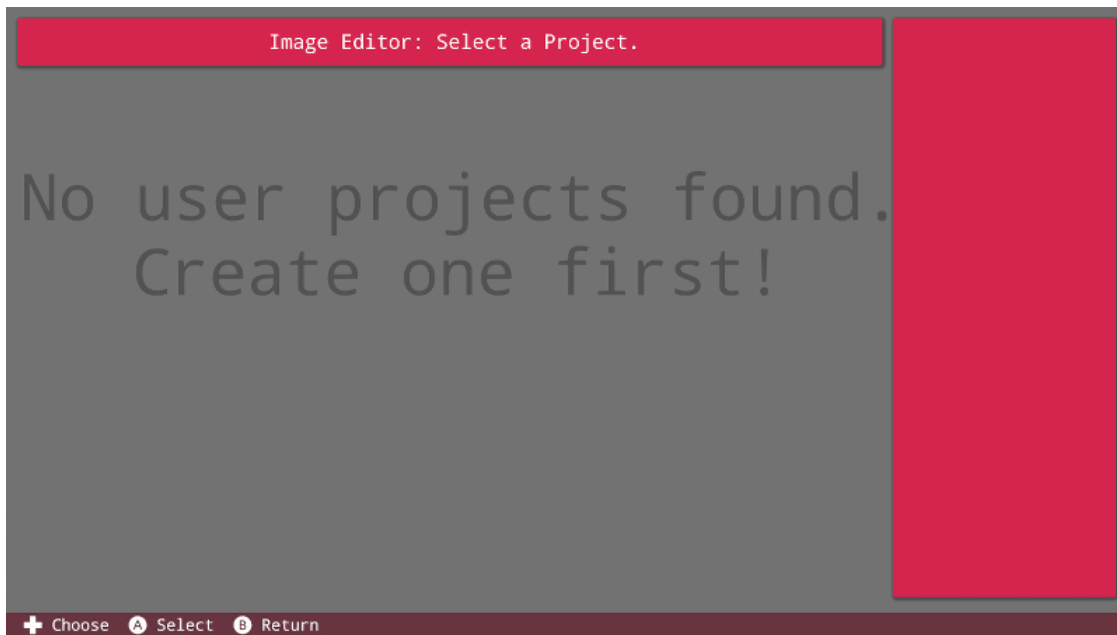
IMAGE EDITOR

Image Editor

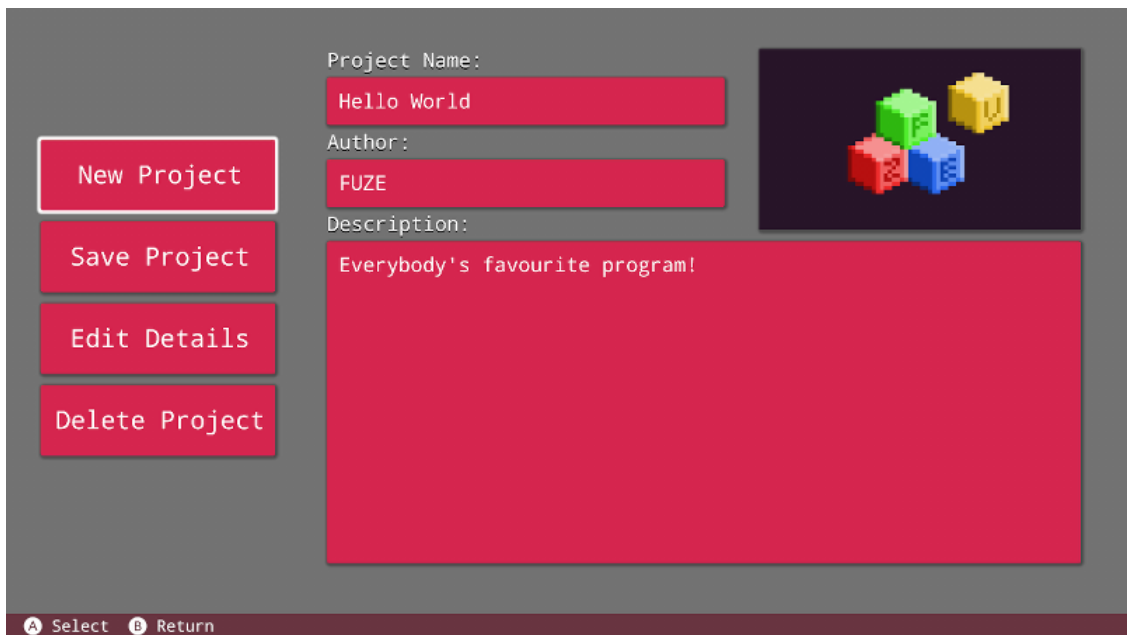
FUZE⁴ Nintendo Switch gives you access to a vast library of assets to use in your projects, but it also allows you can create your own sprites and levels using the Image and Map editors!

<>

From the Main Menu, select the 'Tools' icon followed by the 'Image Editor' icon. If it's your first time using FUZE, you'll see the following screen:

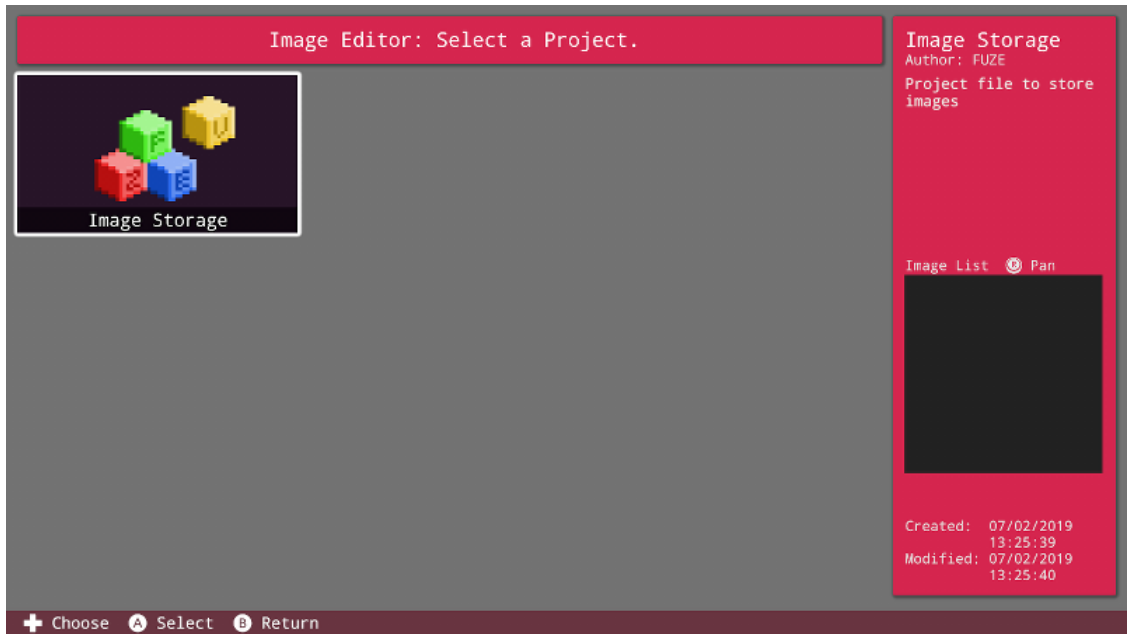


As you can see, we have nothing to do! User created images are stored in individual project files. Let's create a project which can store our images in. Return to the Main Menu and select the 'Project' icon:

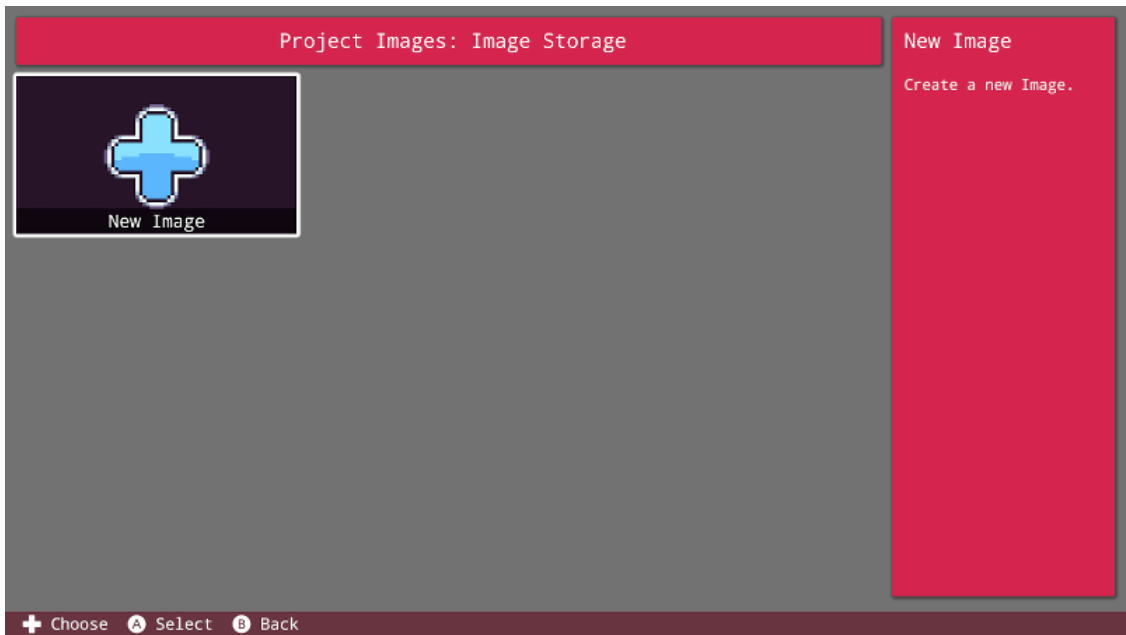


Here we can see the default loaded project which comes with FUZE. Everybody's favourite - "Hello World". We want to create a new project to store images into. Select the 'New Project' button and enter a name for the new project. We'll go with "Image Storage" for this example.

Once you've created the project you'll be taken to Code Editor. Return to the Main Menu using the minus button on the Joy-Con controller. From here, click the 'Tools' icon followed by 'Image Editor':



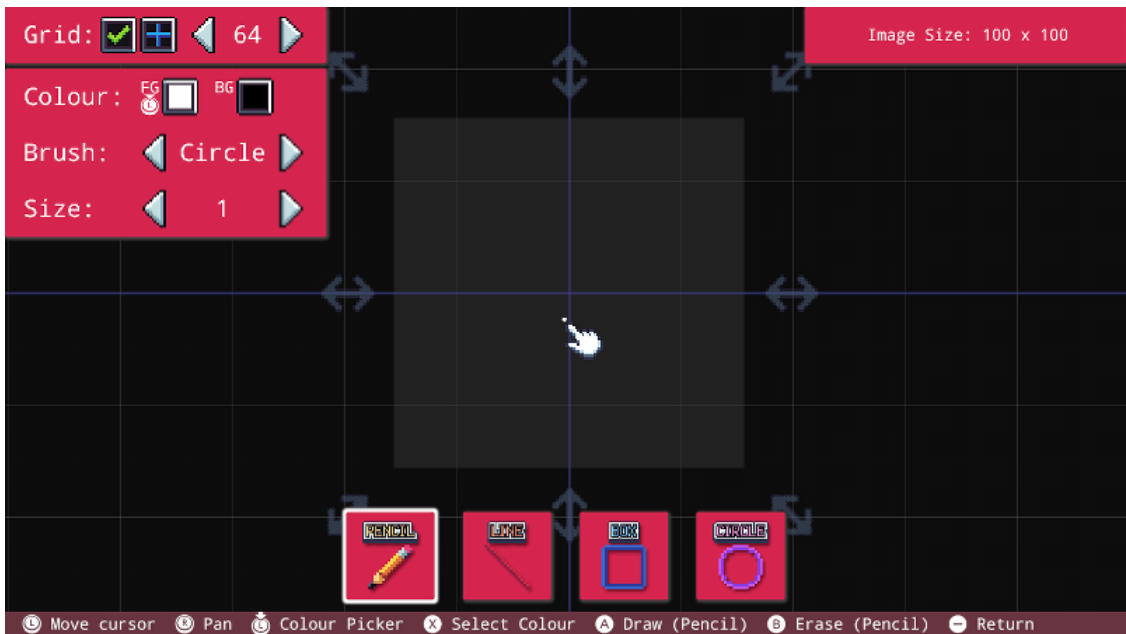
We can now see our newly created project. Click the project icon and you'll be taken to the next window:



This window is where we can see all the images stored within this project. Since we don't have any, let's click the 'New Image' button to get started!

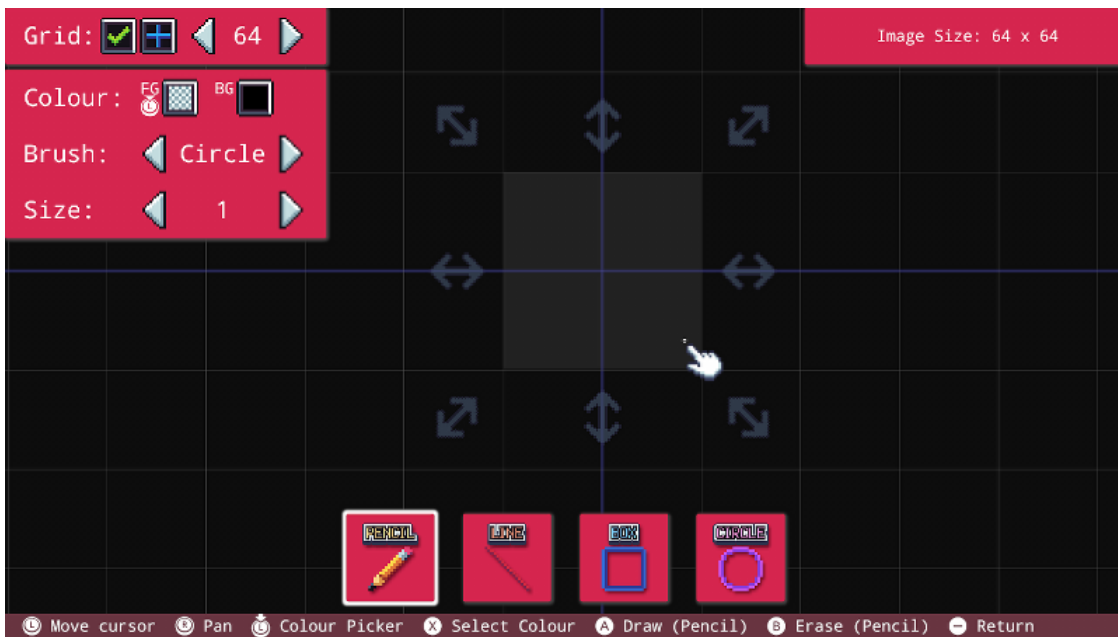
You will be prompted to enter a name for your image, then press the plus button to confirm. Once complete, you'll be taken to the Image Editor.

In the Image Editor Screen



Right away there are a few things to take note of. Take a look at the box in the top right of the screen. This is the size of your image in pixels. By default this will be 100 by 100 pixels.

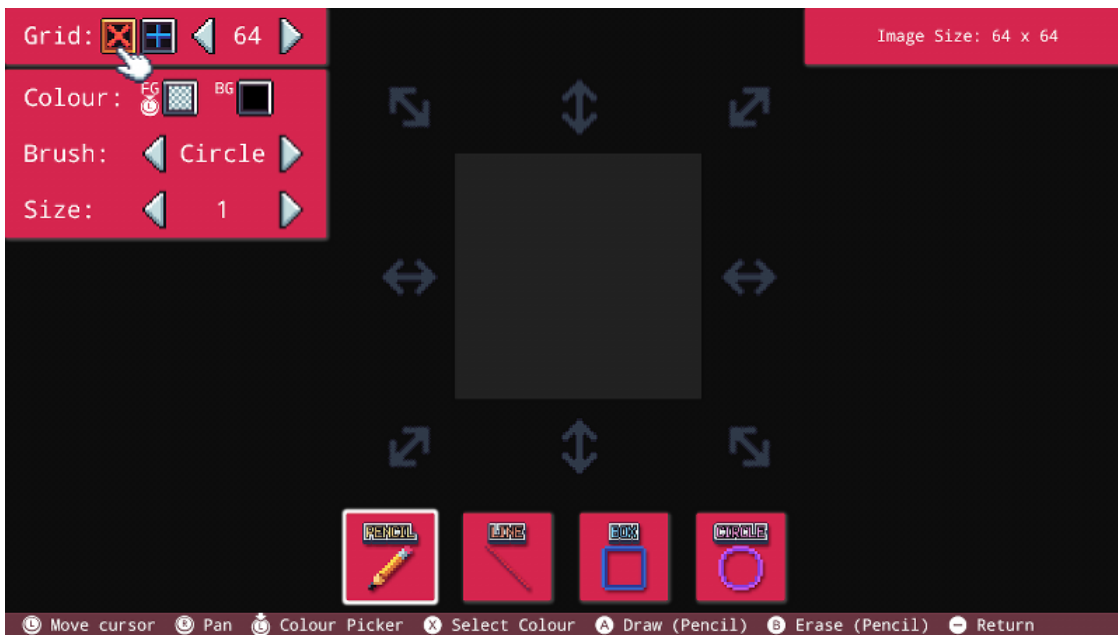
To adjust your image size, move the cursor over one of the grey arrows at the edges of the image. Select it with the A button, then move the control stick around to adjust the size.



Once you've got the size you want, press the A button again to confirm. This will also centre the image. As you can see, we've gone for an image size of 64 by 64 pixels and the image area is centred on screen.

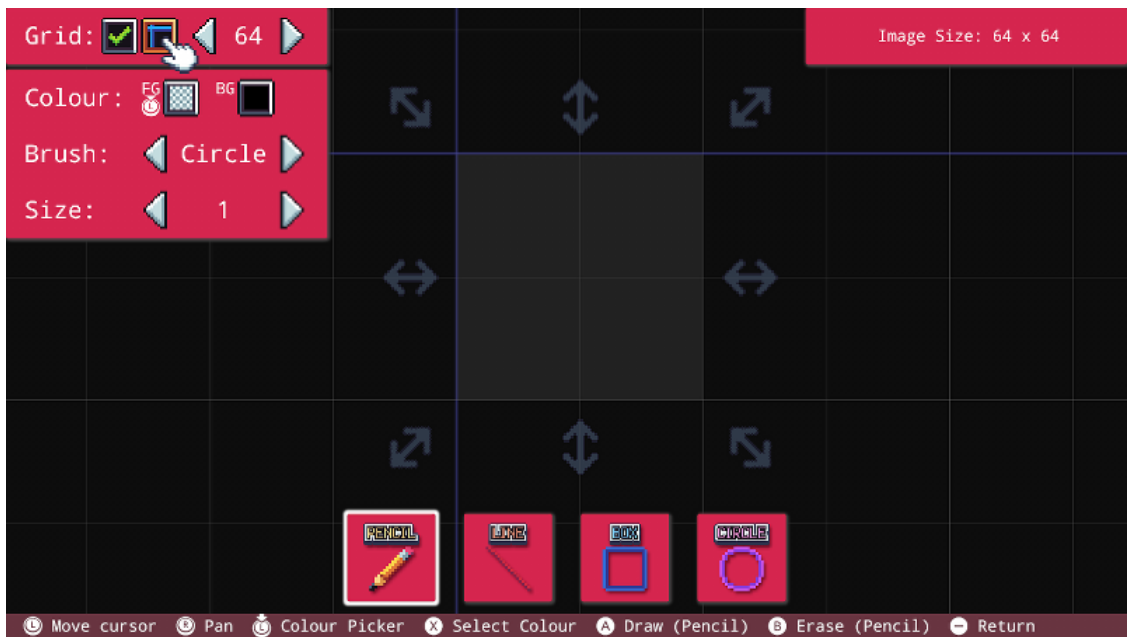
Grid Options

Next up, take a look at the box at the top left of the screen. Here we'll find our grid options. Move your cursor over the small box with a green tick and press the A button to toggle the grid on and off:

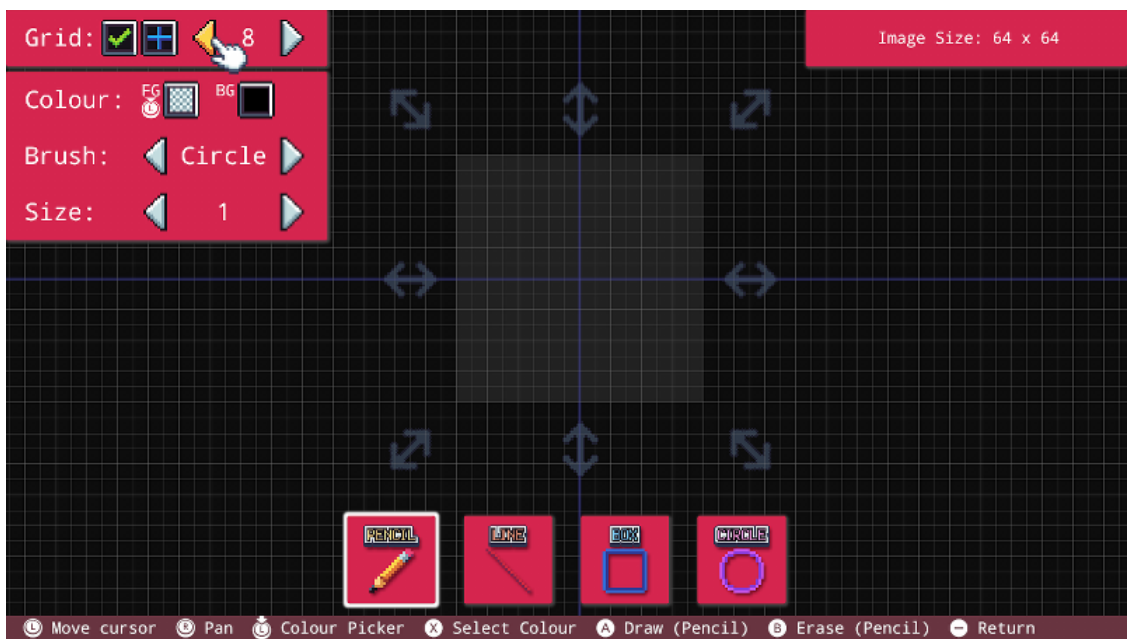


Our grid has vanished and the small box now displays a red cross rather than a green tick. Press the box again to bring the grid back.

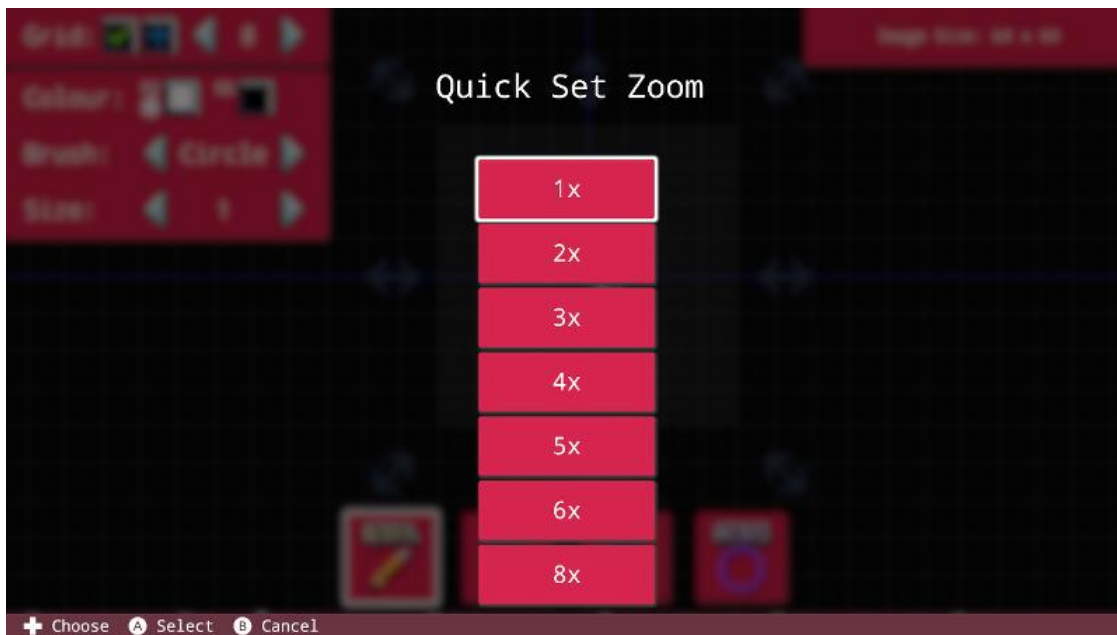
The next box along can be pressed to change the origin point of the image.



Finally, the arrow buttons either side of the number '64' on the grid options panel will change the density of the grid. Selecting a lower number will increase the amount of visible squares and help for finer detailing:



Use the ZR and ZL buttons to control the zoom. ZR will zoom the image in, whilst ZL zooms out. Pressing both the ZR and ZL buttons at the same time will bring up the quick zoom menu:



From here you can select a zoom level quickly.

That about covers it for setting up our image! Let's see what we can do with the brush.

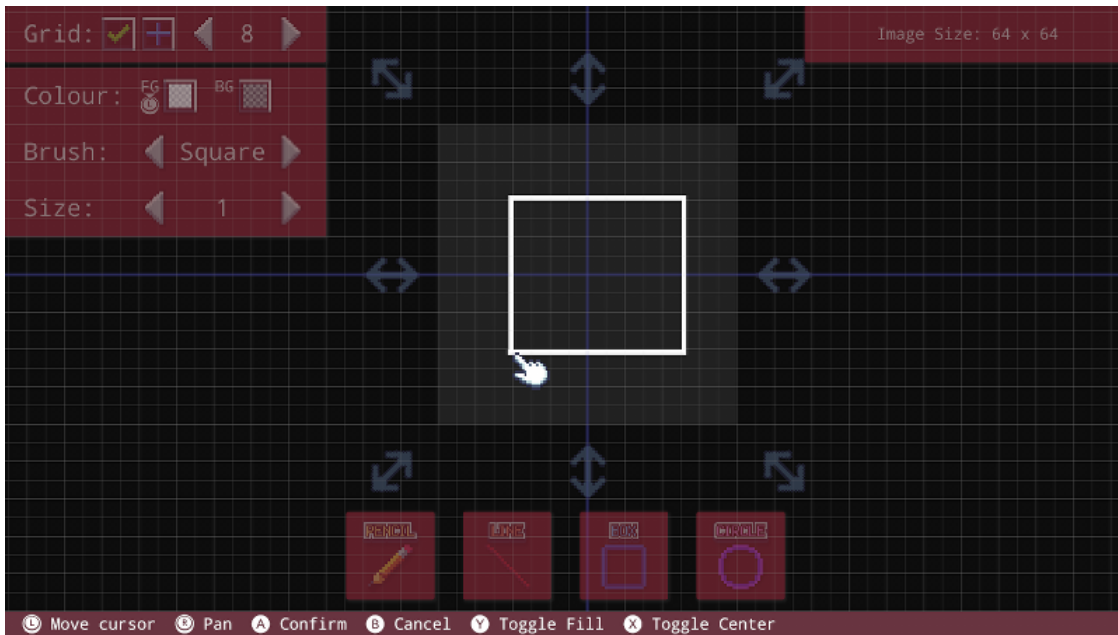
Brush Options

The box just beneath the Grid Options are our Brush Options. We can change the colour, the shape and the size. We'll get to colour shortly, for now let's focus on the options beneath.

The 'Brush' setting just below will change the brush shape between a square or a circle. When drawing small images pixel by pixel, the 'Square' brush option is ideal - whereas for colouring a large region, you might find the 'Circle' brush more suitable.

Lastly, the 'Size' setting just beneath will change the size of your brush. Who'd have thought?!

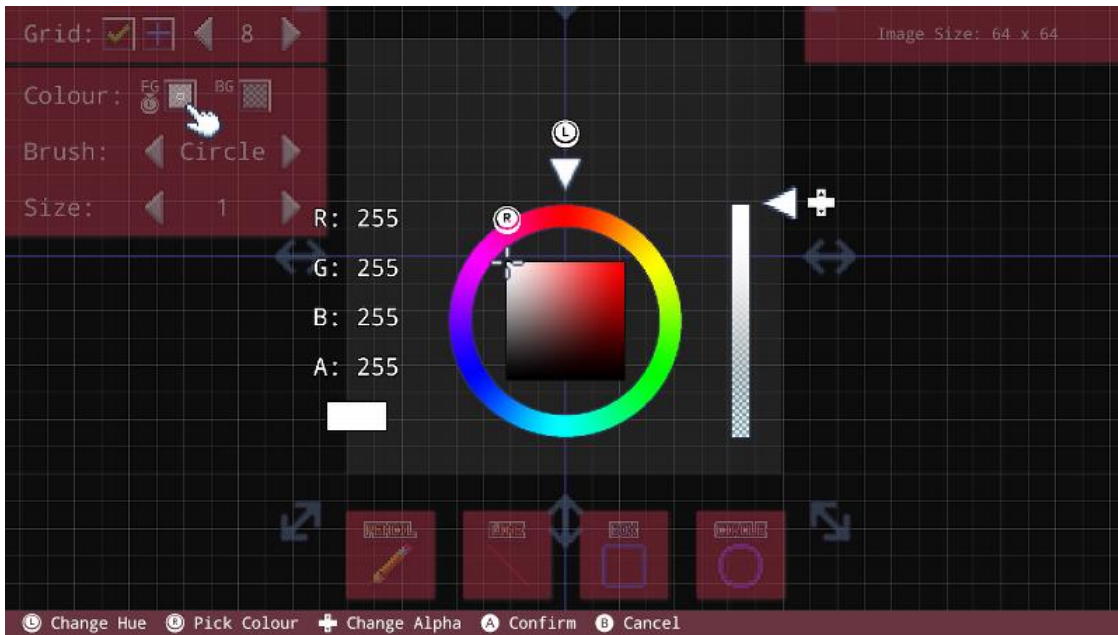
Pressing the L and R buttons will change the selection at the bottom of the screen between 'Pencil', 'Line', 'Box' and 'Circle' modes. This changes the way we draw. Selecting one of these tools will allow you to draw simple shapes easily. When drawing either a box or a circle, you will see a couple of new options displayed:



While drawing either a box or a circle, notice the command bar prompts at the bottom of the screen. Pressing the Y button will fill the object in with the selected background colour. Pressing the X button toggles the centre of the object.

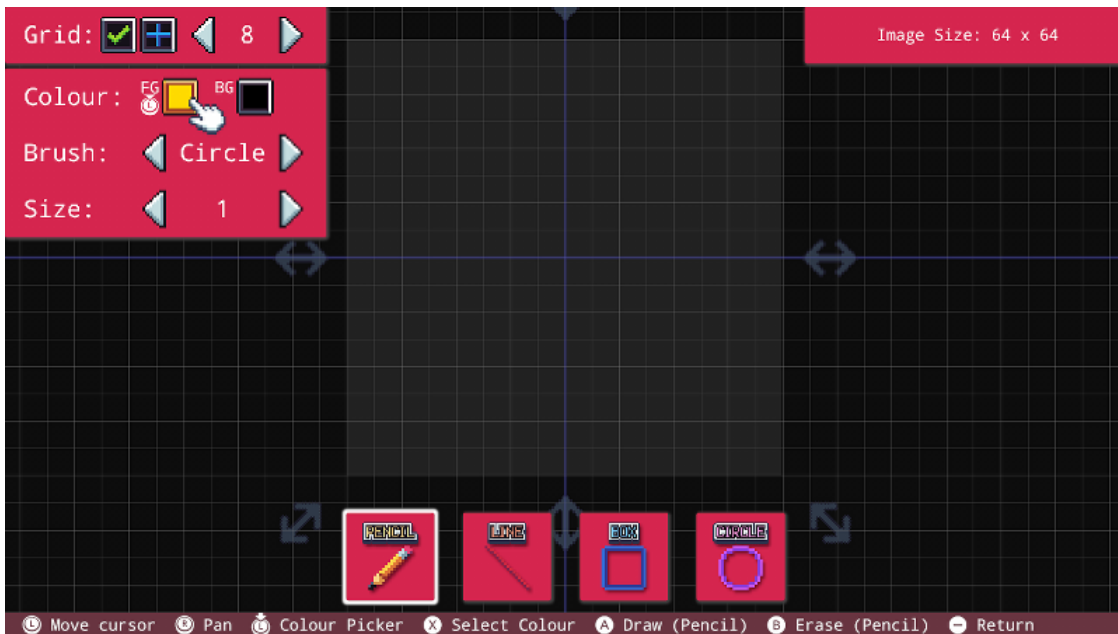
Colours

Let's take our eyes back up to the 'Colour' option at the top left. We have two boxes here, one for 'foreground' (FG) and one for 'background' (BG). Select the foreground box and select a colour. You should see the colour picker on screen:



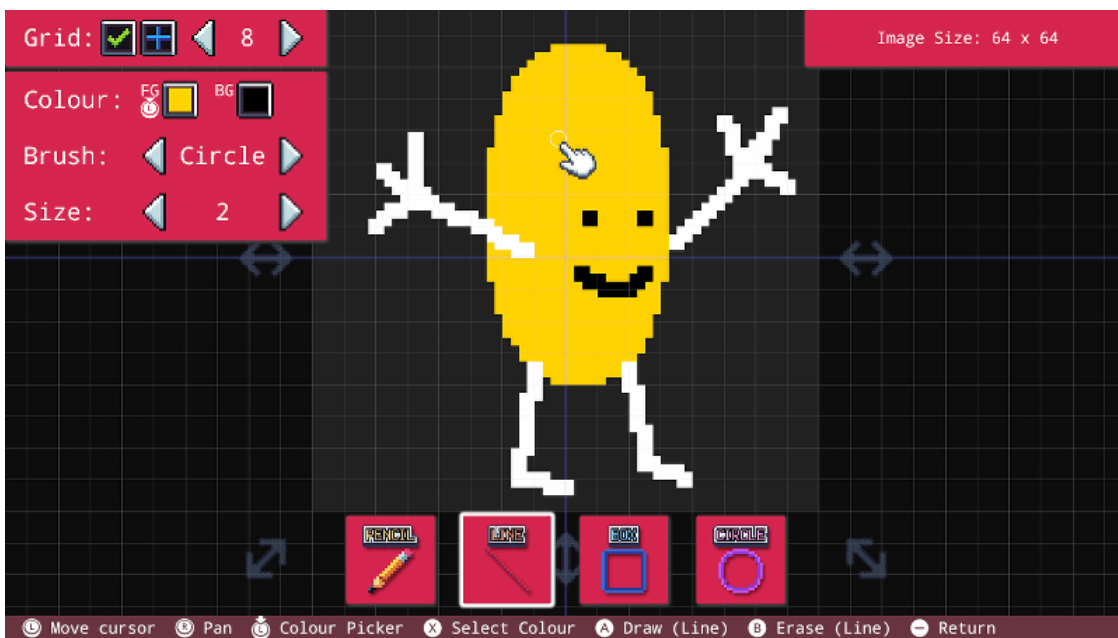
Here you can move the left and right control sticks to select a colour and hue. You can also use the up and down directional buttons to adjust the alpha (transparency).

Once you've found a colour you want to use, press the A button to confirm. This will change the colour of the box foreground box:



When we draw using the A button we are using the foreground colour.

Let's draw a simple image using a couple of colours:



There we go! Here is my brilliant (if I do say so myself) character.

There is a very useful tool in the image editor to select a colour we have already used. Currently the paint is set to a yellow colour. If we want to select the same white we've used in the image, simply place the cursor over the desired colour and press the X button:



Notice that the foreground colour in the box has changed to white? Using this tool we can easily grab colours from anywhere in our image.

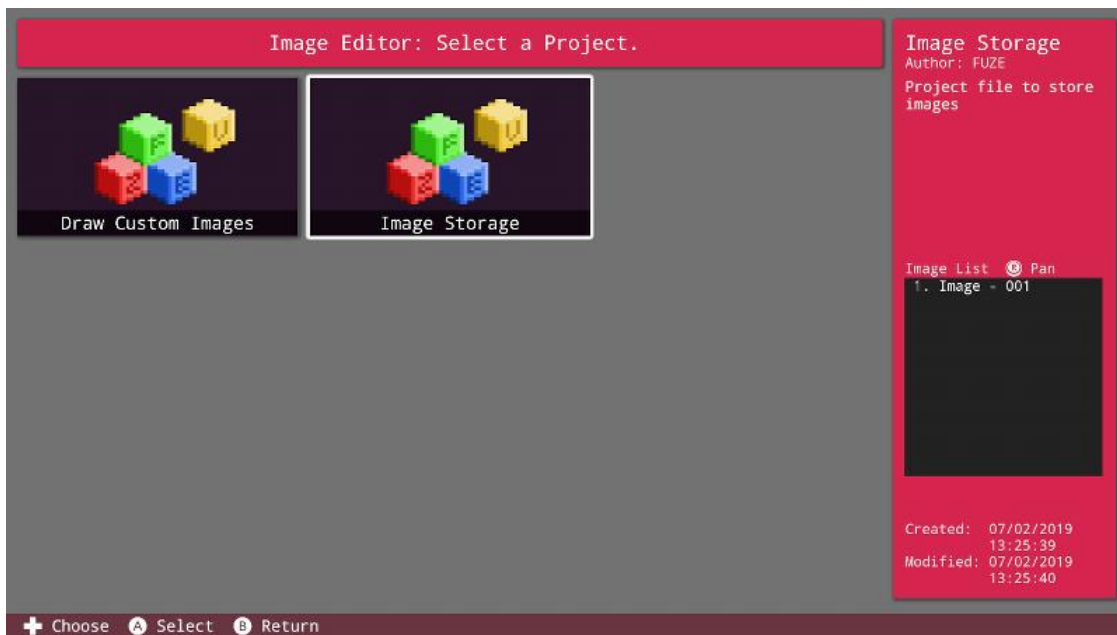
You can also press the left stick to bring up the colour picker at any time.

Loading and Drawing Your Image

Alright we've got a finished image here, let's load it into a program. Since we're using this project as just image storage, we'll want to copy this image to a new project.

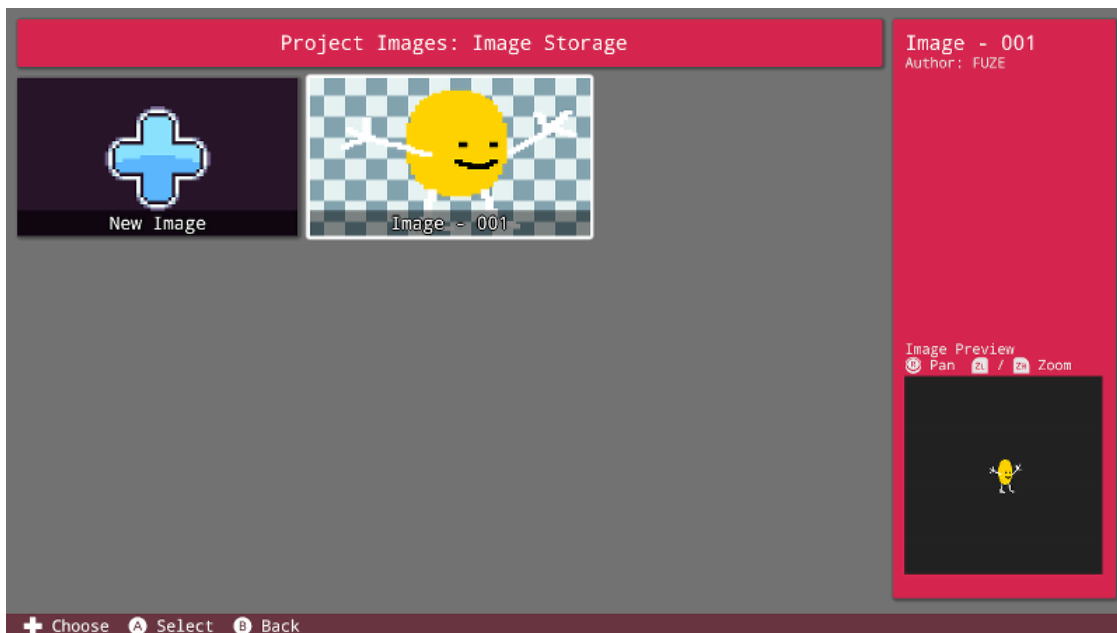
Begin a new project by returning to the Main Menu and selecting the 'Project' icon. Enter the title, author and description if desired. Once finished, you'll be in the Code Editor. Return to the Main Menu using the minus button.

Go to 'Tools', then 'Image Editor' like before. You should see something like this:

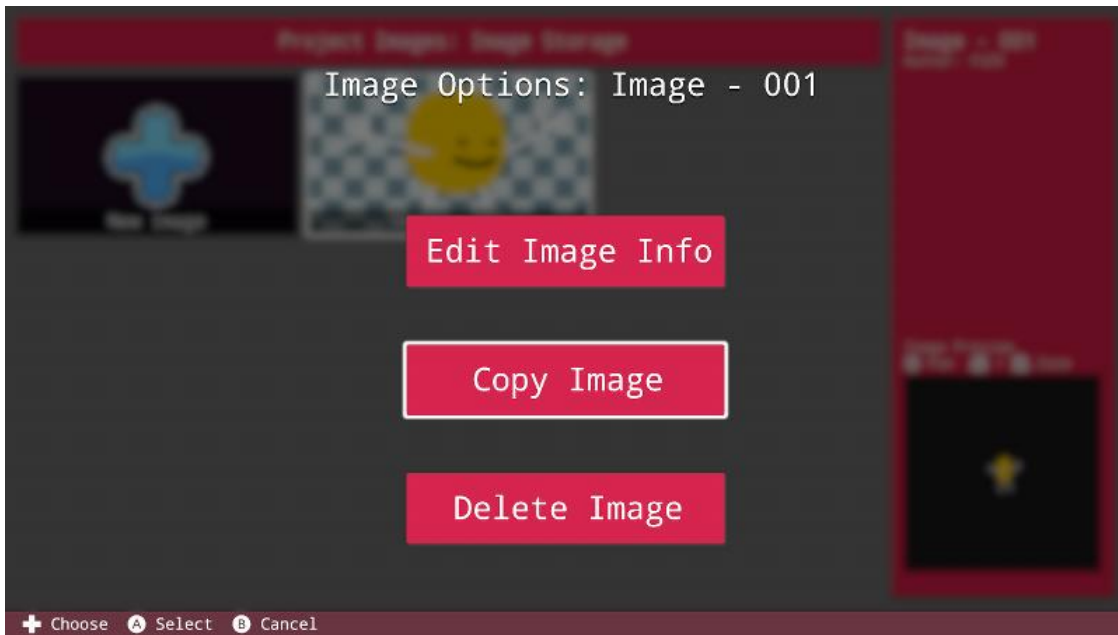


On the left we have the icon for the newly created project which we want to use the image in. On the right, the cursor is over our image storage project where the image is currently saved. You can see a list of the stored images in the project on the right of the screen in the project information panel.

Selecting the 'Image Storage' project icon will take us to the images stored in that project.

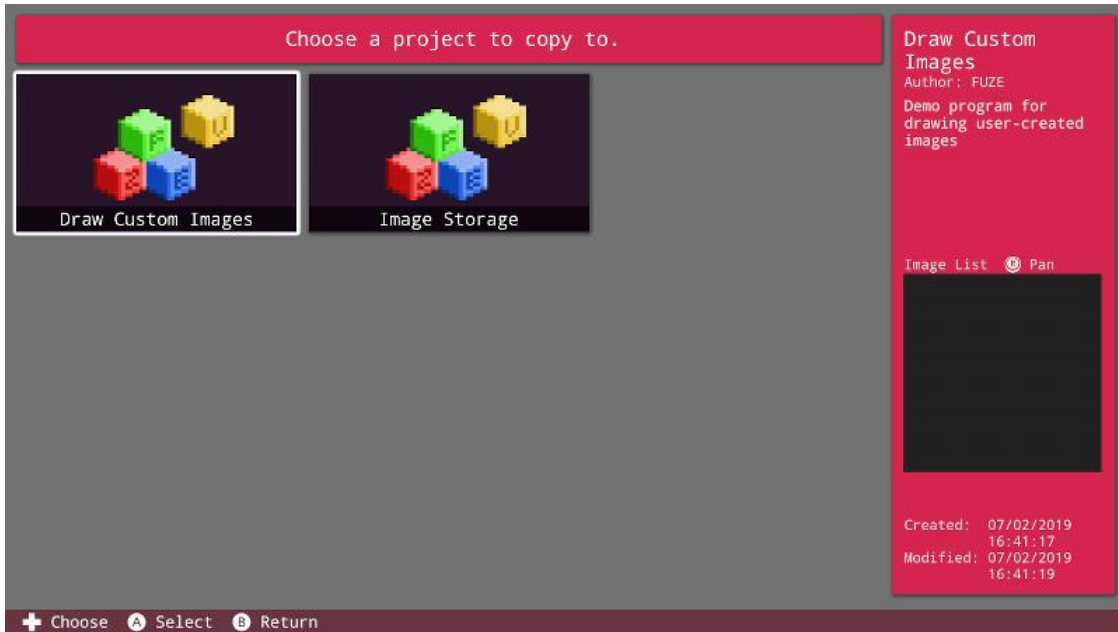


As you can see our image is now displayed here. Move the cursor on to the image and press the X button to view the image options:



Select the 'Copy Image' option and you'll see two further choices. You may either copy the image to the same project (duplicating it) or you can copy the image to another project for use.

Select 'Copy To Another Project' then select the newly created project which we want to copy the image to.



Once you've selected the project, you'll be prompted to input a new name for the image if desired.

The Code

Once your project file contains the image you want to use, it's time to load and draw the image using code!

Enter the following into the code editor:

```
1. img = loadImage( "Image - 001", false )
2.
3. loop
4.   clear()
5.   drawImage( img, gWidth() / 2, gHeight() / 2, 4 )
6.   update()
7. repeat
```

On line 1 we store our image into a **variable** using the `loadImage()` **function**. The filename on line 1 "Image - 001" will of course need to be the name of your image. The `false` in the `LoadImage()` brackets sets no filter to the image. This will keep the edges of our pixels nice and sharp. If you want to blur the edges of your image, set this to `true`.

Next, we have a very simple **loop** in which we simply use the `drawImage()` **function** to draw our image in the middle of the screen, with a scale multiplier of 4.

There we have it! We have created our very own image and drawn it to the screen. What more could you do with it? The limit is your imagination! Check out the image commands linked below to see more of the 2D graphics functions available to you.

You could very easily draw multiple images of different animation frames, store them in a big array and create your own animated sprite!

Image Commands

[drawImage\(\)](#), [drawImageEx\(\)](#), [freeImage\(\)](#), [imageSize\(\)](#), [loadImage\(\)](#)

GETTING STARTED

Keyboard Shortcuts

You can connect a USB keyboard to your Nintendo Switch Console to make typing quicker and more accurate.

While using a USB keyboard there are a number of very helpful shortcuts worth knowing!

Function Keys

At the top of the keyboard you will find a row of F keys (function keys).

Each one does something different in **FUZE⁴ Nintendo Switch**. Take a look at the list below:

F1 - Opens the Help menu. If you are in the code editor, this will open the in-editor Help menu. If you're on the Main Menu it will take you to the main Help section.

F2 - Takes you to the **Media Browser**.

F3 - **Saves** your project.

F5 - Pressing F5 will **run** your program. This works no matter where you are in FUZE, unless you are editing a program's description! When editing a program's description, **F5** will confirm and return you to the program.

F6 - Takes you to the **Image Editor**.

F7 - Takes you to the **Map Editor**.

F8 - Takes you to the **Settings** menu.

F9 - Takes you to the **Code Editor**.

F10 - Takes you back to the **Main Menu**.

F11 - Toggles text edit/help documentation view in the **Code Editor**.

Editing Shortcuts

There are also a number of handy shortcuts to use while writing code in the **Code Editor**. These are:

Shift - Select text (hold to select)

Alt - Select text (press to toggle on/off)

Ctrl + K - Show / Hide Keyboard

Ctrl + B - Bookmarks

Ctrl + V - Paste code

Ctrl + C - Copy selected code

Ctrl + X - Cut selected code

Keywords

K E Y W O R D S

KEYWORDS

and

Purpose

Join two conditions together

Description

The resulting condition is true if both of the conditions are true

Syntax

```
if condition1 and condition2 then ... endIf // ... is executed ONLY if both conditions are met
```

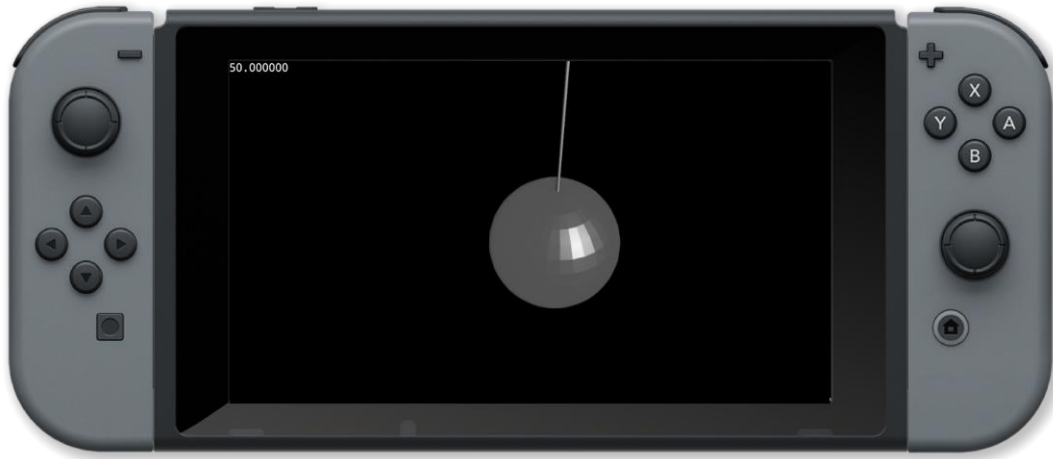
Arguments

condition1 first condition

condition2 second condition

Example

```
setCamera( { 0, 10, 10 }, { 0, 0, 0 } )
bright = 50
light = worldLight( { -5, -5, -5 }, white, bright )
lighton = true
ballmodel = loadModel( "Kat/Discoball" )
ball = placeObject( ballmodel, { 0, 0, 0 }, { 10, 10, 10 } )
loop
  c = controls( 0 )
  if c.x and !lighton then
    light = worldLight( { -5, -5, -5 }, white, bright )
    lighton = true
  endIf
  if c.a and lighton then
    removeLight( light )
    lighton = false
  endIf
  rotateObject( ball, { 0, 1, 0 }, 1.0 )
  drawObjects()
  printAt( 0, 0, "Press X to switch on the light" )
  printAt( 0, 1, "Press A to switch off the light" )
  update()
repeat
```

Associated Commands

and, else, endif, if, or, then

KEYWORDS

array

Purpose

Create an array of the integer type

Description

Defines a table of variables Can be used within a structure definition or standalone. Default array type is integer.

Syntax

```
struct name
  array field1
  ...
  typen fieldn
endStruct
```

Arguments

name name of the structure

field1 name of the first field

fieldn name of the last field

typen type of the last field

Example

```
// Define a property of a structure type as an array
struct person
  string name
  int age
  float height
  array interests[3]
endStruct

// Define an integer array of 10 elements
array data[10]
```

Associated Commands

array, int, float, endStruct, string, struct, vector

KEYWORDS

break

Purpose

Break out of a loop early

Description

Stop a loop from repeating before the exit condition is met

Syntax

```
while condition loop ... break ... repeat // Loop while condition is true or BREAK is executed
```

Arguments

condition boolean condition that stops the loop when false

Example

```
loop
  c = controls( 0 )
  printAt( 0,0, "Press A to exit program" )
  if c.a then
    break
  endIf
  update()
repeat
```

Associated Commands

for, repeat, step, to, while

KEYWORDS

else

Purpose

Conditionally execute a block of code when the condition is false

Description

Used to execute a block of code if the condition in the if statement is not met

Syntax

```
if condition then ... else ... endif // if condition is met execute first ... otherwise execute second ...
```

Arguments

condition condition to be tested. This can be a compound condition using AND and OR

Example

```
limit = 5
y = 0

for i = 0 to 11 loop
  if i < limit then
    printAt( 0, y, "Number: ", i, " is less than ", limit )
  else
    if i == limit then
      printAt( 0, y, "Number: ", i, " is equal to ", limit )
    else
      printAt( 0, y, "Number: ", i, " is more than ", limit )
    endif
  endif
endif
repeat

update()
sleep( 5 )
```

Associated Commands

and, else, endif, if, or, then

KEYWORDS

endif

Purpose

Marks the end of a conditional code block

Description

This ends the conditional if statement and returns to unconditional execution

Syntax

```
if condition then ... else ... endIf // if condition is met execute first ... otherwise execute second ...
```

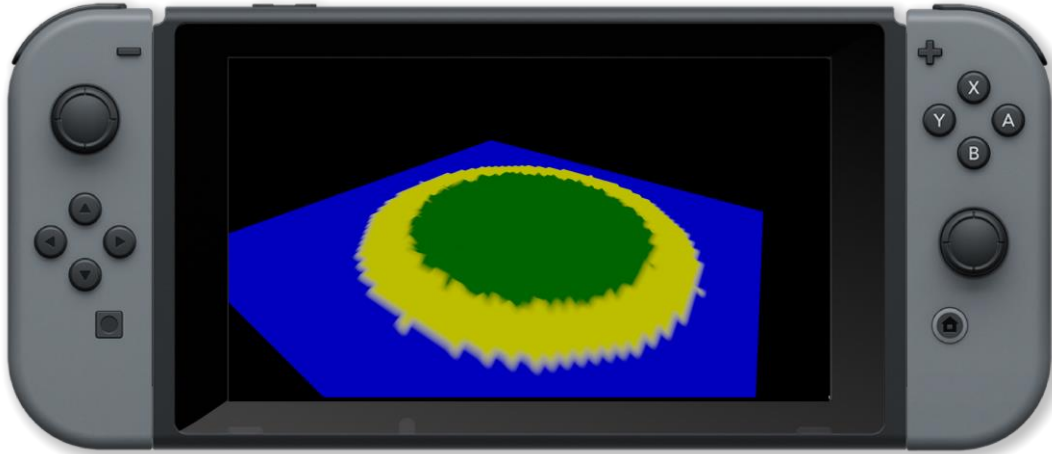
Arguments

condition condition to be tested. This can be a compound condition using AND and OR

Example

```
gsize = 64
landscape = createterrain( gsize, 1 )
height = 0
colour = white
for x = 0 to gsize loop
    for y = 0 to gsize loop
        d = distance( { x, y }, { gsize / 2, gsize / 2 } )
        if d > 24 then // sea level
            height = 0
            colour = blue
        else
            if d > 18 then // beach
                height = 1
                colour = yellow
            else // hills
                height = rnd( 2 ) + 1
                colour = green
            endIf
        endIf
        setTerrainPoint( landscape, x, y, height, colour )
    repeat
repeat
setCamera( { gsize / 2, 50, gsize / 2 }, { gsize / 2.0, 0, gsize / 2.00001 } )
setAmbientlight( { 0.5, 0.5, 0.5 } )
island = placeObject( landscape, { gsize / 2, 0, gsize / 2 }, { 1, 1, 1 } )
loop
    c = controls( 0 ) // rotate using joysticks
    rotateObject( island, { 1, 0, 0 }, c.ly )
```

```
rotateObject( island, { 0, 0, 1 }, c.lx )  
rotateObject( island, { 0, 1, 0 }, c.rx )  
drawObjects()  
update()  
repeat
```



Associated Commands

and, else, if, or, then

KEYWORDS

endStruct

Purpose

End a structured variable definition

Description

Marks the end of a structured variable definition

Syntax

```
struct name
  type1 field1
  ...
  typen fieldn
endStruct
```

Arguments

name name of the structure

field1 name of the first field

type1 type of the first field

fieldn name of the last field

typen type of the last field

Example

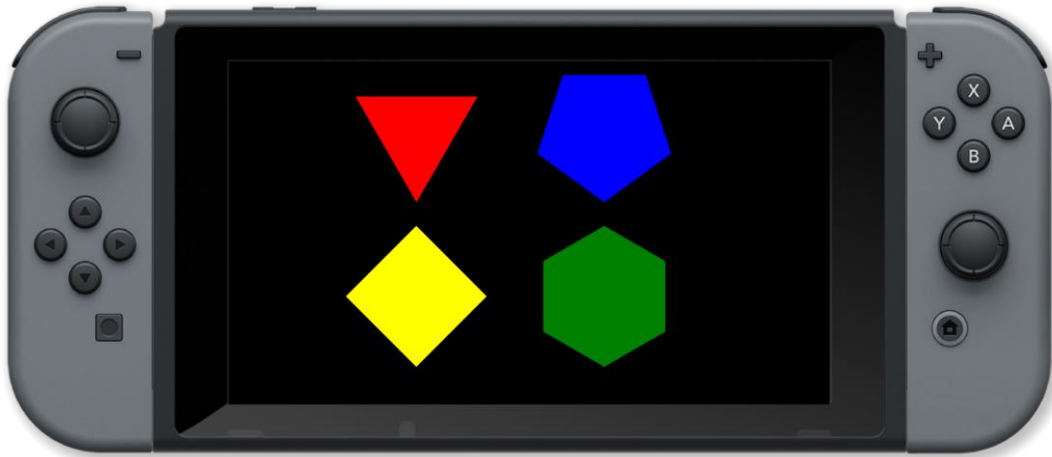
```
struct shape
  string name
  int sides
  int size
  vector pos
  int col
endStruct

shape shapes[3]
shapes[0] = [ .name = "triangle", .sides=3, .size=150, .pos = { 400, 150 }, .col = red ]
shapes[1] = [ .name = "square", .sides=4, .size=150, .pos = { 400, 500 }, .col = yellow ]
shapes[2] = [ .name = "pentagon", .sides=5, .size=150, .pos = { 800, 150 }, .col = blue ]
shapes[3] = [ .name = "hexagon", .sides=6, .size=150, .pos = { 800, 500 }, .col = green ]

loop
  clear()
  printat( 0, 0, "Press A to show labels" )
  c = controls( 0 )
```

```
for i = 0 to 4 loop
  drawShape( shapes[i], c.a )
repeat
  update()
repeat

function drawShape( s, label )
  circle( s.pos.x, s.pos.y, s.size, s.sides, s.col, 0 )
  if label then
    drawText( s.pos.x - s.size/2, s.pos.y, s.size / 5, black, s.name )
  endif
return void
```



Associated Commands

array, int, float, string, struct, vector

KEYWORDS

float

Purpose

Initialise a floating point variable

Description

Defines a variable as being of the float (floating point) type.

Syntax

```
struct name
  field1 float
  ...
  fieldn typen
endStruct
```

Arguments

name name of the structure

field1 name of the first field

fieldn name of the last field

typen type of the last field

Example

```
// Define a float variable within a structure definition
struct person
  string name
  int age
  float height
  array interests[3]
endStruct

// Initialise a float array of ten elements
float num[10]
```

Associated Commands

array, int, float, endStruct, string, struct, vector

KEYWORDS

for

Purpose

Repeat a section of code for a specific number of times

Description

The loop is executed until the value of the loop index variable goes from the start value to one step before the end value in increments of step

Syntax

```
for index = start to end loop ... repeat // Loop over values
```

```
for index = start to end step amount loop ... repeat // Loop over values with step
```

Arguments

index loop index variable

start start value of index

end end value of index (loop is not executed with this value)

amount amount to change index variable (default is one)

Example

```
// Draw 100 random boxes
clear()
for i = 1 to 100 loop
  // Pick random colour
  col = { random( 101 ) / 100, random( 101 ) / 100, random( 101 ) / 100, random( 101 ) / 100 }
  x = random( gWidth() )
  y = random( gHeight() )
  width = random( gWidth() / 4 )
  height = random( gHeight() / 4 )
  outline = random( 2 )
  box( x, y, width, height, col, outline )
  update()
repeat
// Wait 3 seconds
sleep( 3 )
```



Associated Commands

for, repeat, step, to, while

KEYWORDS

function

Purpose

Create a user defined function

Description

Allows the user to create their own functions. This allows code to be reused and makes it easier to read and maintain

Syntax

```
function name() ... return value // function with no arguments
```

```
function name(argument1, ... argumentn) ... return value // function with n arguments
```

Arguments

name name of the function

argument1 first parameter of the function

argumentn last parameter of the function

value return value of the function (void if no value is returned)

Example

```
for size = 1 to 200 step 1 loop
  clear()
  centreText( "Hello World", size )
  update()
repeat

// Centre a text string on the screen
function centreText( message, size )
  textSize( size )
  tw = textWidth( message )
  drawText( ( gwidth() - tw ) / 2, ( gheight() - size ) / 2, size, white, message )
return void
```



Associated Commands

`function`, `return`, `void`

KEYWORDS

if

Purpose

Conditionally execute a block of code when condition is true

Description

Execute a block of code only if the specified condition is true (1)

Syntax

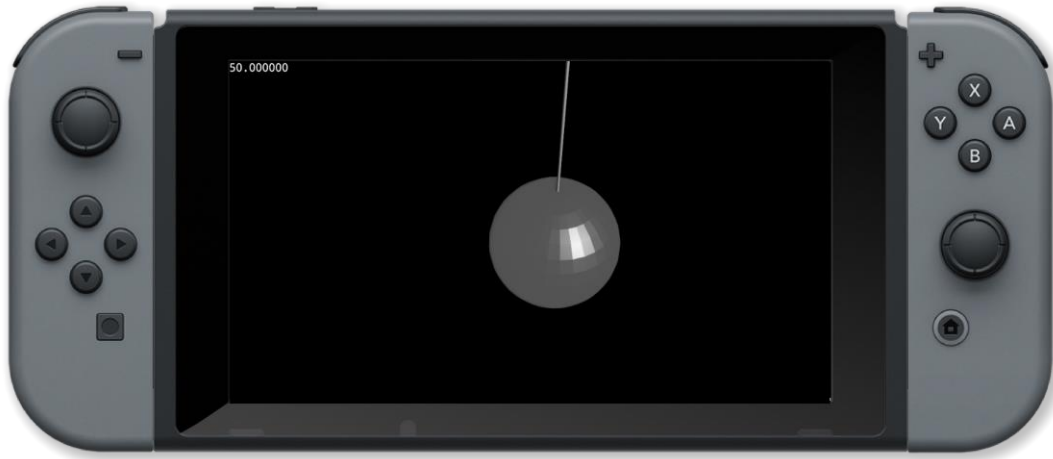
```
if condition then ... endIf // ... is executed ONLY if condition is met  
if condition then ... else ... endIf // if condition is met execute first ... otherwise execute second ...
```

Arguments

condition condition to be tested. This can be a compound condition using AND and OR

Example

```
setcamera( {0, 10, 10 }, { 0, 0, 0 } )  
bright = 50  
light = worldLight( { -5, -5, -5 }, white, bright )  
lighton = true  
ballmodel = loadModel( "Kat/Discoball" )  
ball = placeObject( ballmodel, { 0, 0, 0 }, { 10, 10, 10 } )  
loop  
  c = controls( 0 )  
  if c.x and not lighton then  
    light = worldLight( { -5, -5, -5 }, white, bright )  
    lighton = true  
  endIf  
  if c.a and lighton then  
    removeLight( light )  
    lighton = false  
  endIf  
  rotateObject(ball, { 0, 1, 0 }, 1.0)  
  drawObjects()  
  printAt( 0, 0, "Press X to switch on the light" )  
  printAt( 0, 1, "Press A to switch off the light" )  
  update()  
repeat
```



Associated Commands

and, else, endif, if, or, then

KEYWORDS

int

Purpose

Create an integer variable

Description

Create a variable of the integer type. Can also be used as an array definition and within structure definitions.

Syntax

```
struct name
  int field1
  ...
  typen fieldn
endStruct
```

Arguments

name name of the structure

field1 name of the first field

fieldn name of the last field

typen type of the last field

Example

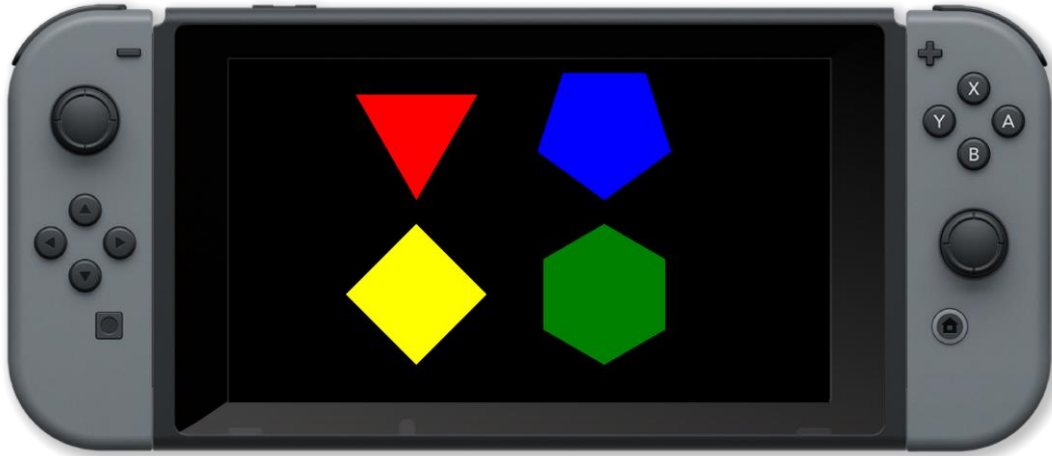
```
struct shape
  string name
  int sides
  int size
  vector pos
  int col
endStruct

shape shapes[3]
shapes[0] = [ .name = "triangle", .sides=3, .size=150, .pos = { 400, 150 }, .col = red ]
shapes[1] = [ .name = "square", .sides=4, .size=150, .pos = { 400, 500 }, .col = yellow ]
shapes[2] = [ .name = "pentagon", .sides=5, .size=150, .pos = { 800, 150 }, .col = blue ]
shapes[3] = [ .name = "hexagon", .sides=6, .size=150, .pos = { 800, 500 }, .col = green ]

loop
  clear()
  printAt( 0,0,"Press A to show labels" )
  c = controls( 0 )
  for i = 0 to 4 loop
```



```
drawShape( shapes[i], c.a )
repeat
  update()
repeat
function drawShape( s, label )
  circle( s.pos.x, s.pos.y, s.size, s.sides, s.col, 0 )
  if label then
    drawText( s.pos.x - s.size/2, s.pos.y, s.size / 5, black, s.name )
  endIf
return void
```



Associated Commands

array, int, float, endStruct, string, struct, vector

KEYWORDS

loop

Purpose

Repeat a section of code

Description

Repeat a section of code a specified number of times or until a condition is met (or forever)

Syntax

```
loop ... repeat // Loop forever  
while condition loop ... repeat // Loop while condition is true  
for index = start to end loop ... repeat // Loop over values  
for index = start to end step amount loop ... repeat // Loop over values with step
```

Arguments

condition boolean condition that stops the loop when false

index loop index variable

start start value of index

end end value of index (loop is not executed with this value)

amount amount to change index variable

Example

```
// Draw 100 random boxes  
clear()  
for i = 1 to 100 loop  
  // Pick random colour  
  col = { random( 101 ) / 100, random( 101 ) / 100, random( 101 ) / 100, random( 101 ) / 100 }  
  x = random( gWidth() )  
  y = random( gHeight() )  
  width = random( gWidth() / 4 )  
  height = random( gHeight() / 4 )  
  outline = random( 2 )  
  box( x, y, width, height, col, outline )  
  update()  
repeat  
// Wait 3 seconds  
sleep( 3 )
```



Associated Commands

for, repeat, step, to, while

KEYWORDS

not

Purpose

Negate a condition

Description

Execute a block of code only if the specified condition is false (0)

Syntax

```
if not condition then ... endif // ... is executed ONLY if condition is not met
if not condition then ... else ... endif // if condition is not met execute first ... otherwise execute second ...
```

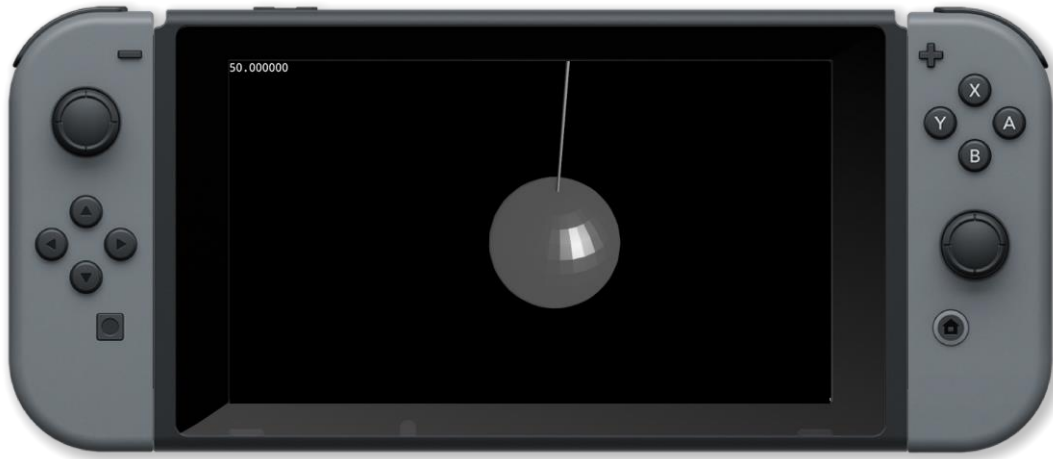
Arguments

condition condition to be tested. This can be a compound condition using AND and OR

Example

```
setCamera( {0, 10, 10 }, { 0, 0, 0 } )
bright = 50
light = worldLight( { -5, -5, -5 }, white, bright )
lighton = true
ballmodel = loadModel( "Kat/Discoball" )
ball = placeObject( ballmodel, { 0, 0, 0 }, { 10, 10, 10 } )

loop
  c = controls( 0 )
  if c.x and not lighton then
    light = worldLight( { -5, -5, -5 }, white, bright )
    lighton = true
  endif
  if c.a and lighton then
    removeLight( light )
    lighton = false
  endif
  rotateObject( ball, { 0, 1, 0 }, 1.0 )
  drawObjects()
  printAt( 0, 0, "Press X to switch on the light" )
  printAt( 0, 1, "Press A to switch off the light" )
  update()
repeat
```



Associated Commands

and, else, endif, if, or, then

KEYWORDS

or

Purpose

Specifies alternate condition

Description

The resulting condition is true if one of the conditions are true

Syntax

```
if condition1 or condition2 then ... endIf // ... is executed either condition is true
```

Arguments

condition1 first condition

condition2 second condition

Example

```
image = loadImage( "Untied Games/Enemy small top C", false )
ship = createSprite()
setSpriteImage( ship, image )
lastpos = { gWidth() / 2, gHeight() / 2 }
setSpriteLocation( ship, lastpos )
setSpriteScale( ship, { 20, 20 } )
rv = -0.5
gv = 0.5
bv = 0

loop
  clear()
  sc = getSpriteColour( ship )
  if sc.r > 1 or sc.r < 0 then
    rv = -rv
  endIf
  if sc.b > 1 or sc.b < 0 then
    gv = -gv
  endIf
  setSpriteColourSpeed( ship, { rv, gv, bv, 0 } )
  updateSprites()
  drawSprites()
  update()
repeat
```



Associated Commands

and, else, endif, if, or, then

KEYWORDS

repeat

Purpose

End of a loop or repeated section of code

Description

Marks the end of a section of code to be repeated. Control is passed back to the previous LOOP keyword. The loop is repeated a specified number of times or until a condition is met (or forever)

Syntax

```
loop ... repeat // Loop forever  
  
while condition loop ... repeat // Loop while condition is true  
  
for index = start to end loop ... repeat // Loop over values  
  
for index = start to end step amount loop ... repeat // Loop over values with step
```

Arguments

condition boolean condition that stops the loop when false

index loop index variable

start start value of index

end end value of index (loop is not executed with this value)

amount amount to change index variable (default is one)

Example

```
// Draw 100 random boxes  
clear()  
for i = 1 to 100 loop  
  // Pick random colour  
  col = { random( 101 ) / 100, random( 101 ) / 100, random( 101 ) / 100, random( 101 ) / 100 }  
  x = random( gWidth() )  
  y = random( gHeight() )  
  width = random( gWidth() / 4 )  
  height = random( gHeight() / 4 )  
  outline = random( 2 )  
  box( x, y, width, height, col, outline )  
  update()  
repeat  
// Wait 3 seconds  
sleep( 3 )
```




Associated Commands

for, repeat, step, to, while

KEYWORDS

return

Purpose

Return a value from a user defined function

Description

Returns a value from a function and resumes execution from the point where the function was called

Syntax

```
function name() ... return value // function with no arguments
```

```
function name(argument1, ... argumentn) ... return value // function with n arguments
```

Arguments

name name of the function

argument1 first parameter of the function

argumentn last parameter of the function

value return value of the function (void if no value is returned)

Example

```
y = 0
for i = 1 to 11 loop
    square = calculateSquare( i )
    printAt( 0, y, square )
    y += 1
repeat

update()
sleep( 3 )

function calculateSquare( num )
    num *= num
return num
```



Associated Commands

`function`, `return`, `void`

KEYWORDS

step

Purpose

Specifies the increment value of the loop index variable

Description

The loop is executed until the value of the loop index variable goes from the start value to one step before the end value in increments of step

Syntax

```
for index = start to end step amount loop ... repeat // Loop over values with step
```

Arguments

index loop index variable

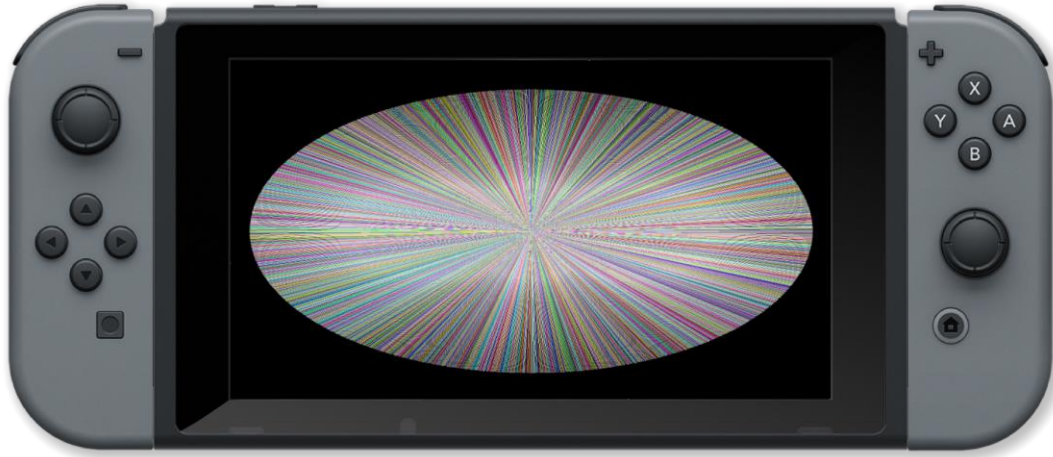
start start value of index

end end value of index (loop is not executed with this value)

amount amount to change index variable (default is one)

Example

```
clear()
radians( true )
centre = { gwidth() / 2, gHeight() / 2 }
for angle = 0 to 2 * pi step 0.005 loop
  col = { random( 101 ) / 100, random( 101 ) / 100, random( 101 ) / 100, 1.0 }
  result = sinCos( angle )
  point = { 600 * result.y + centre.x, 300 * result.x + centre.y }
  line( centre, point, col )
repeat
update()
sleep( 3 )
```



Associated Commands

for, repeat, step, to, while

KEYWORDS

string

Purpose

Create a string variable

Description

Initialises a variable of the string type. Can be used to initialise an array or within a structure definition

Syntax

```
struct name
  string field1
  ...
  typen fieldn
endStruct
```

Arguments

name name of the structure

field1 name of the first field

fieldn name of the last field

typen type of the last field

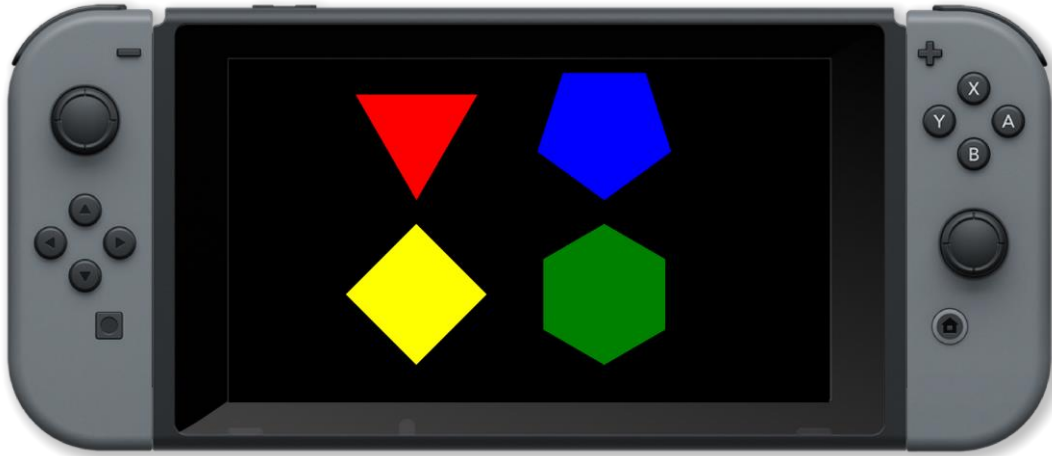
Example

```
struct shape
  string name
  int sides
  int size
  vector pos
  int col
endStruct

shape shapes[3]
shapes[0] = [ .name = "triangle", .sides=3, .size=150, .pos = { 400, 150 }, .col = red ]
shapes[1] = [ .name = "square", .sides=4, .size=150, .pos = { 400, 500 }, .col = yellow ]
shapes[2] = [ .name = "pentagon", .sides=5, .size=150, .pos = { 800, 150 }, .col = blue ]
shapes[3] = [ .name = "hexagon", .sides=6, .size=150, .pos = { 800, 500 }, .col = green ]

loop
  clear()
  printAt( 0, 0, "Press A to show labels" )
  c = controls( 0 )
  for i = 0 to 4 loop
```

```
drawShape(shapes[i], c.a )
repeat
update()
repeat
function drawShape( s, label )
circle( s.pos.x, s.pos.y, s.size, s.sides, s.col, 0 )
if label then
drawText( s.pos.x - s.size/2, s.pos.y, s.size / 5, black, s.name )
endIf
return void
```



Associated Commands

array, int, float, endStruct, string, struct, vector

KEYWORDS

struct

Purpose

Create a structured variable type

Description

Allows the user to create their own complex variable types. This allows the grouping of related information into a single variable

Syntax

```
struct name
  type1 field1
  ...
  typen fieldn
endStruct
```

Arguments

name name of the structure

field1 name of the first field

type1 type of the first field

fieldn name of the last field

typen type of the last field

Example

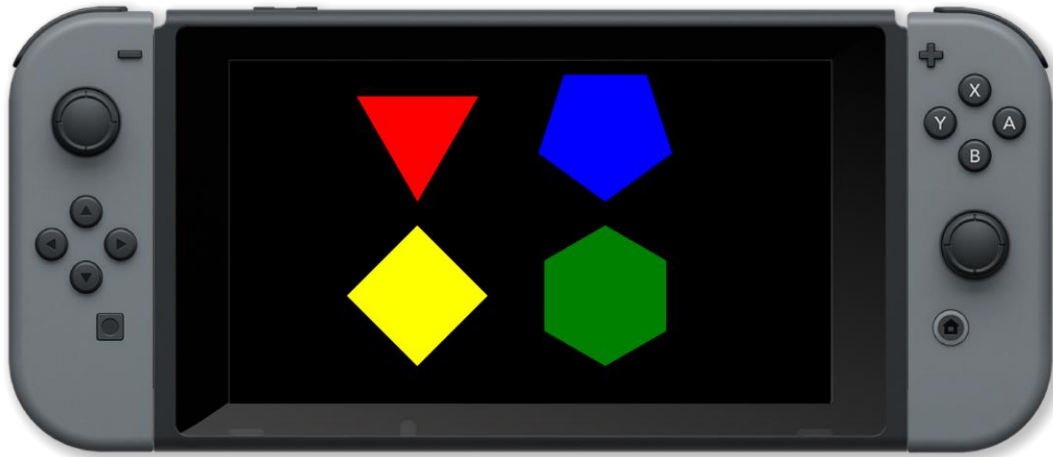
```
struct shape
  string name
  int sides
  int size
  vector pos
  int col
endStruct

shape shapes[3]
shapes[0] = [ .name = "triangle", .sides=3, .size=150, .pos = { 400, 150 }, .col = red ]
shapes[1] = [ .name = "square", .sides=4, .size=150, .pos = { 400, 500 }, .col = yellow ]
shapes[2] = [ .name = "pentagon", .sides=5, .size=150, .pos = { 800, 150 }, .col = blue ]
shapes[3] = [ .name = "hexagon", .sides=6, .size=150, .pos = { 800, 500 }, .col = green ]

loop
  clear()
  printAt( 0, 0, "Press A to show labels" )
```



```
c = controls( 0 )
for i = 0 to 4 loop
    drawShape( shapes[i], c.a )
repeat
    update()
endRepeat
function drawshape(s, label)
    circle( s.pos.x, s.pos.y, s.size, s.sides, s.col, 0 )
    if label then
        drawText( s.pos.x - s.size/2, s.pos.y, s.size / 5, black, s.name )
    endIf
endFunction
return void
```



Associated Commands

array, int, float, endStruct, string, struct, vector

KEYWORDS

then

Purpose

Marks the end of a condition and the start of a block of conditional code

Description

Code after this is executed up to an ELSE or endif statement if the condition is met

Syntax

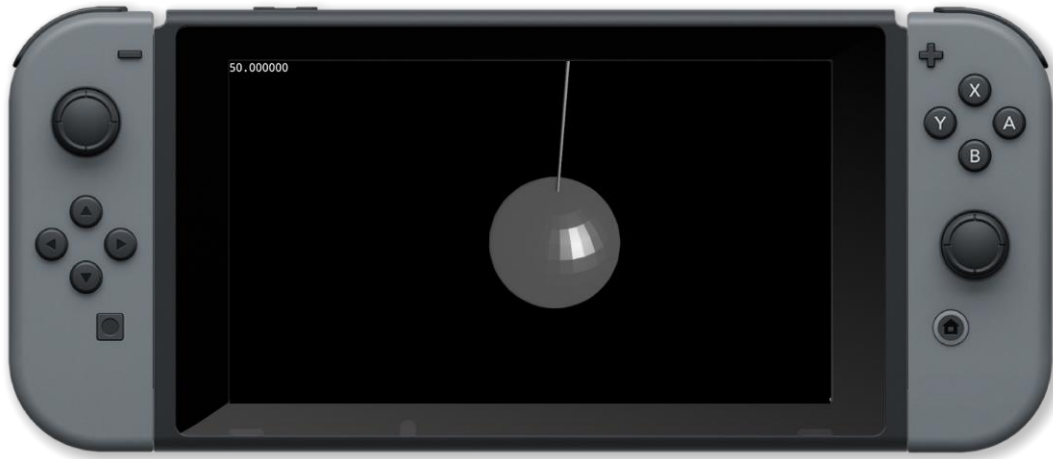
```
if condition then ... endif // ... is executed ONLY if condition is met  
if condition then ... else ... endif // if condition is met execute first ... otherwise execute second ...
```

Arguments

condition condition to be tested. This can be a compound condition using AND and OR

Example

```
setCamera( {0, 10, 10 }, { 0, 0, 0 } )  
bright = 50  
light = worldLight( { -5, -5, -5 }, white, bright )  
lighton = true  
ballmodel = loadModel( "Kat/Discoball" )  
ball = placeObject( ballmodel, { 0, 0, 0 }, { 10, 10, 10 } )  
  
loop  
  c = controls( 0 )  
  if c.x and not lighton then  
    light = worldLight( { -5, -5, -5 }, white, bright )  
    lighton = true  
  endif  
  if c.a and lighton then  
    removeLight( light )  
    lighton = false  
  endif  
  rotateObject( ball, { 0, 1, 0 }, 1.0 )  
  drawObjects()  
  printAt( 0, 0, "Press X to switch on the light" )  
  printAt( 0, 1, "Press A to switch off the light" )  
  update()  
repeat
```



Associated Commands

`and`, `else`, `endif`, `if`, `or`, `then`

KEYWORDS

to

Purpose

Separates the start and end values in a FOR loop

Description

The value before this is the start value of the index variable and the one after is the end value. The loop is executed until the value of the loop index variable goes from the start value to one step before the end value in increments of step

Syntax

```
for index = start to end loop ... repeat // Loop over values
```

```
for index = start to end step step loop ... repeat // Loop over values with step
```

Arguments

index loop index variable

start start value of index

end end value of index (loop is not executed with this value)

step amount to change index variable (default is one)

Example

```
// Draw 100 random boxes
clear()
for i = 1 to 100 loop
  // Pick random colour
  col = { random( 101 ) / 100, random( 101 ) / 100, random( 101 ) / 100, random( 101 ) / 100 }
  x = random( gWidth() )
  y = random( gHeight() )
  width = random( gWidth() / 4 )
  height = random( gHeight() / 4 )
  outline = random( 2 )
  box( x, y, width, height, col, outline )
  update()
repeat
// Wait 3 seconds
sleep( 3 )
```



Associated Commands

for, repeat, step, to, while

KEYWORDS

vector

Purpose

Create a variable of the vector type

Description

Initialises a variable of the vector type. Can be used to initialise an array or within a structure definition. The vector can have up to 4 dimensions (x, y, z and w / r, g, b and a)

Syntax

```
struct name
  vector field1
  ...
  typen fieldn
endStruct
```

Arguments

name name of the structure

field1 name of the first field

fieldn name of the last field

typen type of the last field

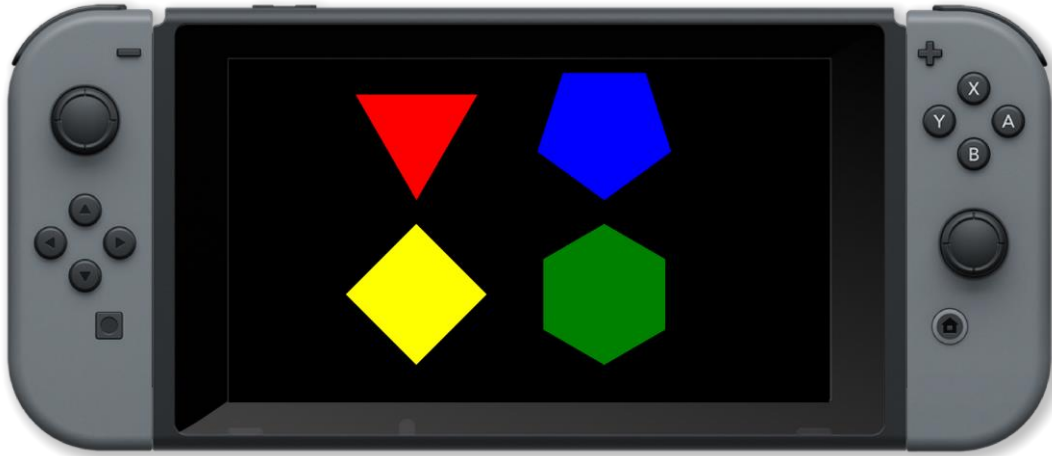
Example

```
struct shape
  string name
  int sides
  int size
  vector pos
  int col
endStruct

shape shapes[3]
shapes[0] = [ .name = "triangle", .sides=3, .size=150, .pos = { 400, 150 }, .col = red ]
shapes[1] = [ .name = "square", .sides=4, .size=150, .pos = { 400, 500 }, .col = yellow ]
shapes[2] = [ .name = "pentagon", .sides=5, .size=150, .pos = { 800, 150 }, .col = blue ]
shapes[3] = [ .name = "hexagon", .sides=6, .size=150, .pos = { 800, 500 }, .col = green ]

loop
  clear()
  printAt( 0, 0, "Press A to show labels" )
  c = controls( 0 )
  for i = 0 to 4 loop
```

```
drawShape( shapes[i], c.a )
repeat
update()
repeat
function drawshape( s, label )
circle( s.pos.x, s.pos.y, s.size, s.sides, s.col, 0 )
if label then
drawText( s.pos.x - s.size/2, s.pos.y, s.size / 5, black, s.name )
endIf
return void
```



Associated Commands

array, int, float, endStruct, string, struct, vector

KEYWORDS

void

Purpose

A value that indicates that a user defined function returns no value

Description

A special value that actually has “no value”

Syntax

```
function name() ... return void // function with no arguments
```

```
function name(argument1, ... argumentn) ... return void // function with n arguments
```

Arguments

name name of the function

argument1 first parameter of the function

argumentn last parameter of the function

Example

```
for size = 1 to 200 step 1 loop
  clear()
  centreText( "Hello World", size )
  update()
repeat

// Centre a text string on the screen
function centreText( message, size )
  textSize( size )
  tw = textWidth( message )
  drawtext( ( gWidth() - tw ) / 2, ( gHeight() - size ) / 2, size, white, message )
return void
```




Associated Commands

`function`, `return`, `void`

KEYWORDS

while

Purpose

Repeat a section of code

Description

Repeat a section of code until a condition is met

Syntax

```
while condition loop ... repeat // Loop while condition is true
```

Arguments

condition boolean condition that stops the loop when false

Example

```
image = loadImage( "Untied Games/Enemy small top C", false )
ship = []
for i = 0 to 2 loop
  ship[i] = createSprite()
  setSpriteImage( ship[i], image )
  setSpriteScale( ship[i], { 5, 5 } )
  setSpriteCollisionShape( ship[i], SHAPE_TRIANGLE, 25, 25, 180 )
  ship[i].show_collision_shape = true
repeat

setSpriteRotation( ship[0], 270 )
setSpriteSpeed( ship[0], { 240, 0 } )
setSpriteSpeed( ship[1], { 0, 120 } )
setSpriteColour( ship[1], { 0, 0, 1, 1 } )
setSpriteLocation( ship[0], { 0, gHeight() / 2 } )
setSpriteLocation( ship[1], { gWidth() / 2, 0 } )

collide = false
while !collide loop
  clear()
  updateSprites()
  drawSprites()
  update()
  collide = detectSpriteCollision( ship[0], ship[1] )
repeat
```



Associated Commands

for, repeat, step, to, while

OPERATORS

Symbols and Operators

Operators are tools to perform operations on a number. Some operators you might recognise already are +, -, <, etc. Operators are used to affect one or more items. For example, we might compare two items using operators, or add two items together.

Priority

Some operators have a higher priority than others. This means they happen **first** in a line of code even if they do not appear first. For example:

```
a = 10 * 30 + 5
```

In this example, the multiply will happen **before** the addition, giving a result of 350. This is because the multiplication operation has a **higher priority** than addition.

When operators have the same priority, they happen in order of appearance. For example:

```
a = 10 + 30 - 5
```

In this example, the addition happens **before** the subtraction because it **appears first**.

Below are all of the operators in **FUZE4 Nintendo Switch** listed in order of priority. Higher priority operators happen before lower. Operators in the same group have the same priority.

Group 0 - Brackets

Brackets behave in a special way. Anything inside brackets will take place first, even if the operations within the brackets have a lower priority than operations outside them. Of course, operations within the brackets still behave in terms of their normal priority.

() Parentheses - Used to change the order of evaluation. Operations inside parentheses will happen before operations on the outside. Parentheses are also used to set the arguments of a function call.

[] Square Brackets - Used with arrays to define a number of elements, or to index into an array.

{ } Curly Brackets - Used to define a vector. Empty curly brackets sets default values of { 0, 0, 0, 0 }.

Group 1 - Highest Priority

NOT, ! Not - Performs the logical inversion on a given value. False becomes true, true becomes false. Either NOT or ! can be used.

BNOT, ~ Binary Not - Performs the Not operation on a bitwise basis. Either BNOT or ~ can be used.

Group 2

* Multiply - Used to multiply a value by another.

/ Divide - Used to divide a value by another.

MOD, % Modulo - Used to give the remainder of a division. Either MOD or % can be used.

Group 3

+ Add - Used to add two values together. Also used to join string content together.

- Minus - Used to subtract a value from another. Also used as a sign to indicate a negative number.

<< Shift Left - Used to perform the bit shift operation to the left. Effectively multiplies a value by two for the number of bits shifted.

>> Shift Right - Used to perform the bit shift operation to the right. Effectively divides a value by two for the number of bits shifted.

& Binary And - Performs the **and** operation on a bitwise basis.

| Binary Or - Performs the **or** operation on a bitwise basis.

^ Exclusive Or - Used to perform the exclusive **or** operation

Group 4

== Double Equals - Used to compare two values and determine whether they are exactly equal to each other.

< Less Than - Used to compare two values and determine whether one is less than the other.

<= Less Than or Equal To - Used to compare to values and determine whether one is less than or equal to the other.

> Greater Than - Used to compare two values and determine whether one is greater than the other.

>= Greater Than or Equal To - Used to compare two values and determine whether one is greater than or equal to the other.

!= Not Equal To - Used to compare two values and determine whether they are not equal to each other.

Group 5

AND Logical And - Used to compare two values to determine whether they are both true.

Group 6

OR Logical Or - Used to compare two values to determine whether either are true.

Group 7 - Lowest Priority

= Assign - Used to assign a value to a variable.

- $+=$ Plus Equals - Used to perform an addition on the contents of a variable.
- $-=$ Minus Equals - Used to perform a subtraction on the contents of a variable.
- $*=$ Times Equals - Used to perform a multiplication on the contents of a variable.
- $/=$ Divide Equals - Used to perform a division on the contents of a variable.

OPERATORS

add

Purpose

Addition operator +

Description

Finds the sum of the numbers either side of the + operator. The result is the first number increased by the second

Syntax

```
result = number1 + number2
```

Arguments

result sum of number1 and number2

number1 first number

number2 second number

Example

```
answer = "0"
correct = 2 + 2
while int( answer ) != correct loop
    answer = input( "What is 2 + 2?", false )
    if int( answer ) != correct then
        print( "Sorry that is incorrect. Please try again" )
        for i = 0 to 200 loop
            update()
        repeat
    endif
repeat
print( "That is correct!" )
sleep( 3 )
```

Associated Commands

add, divide, modulus, multiply, subtract

OPERATORS

divide

Purpose

Division operator /

Description

This is the opposite of the multiplication operator. The result is the number of times you can subtract the first number from the second

Syntax

```
result = number1 / number2
```

Arguments

result number1 divided by number2

number1 first number

number2 second number

Example

```
message = "Hello World"  
for size = 1 to 200 step 1 loop  
  clear()  
  textSize( size )  
  tw = textWidth( message )  
  drawText( ( gWidth() - tw ) / 2, ( gHeight() - size ) / 2, size, white, message )  
  update()  
repeat
```



Associated Commands

add, divide, modulus, multiply, subtract

OPERATORS

modulus

Purpose

Modulus operator %

Description

Finds the remainder when one number is dived by another

Syntax

```
result = number1 % number2
```

Arguments

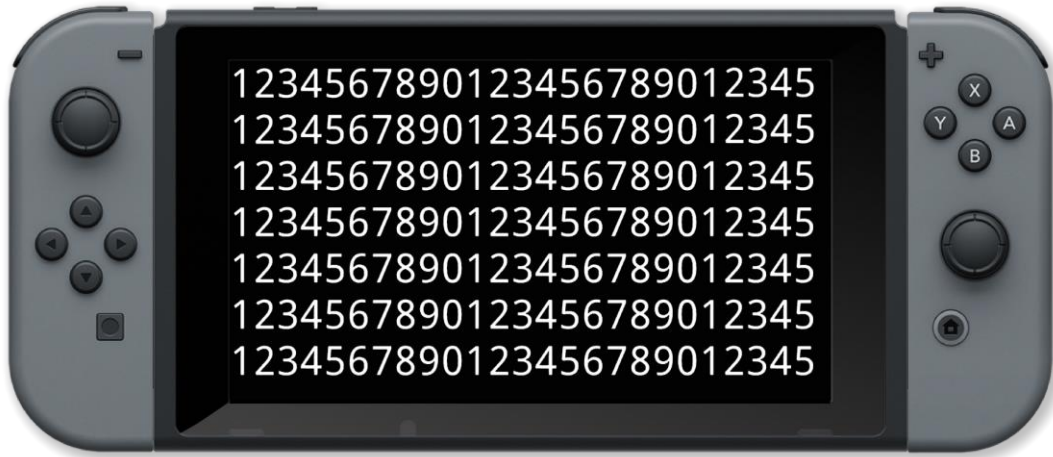
number1 first number

number2 second number

result the remainder when number1 is divided by number2

Example

```
textsize( 100 )
for y = 0 to theight() loop
  for x = 0 to twidth() loop
    printAt( x, y, ( x + 1 ) % 10 )
    update()
  repeat
repeat
for i = 1 to 100 loop
  update()
repeat
```



Associated Commands

[add](#), [divide](#), [modulus](#), [multiply](#), [subtract](#)

OPERATORS

multiply

Purpose

Multiplication operator *

Description

Finds the product of the numbers either side of the * operator. The result is the first number added to itself the second number of times.

Syntax

```
result = number1 * number2
```

Arguments

result product of number1 and number2

number1 first number

number2 second number

Example

```
answer = "0"  
correct = 6 * 7  
while ( int( answer ) != correct ) loop  
    clear()  
    answer = input( "What is 6 times 7?", false )  
    if ( int( answer ) != correct ) then  
        print ( "Sorry that is incorrect. Please try again" )  
        for i = 0 to 200 loop  
            update()  
        repeat  
    endif  
repeat  
print( "That is correct!" )  
sleep( 3 )
```

Associated Commands

add, divide, modulus, multiply, subtract

OPERATORS

subtract

Purpose

Subtraction operator -

Description

Finds the difference between the numbers either side of the - operator. The result is the first number decreased by the second

Syntax

```
result = number1 - number2
```

Arguments

result difference between number1 and number2

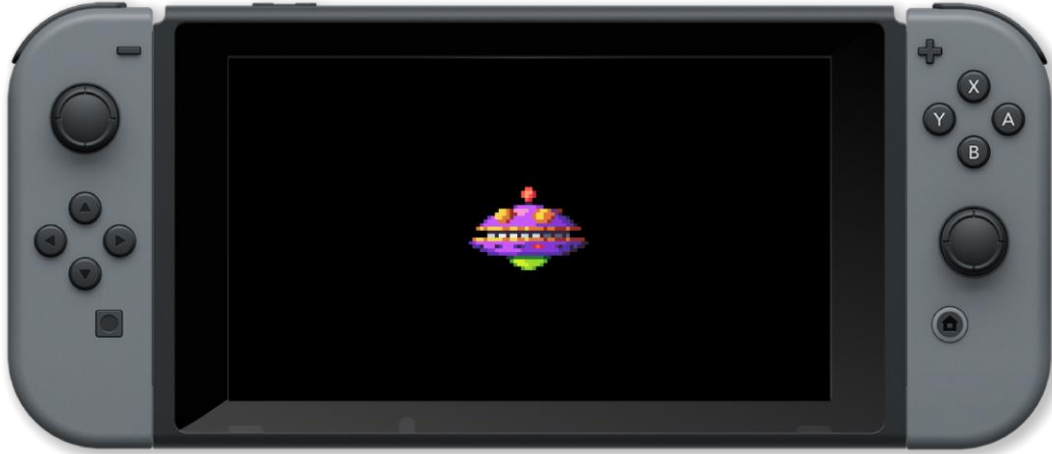
number1 first number

number2 second number

Example

```
image = loadImage( "Untied Games/Enemy A", false )
enemy = []
for i = 0 to 4 loop
    enemy[ i ] = createSprite()
    setSpriteImage( enemy[ i ], image)
    setSpriteAnimation( enemy[ i ], 0, 4, 20 )
    setSpriteLocation( enemy[ i ], { (i % 2) * 400 + 400, int(i / 2) * 300 + 200 } )
    setSpriteScale( enemy[ i ], { 4, 4 } )
repeat
    camera = getSpriteCamera()
    rotation = getSpriteCameraRotation()
loop
    clear()
    c = controls( 0 )
    printAt( 0, 0, "Camera position: x = ", camera.x, " y = ", camera.y, " z = ", camera.z, " rotation: ", rotation )
    printAt(0, 1, "Use left joypad to pan, right joypad to zoom/rotate")
    if c.up then
        camera.y -= 5
    endIf
    if c.down then
        camera.y += 5
    endIf
    if c.left then
        camera.x -= 5
    endIf
    if c.right then
        camera.x += 5
    endIf
    if c.x then
        camera.z += 0.05
    endIf
    if c.b then
        camera.z -= 0.05
    endIf
    if c.y then
        rotation -= 0.5
    endIf
    if c.a then
```

```
rotation += 0.5
endIf
setSpriteCamera( camera.x, camera.y, camera.z )
setSpriteCameraRotation( rotation )
updateSprites()
drawSprites()
update()
repeat
```



Associated Commands

add, divide, modulus, multiply, subtract

OPERATORS

and

Purpose

Bitwise and operator &

Description

Sets bits in the result where both equivalent bits are set in the number either side of the operator

Syntax

```
result = number1 & number2
```

Arguments

number1 first binary number

number2 second binary number

result resulting number with just the bits that are set in *number1* and *number2*

Example

```
loop
  clear()
  textSize( 50 )
  byte1 = 123
  byte2 = 234
  printAt( 0, 0, "byte1 = ", bin2str( byte1 ) )
  printAt( 0, 1, "byte2 = ", bin2str( byte2 ) )
  printAt( 0, 2, "byte1 & byte2 = ", bin2str( byte1 & byte2 ) )
  update()
repeat

function bin2str( byte )
  result = ""
  for i = 0 to 8 loop
    bit = byte & 1
    if bit then
      result = "1" + result
    else
      result = "0" + result
    endif
    byte = byte >> 1
  repeat
return result
```

Associated Commands

and, not, or, shiftLeft, shiftRight, xor

OPERATORS

not

Purpose

Bitwise not operator ~

Description

Toggles bits in a binary number so that 1s become 0s and 0s become 1s

Syntax

```
result = ~number
```

Arguments

number binary number

result result when you toggle all of the bits in number

Example

```
loop
  textsize( 50 )
  byte1 = 123
  byte2 = 234
  printat( 0, 0, "byte1 =           ", bin2str( byte1 ))
  printat( 0, 1, "byte2 =           ", bin2str( byte2 ))
  printat( 0, 2, "~byte1 =          ", bin2str( ~byte1 ))
  printat( 0, 3, "~byte2 =          ", bin2str( ~byte2 ))
  update()
repeat

function bin2str( byte )
  result = ""
  for i = 0 to 8 loop
    bit = byte & 1
    if bit then
      result = "1" + result
    else
      result = "0" + result
    endif
    byte = byte >> 1
  repeat
return result
```

Associated Commands

[and](#), [not](#), [or](#), [shiftLeft](#), [shiftRight](#), [xor](#)

OPERATORS

or

Purpose

Bitwise or operator |

Description

Performs the OR operation on a bitwise basis. Checks one bit against another and returns true if either are true

Syntax

```
result = number1 | number2
```

Arguments

number1 first binary number

number2 second binary number

result resulting number with the bits that are set in *number1* or *number2*

Example

```
loop
  textsize( 50 )
  byte1 = 123
  byte2 = 234
  printAt( 0, 0, "byte1 = ", bin2str( byte1 ) )
  printAt( 0, 1, "byte2 = ", bin2str( byte2 ) )
  printAt( 0, 2, "byte1 | byte2 = ", bin2str( byte1 | byte2 ) )
  update()
repeat

function bin2str( byte )
  result = ""
  for i = 0 to 8 loop
    bit = byte & 1
    if bit then
      result = "1" + result
    else
      result = "0" + result
    endIf
    byte = byte >> 1
  repeat
return result
```

Associated Commands

and, not, or, shiftLeft, shiftRight, xor

OPERATORS

shiftLeft

Purpose

Bit shift left operator <<

Description

Shift all of the bits in a binary number to the left the specified number of times. The new rightmost bits are set to zero. Bitshift left has the effect of multiplying the value of the binary number by 2

Syntax

```
result = number1 << number2
```

Arguments

number1 first binary number

number2 number of bits to shift

result resulting number with the bits of *number1* shifted left *number2* times

Example

```
loop
  textsize( 50 )
  byte1 = 123
  byte2 = 234
  printAt( 0, 0, "byte1 =      ", bin2str( byte1 ) )
  printAt( 0, 1, "byte2 =      ", bin2str( byte2 ) )
  printAt( 0, 2, "byte1 << 1 =   ", bin2str( byte1 << 1 ) )
  update()
repeat

function bin2str( byte )
  result = ""
  for i = 0 to 8 loop
    bit = byte & 1
    if bit then
      result = "1" + result
    else
      result = "0" + result
    endIf
    byte = byte >> 1
  repeat
return result
```

Associated Commands

and, not, or, shiftLeft, shiftRight, xor

OPERATORS

shiftRight

Purpose

Bit shift right operator >>

Description

Shift all of the bits in a binary number to the right the specified number of times. The new leftmost bits are set to zero

Syntax

```
result = number1 >> number2
```

Arguments

number1 first binary number

number2 number of bits to shift

result resulting number with the bits of *number1* shifted right *number2* times

Example

```
loop
  textsize( 50 )
  byte1 = 123
  byte2 = 234
  printAt( 0, 0, "byte1 = ", bin2str( byte1 ) )
  printAt( 0, 1, "byte2 = ", bin2str( byte2 ) )
  printAt( 0, 2, "byte1 >> 1 = ", bin2str( byte1 >> 1 ) )
  update()
repeat

function bin2str( byte )
  result = ""
  for i = 0 to 8 loop
    bit = byte & 1
    if bit then
      result = "1" + result
    else
      result = "0" + result
    endIf
    byte = byte >> 1
  repeat
return result
```

Associated Commands

and, not, or, shiftLeft, shiftRight, xor

OPERATORS

xor

Purpose

Bitwise exclusive or operator ^

Description

Sets bits in the result where equivalent bits are different in the number either side of the operator

Syntax

```
result = number1 ^ number2
```

Arguments

number1 first binary number

number2 second binary number

result resulting number with the bits set that are different between *number1* and *number2*

Example

```
loop
  textsize( 50 )
  byte1 = 123
  byte2 = 234
  printAt( 0, 0, "byte1 = ", bin2str( byte1 ) )
  printAt( 0, 1, "byte2 = ", bin2str( byte2 ) )
  printAt( 0, 2, "byte1 ^ byte2 = ", bin2str( byte1 ^ byte2 ) )
  update()
repeat

function bin2str( byte )
  result = ""
  for i = 0 to 8 loop
    bit = byte & 1
    if bit then
      result = "1" + result
    else
      result = "0" + result
    endif
    byte = byte >> 1
  repeat
return result
```

Associated Commands

and, not, or, shiftLeft, shiftRight, xor

OPERATORS

equals

Purpose

Equals operator ==

Description

This is true if the value of first expression is equal to the value of the second

Syntax

```
result = expression1 == expression2
```

Arguments

result true if value of expression1 is equal to the value of expression2

expression1 first expression

expression2 second expression

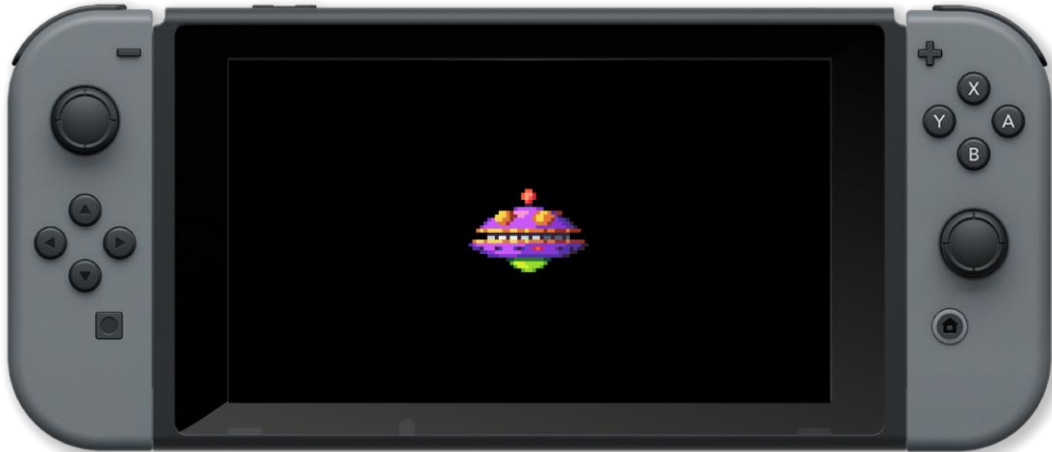
Example

```
image = loadImage( "Untied Games/Enemy A", false )
enemy = []
for i = 0 to 4 loop
    enemy[i] = createSprite()
    setSpriteImage( enemy[i], image )
    setSpriteAnimation( enemy[i], image, 0, 4, 20 )
    setSpriteLocation( enemy[i], { ( i % 2 ) * 400 + 400, int( i / 2 ) * 300 + 200 } )
    setSpriteScale( enemy[i], { 4, 4 } )
repeat

camera = getSpriteCamera()
rotation = getSpriteCameraRotation()

loop
    clear()
    c = controls( 0 )
    printAt( 0, 0, "Camera position: x = ", camera.x, " y = ", camera.y, " z = ", camera.z, " rotation: ", rotation )
    printAt( 0, 1, "Use left joypad to pan, right joystick to zoom/rotate" )
    if c.up then
        camera.y -= 5
    endIf
    if c.down then
        camera.y += 5
    endIf
    if c.left then
        camera.x -= 5
    endIf
    if c.right then
        camera.x += 5
    endIf
    if c.x then
        camera.z += 0.05
    endIf
    if c.b then
        camera.z -= 0.05
    endIf
    if c.y then
        rotation -= 0.5
    endIf
    if c.a then
```

```
rotation += 0.5
endIf
setSpriteCamera( camera.x, camera.y, camera.z )
setSpriteCameraRotation( rotation )
updateSprites()
drawSprites()
update()
repeat
```



Associated Commands

[equals](#), [lessThan](#), [greaterThan](#), [lessThanEquals](#), [greaterThanEquals](#), [notEquals](#)

OPERATORS

greaterThan

Purpose

Greater than operator >

Description

This is true if the value of first expression is greater than the value of the second

Syntax

```
result = expression1 > expression2
```

Arguments

result true if value of expression1 is greater than value of expression2

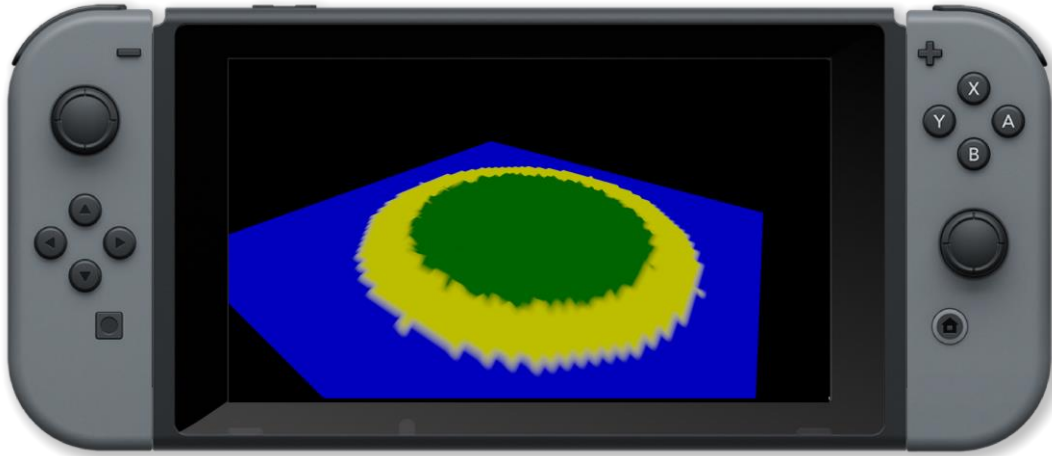
expression1 first expression

expression2 second expression

Example

```
gsize = 64
landScape = createTerrain( gsize, 1 )
height = 0
colour = white
for x = 0 to gsize loop
    for y = 0 to gsize loop
        d = distance ( { x, y }, { gsize / 2, gsize / 2 } )
        if d > 24 then // sea level
            height = 0
            colour = blue
        else
            if (d > 18) then // beach
                height = 1
                colour = yellow
            else // hills
                height = rnd(2) + 1
                colour = green
            endif
        endif
        setTerrainPoint( landscape, x, y, height, colour )
    repeat
repeat
setCamera( { gsize / 2, 50, gsize / 2 }, { gsize / 2.0, 0, gsize / 2.00001 } )
setAmbientLight( { 0.5, 0.5, 0.5 } )
```

```
island = placeObject( landscape, { gsize / 2, 0, gsize / 2 }, { 1, 1, 1 } )  
  
loop  
  c = controls( 0 ) // rotate using joysticks  
  rotateObject( island, { 1, 0, 0 }, c.ly )  
  rotateObject( island, { 0, 0, 1 }, c.lx )  
  rotateObject( island, { 0, 1, 0 }, c.rx )  
  drawObjects()  
  update()  
repeat
```



Associated Commands

[equals](#), [lessThan](#), [greaterThan](#), [lessThanEquals](#), [greaterThanEquals](#), [notEquals](#)

OPERATORS

greaterThanEquals

Purpose

Greater than or equals operator >=

Description

This is true if the value of first expression is greater than or equal to the value of the second

Syntax

```
result = expression1 >= expression2
```

Arguments

result true if value of expression1 is greater than or equal to value of expression2

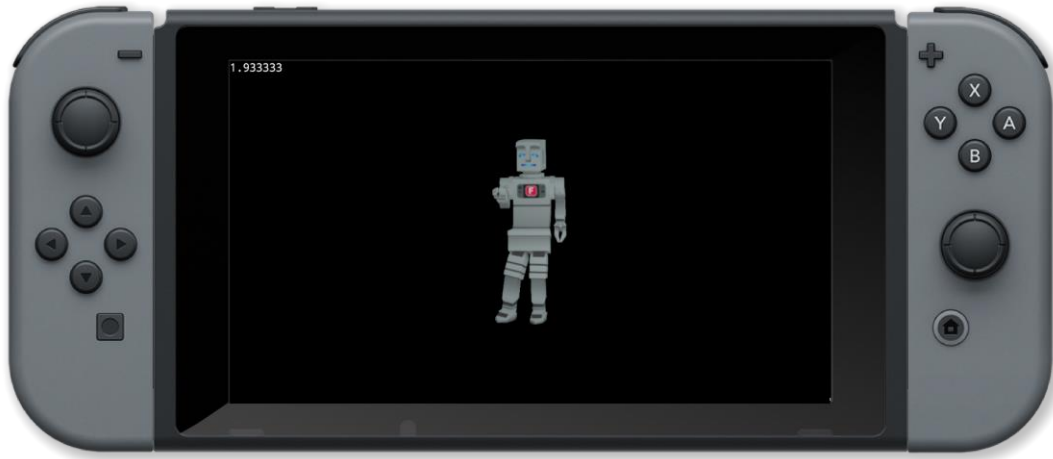
expression1 first expression

expression2 second expression

Example

```
cb = loadModel( "Kat/Colin" )
pointLight( { 0.5, 1.3, 2 }, white, 4 )
setAmbientLight( { 0.5, 0.5, 0.5 } )
colin = placeObject( cb, { 0, 0, 0 }, { 1, 1, 1 } )
setCamera( { 0, 10, 10 }, { 0, 5, 0 } )
animID = 7 // the robot
animlength = animationLength( colin, animID )
animframe = 0

loop
  clear()
  animframe = animframe + 1 / 60
  if animframe >= animlength then
    animframe = 0
  endIf
  updateAnimation( colin, animID, animframe )
  drawObjects()
  printAt( 0, 0, "length: ", animlength, " frame: ", animframe )
  update()
repeat
```

Associated Commands

`equals`, `lessThan`, `greaterThan`, `lessThanEquals`, `greaterThanEquals`, `notEquals`

OPERATORS

lessThan

Purpose

Less than operator <

Description

This is true if the value of first expression is less than the value of the second

Syntax

```
result = expression1 < expression2
```

Arguments

result true if value of expression1 is less than value of expression2

expression1 first expression

expression2 second expression

Example

```
hour="00"  
minute="00"  
second="00"  
size=100.0  
textsize(size)  
loop  
  clear()  
  c = clock()  
  if c.hour < 10 then  
    hour= "0" + str( c.hour )  
  else  
    hour= str(c.hour)  
  endif  
  if c.minute < 10 then  
    minute = "0" + str( c.minute )  
  else  
    minute=str( c.minute )  
  endif  
  if (c.second < 10) then  
    second = "0" + str( c.second )  
  else  
    second = str( c.second )  
  endif  
  now = hour + ":" + minute + ":" + second  
  tw = textWidth( now )
```

```
drawText( (gWidth() - tw) / 2, ( gHeight() - size ) / 2, size, white, now )  
update()  
repeat
```

Associated Commands

[equals](#), [lessThan](#), [greaterThan](#), [lessThanEquals](#), [greaterThanEquals](#), [notEquals](#)

OPERATORS

lessThanEquals

Purpose

Less than or equal operator <=

Description

This is true if the value of first expression is less than or equal to the value of the second

Syntax

```
result = expression1 <= expression2
```

Arguments

result true if value of expression1 is less or equal to the value of expression2

expression1 first expression

expression2 second expression

Example

```
pos = { 960, 540 }
vel = { 0, 0 }
col = { 1, 0, 0, 1 }
rt = createImage( 1920, 1080, true, image_rgb )
loop
  c = controls( 0 )
  vel += { c.lx, -c.ly }
  pos += vel
  vel *= 0.95

  if col.r > 0 and col.b <= 0 then
    col.r -= 0.01
    col.g += 0.01
  else
    if col.g > 0 then
      col.g -= 0.01
      col.b += 0.01
    else
      col.b -= 0.01
      col.r += 0.01
    endif
  endif
endif

setDrawTarget( rt )
box( 0, 0, 1920, 1080, { 0, 0, 0, 0.25 }, false )
circle( pos.x, pos.y, 50, 32, col, false )

setDrawTarget( framebuffer )
clear()
renderEffect( rt, framebuffer, fx_motionblur, [ 1 / 1920, 1 / 1080, vel.x / 2, vel.y / 2 ] )
```

```
update()  
repeat
```



Associated Commands

`equals`, `lessThan`, `greaterThan`, `lessThanEquals`, `greaterThanEquals`, `notEquals`

OPERATORS

notEquals

Purpose

Not equals operator !=

Description

This is true if the value of first expression is not equal to the value of the second

Syntax

```
result = expression1 != expression2
```

Arguments

result true if value of expression1 is not equal to the value of expression2

expression1 first expression

expression2 second expression

Example

```
radians( 1 )
image = loadImage( "Untied Games/Enemy small top C", false )
ship = createSprite( )
setSpriteImage( ship, image )
lastpos = { gwidth() / 2, gheight() / 2 }
setSpriteLocation( ship, lastpos )
setSpriteScale( ship, { 4, 4 } )
loop
  clear( )
  c = controls( 0 )
  printAt( 0, 0, "Use left joystick to control sprite" )
  setSpriteSpeed( ship, { 480 * c.lx, -480 * c.ly } )
  curpos = getSpriteLocation( ship )
  if( curpos != lastpos ) then
    setSpriteRotation( ship, -pi / 2 + atan2(curpos.y - lastpos.y, curpos.x - lastpos.x) )
    lastpos = curpos
  endif
  updateSprites()
  drawSprites()
  update()
repeat
```



Associated Commands

`equals`, `lessThan`, `greaterThan`, `lessThanEquals`, `greaterThanEquals`, `notEquals`

COMMAND REFERENCE

COMMAND REFERENCE

2D Graphics

COMMAND REFERENCE

box()

Purpose

Draw a box (rectangle)

Description

Draws a filled or outline box with the given width and height at the specified x and y coordinates in the specified colour.

Syntax

```
box( x, y, width, height, colour, outline )
```

Arguments

x horizontal screen position in pixels

y vertical screen position in pixels

width width in pixels

height height in pixels

colour colour name or RGB values { red, green, blue, opacity } between 0 and 1

outline If true then only the outline is drawn otherwise the shape is filled.

Example

```
// Draw 100 random boxes
clear()
for i = 0 to 100 loop
  // Pick random colour
  col = { random( 101 ) / 100, random( 101 ) / 100, random( 101 ) / 100, random( 101 ) / 100 }
  x = random( gWidth() )
  y = random( gHeight() )
  width = random( gWidth() / 4 )
  height = random( gHeight() / 4 )
  outline = random( 2 )
  box( x, y, width, height, col, outline )
  update()
repeat
// Wait 3 seconds
sleep( 3 )
```



Associated Commands

`circle()`, `line()`, `triangle()`

COMMAND REFERENCE

centreSpriteCamera()

Purpose

Centre the sprite camera

Description

Used to centre the sprite camera around a supplied position

Syntax

```
centreSpriteCamera( pos )  
centreSpriteCamera( xpos, ypos )
```

Arguments

pos vector position of the centre point { x, y }

xpos float position of the centre point on the x axis

ypos float position of the centre point on the y axis

Example

```
image = loadImage( "Untied Games/Shroom Hopper A" )  
spr = createSprite()  
setSpriteImage( spr, img )  
setSpriteAnimation( spr, 42, 49, 10 )  
setSpriteCamera( 0, 0, 8 )  
  
centrePoint = { 0, 0 }  
  
loop  
  clear()  
  updateSprites()  
  j = controls( 0 )  
  centrePoint += { -j.lx, j.ly }  
  // centre sprite camera around { 0, 0 }  
  // use left control stick to adjust centre point  
  centreSpriteCamera( centrePoint )  
  drawSprites()  
  update()  
repeat
```



Associated Commands

`setSpriteCamera()`, `getSpriteCamera()`, `getSpriteCameraRotation()`, `setSpriteCameraRotation()`

COMMAND REFERENCE

circle()

Purpose

Draw a circle

Description

Draws a filled or outline circle with the given radius at the specified x and y coordinates and in the specified colour.

Syntax

```
circle( x, y, radius, vertices, colour, outline )
```

Arguments

x horizontal screen position in pixels

y vertical screen position in pixels

radius radius in pixels

vertices number of vertices in the circle (higher will be smoother)

colour colour name or RGB values { red, green, blue, opacity } between 0 and 1

outline If true then only the outline is drawn otherwise the shape is filled.

Example

```
clear()
for i = 1 to 100 loop
// Pick random colour
  col = { random( 101 ) / 100, random( 101 ) / 100, random( 101 ) / 100, random( 101 ) / 100 }
  x = random( gWidth() )
  y = random( gHeight() )
  radius = random( gWidth() / 4 )
  vertices = 32
  outline = random( 2 )
  circle( x, y, radius, vertices, col, outline )
  update()
repeat
// Wait 10 seconds
sleep( 10 )
```



Associated Commands

`box()`, `line()`, `triangle()`

COMMAND REFERENCE

collideMap()

Purpose

Used to cause sprites to interact with map collision box data.

Description

Receives a sprite handle and returns an array of structures detailing collision data.

Syntax

```
collide = detectMapCollision( sprite )
```

Arguments

sprite Handle of the sprite being collided

*collide.exist Boolean value (true or false) to indicate whether a collision has occurred

*collide.spriteHandle of the sprite being collided

*collide.spriteHandle of the sprite being collided

*collide.resolResolution vector of the given sprite

*collide.resolResolution vector of the given sprite

Example

```
// To view this map demo, please load the project "collideMap() Demo" from FUZE Programs.  
// Maps must be stored in the project you wish to load them into.  
  
img = loadImage( "Untied Games/Bat and Ball ball" )  
  
plr = [  
    .spr = createSprite(),  
    .vel = {}  
]  
  
setSpriteImage( plr.spr, img )  
setSpriteScale( plr.spr, { 1, 1 } )  
  
loadMap( "map1" )  
  
setSpriteCamera( 0, 0, 2 )  
  
loop  
    centreSpriteCamera( 0, 0 )  
    clear()  
    updateSprites()  
  
c = controls( 0 )
```



```

plr.vel += { c.lx, -c.ly } * 80
plr.vel *= 0.87

setSpriteSpeed( plr.spr, plr.vel )

drawMapLayer( 0 )
drawMapLayer( 1 )

collision = collideMap( plr.spr )

drawSprites()

printAt( 0, 0, "Move the ball using the left control stick" )
printAt( 0, 2, "Collision exists: " )
printAt( 0, 4, "Collision Resolution Vector: " )

if len( collision ) > 0 then
    printAt( 30, 2, collision[0].exists )
    printAt( 30, 4, collision[0].resolution_a )
endif

update()
repeat

```

Associated Commands

COMMAND REFERENCE

collideSprites()

Purpose

Collides two sprites

Description

Syntax

```
c = collideSprites( spriteA, spriteB, resolve1, resolve2 )
```

Arguments

spriteA handle of first sprite

spriteB handle of second sprite

resolve1 if true the first sprite can be moved by the collision

resolve2 if true the second sprite can be moved by the collision

c.exists true if collision occurred

c.a first sprite in the collision

c.b second sprite in the collision

**c.resolution_vector* representing how sprite A was pushed during the collision

**c.resolution_vector* representing how sprite B was pushed during the collision

Example

```
image = loadImage( "Untied Games/Enemy small top C", false )
ship = []
for i = 0 to 2 loop
    ship[i] = createSprite( image )
    setSpriteImage( ship[i], image )
    setSpriteScale( ship[i], { 5, 5 } )
    setSpriteCollisionShape( ship[i], SHAPE_TRIANGLE, 25, 25, 180 )
    ship[i].show_collision_shape = true
repeat
setSpriteRotation( ship[0], 270 )
setSpriteSpeed( ship[0], { 240, 0 } )
setSpriteSpeed( ship[1], { 0, 120 } )
setSpriteColour( ship[1], { 0, 0, 1, 1 } )
setSpriteLocation( ship[0], { 0, gHeight() / 2 } )
setSpriteLocation( ship[1], { gWidth() / 2, 0 } )
while ship[0].x < gWidth() loop
    clear()
```

```
updateSprites()  
collideSprites( ship[0], ship[1] )  
drawSprites()  
update()  
repeat
```



Associated Commands

`detectSpriteCollision()`, `setSpriteCollisionShape()`

COMMAND REFERENCE

copyImage()

Purpose

Create a copy of an image

Description

Creates a copy of an image with adjustable source region

Syntax

```
handle = copyImage( imageHandle, source )
```

Arguments

handle Variable which will store the new image

imageHandle Handle of the image to copy

source Vector describing the desired region to copy

Example

```
// load image to copy
img = loadImage("Ansimuz/CyberpunkStreetLayer0")
// create copy with region x = 50, y = 50, width = 100, height = 100
img_copy = copyImage( img, { 50, 50, 100, 100 } )

// draw original image at top left of screen with a scale of 2
drawImage( img, 0, 0, 2 )
// draw copied image region at centre of screen with a scale of 3
drawImage( img_copy, gwidth() / 2, 0, 3 )

update()
sleep( 3 )
```



Associated Commands

`clear()`, `createImage()`, `drawImage()`, `drawImageEx()`, `drawQuad()`, `drawSheet()`, `loadImage()`,
`update()`, `uploadImage()`

COMMAND REFERENCE

copyShape()

Purpose

Copy a created shape

Description

Creates a copy of a supplied shape to be drawn with `drawShape()`

Syntax

```
newShape = copyShape( shape )
```

Arguments

newShape Handle which stores the newly copied shape

shape Handle which stores the shape to copy

Example

Associated Commands

`createBox()`, `createCircle()`, `createCurve()`, `createLine()`, `createLineStrip()`, `createPoly()`, `createStar()`, `createTriangle()`, `deleteShape()`, `drawShape()`, `getShapeBounds()`, `getShapeLocation()`, `getShapeRotation()`, `getShapeScale()`, `getShapeTint()`, `getVertex()`, `getVertexColour()`, `getVertexLineColour()`, `getVertexLineThickness()`, `joinShapes()`, `moveShape()`, `numVerts()`, `rotateShape()`, `scaleShape()`, `setShapeColour()`, `setShapeLineStyle()`, `setShapeRotation()`, `setShapeScale()`, `setShapeScaleModeLocal()`, `setShapeTint()`, `setVertex()`, `setVertexColour()`, `setVertexLineStyle()`

COMMAND REFERENCE

createBox()

Purpose

Creates a box (rectangle) to be drawn with `drawShape()`

Description

Creates a box with centre origin to be drawn at the specified x and y location with the specified width and height

Syntax

```
shape = createBox( x, y, width, height )
```

Arguments

shape Handle which stores the newly created shape

x Horizontal screen position in pixels

y Vertical screen position in pixels

width Width in pixels

height Height in pixels

Example

```
// draw a multicoloured rectangle on the screen
box_1 = createBox( gwidth() / 2, gheight() / 2, gwidth(), gheight() )

setVertexColour( box_1, 0, bisque )
setVertexColour( box_1, 1, cyan )
setVertexColour( box_1, 2, fuzeblue )
setVertexColour( box_1, 3, fuzepink )

drawShape( box_1 )
update()
sleep( 3 )
```



Associated Commands

`copyShape()`, `createCircle()`, `createCurve()`, `createLine()`, `createLineStrip()`, `createPoly()`,
`createStar()`, `createTriangle()`, `deleteShape()`, `drawShape()`, `getShapeBounds()`,
`getShapeLocation()`, `getShapeRotation()`, `getShapeScale()`, `getShapeTint()`, `getVertex()`,
`getVertexColour()`, `getVertexLineColour()`, `getVertexLineThickness()`, `joinShapes()`, `moveShape()`,
`numVerts()`, `rotateShape()`, `scaleShape()`, `setShapeColour()`, `setShapeLineStyle()`,
`setShapeRotation()`, `setShapeScale()`, `setShapeScaleModeLocal()`, `setShapeTint()`, `setVertex()`,
`setVertexColour()`, `setVertexLineStyle()`

COMMAND REFERENCE

createCircle()

Purpose

Creates a circle to be drawn with `drawShape()`

Description

Creates a circle with centre origin to be drawn at the specified x and y location with the specified radius and number of vertices

Syntax

```
shape = createCircle( x, y, radius, vertices )
```

Arguments

shape Handle to store the newly created circle

x Horizontal screen position of the circle

y Vertical screen position of the circle

radius Distance from the centre of the circle to the edge (in pixels)

vertices Number of vertices (points) making up the circle

Example

```
// draw a multicoloured circle on screen
shape_1 = createCircle( gwidth() / 2, gheight() / 2, 500, 360 )

for i = 0 to 360 loop
    setVertexColour( shape_1, i, { random( 1.0 ), random( 1.0 ), random( 1.0 ), 1 } )
repeat

drawShape( shape_1 )
update()
sleep( 3 )
```

Associated Commands

`copyShape()`, `createBox()`, `createCurve()`, `createLine()`, `createLineStrip()`, `createPoly()`, `createStar()`, `createTriangle()`, `deleteShape()`, `drawShape()`, `getShapeBounds()`, `getShapeLocation()`, `getShapeRotation()`, `getShapeScale()`, `getShapeTint()`, `getVertex()`, `getVertexColour()`, `getVertexLineColour()`, `getVertexLineThickness()`, `joinShapes()`, `moveShape()`, `numVerts()`, `rotateShape()`, `scaleShape()`, `setShapeColour()`, `setShapeLineStyle()`, `setShapeRotation()`, `setShapeScale()`, `setShapeScaleModeLocal()`, `setShapeTint()`, `setVertex()`, `setVertexColour()`, `setVertexLineStyle()`

COMMAND REFERENCE

createCurve()

Purpose

Create a curve between points

Description

Creates a curve between any number of supplied points on screen to be drawn with `drawShape()`

Syntax

```
shape = createCurve( point1, point2, ... pointN )  
shape = createCurve( points )
```

Arguments

shape Handle which stores the newly created curve

point1 Vector describing the position of the first point in the curve

point2 Vector describing the position of the second point in the curve

pointN Vector describing the position of the Nth point in the curve (any number of points can be supplied)

points Array of vector points to draw the curve between

Example

Associated Commands

`copyShape()`, `createBox()`, `createCircle()`, `createLine()`, `createLineStrip()`, `createPoly()`, `createStar()`, `createTriangle()`, `deleteShape()`, `drawShape()`, `getShapeBounds()`, `getShapeLocation()`, `getShapeRotation()`, `getShapeScale()`, `getShapeTint()`, `getVertex()`, `getVertexColour()`, `getVertexLineColour()`, `getVertexLineThickness()`, `joinShapes()`, `moveShape()`, `numVerts()`, `rotateShape()`, `scaleShape()`, `setShapeColour()`, `setShapeLineStyle()`, `setShapeRotation()`, `setShapeScale()`, `setShapeScaleModeLocal()`, `setShapeTint()`, `setVertex()`, `setVertexColour()`, `setVertexLineStyle()`

COMMAND REFERENCE

createImage()

Purpose

Create an image

Description

Create an image of the specified size and type which can then be drawn onto

Syntax

```
handle = createImage( width, height, filter, type )
```

Arguments

handle Variable which stores the desired image file

width Desired on-screen width in pixels

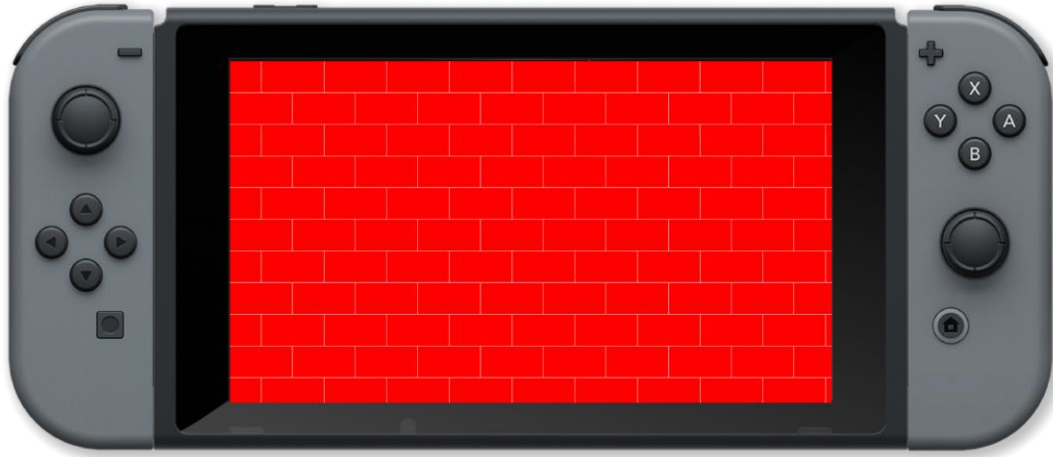
height Desired on-screen height in pixels

filter Sets filtering on (true) or off (false) - generally on for real images and off for pixel art

type The type of the image: image_rgb for 24 bits per pixel, image_rgba for 32bpp with alpha channel, image_rgb_hdr for 48bpp, image_rgba_hdr for 64 bpp with alpha channel

Example

```
w = 200
// Create a tile
img = createImage( w, w, true, image_rgb )
setDrawTarget( img )
box( 0, 0, w, w, red, 0 )
box( 0, w/2, w - 1, w / 2, white, 1 )
line( { w / 2 }, { w / 2, w / 2 }, white )
// draw tiles on the screen
setDrawTarget( framebuffer )
for y = 1 to gHeight() step w loop
  for x = 1 to gWidth() step w loop
    drawImage( img, x, y, 1 )
    update()
    sleep( 0.2 )
  repeat
repeat
sleep( 3 )
```



Associated Commands

`clear()`, `drawImage()`, `drawImageEx()`, `drawQuad()`, `drawSheet()`, `loadImage()`, `update()`, `uploadImage()`

COMMAND REFERENCE

createLine()

Purpose

Create a line

Description

Creates a line between two supplied points on screen to be drawn with `drawShape()`

Syntax

```
shape = createLine( x1, y1, x2, y2 )
```

Arguments

shape Handle which stores the newly created line

x1 Horizontal screen position of the first point in the line

y1 Vertical screen position of the first point in the line

x2 Horizontal screen position of the second point in the line

y2 Vertical screen position of the second point in the line

Example

Associated Commands

`copyShape()`, `createBox()`, `createCircle()`, `createCurve()`, `createLineStrip()`, `createPoly()`, `createStar()`, `createTriangle()`, `deleteShape()`, `drawShape()`, `getShapeBounds()`, `getShapeLocation()`, `getShapeRotation()`, `getShapeScale()`, `getShapeTint()`, `getVertex()`, `getVertexColour()`, `getVertexLineColour()`, `getVertexLineThickness()`, `joinShapes()`, `moveShape()`, `numVerts()`, `rotateShape()`, `scaleShape()`, `setShapeColour()`, `setShapeLineStyle()`, `setShapeRotation()`, `setShapeScale()`, `setShapeScaleModeLocal()`, `setShapeTint()`, `setVertex()`, `setVertexColour()`, `setVertexLineStyle()`

COMMAND REFERENCE

createLineStrip()

Purpose

Create a series of lines between points

Description

Creates a line between any number of supplied points on screen to be drawn with `drawShape()`

Syntax

```
shape = createLineStrip( point1, point2, ... pointN )  
shape = createLineStrip( points )
```

Arguments

shape Handle which stores the newly created shape

point1 Vector describing the position of the first point in the line

point2 Vector describing the position of the second point in the line

pointN Vector describing the position of the Nth point in the line (any number of points can be supplied)

points Array of vector points to draw lines between

Example

Associated Commands

`copyShape()`, `createBox()`, `createCircle()`, `createCurve()`, `createLine()`, `createPoly()`, `createStar()`, `createTriangle()`, `deleteShape()`, `drawShape()`, `getShapeBounds()`, `getShapeLocation()`, `getShapeRotation()`, `getShapeScale()`, `getShapeTint()`, `getVertex()`, `getVertexColour()`, `getVertexLineColour()`, `getVertexLineThickness()`, `joinShapes()`, `moveShape()`, `numVerts()`, `rotateShape()`, `scaleShape()`, `setShapeColour()`, `setShapeLineStyle()`, `setShapeRotation()`, `setShapeScale()`, `setShapeScaleModeLocal()`, `setShapeTint()`, `setVertex()`, `setVertexColour()`, `setVertexLineStyle()`

COMMAND REFERENCE

createPoly()

Purpose

Creates a polygon to be drawn with `drawShape()`

Description

Creates a polygon with centre origin to be drawn at the specified x and y location with the specified dimensions and number of points

Syntax

```
shape = createPoly( point1, point2, ... pointN )  
shape = createPoly( points )
```

Arguments

shape Handle to store the newly created shape

point1 Position vector for the first point of the polygon

point2 Position vector for the the second point of the polygon

pointN Position vector for the Nth point of the polygon (you can create any number of points)

points Array of vectors describing the desired points of the polygon

Example

```
// draw a multicoloured irregular 4 sided polygon on screen  
w = gwidth()  
h = gheight()  
  
points = [  
  { w / 3, h / 3 },  
  { w - w / 3, h / 3 },  
  { w - w / 4, h / 1.5 },  
  { w / 8, h / 2.5 }  
]  
  
cols = [  
  red,  
  green,  
  blue,  
  bisque  
]  
  
shape_1 = createPoly( points )
```

```
for i = 0 to len( cols ) loop
    setVertexColour( shape_1, i, cols[i] )
repeat

drawShape( shape_1 )
update()
sleep( 3 )
```



Associated Commands

copyShape(), createBox(), createCircle(), createCurve(), createLine(), createLineStrip(), createStar(), createTriangle(), deleteShape(), drawShape(), getShapeBounds(), getShapeLocation(), getShapeRotation(), getShapeScale(), getShapeTint(), getVertex(), getVertexColour(), getVertexLineColour(), getVertexLineThickness(), joinShapes(), moveShape(), numVerts(), rotateShape(), scaleShape(), setShapeColour(), setShapeLineStyle(), setShapeRotation(), setShapeScale(), setShapeScaleModeLocal(), setShapeTint(), setVertex(), setVertexColour(), setVertexLineStyle()

COMMAND REFERENCE

createSprite()

Purpose

Create a new sprite.

Description

Create a new sprite

Syntax

```
handle = createSprite( )
```

Arguments

Example

```
image = loadImage( "Untied Games/Explosion 01", false )
explosion = createSprite()
setSpriteImage( explosion, image )
setSpriteAnimation( explosion, 0, 69, 60 )
setSpriteLocation( explosion, { gWidth() / 2, gHeight() / 2 } )
setSpriteScale( explosion, { 5, 5 } )
tiles = getSpriteAnimFrameCount( explosion )

for i = 0 to tiles loop
  clear()
  setSpriteAnimFrame( explosion, i )
  drawSprites()
  update()
repeat
```



Associated Commands

`createSprite()`, `deltaTime()`, `drawSprite()`, `drawSprites()`, `removeSprite()`, `updateSprites()`,
`updateSprite()`

COMMAND REFERENCE

createStar()

Purpose

Creates a star to be drawn with `drawShape()`

Description

Creates a star with centre origin to be drawn at the specified x and y location with the specified dimensions and number of points

Syntax

```
shape = createStar( x, y, innerRadius, outerRadius, numPoints )
```

Arguments

shape Handle which stores the newly created shape

x Horizontal screen position in pixels

y Vertical screen position in pixels

width Radius of the inside section of the star

height Radius of the outside section of the star

Example

```
// draw a multicoloured 7-pointed star on the screen
shape_1 = createStar( gwidth() / 2, gheight() / 2, 100, 300, 7 )

points = [
    red,
    orange,
    yellow,
    green,
    blue,
    indigo,
    violet
]

for i = 0 to len( points ) loop
    setVertexColour( shape_1, i, points[i] )
repeat

for i = len( points ) to 0 step -1 loop
    setVertexColour( shape_1, i + 7, points[i - 1] )
repeat
```

```
drawShape( shape_1 )  
update()  
sleep( 3 )
```

Associated Commands

[copyShape\(\)](#), [createBox\(\)](#), [createCircle\(\)](#), [createCurve\(\)](#), [createLine\(\)](#), [createLineStrip\(\)](#), [createPoly\(\)](#), [createTriangle\(\)](#), [deleteShape\(\)](#), [drawShape\(\)](#), [getShapeBounds\(\)](#), [getShapeLocation\(\)](#), [getShapeRotation\(\)](#), [getShapeScale\(\)](#), [getShapeTint\(\)](#), [getVertex\(\)](#), [getVertexColour\(\)](#), [getVertexLineColour\(\)](#), [getVertexLineThickness\(\)](#), [joinShapes\(\)](#), [moveShape\(\)](#), [numVerts\(\)](#), [rotateShape\(\)](#), [scaleShape\(\)](#), [setShapeColour\(\)](#), [setShapeLineStyle\(\)](#), [setShapeRotation\(\)](#), [setShapeScale\(\)](#), [setShapeScaleModeLocal\(\)](#), [setShapeTint\(\)](#), [setVertex\(\)](#), [setVertexColour\(\)](#), [setVertexLineStyle\(\)](#)

COMMAND REFERENCE

createTriangle()

Purpose

Creates a triangle to be drawn with drawShape()

Description

Creates a triangle with centre origin to be drawn at the specified x and y locations

Syntax

```
shape = createTriangle( x1, y1, x2, y2, x3, y3 )
```

Arguments

shape Handle which stores the newly created shape

x1 Horizontal screen position in pixels of the first point

y1 Vertical screen position in pixels of the first point

x2 Horizontal screen position in pixels of the second point

y2 Vertical screen position in pixels of the second point

x3 Horizontal screen position in pixels of the third point

y3 Vertical screen position in pixels of the third point

Example

```
// draw a multicoloured triangle on the screen
x = 0
y = 1

points = [
  [ gwidth() / 3, gheight() / 3 ],
  [ gwidth() / 3 + gwidth() / 3, gheight() / 3 ],
  [ gwidth() / 2, gheight() - gheight() / 3 ]
]

shape_1 = createTriangle( points[0][x], points[0][y], points[1][x], points[1][y], points[2][x], points[2][y] )

setVertexColour( shape_1, 0, red )
setVertexColour( shape_1, 1, green )
setVertexColour( shape_1, 2, blue )

drawShape( shape_1 )
update()
sleep( 3 )
```



Associated Commands

`copyShape()`, `createBox()`, `createCircle()`, `createCurve()`, `createLine()`, `createLineStrip()`, `createPoly()`, `createStar()`, `deleteShape()`, `drawShape()`, `getShapeBounds()`, `getShapeLocation()`, `getShapeRotation()`, `getShapeScale()`, `getShapeTint()`, `getVertex()`, `getVertexColour()`, `getVertexLineColour()`, `getVertexLineThickness()`, `joinShapes()`, `moveShape()`, `numVerts()`, `rotateShape()`, `scaleShape()`, `setShapeColour()`, `setShapeLineStyle()`, `setShapeRotation()`, `setShapeScale()`, `setShapeScaleModeLocal()`, `setShapeTint()`, `setVertex()`, `setVertexColour()`, `setVertexLineStyle()`

COMMAND REFERENCE

deleteShape()

Purpose

Delete a shape drawn using `drawShape()`

Description

Fully removes all traces of the supplied shape. This also renders the handle assigned to the shape as void

Syntax

```
deleteShape( shape )
```

Arguments

shape Handle which stores the shape to delete

Example

```
w = gwidth()
h = gheight()
radius = 100

shape1 = createCircle( w / 3, h / 2, radius, 360 )
shape2 = createCircle( w - w / 3, h / 2, radius, 360 )
shape3 = 0

join = false
dist = 0

loop
  clear( grey )
  j = controls( 0 )

  if !join then
    shape1Location = getShapeLocation( shape1 )
    shape2Location = getShapeLocation( shape2 )
    dist = distance( shape1Location, shape2Location )
  endif

  if dist < radius * 2 and !join then
    shape3 = joinShapes( shape1, shape2 )
    deleteShape( shape1 )
    deleteShape( shape2 )
    join = true
  endif
endif
```

```
if join then
    moveShape( shape3, { j.lx, -j.ly } * 6 )
    drawShape( shape3 )
else
    moveShape( shape1, { j.lx, -j.ly } * 6 )
    drawShape( shape1 )
    drawShape( shape2 )
endif

update()
repeat
```

Associated Commands

copyShape(), createBox(), createCircle(), createCurve(), createLine(), createLineStrip(), createPoly(), createStar(), createTriangle(), drawShape(), getShapeBounds(), getShapeLocation(), getShapeRotation(), getShapeScale(), getShapeTint(), getVertex(), getVertexColour(), getVertexLineColour(), getVertexLineThickness(), joinShapes(), moveShape(), numVerts(), rotateShape(), scaleShape(), setShapeColour(), setShapeLineStyle(), setShapeRotation(), setShapeScale(), setShapeScaleModeLocal(), setShapeTint(), setVertex(), setVertexColour(), setVertexLineStyle()

COMMAND REFERENCE

deltaTime()

Purpose

Get the time difference between frames

Description

Time difference between the current frame and the previous frame, in seconds

Syntax

```
deltatime = deltaTime( )
```

Arguments

deltatime time difference between the current frame and the previous frame, in seconds

Example

```
image = loadImage( "Untied Games/Enemy A", false )
enemy = []
for i = 0 to 4 loop
    enemy[i] = createSprite()
    setSpriteImage( enemy[i], image )
    setSpriteAnimation( enemy[i], 0, 4, 20 )
    setSpriteLocation( enemy[i], { ( i % 2 ) * 400 + 400, int( i / 2 ) * 300 + 200 } )
    setSpriteScale( enemy[i], { 4, 4 } )
repeat
loop
    clear()
    // updateSprite(enemy[0])           not updated
    updateSprite( enemy[1] )           // normal speed
    updateSprite( enemy[2], deltaTime() / 2 ) // half speed
    updateSprite( enemy[3], deltaTime() * 2 ) // double speed
    drawSprites()
    update()
repeat
```



Associated Commands

`createSprite()`, `drawSprite()`, `drawSprites()`, `removeSprite()`, `updateSprites()`, `updateSprite()`

COMMAND REFERENCE

detectMapCollision()

Purpose

Used to detect whether a given sprite has collided with map collision boxes

Description

Receives a sprite handle and returns either true (collided) or false (not collided) if the sprite has collided with a map collision box data.

Syntax

```
collide = detectMapCollision( sprite )
```

Arguments

collide Returned Boolean value to indicate collision. Returns true (1) if collided, false (0) if not collided.

sprite Handle of the sprite being checked.

Example

```
// To view this map demo, please load the project "Map Collision" from FUZE Programs.
// Maps must be stored in the project you wish to load them into.

loadMap( "map1" )
img = loadImage( "Untied Games/Bat and Ball ball" )

plr = [
  .spr = createSprite(),
  .vel = {},
  .col = white
]

setSpriteImage( plr.spr, img )
setSpriteScale( plr.spr, { 1, 1 } )
setSpriteCamera( 0, 0, 2 )

loop
  centreSpriteCamera( 0, 0 )
  clear()
  updateSprites()

  c = controls( 0 )

  plr.vel += { c.lx, -c.ly } * 80
  plr.vel *= 0.87

  setSpriteSpeed( plr.spr, plr.vel )
  setSpriteColour( plr.spr, plr.col )

  drawMapLayer( 0 )
  drawMapLayer( 1 )

  if detectMapCollision( plr.spr ) then
    plr.col = red
  else
```

```
   plr.col = white
endif

drawSprites()

printAt( 0, 0, "Move the ball by using the left control stick" )
printAt( 0, 2, "Ball will appear red when colliding with map collision boxes, white when not colliding" )

update()
repeat
```

Associated Commands

COMMAND REFERENCE

detectSpriteCollision()

Purpose

Detect if two sprites have collided

Description

Returns true if there is a collision between the two sprites, false if not

Syntax

```
result = detectSpriteCollision( spriteA, spriteB )
```

Arguments

spriteA handle of first sprite

spriteB handle of second sprite

result true if *spriteA* and *spriteB* have collided

Example

```
image = loadImage( "Untied Games/Enemy small top C", false )
ship = []
for i = 0 to 2 loop
    ship[i] = createSprite()
    setSpriteImage( ship[i], image )
    setSpriteScale( ship[i], { 5, 5 } )
    setSpriteCollisionShape( ship[i], SHAPE_TRIANGLE, 25, 25, 180 )
    ship[i].show_collision_shape = true
repeat

setSpriteRotation( ship[0], 270 )
setSpriteSpeed( ship[0], { 240, 0 } )
setSpriteSpeed( ship[1], { 0, 120 } )
setSpriteColour( ship[1], { 0, 0, 1, 1 } )
setSpriteLocation( ship[0], { 0, gHeight() / 2 } )
setSpriteLocation( ship[1], { gWidth() / 2, 0 } )

collide = false
while !collide loop
    clear()
    updateSprites()
    drawSprites()
    update()
    collide = detectSpriteCollision( ship[0], ship[1] )
repeat
```



Associated Commands

`collideSprites()`, `setSpriteCollisionShape()`

COMMAND REFERENCE

drawImage()

Purpose

Draw a previously loaded image file

Description

Draws part or all of an image file at the specified location on the screen

Syntax

```
drawImage( handle, x, y )  
drawImage( handle, x, y, scale )  
drawImage( handle, { sourceX, sourceY, sourceW, sourceH }, { x, y, width, height } )
```

Arguments

handle Variable which stores the desired image file

sourceX Horizontal pixel coordinate in the source image from which to begin drawing

sourceY Vertical pixel coordinate in the source image from which to begin drawing

sourceW Width (in pixels) of the source image to draw

sourceH Height (in pixels) of the source image to draw

x Desired on-screen horizontal axis location

y Desired on-screen vertical axis location

scale Amount by which image should be scaled

height Desired on-screen width in pixels

type The type of the image e.g. image_rgb for 24bpp

Example

```
roll = 0  
clear()  
image = loadImage( "Colin Brown/Dice", false )  
size = tileSize( image, 0 )  
for i = 1 to 10 loop  
  clear()  
  roll = random(6) + 1  
  x = size.x - ( size.x * ( roll % 2 ) )  
  y = size.y * ( ceil( roll / 2 ) - 1 )  
  drawImage( image, { x, y, size.x, size.y }, { 0, 0, size.x, size.y } )
```

```
update()  
sleep( 0.3 )  
repeat  
printAt( 0, 15, "You rolled a ", roll )  
update()  
sleep( 3 )
```



Associated Commands

`clear()`, `createImage()`, `drawImageEx()`, `drawQuad()`, `drawSheet()`, `loadImage()`, `update()`, `uploadImage()`

COMMAND REFERENCE

drawImageEx()

Purpose

Draw a previously loaded image file (extended)

Description

Draws an image file at the specified location on the screen. The image can be scaled rotated and tinted

Syntax

```
drawImageEx( handle, location, rotation, scale, tint, origin )
```

Arguments

handle variable which stores the desired image file

location vector screen position to start drawing the image { x, y }

rotation angle to rotate image in default units

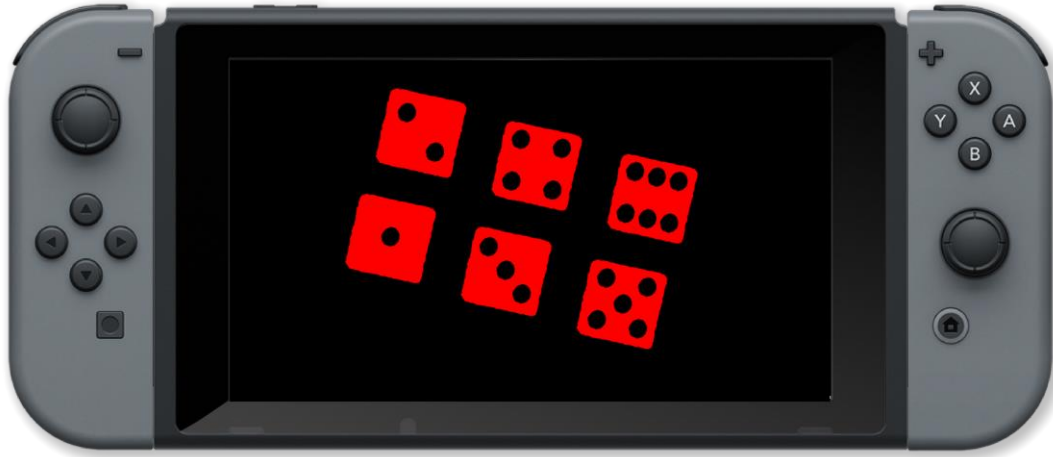
scale vector containing the horizontal and vertical scale factors { x, y }

tint colour name or RGB values { red, green, blue, opacity } between 0 and 1

origin origin point of the screen(default is { 0, 0 } which is the top left)

Example

```
image = loadImage( "Colin Brown/Dice", false )
location = { gWidth() / 2, gHeight() / 2 }
rotation = 0
scale = { 0.5, 0.5 }
tint = red
origin = { 0, 0 }
loop
  clear()
  rotation = rotation + 1
  drawImageEx( image, location, rotation, scale, tint, origin )
  update()
repeat
```



Associated Commands

`clear()`, `createImage()`, `drawImage()`, `drawQuad()`, `drawSheet()`, `loadImage()`, `update()`, `uploadImage()`

COMMAND REFERENCE

drawMap()

Purpose

Draw a tile map

Description

Draw a tile map previously loaded with loadMap()

Syntax

```
drawMap( )
```

Arguments

Example

```
// To view this map demo, please load the project "drawMap() Demo" from FUZE Programs.  
// Maps must be stored in the project you wish to load them into.  
  
maps = [  
    "map1",  
    "map2"  
]  
  
m = 0  
press = false  
  
loadMap( maps[m] )  
setSpriteCamera( 0, 0, 2 )  
  
loop  
    centreSpriteCamera( 0, 0 )  
    clear()  
  
    c = controls( 0 )  
  
    if !c.a then  
        press = false  
    endIf  
  
    drawMap()  
  
    if c.a and !press then  
        press = true  
        unloadMap()  
        m += 1  
        if m >= 2 then  
            m = 0  
        endIf  
        loadMap( maps[m] )  
    endIf  
endLoop
```

```
endif  
  
printAt( 0, 0, "Press A button to swap between maps" )  
printAt( 0, 2, "Currently viewing: " + maps[m] )  
  
update()  
repeat
```

Associated Commands

`drawMapLayer()`, `loadMap()`, `unloadMap()`

COMMAND REFERENCE

drawMapLayer()

Purpose

Draw a tile map layer

Description

Draw a tile map layer previously loaded with loadmap

Syntax

```
drawMapLayer( layer )
```

Arguments

layer layer number of the map to draw (zero based)

Example

```
// To view this map demo, please load the project "Map Commands Demo" from FUZE Programs.
// Maps must be stored in the project you wish to load them into.

maps = [
    "map1",
    "map2"
]

m = 0
layer = false

press = [
    .a = false,
    .up = false
]

loadMap( maps[m] )
setSpriteCamera( 0, 0, 2 )

loop
    centerSpriteCamera( 0, 0 )
    clear()

    c = controls( 0 )

    if !c.a then
        press.a = false
    endIf
    if !c.up then
        press.up = false
    endIf
```

```

drawMapLayer( 0 )

if layer then
    drawMapLayer( 1 )
endif

if c.up and !press.up then
    press.up = true
    layer = !layer
endif

if c.a and !press.a then
    press.a = true
    unloadMap()
    m += 1
    if m >= 2 then
        m = 0
    endif
    loadMap( maps[m] )
endif

printAt( 0, 0, "Press A button to swap between maps" )
printAt( 0, 2, "Currently viewing: " + maps[m] )
printAt( 0, 4, "Press Up directional button to toggle additional layers" )

update()
repeat

```

Associated Commands

`drawMap()`, `loadMap()`, `unloadMap()`

COMMAND REFERENCE

drawQuad()

Purpose

Draw an image or portion of an image on an area of the screen

Description

Draw the specified part of an image to the specified part of the screen and with the specified tint

Syntax

```
drawQuad( handle, { sourcecx, sourcecy, sourcew, sourceh }, points, tint )
```

Arguments

handle Variable which stores the desired image file

sourcecx Horizontal pixel coordinate in the source image from which to begin drawing

sourcecy Vertical pixel coordinate in the source image from which to begin drawing

sourcew Width (in pixels) of the source image to draw

sourceh Height (in pixels) of the source image to draw

points Array of points of the target area of the screen (top left, top right, bottom right, bottom left)

tint colour name or RGB values { red, green, blue, opacity } between 0 and 1

Example

```
// draw image in centre with 100 pixel border and green tint
image = loadImage( "Ansimuz/CyberpunkStreetLayer2", false )
size = imageSize( image )
points = []
points[0] = { 100, 100 }
points[1] = { gWidth() - 100, 100 }
points[2] = { gWidth() - 100, gHeight() - 100 }
points[3] = { 100, gHeight() - 100 }
drawQuad( image, { 0, 0, size.x, size.y }, points, green )
update()
sleep( 3 )
```



Associated Commands

`clear()`, `createImage()`, `drawImage()`, `drawImageEx()`, `drawSheet()`, `loadImage()`, `update()`, `uploadImage()`

COMMAND REFERENCE

drawShape()

Purpose

Draws a shape created with the advanced shape functions

Description

Draws the shape stored in the supplied handle

Syntax

```
drawShape( shape )
```

Arguments

shape Handle which stores the shape to draw

Example

```
// draw a multicoloured rectangle on the screen
box1 = createBox( gwidth() / 2, gheight() / 2, gwidth(), gheight() )

setVertexColour( box1, 0, bisque )
setVertexColour( box1, 1, cyan )
setVertexColour( box1, 2, fuzeblue )
setVertexColour( box1, 3, fuzepink )

drawShape( box1 )
update()
sleep( 3 )
```



Associated Commands

copyShape(), createBox(), createCircle(), createCurve(), createLine(), createLineStrip(),
createPoly(), createStar(), createTriangle(), deleteShape(), getShapeBounds(),
getShapeLocation(), getShapeRotation(), getShapeScale(), getShapeTint(), getVertex(),
getVertexColour(), getVertexLineColour(), getVertexLineThickness(), joinShapes(), moveShape(),
numVerts(), rotateShape(), scaleShape(), setShapeColour(), setShapeLineStyle(),
setShapeRotation(), setShapeScale(), setShapeScaleModeLocal(), setShapeTint(), setVertex(),
setVertexColour(), setVertexLineStyle()

COMMAND REFERENCE

drawSheet()

Purpose

Draw a tile from a tiled image

Description

Draw the specified tile number from a tiled image at the specified location and size

Syntax

```
drawSheet( handle, tileno, { xpos, ypos, width, height } )
```

Arguments

handle Variable which stores the desired image file

tileno Tile number to display (zero based)

xpos Horizontal screen position in pixels

ypos Vertical screen position in pixels

width Desired on-screen width in pixels

height Desired on-screen height in pixels

Example

```
image = loadImage( "Untied Games/Enemy A", false )
pos = { gwidth() / 2, gheight() / 2 }
loop
  for i = 0 to 4 loop
    clear()
    drawImage( image, 0, 0, 1 )
    drawSheet( image, i, { pos.x - 100, pos.y - 100, 200, 200 } )
    update()
    sleep( 0.1 )
  repeat
repeat
```



Associated Commands

`clear()`, `createImage()`, `drawImage()`, `drawImageEx()`, `drawQuad()`, `loadImage()`, `update()`, `uploadImage()`

COMMAND REFERENCE

drawSprite()

Purpose

Draw the specified sprite

Description

Draw only the specified sprite in the current position, orientation and colour

Syntax

```
drawSprite( sprite )
```

Arguments

sprite handle of the sprite

Example

```
image = loadImage( "Untied Games/Enemy A", false )
enemy = []
for i = 0 to 4 loop
    enemy[i] = createSprite()
    setSpriteImage( enemy[i], image )
    setSpriteAnimation( enemy[i], 0, 4, 20 )
    setSpriteLocation( enemy[i], { ( i % 2 ) * 400 + 400, int(i / 2) * 300 + 200 } )
    setSpriteScale( enemy[i], { 4, 4 } )
repeat

loop
    clear()
    printAt( 0, 0, "Press buttons X, A, Y and B to draw sprites" )
    updateSprite( enemy[1] )
    updateSprite( enemy[2], deltaTime() / 2 )
    updateSprite( enemy[3], deltaTime() * 2 )
    c = controls(0)
    if c.x then
        drawSprite( enemy[0] )
    endif
    if c.a then
        drawSprite( enemy[1] )
    endif
    if c.b then
        drawSprite( enemy[2] )
    endif
    if c.y then
        drawSprite( enemy[3] )
    endif
    update()
repeat
```



Associated Commands

`createSprite()`, `deltaTime()`, `drawSprite()`, `drawSprites()`, `removeSprite()`, `updateSprites()`, `updateSprite()`

COMMAND REFERENCE

drawSprites()

Purpose

Draw all sprites

Description

Draw all sprites in their current position, orientation and colour

Syntax

```
drawSprites( )
```

Arguments

Example

```
radians( 1 )
image = loadImage( "Untied Games/Enemy small top C", false )
ship = createSprite( )
setSpriteImage( ship, image )
lastpos = { gWidth() / 2, gHeight() / 2 }
setSpriteLocation( ship, lastpos )
setSpriteScale( ship, { 4, 4 } )

loop
  clear()
  c = controls( 0 )
  printAt( 0, 0, "Use left control stick to control sprite" )
  setSpriteSpeed( ship, { 480 * c.lx, -480 * c.ly } )
  curpos = getSpriteLocation( ship )
  if curpos != lastpos then
    setSpriteRotation( ship, -pi / 2 + atan2( curpos.y - lastpos.y, curpos.x - lastpos.x ) )
    lastpos = curpos
  endif
  updateSprites()
  drawSprites()
  update()
repeat
```



Associated Commands

`createSprite()`, `deltaTime()`, `drawSprite()`, `drawSprites()`, `removeSprite()`, `updateSprites()`, `updateSprite()`

COMMAND REFERENCE

freeImage()

Purpose

Dispose of an image that is no longer required

Description

Free up the memory allocated to an image so that it can be used again

Syntax

```
freeImage( handle )
```

Arguments

handle Variable which stores the desired image file

Example

```
dice = loadImage( "Colin Brown/Dice", false )
freeImage( dice )
drawImage( dice, 0, 0 ) // This will cause an error
update()
sleep( 3 )
```

Associated Commands

`createImage()`, `loadImage()`, `uploadImage()`

COMMAND REFERENCE

getShapeBounds()

Purpose

Find the boundaries of a supplied shape

Description

Returns the boundaries (edges) of a shape drawn with `drawShape()`

Syntax

```
boundaries = getShapeBounds( shape )
```

Arguments

shape Handle which stores the shape in question

boundaries Vector describing the boundaries of the shape

Example

Associated Commands

`copyShape()`, `createBox()`, `createCircle()`, `createCurve()`, `createLine()`, `createLineStrip()`, `createPoly()`, `createStar()`, `createTriangle()`, `deleteShape()`, `drawShape()`, `getShapeLocation()`, `getShapeRotation()`, `getShapeScale()`, `getShapeTint()`, `getVertex()`, `getVertexColour()`, `getVertexLineColour()`, `getVertexLineThickness()`, `joinShapes()`, `moveShape()`, `numVerts()`, `rotateShape()`, `scaleShape()`, `setShapeColour()`, `setShapeLineStyle()`, `setShapeRotation()`, `setShapeScale()`, `setShapeScaleModeLocal()`, `setShapeTint()`, `setVertex()`, `setVertexColour()`, `setVertexLineStyle()`

COMMAND REFERENCE

getShapeLocation()

Purpose

Find the pixel co-ordinate location of a given shape

Description

Returns the x and y screen positions (as a vector) in pixels of a supplied shape

Syntax

```
location = getShapeLocation( shape )
```

Arguments

location Handle to store the returned position vector

shape Handle which stores the shape in question

Example

```
shape = createCircle( gwidth() / 2, gheight() / 2, 200, 360 )  
  
loop  
  clear( grey )  
  pos = getShapeLocation( shape )  
  print( pos )  
  drawShape( shape )  
  update()  
repeat
```

Associated Commands

copyShape(), createBox(), createCircle(), createCurve(), createLine(), createLineStrip(), createPoly(), createStar(), createTriangle(), deleteShape(), drawShape(), getShapeBounds(), getShapeRotation(), getShapeScale(), getShapeTint(), getVertex(), getVertexColour(), getVertexLineColour(), getVertexLineThickness(), joinShapes(), moveShape(), numVerts(), rotateShape(), scaleShape(), setShapeColour(), setShapeLineStyle(), setShapeRotation(), setShapeScale(), setShapeScaleModeLocal(), setShapeTint(), setVertex(), setVertexColour(), setVertexLineStyle()

COMMAND REFERENCE

getShapeRotation()

Purpose

Find the rotation of a shape in degrees or radians

Description

Returns the amount of rotation applied to a shape drawn with `drawShape()`

Syntax

```
rotation = getShapeRotation( shape )
```

Arguments

shape Handle which stores the shape in question

rotation Handle which stores the amount of rotation applied to the supplied shape

Example

Associated Commands

`copyShape()`, `createBox()`, `createCircle()`, `createCurve()`, `createLine()`, `createLineStrip()`, `createPoly()`, `createStar()`, `createTriangle()`, `deleteShape()`, `drawShape()`, `getShapeBounds()`, `getShapeLocation()`, `getShapeScale()`, `getShapeTint()`, `getVertex()`, `getVertexColour()`, `getVertexLineColour()`, `getVertexLineThickness()`, `joinShapes()`, `moveShape()`, `numVerts()`, `rotateShape()`, `scaleShape()`, `setShapeColour()`, `setShapeLineStyle()`, `setShapeRotation()`, `setShapeScale()`, `setShapeScaleModeLocal()`, `setShapeTint()`, `setVertex()`, `setVertexColour()`, `setVertexLineStyle()`

COMMAND REFERENCE

getShapeScale()

Purpose

Find the current scale multiplier of a shape drawn with `drawShape()`

Description

Returns a vector describing the current scale multiplier of a supplied shape

Syntax

```
scale = getShapeScale( shape )
```

Arguments

shape Handle which stores the shape in question

scale Vector which describes the x and y scale multiplier of the shape

Example

Associated Commands

`copyShape()`, `createBox()`, `createCircle()`, `createCurve()`, `createLine()`, `createLineStrip()`, `createPoly()`, `createStar()`, `createTriangle()`, `deleteShape()`, `drawShape()`, `getShapeBounds()`, `getShapeLocation()`, `getShapeRotation()`, `getShapeTint()`, `getVertex()`, `getVertexColour()`, `getVertexLineColour()`, `getVertexLineThickness()`, `joinShapes()`, `moveShape()`, `numVerts()`, `rotateShape()`, `scaleShape()`, `setShapeColour()`, `setShapeLineStyle()`, `setShapeRotation()`, `setShapeScale()`, `setShapeScaleModeLocal()`, `setShapeTint()`, `setVertex()`, `setVertexColour()`, `setVertexLineStyle()`

COMMAND REFERENCE

getShapeTint()

Purpose

Find the tint (colour) of a shape

Description

Returns the colour vector (RGBA) of a shape drawn with `drawShape()`

Syntax

```
tint = getShapeTint( shape )
```

Arguments

shape Handle which stores the shape in question

tint RGBA vector which describes the colour of the supplied shape

Example

Associated Commands

`copyShape()`, `createBox()`, `createCircle()`, `createCurve()`, `createLine()`, `createLineStrip()`, `createPoly()`, `createStar()`, `createTriangle()`, `deleteShape()`, `drawShape()`, `getShapeBounds()`, `getShapeLocation()`, `getShapeRotation()`, `getShapeScale()`, `getVertex()`, `getVertexColour()`, `getVertexLineColour()`, `getVertexLineThickness()`, `joinShapes()`, `moveShape()`, `numVerts()`, `rotateShape()`, `scaleShape()`, `setShapeColour()`, `setShapeLineStyle()`, `setShapeRotation()`, `setShapeScale()`, `setShapeScaleModeLocal()`, `setShapeTint()`, `setVertex()`, `setVertexColour()`, `setVertexLineStyle()`

COMMAND REFERENCE

getSpriteAnimFrame()

Purpose

Get the current frame in an animated sprite

Description

Find the number of the current frame in an animated sprite

Syntax

```
frame = getSpriteAnimFrame( sprite )
```

Arguments

sprite handle of the sprite

frame number of the current animation frame in the sprite

Example

```
image = loadImage( "Untied Games/Enemy A", false )
enemy = createSprite()
setSpriteImage( enemy, image )
setSpriteAnimation( enemy, 0, 4, 2 )
lastpos = { gWidth() / 2, gHeight() / 2 }
setSpriteLocation( enemy, lastpos )
setSpriteScale( enemy, { 8, 8 } )

loop
  clear()
  frame = getSpriteAnimFrame( enemy )
  printAt( 0, 0, "Frame ", int( frame ) )
  updateSprites()
  drawSprites()
  update()
repeat
```

Associated Commands

getSpriteAnimFrameCount(), getSpriteAnimSpeed(), setSpriteAnimation(), setSpriteAnimFrame(), setSpriteAnimSpeed()

COMMAND REFERENCE

getSpriteAnimFrameCount()

Purpose

Find the number of frames in an animated sprite

Description

Syntax

```
count = getSpriteAnimFrameCount( sprite )
```

Arguments

sprite handle of the sprite

count number of animation frames in the sprite

Example

```
image = loadImage( "Untied Games/Explosion 01", false )
explosion = createsprite()
setSpriteImage( explosion, image )
setSpriteAnimation( explosion, 0, 69, 60 )
setSpriteLocation( explosion, { gWidth() / 2, gHeight() / 2 } )
setSpriteScale( explosion, { 5, 5 } )
tiles = getSpriteAnimFrameCount( explosion )

for i = 0 to tiles loop
  clear()
  setSpriteAnimFrame( explosion, i )
  drawSprites()
  update()
repeat
```




Associated Commands

`getSpriteAnimFrame()`, `getSpriteAnimSpeed()`, `setSpriteAnimation()`, `setSpriteAnimFrame()`,
`setSpriteAnimSpeed()`

COMMAND REFERENCE

getSpriteAnimSpeed()

Purpose

Find the speed of a sprites animation

Description

Get the current speed of animation of a sprite

Syntax

```
speed = getSpriteAnimSpeed( sprite )  
speed = sprite.anim_speed
```

Arguments

sprite The handle of the sprite

speed The speed of the animation

Example

```
image = loadImage( "Untied Games/Enemy A", false )  
enemy = createSprite()  
setSpriteImage( enemy, image )  
speed = 0  
maxspeed = 50  
setSpriteAnimation( enemy, 0, 4, speed )  
lastpos = { gWidth() / 2, gHeight() / 2 }  
setSpriteLocation( enemy, lastpos )  
setSpriteScale( enemy, { 8, 8 } )  
  
loop  
  clear()  
  printAt( 0, 0, "Use left joystick to control animation speed" )  
  c = controls( 0 )  
  speed = getSpriteAnimSpeed( enemy )  
  if abs( speed + c.lx ) < maxspeed then  
    setSpriteAnimSpeed( enemy, speed + c.lx )  
  endif  
  updateSprites()  
  drawSprites()  
  update()  
repeat
```



Associated Commands

`getSpriteAnimFrame()`, `getSpriteAnimFrameCount()`, `setSpriteAnimation()`,
`setSpriteAnimFrame()`, `setSpriteAnimSpeed()`

COMMAND REFERENCE

getSpriteCamera()

Purpose

Get the sprite camera position

Description

Find the current sprite camera position. The initial position is (0, 0, 1)

Syntax

```
camera = getSpriteCamera( )
```

Arguments

camera A vector containing the current sprite camera position { x, y, z }

Example

```
image = loadImage( "Untied Games/Enemy A", false )
enemy = []
for i = 0 to 4 loop
    enemy[i] = createsprite()
    setSpriteImage( enemy[i], image )
    setSpriteAnimation( enemy[i], 0, 4, 20 )
    setSpriteLocation( enemy[i], { ( i % 2 ) * 400 + 400, int( i / 2 ) * 300 + 200 } )
    setSpriteScale( enemy[i], { 4, 4 } )
repeat

camera = getSpriteCamera()
rotation = getSpriteCameraRotation()
loop
    clear()
    c = controls( 0 )
    printAt( 0, 0, "Camera position: x = ", camera.x, " y = ", camera.y, " z = ", camera.z, " rotation: ", rotation )
    printAt( 0, 1, "Use left joypad to pan, right joypad to zoom/rotate" )
    if c.up then
        camera.y -= 5
    endif
    if c.down then
        camera.y += 5
    endif
    if c.left then
        camera.x -= 5
    endif
    if c.right then
        camera.x += 5
    endif
    if c.x then
        camera.z += 0.05
    endif
    if c.b then
        camera.z -= 0.05
    endif
    if c.y then
        rotation -= 0.5
    endif
    if c.a then
        rotation += 0.5
    endif
    setSpriteCamera( camera.x, camera.y, camera.z )
    setSpriteCameraRotation( rotation )
    updateSprites()
    drawSprites()
    update()
repeat
```



Associated Commands

`centreSpriteCamera()`, `getSpriteCameraRotation()`, `setSpriteCamera()`, `setSpriteCameraRotation()`

COMMAND REFERENCE

getSpriteCameraRotation()

Purpose

Get the sprite camera rotation angle

Description

Returns the current rotation angle of the sprite camera

Syntax

```
angle = getSpriteCameraRotation( )
```

Arguments

angle camera rotation angle in the default units

Example

```
image = loadImage( "Untied Games/Enemy A", false )
enemy = []
for i = 0 to 4 loop
    enemy[i] = createSprite( )
    setSpriteImage( enemy[i], image )
    setSpriteAnimation( enemy[i], 0, 4, 20 )
    setSpriteLocation( enemy[i], { ( i % 2 ) * 400 + 400, int(i / 2) * 300 + 200 } )
    setSpriteScale( enemy[i], { 4, 4 } )
repeat

camera = getSpriteCamera()
rotation = getSpriteCameraRotation()
loop
    clear()
    c = controls( 0 )
    printAt( 0, 0, "Camera position: x = ", camera.x, " y = ", camera.y, " z = ", camera.z, " rotation: ", rotation )
    printAt( 0, 1, "Use left joypad to pan, right joypad to zoom/rotate" )
    if c.up then
        camera.y -= 5
    endif
    if c.down then
        camera.y += 5
    endif
    if c.left then
        camera.x -= 5
    endif
    if c.right then
        camera.x += 5
    endif
    if c.x then
        camera.z += 0.05
    endif
    if c.b then
        camera.z -= 0.05
    endif
    if c.y then
        rotation -= 0.5
    endif
    if c.a then
        rotation += 0.5
    endif
    setSpriteCamera( camera.x, camera.y, camera.z )
    setSpriteCameraRotation( rotation )
    updateSprites()
    drawSprites()
    update()
repeat
```



Associated Commands

`centreSpriteCamera()`, `getSpriteCamera()`, `setSpriteCamera()`, `setSpriteCameraRotation()`

COMMAND REFERENCE

getSpriteColour()

Purpose

Get the colour values of a sprite

Description

Gets the red, green, blue and alpha (opacity) values for a sprite

Syntax

```
colour = getSpriteColour( sprite )  
  
red = sprite.r; green = sprite.g; blue = sprite.b; alpha = sprite.a
```

Arguments

sprite handle of the created sprite

colour vector containing the colour values { r, g, b, a }

red value of red colour of the sprite

green value of green colour of the sprite

blue value of blue colour of the sprite

alpha value of alpha (opacity) of the sprite

Example

```
image = loadImage( "Untied Games/Enemy small top C", false )  
ship = createSprite()  
setSpriteImage( ship, image )  
lastpos = { gWidth() / 2, gHeight() / 2 }  
setSpriteLocation( ship, lastpos )  
setSpriteScale( ship, { 20, 20 } )  
rv = -0.5  
gv = 0.5  
bv = 0  
  
loop  
  clear()  
  sc = getSpriteColour( ship )  
  if sc.r > 1 or sc.r < 0 then  
    rv = -rv  
  endif  
  if sc.b > 1 or sc.b < 0 then  
    gv = -gv  
  endif  
endIf
```



```
setSpriteColourSpeed( ship, { rv, gv, bv, 0 } )  
updateSprites()  
drawSprites()  
update()  
repeat
```



Associated Commands

`getSpriteColourSpeed()`, `setSpriteColour()`, `setSpriteColourSpeed()`

COMMAND REFERENCE

getSpriteColourSpeed()

Purpose

Get the colour speeds of a sprite

Description

Get the rates of change of the colours of a sprite. These are the amounts that the sprite colours are changed by when updatesprites is called

Syntax

```
colourspeed = getSpriteColourSpeed( sprite )  
  
rspeed = sprite.r_speed; gspeed = sprite.g_speed; bspeed = sprite.b_speed; aspeed = sprite.a_speed;
```

Arguments

handle handle of the created sprite

colourv vector containing the colour speed values { r, g, b, a }

rspeed amount to add to the red colour of the sprite at each updatesprites call

gspeed amount to add to the blue colour of the sprite at each updatesprites call

bspeed amount to add to the green colour of the sprite at each updatesprites call

aspeed amount to add to the opacity of the sprite at each updatesprites call

Example

```
image = loadImage( "Untied Games/Enemy small top C", false )  
ship = createSprite()  
setSpriteImage( ship, image )  
lastpos = { gWidth() / 2, gHeight() / 2 }  
setSpriteLocation( ship, lastpos )  
setSpriteScale( ship, { 20, 20 } )  
setSpriteColourSpeed( ship, { -0.5, 0.5, 0, 0 } )  
  
loop  
  clear()  
  sc = getSpriteColour( ship )  
  cv = getSpriteColourSpeed( ship )  
  if sc.r > 1 or sc.r < 0 then  
    setSpriteColourSpeed( ship, { -cv.r, cv.g, cv.b, 0 } )  
  endif  
  if sc.g > 1 or sc.g < 0 then  
    setSpriteColourSpeed( ship, { cv.r, -cv.g, cv.b, 0 } )  
  endif
```

```
updateSprites()  
drawSprites()  
update()  
sleep( 0.1 )  
repeat
```



Associated Commands

`getSpriteColour()`, `setSpriteColour()`, `setSpriteColourSpeed()`

COMMAND REFERENCE

getSpriteDepth()

Purpose

Get a sprites depth

Description

Gets the visual depth of the sprite. For drawing, sprites will automatically be sorted by their depth from negative (earliest drawing) to positive (latest drawing).

Syntax

```
depth = getSpriteDepth( sprite )
```

```
depth = sprite.depth
```

Arguments

sprite handle of the created sprite

depth visual depth of the sprite

Example

```
image = loadImage( "Untied Games/Enemy small top C", false )
ship = []

for i = 0 to 2 loop
    ship[i] = createSprite()
    setSpriteImage( ship[i], image )
    setSpriteScale( ship[i], { 5, 5 } )
    setSpriteDepth( ship[i], rnd( 10 ) )
repeat

setSpriteRotation( ship[0], 270 )
setSpriteLocation( ship[0], { 0, gHeight() / 2 } )
setSpriteLocation( ship[1], { gWidth() / 2, 0 } )
setSpriteSpeed( ship[0], { 240, 0 } )
setSpriteSpeed( ship[1], { 0, 120 } )
setSpriteColour( ship[0], { 0, 0, 1, 1 } )

while ship[0].x < gwidth() loop
    clear()
    depth0 = getSpriteDepth(ship[0])
    depth1 = getSpriteDepth(ship[1])
    if depth0 < depth1 then
        printAt( 0, 0, "Red ship is on top" )
    endIf
    if depth1 < depth0 then
```

```
    printAt( 0, 0, "Blue ship is on top" )
  endIf
  updateSprites()
  drawSprites()
  update()
repeat
```



Associated Commands

`setSpriteDepth()`

COMMAND REFERENCE

getSpriteImage()

Purpose

Get the image associated with a sprite

Description

Get the original image that was used to create the specified sprite

Syntax

```
image = getSpriteImage( sprite )  
image = sprite.image
```

Arguments

sprite The handle of the sprite

image The handle of the associated image

Example

```
image = loadImage( "Untied Games/Explosion 01", false )  
explosion = createsprite()  
setSpriteImage( explosion, image )  
setSpriteLocation( explosion, { gWidth() / 2, gHeight() / 2 } )  
setSpriteScale( explosion, { 5, 5 } )  
tsize = tileSize( image, 0 )  
spriteImage = getSpriteImage( explosion )  
isize = imageSize( spriteImage )  
tiles = ( isize.x / tsize.x ) * ( isize.y / tsize.y )  
setSpriteAnimation( explosion, 0, tiles - 1, 60 )  
  
for i = 0 to tiles loop  
  clear()  
  updateSprites()  
  drawSprites()  
  update()  
repeat
```



Associated Commands

`getSpriteImageSize()`, `setSpriteImage()`

COMMAND REFERENCE

getSpriteImageSize()

Purpose

Get the size of the image associated with a sprite

Description

Get the size of the original image that was used to create the specified sprite or the tile size if an animation has been set

Syntax

```
size = getSpriteImageSize( sprite )
```

Arguments

sprite The handle of the sprite

size A vector containing the width and height of the original image { x, y } or the tile size

Example

```
image = loadImage( "Untied Games/Explosion 01", false )
explosion = createsprite()
setSpriteImage( explosion, image )
setSpriteLocation( explosion, { gWidth() / 2, gHeight() / 2 } )
setSpriteScale( explosion, { 5, 5 } )
tsize = tileSize( image, 0 )
isize = getSpriteImageSize( explosion )
tiles = ( isize.x / tsize.x ) * ( isize.y / tsize.y )
setSpriteAnimation( explosion, 0, tiles, 60 )

for i = 0 to tiles loop
  clear()
  updateSprites()
  drawSprites()
  update()
repeat
```




Associated Commands

`getSpriteImage()`, `setSpriteImage()`

COMMAND REFERENCE

getSpriteLocation()

Purpose

Get the position of a sprite on the screen

Description

Get the horizontal and vertical position of a sprite

Syntax

```
position = getSpriteLocation( sprite )  
  
xpos = sprite.x; ypos = sprite.y
```

Arguments

sprite The handle of the created sprite

position A vector containing the x and y coordinates of the sprite { x, y }

xpos The horizontal position on the screen in pixels

ypos The vertical position on the screen in pixels

Example

```
radians( true )  
image = loadImage( "Untied Games/Enemy small top C", false )  
ship = createSprite()  
setSpriteImage( ship, image )  
lastpos = { gWidth() / 2, gHeight() / 2 }  
setSpriteLocation( ship, lastpos )  
setSpriteScale( ship, { 4, 4 } )  
  
loop  
  clear()  
  c = controls( 0 )  
  printAt( 0, 0, "Use left joystick to control sprite" )  
  setSpriteSpeed( ship, { 480 * c.lx, -480 * c.ly } )  
  curpos = getSpriteLocation( ship )  
  if curpos != lastpos then  
    setSpriteRotation( ship, -pi / 2 + atan2( curpos.y - lastpos.y, curpos.x - lastpos.x ) )  
    lastpos = curpos  
  endif  
  updateSprites()  
  drawSprites()  
  update()  
repeat
```



Associated Commands

`getSpriteOrigin()`, `setSpriteLocation()`, `setSpriteOrigin()`

COMMAND REFERENCE

getSpriteOrigin()

Purpose

Find the origin point of a sprite

Description

Find the origin point of the specified sprite. Default is the centre (0, 0)

Syntax

```
origin = getSpriteOrigin( sprite )
```

Arguments

sprite handle of the sprite

origin origin point of the sprite { x, y }

Example

```
image = loadImage( "Untied Games/Enemy A", false )
enemy = createSprite()
setSpriteImage( enemy, image )
setSpriteAnimation( enemy, 0, 4, 20 )
setSpriteLocation( enemy, { 0, 0 } )
size = getSpriteSize( enemy )
setSpriteScale( enemy, { 8, 8 } )

loop
  clear()
  origin = getSpriteOrigin( enemy )
  printAt( 20, 10, "Sprite origin x = ", origin.x, " y = ", origin.y )
  printAt( 20, 11, "Press A to move origin to the top left" )
  printAt( 20, 12, "Press B to move origin to the centre" )
  c = controls( 0 )
  if c.a then
    setSpriteOrigin( enemy, { -size.x / 2, -size.y / 2 } )
  endIf
  if c.b then
    setSpriteOrigin( enemy, { 0, 0 } )
  endIf
  updateSprites()
  drawSprites()
  update()
repeat
```



Associated Commands

`getSpriteLocation()`, `setSpriteLocation()`, `setSpriteOrigin()`

COMMAND REFERENCE

getSpriteRotation()

Purpose

Get the rotation angle of a sprite

Description

Get the current rotation angle of a sprite in the default angle units

Syntax

```
angle = getSpriteRotation( sprite )  
angle = sprite.rotation
```

Arguments

sprite handle of the created sprite

rotation angle in the default units

Example

```
image = loadImage( "Untied Games/Enemy small top C", false )  
ship = createSprite()  
setSpriteImage( ship, image )  
lastpos = { gWidth() / 2, gHeight() / 2 }  
setSpriteLocation( ship, lastpos )  
setSpriteScale( ship, { 10, 10 } )  
setSpriteRotationSpeed( ship, 60 )  
  
loop  
  clear()  
  angle = getSpriteRotation( ship )  
  if angle > 360 then  
    setSpriteRotationSpeed( ship, -60 )  
  endif  
  if angle < 0 then  
    setSpriteRotationSpeed( ship, 60 )  
  endif  
  updateSprites()  
  drawSprites()  
  update()  
repeat
```



Associated Commands

`getSpriteRotationSpeed()`, `setSpriteRotation()`, `setSpriteRotationSpeed()`

COMMAND REFERENCE

getSpriteRotationSpeed()

Purpose

Get a sprites rotation speed

Description

This is the amount that the sprite's rotation angle is changed by when updatesprites is called

Syntax

```
rotatespeed = getSpriteRotationSpeed( sprite )  
rotatespeed = sprite.rotation_speed
```

Arguments

sprite handle of the created sprite

rotatespeed amount to add to the rotation angle of the sprite at each updatesprites call

Example

```
image = loadImage( "Untied Games/Enemy small top C", false )  
ship = createSprite()  
setSpriteImage( ship, image )  
lastpos = { gWidth() / 2, gHeight() / 2 }  
setSpriteLocation( ship, lastpos )  
setSpriteScale( ship, { 10, 10 } )  
maxrs = 240 // max rotation speed  
accr = 1    // accelaration  
  
loop  
  clear()  
  rs = getSpriteRotationSpeed( ship )  
  if abs( rs ) > maxrs then  
    accr = -accr  
  endif  
  setSpriteRotationSpeed( ship, rs + accr )  
  updateSprites()  
  drawSprites()  
  update()  
repeat
```




Associated Commands

`getSpriteRotation()`, `setSpriteRotation()`, `setSpriteRotationSpeed()`

COMMAND REFERENCE

getSpriteScale()

Purpose

Get a sprite's scale factor

Description

This is the amount that the sprite's scale factor is changed by when `updatesprites` is called

Syntax

```
scale = getSpriteScale( sprite )  
  
xscale = sprite.xscale; yscale = sprite.yscale
```

Arguments

handle handle of the created sprite

scale vector containing the scale factor values { x, y }

xscale amount to add to the horizontal scale of the sprite at each `updatesprites` call

yscale amount to add to the vertical scale of the sprite at each `updatesprites` call

Example

```
image = loadImage( "Untied Games/Enemy small top C", false )  
ship = createSprite()  
setSpriteImage( ship, image )  
lastpos = { gWidth() / 2, gHeight() / 2 }  
setSpriteLocation( ship, lastpos )  
setSpriteScale( ship, { 10, 10 } )  
minsf = 10 // min scale factor  
maxsf = 20 // max scale factor  
sv = 5  
  
loop  
  clear()  
  sf = getSpriteScale( ship )  
  if sv > 0 and sf.x > maxsf then  
    sv = -sv  
  endif  
  if sv < 0 and sf.x < minsf then  
    sv = -sv  
  endif  
  setSpriteScaleSpeed(ship, { sv, sv } )  
  updateSprites()  
  drawSprites()
```

```
update()  
repeat
```



Associated Commands

`getSpriteScaleSpeed()`, `getSpriteSize()`, `setSpriteScale()`, `setSpriteScaleSpeed()`

COMMAND REFERENCE

getSpriteScaleSpeed()

Purpose

Get a sprites scale speed

Description

This is the amount by which the sprite's scale factor is changed by when updatesprites is called

Syntax

```
scalespeed = getSpriteScaleSpeed( sprite )  
xscalespeed = sprite.xscale_speed; yscalespeed = sprite.yscale_speed
```

Arguments

sprite handle of the created sprite

scalespeed vector containing the scale speed values { x, y }

xscalespeed amount to add to the horizontal scale of the sprite at each updatesprites call

yscalespeed amount to add to the vertical scale of the sprite at each updatesprites call

Example

```
image = loadImage( "Untied Games/Enemy small top C", false )  
ship = createSprite()  
setSpriteImage( ship, image )  
lastpos = { gWidth() / 2, gHeight() / 2 }  
setSpriteLocation( ship, lastpos )  
setSpriteScale( ship, { 10, 10 } )  
maxsv = 30 // max scale speed  
accsv = 1 // accelaration  
  
loop  
  clear()  
  sv = getSpriteScaleSpeed( ship )  
  if abs(sv.x) > maxsv then  
    accsv = -accsv  
  endif  
  setSpriteScaleSpeed( ship, { sv.x + accsv, sv.y + accsv } )  
  updateSprites()  
  drawSprites()  
  update()  
repeat
```



Associated Commands

`getSpriteScale()`, `getSpriteSize()`, `setSpriteScale()`, `setSpriteScaleSpeed()`

COMMAND REFERENCE

getSpriteSize()

Purpose

Find the size of a sprite

Description

Find the size of a sprite at the current scale factor. If the sprite was created from a tiled image then the tile size is returned at the current scale factor

Syntax

```
size = getSpriteSize( sprite )
```

Arguments

sprite handle of the sprite

size vector containing the width and height of the image (or tile) at the current scale factor { x, y }

Example

```
image = loadImage( "Untied Games/Player ships", false )
ship = createSprite()
setSpriteImage( ship, image )
setSpriteAnimation( ship, 0, 0, 0 )
setSpriteLocation( ship, { gWidth() / 2, gHeight() / 2 } )
setSpriteScale( ship, { 5, 5 } )

loop
  clear()
  size = getSpriteSize( ship )
  printAt( 0, 0, "Sprite Width: " + size.x )
  printAt( 0, 1, "Sprite Height: " + size.y )
  drawSprites()
  update()
repeat
```



Associated Commands

`getSpriteScale()`, `getSpriteScaleSpeed()`, `setSpriteScale()`, `setSpriteScaleSpeed()`

COMMAND REFERENCE

getSpriteSpeed()

Purpose

Set a sprites speed

Description

Set a sprites horizontal and vertical speed. This is the amount that the sprite is moved by in each axis when updatesprites is called

Syntax

```
speed = getSpriteSpeed( sprite )  
  
xspeed = sprite.x_speed; yspeed = sprite.y_speed
```

Arguments

sprite handle of the created sprite

speed vector containing the horizontal and vertical speeds { x, y }

xspeed amount to add to the x position of the sprite at each updatesprites call

yspeed amount to add to the y position of the sprite at each updatesprites call

Example

```
radians( true )  
image = loadImage( "Untied Games/Enemy small top C", false )  
ship = createSprite()  
setSpriteImage( ship, image )  
lastpos = { gWidth() / 2, gHeight() / 2 }  
setSpriteLocation( ship, lastpos )  
setSpriteScale( ship, { 4, 4 } )  
  
loop  
  clear()  
  c = controls( 0 )  
  printAt( 0, 0, "Use left joystick to control sprite" )  
  setSpriteSpeed( ship, { 600 * c.lx, -600 * c.ly } )  
  curpos = getSpriteLocation( ship )  
  if curpos != lastpos then  
    setSpriteRotation( ship, -pi / 2 + atan2( curpos.y - lastpos.y, curpos.x - lastpos.x ) )  
    lastpos = curpos  
  endif  
  sv = getSpriteSpeed( ship )  
  speed = sqrt( sv.x * sv.x + sv.y * sv.y )  
  printAt( 0, 1, "speed ", int( speed ) ) // print current speed  
  updateSprites()  
  drawSprites()  
  update()  
repeat
```




Associated Commands

`setSpriteSpeed()`

COMMAND REFERENCE

getSpriteVisibility()

Purpose

Find out if a sprite is visible

Description

Get the current visibility state of the specified sprite

Syntax

```
shown = getSpriteVisibility( sprite )  
  
shown = sprite.visible
```

Arguments

sprite handle of the sprite

shown if the sprite is currently visible

Example

```
image = loadImage( "Untied Games/Enemy A", false )  
enemy = createSprite()  
setSpriteImage( enemy, image )  
speed = 20  
setSpriteAnimation( enemy, 0, 4, speed )  
lastpos = { gWidth() / 2, gHeight() / 2 }  
setSpriteLocation( enemy, lastpos )  
setSpriteScale( enemy, { 8, 8 } )  
  
loop  
  clear()  
  c = controls( 0 )  
  printAt( 0, 0, "Hold down the A button to show the sprite" )  
  setSpriteVisibility( enemy, c.a )  
  visible = getSpriteVisibility( enemy )  
  if visible then  
    printAt( 0, 1, "Sprite is visible" )  
  else  
    printAt( 0, 1, "Sprite is invisible" )  
  endif  
  updateSprites()  
  drawSprites()  
  update()  
repeat
```



Associated Commands

`setSpriteVisibility()`

COMMAND REFERENCE

getVertex()

Purpose

Find the screen position of a vertex (point) in a shape

Description

Returns a vector describing the screen position of a desired vertex in a shape drawn with `drawShape()`

Syntax

```
position = getVertex( shape, vertex )
```

Arguments

shape Handle which stores the shape in question

position Vector describing the screen x and y position of the desired vertex

vertex Float index of the desired vertex (begins at 0, clockwise)

Example

Associated Commands

`copyShape()`, `createBox()`, `createCircle()`, `createCurve()`, `createLine()`, `createLineStrip()`, `createPoly()`, `createStar()`, `createTriangle()`, `deleteShape()`, `drawShape()`, `getShapeBounds()`, `getShapeLocation()`, `getShapeRotation()`, `getShapeScale()`, `getShapeTint()`, `getVertexColour()`, `getVertexLineColour()`, `getVertexLineThickness()`, `joinShapes()`, `moveShape()`, `numVerts()`, `rotateShape()`, `scaleShape()`, `setShapeColour()`, `setShapeLineStyle()`, `setShapeRotation()`, `setShapeScale()`, `setShapeScaleModeLocal()`, `setShapeTint()`, `setVertex()`, `setVertexColour()`, `setVertexLineStyle()`

COMMAND REFERENCE

getVertexColour()

Purpose

Gets the colour of a vertex

Description

Returns the vector (RGBA) colour of a supplied vertex in a shape drawn with `drawShape()`

Syntax

```
colour = getVertexColour( shape, vertex )
```

Arguments

colour Vector (RGBA) describing the colour of the supplied vertex

shape Handle of the shape in question

vertex Number of the desired vertex (0 - N)

Example

```
shape = createCircle( gwidth() / 2, gheight() / 2, 200, 360 )  
  
for i = 0 to 360 loop  
    setVertexColour( shape, i, { random( 1.0 ), random( 1.0 ), random( 1.0 ), 1 } )  
repeat  
  
loop  
    clear()  
    col = getVertexColour( shape, 180 )  
    print( col )  
    drawShape( shape )  
    update()  
repeat
```

Associated Commands

`copyShape()`, `createBox()`, `createCircle()`, `createCurve()`, `createLine()`, `createLineStrip()`, `createPoly()`, `createStar()`, `createTriangle()`, `deleteShape()`, `drawShape()`, `getShapeBounds()`, `getShapeLocation()`, `getShapeRotation()`, `getShapeScale()`, `getShapeTint()`, `getVertex()`, `getVertexLineColour()`, `getVertexLineThickness()`, `joinShapes()`, `moveShape()`, `numVerts()`, `rotateShape()`, `scaleShape()`, `setShapeColour()`, `setShapeLineStyle()`, `setShapeRotation()`, `setShapeScale()`, `setShapeScaleModeLocal()`, `setShapeTint()`, `setVertex()`, `setVertexColour()`, `setVertexLineStyle()`

COMMAND REFERENCE

getVertexLineColour()

Purpose

Find the colour (tint) of a supplied vertex line

Description

Returns a colour vector (RGBA) describing the colour of a supplied vertex line in a shape drawn with `drawShape()`

Syntax

```
colour = getVertexLineColour( shape, vertex )
```

Arguments

shape Handle which stores the shape in question

colour Vector (RGBA) describing the colour of the line through the supplied vertex

vertex Integer index of the desired vertex (begins at 0, clockwise)

Example

Associated Commands

`copyShape()`, `createBox()`, `createCircle()`, `createCurve()`, `createLine()`, `createLineStrip()`, `createPoly()`, `createStar()`, `createTriangle()`, `deleteShape()`, `drawShape()`, `getShapeBounds()`, `getShapeLocation()`, `getShapeRotation()`, `getShapeScale()`, `getShapeTint()`, `getVertex()`, `getVertexColour()`, `getVertexLineThickness()`, `joinShapes()`, `moveShape()`, `numVerts()`, `rotateShape()`, `scaleShape()`, `setShapeColour()`, `setShapeLineStyle()`, `setShapeRotation()`, `setShapeScale()`, `setShapeScaleModeLocal()`, `setShapeTint()`, `setVertex()`, `setVertexColour()`, `setVertexLineStyle()`

COMMAND REFERENCE

getVertexLineThickness()

Purpose

Find the thickness of a supplied vertex line

Description

Returns a float describing the thickness (in pixels) of the line through a supplied vertex in a shape drawn with `drawShape()`

Syntax

```
thickness = getVertexLineThickness( shape, vertex )
```

Arguments

shape Handle which stores the shape in question

thickness Float describing the thickness (in pixels) of the line through the supplied vertex

vertex Integer index of the desired vertex (begins at 0, clockwise)

Example

Associated Commands

`copyShape()`, `createBox()`, `createCircle()`, `createCurve()`, `createLine()`, `createLineStrip()`, `createPoly()`, `createStar()`, `createTriangle()`, `deleteShape()`, `drawShape()`, `getShapeBounds()`, `getShapeLocation()`, `getShapeRotation()`, `getShapeScale()`, `getShapeTint()`, `getVertex()`, `getVertexColour()`, `getVertexLineColour()`, `joinShapes()`, `moveShape()`, `numVerts()`, `rotateShape()`, `scaleShape()`, `setShapeColour()`, `setShapeLineStyle()`, `setShapeRotation()`, `setShapeScale()`, `setShapeScaleModeLocal()`, `setShapeTint()`, `setVertex()`, `setVertexColour()`, `setVertexLineStyle()`

COMMAND REFERENCE

imageH()

Purpose

Get the height of a loaded image

Description

Returns the height in pixels of an image file previously loaded with loadImage

Syntax

```
imageHeight = imageH( image )
```

Arguments

imageHeight Handle which stores the result of the function call

image Handle which stores the image

Example

```
img = loadImage( "Ansimuz/CyberpunkStreetLayer0" )  
  
imageHeight = imageH( img )  
  
print( imageHeight )  
update()  
sleep( 3 )
```

Associated Commands

clear(), createImage(), drawImage(), imageW(), imageSize(), drawSheet(), loadImage(), update(), uploadImage()

COMMAND REFERENCE

imageSize()

Purpose

Get the size of a loaded image

Description

Returns the width and height of an image previously loaded with loadImage

Syntax

```
size = imageSize( sprite )
```

Arguments

sprite handle returned by a call to loadImage

size vector containing the width (.x) and height (.y) of the image

Example

```
// Scale an image to fit to the screen
img = loadImage( "Colin Brown/DungeonB", false )
size = imageSize( img )
scale = min( gwidth() / size.x, gheight() / size.y )
drawImage( img, 0, 0, scale )
update()
sleep( 3 )
```



Associated Commands

clear(), createImage(), drawImage(), drawQuad(), drawSheet(), loadImage(), update(), uploadImage()

COMMAND REFERENCE

imageW()

Purpose

Get the width of a loaded image

Description

Returns the width in pixels of an image file previously loaded with loadImage

Syntax

```
imageWidth = imageW( image )
```

Arguments

imageWidth Handle which stores the result of the function call

image Handle which stores the image

Example

```
img = loadImage( "Ansimuz/CyberpunkStreetLayer0" )  
  
imageWidth = imageW( img )  
  
print( imageWidth )  
update()  
sleep( 3 )
```

Associated Commands

[clear\(\)](#), [createImage\(\)](#), [drawImage\(\)](#), [imageH\(\)](#), [imageSize\(\)](#), [drawSheet\(\)](#), [loadImage\(\)](#), [update\(\)](#), [uploadImage\(\)](#)

COMMAND REFERENCE

joinShapes()

Purpose

Join two shapes drawn using `drawShape()` together

Description

Create a new shape by combining two other shapes

Syntax

```
newShape = joinShapes( shape1, shape2 )
```

Arguments

newShape Handle which stores the newly created shape

shape1 Handle which stores the first shape to join

shape2 Handle which stores the second shape to join

Example

```
w = gwidth()
h = gheight()
radius = 100

shape1 = createCircle( w / 3, h / 2, radius, 360 )
shape2 = createCircle( w - w / 3, h / 2, radius, 360 )
shape3 = 0

join = false
dist = 0

loop
  clear( grey )
  j = controls( 0 )

  if !join then
    shape1Location = getShapeLocation( shape1 )
    shape2Location = getShapeLocation( shape2 )
    dist = distance( shape1Location, shape2Location )
  endif

  if dist < radius * 2 and !join then
    shape3 = joinShapes( shape1, shape2 )
    deleteShape( shape1 )
    deleteShape( shape2 )
    join = true
```

```

endif

if join then
    moveShape( shape3, { j.lx, -j.ly } * 6 )
    drawShape( shape3 )
else
    moveShape( shape1, { j.lx, -j.ly } * 6 )
    drawShape( shape1 )
    drawShape( shape2 )
endif

update()
repeat

```

Associated Commands

[createLine\(\)](#), [createLineStrip\(\)](#), [createCurve\(\)](#), [createCircle\(\)](#), [createPoly\(\)](#), [createTriangle\(\)](#),
[createStar\(\)](#), [createBox\(\)](#), [copyShape\(\)](#), [deleteShape\(\)](#), [drawShape\(\)](#), [moveShape\(\)](#),
[getShapeBounds\(\)](#), [getShapeLocation\(\)](#), [setShapeLocation\(\)](#), [getShapeRotation\(\)](#),
[setShapeRotation\(\)](#), [rotateShape\(\)](#), [getShapeScale\(\)](#), [setShapeScale\(\)](#), [scaleShape\(\)](#),
[getShapeTint\(\)](#), [setShapeTint\(\)](#), [numVerts\(\)](#), [getVertex\(\)](#), [setVertex\(\)](#), [getVertexColour\(\)](#),
[setVertexColour\(\)](#), [getVertexLineThickness\(\)](#), [setVertexLineStyle\(\)](#), [setShapeColour\(\)](#),
[setShapeLineStyle\(\)](#), [setShapeScaleModeLocal\(\)](#)

COMMAND REFERENCE

line()

Purpose

Draw a line

Description

Draw a line between two points in the specified colour

Syntax

```
line( point1, point2, colour )
```

Arguments

point1 screen coordinates of first point in pixels { x, y }

point2 screen coordinates of second point in pixels { x, y }

colour colour name or RGB values { red, green, blue, opacity } between 0 and 1

Example

```
// Draw 100 random lines
clear()
for i = 1 to 100 loop
  point1 = { random( gWidth() ), random( gHeight() ) }
  point2 = { random( gWidth() ), random( gHeight() ) }
  col = { random( 101 ) / 100, random( 101 ) / 100, random( 101 ) / 100, random( 101 ) / 100 }
  line( point1, point2, col )
  update()
repeat

// Wait 10 seconds
sleep( 10 )
```

Associated Commands

[box\(\)](#), [circle\(\)](#), [triangle\(\)](#)

COMMAND REFERENCE

loadImage()

Purpose

Load an image from a file

Description

Loads an image from the file specified into memory ready for display

Syntax

```
handle = loadImage( filename )  
handle = loadImage( filename, filter )
```

Arguments

handle variable which stores the desired image file

filename relative path of the image to load

filter set filtering on or off - generally on for real images and off for pixel art

Example

```
roll = 0  
clear()  
image = loadImage( "Colin Brown/Dice", false )  
size = tileSize( image, 0 )  
  
for i = 1 to 10 loop  
  clear()  
  roll = random( 6 ) + 1  
  x = size.x - ( size.x * ( roll % 2 ) )  
  y = size.y * ( ceil( roll / 2 ) - 1 )  
  drawimage( image, { x, y, size.x, size.y }, { 0, 0, size.x, size.y } )  
  update()  
  sleep( 0.3 )  
repeat  
printAt( 0, 15, "You rolled a ", roll )  
update()  
sleep( 3 )
```



Associated Commands

`clear()`, `createImage()`, `drawImage()`, `drawImageEx()`, `drawQuad()`, `drawSheet()`, `update()`, `uploadImage()`

COMMAND REFERENCE

loadMap()

Purpose

Load a tile map file

Description

Load a tile map file created in the tile map editor into memory

Syntax

```
loadMap( filename )
```

Arguments

filename The name of the map created with the tile map editor

Example

```
// To view this map demo, please load the project "Map Commands Demo" from FUZE Programs.
// Maps must be stored in the project you wish to load them into.

maps = [
    "map1",
    "map2"
]

m = 0
layer = false

press = [
    .a = false,
    .up = false
]

loadMap( maps[m] )
setSpriteCamera( 0, 0, 2 )

loop
    centerSpriteCamera( 0, 0 )
    clear()

    c = controls( 0 )

    if !c.a then
        press.a = false
    endIf
    if !c.up then
        press.up = false
    endIf
```

```

drawMapLayer( 0 )

if layer then
    drawMapLayer( 1 )
endif

if c.up and !press.up then
    press.up = true
    layer = !layer
endif

if c.a and !press.a then
    press.a = true
    unloadMap()
    m += 1
    if m >= 2 then
        m = 0
    endif
    loadMap( maps[m] )
endif

printAt( 0, 0, "Press A button to swap between maps" )
printAt( 0, 2, "Currently viewing: " + maps[m] )
printAt( 0, 4, "Press Up directional button to toggle additional layers" )

update()
repeat

```

Associated Commands

[drawMap\(\)](#), [drawMapLayer\(\)](#), [unloadMap\(\)](#)

COMMAND REFERENCE

moveShape()

Purpose

Apply movement to a shape drawn with `drawShape()`

Description

Applies movement on a pixel-per-frame basis to a shape's x and y screen position

Syntax

```
moveShape( shape, x, y )  
moveShape( shape, axes )
```

Arguments

shape Handle which stores the shape to move

x Amount (in pixels) to move the shape on the horizontal axis

y Amount (in pixels) to move the shape on the vertical axis

axes Vector describing the amount (in pixels) to move the shape on both axes

Example

```
shape = createCircle( gwidth() / 2, gheight() / 2, 200, 360 )  
  
// move the circle using the left control stick values  
loop  
  clear( grey )  
  j = controls( 0 )  
  moveShape( shape, { j.lx, -j.ly } * 5 )  
  drawShape( shape )  
  update()  
repeat
```

Associated Commands

`copyShape()`, `createBox()`, `createCircle()`, `createCurve()`, `createLine()`, `createLineStrip()`, `createPoly()`, `createStar()`, `createTriangle()`, `deleteShape()`, `drawShape()`, `getShapeBounds()`, `getShapeLocation()`, `getShapeRotation()`, `getShapeScale()`, `getShapeTint()`, `getVertex()`, `getVertexColour()`, `getVertexLineColour()`, `getVertexLineThickness()`, `joinShapes()`, `numVerts()`, `rotateShape()`, `scaleShape()`, `setShapeColour()`, `setShapeLineStyle()`, `setShapeRotation()`, `setShapeScale()`, `setShapeScaleModeLocal()`, `setShapeTint()`, `setVertex()`, `setVertexColour()`, `setVertexLineStyle()`

COMMAND REFERENCE

numTiles()

Purpose

Find the number of tiles in a tilesheet

Description

Returns the number of tiles in a supplied sheet. Returns a 0 if supplied image is not a tilesheet

Syntax

```
number = numTiles( tilesheet )
```

Arguments

tilesheet handle of the tilesheet

count number of tiles in the tilesheet

Example

```
img = loadImage( "Ansimuz/LadyIdle" )  
  
n = numTiles( img )  
print( n )  
update()  
sleep( 3 )
```

Associated Commands

[drawImage\(\)](#), [drawImageEx\(\)](#), [drawSheet\(\)](#), [tileSize\(\)](#)

COMMAND REFERENCE

numVerts()

Purpose

Find the number of vertices (points) in a shape

Description

Returns the number of vertices (points) in a shape drawn with `drawShape()`

Syntax

```
points = numVerts( shape )
```

Arguments

shape Handle which stores the shape in question

points Integer number of vertices (points) in the shape

Example

Associated Commands

`copyShape()`, `createBox()`, `createCircle()`, `createCurve()`, `createLine()`, `createLineStrip()`, `createPoly()`, `createStar()`, `createTriangle()`, `deleteShape()`, `drawShape()`, `getShapeBounds()`, `getShapeLocation()`, `getShapeRotation()`, `getShapeScale()`, `getShapeTint()`, `getVertex()`, `getVertexColour()`, `getVertexLineColour()`, `getVertexLineThickness()`, `joinShapes()`, `moveShape()`, `rotateShape()`, `scaleShape()`, `setShapeColour()`, `setShapeLineStyle()`, `setShapeRotation()`, `setShapeScale()`, `setShapeScaleModeLocal()`, `setShapeTint()`, `setVertex()`, `setVertexColour()`, `setVertexLineStyle()`

COMMAND REFERENCE

plot()

Purpose

Plot a single point on the screen.

Description

Set a single pixel on the screen at the specified location to the specified colour.

Syntax

```
plot( x, y, colour )
```

Arguments

x horizontal screen position in pixels

y vertical screen position in pixels

colour colour name or RGB values { red, green, blue, opacity } between 0 and 1

Example

```
// Draw 1000 random points
clear()
for i = 1 to 1000 loop
  x = random( gWidth() )
  y = random( gHeight() )
  col = { random( 101 ) / 100, random( 101 ) / 100, random( 101 ) / 100, random( 101 ) / 100 }
  plot( x, y, col )
  update()
repeat
// Wait 10 seconds
sleep( 10 )
```

Associated Commands

[box\(\)](#), [circle\(\)](#), [line\(\)](#), [triangle\(\)](#)

COMMAND REFERENCE

removeSprite()

Purpose

Remove a sprite that is no longer needed

Description

Free up the memory allocated to a sprite so that it can be reused

Syntax

```
removeSprite( sprite )
```

Arguments

handle The handle of the created sprite

Example

```
image = loadImage( "Untied Games/Enemy small top C", false )
ship = createSprite()
setSpriteImage( ship, image )
lastpos = { gWidth() / 2, gHeight() / 2 }
setSpriteLocation( ship, lastpos )
setSpriteScale( ship, { 4, 4 } )
drawSprites()
update()
sleep( 3 )
clear()
removeSprite( ship )
drawSprites() // nothing is drawn
update()
sleep( 3 )
```

Associated Commands

[createSprite\(\)](#), [deltaTime\(\)](#), [drawSprites\(\)](#), [updateSprites\(\)](#), [updateSprite\(\)](#)

COMMAND REFERENCE

renderEffect()

Purpose

Apply a visual effect

Description

Apply a visual effect to a 2D image

Syntax

```
renderEffect( image, target, effect, arguments )
```

Arguments

image source image (or framebuffer for screen)

target target image (or framebuffer for screen)

effect handle of effect e.g. *fx_motionblur*

arguments list of arguments for the effect. Parameters are as follows (unused indicates an entry is required but has no effect):

fx_blur [1 / width, 1 / height, dirX, dirY]

fx_chromaticAberration [centreX, centreY, scale]

fx_colourAdjust [biasR, biasG, biasB, unused, gainR, gainG, gainB, unused, curveR, curveG, curveB, saturation]

fx_crt [lines, strength, focus]

fx_gb [whiteLevel]

fx_kawaseBlur [1 / width, 1 / height, iteration]

fx_motionBlur [1 / width, 1 / height, dirX, dirY]

fx_outline [threshold, unused, unused, unused, outlineR, outlineG, outlineB]

fx_posterize [levels]

fx_radialBlur [1 / width, 1 / height, centreX, centreY, scale]

fx_sobel []

fx_threshold [threshold]

fx_tonemap [exposure, whitePoint]

fx_vignette [centreX, centreY, scale]

Example

```
pos = { 960, 540 }
vel = { 0, 0 }
col = { 1, 0, 0, 1 }
rt = createImage( 1920, 1080, true, image_rgb )
loop
  c = controls( 0 )
  vel += { c.lx, -c.ly }
  pos += vel
  vel *= 0.95

  if col.r > 0 and col.b <= 0 then
    col.r -= 0.01
    col.g += 0.01
  else
    if col.g > 0 then
      col.g -= 0.01
      col.b += 0.01
    else
      col.b -= 0.01
      col.r += 0.01
    endif
  endif
endIf

setDrawTarget( rt )
box( 0, 0, 1920, 1080, { 0, 0, 0, 0.25 }, false )
circle( pos.x, pos.y, 50, 32, col, false )

setDrawTarget( framebuffer )
clear()
renderEffect( rt, framebuffer, fx_motionblur, [ 1 / 1920, 1 / 1080, vel.x / 2, vel.y / 2 ] )

update()
repeat
```



Associated Commands

`setDrawTarget()`

COMMAND REFERENCE

rotateShape()

Purpose

Apply rotation to a shape drawn with `drawShape()`

Description

Rotates a shape by a given number of degrees or radians

Syntax

```
rotateShape( shape, amount )
```

Arguments

shape Handle which stores the shape to move

amount Number of degrees or radians to rotate each frame. Negative numbers produce anti-clockwise rotation

Example

```
shape = createBox( gwidth() / 2, gheight() / 2, 200, 300 )

// rotate the box with the left control stick y axis
loop
  clear( grey )
  j = controls( 0 )
  rotateShape( shape, j.ly )
  drawShape( shape )
  update()
repeat
```

Associated Commands

`copyShape()`, `createBox()`, `createCircle()`, `createCurve()`, `createLine()`, `createLineStrip()`, `createPoly()`, `createStar()`, `createTriangle()`, `deleteShape()`, `drawShape()`, `getShapeBounds()`, `getShapeLocation()`, `getShapeRotation()`, `getShapeScale()`, `getShapeTint()`, `getVertex()`, `getVertexColour()`, `getVertexLineColour()`, `getVertexLineThickness()`, `joinShapes()`, `moveShape()`, `numVerts()`, `scaleShape()`, `setShapeColour()`, `setShapeLineStyle()`, `setShapeRotation()`, `setShapeScale()`, `setShapeScaleModeLocal()`, `setShapeTint()`, `setVertex()`, `setVertexColour()`, `setVertexLineStyle()`

COMMAND REFERENCE

scaleShape()

Purpose

Sets a scale direction to be applied to a shape

Description

Used to apply a scale direction to a shape drawn with `drawShape()`

Syntax

```
scaleShape( shape, scale )  
scaleShape( shape, dirX, dirY )
```

Arguments

shape Handle which stores the shape in question

scale Vector which describes the x and y scale direction to be applied to the shape

scaleX Float scale direction to be applied to the horizontal axis of the shape

scaleY Float scale direction to be applied to the vertical axis of the shape

Example

Associated Commands

`copyShape()`, `createBox()`, `createCircle()`, `createCurve()`, `createLine()`, `createLineStrip()`, `createPoly()`, `createStar()`, `createTriangle()`, `deleteShape()`, `drawShape()`, `getShapeBounds()`, `getShapeLocation()`, `getShapeRotation()`, `getShapeScale()`, `getShapeTint()`, `getVertex()`, `getVertexColour()`, `getVertexLineColour()`, `getVertexLineThickness()`, `joinShapes()`, `moveShape()`, `numVerts()`, `rotateShape()`, `setShapeColour()`, `setShapeLineStyle()`, `setShapeRotation()`, `setShapeScale()`, `setShapeScaleModeLocal()`, `setShapeTint()`, `setVertex()`, `setVertexColour()`, `setVertexLineStyle()`

COMMAND REFERENCE

setBlend()

Purpose

Set the blend mode

Description

Sets the function used to combine drawn pixels with the background

Syntax

```
setBlend( mode )
```

Arguments

mode blend mode : none (0), mix (1), add (2), subtract (3) or multiply (4)

Example

```
palette = [  
    red,  
    green,  
    blue,  
    yellow,  
    purple,  
    lime,  
    orange,  
    bisque,  
    turquoise,  
    teal,  
    cyan  
]  
  
pos = { 120, 120 }  
img = loadImage( "Finalbossblues/monster_elk", false )  
  
blendTypes = [  
    "None",  
    "Mix",  
    "Add",  
    "Subtract",  
    "Multiply"  
]  
  
type = 0  
press = false  
  
loop
```

```

clear()
c = controls( 0 )
pos.x += c.lx * 4
pos.y -= c.ly * 4
if c.right and !press then
    type += 1
    press = true
endIf
if !c.right then
    press = false
endIf
if type > 4 then
    type = 0
endIf

setBlend( 0 )
w = gWidth() / len( palette )
for i = 0 to len( palette ) loop
    box( i * w, gHeight() / 10, w, gHeight(), palette[i], false )
repeat

setBlend( type )
drawImage( img, pos.x, pos.y, 2 )

setBlend( 1 )
printAt( 0, 0, "Use Joy-Con Left Control Stick to move the image" )
printAt( 0, 1, "Press Right Directional Button to adjust blend type" )
printAt( 0, 2, "Blend Type: " + blendTypes[type] )
update()
repeat

```

Associated Commands

COMMAND REFERENCE

setShapeColour()

Purpose

Applies a colour to a shape

Description

Used to apply a colour vector (RGBA) to a shape drawn with `drawShape()`

Syntax

```
setShapeColour( shape, colour )
```

Arguments

shape Handle which stores the shape in question

colour RGBA vector which describes the colour to be applied to the supplied shape

Example

Associated Commands

`copyShape()`, `createBox()`, `createCircle()`, `createCurve()`, `createLine()`, `createLineStrip()`, `createPoly()`, `createStar()`, `createTriangle()`, `deleteShape()`, `drawShape()`, `getShapeBounds()`, `getShapeLocation()`, `getShapeRotation()`, `getShapeScale()`, `getShapeTint()`, `getVertex()`, `getVertexColour()`, `getVertexLineColour()`, `getVertexLineThickness()`, `joinShapes()`, `moveShape()`, `numVerts()`, `rotateShape()`, `scaleShape()`, `setShapeColour()`, `setShapeLineStyle()`, `setShapeRotation()`, `setShapeScale()`, `setShapeScaleModeLocal()`, `setVertex()`, `setVertexColour()`, `setVertexLineStyle()`

COMMAND REFERENCE

setShapeLineStyle()

Purpose

Set the draw style for the outline of a shape

Description

Used to change the colour and thickness of the outline in a shape drawn with `drawShape()`

Syntax

```
setShapeLineStyle( shape, thickness, tint )
```

Arguments

shape Handle which stores the shape in question

thickness Float describing the thickness (in pixels, from the centre of the line outward) of the line

tint Vector (RGBA) to set the colour of the outline

Example

Associated Commands

`copyShape()`, `createBox()`, `createCircle()`, `createCurve()`, `createLine()`, `createLineStrip()`, `createPoly()`, `createStar()`, `createTriangle()`, `deleteShape()`, `drawShape()`, `getShapeBounds()`, `getShapeLocation()`, `getShapeRotation()`, `getShapeScale()`, `getShapeTint()`, `getVertex()`, `getVertexColour()`, `getVertexLineColour()`, `getVertexLineThickness()`, `joinShapes()`, `moveShape()`, `numVerts()`, `rotateShape()`, `scaleShape()`, `setShapeColour()`, `setShapeRotation()`, `setShapeScale()`, `setShapeScaleModeLocal()`, `setShapeTint()`, `setVertex()`, `setVertexColour()`, `setVertexLineStyle()`

COMMAND REFERENCE

setShapeLocation()

Purpose

Set the pixel co-ordinate location of a given shape to be drawn with `drawShape()`

Description

Changes the on-screen location of a shape, overwriting its original position

Syntax

```
setShapeLocation( shape, x, y )  
setShapeLocation( shape, location )
```

Arguments

shape Handle which stores the shape to move

x New horizontal screen position (in pixels)

y New vertical screen position (in pixels)

location Vector which describes the new screen position for the shape

Example

```
shape = createCircle( gwidth() / 2, gheight() / 2, 200, 360 )  
  
setShapeLocation( shape, 500, 500 )  
  
loop  
  clear( grey )  
  drawShape( shape )  
  update()  
repeat
```

Associated Commands

`copyShape()`, `createBox()`, `createCircle()`, `createCurve()`, `createLine()`, `createLineStrip()`, `createPoly()`, `createStar()`, `createTriangle()`, `deleteShape()`, `drawShape()`, `getShapeBounds()`, `getShapeLocation()`, `getShapeRotation()`, `getShapeScale()`, `getShapeTint()`, `getVertex()`, `getVertexColour()`, `getVertexLineColour()`, `getVertexLineThickness()`, `joinShapes()`, `moveShape()`, `numVerts()`, `rotateShape()`, `scaleShape()`, `setShapeColour()`, `setShapeLineStyle()`, `setShapeRotation()`, `setShapeScale()`, `setShapeScaleModeLocal()`, `setShapeTint()`, `setVertex()`, `setVertexColour()`, `setVertexLineStyle()`

COMMAND REFERENCE

setShapeRotation()

Purpose

Sets the rotation of a shape in degrees or radians

Description

Used to apply rotation to a shape drawn with `drawShape()`

Syntax

```
setShapeRotation( shape, amount )
```

Arguments

shape Handle which stores the shape to rotate

amount Float number of degrees (or radians) to rotate the shape by around the origin. Negative numbers apply counter-clockwise rotation

Example

Associated Commands

`copyShape()`, `createBox()`, `createCircle()`, `createCurve()`, `createLine()`, `createLineStrip()`, `createPoly()`, `createStar()`, `createTriangle()`, `deleteShape()`, `drawShape()`, `getShapeBounds()`, `getShapeLocation()`, `getShapeRotation()`, `getShapeScale()`, `getShapeTint()`, `getVertex()`, `getVertexColour()`, `getVertexLineColour()`, `getVertexLineThickness()`, `joinShapes()`, `moveShape()`, `numVerts()`, `rotateShape()`, `scaleShape()`, `setShapeColour()`, `setShapeLineStyle()`, `setShapeScale()`, `setShapeScaleModeLocal()`, `setShapeTint()`, `setVertex()`, `setVertexColour()`, `setVertexLineStyle()`

COMMAND REFERENCE

setShapeScale()

Purpose

Sets a scale multiplier to a supplied shape

Description

Used to apply a scale multiplier to a shape drawn with `drawShape()`

Syntax

```
setShapeScale( shape, scale )  
setShapeScale( shape, scaleX, scaleY )
```

Arguments

shape Handle which stores the shape in question

scale Vector which describes the x and y scale to be applied to the shape

scaleX Float scale multiplier to be applied to the horizontal axis of the shape

scaleY Float scale multiplier to be applied to the vertical axis of the shape

Example

Associated Commands

`copyShape()`, `createBox()`, `createCircle()`, `createCurve()`, `createLine()`, `createLineStrip()`, `createPoly()`, `createStar()`, `createTriangle()`, `deleteShape()`, `drawShape()`, `getShapeBounds()`, `getShapeLocation()`, `getShapeRotation()`, `getShapeScale()`, `getShapeTint()`, `getVertex()`, `getVertexColour()`, `getVertexLineColour()`, `getVertexLineThickness()`, `joinShapes()`, `moveShape()`, `numVerts()`, `rotateShape()`, `scaleShape()`, `setShapeColour()`, `setShapeLineStyle()`, `setShapeRotation()`, `setShapeScaleModeLocal()`, `setShapeTint()`, `setVertex()`, `setVertexColour()`, `setVertexLineStyle()`

COMMAND REFERENCE

setShapeScaleModeLocal()

Purpose

Sets the scale mode the local as opposed to global for a shape

Description

Syntax

```
setShapeScaleModeLocal( shape, enabled )
```

Arguments

shape Handle which stores the shape in question

enabled Integer boolean value (0 or 1) for enabled or disabled. Enabled gives local scale mode, disabled gives global

Example

Associated Commands

[copyShape\(\)](#), [createBox\(\)](#), [createCircle\(\)](#), [createCurve\(\)](#), [createLine\(\)](#), [createLineStrip\(\)](#), [createPoly\(\)](#), [createStar\(\)](#), [createTriangle\(\)](#), [deleteShape\(\)](#), [drawShape\(\)](#), [getShapeBounds\(\)](#), [getShapeLocation\(\)](#), [getShapeRotation\(\)](#), [getShapeScale\(\)](#), [getShapeTint\(\)](#), [getVertex\(\)](#), [getVertexColour\(\)](#), [getVertexLineColour\(\)](#), [getVertexLineThickness\(\)](#), [joinShapes\(\)](#), [moveShape\(\)](#), [numVerts\(\)](#), [rotateShape\(\)](#), [scaleShape\(\)](#), [setShapeColour\(\)](#), [setShapeLineStyle\(\)](#), [setShapeRotation\(\)](#), [setShapeScale\(\)](#), [setShapeTint\(\)](#), [setVertex\(\)](#), [setVertexColour\(\)](#), [setVertexLineStyle\(\)](#)

COMMAND REFERENCE

setShapeTint()

Purpose

Applies a tint (colour) to a shape

Description

Used to apply a colour vector (RGBA) to a shape drawn with `drawShape()`

Syntax

```
setShapeTint( shape, tint )
```

Arguments

shape Handle which stores the shape in question

tint RGBA vector which describes the colour to be applied to the supplied shape

Example

Associated Commands

`copyShape()`, `createBox()`, `createCircle()`, `createCurve()`, `createLine()`, `createLineStrip()`, `createPoly()`, `createStar()`, `createTriangle()`, `deleteShape()`, `drawShape()`, `getShapeBounds()`, `getShapeLocation()`, `getShapeRotation()`, `getShapeScale()`, `getShapeTint()`, `getVertex()`, `getVertexColour()`, `getVertexLineColour()`, `getVertexLineThickness()`, `joinShapes()`, `moveShape()`, `numVerts()`, `rotateShape()`, `scaleShape()`, `setShapeColour()`, `setShapeLineStyle()`, `setShapeRotation()`, `setShapeScale()`, `setShapeScaleModeLocal()`, `setVertex()`, `setVertexColour()`, `setVertexLineStyle()`

COMMAND REFERENCE

setSpriteAnimation()

Purpose

Animate a sprite

Description

Create a sprite from a tile sheet and animate it.

Syntax

```
setSpriteAnimation( sprite, startTile, endTile )  
setSpriteAnimation( sprite, startTile, endTile, speed )
```

Arguments

sprite The handle of the sprite

startTile The number of the first tile in the animation sequence (0 based)

endTile The number of the last tile in the animation sequence (0 based)

speed The speed of the animation (default is 10)

Example

```
image = loadImage( "Untied Games/Enemy A", false )  
enemy = createSprite()  
setSpriteImage( enemy, image )  
setSpriteAnimation( enemy, 0, 4, 20 )  
lastpos = { gWidth() / 2, gHeight() / 2 }  
setSpriteLocation( enemy, lastpos )  
setSpriteScale( enemy, { 8, 8 } )  
  
loop  
  clear()  
  updateSprites()  
  drawSprites()  
  update()  
repeat
```



Associated Commands

`getSpriteAnimFrame()`, `getSpriteAnimFrameCount()`, `getSpriteAnimSpeed()`,
`setSpriteAnimFrame()`, `setSpriteAnimSpeed()`

COMMAND REFERENCE

setSpriteAnimFrame()

Purpose

Set the current frame in an animated sprite

Description

Set the current frame in an animated sprite to be the specified frame number

Syntax

```
setSpriteAnimFrame( sprite, frame )
```

Arguments

sprite handle of the sprite

frame number of the current animation frame in the sprite

Example

```
image = loadImage( "Untied Games/Explosion 01", false )
explosion = createSprite()
setSpriteImage( explosion, image )
setSpriteAnimation( explosion, 0, 69, 60 )
setSpriteLocation( explosion, { gWidth() / 2, gHeight() / 2 } )
setSpriteScale( explosion, { 5, 5 } )
tiles = getSpriteAnimFrameCount( explosion )

for i = 0 to tiles loop
  clear()
  setSpriteAnimFrame( explosion, i )
  drawSprites()
  update()
repeat
```



Associated Commands

`getSpriteAnimFrame()`, `getSpriteAnimFrameCount()`, `getSpriteAnimSpeed()`,
`setSpriteAnimation()`, `setSpriteAnimSpeed()`

COMMAND REFERENCE

setSpriteAnimSpeed()

Purpose

Change the speed of a sprites animation

Description

Speed up or slow down the speed of animation of a sprite

Syntax

```
setSpriteAnimSpeed( sprite, speed )  
  
sprite.anim_speed = speed
```

Arguments

sprite handle of the sprite

speed speed of the animation

Example

```
image = loadImage( "Untied Games/Enemy A", false )  
enemy = createsprite()  
setSpriteImage( enemy, image )  
setSpriteAnimation( enemy, 0, 4, 0 )  
lastpos = { gWidth() / 2, gHeight() / 2 }  
setSpriteLocation( enemy, lastpos )  
setSpriteScale( enemy, { 8, 8 } )  
  
loop  
  clear()  
  c = controls( 0 )  
  printAt( 0, 0, "Use left joystick to control animation speed" )  
  setSpriteAnimSpeed( enemy, c.lx * 30 )  
  updateSprites()  
  drawSprites()  
  update()  
repeat
```



Associated Commands

`getSpriteAnimFrame()`, `getSpriteAnimFrameCount()`, `getSpriteAnimSpeed()`,
`setSpriteAnimation()`, `setSpriteAnimFrame()`

COMMAND REFERENCE

setSpriteCamera()

Purpose

Move the sprite camera position

Description

Move the camera position for all sprites. The initial position is (0, 0, 1)

Syntax

```
setSpriteCamera( pos )  
  
setSpriteCamera( xpos, ypos )  
  
setSpriteCamera( xpos, ypos, zpos )
```

Arguments

pos position vector of the camera { x, y, z }

xpos position of the camera in the x axis (pan left/right)

ypos position of the camera in the y axis (pan up/down)

zpos position of the camera in the z axis (zoom in/out)

Example

```
image = loadImage( "Untied Games/Enemy A", false )  
enemy = []  
for i = 0 to 4 loop  
    enemy[i] = createsprite()  
    setSpriteImage( enemy[i], image )  
    setSpriteAnimation( enemy[i], 0, 4, 20 )  
    setSpriteLocation( enemy[i], { ( i % 2 ) * 400 + 400, int( i / 2 ) * 300 + 200 } )  
    setSpriteScale( enemy[i], { 4, 4 } )  
repeat  
camera = getSpriteCamera()  
rotation = getSpriteCameraRotation()  
loop  
    clear()  
    c = controls( 0 )  
    printAt( 0, 0, "Camera posotion: x = ", camera.x, " y = ", camera.y, " z = ", camera.z, " rotation: ", rotation )  
    printAt( 0, 1, "Use left joypad to pan, right joypad to zoom/rotate" )  
    if c.up then  
        camera.y -= 5  
    endIf  
    if c.down then  
        camera.y += 5  
    endIf  
    if c.left then  
        camera.x -= 5  
    endIf  
    if c.right then  
        camera.x += 5  
    endIf  
    if c.x then  
        camera.z += 0.05  
    endIf
```

```
if c.b then
  camera.z -= 0.05
endIf
if c.y then
  rotation -= 0.5
endIf
if c.a then
  rotation += 0.5
endIf
setSpriteCamera( camera.x, camera.y, camera.z )
setSpriteCameraRotation( rotation )
updateSprites()
drawSprites()
update()
repeat
```



Associated Commands

`centreSpriteCamera()`, `getSpriteCamera()`, `getSpriteCameraRotation()`, `setSpriteCameraRotation()`

COMMAND REFERENCE

setSpriteCameraRotation()

Purpose

Rotate the sprite camera

Description

Set the sprite camera rotation angle

Syntax

```
setSpriteCameraRotation( angle )
```

Arguments

angle camera rotation angle in the default units

Example

```
image = loadImage( "Untied Games/Enemy A", false )
enemy = []
for i = 0 to 4 loop
    enemy[i] = createSprite()
    setSpriteImage( enemy[i], image )
    setSpriteAnimation( enemy[i], 0, 4, 20 )
    setSpriteLocation(enemy[i], { i % 2 } * 400 + 400, int( i / 2 ) * 300 + 200 )
    setSpriteScale( enemy[i], { 4, 4 } )
repeat

camera = getSpriteCamera()
rotation = getSpriteCameraRotation()
loop
    clear()
    c = controls( 0 )
    printAt( 0, 0, "Camera position: x = ", camera.x, " y = ", camera.y, " z = ", camera.z, " rotation: ", rotation )
    printAt( 0, 1, "Use left joypad to pan, right joypad to zoom/rotate" )
    if c.up then
        camera.y -= 5
    endIf
    if c.down then
        camera.y += 5
    endIf
    if c.left then
        camera.x -= 5
    endIf
    if c.right then
        camera.x += 5
    endIf
    if c.x then
        camera.z += 0.05
    endIf
    if c.b then
        camera.z -= 0.05
    endIf
    if c.y then
        rotation -= 0.5
    endIf
    if c.a then
        rotation += 0.5
    endIf
    setSpriteCamera( camera.x, camera.y, camera.z )
    setSpriteCameraRotation( rotation )
    updateSprites()
    drawSprites()
    update()
repeat
```



Associated Commands

`centreSpriteCamera()`, `getSpriteCamera()`, `getSpriteCameraRotation()`, `setSpriteCamera()`

COMMAND REFERENCE

setSpriteCollisionShape()

Purpose

Sets the sprite's collision shape

Description

Sets the sprite's collision shape to the given shape. The collision shape will rotate, scale, and move along with the sprite. The size will be automatically set according to the sprite's image or tile dimensions.

Syntax

```
setSpriteCollisionShape( sprite, shape )  
setSpriteCollisionShape( sprite, shape, width, height, rotation )
```

Arguments

sprite handle of the sprite

shape shape of the sprite's collision (SHAPE_BOX, SHAPE_TRIANGLE, or SHAPE_CIRCLE)

width width of the shape in pixels

height height of the shape in pixels

rotation rotation of the shape in default units

Example

```
image = loadImage( "Untied Games/Enemy small top C", false )  
ship = []  
for i = 0 to 2 loop  
    ship[i] = createSprite()  
    setSpriteImage( ship[i], image )  
    setSpriteScale( ship[i], { 5, 5 } )  
    setSpriteCollisionShape( ship[i], SHAPE_TRIANGLE, 25, 25, 180 )  
    ship[i].show_collision_shape = true  
repeat  
setSpriteRotation( ship[0], 270 )  
setSpriteSpeed( ship[0], { 240, 0 } )  
setSpriteSpeed( ship[1], { 0, 120 } )  
setSpriteColour( ship[1], { 0, 0, 1, 1 } )  
setSpriteLocation( ship[0], { 0, gHeight() / 2 } )  
setSpriteLocation( ship[1], { gWidth() / 2, 0 } )  
collide = false
```

```
while !collide loop
  clear()
  updateSprites()
  drawSprites()
  update()
  collide = detectSpriteCollision( ship[0],ship[1] )
repeat
```



Associated Commands

`collideSprites()`, `detectSpriteCollision()`

COMMAND REFERENCE

setSpriteColour()

Purpose

Set the colour of a sprite

Description

Sets the red, green, blue and opacity (alpha) values for a sprite

Syntax

```
setSpriteColour ( sprite, colour )  
  
setSpriteColour ( sprite, red, green, blue, alpha )  
  
sprite.r = red; sprite.g = green; sprite.b = blue; sprite.a = alpha
```

Arguments

sprite The handle of the created sprite

colour named colour or vector { r, g, b, a }

red red value between 0 and 1

green green value between 0 and 1

blue blue value between 0 and 1

alpha opacity value between 0 and 1

Example

```
image = loadImage( "Untied Games/Enemy small top C", false )  
ship = createSprite()  
setSpriteImage( ship, image )  
lastpos = { gWidth() / 2, gHeight() / 2 }  
setSpriteLocation( ship, lastpos )  
setSpriteScale( ship, { 10, 10 } )  
  
loop  
  clear()  
  r = random( 101 ) / 100  
  g = random( 101 ) / 100  
  b = random( 101 ) / 100  
  setSpriteColour( ship, { r, g, b, 1 } )  
  updateSprites()  
  drawSprites()  
  update()
```

```
sleep( 0.1 )  
repeat
```



Associated Commands

`getSpriteColour()`, `getSpriteColourSpeed()`, `setSpriteColourSpeed()`

COMMAND REFERENCE

setSpriteColourSpeed()

Purpose

Set the colour speeds of a sprite

Description

Set the rates of change of the colours of a sprite. These are the amounts that the sprite colours are changed by when updatesprites is called

Syntax

```
setSpriteColourSpeed( sprite, rgbaSpeed )  
setSpriteColourSpeed( sprite, rSpeed, gSpeed, bSpeed, aSpeed )  
sprite.r_speed = rspeed; sprite.g_speed = gspeed; sprite.b_speed = bspeed; sprite.a_speed = aspeed
```

Arguments

sprite handle of the created sprite

rgbaSpeed amount to add to the red, green, blue and opacity of the sprite at each updatesprites call

rSpeed amount to add to the red colour of the sprite at each updatesprites call

gSpeed amount to add to the blue colour of the sprite at each updatesprites call

bSpeed amount to add to the green colour of the sprite at each updatesprites call

aSpeed amount to add to the opacity of the sprite at each updatesprites call

Example

```
image = loadImage( "Untied Games/Enemy small top C", false )  
ship = createSprite()  
setSpriteImage( ship, image )  
lastpos = { gWidth() / 2, gHeight() / 2 }  
setSpriteLocation( ship, lastpos )  
setSpriteScale( ship, { 20, 20 } )  
rv = -0.5  
gv = 0.5  
bv = 0  
  
loop  
  clear()  
  sc = getSpriteColour( ship )  
  if sc.r > 1 or sc.r < 0 then  
    rv = -rv
```

```
endIf
if sc.b > 1 or sc.b < 0 then
    gv = -gv
endIf
setSpriteColourSpeed( ship, { rv, gv, bv, 0 } )
updateSprites()
drawSprites()
update()
repeat
```



Associated Commands

`getSpriteColour()`, `getSpriteColourSpeed()`, `setSpriteColour()`

COMMAND REFERENCE

setSpriteDepth()

Purpose

Set a sprites depth

Description

Sets the visual depth of the sprite. For drawing, sprites will automatically be sorted by their depth from negative (earliest drawing) to positive (latest drawing).

Syntax

```
setSpriteDepth( sprite, depth )  
  
sprite.depth = depth
```

Arguments

sprite The handle of the created sprite

depth The visual depth of the sprite

Example

```
image = loadImage( "Untied Games/Enemy small top C", false )  
ship = []  
for i = 0 to 2 loop  
    ship[i] = createSprite()  
    setSpriteImage( ship[i], image )  
    setSpriteScale( ship[i], { 5, 5 } )  
    setSpriteDepth( ship[i], i )  
repeat  
  
setSpriteRotation( ship[0], 270 )  
setSpriteLocation( ship[0], { 0, gHeight() / 2 } )  
setSpriteLocation( ship[1], { gWidth() / 2, 0 } )  
setSpriteSpeed( ship[0], { 240, 0 } )  
setSpriteSpeed( ship[1], { 0, 120 } )  
setSpriteColour( ship[1], { 0, 0, 1, 1 } )  
  
while ship[0].x < gwidth() loop  
    clear()  
    printAt( 0, 0, "Blue ship is drawn first so red ship passes over it" )  
    updateSprites()  
    drawSprites()  
    update()  
repeat
```



Associated Commands

`getSpriteDepth()`

COMMAND REFERENCE

setSpriteImage()

Purpose

Change the image associated with a sprite

Description

Set the sprites image to the one specified

Syntax

```
setSpriteImage( sprite, image )  
  
sprite.image = image
```

Arguments

sprite handle of the sprite

image handle of the image

Example

```
image = []  
image[1] = loadImage( "Untied Games/Enemy A", false )  
image[2] = loadImage( "Untied Games/Enemy B", false )  
enemy = createSprite()  
setSpriteImage( enemy, image[1] )  
setSpriteAnimation( enemy, 0, 4, 20 )  
lastpos = { gWidth() / 2, gHeight() / 2 }  
setSpriteLocation( enemy, lastpos )  
setSpriteScale( enemy, { 8, 8 } )  
current = 1  
last = 1  
  
loop  
  clear()  
  c = controls( 0 )  
  printAt( 0, 0, "Press A for image 1 and B for image 2" )  
  if c.a then  
    current = 1  
  endif  
  if c.b then  
    current = 2  
  endif  
  if current != last then  
    setSpriteImage( enemy, image[current] )  
    setSpriteAnimation( enemy, 0, 4, 20 )  
    last = current
```

```
endIf  
updateSprites()  
drawSprites()  
update()  
repeat
```



Associated Commands

`createSprite()`, `getSpriteImage()`, `getSpriteImageSize()`, `setSpriteText()`

COMMAND REFERENCE

setSpriteLocation()

Purpose

Position a sprite on the screen

Description

Set the horizontal and vertical position of a sprite

Syntax

```
setSpriteLocation( sprite, pos )  
setSpriteLocation( sprite, xpos, ypos )  
sprite.x = xpos; sprite.y = ypos
```

Arguments

sprite handle of the created sprite

pos horizontal and vertical position on the screen in pixels { x, y }

xpos horizontal position on the screen in pixels

ypos vertical position on the screen in pixels

Example

```
radians( true )  
image = loadImage( "Untied Games/Enemy small top C", false )  
ship = createSprite()  
setSpriteImage( ship, image )  
lastpos = { gWidth() / 2, gHeight() / 2 }  
setSpriteLocation( ship, lastpos )  
setSpriteScale( ship, { 8, 8 } )  
  
loop  
  clear()  
  c = controls( 0 )  
  printAt( 0, 0, "Use left joystick to control sprite" )  
  setSpriteSpeed( ship, { 480 * c.lx, -480 * c.ly } )  
  curpos = getSpriteLocation( ship )  
  if curpos != lastpos then  
    setSpriteRotation( ship, -pi / 2 + atan2( curpos.y - lastpos.y, curpos.x - lastpos.x ) )  
    lastpos = curpos  
  endif  
  updateSprites()  
  drawSprites()  
  update()  
repeat
```



Associated Commands

`getSpriteLocation()`, `getSpriteOrigin()`, `setSpriteOrigin()`

COMMAND REFERENCE

setSpriteOrigin()

Purpose

Change the origin point of a sprite

Description

By default the origin of a sprite (0, 0) is the centre. This function allows you to change it to be, for example, the top left

Syntax

```
setSpriteOrigin( sprite, pos )  
setSpriteOrigin( sprite, xpos, ypos )
```

Arguments

sprite handle of the sprite

pos horizontal and vertical origin position relative to the centre { x, y }

xpos horizontal origin position relative to the centre

ypos vertical origin position relative to the centre

Example

```
image = loadImage( "Untied Games/Enemy A", false )  
enemy = createSprite()  
setSpriteImage( enemy, image )  
setSpriteAnimation( enemy, 0, 4, 20 )  
setSpriteLocation( enemy, { 0, 0 } )  
size = getSpriteSize( enemy )  
setSpriteScale( enemy, { 8, 8 } )  
  
loop  
  clear()  
  origin = getSpriteOrigin( enemy )  
  printAt( 20, 10, "Sprite origin x = ", origin.x, " y = ", origin.y )  
  printAt( 20, 11, "Press A to move origin to the top left" )  
  printAt( 20, 12, "Press B to move origin to the centre" )  
  c = controls( 0 )  
  if c.a then  
    setSpriteOrigin( enemy, { -size.x / 2, -size.y / 2 } )  
  endif  
  if c.b then  
    setSpriteOrigin( enemy, { 0, 0 } )  
  endif
```

```
updateSprites()  
drawSprites()  
update()  
repeat
```



Associated Commands

`getSpriteLocation()`, `getSpriteOrigin()`, `setSpriteLocation()`

COMMAND REFERENCE

setSpriteRotation()

Purpose

Rotate a sprite

Description

Set the rotation angle of a sprite

Syntax

```
setSpriteRotation( sprite, angle )  
  
sprite.rotation = angle
```

Arguments

sprite handle of the created sprite

rotation angle in default units

Example

```
image = loadImage( "Untied Games/Enemy small top C", false )  
ship = createSprite()  
setSpriteImage( ship, image )  
lastpos = { gWidth() / 2, gHeight() / 2 }  
setSpriteLocation( ship, lastpos )  
setSpriteScale( ship, { 8, 8 } )  
  
for angle = 0 to 360 loop  
  clear()  
  setSpriteRotation( ship, angle )  
  drawSprites()  
  update()  
repeat
```



Associated Commands

`getSpriteRotation()`, `getSpriteRotationSpeed()`, `setSpriteRotationSpeed()`

COMMAND REFERENCE

setSpriteRotationSpeed()

Purpose

Set a sprites rotation speed

Description

Set a sprites speed of rotation. This is the amount that the sprite rotation angle is changed by when updatesprites is called

Syntax

```
setSpriteRotationSpeed( sprite, rotationSpeed )  
  
sprite.rotation_speed = rotationSpeed
```

Arguments

sprite handle of the created sprite

*rotationSpeed amount to add to the rotation angle of the sprite at each updatesprites call

Example

```
radians( true )  
image = loadImage( "Untied Games/Enemy small top C", false )  
ship = createSprite()  
setSpriteImage( ship, image )  
lastpos = { gWidth() / 2, gHeight() / 2 }  
setSpriteLocation( ship, lastpos )  
setSpriteScale( ship, { 10, 10 } )  
rv = 0  
maxrv = 30 // max rotation speed  
accrv = 0.3 // acceleration  
  
loop  
  clear()  
  c = controls( 0 )  
  printAt( 0, 0, "Use left joystick left or right to change rotation speed" )  
  rv = rv + c.lx * accrv  
  if abs( rv ) > maxrv then  
    rv = rv / abs( rv ) * maxrv  
  endif  
  setSpriteRotationSpeed( ship, rv )  
  updateSprites()  
  drawSprites()  
  update()  
repeat
```



Associated Commands

`getSpriteRotation()`, `getSpriteRotationSpeed()`, `setSpriteRotation()`

COMMAND REFERENCE

setScale()

Purpose

Scale a sprite

Description

Scale a sprite up or down in the horizontal and vertical axis

Syntax

```
setScale( sprite, scale )  
  
setScale( sprite, { xScale, yScale } )  
  
sprite.xscale = xScale; sprite.yscale = yScale
```

Arguments

sprite handle of the created sprite

scale amount to scale the sprite in the horizontal and vertical axes { x, y }

xScale amount to scale the sprite in the horizontal axis

yScale amount to scale the sprite in the vertical axis

Example

```
image = loadImage( "Untied Games/Enemy small top C", false )  
ship = createSprite()  
setSpriteImage( ship, image )  
lastpos = { gWidth() / 2, gHeight() / 2 }  
setSpriteLocation( ship, lastpos )  
setScale(ship, { 5, 5 } )  
  
loop  
  clear()  
  c = controls( 0 )  
  printAt( 0, 0, "Use left joystick to resize sprite" )  
  setSpriteScale( ship, { 10 * abs( c.lx ) + 5, 10 * abs( c.ly ) + 5 } )  
  updateSprites()  
  drawSprites()  
  update()  
repeat
```



Associated Commands

`getSpriteScale()`, `getSpriteScaleSpeed()`, `getSpriteSize()`, `setSpriteScaleSpeed()`

COMMAND REFERENCE

setSpriteScaleSpeed()

Purpose

Set a sprites scale speeds

Description

Set a sprites scaling speeds. These are the amount that the sprite scale is changed by in the horizontal and vertical axes when updatesprites is called

Syntax

```
setSpriteScaleSpeed( sprite, scaleSpeed )  
setSpriteScaleSpeed( sprite, xScaleSpeed, yScaleSpeed )  
sprite.xscale_speed = xScaleSpeed; sprite.yscale_speed = yScaleSpeed
```

Arguments

sprite handle of the created sprite

scaleSpeed amount to add to the horizontal and vertical scale of the sprite at each updatesprites call { x, y }

xScaleSpeed amount to add to the horizontal scale of the sprite at each updatesprites call

yscaleSpeed amount to add to the vertical scale of the sprite at each updatesprites call

Example

```
image = loadImage( "Untied Games/Enemy small top C", false )  
ship = createSprite()  
setSpriteImage( ship, image )  
lastpos = { gWidth() / 2, gHeight() / 2 }  
setSpriteLocation( ship, lastpos )  
setSpriteScale( ship, { 1, 1 } )  
sv = 0  
maxsv = 30 // max scale speed  
accsv = 0.6 // acceleration  
  
loop  
  clear()  
  c = controls( 0 )  
  printAt( 0, 0, "Use left joystick left or right to change scale speed" )  
  sv = sv + c.lx * accsv  
  if abs( sv ) > maxsv then  
    sv = sv / abs( sv ) * maxsv  
  endIf
```

```
setSpriteScaleSpeed( ship, { sv, sv } )  
updateSprites()  
drawSprites()  
update()  
repeat
```



Associated Commands

`getSpriteScale()`, `getSpriteScaleSpeed()`, `setSpriteScale()`

COMMAND REFERENCE

setSpriteSpeed()

Purpose

Set a sprites speed

Description

Set a sprites horizontal and vertical speed. This is the amount that the sprite is moved by in each axis when updatesprites is called

Syntax

```
setSpriteSpeed( sprite, speed )  
  
setSpriteSpeed( sprite, xspeed, yspeed )  
  
sprite.x_speed = xspeed; sprite.y_speed = yspeed
```

Arguments

sprite handle of the created sprite

speed amount to add to the x and y positions of the sprite at each updatesprites call { x, y }

xspeed amount to add to the x position of the sprite at each updatesprites call

yspeed amount to add to the y position of the sprite at each updatesprites call

Example

```
radians( true )  
image = loadImage( "Untied Games/Enemy small top C", false )  
ship = createSprite()  
setSpriteImage( ship, image )  
lastpos = { gWidth() / 2, gHeight() / 2 }  
setSpriteLocation( ship, lastpos )  
setSpriteScale( ship, { 8, 8 } )  
  
loop  
  clear()  
  c = controls( 0 )  
  printAt( 0, 0, "Use left joystick to control sprite" )  
  setSpriteSpeed( ship, { 600 * c.lx, -600 * c.ly } )  
  curpos = getSpriteLocation( ship )  
  if curpos != lastpos then  
    setSpriteRotation( ship, -pi / 2 + atan2( curpos.y - lastpos.y, curpos.x - lastpos.x ) )  
    lastpos = curpos  
  endif  
  updateSprites()  
  drawSprites()  
  update()  
repeat
```



Associated Commands

`getSpriteSpeed()`

COMMAND REFERENCE

setSpriteText()

Purpose

Create a text sprite

Description

Create a sprite containing text in the specified font size and colour

Syntax

```
setSpriteText( sprite, fontsize, tint, arguments )
```

Arguments

sprite handle of the sprite

fontsize fontsize of the text

tint colour of the text

arguments comma separated list of text arguments

Example

```
image = loadImage( "Untied Games/Enemy A", false )
sprite = []
for i = 0 to 4 loop
    sprite[i] = createSprite()
    setSpriteText( sprite[i], 50, red, "Sprite ", i )
repeat
loop
    clear()
    printAt( 0, 0, "Press buttons X, A, Y and B to draw sprites")
    updateSprites( )
    c = controls(0)
    if c.x then
        drawSprite( sprite[0] )
    endif
    if c.a then
        drawSprite( sprite[1] )
    endif
    if c.b then
        drawSprite( sprite[2] )
    endif
    if c.y then
        drawSprite( sprite[3] )
    endif
```

```
update()  
repeat
```

Associated Commands

`createSprite()`, `setSpriteImage()`

COMMAND REFERENCE

setSpriteVisibility()

Purpose

Hide or reveal a sprite

Description

Sets the visibility of a sprite. If false then the sprite is not displayed

Syntax

```
setSpriteVisibility( sprite, visibility )  
  
sprite.visible = show
```

Arguments

sprite handle of the sprite

visibility if true then the sprite is displayed

Example

```
image = loadImage( "Untied Games/Enemy A", false )  
enemy = createsprite()  
setSpriteImage( enemy, image )  
speed = 20  
setSpriteAnimation( enemy, 0, 4, speed )  
lastpos = { gWidth() / 2, gHeight() / 2 }  
setSpriteLocation( enemy, lastpos )  
setSpriteScale(enemy, { 8, 8 } )  
  
loop  
  clear()  
  c = controls( 0 )  
  printAt( 0, 0, "Press the A button to show sprite" )  
  setSpriteVisibility( enemy, c.a )  
  updateSprites()  
  drawSprites()  
  update()  
repeat
```



Associated Commands

`getSpriteVisibility()`

COMMAND REFERENCE

setVertex()

Purpose

Sets the screen position of a vertex (point) in a shape

Description

Used to change the on-screen position of a vertex in a shape drawn with `drawShape()`

Syntax

```
setVertex( shape, vertex, position )
```

Arguments

shape Handle which stores the shape in question

position Vector describing the desired x and y position of the supplied vertex

vertex Float index of the desired vertex (begins at 0, clockwise)

Example

Associated Commands

`copyShape()`, `createBox()`, `createCircle()`, `createCurve()`, `createLine()`, `createLineStrip()`, `createPoly()`, `createStar()`, `createTriangle()`, `deleteShape()`, `drawShape()`, `getShapeBounds()`, `getShapeLocation()`, `getShapeRotation()`, `getShapeScale()`, `getShapeTint()`, `getVertex()`, `getVertexColour()`, `getVertexLineColour()`, `getVertexLineThickness()`, `joinShapes()`, `moveShape()`, `numVerts()`, `rotateShape()`, `scaleShape()`, `setShapeColour()`, `setShapeLineStyle()`, `setShapeRotation()`, `setShapeScale()`, `setShapeScaleModeLocal()`, `setShapeTint()`, `setVertexColour()`, `setVertexLineStyle()`

COMMAND REFERENCE

setVertexColour()

Purpose

Sets the vertex colour of a shape drawn with `drawShape()`

Description

Used to adjust the tint (colour) of a supplied vertex in a shape. Creates a gradient toward other vertex colours

Syntax

```
setVertexColour( shape, vertex, colour )
```

Arguments

shape Handle of the shape in question

vertex Number of the desired vertex (0 - N)

colour Vector (RGBA) colour of the desired vertex

Example

```
// draw a multicoloured rectangle on the screen
box1 = createBox( gwidth() / 2, gheight() / 2, gwidth(), gheight() )

setVertexColour( box1, 0, bisque )
setVertexColour( box1, 1, cyan )
setVertexColour( box1, 2, fuzeblue )
setVertexColour( box1, 3, fuzepink )

drawShape( box1 )
update()
sleep( 3 )
```



Associated Commands

`createLine()`, `createLineStrip()`, `createCurve()`, `createCircle()`, `createTriangle()`, `createPoly()`,
`createStar()`, `createBox()`, `copyShape()`, `deleteShape()`, `drawShape()`, `joinShapes()`, `moveShape()`,
`getShapeBounds()`, `getShapeLocation()`, `getShapeRotation()`, `setShapeRotation()`, `rotateShape()`,
`getShapeScale()`, `setShapeScale()`, `scaleShape()`, `getShapeTint()`, `setShapeTint()`, `numVerts()`,
`getVertex()`, `setVertex()`, `getVertexColour()`, `getVertexLineThickness()`, `setVertexLineStyle()`,
`setShapeColour()`, `setShapeLineStyle()`, `setShapeScaleModeLocal()`

COMMAND REFERENCE

setVertexLineStyle()

Purpose

Set the draw style for a vertex line

Description

Used to change the colour and thickness of a vertex line in a shape drawn with `drawShape()`

Syntax

```
setVertexLineStyle( shape, vertex, thickness, tint )
```

Arguments

shape Handle which stores the shape in question

vertex Integer index of the desired vertex (begins at 0, clockwise)

thickness Float describing the thickness (in pixels) of the line through the supplied vertex

tint Vector (RGBA) to set the colour of the line through the supplied vertex

Example

Associated Commands

`copyShape()`, `createBox()`, `createCircle()`, `createCurve()`, `createLine()`, `createLineStrip()`, `createPoly()`, `createStar()`, `createTriangle()`, `deleteShape()`, `drawShape()`, `getShapeBounds()`, `getShapeLocation()`, `getShapeRotation()`, `getShapeScale()`, `getShapeTint()`, `getVertex()`, `getVertexColour()`, `getVertexLineColour()`, `getVertexLineThickness()`, `joinShapes()`, `moveShape()`, `numVerts()`, `rotateShape()`, `scaleShape()`, `setShapeColour()`, `setShapeLineStyle()`, `setShapeRotation()`, `setShapeScale()`, `setShapeScaleModeLocal()`, `setShapeTint()`, `setVertex()`, `setVertexColour()`

COMMAND REFERENCE

setView()

Purpose

Set custom screen co-ordinates for drawing

Description

Allows user to input their own screen co-ordinates without changing screen resolution.

Syntax

```
setView( left, top, right, bottom )
```

Arguments

left value for the left hand side of the view

top value for the top of the view

right value for the right hand side of the view

bottom value for the bottom of the view

Example

```
image = loadImage( "Untied Games/Enemy small top C", false )  
  
loop  
  clear()  
  setView( 0, 0, imageSize( image ).x, imageSize( image ).y )  
  // Image will fill the view  
  drawImage( image, 0, 0 )  
  update()  
repeat
```

Associated Commands

COMMAND REFERENCE

tileSize()

Purpose

Get the size of a tile in a tiled image

Description

Get the size of the specified tile in the specified image

Syntax

```
size = tileSize( image, tile )
```

Arguments

image handle of the image

tile tile number to find the size of (zero based)

size size of the specified tile { x, y }

Example

```
roll = 0
clear()
image = loadImage( "Colin Brown/Dice", false )
size = tileSize( image, 0 )
for i = 1 to 10 loop
    clear()
    roll = random( 6 ) + 1
    x = size.x - ( size.x * ( roll % 2 ) )
    y = size.y * ( ceil( roll / 2 ) - 1 )
    drawImage( image, { x, y, size.x, size.y }, { 0, 0, size.x, size.y } )
    update()
    sleep( 0.3 )
repeat
printAt( 0, 15, "You rolled a ", roll )
update()
sleep( 3 )
```

Associated Commands

COMMAND REFERENCE

triangle()

Purpose

Draw a triangle

Description

Draws a filled or outline triangle with vertices at the given points and in the specified colour.

Syntax

```
triangle( point1, point2, point3, colour, outline )
```

Arguments

point1 screen coordinates of first point in pixels

point2 screen coordinates of second point in pixels

point3 screen coordinates of second point in pixels

colour colour name or RGB values { red, green, blue, opacity } between 0 and 1

outline If true then only the outline is drawn otherwise the shape is filled.

Example

```
// Draw 100 random triangles
clear()
for i = 1 to 100 loop
  // Pick random colour
  col = { random( 101 ) / 100, random( 101 ) / 100, random( 101 ) / 100, random( 101 ) / 100 }
  point1 = { random( gWidth() ), random( gHeight() ) }
  point2 = { random( gWidth() ), random( gHeight() ) }
  point3 = { random( gWidth() ), random( gHeight() ) }
  outline = random( 2 )
  triangle( point1, point2, point3, col, outline )
  update()
repeat

for i = 1 to 100 loop
  update()
repeat
```



Associated Commands

`box()`, `circle()`, `line()`

COMMAND REFERENCE

unloadMap()

Purpose

Unload the current tile map

Description

Unload the current tile map so that another one can be loaded

Syntax

.....

Arguments

Example

```
// To view this map demo, please load the project "Map Commands Demo" from FUZE Programs.
// Maps must be stored in the project you wish to load them into.

maps = [
    "map1",
    "map2"
]

m = 0
layer = false

press = [
    .a = false,
    .up = false
]

loadMap( maps[m] )
setSpriteCamera( 0, 0, 2 )

loop
    centerSpriteCamera( 0, 0 )
    clear()

    c = controls( 0 )

    if !c.a then
        press.a = false
    endif
    if !c.up then
        press.up = false
    endif

    drawMapLayer( 0 )
```

```

if layer then
    drawMapLayer( 1 )
endIf

if c.up and !press.up then
    press.up = true
    layer = !layer
endIf

if c.a and !press.a then
    press.a = true
    unloadMap()
    m += 1
    if m >= 2 then
        m = 0
    endIf
    loadMap( maps[m] )
endIf

printAt( 0, 0, "Press A button to swap between maps" )
printAt( 0, 2, "Currently viewing: " + maps[m] )
printAt( 0, 4, "Press Up directional button to toggle additional layers" )

update()
repeat

```

Associated Commands

`drawMap()`, `drawMapLayer()`, `loadMap()`

COMMAND REFERENCE

updateSprite()

Purpose

Process speed additions for a sprite

Description

Adds the current speed values to the position, rotation and colour the sprites causing it to move, rotate or change colour when drawn.

Syntax

```
updateSprite( sprite )  
updateSprite( sprite, deltatime )
```

Arguments

sprite handle of the sprite

deltatime time difference between the current frame and the previous frame, in seconds

Example

```
image = loadImage( "Untied Games/Enemy A", false )  
enemy = []  
for i = 0 to 4 loop  
    enemy[i] = createSprite()  
    setSpriteImage( enemy[i], image )  
    setSpriteAnimation( enemy[i], 0, 4, 20 )  
    setSpriteLocation( enemy[i], { ( i % 2 ) * 400 + 400, int(i / 2) * 300 + 200 } )  
    setSpriteScale( enemy[i], { 4, 4 } )  
repeat  
loop  
    clear()  
    updateSprite( enemy[1] )  
    updateSprite( enemy[2], deltaTime() / 2 )  
    updateSprite( enemy[3], deltaTime() * 2 )  
    drawsprites()  
    update()  
repeat
```



Associated Commands

`createSprite()`, `deltaTime()`, `drawSprite()`, `drawSprites()`, `removeSprite()`, `updateSprites()`, `updateSprite()`

COMMAND REFERENCE

updateSprites()

Purpose

Process speed additions for all sprites

Description

Adds the current speed values to the position, rotation and colour of all sprites causing them to move, rotate or change colour when drawn.

Syntax

```
updateSprites( )  
updateSprites( sprites )  
updateSprites( deltatime )  
updateSprites( sprites, deltatime )
```

Arguments

sprites array of sprites to be updated

deltatime time difference between the current frame and the previous frame, in seconds

Example

```
image = loadImage( "Untied Games/Enemy A", false )  
enemy = []  
for i = 0 to 4 loop  
    enemy[i] = createSprite( )  
    setSpriteImage( enemy[i], image )  
    setSpriteAnimation( enemy[i], 0, 4, 20 )  
    setSpriteLocation( enemy[i], { ( i % 2 ) * 400 + 400, int( i / 2 ) * 300 + 200 } )  
    setSpriteScale( enemy[i], { 4, 4 } )  
repeat  
loop  
    clear()  
    printAt( 0, 0, "Press X for normal speed" )  
    printAt( 0, 1, "Press A for half speed" )  
    printAt( 0, 2, "Press B for double speed" )  
    c = controls( 0 )  
    if c.x then  
        updateSprites()  
    endif  
    if c.a then  
        updateSprites( enemy, deltaTime() / 2 )  
    endif  
    if c.b then
```

```
    updateSprites( enemy, deltaTime() * 2 )  
  endIf  
  drawSprites()  
  update()  
repeat
```



Associated Commands

`createSprite()`, `deltaTime()`, `drawSprite()`, `drawSprites()`, `removeSprite()`, `updateSprites()`, `updateSprite()`

COMMAND REFERENCE

uploadImage()

Purpose

Create an image in code

Description

Create an image from pixel data stored in the actual code of the program

Syntax

```
handle = uploadImage( pixeldata, width, height, filter )
```

Arguments

pixeldata An array of (width x height) colours

width Desired on-screen width in pixels

height Desired on-screen width in pixels

filter Sets filtering on or off - generally on for real images and off for pixel art

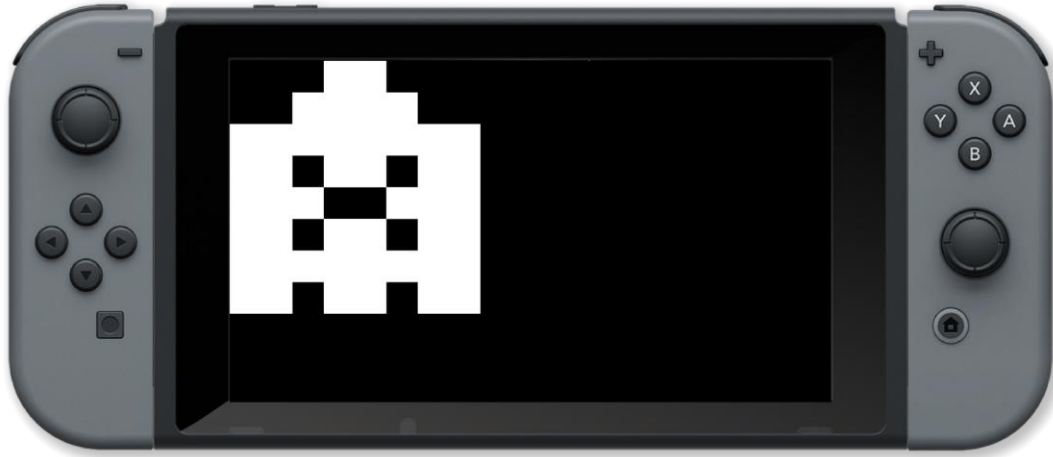
handle handle of the image

Example

```
t = { 0, 0, 0, 0 } // transparent
w = white
alien = [
  t, t, t, w, w, t, t, t,
  t, t, w, w, w, w, t, t,
  t, w, w, w, w, w, w, t,
  w, w, w, w, w, w, w, w,
  w, w, t, w, w, t, w, w,
  w, w, w, t, t, w, w, w,
  w, w, t, w, w, t, w, w,
  w, w, w, w, w, w, w, w,
  w, w, t, w, w, t, w, w
]

alienimage = uploadImage( alien, 8, 9, false )
drawImage( alienimage, 0, 0, 50 )

loop
  update()
repeat
```



Associated Commands

`clear()`, `createImage()`, `drawImage()`, `drawImageEx()`, `drawQuad()`, `drawSheet()`, `loadImage()`, `update()`

COMMAND REFERENCE

3D Graphics

COMMAND REFERENCE

animationLength()

Purpose

Find the length of a 3D animation

Description

Some 3D models contain animation sequences. This finds the length of the specified animation sequence

Syntax

```
length = animationLength( object, animation )
```

Arguments

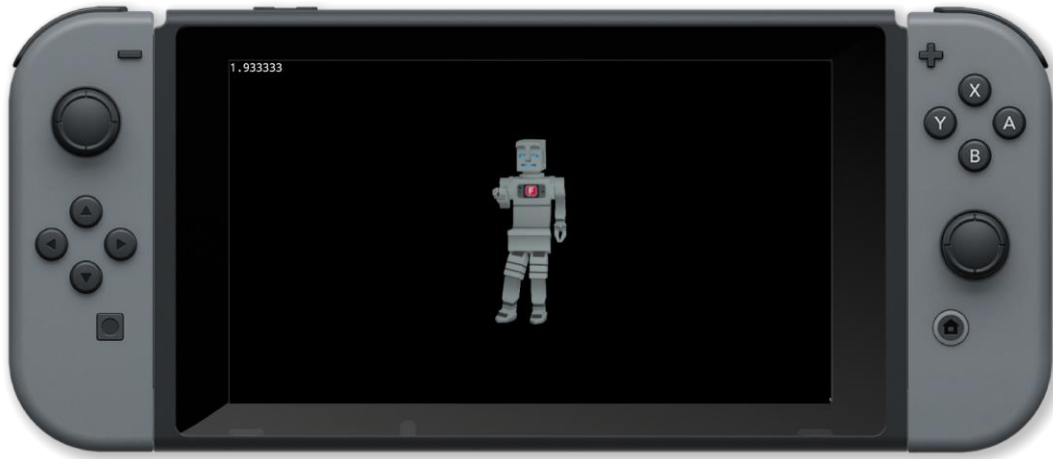
length length of animation in seconds

object handle of the animated 3D object

animation index of the animation

Example

```
cb = loadModel( "Kat/Colin" )
pointLight( { 0.5, 1.3, 2 }, white, 4 )
setAmbientLight( { 0.5, 0.5, 0.5 } )
colin = placeObject( cb, { 0, 0, 0 }, { 1, 1, 1 } )
setCamera( { 0, 10, 10 }, { 0, 5, 0 } )
animID = 7 // the robot
animLength = animationLength( colin, animID )
animFrame = 0
loop
  clear()
  animFrame = animFrame + 1/60
  if animFrame >= animLength then
    animFrame = 0
  endIf
  updateAnimation( colin, animID, animFrame )
  drawObjects()
  printAt( 0, 0, "length: ", animLength, " frame: ", animFrame )
  update()
repeat
```



Associated Commands

`numAnimations()`, `updateAnimation()`

COMMAND REFERENCE

createTerrain()

Purpose

Create a 3D terrain

Description

Create a 3D terrain by setting the height and colour of points on a grid

Syntax

```
handle = createTerrain( gridsize, filter )
```

Arguments

handle identifier of the created terrain

gridsize size of the grid (length of one side)

filter Smoothing filter (0 = no filter)

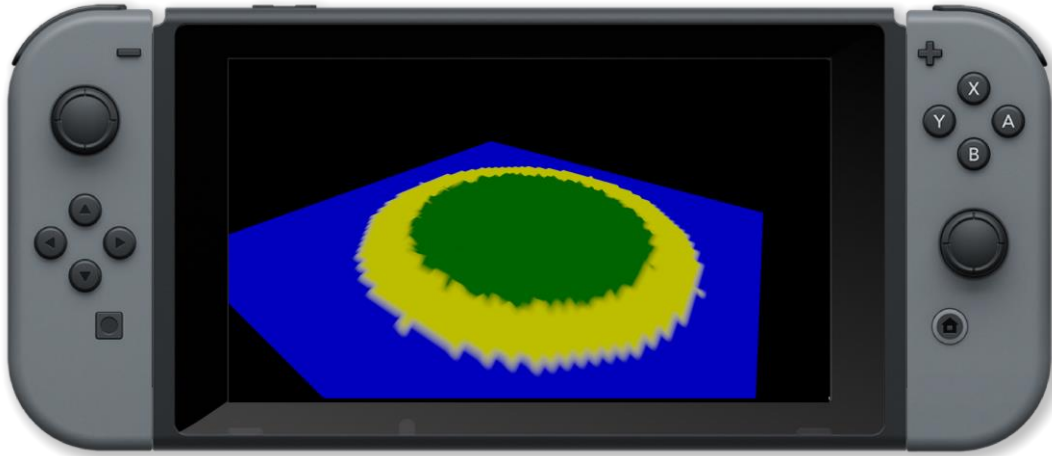
Example

```
gsize = 64
landscape = createTerrain( gsize, 1 )
height = 0
colour = white

for x = 0 to gsize loop
  for y = 0 to gsize loop
    d = distance ( { x, y }, { gsize / 2, gsize / 2 } )
    if d > 24 then // sea level
      height = 0
      colour = blue
    else
      if d > 18 then // beach
        height = 1
        colour = yellow
      else // hills
        height = rnd( 2 ) + 1
        colour = green
      endIf
    endIf
    setTerrainPoint( landscape, x, y, height, colour )
  repeat
repeat

setCamera( { gsize / 2, 50, gsize / 2 }, { gsize / 2.0, 0, gsize / 2.00001 } )
setAmbientLight( { 0.5, 0.5, 0.5 } )
```

```
island = placeObject( landscape, { gsize / 2, 0, gsize / 2 }, { 1, 1, 1 } )  
  
loop  
  c = controls( 0 ) // rotate using joysticks  
  rotateObject( island, { 1, 0, 0 }, c.ly )  
  rotateObject( island, { 0, 0, 1 }, c.lx )  
  rotateObject( island, { 0, 1, 0 }, c.rx )  
  drawobjects()  
  update()  
repeat
```



Associated Commands

`setTerrainPoint()`, `updateTerrain()`

COMMAND REFERENCE

drawObjects()

Purpose

Draw all 3D objects

Description

Draws all of the current 3D objects to the framebuffer in their current positions using the current camera position and lighting

Syntax

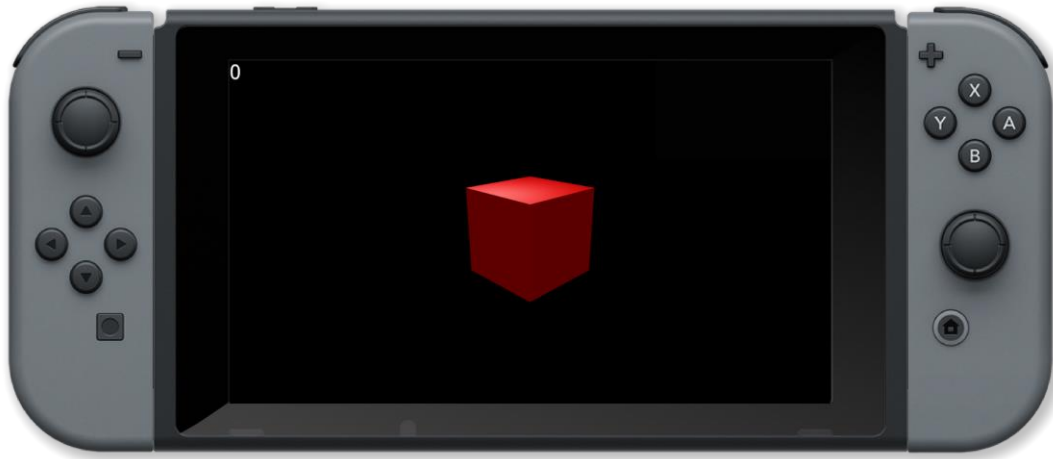
```
drawObjects( )
```

Arguments

Example

```
obj = placeObject( cube, { 0, 0, 0 }, { 2, 2, 2 } )
setObjectMaterial( obj, red, 1, 1 )
setCamera( { 5, 5, 10 }, { 0, 0, 0 } )
light = pointLight( { 0, 4, 2 }, white, 100 )
x = 0

loop
  clear()
  c = controls( 0 )
  setLightPos( light, { x, 4, 2 } )
  if c.left then
    x = x - 0.2
  endIf
  if c.right then
    x = x + 0.2
  endIf
  rotateObject( obj, { 1, 1, 1 }, 0.5 )
  drawObjects()
  printAt( 0, 0, "Use left and right directional buttons to move light source: ", x )
  update()
repeat
```

Associated Commands

`placeObject()`, `pointLight()`, `rotateObject()`, `setCamera()`, `setObjectMaterial()`

COMMAND REFERENCE

loadModel()

Purpose

Load a 3D model ready for display

Description

Load a 3D model that can then be placed in the screen buffer using placeobject and displayed using drawobjects

Syntax

```
handle = loadModel( filename )
```

Arguments

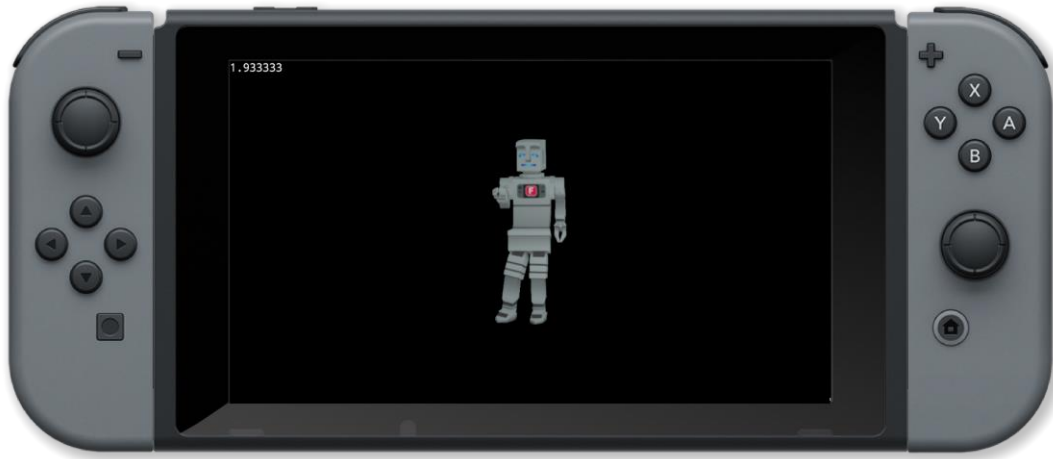
handle variable which stores the desired 3D model file

filename relative path of the 3D model to load

Example

```
cb = loadModel( "Kat/Colin" )
pointLight( { 0.5, 1.3, 2 }, white, 4 )
setAmbientLight( { 0.5, 0.5, 0.5 } )
colin = placeObject( cb, { 0, 0, 0 }, { 1, 1, 1 } )
setCamera( { 0, 10, 10 }, { 0, 5, 0 } )
animID = 6 // walking
animLength = animationLength( colin, animID )
animFrame = 0

loop
  clear()
  animframe = animframe + 1/60
  if animframe >= animlength then
    animframe = 0
  endif
  updateAnimation( colin, animID, animframe )
  drawObjects()
  printAt( 0, 0, "length: ", animlength, " frame: ", animframe )
  update()
repeat
```



Associated Commands

`drawObjects()`, `placeObject()`, `removeObject()`, `rotateObject()`, `setObjectMaterial()`

COMMAND REFERENCE

numAnimations()

Purpose

Find the length of a 3D animation

Description

Some 3D models contain animation sequences. This finds the number of animations within a given 3D model

Syntax

```
count = numAnimations( object )
```

Arguments

object handle of the animated 3D object

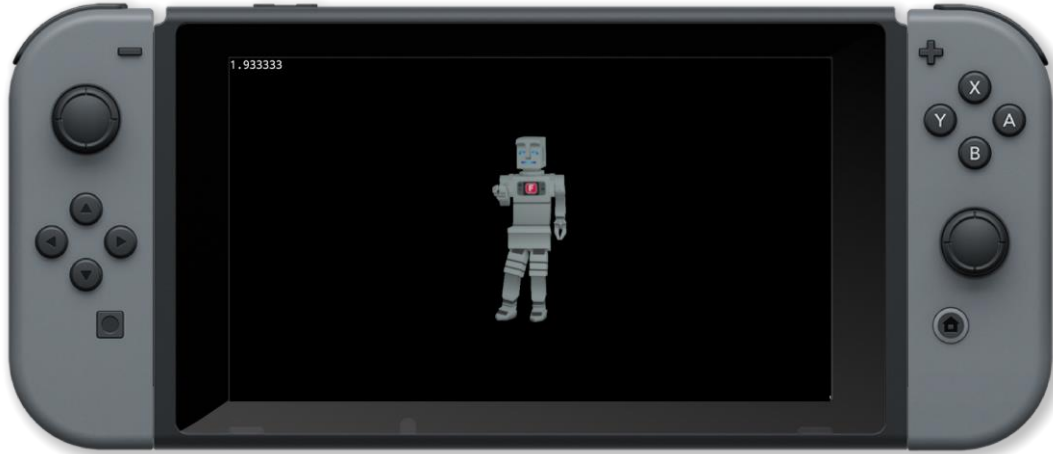
count number of animations in the model

Example

```
cb = loadModel( "Kat/Colin" )
pointLight( { 0.5, 1.3, 2 }, white, 4 )
setAmbientLight( { 0.5, 0.5, 0.5 } )
colin = placeObject( cb, { 0, 0, 0 }, { 1, 1, 1 } )
setCamera( { 0, 10, 10 }, { 0, 5, 0 } )
animcount = numAnimations( colin )
animID = 0
animframe = 0

loop
  clear()
  c = controls( 0 )
  if c.up then // up to increase animation id
    animID += 1
    sleep( 0.3 )
  endIf
  if c.down then // down to decrease animation id
    animID -= 1
    sleep( 0.3 )
  endIf
  animID = clamp( animID, 0, animcount - 1 )
  animLength = animationLength( colin, animID )
  animframe = animframe + 1 / 60
  if animframe >= animlength then
    animframe = 0
  endIf
  updateAnimation( colin, animID, animframe )
```

```
drawObjects()  
printAt( 0, 0, "Use up/down arrows to change ID: ", int( animID ) )  
printAt( 0, 1, "length: ", animlength, " frame: ", animframe )  
update()  
repeat
```



Associated Commands

`animationLength()`, `updateAnimation()`

COMMAND REFERENCE

objectPointAt()

Purpose

Set where a 3D object is pointing

Description

Change the point where a 3D object is pointing to

Syntax

```
objectPointAt( handle, point )
```

Arguments

handle variable which stores the placed 3D object

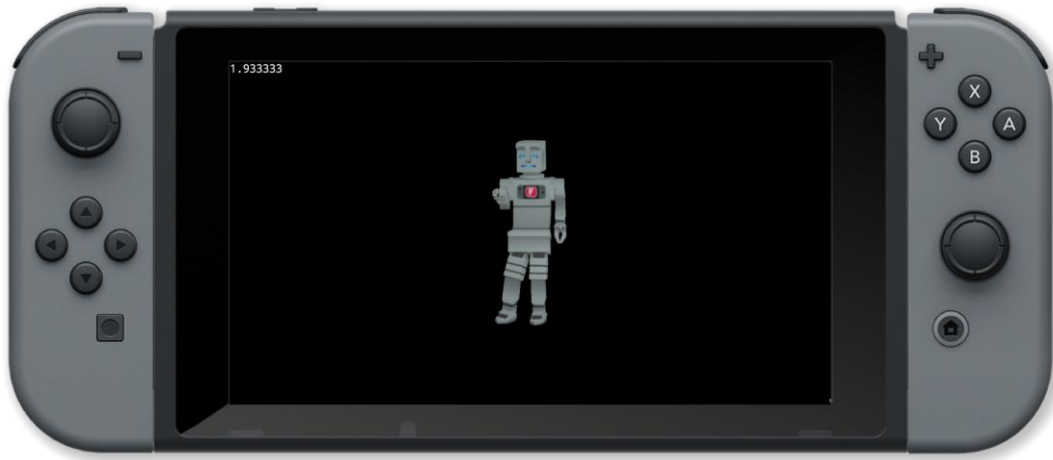
point vector containing the point in 3 dimensions { x, y, z }

Example

```
cb = loadModel( "Kat/Colin" )
pointLight( { 0.5, 1.3, 2 }, white, 4 )
setambientlight( { 0.5, 0.5, 0.5 } )
pos = { 0, 0, 0 }
scale = { 1, 1, 1 }
colin = placeObject( cb, pos, scale )
point = { 0, 10, 10 }
setcamera( point, { 0, 5, 0 } )
animID = 10
animlength = animationLength( colin, animID )
animframe = 0

loop
  clear()
  c = controls( 0 )
  if c.left then
    point.x -= 0.1
  endIf
  if c.right then
    point.x += 0.1
  endIf
  if c.up then
    point.y += 0.1
  endIf
  if c.down then
    point.y -= 0.1
  endIf
  objectPointAt( colin, point )
```

```
animframe = animframe + 1 / 60
if animframe >= animlength then
    animframe = 0
endIf
updateAnimation( colin, animID, animframe )
drawObjects()
printAt( 0, 0, "Use left joypad to move where object is pointing" )
update()
repeat
```



Associated Commands

`drawObjects()`, `loadModel()`, `objectPointAt()`, `placeObject()`, `removeObject()`, `rotateObject()`, `setObjectMaterial()`, `setObjectPos()`, `setObjectScale()`

COMMAND REFERENCE

placeObject()

Purpose

Place a 3D object

Description

Place one of the predefined 3D objects in the screen buffer at the specified location and scale

Syntax

```
handle = placeObject( object, location, scale )
```

Arguments

handle handle of the placed 3D object

object handle of the 3D object or predefined object (cube, sphere, pyramid, cone, cylinder, wedge and hemisphere)

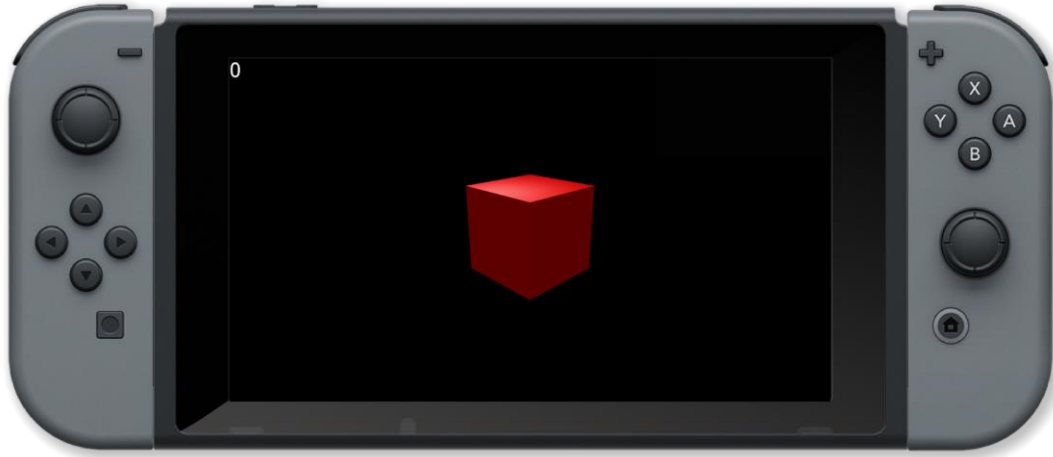
location position vector in 3 dimensional space { x, y, z }

scale scale vector in 3 dimensional space { x, y, z }

Example

```
obj = placeObject( cube, { 0, 0, 0 }, { 2, 2, 2 } )
setObjectMaterial( obj, red, 1, 1 )
setCamera( { 5, 5, 10 }, { 0, 0, 0 } )
light = pointLight( { 0, 4, 2 }, white, 100 )
x = 0

loop
  clear()
  c = controls( 0 )
  setLightPos( light, { x, 4, 2 } )
  if c.left then
    x = x - 0.2
  endIf
  if c.right then
    x = x + 0.2
  endIf
  rotateObject( obj, { 1, 1, 1 }, 0.5 )
  drawObjects()
  printat( 0, 0, "Use left and right arrows to move light source: ",x )
  update()
repeat
```

Associated Commands

`drawObjects()`, `loadModel()`, `objectPointAt()`, `removeObject()`, `rotateObject()`, `setObjectMaterial()`,
`setObjectPos()`, `setObjectScale()`

COMMAND REFERENCE

pointLight()

Purpose

Creates a pinpoint light source in 3D space

Description

Creates a pinpoint light source in the specified position of the specified colour and brightness

Syntax

```
handle = pointLight( position, colour, brightness )
```

Arguments

handle The handle of the light source

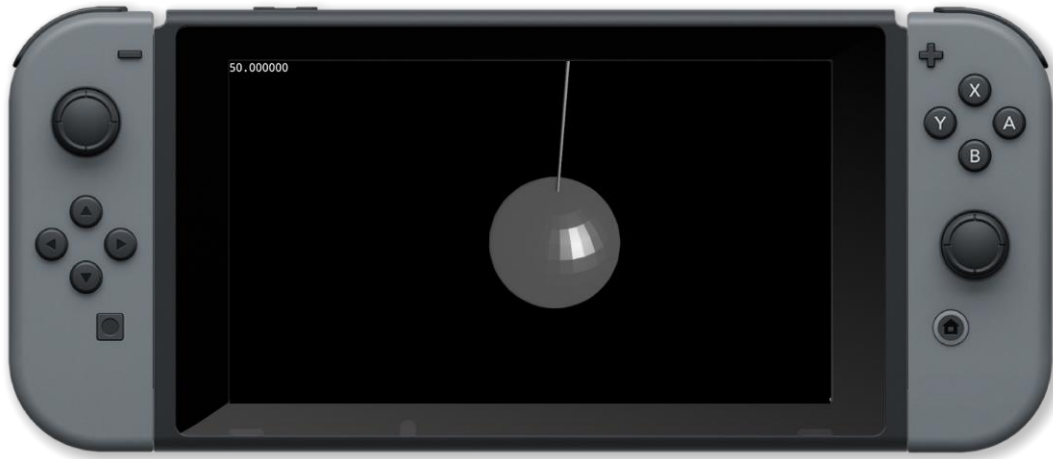
position A position vector in 3 dimensional space { x, y, z } where the light source is located

colour colour name or RGB values { red, green, blue, opacity } between 0 and 1

brightness A value to indicate the brightness of the light source

Example

```
setcamera( {0, 10, 10 }, { 0, 0, 0 } )
bright = 50
light = pointLight( { 5, 5, 5 }, red, bright )
ballModel = loadModel( "Kat/Discoball" )
ball = placeObject( ballmodel, { 0, 0, 0 }, { 10, 10, 10 } )
loop
  c = controls( 0 )
  if (c.up) then
    bright = bright + 1
  endIf
  if (c.down) then
    bright = bright - 1
  endIf
  bright = clamp( bright, 0, 100 )
  setlightbrightness( light, bright )
  rotateobject( ball, { 0, 1, 0 }, 1.0 )
  drawobjects()
  printat( 0, 0, "Use up and down arrows to adjust brightness: ", bright )
  update()
repeat
```



Associated Commands

`pointShadowLight()`, `removeLight()`, `setAmbientLight()`, `setLightBrightness()`, `setLightColour()`, `setLightDir()`, `setLightPos()`, `setLightSpread()`, `spotLight()`, `worldLight()`, `worldShadowLight()`

COMMAND REFERENCE

pointShadowLight()

Purpose

Creates a pinpoint light source in 3D space that casts a shadow

Description

Creates a pinpoint light source in the specified position of the specified colour and brightness that casts a shadow

Syntax

```
handle = pointShadowLight( position, colour, brightness )
```

Arguments

handle The handle of the light source

position A position vector in 3 dimensional space { x, y, z } where the light source is located

colour colour name or RGB values { red, green, blue, opacity } between 0 and 1

brightness A value to indicate the brightness of the light source

resolution resolution of shadows (higher is smoother)

Example

```
setCamera( { 0, 4, 10 }, { 0, 0, 0 } )

obj = [
  placeObject( cube, { 0, 0, 0 }, { 4, 0.1, 4 } )
  placeObject( cube, { 0, 2, 0 }, { 1, 1, 1 } )
]

setObjectMaterial( obj[0], bisque, 0, 1 )
setObjectMaterial( obj[1], cyan, 0, 1 )

lightpos = { 0, 6, 2 }
light = pointShadowLight( lightpos, white, 50, 1024 )

loop
  clear()
  c = controls( 0 )
  lightpos += { c.lx, c.ry, -c.ly } * 0.1
  setLightPos( light, lightpos )
  rotateObject( obj[1], { 0, 1, 0 }, 1 )
  drawObjects()
```

```
printAt( 0, 0, "Use Joy-Con Control Sticks to adjust light position" )  
printAt( 0, 2, "
```

Associated Commands

`pointLight()`, `removeLight()`, `setAmbientLight()`, `setLightBrightness()`, `setLightColour()`,
`setLightDir()`, `setLightPos()`, `setLightSpread()`, `spotLight()`, `worldLight()`, `worldShadowLight()`

COMMAND REFERENCE

removeLight()

Purpose

Remove a light source

Description

Switches off a light source. The handle is invalid after removal and should not be used

Syntax

```
removeLight( light )
```

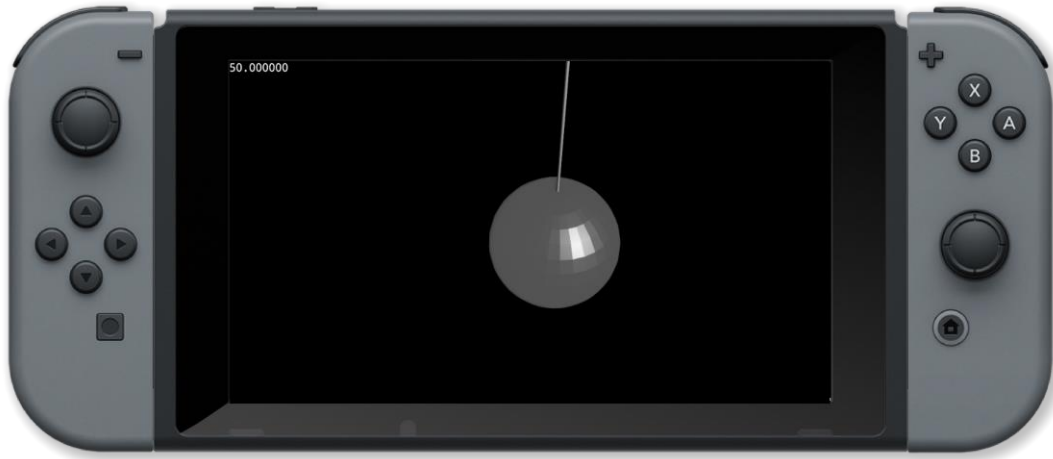
Arguments

light handle of the light source

Example

```
setCamera( {0, 10, 10 }, { 0, 0, 0 } )
bright = 50
light = worldLight( { -5, -5, -5 }, white, bright )
lighton = true
ballmodel = loadModel( "Kat/Discoball" )
ball = placeObject( ballmodel, { 0, 0, 0 }, { 10, 10, 10 } )

loop
  clear()
  c = controls( 0 )
  if c.x and !lighton then
    light = worldLight( { -5, -5, -5 }, white, bright )
    lighton = true
  endif
  if c.a and lighton then
    removeLight( light )
    lighton = false
  endif
  rotateObject( ball, { 0, 1, 0 }, 1.0 )
  drawObjects()
  printAt( 0, 0, "Press X to switch on the light" )
  printAt( 0, 1, "Press A to switch off the light" )
  update()
repeat
```



Associated Commands

`pointLight()`, `pointShadowLight()`, `setLightBrightness()`, `setLightColour()`, `setLightDir()`,
`setLightPos()`, `setLightSpread()`, `spotLight()`, `worldLight()`, `worldShadowLight()`

COMMAND REFERENCE

removeObject()

Purpose

Remove a 3D object

Description

Remove a 3D object previously placed in the screen buffer by place object. The handle becomes invalid after removal and should not be used

Syntax

```
removeObject( handle )
```

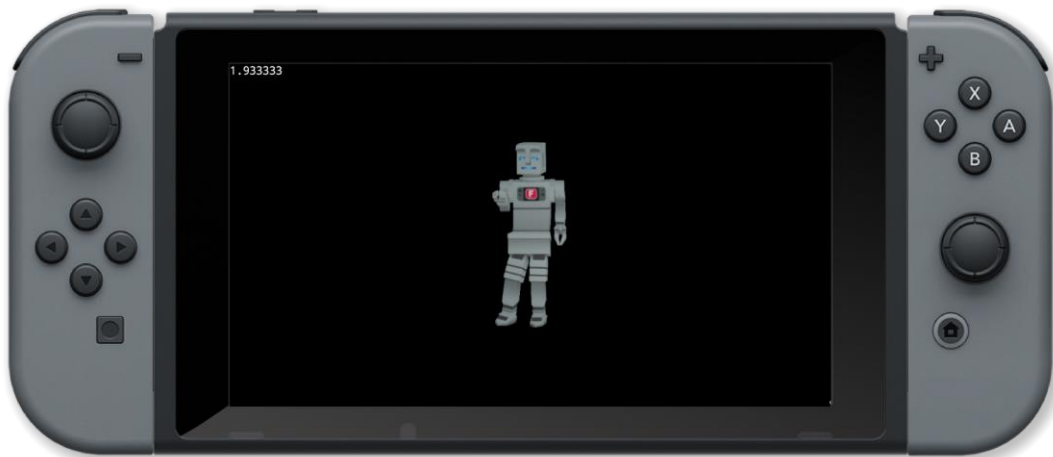
Arguments

handle variable which stores the placed 3D object

Example

```
`` cb = loadModel( "Kat/Colin" ) pointLight( { 0.5, 1.3, 2 }, white, 4 ) setAmbientLight( { 0.5, 0.5, 0.5 } ) colin = placeObject( cb, { 0, 0.295, 0 }, { 1, 1, 1 } ) placed = true setCamera( { 0, 10, 10 }, { 0, 5, 0 } ) animID = 6 animlength = animationLength( colin, animID ) animframe = 0
```

```
loop clear() c = controls( 0 ) if c.x and placed then removeObject( colin ) placed = false endIf if c.a and !placed then colin = placeObject( cb, { 0, 0.295, 0 }, { 1, 1, 1 } ) placed = true endIf if placed then animframe = animframe + 1/60 if animframe >= animlength then animframe = 0 endIf updateAnimation( colin, animID, animframe ) endIf drawObjects() printAt( 0, 0, "Press X to remove" ) printAt( 0, 1, "Press A to place" ) printAt( 0, 2, "Frame:", animframe ) update() repeat``
```



Associated Commands

`drawObjects()`, `loadModel()`, `objectPointAt()`, `placeObject()`, `rotateObject()`, `setObjectMaterial()`,
`setObjectPos()`, `setObjectScale()`

COMMAND REFERENCE

rotateObject()

Purpose

Rotate a 3D object

Description

Rotate a 3D object through the specified axes and by the specified amount

Syntax

```
rotateObject( handle, axes, amount )
```

Arguments

handle The handle of the placed 3D object

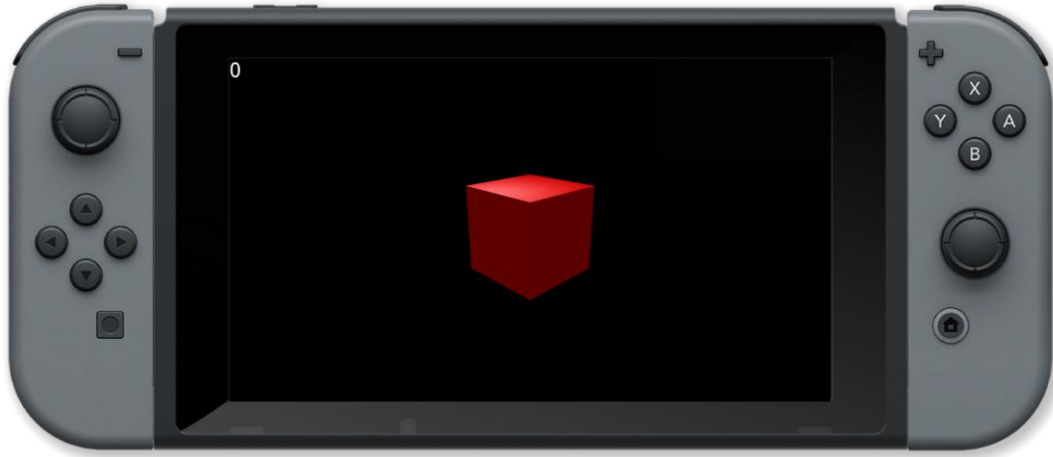
axes vector which describes the axes of rotation { x, y, z } e.g. { 1, 0, 0 } rotates in the x axes only

amount Amount to rotate the object in degrees(0 to 360)

Example

```
obj = placeObject( cube, { 0, 0, 0 }, { 2, 2, 2 } )
setObjectMaterial( obj, red, 1, 1 )
setCamera( { 5, 5, 10 }, { 0, 0, 0 } )
light = pointLight( { 0, 4, 2 }, white, 100 )
x = 0

loop
  clear()
  c = controls( 0 )
  setLightPos( light, { x, 4, 2 } )
  if c.left then
    x = x - 0.2
  endIf
  if c.right then
    x = x + 0.2
  endIf
  rotateObject( obj, { 1, 1, 1 }, 0.5 )
  drawObjects()
  printAt( 0, 0, "Use left and right arrows to move light source: ",x )
  update()
repeat
```



Associated Commands

`drawObjects()`, `loadModel()`, `objectPointAt()`, `placeObject()`, `removeObject()`, `setObjectMaterial()`, `setObjectPos()`, `setObjectScale()`

COMMAND REFERENCE

setAmbientLight()

Purpose

Set the ambient light level

Description

Set the background light levels for red, green and blue

Syntax

```
setAmbientLight( colour )
```

Arguments

colour brightness vector for ambient light { red, green , blue }

Example

```
obj1 = placeObject( cube, { -3, 0, 0 }, { 1, 1, 1 } )
obj2 = placeObject( cube, { 3, 0, 0 }, { 1, 1, 1 } )
setObjectMaterial( obj1, white, 0, 0.05 ) // white, smooth, shiny
setObjectMaterial( obj2, white, 0, 0.05 ) // white, smooth, shiny
floor = placeObject( cube, { 0, -2, 0 }, { 10, 0.05, 10 } )
setObjectMaterial( floor, grey, 0, 1 ) // grey, smooth, not shiny
setCamera( { 1, 0.5, 5 }, { 0, 0, 0 } ) // back a bit and off centre, facing world centre
setAmbientLight( { 0.5, 0.2, 0.6 } ) // light purple
```

loop

```
rotateObject( obj1, { 1, 1, 1 }, 1 ) // in all directions by 1 degree
rotateObject( obj2, { 1, 1, 1 }, -1 ) // in all directions by -1 degree
drawObjects()
update()
```

repeat



Associated Commands

`pointLight()`, `pointShadowLight()`, `removeLight()`, `setLightBrightness()`, `setLightColour()`,
`setLightDir()`, `setLightPos()`, `setLightSpread()`, `spotLight()`, `worldLight()`, `worldShadowLight()`

COMMAND REFERENCE

setCamera()

Purpose

Set the position of the camera in 3D space

Description

Positions the camera in 3D space and where it is pointing to

Syntax

```
setCamera( location, target )
```

Arguments

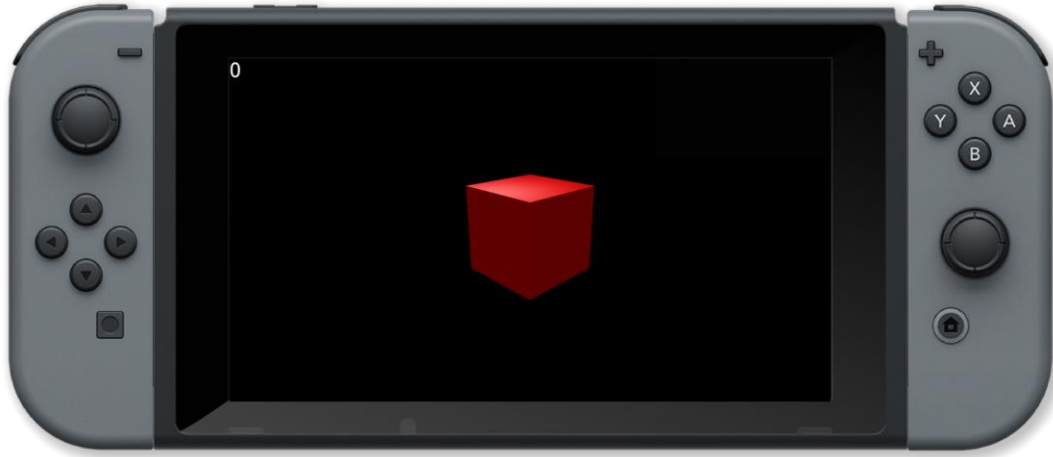
location A position vector in 3 dimensional space { x, y, z } where the camera is located

target A position vector in 3 dimensional space { x, y, z } where the camera is pointing

Example

```
obj = placeObject( cube, { 0, 0, 0 }, { 2, 2, 2 } )
setObjectMaterial( obj, red, 1, 1 )
x = 5
y = 5
setCamera( { x, y, 10 }, { 0, 0, 0 } )
light = pointLight( { 0, 4, 2 }, white, 100 )

loop
  clear()
  c = controls( 0 )
  setCamera( { x, y, 10 }, { 0, 0, 0 } )
  if c.left then
    x -= 0.2
  endif
  if c.right then
    x += 0.2
  endif
  if c.up then
    y += 0.2
  endif
  if c.down then
    y -= 0.2
  endif
  drawObjects()
  printAt( 0, 0, "Use arrows to move camera" )
  printAt( 0, 1, "x = ", x, " y = ", y )
  update()
repeat
```



Associated Commands

`drawObjects()`, `placeObject()`, `pointLight()`, `rotateObject()`, `setObjectMaterial()`

COMMAND REFERENCE

setFov()

Purpose

Set the camera field of view in 3D space

Description

Sets the angular extent of the observable 3D space

Syntax

```
setFov( angle )
```

Arguments

angle angle of the extent of the observable 3D space

Example

```
obj1 = placeObject( cube, { -3, 0, 0 }, { 1, 1, 1 } )
obj2 = placeObject( cube, { 3, 0, 0 }, { 1, 1, 1 } )
setObjectMaterial( obj1, white, 0, 0.05 ) // white, smooth, shiny
setObjectMaterial( obj2, white, 0, 0.05 ) // white, smooth, shiny
floor = placeObject( cube, { 0, -2, 0 }, { 10, 0.05, 10 } )
setObjectMaterial( floor, grey, 0, 1 ) // grey, smooth, not shiny
setCamera( { 1, 0.5, 5 }, { 0, 0, 0 } ) // back a bit and off centre, facing world centre
fov = 60
worldLight( { -1, -0.5, -0.5 }, white, 1 )
loop
  c = controls( 0 )
  if c.up then
    fov += 0.5
  endIf
  if c.down then
    fov -= 0.5
  endIf
  fov = clamp( fov, 30, 90 )
  setFov( fov )
  rotateObject( obj1, { 1, 1, 1 }, 1 ) // in all directions by 1 degree
  rotateObject( obj2, { 1, 1, 1 }, -1 ) // in all directions by -1 degree
  drawObjects()
  printAt( 0, 0, "Move left joyypad up or down to adjust fov: ", fov )
  update()
repeat
```

Associated Commands

[setCamera\(\)](#)

COMMAND REFERENCE

setLightBrightness()

Purpose

Set the brightness of a light source

Description

Set the brightness of a light source to the specified value

Syntax

```
setLightBrightness( light, brightness )
```

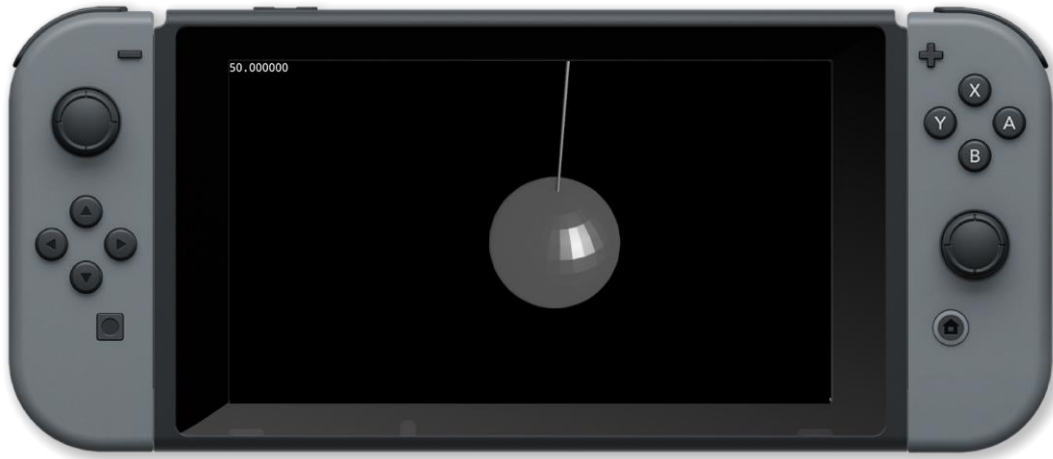
Arguments

light handle of the light source

brightness brightness of the light source (0 - 100)

Example

```
setCamera( { 0, 10, 10 }, { 0, 0, 0 } )
bright = 50
light = worldLight( { -1, -1, -1 }, white, bright )
ballmodel = loadModel( "Kat/Discoball" )
ball = placeObject( ballmodel, { 0, -0, 0 }, { 10, 10, 10 } )
loop
  c = controls( 0 )
  if c.up then
    bright += 1
  endIf
  if c.down then
    bright -= 1
  endIf
  bright = clamp( bright, 0, 100 )
  setLightBrightness( light, bright )
  rotateObject( ball, { 0, 1, 0 }, 1.0 )
  drawObjects()
  printAt( 0, 0, "Use up and down arrows to adjust brightness: ", bright )
  update()
repeat
```



Associated Commands

`pointLight()`, `pointShadowLight()`, `removeLight()`, `setAmbientLight()`, `setLightColour()`,
`setLightDir()`, `setLightPos()`, `setLightSpread()`, `spotLight()`, `worldLight()`, `worldShadowLight()`

COMMAND REFERENCE

setLightColour()

Purpose

Set the colour of a light source

Description

Set the colour of a light source to the specified value

Syntax

```
setLightColour( light, colour )
```

Arguments

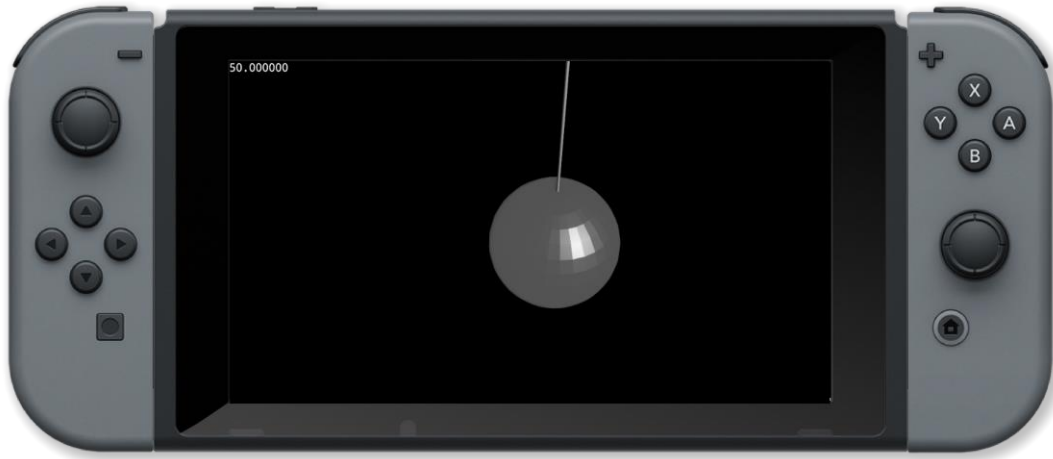
light handle of the light source

colour colour name or RGB values { red, green, blue, opacity } between 0 and 1

Example

```
setCamera( {0, 10, 10 }, { 0, 0, 0 } )
bright = 50
col = white
light = worldLight( { -5, -5, -5 }, col, bright )
ballmodel = loadModel( "Kat/Discoball" )
ball = placeObject( ballmodel, { 0, 0, 0 }, { 10, 10, 10 } )

loop
  clear()
  c = controls( 0 )
  if c.x then
    col = red
  endif
  if c.a then
    col = green
  endif
  if c.b then
    col = blue
  endif
  setLightColour( light, col )
  rotateObject( ball, { 0, 1, 0 }, 1.0 )
  drawObjects()
  printAt( 0, 0, "Press X for red light" )
  printAt( 0, 1, "Press A for green light" )
  printAt( 0, 2, "Press B for blue light" )
  update()
repeat
```



Associated Commands

`pointLight()`, `pointShadowLight()`, `removeLight()`, `setAmbientLight()`, `setLightBrightness()`,
`setLightDir()`, `setLightPos()`, `setLightSpread()`, `spotLight()`, `worldLight()`, `worldShadowLight()`

COMMAND REFERENCE

setLightDir()

Purpose

Set the direction of a light source

Description

Set the direction of a light source to the specified value

Syntax

```
setLightDir( light, direction )
```

Arguments

light handle of the light source

direction direction the light source is pointing { x, y, z }

Example

```
setCamera( { 0, 6, 10 }, { 0, 0, 0 } )
setAmbientLight( { 0.1, 0.1, 0.1 } )
lightDir = { 0, -1, -1 }
light = spotLight( { 0, 4, 4 }, lightDir, white, 50, 2 )

obj = [
  placeObject( cube, { 0, 0, 0 }, { 4, 0.1, 4 } ),
  placeObject( cube, { 0, 1.1, 0 }, { 1, 1, 1 } )
]

setObjectMaterial( obj[0], white, 0, 1 )
setObjectMaterial( obj[1], cyan, 0, 1 )

loop
  c = controls( 0 )
  lightDir += { c.lx, c.ry, -c.ly } * 0.1
  setLightDir( light, lightDir )
  rotateObject( obj[1], { 0, 1, 0 }, 1.0 )
  drawObjects()
  printAt( 0, 0, "Use Joy-Con control sticks to adjust light direction" )
  printAt( 0, 2, "Left Control stick left and right adjusts the X direction: " + lightDir.x )
  printAt( 0, 3, "Left Control stick up and down adjusts the Z direction: " + lightDir.z )
  printAt( 0, 4, "Right Control stick up and down adjusts the Y direction: " + lightDir.y )
  update()
repeat
```

Associated Commands

pointLight(), pointShadowLight(), removeLight(), setAmbientLight(), setLightBrightness(), setLightColour(), setLightPos(), setLightSpread(), spotLight(), worldLight(), worldShadowLight()

COMMAND REFERENCE

setLightPos()

Purpose

Set the position of a light source

Description

Set the position of a light source to the specified value

Syntax

```
setLightPos( light, position )
```

Arguments

light handle of the light source

position A position vector in 3 dimensional space { x, y, z } where the light source is located

Example

```
obj = placeObject( cube, { 0, 0, 0 }, { 4, 0.1, 4 } )
lightpos = { 0, 0, 0 }
light = pointLight( lightpos, white, 50 )
setCamera( { 0, 4, 8 }, { 0, 0, 0 } )

loop
  clear()
  c = controls( 0 )
  // Move the light position using the joysticks
  lightpos += { c.lx, c.ry, -c.ly } * 0.1
  // Restrict height control of light position
  lightpos.y = clamp( lightPos.y, 0.1, 10 )
  setLightPos( light, lightPos )
  drawObjects()
  printAt( 0, 0, "Use Joy-Con Control Sticks to adjust light position" )
  printAt( 0, 1, "Light X Position: " + str( lightPos.x ) )
  printAt( 0, 2, "Light Y Position: " + str( lightPos.y ) )
  printAt( 0, 3, "Light Z Position: " + str( lightPos.z ) )
  update()
repeat
```

Associated Commands

[pointLight\(\)](#), [pointShadowLight\(\)](#), [removeLight\(\)](#), [setAmbientLight\(\)](#), [setLightBrightness\(\)](#), [setLightColour\(\)](#), [setLightDir\(\)](#), [setLightSpread\(\)](#), [spotLight\(\)](#), [worldLight\(\)](#), [worldShadowLight\(\)](#)

COMMAND REFERENCE

setLightSpread()

Purpose

Sets the spread of a light source

Description

Set the amount of spread of a light source to the specified value

Syntax

```
setLightSpread( light, spread )
```

Arguments

light handle of the light source

spread a measure of how much the light from the light source spreads

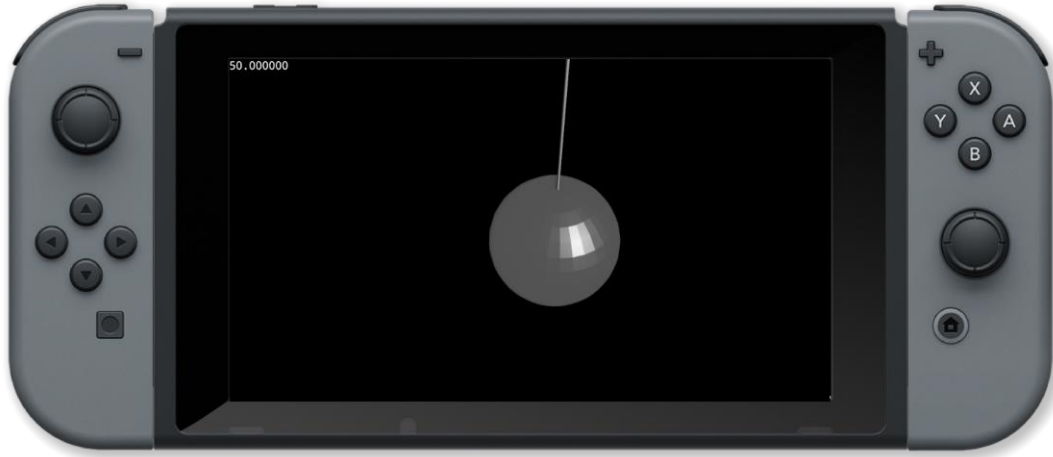
Example

```
setCamera( { 0, 6, 10 }, { 0, 0, 0 } )
spread = 50
light = spotLight( { 0, 4, 0 }, { 4, 0.1, 4 }, white, 100, spread )

obj = [
  placeObject( cube, { 0, 0, 0 }, { 4, 0.1, 4 } ),
  placeObject( cube, { 0, 1.1, 0 }, { 1, 1, 1 } )
]

setObjectMaterial( obj[0], white, 0, 1 )
setObjectMaterial( obj[1], cyan, 0, 1 )

loop
  c = controls( 0 )
  spread += c.ly
  spread = clamp( spread, 0, 100 )
  setLightSpread( light, spread )
  rotateObject( obj[1], { 0, 1, 0 }, 1 )
  drawObjects()
  printAt( 0, 0, "Use Joy-Con left Control Stick to adjust light spread: " + spread )
  update()
repeat
```



Associated Commands

`pointLight()`, `pointShadowLight()`, `removeLight()`, `setAmbientLight()`, `setLightBrightness()`,
`setLightColour()`, `setLightDir()`, `setLightPos()`, `spotLight()`, `worldLight()`, `worldShadowLight()`

COMMAND REFERENCE

setObjectMaterial()

Purpose

Set the material of a 3D object

Description

Changes the way light behaves on the surface of the object.

Syntax

```
setObjectMaterial( handle, colour, metallic, roughness )
```

Arguments

handle The handle of the placed 3D object

colour colour name or RGB values { red, green, blue, opacity } between 0 and 1

metallic A flag to indicate whether the object is metallic(1) or non metallic(0)

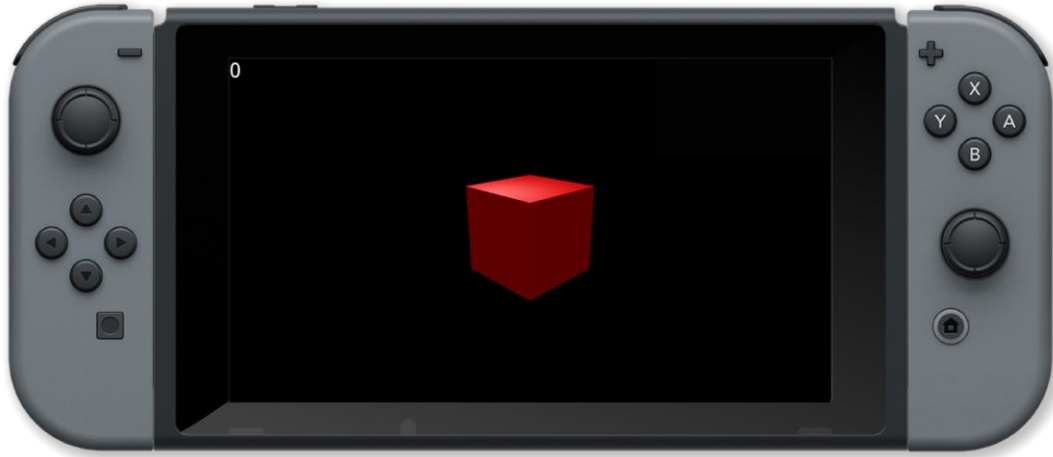
roughness A value for the roughness of the objects between 0 and 1 (0 is completely smooth and 1 is very rough)

Example

```
obj = placeObject( sphere, { 0, 0, 0 }, { 2, 2, 2 } )
setObjectMaterial( obj, red, 1, 1 )
setCamera( { 5, 5, 10 }, { 0, 0, 0 } )
light = pointLight( { 0, 4, 2 }, white, 50 )

loop
  clear()
  c = controls( 0 )
  if c.a then
    setObjectMaterial( obj, red, 0, 1 )
  endif
  if c.b then
    setObjectMaterial( obj, red, 1, 0 )
  endif
  if c.x then
    setObjectMaterial( obj, red, 0, 0 )
  endif
  if c.y then
    setObjectMaterial( obj, red, 1, 1 )
  endif
  drawObjects()
  printAt( 0, 0, "Use A, B, X and Y buttons to change object material" )
```

```
update()  
repeat
```



Associated Commands

`drawObjects()`, `loadModel()`, `objectPointAt()`, `placeObject()`, `removeObject()`, `rotateObject()`,
`setObjectPos()`, `setObjectScale()`

COMMAND REFERENCE

setObjectPos()

Purpose

Set the position of a 3D object

Description

Change the scale factor used for the display of a 3D object and hence it's relative size

Syntax

```
setObjectPos( handle, pos )
```

Arguments

handle variable which stores the placed 3D object

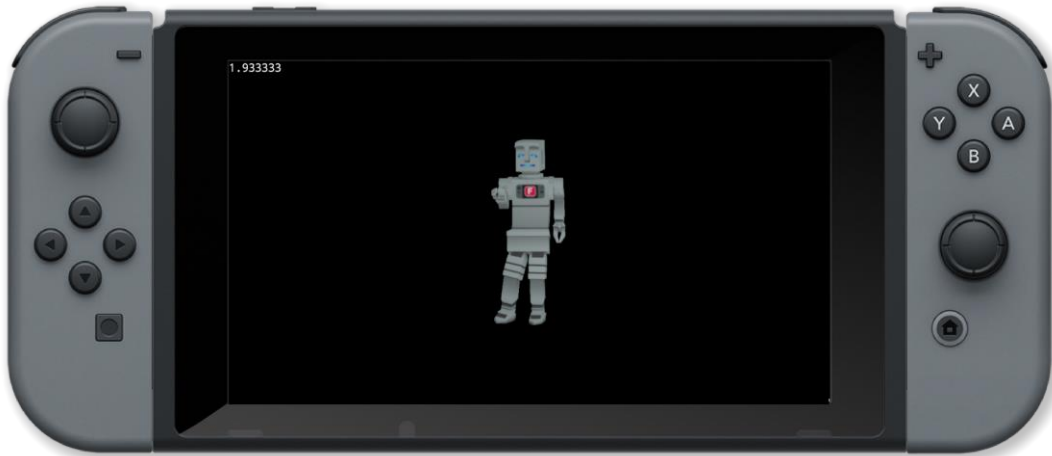
pos vector containing the position 3 dimensions { x, y, z }

Example

```
cb = loadModel( "Kat/Colin" )
pointLight( { 0.5, 1.3, 2 }, white, 4 )
setAmbientLight( { 0.5, 0.5, 0.5 } )
pos = { 0, 0, 0 }
scale = { 1, 1, 1 }
colin = placeObject( cb, pos, scale )
setCamera( { 0, 10, 10 }, { 0, 5, 0 } )
animID = 6
animlength = animationLength( colin, animID )
animframe = 0

loop
  clear()
  c = controls( 0 )
  if c.left then
    pos.x -= 0.1
  endIf
  if c.right then
    pos.x += 0.1
  endIf
  if c.up then
    pos.y += 0.1
  endIf
  if c.down then
    pos.y -= 0.1
  endIf
  setObjectPos( colin, pos )
  animframe = animframe + 1 / 60
```

```
if animframe >= animlength then
  animframe = 0
endIf
updateAnimation( colin, animID, animframe )
drawObjects()
printAt( 0, 0, "Use arrows to move object" )
update()
repeat
```



Associated Commands

`drawObjects()`, `loadModel()`, `objectPointAt()`, `placeObject()`, `removeObject()`, `rotateObject()`, `setObjectMaterial()`, `setObjectScale()`

COMMAND REFERENCE

setObjectScale()

Purpose

Set the size of a 3D object

Description

Change the scale factor used for the display of a 3D object and hence it's relative size

Syntax

```
setObjectScale( handle, scale )
```

Arguments

handle variable which stores the placed 3D object

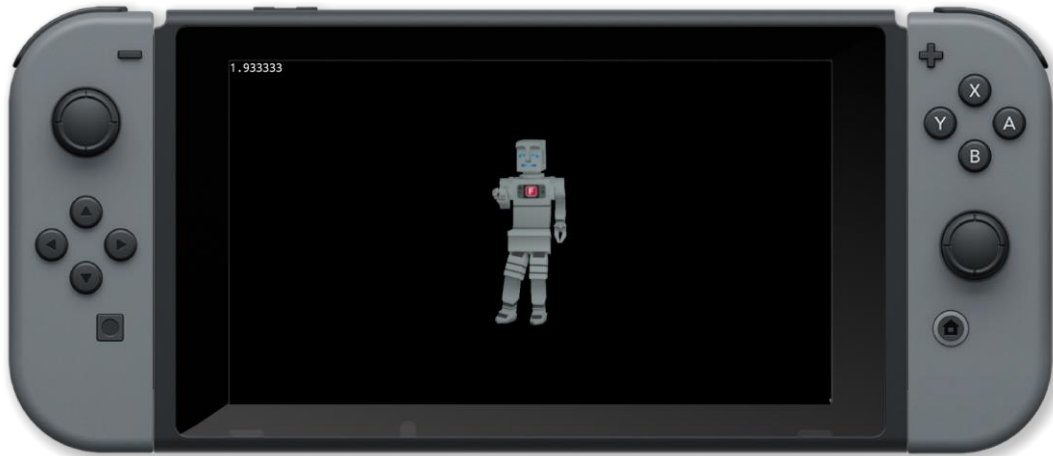
scale vector containing the scale factors in 3 dimensions { x, y, z }

Example

```
cb = loadModel( "Kat/Colin" )
pointLight( { 0.5, 1.3, 2 }, white, 4 )
setAmbientLight( { 0.5, 0.5, 0.5 } )
scale = { 1, 1, 1 }
colin = placeObject( cb, { 0, 0, 0 }, scale )
setcamera( { 0, 10, 10 }, { 0, 5, 0 } )
animID = 6
animlength = animationLength( colin, animID )
animframe = 0

loop
  clear()
  c = controls( 0 )
  if c.up then
    scale.x += 0.1
    scale.y += 0.1
    scale.z += 0.1
  endIf
  if c.down then
    scale.x -= 0.1
    scale.y -= 0.1
    scale.z -= 0.1
  endIf
  scale.x = clamp( scale.x, 0.5, 5 )
  scale.y = clamp( scale.y, 0.5, 5 )
  scale.z = clamp( scale.z, 0.5, 5 )
  setObjectScale( colin, scale )
  animframe = animframe + 1 / 60
```

```
if animframe >= animlength then
  animframe = 0
endIf
updateAnimation( colin, animID, animframe )
drawObjects()
printAt( 0, 1, "Use the up and down arrows to increase or decrease the scale" )
update()
repeat
```



Associated Commands

`drawObjects()`, `loadModel()`, `objectPointAt()`, `placeObject()`, `removeObject()`, `rotateObject()`, `setObjectMaterial()`, `setObjectPos()`

COMMAND REFERENCE

setTerrainPoint()

Purpose

Set a point on a 3D terrain

Description

Set the height and colour of a point on a terrain grid

Syntax

```
setTerrainPoint( terrain, xpos, ypos, height, colour )
```

Arguments

terrain handle of the terrain from createterrain

xpos horizontal position in the grid

ypos vertical position in the grid

height height of the point

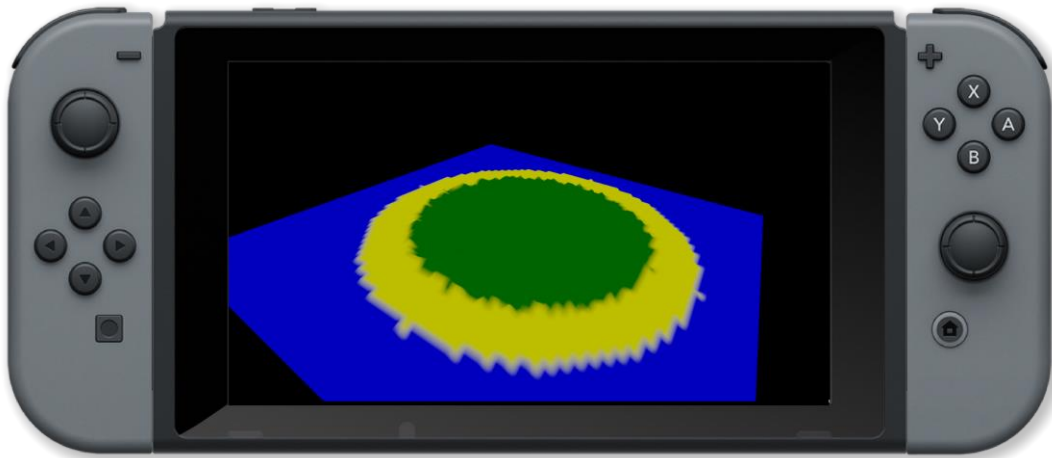
colour colour name or RGB values { red, green, blue, opacity } between 0 and 1

Example

```
gsize = 64
landscape = createTerrain( gsize, 1 )
height = 0
colour = white

for x = 0 to gsize loop
  for y = 0 to gsize loop
    d = distance ( { x, y }, { gsize / 2, gsize / 2 } )
    if d > 24 then // sea level
      height = 0
      colour = blue
    else
      if d > 18 then // beach
        height = 1
        colour = yellow
      else // hills
        height = rnd( 2 ) + 1
        colour = green
      endif
    endif
    setTerrainPoint( landscape, x, y, height, colour )
  repeat
```

```
repeat
setCamera( { gsize / 2, 50, gsize / 2 }, { gsize / 2.0, 0, gsize / 2.00001 } )
setambientlight( { 0.5, 0.5, 0.5 } )
island = placeObject( landscape, { gsize / 2, 0, gsize / 2}, { 1, 1, 1 } )
loop
  printAt( 0, 0, "rotate using joysticks" )
  c = controls( 0 )
  rotateObject( island, { 1, 0, 0 }, c.ly )
  rotateObject( island, { 0, 0, 1 }, c.lx )
  rotateObject( island, { 0, 1, 0 }, c.rx )
  drawObjects()
  update()
repeat
```



Associated Commands

`createTerrain()`, `updateTerrain()`

COMMAND REFERENCE

spotLight()

Purpose

Create a spotlight light source in 3D space

Description

Creates a spotlight source in the specified position of the specified colour, brightness and with the specified spread angle

Syntax

```
handle = spotLight( position, direction, colour, brightness, spread )
```

Arguments

handle handle of the light source

position A position vector in 3 dimensional space { x, y, z } where the light source is located

direction A vector to describe the direction in which the light is pointing

colour colour name or RGB values { red, green, blue, opacity } between 0 and 1

brightness A value to indicate the brightness of the light source

spread angle (in degrees) of the spread of the light

Example

```
setCamera( { 0, 6, 10 }, { 0, 0, 0 } )
spread = 50
light = spotLight( { 0, 4, 0 }, { 0, -1, 0 }, white, 100, spread )

obj = [
  placeObject( cube, { 0, 0, 0 }, { 4, 0.1, 4 } ),
  placeObject( cube, { 0, 1.1, 0 }, { 1, 1, 1 } )
]

setObjectMaterial( obj[0], white, 0, 1 )
setObjectMaterial( obj[1], cyan, 0, 1 )

loop
  c = controls( 0 )
  spread += c.ly
  spread = clamp( spread, 0, 100 )
  setLightSpread( light, spread )
  rotateObject( obj[1], { 0, 1, 0 }, 1.0 )
drawObjects()
```

```
printAt( 0, 0, "Use Joy-Con left control stick adjust spread: " + spread )  
update()  
repeat
```

Associated Commands

pointLight(), pointShadowLight(), removeLight(), setAmbientLight(), setLightBrightness(),
setLightColour(), setLightDir(), setLightPos(), setLightSpread(), worldLight(),
worldShadowLight()

COMMAND REFERENCE

updateAnimation()

Purpose

Update a 3D animation

Description

Some 3D models contain animation sequences. This updates the animation with the specified frame

Syntax

```
updateAnimation( object, animation, frame )
```

Arguments

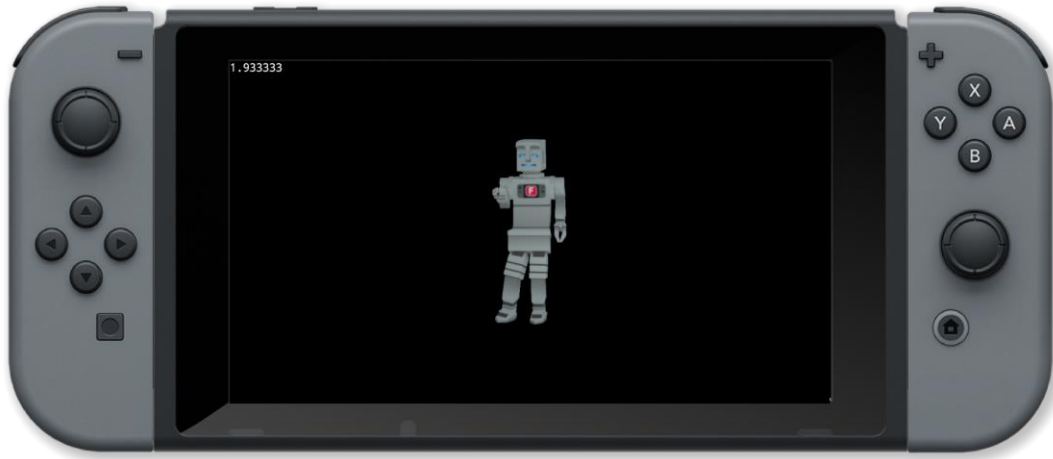
object handle of the animated 3D object

animation index of the animation

frame index of the animation frame

Example

```
cb = loadModel( "Kat/Colin" )
pointLight( { 0.5, 1.3, 2 }, white, 4 )
setAmbientLight( { 0.5, 0.5, 0.5 } )
colin = placeObject( cb, { 0, 0, 0 }, { 1, 1, 1 } )
setCamera( { 0, 10, 10 }, { 0, 5, 0 } )
animID = 6
animlength = animationLength( colin, animID )
animframe = 0
loop
  clear()
  animframe = animframe + 1 / 60
  if animframe >= animlength then
    animframe = 0
  endif
  updateAnimation( colin, animID, animframe )
  drawObjects()
  printAt( 0, 0, animframe )
  update()
repeat
```



Associated Commands

`animationLength()`, `numAnimations()`

COMMAND REFERENCE

updateTerrain()

Purpose

Update a 3D terrain

Description

Update a 3D terrain from arrays of heights and colours

Syntax

```
updateTerrain( terrain, heights, colours )
```

Arguments

terrain identifier of the terrain from createterrain

heights array containing height of each point on the grid

colour array containing colour of each point on the grid

Example

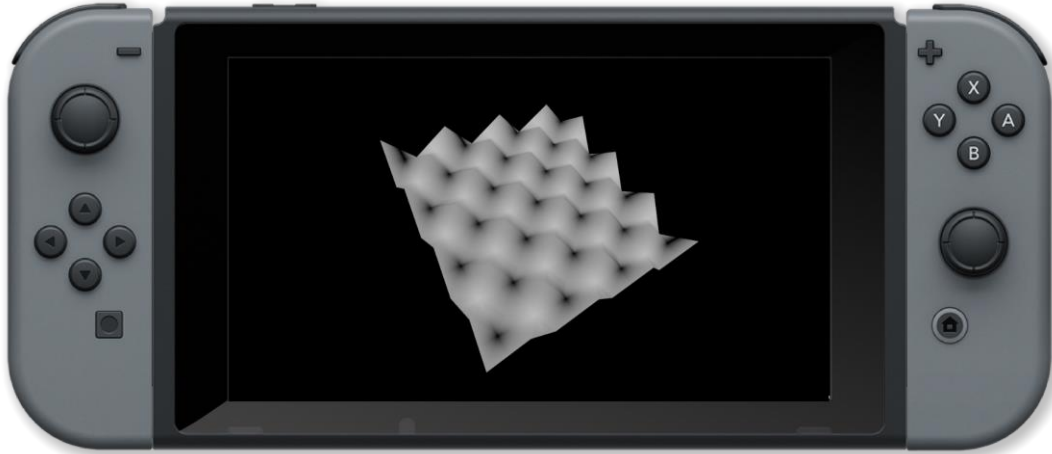
```
gsize = 8
landscape = createTerrain( gsize, 1 )
colours = []
heights = []
palette = []
palette[0] = black
palette[1] = white

j = 0
for i = 0 to 64 loop
    j = j + 1
    if i % 8 == 0 then
        j = j + 1
    endIf
    colours[i] = palette[ j % 2 ]
    heights[i] = float( j % 2 )
repeat

updateTerrain( landscape, heights, colours )
setCamera( { gsize / 2, 10, gsize / 2 }, { gsize / 2.0, 0, gsize / 2.00001 } )
setAmbientLight( { 0.5, 0.5, 0.5 } )
island = placeObject( landscape, { gsize / 2, 0, gsize / 2 }, { 1, 1, 1 } )

loop
    c = controls( 0 ) // rotate using joysticks
    rotateObject( island, { 1, 0, 0 }, c.ly )
```

```
rotateObject( island, { 0, 0, 1 }, c.lx )
rotateObject( island, { 0, 1, 0 }, c.rx )
drawObjects()
update()
repeat
```



Associated Commands

`createTerrain()`, `setTerrainPoint()`

COMMAND REFERENCE

worldLight()

Purpose

Create a world light source

Description

Creates a world light source in the specified direction and with the specified colour and brightness

Syntax

```
handle = worldLight( direction, colour, brightness )
```

Arguments

direction direction the light source is pointing { x, y, z }

colour colour name or RGB values { red, green, blue, opacity } between 0 and 1

brightness brightness of the light source (0 - 100)

handle handle of the light source

Example

```
obj1 = placeObject( cube, { -3, 0, 0 }, { 1, 1, 1 } )
obj2 = placeObject( cube, { 3, 0, 0 }, { 1, 1, 1 } )
setObjectMaterial( obj1, white, 0, 0.05 ) // white, smooth, shiny
setObjectMaterial( obj2, white, 0, 0.05 ) // white, smooth, shiny
floor = placeObject( cube, { 0, -2, 0 }, { 10, 0.05, 10 } )
setObjectMaterial( floor, grey, 0, 1 ) // grey, smooth, not shiny
setCamera( { 1, 0.5, 5 }, { 0, 0, 0 } ) // back a bit and off centre, facing world centre
worldLight( { -1, -0.5, -0.5 }, white, 1 )

loop
  rotateObject( obj1, { 1, 1, 1 }, 1 ) // in all directions by 1 degree
  rotateObject( obj2, { 1, 1, 1 }, -1 ) // in all directions by -1 degree
  drawObjects()
  update()
repeat
```



Associated Commands

`pointLight()`, `pointShadowLight()`, `removeLight()`, `setAmbientLight()`, `setLightBrightness()`,
`setLightColour()`, `setLightDir()`, `setLightPos()`, `setLightSpread()`, `spotLight()`, `worldShadowLight()`

COMMAND REFERENCE

worldShadowLight()

Purpose

Create a world light source that casts a shadow

Description

Creates a world light source in the specified direction and with the specified colour and brightness

Syntax

```
handle = worldShadowLight( centre, direction, colour, brightness, range, resolution )
```

Arguments

centre centre of the range for shadows

direction direction the light source is pointing { x, y, z }

colour colour name or RGB values { red, green, blue, opacity } between 0 and 1

brightness brightness of the light source (0 - 100)

range range of shadows

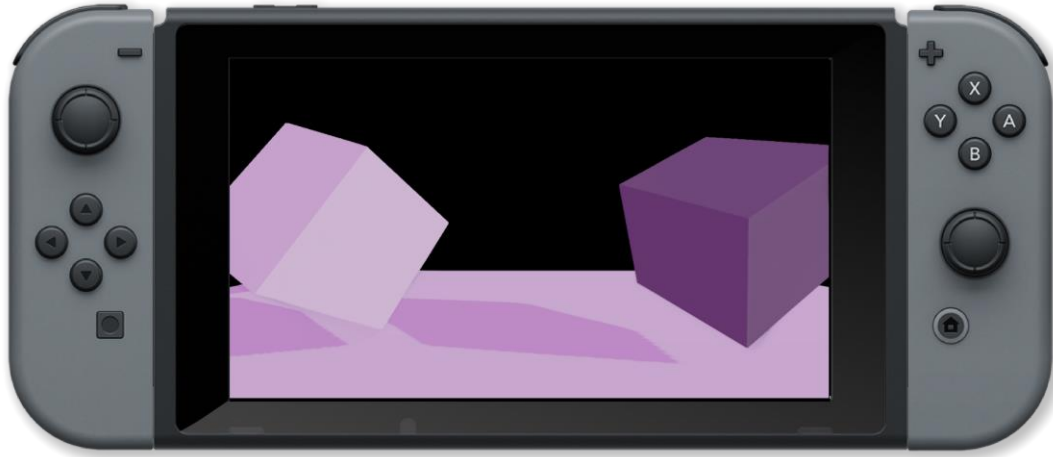
resolution resolution of shadows (higher is smoother)

handle handle of the light source

Example

```
obj1 = placeObject( cube, { -3, 0, 0 }, { 1, 1, 1 } )
obj2 = placeObject( cube, { 3, 0, 0 }, { 1, 1, 1 } )
setObjectMaterial( obj1, white, 0, 0.05 ) // white, smooth, shiny
setobjectmaterial( obj2, white, 0, 0.05 ) // white, smooth, shiny
floor = placeObject( cube, { 0, -2, 0 }, { 10, 0.05, 10 } )
setObjectMaterial( floor, grey, 0, 1 ) // grey, smooth, not shiny
setCamera( { 1, 0.5, 5 }, { 0, 0, 0 } ) // back a bit and off centre, facing world centre
worldShadowLight( { 0, 0, 0 }, { -1, -0.5, -0.5 }, white, 1, 10, 512 )

loop
  rotateObject( obj1, { 1, 1, 1 }, 1 ) // in all directions by 1 degree
  rotateObject( obj2, { 1, 1, 1 }, -1 ) // in all directions by -1 degree
  drawObjects()
  update()
repeat
```



Associated Commands

`pointLight()`, `pointShadowLight()`, `removeLight()`, `setAmbientLight()`, `setLightBrightness()`,
`setLightColour()`, `setLightDir()`, `setLightPos()`, `setLightSpread()`, `spotLight()`, `worldLight()`

COMMAND REFERENCE

Arithmetic

COMMAND REFERENCE

abs()

Purpose

Returns absolute value of the given argument.

Description

Always returns the positive value of a given number. For instance: `abs(-1) = 1`

Syntax

```
absolute = abs( number )
```

Arguments

number A positive or negative number

absolute The positive value of number (the number without the - sign)

Example

```
image = loadImage( "Untied Games/Enemy small top C", false )
ship = createSprite()
setSpriteImage( ship, image )
lastpos = { gWidth() / 2, gHeight() / 2 }
setSpriteLocation( ship, lastpos )
setSpriteScale( ship, { 10, 10 } )
maxrs = 240 // max rotation speed
accr = 1 // accelaration
loop
  clear()
  rs = getSpriteRotationSpeed( ship )
  if ( abs( rs ) > maxrs ) then
    accr = -accr
  endIf
  setSpriteRotationSpeed( ship, rs + accr )
  updateSprites()
  drawSprites()
  update()
repeat
```

Associated Commands

COMMAND REFERENCE

acos()

Purpose

Returns the arc cosine of the supplied argument.

Description

This is the inverse of the COS function, for returning the angle from a given cosine.

Syntax

```
angle = acos( cosine )
```

Arguments

cosine The ratio of the side adjacent to an acute angle in a right-angled triangle to the hypotenuse.

angle The acute angle in a right-angled triangle for the given cosine

Example

```
cosine = 0.5  
angle = acos( cosine )  
print ( "Angle = ", angle )  
update()  
sleep( 3 )
```

Associated Commands

asin(), atan(), atan2(), pi(), radians(), sin(), sinCos(), tan()

COMMAND REFERENCE

asin()

Purpose

Returns the arc sine of the supplied argument.

Description

This is the inverse of the SIN function, for returning the angle from a given sine.

Syntax

```
angle = asin( sine )
```

Arguments

sine The ratio of the side opposite to an acute angle in a right-angled triangle to the hypotenuse.

angle The acute angle in a right-angled triangle for the given sine

Example

```
sine = 0.5  
angle = asin( sine )  
print( "Angle = ", angle )  
update()  
sleep( 3 )
```

Associated Commands

acos(), atan(), atan2(), pi(), radians(), sin(), sinCos(), tan()

COMMAND REFERENCE

atan()

Purpose

Returns the arc tangent of the supplied argument.

Description

This is the inverse of the TAN function, for returning the angle from a given tangent.

Syntax

```
angle = atan( tangent )
```

Arguments

tangent The ratio of the side opposite to an acute angle in a right-angled triangle to the one adjacent.

angle The acute angle in a right-angled triangle for the given tangent

Example

```
tangent = 0.5  
angle = atan( tangent )  
print( "Angle = ", angle )  
update()  
sleep( 3 )
```

Associated Commands

[acos\(\)](#), [asin\(\)](#), [atan2\(\)](#), [pi\(\)](#), [radians\(\)](#), [sin\(\)](#), [sinCos\(\)](#), [tan\(\)](#)

COMMAND REFERENCE

atan2()

Purpose

Returns the arctangent between the point specified and the origin.

Description

Unlike ATAN, atan2 requires two values. It returns the angle between the origin and the point specified.

Syntax

```
angle = atan2( x, y )
```

Arguments

x The number of horizontal pixels from the origin (0,0)

y The number of vertical pixels from the origin (0,0)

angle The angle between the origin and the specified point

Example

```
radians( true )
image = loadImage( "Untied Games/Enemy small top C", false )
ship = createSprite()
setSpriteImage( ship, image )
lastpos = { gWidth() / 2, gHeight() / 2 }
setSpriteLocation( ship, lastpos )
setSpriteScale( ship, { 4, 4 } )
loop
  clear()
  c = controls( 0 )
  printAt( 0, 0, "Use left control stick to control sprite" )
  setSpriteSpeed( ship, { 480 * c.lx, -480 * c.ly } )
  curpos = getSpriteLocation( ship )
  if curpos != lastpos then
    setSpriteRotation( ship, -pi / 2 + atan2( curpos.y - lastpos.y, curpos.x - lastpos.x ) )
    lastpos = curpos
  endif
  updateSprites()
  drawSprites()
  update()
repeat
```

Associated Commands

acos(), asin(), atan(), pi(), radians(), sin(), sinCos(), tan()

COMMAND REFERENCE

bezier()

Purpose

Used to draw quadratic bezier curves

Description

A Bezier curve (pronounced [bezje] in French) is a parametric curve used in computer graphics and related fields.

Syntax

```
point = bezier( point1, point2, point3, factor )  
point = bezier( point1, point2, point3, point4, factor )
```

Arguments

point1 The first point

point2 The second point

point3 The third point

point4 Optional fourth point

factor interpolation factor

point A point on the bezier curve

Example

```
a = 0  
c = { gWidth() / 2 , gHeight() / 2 }  
p1 = { 0, gHeight() / 2 }  
p3 = { gWidth(), gHeight() / 2 }  
loop  
  clear()  
  p2 = c + sincos( a ) * ( gHeight() / 2 )  
  op = p1  
  for i = 0 to 16 loop  
    p = bezier( p1, p2, p3, i / 15 )  
    line( op, p, white )  
    op = p  
  repeat  
  a += 0.01  
  update()  
repeat
```



Associated Commands

`lerp()`, `smoothStep()`

COMMAND REFERENCE

bitCount()

Purpose

Returns the number of bits set (1) in a binary number

Description

Counts how many bits in a binary number are set to 1

Syntax

```
result = bitCount( number )
```

Arguments

number 32 bit signed binary number

result Number of bits set to 1 in number

Example

```
number = 0
for i = 0 to 16 loop
    number = bitSet( number, i, 1 )
    count = bitCount( number )
    printAt( 0, i , number, " ", count )
    update()
repeat
sleep( 3 )
```

Associated Commands

[bitGet\(\)](#), [bitSet\(\)](#), [bitFieldExtract\(\)](#), [bitFieldInsert\(\)](#), [leadingZeroes\(\)](#), [trailingZeroes\(\)](#)

COMMAND REFERENCE

ceil()

Purpose

Returns the ceiling of a real number

Description

Returns the least integer greater than or equal to the real number specified e.g. `ceil(2.4) = 3`

Syntax

```
result = ceil( number )
```

Arguments

number A real number

result The ceiling of number

Example

```
roll = 0
clear()
image = loadImage( "Colin Brown/Dice", false )
size = tileSize( image, 0 )
for i = 1 to 10 loop
    clear()
    roll = random( 6 ) + 1
    x = size.x - ( size.x * ( roll % 2 ) )
    y = size.y * ( ceil( roll / 2 ) - 1 )
    drawImage( image, { x, y, size.x, size.y }, { 0, 0, size.x, size.y } )
    update()
    sleep( 0.3 )
repeat
printAt( 0, 15, "You rolled a ", roll )
update()
sleep( 3 )
```

Associated Commands

`clamp()`, `floor()`, `max()`, `min()`

COMMAND REFERENCE

clamp()

Purpose

Restrict a value to a specified range

Description

If the specified number is below the minimum value returns the minimum value. If it is above the maximum value returns the maximum value. If it is between minimum and maximum then the number is returned.

Syntax

```
result = clamp( number, minimum, maximum )
```

Arguments

number Number to restrict

minimum Minimum value for number

maximum Maximum value for number

result Restricted value

Example

```
setCamera( {0, 10, 10 }, { 0, 0, 0 } )
bright = 50
light = worldLight( { -1, -1, -1 }, white, bright )
ballmodel = loadModel( "Kat/Discoball" )
ball = placeObject( ballmodel, { 0, -0, 0 }, { 10, 10, 10 } )
loop
  c = controls( 0 )
  if c.up then
    bright = bright + 1
  endIf
  if c.down then
    bright = bright - 1
  endIf
  bright = clamp( bright, 0, 100 )
  setLightBrightness( light, bright )
  rotateObject( ball, { 0, 1, 0 }, 1.0 )
  drawObjects()
  printAt( 0, 0, "Use up and down arrows to adjust brightness: ", bright )
  update()
repeat
```

Associated Commands

`ceil()`, `floor()`, `max()`, `min()`

COMMAND REFERENCE

cos()

Purpose

Returns the cosine of the supplied argument.

Description

This is the cosine function which returns the ratio of the side adjacent to an acute angle in a right-angled triangle to the hypotenuse (longest side)

Syntax

```
cosine = cos( angle )
```

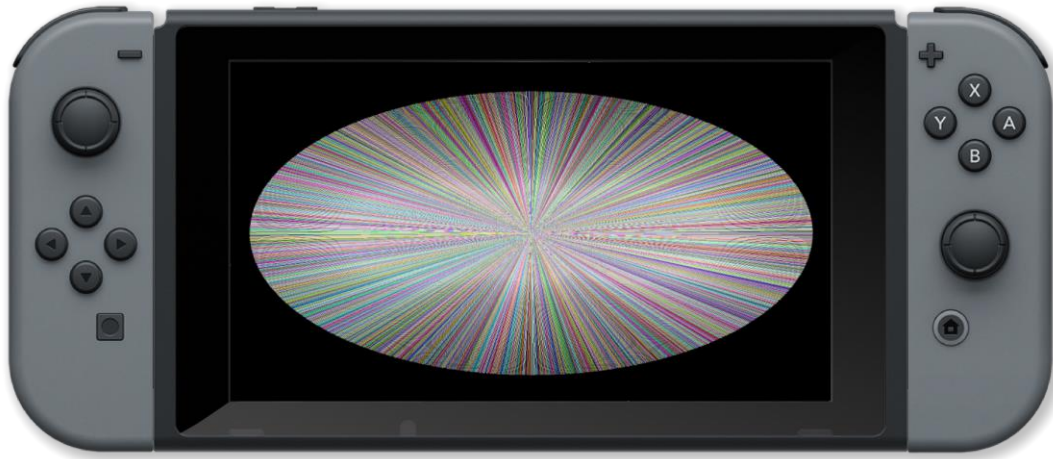
Arguments

angle The acute angle adjacent to the side of a right-angled triangle.

cosine The ratio of the adjacent side to the hypotenuse.

Example

```
clear()
centre = { gWidth() / 2, gHeight() / 2 }
for angle = 0 to 360 loop
    // Pick random colour
    col = { random( 101 ) / 100, random( 101 ) / 100, random( 101 ) / 100, 1.0 }
    point = { 600 * cos( angle ) + centre.x, 300 * sin( angle ) + centre.y }
    line( centre, point, col )
repeat
update()
// Wait 3 seconds
sleep( 3 )
```



Associated Commands

`acos()`, `asin()`, `atan()`, `atan2()`, `pi()`, `radians()`, `sin()`, `sinCos()`, `tan()`

COMMAND REFERENCE

cross()

Purpose

Find the cross product of two vectors

Description

Calculate the cross product of two vectors which is the vector that is at right angles to them both

Syntax

```
vector3 = cross( vector1, vector2 )
```

Arguments

vector1 The first vector (x,y,z)

vector2 The second vector (x,y,z)

vector3 The resulting vector (x,y,z)

Example

```
vector1 = { 2, 3, 4 }  
vector2 = { 5, 6, 7 }  
vector3 = cross( vector1, vector2 )  
print( "(", vector3.x, ", ", vector3.y, ", ", vector3.z, ")" )  
update()  
sleep( 3 )
```

Associated Commands

[dot\(\)](#), [length\(\)](#), [normalize\(\)](#), [reflect\(\)](#), [refract\(\)](#)

COMMAND REFERENCE

distance()

Purpose

Find the distance between 2 points

Description

Returns the distance between 2 points in a 2 or 3 dimensions

Syntax

```
result = distance( point1, point2 )
```

Arguments

point1 Coordinates of first point { x, y, z }

point2 Coordinates of second point { x, y, z }

result Distance between point1 and point2

Example

```
p1 = { 80, 20, 60 }  
p2 = { 30, 50, 70 }  
print( distance( p1, p2 ) )  
update()  
sleep( 3 )
```

Associated Commands

COMMAND REFERENCE

dot()

Purpose

Find the dot product of two vectors

Description

Calculate the dot product (the scalar value from multiplication) of two vectors

Syntax

```
scalar = dot( vector1, vector2 )
```

Arguments

vector1 The first vector (x,y,z)

vector2 The second vector (x,y,z)

scalar The resulting scalar value

Example

```
vector1 = { 2, 3, 4 }  
vector2 = { 5, 6, 7 }  
scalar = dot( vector1, vector2 )  
print( scalar )  
update()  
sleep( 3 )
```

Associated Commands

[cross\(\)](#), [length\(\)](#), [normalize\(\)](#), [reflect\(\)](#), [refract\(\)](#)

COMMAND REFERENCE

float()

Purpose

Convert value to floating point

Description

Convert a string or integer value to a floating point number

Syntax

```
result = float( value )
```

Arguments

value string or int value to be converted

result floating point result

Example

```
print( float( "3.6" ) ) // prints 3.600000
print( float( "3.1415926" ) ) // prints 3.141593
print( float( "99" ) ) // prints 99.000000
print( float( "Hello" ) ) // prints 0.000000
update()
sleep( 3 )
```

Associated Commands

`int()`, `fract()`, `round()`, `str()`, `trunc()`

COMMAND REFERENCE

floor()

Purpose

Returns the floor of a real number

Description

Returns the greatest integer less than or equal to the real number specified e.g. `floor(2.4) = 2`

Syntax

```
result = floor( number )
```

Arguments

number A real number

result The floor of number

Example

```
x = 2.4
// Prints 2
print( floor( x ) )
update()
sleep( 3 )
```

Associated Commands

`ceil()`, `clamp()`, `max()`, `min()`

COMMAND REFERENCE

fract()

Purpose

Get the fractional part of floating point number

Description

Returns the fractional part of a floating point number (the part after the decimal point)

Syntax

```
result = fract( value )
```

Arguments

value floating point value find the fractional part of

result fractional part of value

Example

```
print( fract( 3.1415926 ) ) // prints 0.141593
print( fract( 3 ) ) // prints 0.000000
update()
sleep( 3 )
```

Associated Commands

int(), float(), round(), str(), trunc()

COMMAND REFERENCE

int()

Purpose

Convert value to an integer

Description

Convert a string or float value to an integer

Syntax

```
result = int( value )
```

Arguments

value string or float value to be converted

result integer result

Example

```
print( int( "3.6" ) ) // prints 3
print( int( "3.1415926" ) ) // prints 3
print( int( "99" ) ) // prints 99
print( int( "Hello" ) ) // prints 0
update()
sleep( 3 )
```

Associated Commands

float(), fract(), round(), str(), trunc()

COMMAND REFERENCE

length()

Purpose

Find the length of a vector

Description

Calculates the length of a vector in 1,2,3 or 4 dimensions

Syntax

```
result = length( vector )
```

Arguments

result length of the specified vector

vector vector to find the length of

Example

```
print( length( { 3 } ) ) // prints 3
print( length( { 3, 4 } ) ) // prints 5
print( length( { 3, 4, 5 } ) ) // prints 7.071068
print( length( { 3, 4, 5, 6 } ) ) // prints 9.273619
update()
sleep( 3 )
```

Associated Commands

`cross()`, `dot()`, `normalize()`, `reflect()`, `refract()`

COMMAND REFERENCE

lerp()

Purpose

Linear interpolation

Description

Returns an interpolation between two inputs (v_0 , v_1) for a parameter (t) in the closed unit interval $[0, 1]$. This lerp function is commonly used for alpha blending (the parameter “ t ” is the “alpha value”), and the formula may be extended to blend multiple components of a vector (such as spatial x , y , z axes or r , g , b colour components) in parallel.

Syntax

```
result = lerp( v0, v1, t )
```

Arguments

result linear interpolation

v0 first value

v1 second value

t parameter from 0 to 1

Example

```
image = loadImage( "Untied Games/Knight", false )
sprite = createSprite()
setSpriteImage( sprite, image )
setSpriteAnimation( sprite, 8, 11, 10 )
setSpriteScale( sprite, 5, 5 )
sprite.y = 100
start_x = 0
end_x = gWidth()
t = 0
duration = 5
reverse = false
loop
  clear()
  t += deltaTime() // duration
  if t >= 1 then
    t = 0
    //reverse direction by swapping start and end
    temp = start_x
    start_x = end_x
    end_x = temp
    //reverse the character graphics direction
```

```
    sprite.xscale *= -1
  endIf
  sprite.x = lerp( start_x, end_x, t )
  updateSprites()
  drawSprites()
  update()
repeat
```

Associated Commands

[bezier\(\)](#), [smoothStep\(\)](#)

COMMAND REFERENCE

max()

Purpose

Returns the maximum of two numbers

Description

Find which is the highest of two numbers

Syntax

```
result = max( number1, number2 )
```

Arguments

number1 first number

number2 second number

result whichever is the highest of *number1* or *number2*

Example

```
a = 2
b = -2
// Prints 2
print( max( a, b ) )
update()
sleep( 3 )
```

Associated Commands

[ceil\(\)](#), [clamp\(\)](#), [floor\(\)](#), [min\(\)](#)

COMMAND REFERENCE

min()

Purpose

Returns the minimum of two numbers

Description

Find which is the lowest of two numbers

Syntax

```
result = min( number1, number2 )
```

Arguments

number1 first number

number2 second number

result whichever is the lowest of *number1* or *number2*

Example

```
a = 2
b = -2
// Prints -2
print( min( a, b ) )
update()
sleep( 3 )
```

Associated Commands

[ceil\(\)](#), [clamp\(\)](#), [floor\(\)](#), [max\(\)](#)

COMMAND REFERENCE

normalize()

Purpose

Normalize a vector

Description

Gets the normalized vector of a specified vector which is a vector in the same direction but with length 1 (also called the unit vector)

Syntax

```
result = normalize( vector )
```

Arguments

result length of the specified vector

vector normalized vector

Example

```
result = normalize( { 3, 4, 5 } )  
print( "{ ", result.x, ", ", result.y, ", ", result.z, " }")  
print( length( result ) ) // prints 1.000000  
update()  
sleep( 3 )
```

Associated Commands

[cross\(\)](#), [dot\(\)](#), [length\(\)](#), [reflect\(\)](#), [refract\(\)](#)

COMMAND REFERENCE

pi()

Purpose

Returns the value of pi

Description

Returns an approximation of the value of the constant pi which is the ratio of a circle's circumference to its diameter (approximately 3.14159265) which is widely used in mathematics, specifically trigonometry and geometry.

Syntax

```
value = pi
```

Arguments

value An approximation of the value of pi

Example

```
clear()
radians( true )
centre = { gWidth() / 2, gHeight() / 2 }
for angle = 0 to 2 * pi step 0.005 loop
  col = { random( 101 ) / 100, random( 101 ) / 100, random( 101 ) / 100, 1.0 }
  result = sincos( angle )
  point = { 600 * result.y + centre.x, 300 * result.x + centre.y }
  line( centre, point, col )
  repeat
update()
sleep( 3 )
```



Associated Commands

`acos()`, `asin()`, `atan()`, `atan2()`, `radians()`, `sin()`, `sinCos()`, `tan()`

COMMAND REFERENCE

pow()

Purpose

Raise a number to the power of another

Description

Returns a number to the specified power

Syntax

```
result = pow( number, power )
```

Arguments

result number raised to the specified power

number number to be raised to the specified power

power power to raise number by

Example

```
for i = 0 to 16 loop
  j = pow( 2, i )
  printAt( 0, i, int( j ), " ", trailingZeroes( j ) )
  update()
repeat
sleep( 3 )
```

Associated Commands

rsqrt(), sqrt()

COMMAND REFERENCE

radians()

Purpose

Change the default units for angles

Description

By default all functions that get or return angles use degrees as the unit (360 degrees in a circle). This function allows you to switch to using radians ($2 * \text{PI}$ radians in a circle) and back to degrees

Syntax

```
radians( enable )
```

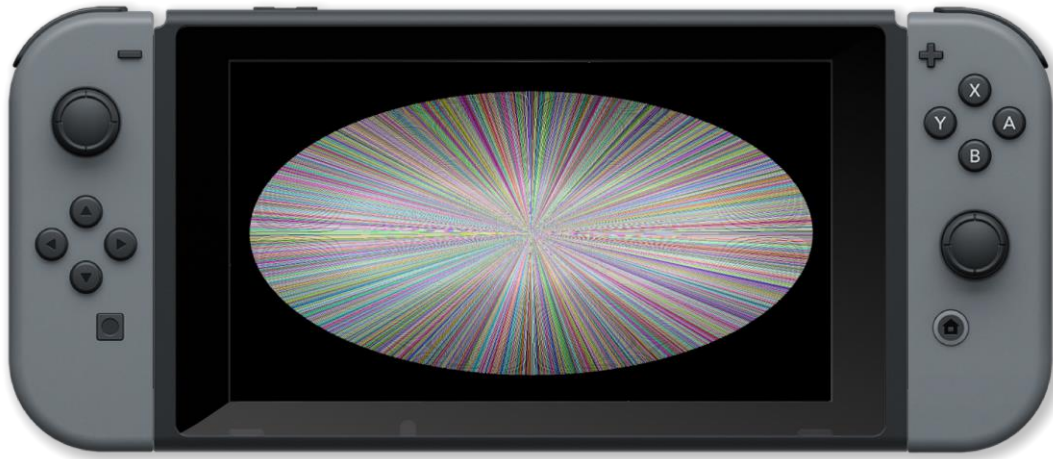
Arguments

enable If 1 (true) then the default unit for angles is switched to radians otherwise it is degrees

Example

```
// The 2 for loops are equivalent using different units for angles
clear()
centre = { gWidth() / 2, gHeight() / 2 }
radians( true )
for angle = 0 to 2 * pi step 0.017 loop
  col = { random( 101 ) / 100, random( 101 ) / 100, random( 101 ) / 100, 1.0 }
  point = { 600 * cos( angle ) + centre.x, 300 * sin( angle ) + centre.y }
  line( centre, point, col )
  update()
repeat

clear()
sleep( 3 )
radians( 0 )
for angle = 0 to 360 step 1 loop
  col = { random( 101 ) / 100, random( 101 ) / 100, random( 101 ) / 100, 1.0 }
  point = { 600 * cos( angle ) + centre.x, 300 * sin( angle ) + centre.y }
  line( centre, point, col )
  update()
repeat
sleep( 3 )
```



Associated Commands

`acos()`, `asin()`, `atan()`, `atan2()`, `pi()`, `sin()`, `sinCos()`, `tan()`

COMMAND REFERENCE

random()

Purpose

Returns a random number in the given range.

Description

This function returns a random number from 0 up to, but not including range.

Syntax

```
number = random( range )
```

Arguments

range The number of different random numbers that can be returned

number A random number between 0 and range -1

Example

```
roll = 0
clear()
image = loadImage( "Colin Brown/Dice", false )
size = tileSize( image, 0 )
for i = 1 to 10 loop
  clear()
  roll = random( 6 ) + 1
  x = size.x - ( size.x * ( roll % 2 ) )
  y = size.y * ( ceil( roll / 2 ) - 1 )
  drawImage( image, { x, y, size.x, size.y }, { 0, 0, size.x, size.y } )
  update()
  sleep( 0.3 )
repeat
printAt( 0, 15, "You rolled a ", roll )
update()
sleep( 3 )
```



Associated Commands

COMMAND REFERENCE

reflect()

Purpose

Find the reflection of a vector

Description

Find the reflection of a vector when it hits a surface

Syntax

```
result = reflect( incident, normal )
```

Arguments

result The resulting reflection vector { x, y, z, w }

incident incident vector { x, y, z, w }

normal normal vector orthogonal to the surface { x, y, z, w }

Example

```
radians( true )
image = loadImage( "Untied Games/Enemy small top C", false )
ship = createSprite( )
setSpriteImage( ship, image )
lastpos = { gwidth() / 2, gheight() / 2 }
setSpriteLocation( ship, lastpos )
setSpriteScale( ship, { 3, 3 } )
size = getSpriteSize( ship )
speed = { 400, 300 }

loop
  clear()
  c = controls( 0 )
  curpos = getSpriteLocation( ship )
  refx = 0
  refy = 0
  if curpos.x < size.x / 2 then
    refx = 1
  endif
  if curpos.y < size.y / 2 then
    refy = 1
  endif
  if curpos.x > gwidth() - size.x / 2 then
    refx = -1
  endif
  if curpos.y > gheight() - size.y / 2 then
    refy = -1
  endif
  speed = reflect( speed, { refx, refy } )
  setSpriteSpeed( ship, speed )
  if curpos != lastpos then
    setSpriteRotation( ship, -pi / 2 + atan2( curpos.y - lastpos.y, curpos.x - lastpos.x ) )
```

```
    lastpos = curpos
  endIf
  updateSprites()
  drawSprites()
  update()
repeat
```



Associated Commands

`cross()`, `dot()`, `length()`, `normalize()`, `refract()`

COMMAND REFERENCE

refract()

Purpose

Find the refraction of a vector

Description

Find the refraction of a vector when it passes through a surface

Syntax

```
result = refract( incident, normal, ior )
```

Arguments

result resulting refraction vector { x, y, z, w }

incident incident vector { x, y, z, w }

normal normal vector orthogonal to the surface { x, y, z, w }

ior index of refraction of the material from which the surface is made

Example

```
centre = { gWidth() / 2, gHeight() / 2 }
dir = { 1, 1 }
ior = 0.5

loop
  clear()
  c = controls( 0 )
  ior += c.lx * 0.1
  refractedDir = refract( dir, { 0, -1 }, ior )
  box( 0, centre.y, gWidth(), centre.x, { 0, 0, 1, 0.2 }, false )
  line( { centre.x - 250 * refractedDir.x, centre.y - 250 * dir.y }, centre, white )
  line( centre, { centre.x + 250 * refractedDir.x, centre.y + 250 * refractedDir.y }, white )
  printAt( 0, 0, "Move left Joy-Con Control stick left or right to adjust index of refraction" )
  printAt( 0, 1, "Index of refraction: " + ior )
  update()
repeat
```

Associated Commands

[cross\(\)](#), [dot\(\)](#), [length\(\)](#), [normalize\(\)](#), [reflect\(\)](#)

COMMAND REFERENCE

rnd()

Purpose

Returns a random number in the given range.

Description

This function returns a random number from 0 up to, but not including range.

Syntax

```
number = rnd( range )
```

Arguments

range The number of different random numbers that can be returned

number A random number between 0 and range -1

Example

```
roll = 0
clear()
image = loadImage( "Colin Brown/Dice", false )
size = tileSize( image, 0 )
for i = 1 to 10 loop
  clear()
  roll = rnd( 6 ) + 1
  x = size.x - ( size.x * ( roll % 2 ) )
  y = size.y * ( ceil( roll / 2 ) - 1 )
  drawImage( image, { x, y, size.x, size.y }, { 0, 0, size.x, size.y } )
  update()
  sleep( 0.3 )
repeat
printAt( 0, 15, "You rolled a ", roll )
update()
sleep( 3 )
```




Associated Commands

COMMAND REFERENCE

round()

Purpose

Round a floating point number

Description

Round a floating point number to the nearest integer

Syntax

```
result = round( value )
```

Arguments

value float value to be rounded

result nearest integer result (0.5 rounds up to 1)

Example

```
print( round( 3.1415926 ) ) // prints 3.000000
print( round( 3.5 ) )      // prints 4.000000
update()
sleep( 3 )
```

Associated Commands

`int()`, `float()`, `fract()`, `str()`, `trunc()`

COMMAND REFERENCE

rsqrt()

Purpose

Find the fast inverse square root of the specified number

Description

Returns the reciprocal (or multiplicative inverse) of the square root of a number. This operation is used in digital signal processing to normalize a vector (scale it to length 1.) e.g. in computer graphics inverse square roots are used to compute angles of incidence and reflection for lighting and shading.

Syntax

```
result = rsqrt( number )
```

Arguments

number The number to find the reciprocal square root of

result The reciprocal square root (1/square root) of the number

Example

```
num = 10  
print( rsqrt( num ) )  
update()  
sleep( 3 )
```

Associated Commands

`pow()`, `sqrt()`

COMMAND REFERENCE

sin()

Purpose

Returns the sine of the supplied argument.

Description

This is the sine function which returns the ratio of the side opposite to an acute angle in a right-angled triangle to the hypotenuse (longest side)

Syntax

```
sine = sin( angle )
```

Arguments

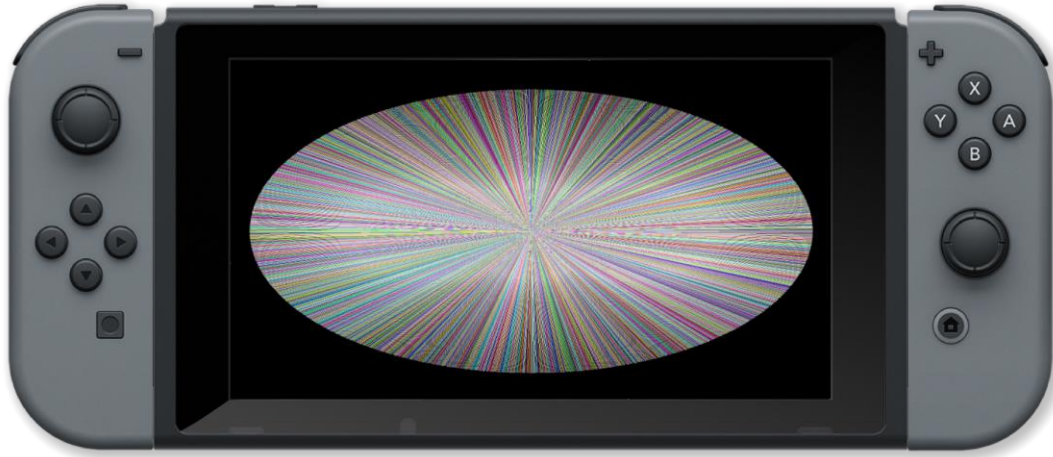
angle The acute angle opposite to the side of a right-angled triangle.

sine The ratio of the side opposite to an acute angle in a right-angled triangle to the hypotenuse.

Example

```
clear()
centre = { gWidth() / 2, gHeight() / 2 }
for angle = 0 to 360 loop
  // Pick random colour
  col = { random( 101 ) / 100, random( 101 ) / 100, random( 101 ) / 100, 1.0 }
  point = { 600 * cos( angle ) + centre.x, 300 * sin( angle ) + centre.y }
  line( centre, point, col )
repeat

for i = 0 to 100 loop
  update()
repeat
```



Associated Commands

`acos()`, `asin()`, `atan()`, `atan2()`, `pi()`, `radians()`, `sinCos()`, `tan()`

COMMAND REFERENCE

sinCos()

Purpose

Returns the sine and cosine of the supplied angle.

Description

This is the sine (SIN) and cosine (COS) functions combined. If you require both values it is more convenient than calling the two separately.

Syntax

```
result = sinCos( angle )
```

Arguments

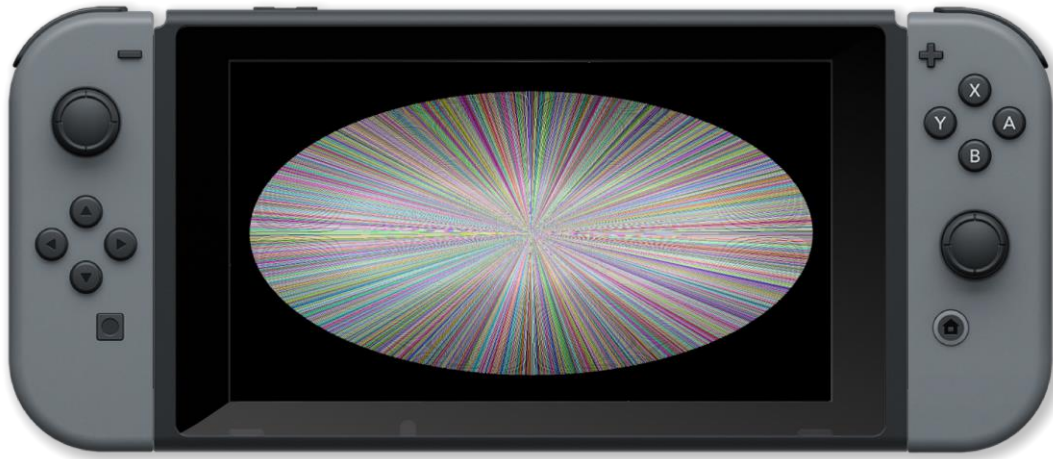
angle An acute angle in a right-angled triangle.

result A vector containing the sine (result.x) and cosine (result.y) values for the specified angle.

Example

```
clear()
centre = { gWidth() / 2, gHeight() / 2 }
for angle = 0 to 360 loop
  col = { random( 101 ) / 100, random( 101 ) / 100, random( 101 ) / 100, 1.0 }
  result = sinCos( angle )
  point = { 600 * result.y + centre.x, 300 * result.x + centre.y }
  line( centre, point, col )
repeat

for i = 1 to 100 loop
  update()
repeat
```



Associated Commands

`acos()`, `asin()`, `atan()`, `atan2()`, `pi()`, `radians()`, `sin()`, `sinCos()`, `tan()`

COMMAND REFERENCE

smoothStep()

Purpose

Hermite interpolation

Description

Perform Hermite interpolation between two values

Syntax

```
result = smoothStep( value0, value1, factor )
```

Arguments

result Hermite interpolation

value0 left edge value

value1 right edge value

factor interpolation factor

Example

```
col = { 0, 1, 1, 1 }  
// draw lines in a gradient fading from black to cyan  
for i = 0 to gWidth() loop  
  col.a = smoothStep( 0, 1, i / gWidth() )  
  line( { i, 0 }, { i, gHeight() }, col )  
repeat  
update()  
sleep( 3 )
```

Associated Commands

[bezier\(\)](#), [lerp\(\)](#)

COMMAND REFERENCE

sqrt()

Purpose

Find the square root of the specified number

Description

The mathematical square root function which is the number which when multiplied by itself will give you the specified number. If you try to find the square root of a negative number it will return nan (not a number)

Syntax

```
result = sqrt( number )
```

Arguments

number number to find the square root of

result square root of the number

Example

```
// Pythagoras: for a right angled triangle with side lengths a,b and c (where c is the longest side) a*a + b*b = c*c
a = 3
b = 4
c = sqrt( a * a + b * b )
print( "c = ", c )
update()
sleep( 3 )
```

Associated Commands

`pow()`, `rsqrt()`

COMMAND REFERENCE

tan()

Purpose

Returns the tangent of the supplied argument.

Description

This is the ratio of the length of the side opposite to the acute angle of a right angle triangle to the length of the side adjacent to it.

Syntax

```
tangent = tan( angle )
```

Arguments

angle The acute angle in a right-angled triangle for the given tangent

tangent The ratio of the side opposite to an acute angle in a right-angled triangle to the one adjacent.

Example

```
angle = 45
tangent = tan( angle )
print( "Tangent = ", tangent )
update()
sleep( 3 )
```

Associated Commands

[acos\(\)](#), [asin\(\)](#), [atan\(\)](#), [atan2\(\)](#), [pi\(\)](#), [radians\(\)](#), [sin\(\)](#), [sinCos\(\)](#)

COMMAND REFERENCE

trunc()

Purpose

Truncates a floating point number

Description

Returns the integer part of a floating point number

Syntax

```
result = trunc( value )
```

Arguments

value floating point value to be truncated

result integer part of a value

Example

```
printAt( 0, 0, trunc( 3.1415926 ) ) // prints 3.000000
printAt( 0, 1, trunc( 3.9 ) )      // prints 3.000000

for i = 1 to 100 loop
  update()
repeat
```

Associated Commands

[int\(\)](#), [float\(\)](#), [fract\(\)](#), [round\(\)](#), [str\(\)](#)

COMMAND REFERENCE

Binary

COMMAND REFERENCE

bitFieldExtract()

Purpose

Extract a number of bits from a binary number

Description

Extract the specified number of bits from a 32 bit signed binary number starting at the specified bit

Syntax

```
result = bitFieldExtract( number, start, count )
```

Arguments

number 32 bit signed binary number

start Start position (first position is 0)

count Number of bits to extract

result The specified bits

Example

```
number = 0
for i = 0 to 16 loop
  number = bitSet( number, i, 1 )
  lowbyte = bitFieldExtract( number, 0, 8 )
  highbyte = bitFieldExtract( number, 8, 8 )
  printAt( 0, i , number, " ", lowbyte, " ", highbyte )
  update()
repeat
sleep( 3 )
```

Associated Commands

[bitCount\(\)](#), [bitGet\(\)](#), [bitSet\(\)](#), [bitFieldInsert\(\)](#), [leadingZeroes\(\)](#), [trailingZeroes\(\)](#)

COMMAND REFERENCE

bitFieldInsert()

Purpose

Insert a number of bits into a binary number

Description

Insert the specified bits into a 32 bit signed binary number starting at the specified bit

Syntax

```
result = bitFieldInsert( number, start, count, value )
```

Arguments

number 32 bit signed binary number

start Start position (first position is 0)

count Number of bits to insert

value Bits to insert

result resulting number when bits have been inserted

Example

```
loop
  textSize( 50 )
  byte1 = 0
  byte2 = 15
  printAt( 0, 0, "byte1 = ", bin2str( byte1 ) )
  printAt( 0, 1, "byte2 = ", bin2str( byte2 ) )
  result = bitFieldInsert( byte1, 2, 4, byte2 )
  printAt( 0, 2, "bitFieldInsert( byte1, 2, 4, byte2 ) = ", bin2str( result ) )
  update()
repeat

function bin2str( byte )
  result = ""
  for i = 0 to 8 loop
    bit = byte & 1
    if bit then
      result = "1" + result
    else
      result = "0" + result
    endif
    byte = byte >> 1
```

```
repeat  
return result
```

Associated Commands

[bitCount\(\)](#), [bitGet\(\)](#), [bitSet\(\)](#), [bitFieldExtract\(\)](#), [bitFieldInsert\(\)](#), [leadingZeroes\(\)](#), [trailingZeroes\(\)](#)

COMMAND REFERENCE

bitGet()

Purpose

Get the value of a bit in a binary number

Description

Find out if the value of the specified bit in a binary number is 0 or 1

Syntax

```
result = bitGet( number, bit )
```

Arguments

number Number value

bit Bit to get the value of (0-31). Bit 31 is the sign bit.

result Resulting bit value (0 or 1)

Example

```
number = random( 100000 ) + 1
bit0 = bitGet( number, 0 )
if bit0 > 0 then
    print( number, " is an odd number" )
else
    print( number, " is an even number" )
endif
update()
sleep( 3 )
```

Associated Commands

[bitCount\(\)](#), [bitSet\(\)](#), [bitFieldExtract\(\)](#), [bitFieldInsert\(\)](#), [leadingZeroes\(\)](#), [trailingZeroes\(\)](#)

COMMAND REFERENCE

bitSet()

Purpose

Set or clear a bit in a binary number

Description

Set (1) or clear (0) the specified bit in a 32 bit signed binary number

Syntax

```
result = bitSet( number, bit, value )
```

Arguments

number Initial number value

bit Bit to set or clear (0-31). Bit 31 is the sign bit.

value 1 (set) or 0 (clear)

result Resulting number value

Example

```
number = 0
for i = 0 to 16 loop
  number = bitSet( number, i, 1 )
  printAt( 0, i , number )
  update()
repeat
sleep( 3 )
```

Associated Commands

[bitCount\(\)](#), [bitGet\(\)](#), [bitFieldExtract\(\)](#), [bitFieldInsert\(\)](#)

COMMAND REFERENCE

leadingZeroes()

Purpose

Find the number of leading zeroes in a binary number

Description

Find the number of leading zeroes in a 64 bit binary number

Syntax

```
result = leadingZeroes( value )
```

Arguments

value The number to find the number of leading zeroes in

result The number of leading zeroes in value

Example

```
for i = 0 to 16 loop
  j = pow( 2, i )
  printAt( 0, i, int( j ), " ", leadingZeroes( j ) )
  update()
repeat
sleep( 3 )
```

Associated Commands

[bitCount\(\)](#), [bitGet\(\)](#), [bitSet\(\)](#), [bitFieldExtract\(\)](#), [bitFieldInsert\(\)](#), [trailingZeroes\(\)](#)

COMMAND REFERENCE

trailingZeroes()

Purpose

Find the number of trailing zeroes in a binary number

Description

Find the number of trailing zeroes in a 64 bit binary number

Syntax

```
result = trailingZeroes( value )
```

Arguments

value The number to find the number of trailing zeroes in

result The number of trailing zeroes in value

Example

```
for i = 0 to 16 loop
  j = pow( 2, i )
  printAt( 0, i, int(j), " ", trailingZeroes( j ) )
  update()
repeat
sleep( 3 )
```

Associated Commands

[bitCount\(\)](#), [bitGet\(\)](#), [bitSet\(\)](#), [bitFieldExtract\(\)](#), [bitFieldInsert\(\)](#), [leadingZeroes\(\)](#)

COMMAND REFERENCE

File Handling

COMMAND REFERENCE

close()

Purpose

Closes a file

Description

Closes a file previously opened with Open

Syntax

```
close( handle )
```

Arguments

handle The handle of the file

Example

```
string = "Hello World"
printAt( 0, 0, "Open file" )
handle = open()
printAt( 0, 1, "Write to file: ", string )
write( handle, string )
printAt( 0, 2, "Close file" )
close( handle )
printAt( 0, 3, "Open file" )
handle = open()
message = read( handle, 11 )
printAt( 0, 4, "Read from file: ", message )
printAt( 0, 5, "Seek to position 6" )
seek( handle, 6 )
message = read( handle, 5 )
printAt( 0, 6, "Read from file: ", message )
printAt( 0, 7, "Close file" )
update()
close( handle )
for i = 1 to 500 loop
    update()
repeat
```

Associated Commands

`close()`, `open()`, `read()`, `seek()`, `write()`

COMMAND REFERENCE

open()

Purpose

Open a file for read or write

Description

Opens a text file for reading or writing and returns a handle

Syntax

```
handle = open( )
```

Arguments

handle The handle of the file

Example

```
string = "Hello World"
printAt( 0, 0, "Open file" )
handle = open()
printAt( 0, 1, "Write to file: ", string )
write( handle, string )
printAt( 0, 2, "Close file" )
close( handle )
printAt( 0, 3, "Open file" )
handle = open()
message = read( handle, 11 )
printAt( 0, 4, "Read from file: ", message )
printAt( 0, 5, "Seek to position 6" )
seek( handle, 6 )
message = read( handle, 5 )
printAt( 0, 6, "Read from file: ", message )
printAt( 0, 7, "Close file" )
update()
close( handle )
for i = 1 to 500 loop
    update()
repeat
```

Associated Commands

`close()`, `open()`, `read()`, `seek()`, `write()`

COMMAND REFERENCE

read()

Purpose

Read from a file

Description

Read a string of text from a file from the current position the specified number of characters

Syntax

```
result = read( handle, count )
```

Arguments

handle handle of the file

count number of characters to read

result resulting text string

Example

```
string = "Hello World"
printAt( 0, 0, "Open file" )
handle = open()
printAt( 0, 1, "Write to file: ", string )
write( handle, string )
printAt( 0, 2, "Close file" )
close( handle )
printAt( 0, 3, "Open file" )
handle = open()
message = read( handle, 11 )
printAt( 0, 4, "Read from file: ", message )
printAt( 0, 5, "Seek to position 6" )
seek( handle, 6 )
message = read( handle, 5 )
printAt( 0, 6, "Read from file: ", message )
printAt( 0, 7, "Close file" )
update()
close( handle )
for i = 1 to 500 loop
    update()
repeat
```

Associated Commands

[close\(\)](#), [open\(\)](#), [seek\(\)](#), [write\(\)](#)

COMMAND REFERENCE

seek()

Purpose

Seek to a position in the file

Description

Move the position to the point specified in the file specified by handle

Syntax

```
Seek( handle, position )
```

Arguments

handle handle of the file

position position to seek to (0 is the start)

Example

```
string = "Hello World"
printAt( 0, 0, "Open file" )
handle = open()
printAt( 0, 1, "Write to file: ", string )
write( handle, string )
printAt( 0, 2, "Close file" )
close( handle )
printAt( 0, 3, "Open file" )
handle = open()
message = read( handle, 11 )
printAt( 0, 4, "Read from file: ", message )
printAt( 0, 5, "Seek to position 6" )
seek( handle, 6 )
message = read( handle, 5 )
printAt( 0, 6, "Read from file: ", message )
printAt( 0, 7, "Close file" )
update()
close( handle )
for i = 1 to 500 loop
    update()
repeat
```

Associated Commands

`close()`, `open()`, `read()`, `seek()`, `write()`

COMMAND REFERENCE

write()

Purpose

Write to a file

Description

Write a string of text to the file specified by handle

Syntax

```
write( handle, text )
```

Arguments

handle The handle of the file

text The string of text to be written

Example

```
string = "Hello World"
printAt( 0, 0, "Open file" )
handle = open()
printAt( 0, 1, "Write to file: ", string )
write( handle, string )
printAt( 0, 2, "Close file" )
close(handle)
printAt( 0, 3, "Open file" )
handle = open()
message = read( handle, 11 )
printAt( 0, 4, "Read from file: ", message )
printAt( 0, 5, "Seek to position 6" )
seek( handle, 6 )
message = read( handle, 5 )
printAt( 0, 6, "Read from file: ", message )
printAt( 0, 7, "Close file" )
update()
close( handle )

for i = 1 to 500 loop
    update()
repeat
```

Associated Commands

`close()`, `open()`, `read()`, `seek()`, `write()`

COMMAND REFERENCE

Input

COMMAND REFERENCE

controls()

Purpose

Read all values from the Joy-Con controllers

Description

Used to establish the state of buttons, position of control sticks and motion sensors. It is advisable not to read the value of this function directly but to assign to to a variable first.

Syntax

```
c = controls( index )
```

Arguments

index Index of the controller (0 to 3)

c.a State of the A button (0 or 1)

c.b State of the B button (0 or 1)

c.x State of the X button (0 or 1)

c.y State of the Y button (0 or 1)

c.up State of the up (^) button (0 or 1)

c.down State of the down (v) button (0 or 1)

c.left State of the left (<) button (0 or 1)

c.right State of the right (>) button (0 or 1)

c.l State of the L button (0 or 1)

c.r State of the R button (0 or 1)

c.zl State of the ZL button (0 or 1)

c.zr State of the ZR button (0 or 1)

c.lx Horizontal position of the left control stick (-1 to 1)

c.ly Vertical position of the left control stick (-1 to 1)

c.rx Horizontal position of the right control stick (-1 to 1)

c.ry Vertical position of the right control stick (-1 to 1)

c.lc State of the left stick press (0 or 1)

c.rc State of the right stick press (0 or 1)

c.velocity[0] Velocity of the left controller (or both if connected) $\{x, y, z\}$

c.velocity[1] Velocity of the right controller (if disconnected) $\{x, y, z\}$

c.orientation[0] Orientation of the left controller (or both if connected) $\{x, y, z\}$

c.orientation[1] Orientation of the right controller (if disconnected) $\{x, y, z\}$

Example

```
loop
  clear()
  c = controls( 0 )
  printAt( 0, 0, "zl: ", c.zl )
  printAt( 0, 1, "l: ", c.l )
  printAt( 0, 3, "lx: ", c.lx )
  printAt( 0, 4, "ly: ", c.ly )
  printAt( 35, 0, "zr: ", c.zr )
  printAt( 35, 1, "r: ", c.r )
  printAt( 38, 4, "A: ", c.a )
  printAt( 35, 5, "B: ", c.b )
  printAt( 35, 3, "X: ", c.x )
  printAt( 32, 4, "Y: ", c.y )
  printAt( 3, 11, "^: ", c.up )
  printAt( 3, 13, "v: ", c.down )
  printAt( 0, 12, "<: ", c.left )
  printAt( 6, 12, "> ", c.right )
  printAt( 33, 11, "rx: ", c.rx )
  printAt( 33, 12, "ry: ", c.ry )
  printAt( 0, 6, "lc: ", c.lc )
  printAt( 33, 14, "rc: ", c.rc )
  printAt( 0, 18, "velocity (l): ", c.velocity[0] )
  printAt( 0, 18, "velocity (r): ", c.velocity[1] )
  printAt( 0, 18, "orientation (l): ", c.orientation[0] )
  printAt( 0, 18, "orientation (r): ", c.orientation[1] )
  update()
repeat
```



Associated Commands

`input()`, `docked()`, `getKeyboardBuffer()`, `showKeyboard()`, `touch()`, `hideKeyboard()`

COMMAND REFERENCE

docked()

Purpose

Find out if the console is in TV mode or handheld mode

Description

Returns true (1) if the console is in TV mode and false (0) if the console is in handheld mode

Syntax

```
result = docked( )
```

Arguments

** True (1) if the console is in TV mode and false (0) if the console is in handheld mode

Example

```
loop
  clear()
  dock = docked()
  if dock then
    print( "Console is in TV mode")
  else
    print( "Console is in handheld mode" )
  endIf
  update()
repeat
```

Associated Commands

[input\(\)](#), [controls\(\)](#), [getKeyboardBuffer\(\)](#), [showKeyboard\(\)](#), [touch\(\)](#), [hideKeyboard\(\)](#)

COMMAND REFERENCE

getKeyboardBuffer()

Purpose

Get the contents of the keyboard buffer

Description

Returns a string containing key presses since the last call to it

Syntax

```
result = getKeyboardBuffer( )
```

Arguments

result string containing key presses since the last call

Example

```
showKeyboard()  
loop  
  c = getKeyboardBuffer()  
  print( c )  
  update()  
repeat
```

Associated Commands

[input\(\)](#), [controls\(\)](#), [docked\(\)](#), [showKeyboard\(\)](#), [touch\(\)](#), [hideKeyboard\(\)](#)

COMMAND REFERENCE

hideKeyboard()

Purpose

Show or the virtual keyboard

Description

Hide the virtual keyboard.

Syntax

```
hideKeyboard( )
```

Arguments

Example

```
loop
  c = controls( 0 )
  printAt( 0, 0, "Press A to show keyboard" )
  printAt( 0, 1, "Press B to hide keyboard" )
  if c.a then
    showKeyboard()
  endIf
  if c.b then
    hideKeyboard()
  endIf
  update()
repeat
```

Associated Commands

[input\(\)](#), [controls\(\)](#), [docked\(\)](#), [getKeyboardBuffer\(\)](#), [showKeyboard\(\)](#), [touch\(\)](#)

COMMAND REFERENCE

input()

Purpose

Allow text input from the keyboard

Description

Read text input from the keyboard into a variable. The virtual keyboard will be shown automatically

Syntax

```
result = input( prompt, multiline )
```

Arguments

prompt text to display above the input box

multiline allow multiple line input

result variable to store the entered text

Example

```
name = input( "What is your name?", false )
printAt( 0, 0, "Hello ", name )
update()
sleep( 3 )
```

Associated Commands

controls(), docked(), getKeyboardBuffer(), showKeyboard(), touch(), hideKeyboard()

COMMAND REFERENCE

showKeyboard()

Purpose

Show the virtual keyboard

Description

Display the virtual keyboard on the screen.

Syntax

```
showKeyboard( )
```

Arguments

Example

```
loop
  c = controls( 0 )
  printAt( 0, 0, "Press A to show keyboard" )
  printAt( 0, 1, "Press B to hide keyboard" )
  if c.a then
    showKeyboard()
  endIf
  if c.b then
    hidekeyboard()
  endIf
  update()
repeat
```

Associated Commands

[input\(\)](#), [controls\(\)](#), [docked\(\)](#), [getKeyboardBuffer\(\)](#), [touch\(\)](#), [hideKeyboard\(\)](#)

COMMAND REFERENCE

touch()

Purpose

Read input from the touch screen

Description

Read the coordinates of up to ten points on the touch screen that have been touched

Syntax

```
list = touch( )
```

Arguments

list An array of up to 10 points that are currently being touched

Example

```
list = []
loop
  clear()
  printAt( 0, 0, "Touch the screen" )
  list = touch()
  count = len( list )
  if count > 0 then
    printAt( 0, 1, "count = ", count, " x0 = ", list[0].x, " y0 = ", list[0].y )
    for i = 0 to count loop
      circle( list[i].x, list[i].y, 75, 100, red, 0 )
    repeat
  endif
  update()
repeat
```

Associated Commands

[input\(\)](#), [controls\(\)](#), [docked\(\)](#), [getKeyboardBuffer\(\)](#), [showKeyboard\(\)](#), [touch\(\)](#), [hideKeyboard\(\)](#)

COMMAND REFERENCE

Screen Display

COMMAND REFERENCE

clear()

Purpose

Clear the screen

Description

Clear the framebuffer and after a call to update the screen. Optionally a colour can be specified to fill the screen with (default is black)

Syntax

```
clear( )  
clear( colour )
```

Arguments

colour colour name or RGB values { red, green, blue, opacity } between 0 and 1

Example

```
showKeyboard()  
clear( { 1, 1, 1, 1 } )  
update()  
sleep( 3 )  
clear( red )  
update()  
sleep( 3 )  
clear()  
update()  
sleep( 3 )
```

Associated Commands

[gHeight\(\)](#), [gWidth\(\)](#), [setDrawTarget\(\)](#), [setMode\(\)](#), [update\(\)](#)

COMMAND REFERENCE

Colours

Below is a chart containing the names for each preset colour in **FUZE⁴ Nintendo Switch**.

All colour names are labels for RGBA (red, blue, green, alpha) vectors. For example, the name `green` is secretly the vector: `{ 0, 1, 0, 1 }`. The first number is the amount of red, next is the amount of green, then the amount of blue and finally we have the transparency.

To find the RGBA values for any of the colours below, simply use the `print()` function to display them! For example, you could write `print(fuzePink)` to display the RGBA values of the lovely `fuzePink` colour.

Here you go, enjoy!

white	rust	lime
platinum	bronze	emerald
lightGrey	coffee	jade
silver	ginger	pine
taupe	caramel	teal
slate	deepOrange	cadetBlue
grey	pumpkin	turquoise
darkGrey	darkOrange	aquamarine
charcoal	orange	aqua
fuzeGrey	darkGold	cyan
black	yellowOchre	lightCyan
maroon	amber	powderBlue
darkRed	gold	babyBlue
burgundy	banana	lightBlue
raspberry	cadmiumYellow	skyBlue
red	yellow	fuzeBlue
scarlet	lemon	cornFlower
crimson	lkhaki	azure

COMMAND REFERENCE

gHeight()

Purpose

Get the height of the screen display

Description

Returns the number of vertical pixels in the screen display

Syntax

```
height = gHeight( )
```

Arguments

height The height of the screen display in pixels

Example

```
// Scale an image to fit to the screen
img = loadImage( "Colin Brown/DungeonB", false )
size = imageSize( img )
scale = min( gWidth() / size.x, gHeight() / size.y )
drawImage( img, 0, 0, scale )
update()
sleep( 3 )
```



Associated Commands

`clear()`, `gWidth()`, `setDrawTarget()`, `setMode()`, `update()`

COMMAND REFERENCE

gWidth()

Purpose

Get the width of the screen display

Description

Returns the number of horizontal pixels in the screen display

Syntax

```
width = gWidth( )
```

Arguments

width The width of the screen display in pixels

Example

```
// Scale an image to fit to the screen
img = loadImage( "Colin Brown/DungeonB", false )
size = imageSize( img )
scale = min( gWidth() / size.x, gHeight() / size.y )
drawImage( img, 0, 0, scale )
update()
sleep( 3 )
```



Associated Commands

`clear()`, `gHeight()`, `setDrawTarget()`, `setMode()`, `update()`

COMMAND REFERENCE

setDrawTarget()

Purpose

Sets the target of draw commands

Description

Sets the target of draw commands to be an image or the framebuffer

Syntax

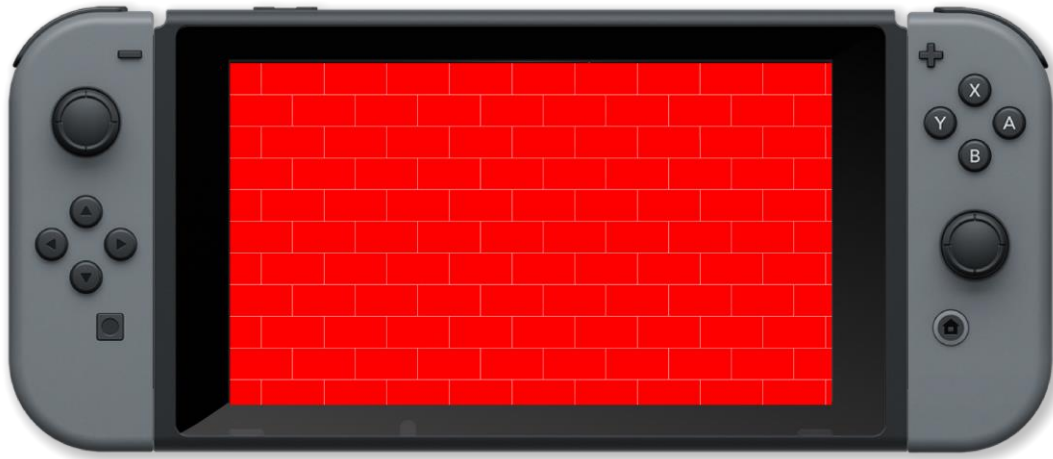
```
setDrawTarget( target )
```

Arguments

target handle of an image created with createimage or framebuffer for the screen

Example

```
w = 200
// Create a tile
img = createImage( w, w, true, image_rgb )
setDrawTarget( img )
box( 0, 0, w, w, red, 0 )
box( 0, w/2, w - 1, w / 2, white, 1 )
line( { w / 2 }, { w / 2, w / 2 }, white )
// draw tiles on the screen
setDrawTarget( framebuffer )
for y = 1 to gHeight() step w loop
  for x = 1 to gWidth() step w loop
    drawImage( img, x, y, 1 )
    update()
    sleep( 0.2 )
  repeat
repeat
sleep( 3 )
```



Associated Commands

`clear()`, `gHeight()`, `gWidth()`, `getDrawTarget()`, `setMode()`, `update()`

COMMAND REFERENCE

setMode()

Purpose

Set the resolution of the screen display

Description

Specify the number of horizontal and vertical pixels in the screen display. The default is 1280 x 720.

Syntax

```
setMode( width, height )
```

Arguments

width width of the screen display in pixels (default is 1280)

height height of the screen display in pixels (default is 720)

Example

```
print( gWidth(), " ", gHeight() )
update()
sleep( 3 )
clear()
setMode( 800, 600 )
print( gWidth(), " ", gHeight() )
update()
sleep( 3 )
```

Associated Commands

[clear\(\)](#), [gHeight\(\)](#), [gWidth\(\)](#), [setDrawTarget\(\)](#), [setMode\(\)](#), [update\(\)](#)

COMMAND REFERENCE

update()

Purpose

Update screen graphics

Description

Render the current frame buffer to the screen. This means that a lot of drawing operations can be performed without slowing down performance

Syntax

```
update( )
```

Arguments

Example

```
loop
  clear()
  col = { random( 101 ) / 100, random( 101 ) / 100, random( 101 ) / 100, 1 }
  x = random( gWidth() )
  y = random( gHeight() )
  for w = 0 to gWidth() step 5 loop
    line( { x, y }, { w, 0 }, col )
    line( { x, y }, { w, gHeight() }, col )
  repeat
  for h = 0 to gHeight() step 5 loop
    line( { x, y }, { 0, h }, col )
    line( { x, y }, { gWidth(), h }, col )
  repeat
  for i = 1 to 500 loop
    update()
  repeat
repeat
```



Associated Commands

`clear()`, `gHeight()`, `gWidth()`, `setDrawTarget()`, `setMode()`

COMMAND REFERENCE

Sound and Music

COMMAND REFERENCE

audioLength()

Purpose

Find the length of an audio sample

Description

Get the audio length for given handle

Syntax

```
length = audioLength( handle )
```

Arguments

handle handle of the audio sample from loadaudio

length length of the audio sample

Example

```
handle = loadAudio( "DavidSilvera/jingle_level_complete_03" )
length = audioLength( handle )
volume = 0.5
pan = 0.5
speed = 1
playAudio( 0, handle, volume, pan, speed, 0 )
start = time()
elapsed = 0
// wait for audio to finish
while elapsed < length loop
    clear()
    now = time()
    elapsed = now - start
    printAt( 0, 0, length, " ", elapsed )
    update()
repeat
printAt( 0, 1, "Finished" )
update()
sleep( 3 )
```

Associated Commands

getChannelStatus(), loadAudio(), note2Freq(), playAudio(), playNote(), setClipper(), setFilter(), setPan(), setVolume(), startChannel(), stopChannel()

COMMAND REFERENCE

getChannelStatus()

Purpose

Find the status of an audio channel

Description

Check to see if audio is being played from a given channel

Syntax

```
status = getChannelStatus( channel )
```

Arguments

channel audio channel from 0 to 15

status status of the channel (in use or not)

Example

```
handle = loadAudio( "DavidSilvera/jingle_level_complete_03" )
volume = 0.5
pan = 0.5
speed = 1
playAudio( 0, handle, volume, pan, speed, 0 )
start = time()
elapsed = 0
// wait for audio to finish
status = getChannelStatus( 0 )
while status loop
    clear()
    now = time()
    elapsed = now - start
    printAt( 0, 0, elapsed )
    update()
    status = getChannelStatus( 0 )
repeat
printAt( 0, 1, "Finished" )
update()
sleep( 3 )
```

Associated Commands

audioLength(), loadAudio(), note2Freq(), playAudio(), playNote(), setPan(), setVolume(), startChannel(), stopChannel()

COMMAND REFERENCE

loadAudio()

Purpose

Load an audio sample

Description

Load the specified audio sample and return a handle

Syntax

```
handle = loadAudio( sample )
```

Arguments

handle handle of the audio sample

sample path to the audio sample in the media library

Example

```
handle = loadAudio( "DavidSilvera/music_evil_presence" )
volume = 0.5
pan = 0.5
speed = 1
playAudio( 0, handle, volume, pan, speed, -1 )
loop
    update()
repeat
```

Associated Commands

[audioLength\(\)](#), [getChannelStatus\(\)](#), [note2Freq\(\)](#), [playAudio\(\)](#), [playNote\(\)](#), [setClipper\(\)](#), [setFilter\(\)](#), [setPan\(\)](#), [setVolume\(\)](#), [startChannel\(\)](#), [stopChannel\(\)](#)

COMMAND REFERENCE

note2Freq()

Purpose

Find the frequency of the specified note

Description

Finds the corresponding frequency for the specified musical note

Syntax

```
frequency = note2Freq( note )
```

Arguments

frequency frequency 10Hz to 20Khz (10 - 20000)

note Int 0 to 128. 0 is C-1, 60 is C4 (Middle C)

Example

```
freq = []
for i = 0 to 7 loop
    freq[i] = note2Freq( i * 2 + 40 )
repeat

for i = 0 to 7 loop
    clear()
    printAt( 0, 0, "Frequency: ", freq[i] )
    update()
    playNote( 0, 0, freq[i], 0.5, 1, 0.5 )
    sleep( 1 )
repeat
```

Associated Commands

[audioLength\(\)](#), [getChannelStatus\(\)](#), [loadAudio\(\)](#), [playAudio\(\)](#), [playNote\(\)](#), [setClipper\(\)](#), [setFilter\(\)](#), [setPan\(\)](#), [setVolume\(\)](#), [startChannel\(\)](#), [stopChannel\(\)](#)

COMMAND REFERENCE

playAudio()

Purpose

Play an audio track

Description

Play the specified audio track at the specified volume, pan and speed on the specified channel

Syntax

```
playAudio( channel, handle, volume, pan, speed, loops )
```

Arguments

channel audio channel from 0 to 15

handle handle of the audio sample from loadaudio

volume volume from 0 to 1

pan stereo pan from left (0) to right (1)

speed speed multiplier (1 is normal speed)

loops number of times to repeat (-1 is forever)

Example

```
handle = loadAudio( "DavidSilvera/music_evil_presence" )
volume = 0.5
pan = 0.5
speed = 1
playAudio( 0, handle, volume, pan, speed, -1 )
loop
  update()
repeat
```

Associated Commands

audioLength(), getChannelStatus(), loadAudio(), note2Freq(), playNote(), setClipper(), setFilter(), setPan(), setVolume(), startChannel(), stopChannel()

COMMAND REFERENCE

playNote()

Purpose

Play a musical note

Description

Play a note of the specified frequency, volume and speed on the specified channel

Syntax

```
playNote( channel, wave, frequency, volume, speed, pan )
```

Arguments

channel audio channel from 0 to 15

wave wave type (0 = Square, 1 = Saw, 2 = Triangle, 3 = Sine, 4 = Noise)

frequency frequency 10Hz to 20Khz (10 - 20000)

volume volume from 0 to 1

speed speed to reach peak volume (higher is faster)

pan stereo pan from left (0) to right (1)

Example

```
pan = 0.5
vol = 0.5
for f = 10 to 20000 step 10 loop
  clear()
  printAt( 0, 0, "Left joypad: left right to pan, up down to raise lower volume" )
  printAt( 0, 1, "Frequency: ", f, " Pan: ", pan, " Volume: ", vol )
  c = controls( 0 )
  if c.left then // left to pan left
    pan = pan - 0.01
  endIf
  if c.right then // right to pan right
    pan = pan + 0.01
  endIf
  if c.up then // up to increase volume
    vol = vol + 0.01
  endIf
  if c.down then // down to decrease volume
    vol = vol - 0.01
  endIf
  pan = max( pan, 0 )
  pan = min( pan, 1 )
  vol = max( vol, 0 )
  vol = min( vol, 1 )
end
```

```
playNote( 0, 0, f, vol, 0.5, pan )  
sleep( 0.05 )  
update()  
repeat
```

Associated Commands

audioLength(), getChannelStatus(), loadAudio(), note2Freq(), playAudio(), setClipper(),
setFilter(), setPan(), setVolume(), startChannel(), stopChannel()

COMMAND REFERENCE

pulseRumble()

Purpose

Pulse the controller rumble motors

Description

Pulse one of the 4 controller motors at the specified frequency, speed and volume

Syntax

```
pulseRumble( controller, channel, speed, volume, frequency )
```

Arguments

controller identifier for the controller (0 is the first)

channel identifier for the motor (0 - top left, 1 - bottom left, 2 - top right, 3 -bottom right)

speed The speed of the pulse - this affects the duration

volume volume (amplitude) of vibration (0 - 1)

frequency frequency of vibration

Example

```
volume = 1
speed = 1
loop
  clear()
  printAt( 0, 0, "Press X to pulse motor 0" )
  printAt( 0, 1, "Press Y to pulse motor 1" )
  printAt( 0, 2, "Press A to pulse motor 2" )
  printAt( 0, 3, "Press B to pulse motor 3" )
  c = controls( 0 )
  motor = -1
  if c.x then
    motor = 0
  endIf
  if c.y then
    motor = 1
  endIf
  if c.a then
    motor = 2
  endIf
  if c.b then
    motor = 3
  endIf
```

```
if motor > -1 then
  pulseRumble( 0, motor, volume, speed, motor * 100 + 100 )
  sleep( 0.2 )
endIf
update()
repeat
```

Associated Commands

COMMAND REFERENCE

setClipper()

Purpose

Set audio clipper

Description

Any sound above the threshold will be attenuated towards it proportional to its level above it. The strength parameter affects the severity of attenuation. A high value will result in stronger attenuation towards threshold, a low value results in weaker. A Very high value will simply clip any audio above the threshold to it

Syntax

```
setClipper( channel, threshold, strength )
```

Arguments

channel audio channel from 0 to 15

threshold threshold in decibels

strength strength of attenuation to the threshold

Example

```
tune = loadAudio( "DavidSilvera/music_funky_bar" )
threshold = 1
strength = 50
playAudio( 0, tune, 1, 0.5, 1, -1 )

loop
  clear()
  c = controls( 0 )
  threshold += c.ly * 0.1
  strength += c.ry * 0.1
  threshold = max( threshold, 0 )
  setClipper( 0, threshold, strength )
  printAt( 0, 0, "Move the left control stick up or down to control threshold" )
  printAt( 0, 1, "Move the right control stick up or down to control strength" )
  printAt( 0, 3, "Threshold: " + threshold )
  printAt( 0, 5, "Attenuation Strength: " + strength )
  update()
repeat
```

Associated Commands

audioLength(), getChannelStatus(), loadAudio(), note2Freq(), playAudio(), playNote(), setFilter(), setPan(), setVolume(), startChannel(), stopChannel()

COMMAND REFERENCE

setEnvelope()

Purpose

Set audio envelope

Description

Set an envelope on an audio channel

Syntax

```
setEnvelope( channel, speed )
```

Arguments

channel audio channel from 0 to 15

speed speed of the envelope

Example

```
envelopeSpeed = 0.01
clip = loadAudio( "DavidSilvera/animals_sheep_3" )
playAudio( 0, clip, 4, 0.5, 1, -1 )

loop
  clear()
  c = controls( 0 )
  envelopeSpeed += c.ly * 0.05
  envelopeSpeed = clamp( envelopeSpeed, 0, 0.3 )
  printAt( 0, 0, "Move Joy-Con left control stick up or down to adjust envelope speed" )
  printAt( 0, 1, "Envelope Speed: " + envelopeSpeed )
  setEnvelope( 0, envelopeSpeed )
  update()
repeat
```

Associated Commands

audioLength(), getChannelStatus(), loadAudio(), note2Freq(), playAudio(), playNote(), setClipper(), setFilter(), setPan(), setVolume(), startChannel(), stopChannel()

COMMAND REFERENCE

setFilter()

Purpose

Set audio filter

Description

Applies a filter to the given audio channel

Syntax

```
setFilter( channel, type, cutoff )
```

Arguments

channel audio channel from 0 to 15

type off (0), lowpass (1) - all frequencies above cutoff cut, highpass (2) - all frequencies below cut

cutoff frequency at which the filter is applied

Example

```
tune = loadAudio( "DavidSilvera/music_disco_funky" )
type = 0
lowpass = 1
highpass = 2
cutoff = 200
press = false
playAudio( 0, tune, 1, 0.5, 1, -1 )

loop
  clear()
  c = controls( 0 )
  if c.a and !press then
    type = lowpass
  endif
  if c.b and !press then
    type = highpass
  endif
  if !c.a and !c.b then
    press = false
  endif
  cutoff += c.ly * 10
  cutoff = max( cutoff, 0 )
  setFilter( 0, type, cutoff )
  printAt( 0, 0, "Move the control stick up and down to adjust filter cutoff frequency" )
  printAt( 0, 2, "Press the A button to select low-pass filter" )
  printAt( 0, 3, "Press the B button to select high-pass filter" )
  printAt( 0, 5, "Filter cutoff frequency: " + cutoff )
  printAt( 0, 6, "Filter type: " + type )
```

```
update()  
repeat
```

Associated Commands

[audioLength\(\)](#), [getChannelStatus\(\)](#), [loadAudio\(\)](#), [note2Freq\(\)](#), [playAudio\(\)](#), [playNote\(\)](#),
[setClipper\(\)](#), [setPan\(\)](#), [setVolume\(\)](#), [startChannel\(\)](#), [stopChannel\(\)](#)

COMMAND REFERENCE

setFrequency()

Purpose

Set audio frequency

Description

Set the frequency on an audio channel

Syntax

```
setFrequency( channel, frequency )
```

Arguments

channel audio channel from 0 to 15

frequency frequency of the sound

Example

```
freq = 432
playNote( 0, 1, freq, 1, 0, 0.5 )

loop
  clear()
  c = controls( 0 )
  freq += c.ly
  freq = clamp( freq, 0, 20000 )
  setFrequency( 0, freq )
  printAt( 0, 0, "Move the left control stick up and down to control frequency of the note" )
  printAt( 0, 2, "Frequency: " + freq )
  update()
repeat
```

Associated Commands

[audioLength\(\)](#), [getChannelStatus\(\)](#), [loadAudio\(\)](#), [note2Freq\(\)](#), [playAudio\(\)](#), [playNote\(\)](#), [setClipper\(\)](#), [setFilter\(\)](#), [setPan\(\)](#), [setVolume\(\)](#), [startChannel\(\)](#), [stopChannel\(\)](#)

COMMAND REFERENCE

setModulator()

Purpose

Set audio modulator

Description

Set a modulator on an audio channel

Syntax

```
setModulator( channel, wave, frequency, scale )
```

Arguments

channel audio channel from 0 to 15

wave wave type (0 = Square, 1 = Saw, 2 = Triangle, 3 = Sine, 4 = Noise)

frequency frequency 10Hz to 20Khz (10 - 20000)

scale modulator scale

Example

```
waveType = [ "Square", "Saw", "Triangle", "Sine", "Noise" ]
w = 0
press = false
modFreq = 0
modScale = 0
playNote( 0, 3, 432, 1, 0, 0.5 )

loop
  clear()
  c = controls( 0 )
  if c.right and !press then
    w += 1
    press = false
  endif
  if c.left and !press then
    w -= 1
    press = false
  endif
  if !c.right and !c.left then
    press = false
  endif
  modFreq += c.ly
  modScale += c.ry
  w = clamp( w, 0, 4 )
  modFreq = max( modFreq, 0 )
  modScale = max( modScale, 0 )
  setModulator( 0, w, modFreq, modScale )
  printAt( 0, 0, "Press left or right directional buttons to change modulation wave type" )
  printAt( 0, 1, "Wave Type: " + waveType[w] )
  printAt( 0, 3, "Move the left control stick up or down to adjust modulation frequency" )
```

```
printAt( 0, 4, "Modulation Frequency: " + modFreq )
printAt( 0, 6, "Move the right control stick up or down to adjust modulation scale" )
printat( 0, 7, "Modulation Scale: " + modScale )
update()
repeat
```

Associated Commands

[audioLength\(\)](#), [getChannelStatus\(\)](#), [loadAudio\(\)](#), [note2Freq\(\)](#), [playAudio\(\)](#), [playNote\(\)](#),
[setClipper\(\)](#), [setFilter\(\)](#), [setPan\(\)](#), [setVolume\(\)](#), [startChannel\(\)](#), [stopChannel\(\)](#)

COMMAND REFERENCE

setPan()

Purpose

Change the audio stereo position

Description

Set the stereo position for an audio channel

Syntax

```
setPan( channel, pan )
```

Arguments

channel audio channel from 0 to 15

pan stereo pan from left (0) to right (1)

Example

```
handle = loadAudio( "DavidSilvera/music_evil_presence" )
vol = 0.5
pan = 0.5
speed = 1
playAudio( 0, handle, vol, pan, speed, -1 )

loop
  clear()
  c = controls( 0 )
  printat( 0, 0, "Use left arrow to pan left" )
  if c.left then
    pan -= 0.01
  endif
  printat( 0, 1, "Use right arrow to pan left" )
  if c.right then
    pan += 0.01
  endif
  pan = max( pan, 0 )
  pan = min( pan, 1 )
  setPan( 0, pan )
  printat( 0, 2, "Pan: ", pan )
  update()
repeat
```

Associated Commands

audioLength(), getChannelStatus(), loadAudio(), note2Freq(), playAudio(), playNote(), setClipper(), setFilter(), setVolume(), startChannel(), stopChannel()

COMMAND REFERENCE

setReverb()

Purpose

Set audio reverberation

Description

Set the amount of reverberation on an audio channel

Syntax

```
setReverb( channel, delay, attenuation )
```

Arguments

channel audio channel from 0 to 15

delay amount of delay

attenuation amount of attenuation

Example

```
delay = 20000
attenuation = 0.1
tune = loadAudio( "David Silvera/music_funky_bar" )
playAudio( 0, tune, 1, 0.5, 1, -1 )

loop
  clear()
  c = controls( 0 )
  delay += c.ly * 8
  attenuation += c.ry / 8
  if attenuation > 1 then
    attenuation = 1
  endIf
  if attenuation < -1 then
    attenuation = -1
  endIf
  setReverb( 0, delay, attenuation )
  printAt( 0, 0, "Push left control stick up or down to control delay" )
  printAt( 0, 1, "Push right control stick up or down to control attenuation" )
  printAt( 0, 3, "Delay: " + delay + " milliseconds" )
  printAt( 0, 4, "Attenuation multiplier: " + attenuation )
  update()
repeat
```

Associated Commands

audioLength(), getChannelStatus(), loadAudio(), note2Freq(), playAudio(), playNote(),
setClipper(), setFilter(), setPan(), setVolume(), startChannel(), stopChannel()

COMMAND REFERENCE

setRumble()

Purpose

Start or stop the controller rumble motors

Description

Activate one of the 4 controller motors at the specified frequency and volume

Syntax

```
setRumble( controller, channel, volume, frequency )
```

Arguments

controller identifier for the controller (0 to 3: 0 is the first)

channel identifier for the motor (0 - top left, 1 - bottom left, 2 - top right, 3 -bottom right)

volume volume (amplitude) of vibration (0 - 1)

frequency frequency of vibration

Example

```
motors = []
frequency = []
for i = 0 to 4 loop
    motors[i] = false
    frequency[i] = i * 100 + 100
repeat
volume = 1

loop
    clear()
    printAt( 0, 0, "Press X to toggle motor 0" )
    printAt( 0, 1, "Press Y to toggle motor 1" )
    printAt( 0, 2, "Press A to toggle motor 2" )
    printAt( 0, 3, "Press B to toggle motor 3" )
    for i = 0 to 4 loop
        if motors[i] then
            setRumble( 0, i, volume, frequency[i] ) // turn on motor
        else
            setRumble( 0, i, 0, 0 ) // turn off motor
        endif
        printAt( 30, i, "motor ", i, ": ", motors[i], " frequency: ", frequency[i] )
    repeat
    c = controls( 0 )
    motor = -1
    if c.x then
        motor = 0
    endif
```

```
if c.y then
  motor = 1
endIf
if c.a then
  motor = 2
endIf
if c.b then
  motor = 3
endIf
if motor > -1 then
  motors[motor] = !motors[motor]
  sleep( 0.2 )
endIf
update()
repeat
```

Associated Commands

[pulseRumble\(\)](#)

COMMAND REFERENCE

setVolume()

Purpose

Change the audio volume

Description

Set the volume level for the specified audio channel

Syntax

```
setVolume( channel, volume )
```

Arguments

channel audio channel from 0 to 15

volume volume from 0 to 1

Example

```
handle = loadAudio( "DavidSilvera/music_evil_presence" )
vol = 0.5
pan = 0.5
speed = 1

playAudio( 0, handle, vol, pan, speed, -1 )
loop
  clear()
  c = controls( 0 )
  printAt( 0, 0, "Use up arrow to increase volume" )
  if c.up then
    vol += 0.01
  endif
  printAt( 0, 1, "Use down arrow to decrease volume" )
  if c.down then
    vol -= 0.01
  endif
  vol = clamp( vol, 0, 1 )
  setVolume( 0, vol )
  printAt( 0, 2, "Volume: ", vol )
  update()
repeat
```

Associated Commands

audioLength(), getChannelStatus(), loadAudio(), note2Freq(), playAudio(), playNote(), setClipper(), setFilter(), setPan(), startChannel(), stopChannel()

COMMAND REFERENCE

startChannel()

Purpose

Start an audio channel

Description

Starts sound being played on the specified audio channel

Syntax

```
startChannel( channel )
```

Arguments

channel audio channel from 0 to 15

Example

```
handle = loadAudio( "DavidSilvera/music_evil_presence" )
volume = 0.5
pan = 0.5
speed = 1

playAudio( 0, handle, volume, pan, speed, -1 )
loop
  clear()
  c = controls( 0 )
  printAt( 0, 0, "Press A to stop music" )
  if c.a then
    stopChannel( 0 )
  endIf
  printAt( 0, 1, "Press B to start music" )
  if c.b then
    startChannel( 0 )
  endIf
  update()
repeat
```

Associated Commands

audioLength(), getChannelStatus(), loadAudio(), note2Freq(), playAudio(), playNote(), setClipper(), setFilter(), setPan(), setVolume(), startChannel(), stopChannel()

COMMAND REFERENCE

stopChannel()

Purpose

Stop an audio channel

Description

Stops sound being played on the specified audio channel

Syntax

```
stopChannel( channel )
```

Arguments

channel audio channel from 0 to 15

Example

```
handle = loadAudio( "DavidSilvera/music_evil_presence" )
volume = 0.5
pan = 0.5
speed = 1
playAudio( 0, handle, volume, pan, speed, -1 )

loop
  clear()
  c = controls( 0 )
  printAt( 0, 0, "Press A to stop music" )
  if c.a then
    stopChannel( 0 )
  endif
  printAt( 0, 1, "Press B to start music" )
  if c.b then
    startChannel( 0 )
  endif
  update()
repeat
```

Associated Commands

audioLength(), getChannelStatus(), loadAudio(), note2Freq(), playAudio(), playNote(), setClipper(), setFilter(), setPan(), setVolume(), startChannel()

COMMAND REFERENCE

Text Handling

COMMAND REFERENCE

chr()

Purpose

Returns the text character of a given unicode value.

Description

The character set used by FUZE⁴ Nintendo Switch is the official unicode standard.

Syntax

```
character = chr( number )
```

Arguments

character A positive number

number The returned text character

Example

```
// display each capital letter of the alphabet
a = 65

loop
  clear()
  print( chr( a ) )
  a += 0.1
  if a >= 91 then
    a = 65
  endif
  update()
repeat
```

Associated Commands

COMMAND REFERENCE

chrVal()

Purpose

Returns the unicode character value of a given single character string.

Description

Takes a single text character and returns the official unicode standard value

Syntax

```
value = chrVal( string )
```

Arguments

value The returned unicode value

number The given text character

Example

```
// display each unicode value for the letters in FUZE
letters = [ "F", "U", "Z", "E" ]

for i = 0 to len( letters ) loop
    print( chrVal( letters[i] ), " " )
repeat

update()
sleep( 3 )
```

Associated Commands

COMMAND REFERENCE

cursor()

Purpose

Set the current text cursor position

Description

Set the current cursor position which will be used by the next print function call

Syntax

```
cursor( x, y )
```

Arguments

x The horizontal cursor position to start printing.

y The vertical cursor position to start printing.

Example

```
hideKeyboard()  
ink( red )  
textsize( 50 )  
message = "Hello World"  
cursor( ( tWidth() - len( message ) ) / 2, tHeight() / 2 )  
print( message )  
update()  
sleep( 3 )
```

Associated Commands

[drawText\(\)](#), [ink\(\)](#), [len\(\)](#), [print\(\)](#), [printAt\(\)](#), [stringHash\(\)](#), [textSize\(\)](#), [tHeight\(\)](#), [tWidth\(\)](#)

COMMAND REFERENCE

drawText()

Purpose

Draw text on the screen

Description

Draw text on the screen at the specified location and size and in the specified colour

Syntax

```
drawText( x, y, size, colour, text )
```

Arguments

x horizontal position to start drawing text in pixels

y vertical position to start drawing text in pixels

size height of text in pixels

colour colour name or RGB values { red, green, blue, opacity } between 0 and 1

text text to draw

Example

```
for size = 1 to 200 step 1 loop
  clear()
  centreText( "Hello World", size )
  update()
repeat

function centreText( message, size )
  textSize( size )
  tw = textWidth( message )
  drawText( ( gWidth() - tw ) / 2, ( gHeight() - size ) / 2, size, white, message )
return void
```



Associated Commands

`cursor()`, `ink()`, `len()`, `print()`, `printAt()`, `stringHash()`, `textSize()`, `tHeight()`, `tWidth()`

COMMAND REFERENCE

ink()

Purpose

Set the ink colour for text printing

Description

Sets the default colour for text printing functions

Syntax

```
ink( colour )
```

Arguments

colour colour name or RGB values { red, green, blue, opacity } between 0 and 1

Example

```
hideKeyboard()  
ink( red )  
textSize( 50 )  
message = "Hello World"  
cursor( ( tWidth() - len( message ) ) / 2, tHeight() / 2 )  
print( message )  
update()  
sleep( 3 )
```

Associated Commands

[cursor\(\)](#), [drawText\(\)](#), [len\(\)](#), [print\(\)](#), [printAt\(\)](#), [stringHash\(\)](#), [textSize\(\)](#), [tHeight\(\)](#), [tWidth\(\)](#)

COMMAND REFERENCE

len()

Purpose

Find the length of a string or array

Description

Returns the number of characters in a text string or the number of items in an array

Syntax

```
stringlen = len( string )  
arraylen = len( array )
```

Arguments

string string to find the length of

array array to find the length of

stringlen number of characters in the specified string

arraylen number of entries in the specified array

Example

```
// Show the length of a string  
s = "Hello World"  
printAt( 0, 0, "Length of ", s, " is ", len( s ) )  
  
// Show the length of an array  
a = []  
a[9] = s  
printAt( 0, 1, "Length of a is ", len( a ) )  
update()  
sleep( 3 )
```

Associated Commands

[cursor\(\)](#), [drawText\(\)](#), [ink\(\)](#), [print\(\)](#), [printAt\(\)](#), [stringHash\(\)](#), [textSize\(\)](#), [tHeight\(\)](#), [tWidth\(\)](#)

COMMAND REFERENCE

print()

Purpose

Print text to the screen

Description

Prints text to the screen at the current cursor position and text size and in the current ink colour

Syntax

```
print( values )
```

Arguments

values A comma separated list of values to print.

Example

```
hideKeyboard()  
ink( red )  
textSize( 50 )  
message = "Hello World"  
cursor( ( tWidth() - len( message ) ) / 2, tHeight() / 2 )  
print( message )  
update()  
sleep( 3 )
```

Associated Commands

[cursor\(\)](#), [drawText\(\)](#), [ink\(\)](#), [len\(\)](#), [printAt\(\)](#), [stringHash\(\)](#), [textSize\(\)](#), [tHeight\(\)](#), [tWidth\(\)](#)

COMMAND REFERENCE

printAt()

Purpose

Print text to the screen at the specified location

Description

Prints text to the screen at the specified position and text size and in the current ink colour

Syntax

```
printAt( x, y, values )
```

Arguments

x The horizontal cursor position to start printing.

y The vertical cursor position to start printing.

values A comma separated list of values to print.

Example

```
hideKeyboard()  
ink( red )  
textSize( 50 )  
message = "Hello World"  
printAt( ( tWidth() - len( message ) ) / 2, tHeight() / 2, message )  
update()  
sleep( 3 )
```

Associated Commands

`cursor()`, `drawText()`, `ink()`, `len()`, `print()`, `stringHash()`, `textSize()`, `tHeight()`, `tWidth()`

COMMAND REFERENCE

str()

Purpose

Convert a value to a string

Description

Convert an integer or floating point number to a string

Syntax

```
result = str( int )
```

```
result = str( float )
```

Arguments

int int value to be converted

float floating point value to be converted

result string result

Example

```
hour = "00"  
minute = "00"  
second = "00"  
size = 100.0  
textSize( size )  
  
loop  
  clear()  
  c = clock()  
  if c.hour < 10 then  
    hour = "0" + str( c.hour )  
  else  
    hour = str( c.hour )  
  endif  
  if c.minute < 10 then  
    minute = "0" + str( c.minute )  
  else  
    minute = str( c.minute )  
  endif  
  if c.second < 10 then  
    second = "0" + str( c.second )  
  else  
    second = str( c.second )  
  endif
```

```
now = hour + ":" + minute + ":" + second
tw = textWidth( now )
drawText( ( gWidth() - tw ) / 2, ( gHeight() - size ) / 2, size, white, now )
update()
repeat
```

Associated Commands

`int()`, `float()`, `fract()`, `round()`, `trunc()`

COMMAND REFERENCE

strBeginsWith()

Purpose

Used to determine whether a string begins with a specific character or series of characters.

Description

Receives two strings of any length, then returns either true or false to indicate whether the supplied string begins with the other.

Syntax

```
result = strBeginsWith( string, to_find )
```

Arguments

result True (1) or false (0) result to indicate whether the string begins with the search target

string The string being checked

to_find The string being searched for

Example

```
string = "FUZE"  
to_find = "F"  
  
loop  
  clear()  
  if strBeginsWith( string, to_find )  
    print( "String '", string, "' begins with '", to_find, "' )  
  else  
    print( "String '", string, "' does not begin with '", to_find, "' )  
  endif  
  update()  
repeat
```

Associated Commands

[str\(\)](#), [strContains\(\)](#), [strEndsWith\(\)](#), [strFind\(\)](#), [strReplace\(\)](#), [stringHash\(\)](#)

COMMAND REFERENCE

strContains()

Purpose

Used to determine whether a string contains a specific character or series of characters.

Description

Receives two strings of any length, then returns either true or false to indicate whether the supplied string contains the other.

Syntax

```
result = strBeginsWith( string, to_find )
```

Arguments

result True (1) or false (0) result to indicate whether the string contains the search target

string The string being checked

to_find To string being searched for

Example

```
string = "FUZE"  
to_find = "Z"  
  
loop  
  clear()  
  if strBeginsWith( string, to_find )  
    print( "String '", string, "' contains '", to_find, "' )  
  else  
    print( "String '", string, "' does not contain '", to_find, "' )  
  endif  
  update()  
repeat
```

Associated Commands

[str\(\)](#), [strBeginsWith\(\)](#), [strEndsWith\(\)](#), [strFind\(\)](#), [strReplace\(\)](#), [stringHash\(\)](#)

COMMAND REFERENCE

strEndsWith()

Purpose

Used to determine whether a string begins with a specific character or series of characters.

Description

Receives two strings of any length, then returns either true or false to indicate whether the supplied string ends with the other.

Syntax

```
result = strEndsWith( string, to_find )
```

Arguments

result True (1) or false (0) result to indicate whether the string ends with the string being searched for

string The string being checked

to_find The string being searched for

Example

```
string = "FUZE"  
to_find = "E"  
  
loop  
  clear()  
  if strEndsWith( string, to_find )  
    print( "String '", string, "' ends with '", to_find, "' )  
  else  
    print( "String '", string, "' does not end with '", to_find, "' )  
  endif  
  update()  
repeat
```

Associated Commands

str(), strBeginsWith(), strContains(), strFind(), strReplace(), stringHash()

COMMAND REFERENCE

strFind()

Purpose

Used to find the location of a string within another.

Description

Receives two strings, one to search and the other to locate. Returns the text character location of the search target within the supplied string.

Syntax

```
result = strFind( string, to_find )
```

Arguments

result Position within string at which the search target appears (begins counting at 0)

string The string being checked

to_find The search target

Example

```
string = "FUZE"  
to_find = "Z"  
  
location = strFind( string, to_find )  
  
print( "Search target '", to_find, "' appears at character ", location, " in string '", string, "' )  
update()  
sleep(5)
```

Associated Commands

[str\(\)](#), [strBeginsWith\(\)](#), [strContains\(\)](#), [strEndsWith\(\)](#), [strReplace\(\)](#), [stringHash\(\)](#)

COMMAND REFERENCE

stringHash()

Purpose

Compute a hash for the specified string

Description

A hash function is one can be used to map data of arbitrary size to data of a fixed size. This can be used for rapid data retrieval, deuplication and protection of sensitive data

Syntax

```
hash = stringHash( string )
```

Arguments

string string to create the hash from

hash hash for the specified string from

Example

```
p = input( "Enter password", false )
if stringHash( p ) == 402054200 then
    print( "Correct" )
else
    print( "Incorrect" )
endif
update()
sleep( 3 )
```

Associated Commands

[str\(\)](#), [strBeginsWith\(\)](#), [strContains\(\)](#), [strEndsWith\(\)](#), [strFind\(\)](#), [stringHash\(\)](#)

COMMAND REFERENCE

strReplace()

Purpose

Used to find a string within another and replace it.

Description

Receives three strings, a string to search, a string to search for, and string to replace the found string with.

Syntax

```
newString = strFind( string, to_find, replace )
```

Arguments

newString Newly created string with the replacement

string The string being checked

to_find The search target

replace The string to replace with

Example

```
string = "FUSE"  
to_find = "S"  
replace = "Z"  
  
newString = strFind( string, to_find, replace )  
  
print( newString )  
update()  
sleep(3)
```

Associated Commands

[str\(\)](#), [strBeginsWith\(\)](#), [strContains\(\)](#), [strEndsWith\(\)](#), [strFind\(\)](#), [stringHash\(\)](#)

COMMAND REFERENCE

textSize()

Purpose

Sets the size of text

Description

Sets the default text size (height) to be used by the print functions

Syntax

```
textSize( size )
```

Arguments

size The size (height) of text to be used by print and printat functions

Example

```
message = "Hello World"
for size = 1 to 200 step 1 loop
  clear()
  textSize( size )
  tw = textWidth( message )
  drawText( ( gWidth() - tw ) / 2, ( gHeight() - size ) / 2, size, white, message )
  update()
repeat
```



Associated Commands

[cursor\(\)](#), [drawText\(\)](#), [ink\(\)](#), [len\(\)](#), [print\(\)](#), [printAt\(\)](#), [stringHash\(\)](#), [tHeight\(\)](#), [tWidth\(\)](#)

COMMAND REFERENCE

textWidth()

Purpose

Find the width in pixels of a string

Description

Receives a single string argument, returns the width (in pixels) of the given string

Syntax

```
result = textWidth( text )
```

Arguments

text text to find the size of

result width of the text at the current textSize

Example

```
message = "Hello World"  
for size = 1 to 200 step 1 loop  
  clear()  
  textSize( size )  
  tw = textWidth( message )  
  drawText( ( gWidth() - tw ) / 2, ( gHeight() - size ) / 2, size, white, message )  
  update()  
repeat
```



Associated Commands

[cursor\(\)](#), [drawText\(\)](#), [ink\(\)](#), [len\(\)](#), [print\(\)](#), [printAt\(\)](#), [stringHash\(\)](#), [textSize\(\)](#), [tHeight\(\)](#), [tWidth\(\)](#)

COMMAND REFERENCE

tHeight()

Purpose

Get the text height of the display

Description

Find the number of characters that will fit on the screen vertically at the current text size

Syntax

```
textheight = tHeight ( )
```

Arguments

textheight The number of characters that will fit on the screen vertically at the current fontsize

Example

```
clear()
textSize( 100 )
for y = 0 to tHeight() loop
  for x = 0 to tWidth() loop
    printAt( x, y, y + 1 )
    update()
  repeat
repeat

for i = 1 to 100 loop
  update()
repeat
```



Associated Commands

`cursor()`, `drawText()`, `ink()`, `len()`, `print()`, `printAt()`, `stringHash()`, `textSize()`, `tWidth()`

COMMAND REFERENCE

tWidth()

Purpose

Get the text width of the display

Description

Find the number of characters that will fit on the screen horizontally at the current text size

Syntax

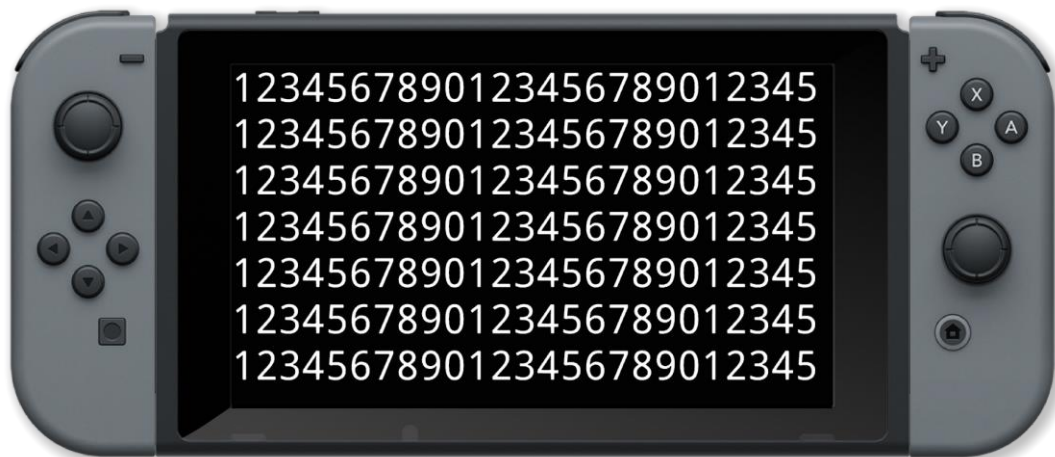
```
textwidth = tWidth( )
```

Arguments

textwidth The number of characters that will fit on the screen horizontally at the current fontsize

Example

```
textSize( 100 )  
for y = 0 to tHeight() loop  
  for x = 0 to tWidth() loop  
    printAt( x, y, ( x + 1 ) % 10 )  
    update()  
  repeat  
repeat  
for i = 1 to 100 loop  
  update()  
repeat
```



Associated Commands

[cursor\(\)](#), [drawText\(\)](#), [ink\(\)](#), [len\(\)](#), [print\(\)](#), [printAt\(\)](#), [stringHash\(\)](#), [textSize\(\)](#), [tHeight\(\)](#)

COMMAND REFERENCE

Time and Date

COMMAND REFERENCE

clock()

Purpose

Get the current date and time

Description

Returns a structure containing the current system date and time

Syntax

```
c = clock( )
```

Arguments

c.year current year

c.month current month of the year (1-12)

c.day current day of the month (1-31)

c.hour current hour (1-24)

c.minute current minute (0-59)

c.second current second (0-59)

Example

```
hour = "00"  
minute = "00"  
second = "00"  
size = 100.0  
textSize( size )  
loop  
  clear()  
  c = clock()  
  if c.hour < 10 then  
    hour = "0" + str( c.hour )  
  else  
    hour = str( c.hour )  
  endif  
  if c.minute < 10 then  
    minute = "0" + str( c.minute )  
  else  
    minute = str( c.minute )  
  endif  
  if c.second < 10 then  
    second = "0" + str( c.second )  
  else
```

```
        second = str( c.second )
    endif
    now = hour + ":" + minute + ":" + second
    tw = textWidth( now )
    drawText( ( gwidth() - tw ) / 2, ( gheight() - size ) / 2, size, white, now )
    update()
repeat
```

Associated Commands

[setTimer\(\)](#), [sleep\(\)](#), [startTimer\(\)](#), [stopTimer\(\)](#), [time\(\)](#)

COMMAND REFERENCE

setTimer()

Purpose

Create a new timer

Description

Create a new timer which is used to call a function a number of times with a fixed interval

Syntax

```
handle = setTimer( interval, count, functionName( arguments ) )
```

Arguments

handle The handle of the timer

interval The interval between each function call

count The number of times to call the function

functionName The function to call

arguments The arguments to pass to the function

Example

```
count = 10
handle = setTimer( 1, 11, showCount() )
stopTimer( handle )

loop
  c = controls( 0 )
  printAt( 0, 0, "Press A to start timer" )
  if c.a then
    startTimer( handle )
  endIf
  printAt( 0, 1, "Press B to stop timer" )
  if c.b then
    stopTimer( handle )
  endIf
  update()
repeat

function showCount()
  clear()
  textSize( 500 )
  printAt( tWidth() / 2, tHeight() / 2, count )
  textSize( 29 )
```

```
count -= 1  
return void
```

Associated Commands

clock(), sleep(), startTimer(), stopTimer(), time()

COMMAND REFERENCE

sleep()

Purpose

Go to sleep

Description

Do nothing (and become unresponsive) for the specified period

Syntax

```
sleep( time )
```

Arguments

time Time to sleep for in seconds

Example

```
print( "Wait for 3 seconds" )  
update()  
sleep( 3 )
```

Associated Commands

clock(), setTimeout(), setInterval(), clearTimeout(), clearInterval()

COMMAND REFERENCE

startTimer()

Purpose

Start a timer

Description

Start a timer that has been stopped with stopTimer

Syntax

```
startTimer( handle )
```

Arguments

handle handle of the timer from settimer

Example

```
count = 10
handle = setTimer( 1, 11, showCount() )
stoptimer( handle )

loop
  c = controls( 0 )
  printAt( 0, 0, "Press A to start timer" )
  if c.a then
    startTimer( handle )
  endif
  printAt( 0, 1, "Press B to stop timer" )
  if c.b then
    stopTimer( handle )
  endif
  update()
repeat

function showCount()
  clear()
  textSize( 500 )
  printAt( tWidth() / 2, tHeight() / 2, count )
  textSize( 29 )
  count -= 1
return void
```

Associated Commands

clock(), setTimer(), sleep(), stopTimer(), time()

COMMAND REFERENCE

stopTimer()

Purpose

Stop a timer

Description

Stop a timer that has been started with `setTimer` or `startTimer`

Syntax

```
stopTimer( handle )
```

Arguments

handle handle of the timer from `setTimer`

Example

```
count = 10
handle = setTimer( 1, 11, showCount() )
stopTimer( handle )
loop
    c = controls( 0 )
    printAt( 0, 0, "Press A to start timer" )
    if c.a then
        startTimer( handle )
    endif
    printAt( 0, 1, "Press B to stop timer" )
    if c.b then
        stopTimer( handle )
    endif
    update()
repeat

function showCount()
    clear()
    textSize( 500 )
    printAt( tWidth() / 2, tHeight() / 2, count )
    textSize( 29 )
    count -= 1
return void
```

Associated Commands

`clock()`, `setTimer()`, `sleep()`, `startTimer()`, `time()`

COMMAND REFERENCE

time()

Purpose

Get the current system tick count

Description

Returns the current system tick value in seconds

Syntax

```
result = time( )
```

Arguments

result current system tick value in seconds

Example

```
loop
  clear()
  printAt( 0, 0, time() )
  wait( 3 )
  printAt( 0, 1, time() )
  wait( 3 )
  update()
repeat

// wait for interval seconds
function wait( interval )
  now = time()
  end = now + interval
  while end > now loop
    update()
    now = time()
  repeat
return void
```

Associated Commands

`clock()`, `setTimer()`, `sleep()`, `startTimer()`, `stopTimer()`

ABOUT

Intro

Millions of years ago our world was invaded by a species of bodiless creatures called the Geeks. Millenia upon millennia passed. They waited. They waited some more. They grew bored. Eventually Humans evolved. The Geeks watched. At first, they were entertained by the Human's incessant ability to invent utterly useless things they would place a great and much misguided importance upon. They watched some more. They grew bored again.

Eventually the Geeks became able to influence the thoughts and actions of the Humans. Among many other fantastic contributions to Human nature, one single phrase, a single phrase above all others, was to entrench itself deep into Human consciousness. It was: "The Geek shall inherit the Earth".

However, over the next couple of thousand years this somehow mutated into "The **meek** shall inherit the Earth".

The Geeks were not pleased by this misrepresentation and went to work on correcting the problem. Over the next few hundred years and up until the mid-20th century the Geeks perfected a method of taking over the hearts and minds of thousands of Humans. Instead of inventing new types of chair, handbag or another fancy cheese, they put their minds to developing powerful computational technologies that would pave the way for a new dawn of invention and entertainment.

The first wave of these new systems were far too complex for ordinary, unoccupied Humans to comprehend but more were soon to follow. Then, in the 1970s the Home Computer arrived and, finally, computing for the masses that even the lowliest of Human intelligence could grasp.

Alan Turing, Bill Gates, Steve Wozniak, Steve Jobs, Grace Murray Hopper, Nolan Bushnel, Jack Tramiel, Sir Clive Sinclair, Seymour Crey, Ada Lovelace, Charles Babbage, Gottfried Leibniz, Hermann Hauser and Chris Curry are but just a few of the most famous and recognised Geeks. There are many, many more and by far too many to mention but one thing is certain. It was not long ago that they got their way and fulfilled their dream. Take a look around you... The Geeks certainly did inherit the earth!

FUZE Technologies Ltd knows a good thing when it sees it and it wants a piece of the action. FUZE⁴ Nintendo Switch is our offering to budding Geeks everywhere. We hope you like it.

Thank You

FUZE⁴ Nintendo Switch has been developed in the UK by a very small team. It is also a project that has run alongside the usual FUZE activities including delivering hundreds of FUZE Coding workshops to young people across the UK. When we originally announced we were embarking on the project a web page accepting donations was introduced on the FUZE website.

We can't shout this loud enough, but a HUGE thank you from our team to all our donators. It meant the world to us to receive your support and we would not have made it without you!

Additional thanks for their ongoing support, encouragement and general awesomeness goes to:

Charlie! Shila Odedra-Silvera, Leonard Teague, Helen & Anna Silvera, Emma, Mark and Josh Mulcahy, Nic, Caroline and Josh, Philip Prall, Jared Forrester, James Hands, Lucie Dickinson, Luke Schofield, Christian Hegyi, Martin White, Anil & Milan Modhwadia, Andrew and Lorna Johns, Ellis Durden, James Silcox of Reach the Core, John Ronane, Dawn, Chris, Charlotte and Elizabeth Basnett, Rod and Iris Ellis, Shane, Leanne, Luke and Mia Ellis, The Rgt Honourable John Bercow, Ickford Combined School, Wheatley Park School, The Hall School, John Hampden School, Alex and Vicki and Adrian Mietusiewicz

The FUZE team

Jon Silvera Paragon of Adjudication & Execution, Founder & CEO / Project Manager & Investor

Jon Clough Paragon of Integrity, Finance Director & Investor

Derek Taylor Investor

Colin Bodley Paragon of Stoicism, Programmer's Reference Guide, Help contributor, product tester and Investor

Luke Mulcahy Paragon of Wisdom, Lead Programmer including 3D, 2D & Audio engines, Editor and much, much more

David Silvera Paragon of Truth, Designer, Programmer, Tutorials & help content & Audio Assets / technical consultant and Head Tutor

Will Tice Programmer including User Interface, Editor support, 2D Sprites, Map and image tools and much more

Kat Deak Paragon of Artistry, 3D Graphics artist & contributor, 3D model quality control & product testing

Mike Green Additional programming support, web development and product quality control

Molly Odedra-Silvera Paragon of Oration, Marketing support, Product tester, quality control and Assistant Tutor

Lizzie Botelle Paragon of Orthography, Product tester & quality control

Ben Taylor Product tester, quality control and Assistant Tutor

Grace Odedra-Silvera, Charlotte Reeve, Hannah & Mica Assistant Tutors

Contact

FUZE Technologies Ltd 15 Clearfields Farm Wotton Underwood Aylesbury, HP18 0RE United Kingdom

email: contact@fuze.co.uk phone: 44 (0)1844 239 432 social: @fuzecoding web: fuze.co.uk

License agreements

Contributing Artist Agreements (Jessica Clayton / Alienoutcast) - (Luis Zuno / ANSIMUZ) - (Hasan Bayat / Bayat) - (Michael Lohr / Broken Vector) - (Colin Brown) - (David Silvera / FUZE Technologies Ltd) - (Dominik Gabriel / Cryptogene) - (Ajay Karat / Devil's Garage) - (Valerij Golovin / DinV Studio) - (Eder Avila Muniz / EderMuniz) - (Emerald Eel Entertainment) - (Keith Fox / Fertile Soil Productions) - (Jason Perry / finalbossblues) - (Gijs De Mik) - (Kat Deak / FUZE Technologies Ltd) - (Kenney Vleugels / Pixeland Parkstad / Kenney) - (Graeme Houston / Kuro Ren) - (Hamza Cavus / MoonStar) - (Pipo / Hiroshi Suzuki) - (Jei Oakley / Pixelsnplay) - (Tomás Laulhé / Quaternius) - (Krzysztof Dycha / Ravenmore) - (Joshua Von Ros / RVROS) - (Lavinia / Selavi Games) - (Felipe Díaz Flores / Sinestesia) - (Stijn Van Wakeren) - (William S. Tice / Untied Games) - (Vadi Godfried / Vadi-Va)

All game assets are provided under the following agreement:

User License Agreement for game assets included with FUZE®4 Nintendo Switch

Definitions:

FUZE FUZE® is a coding environment designed to make it as easy as possible to learn to code and program games. It is available on multiple hardware platforms.

ASSETS Digitally represented 2D artwork, tile maps, sprite sheets, 3D models, 3D animations and materials, screen fonts, audio clips, music and any other game related content supplied with the intention of being used to create games or applications in the **FUZE** coding environment.

ARTIST [LICENSOR] LICENSOR Original author responsible for the creation and or supply of **ASSETS**.

COMPANY FUZE Technologies Ltd. The developer and worldwide exclusive publisher of **FUZE**.

USER Recipient, having acquired the use of the **FUZE** coding environment on any platform.

Therefore: It is understood that **LICENSOR** has granted **COMPANY** the non-exclusive right to bundle **ASSETS** with **FUZE** on its associated platforms. While **ASSETS** may be available from other sources the license granted to **COMPANY** is specific to **COMPANY** and should not be misconstrued with any other license provided by **LICENSOR** to any other party.

Specifically: **COMPANY** grants the **USER** the right to include, edit and or manipulate **ASSETS** for use within their own **FUZE** projects. The **USER** must clearly display attribution to the **LICENSOR** and **FUZE** within the project and, in the case of commercial versions, on any marketing materials promoting the project.

USER may not, without explicit written approval from **LICENSOR**, redistribute **ASSETS** for free or commercially outside the scope of being included within **USER** projects. **USER** projects may not invite the extraction of **ASSETS**.

USER may not sublicense or redistribute **ASSETS** beyond the scope of this agreement or in any standalone format to any third party.

For the avoidance of doubt **USER** may include **ASSETS** in their own **FUZE** projects but not redistribute in any way outside the scope of this agreement.

There is no date period to this agreement. It remains in force unless released by **COMPANY**.

If in doubt or if you would like to discuss this license please email contact@fuze.co.uk

© FUZE Technologies Ltd. Company Registration Number 08837428. 15 Clearfields Farm, Wotton Underwood, Buckinghamshire, HP18 0RS, England

FUZE End User LICENSE Agreement 1) Definitions The following definitions apply to this agreement:

- a. “**FUZE**” being the company, FUZE Technologies Ltd as registered in the United Kingdom, registration number: 8837428
- b. “**SOFTWARE**” refers to the product **LICENSE** under this agreement, the application in object code and or binary formats including **UPDATEs** and **ASSETS**.
- c. “**ASSETS**” refers to the ‘video game’ content included in a **FUZE PRODUCT** in addition to the **SOFTWARE**, such as 3D model files and object files, audio files, video files and image files, as well as other sounds and templates that contain such files.
- d. “**DEVICE**” refers to any electronic physical or virtual computing device (e.g. PC, laptop, workstation, video game console, tablet, mobile phone, an instance of a virtual machine, etc.).
- e. “**UPDATE**” refers to an updated version of **SOFTWARE**. An **UPDATE** constitutes a modified, improved or fixed version of **SOFTWARE**. It does not include new versions of a **SOFTWARE**.
- f. “**LICENSE**” refers to the **LICENSE** assigned to a specific **DEVICE** belonging to the customer following installation and if required activation of the **SOFTWARE**.
- g. “**CONTRIBUTING ARTIST**” refers to the artist responsible for creating **ASSETS** included with the **SOFTWARE**.
- h. “**OFFICIAL CHANNELS**” refers to the sales distribution network of approved resellers appointed by **FUZE**.
- i. “**COMMERCIAL USE**” means use of the **SOFTWARE** or **ASSETS** for the direct or indirect purpose of financial benefit (e.g. by means of sale, licensing, advertising, etc.).

2) Purpose a. Subject to the conditions within and for the duration of this agreement, **FUZE** grants you (the user) the non-exclusive and non-transferable right to use the respective **SOFTWARE** on one **DEVICE**. **FUZE** retains ownership, copyright and other proprietary rights related to the **SOFTWARE**. You (the customer) acknowledge **FUZE**’s ownership as well as all proprietary rights to the **SOFTWARE**, **ASSETS**, backup copies and documentation. The buyer of the **SOFTWARE** is solely responsible for the proper contractual use of the **SOFTWARE**. b. Only users who have purchased the **SOFTWARE** via **OFFICIAL CHANNELS** are authorized to receive **UPDATEs**.

3) Installation and Registration a. Depending on the **DEVICE** you are installing **SOFTWARE** on- to you may receive a unique **LICENSE** number to enter during installation. An online registration

may also be required before **SOFTWARE** is activated. If the number of user installations exceeds the number of allowed installations specified in the **LICENSE** is exceeded, the **LICENSE** may be deactivated by **FUZE**. In such cases the user should contact **FUZE** to request reactivation.

4) LICENSE verification a. **SOFTWARE** generally requires an internet connection to install and activate.

5) Using the SOFTWARE and ASSETS for commercial purposes a. Where technically possible the **SOFTWARE** may be used for commercial purposes subject to the terms in clause 6.

6) ASSETS a. Where technically possible **ASSETS** may be used for commercial purposes subject to the terms in clause 6 and as specified in the **CONTRIBUTING ARTIST** Agreements. This applies to games, demos, applications, example programs and or modified versions of the **ASSETS** or any project including any **ASSET**. b. **FUZE** grants the user the right to include, edit and or manipulate **ASSETS** for use within their own **FUZE** projects. The user must clearly display attribution to the **CONTRIBUTING ARTIST** and **FUZE** within the project and on any materials promoting the project. c. Exploitation of **ASSETS** outside the scope of personally created work, i.e. outside of **SOFTWARE**, is prohibited. For the avoidance of doubt; **ASSETS**, Programs, Projects, Manuals, Demonstration and examples may not be extracted and used separately for commercial or non-commercial purposes. d. User may not, without explicit written approval from **LICENSOR**, redistribute **ASSETS** for free or commercially, outside the scope of being included within a user's project. User projects may not invite the extraction of **ASSETS**. e. Users may not sublicense or redistribute **ASSETS** beyond the scope of this agreement or in any standalone format to any third party.

7) Copying, renting and redistribution a. You are prohibited from copying the licensed program and the written documentation either partially or in its entirety. This excludes your right to make a digital copy of the software for backup purposes. Back-up copies may not be redistributed. b. The **SOFTWARE** as well as the written documentation may not be commercially rented out or commercially lent in any other form to a third party in exchange for payment. This also applies to lending of the **SOFTWARE** in a pre-installed form on a **DEVICE** that is commercially offered to third parties in exchange for payment. c. You may not make any changes to the **SOFTWARE**, personally or by third parties. You may not disassemble the **SOFTWARE** into its components, nor modify the object code, decode, copy or use it in any way other than that foreseen in the contract.

8) Transfer of rights a. The transfer of rights and obligations under this agreement to third parties is only permitted on authorisation from **FUZE** with the exception of personal transfer of the legally acquired **SOFTWARE** by the rightful owner. In case of the ownership of the rightfully acquired **SOFTWARE** being transferred in this fashion, the original owner is obliged to destroy all back-up copies and to delete the installation. A digital transfer of a **SOFTWARE** (a download) is prohibited.

9) Guarantee and liability a. You are aware that software programs, **ASSETS** and associated documentation may contain errors, and that it is not possible to develop data processing programs in such a way that they are error-free for all usage scenarios and all customer requirements, or error-free in conjunction with all third-party programs and hardware. **FUZE** provides no assurances of particular features and usability related to planned customer-specific applications. b. In case of paid products and services, **FUZE** is only liable to slight negligible damages incurred by it or its assistant(s) if a duty is violated, even if it is extra-contractual, the adherence to which is of special importance in order to be in compliance with contractual use

(Cardinal duty), as well in cases of damage to life, body and health. c. For non-observance of a cardinal obligation, the liability is limited to the damage which must be typically expected within the scope of this agreement if there is no intention or gross negligence or if **FUZE** must incur liability because of fatal injury, physical injury or health hazards. d. **FUZE** shall not be liable for damage which can be controlled by the other contracting party or which the other contracting party could have prevented by taking measures which can be reasonably expected. **FUZE** is not liable for data loss. e. In any event, **FUZE**'s liability is limited to four times the amount paid for the **LICENSE** fee by the customer. This exclusion does not apply to damage caused through intent or gross negligence on the part of **FUZE**. f. In case of paid products and services, the guarantee against deficiency in material and defects in title is limited to fraudulent concealment of defects by **FUZE** in consideration of free licensing of the product. g. Statutory liability in case of personal damages and damages pursuant to the Product Liability Act remains unaffected. h. A change in the burden of proof to the disadvantage of the customer is not related to the foregoing provision. i. Insofar as **SOFTWARE** contains functions that operate via an online server, **FUZE** retains the right to end the offering at any time. Availability will not be guaranteed.

10) Licence conditions of other manufacturers a. If the **SOFTWARE** contains additional software from other manufacturers, or should additional software be integrated, then compliance with the use and license conditions of the manufacturer of said delivered additional software is also compulsory. If **SOFTWARE** contains additional software, you can view the respective use and licensing terms in the corresponding file.

11) Support a. **FUZE** offers electronic Internet support during the warranty period. This encompasses clarification of installation questions and installation problems by Internet or email. The rendering of support is at the sole discretion of **FUZE** and is not connected with any guarantee or warranty.

12) Other a. This agreement constitutes the entire agreement of the parties regarding the contract purpose. Collateral agreements shall not exist. No verbal or written statements made by **FUZE** or any **FUZE** employee can alter or question the validity of this **LICENSE** agreement.

13) Validity of contractual conditions a. Should one or more of the conditions in this contract be or become invalid, this will not affect the validity of the remaining contract. A substitute provision will replace the invalid condition, such as comes closest to the intended purpose. The contract is subject to the laws of the United Kingdom.

QUICK REFERENCE

2D Graphics

`box` (x, y, width, height, colour, outline)

`centreSpriteCamera` (pos)

`centreSpriteCamera` (xpos, ypos)

`circle` (x, y, radius, vertices, colour, outline)

`collideMap` (sprite)

`result = collideSprites` (spriteA, spriteB, resolve1, resolve2)

`handle = copyImage` (imageHandle, source)

`copyShape` (shape)

`createBox` (x, y, width, height)

`createCircle` (x, y, radius, vertices)

`createCurve` (point1, point2, ... pointN)

`createCurve` (points)

`handle = createImage` (width, height, filter, type)

`createLine` (x1, y1, x2, y2)

`createLineStrip` (point1, point2, ... pointN)

`createLineStrip` (points)

`createPoly` (point1, point2, ... pointN)

`createPoly` (points)

`handle = createSprite` ()

`createStar` (x, y, innerRadius, outerRadius, numPoints)

`createTriangle` (x1, y1, x2, y2, x3, y3)

`deleteShape` (shape)

`result = deltaTime` ()

`result = detectMapCollision` (sprite)

`result = detectSpriteCollision` (spriteA, spriteB)

`drawImage` (handle, x, y)

`drawImage` (handle, x, y, scale)

`drawImage (handle, { sourceX, sourceY, sourceW, sourceH , { x, y, width, height } })`
`drawImageEx (handle, location, rotation, scale, tint, origin)`
`drawMap ()`
`drawMapLayer (layer)`
`drawQuad (handle, { sourcex, sourcey, sourcew, sourceh , points, tint })`
`drawShape (shape)`
`drawSheet (handle, tileno, { xpos, ypos, width, height })`
`drawSprite (sprite)`
`drawSprites ()`
`freeImage (handle)`
`getShapeBounds (shape)`
`getShapeLocation (shape)`
`getShapeRotation (shape)`
`getShapeScale (shape)`
`getShapeTint (shape)`
`result = getSpriteAnimFrame (sprite)`
`result = getSpriteAnimFrameCount (sprite)`
`vectorResult = getSpriteAnimSpeed (sprite)`
`vectorResult = getSpriteCamera ()`
`result = getSpriteCameraRotation ()`
`vectorResult = getSpriteColour (sprite)`
`vectorResult = getSpriteColourSpeed (sprite)`
`result = getSpriteDepth (sprite)`
`handle = getSpriteImage (sprite)`
`vectorResult = getSpriteImageSize (sprite)`
`vectorResult = getSpriteLocation (sprite)`
`vectorResult = getSpriteOrigin (sprite)`
`result = getSpriteRotation (sprite)`
`result = getSpriteRotationSpeed (sprite)`

`vectorResult = getSpriteScale (sprite)`
`vectorResult = getSpriteScaleSpeed (sprite)`
`vectorResult = getSpriteSize (sprite)`
`vectorResult = getSpriteSpeed (sprite)`
`result = getSpriteVisibility (sprite)`
`getVertex (shape, vertex)`
`getVertexColour (shape, vertex)`
`getVertexLineColour (shape, vertex)`
`getVertexLineThickness (shape, vertex)`
`result = imageH (image)`
`vectorResult = imageSize (sprite)`
`result = imageW (image)`
`joinShapes (shape1, shape2)`
`line (point1, point2, colour)`
`handle = loadImage (filename)`
`handle = loadImage (filename, filter)`
`loadMap (filename)`
`moveShape (shape, x, y)`
`moveShape (shape, axes)`
`result = numTiles (tilesheet)`
`numVerts (shape)`
`plot (x, y, colour)`
`removeSprite (sprite)`
`renderEffect (image, target, effect, arguments)`
`rotateShape (shape, amount)`
`scaleShape (shape, scale)`
`scaleShape (shape, dirX, dirY)`
`setBlend (mode)`
`setShapeColour (shape, colour)`

setShapeLineStyle (shape, thickness, tint)
setShapeLocation (shape, x, y)
setShapeLocation (shape, location)
setShapeRotation (shape, amount)
setShapeScale (shape, scale)
setShapeScale (shape, scaleX, scaleY)
setShapeScaleModeLocal (shape, enabled)
setShapeTint (shape, tint)
setSpriteAnimation (sprite, startTile, endTile)
setSpriteAnimation (sprite, startTile, endTile, speed)
setSpriteAnimFrame (sprite, frame)
setSpriteAnimSpeed (sprite, speed)
setSpriteCamera (pos)
setSpriteCamera (xpos, ypos)
setSpriteCamera (xpos, ypos, zpos)
setSpriteCameraRotation (angle)
setSpriteCollisionShape (sprite, shape)
setSpriteCollisionShape (sprite, shape, width, height, rotation)
setSpriteColour (sprite, colour)
setSpriteColour (sprite, red, green, blue, alpha)
setSpriteColourSpeed (sprite, rgbaSpeed)
setSpriteColourSpeed (sprite, rSpeed, gSpeed, bSpeed, aSpeed)
setSpriteDepth (sprite, depth)
setSpriteImage (sprite, image)
setSpriteLocation (sprite, pos)
setSpriteLocation (sprite, xpos, ypos)
setSpriteOrigin (sprite, pos)
setSpriteOrigin (sprite, xpos, ypos)
setSpriteRotation (sprite, angle)

`setSpriteRotationSpeed` (sprite, rotationSpeed)
`setSpriteScale` (sprite, scale)
`setSpriteScale` (sprite, { xScale, yScale })
`setSpriteScaleSpeed` (sprite, scaleSpeed)
`setSpriteScaleSpeed` (sprite, xScaleSpeed, yScaleSpeed)
`setSpriteSpeed` (sprite, speed)
`setSpriteSpeed` (sprite, xspeed, yspeed)
`setSpriteText` (sprite, fontsize, tint, arguments)
`setSpriteVisibility` (sprite, visibility)
`setVertex` (shape, vertex, position)
`setVertexColour` (shape, vertex, colour)
`setVertexLineStyle` (shape, vertex, thickness, tint)
`setView` (left, top, right, bottom)
vectorResult = `tileSize` (image, tile)
`triangle` (point1, point2, point3, colour, outline)
`updateSprite` (sprite)
`updateSprite` (sprite, deltatime)
`updateSprites` ()
`updateSprites` (sprites)
`updateSprites` (deltatime)
`updateSprites` (sprites, deltatime)
`uploadImage` (pixeldata, width, height, filter)

3D Graphics

result = `animationLength` (object, animation)
handle = `createTerrain` (gridsize, filter)
`drawObjects` ()
handle = `loadModel` (filename)
result = `numAnimations` (object)
`objectPointAt` (handle, point)

`handle = placeObject (object, location, scale)`
`handle = pointLight (position, colour, brightness)`
`handle = pointShadowLight (position, colour, brightness)`
`removeLight (light)`
`removeObject (handle)`
`rotateObject (handle, axes, amount)`
`setAmbientLight (colour)`
`setCamera (location, target)`
`setFov (angle)`
`setLightBrightness (light, brightness)`
`setLightColour (light, colour)`
`setLightDir (light, direction)`
`setLightPos (light, position)`
`setLightSpread (light, spread)`
`setObjectMaterial (handle, colour, metallic, roughness)`
`setObjectPos (handle, pos)`
`setObjectScale (handle, scale)`
`setTerrainPoint (terrain, xpos, ypos, height, colour)`
`handle = spotLight (position, direction, colour, brightness, spread)`
`updateAnimation (object, animation, frame)`
`updateTerrain (terrain, heights, colours)`
`handle = worldLight (direction, colour, brightness)`
`worldShadowLight (centre, direction, colour, brightness, range, resolution)`

Arithmetic

`result = abs (number)`
`result = acos (cosine)`
`result = asin (sine)`
`result = atan (tangent)`
`result = atan2 (x, y)`

vectorResult = bezier (point1, point2, point3, factor)
vectorResult = bezier (point1, point2, point3, point4, factor)
result = bitCount (number)
ceil (number)
result = clamp (number, minimum, maximum)
result = cos (angle)
vectorResult = cross (vector1, vector2)
result = distance (point1, point2)
result = dot (vector1, vector2)
result = float (value)
result = floor (number)
result = fract (value)
result = int (value)
result = length (**vector**)
result = lerp (v0, v1, t)
result = max (number1, number2)
result = min (number1, number2)
vectorResult = normalize (**vector**)
result = pow (number, power)
radians (enable)
result = random (range)
vectorResult = reflect (incident, normal)
vectorResult = refract (incident, normal, ior)
rnd (range)
result = round (value)
result = rsqrt (number)
result = sin (angle)
result = sinCos (angle)
result = smoothStep (value0, value1, factor)

result = `sqrt` (number)

result = `tan` (angle)

result = `trunc` (value)

Binary

result = `bitFieldExtract` (number, start, count)

result = `bitFieldInsert` (number, start, count, value)

result = `bitGet` (number, bit)

result = `bitSet` (number, bit, value)

result = `leadingZeroes` (value)

result = `trailingZeroes` (value)

File Handling

`close` (handle)

`open` ()

`read` (handle, count)

`seek` (handle, position)

`write` (handle, text)

Input

structure = `controls` (index)

result = `docked` ()

result = `getKeyboardBuffer` ()

`hideKeyboard` ()

result = `input` (prompt, multiline)

`showKeyboard` ()

list = `touch` ()

Screen Display

`clear` ()

`clear` (colour)

result = `gHeight` ()

`result = gWidth ()`
`setDrawTarget (target)`
`setMode (width, height)`
`update ()`

Sound and Music

`result = audioLength (handle)`
`status = getChannelStatus (channel)`
`handle = loadAudio (sample)`
`result = note2Freq (note)`
`playAudio (channel, handle, volume, pan, speed, loops)`
`playNote (channel, wave, frequency, volume, speed, pan)`
`pulseRumble (controller, channel, speed, volume, frequency)`
`setClipper (channel, threshold, strength)`
`setEnvelope (channel, speed)`
`setFilter (channel, type, cutoff)`
`setFrequency (channel, frequency)`
`setModulator (channel, wave, frequency, scale)`
`setPan (channel, pan)`
`setReverb (channel, delay, attenuation)`
`setRumble (controller, channel, volume, frequency)`
`setVolume (channel, volume)`
`startChannel (channel)`
`stopChannel (channel)`

Text Handling

`chr (number)`
`chrVal (string)`
`cursor (x, y)`
`drawText (x, y, size, colour, text)`
`ink (colour)`

```
result = len ( string )
result = len ( array )
print ( values )
printAt ( x, y, values )
str ( int )
str ( float )
result = strBeginsWith ( string, to_find )
result = strContains ( string, to_find )
strEndsWith ( string, to_find )
strFind ( string, to_find )
result = stringHash ( string )
strReplace ( string, to_find, replace )
textSize ( size )
result = textWidth ( text )
result = tHeight ()
result = tWidth ()
```

Time and Date

```
structure = clock ()
handle = setTimer ( interval, count, functionName( arguments ) )
sleep ( time )
startTimer ( handle )
stopTimer ( handle )
result = time ()
```

TUTORIALS

TUTORIALS

Tutorial 1: Loops

Every programmer must start somewhere. In this tutorial, we'll be writing a version of perhaps the most famous program in existence: Hello World.

Before we jump right in, we need a couple of things. We'll need a `print()` function to instruct **FUZE⁴ Nintendo Switch** what to print. We'll also need an `update()` function to update the screen with the text we want and finally, we'll need a `sleep()` function at the end so our program stays on screen!

Type (or copy and paste) the code below into the **FUZE** code editor. Run the program with **+**, or the **F5** key if you're using a USB keyboard.

```
1. print( "Hello Nintendo" )
2. update()
3. sleep( 2 )
```

We should see our chosen text appear magnificently on screen for 2 seconds. Now let's make **FUZE** do this over and over again, using one of the most important concepts in programming: a **loop**. For this, we can remove the `sleep()` function because our program will run continuously. We will still need the `update()` though, because we want our text to appear on the screen!

```
1. loop
2.     print( "Hello Nintendo" )
3.     update()
4. repeat
```

Think of a **loop** like a sandwich. We have a top piece of bread (**loop**), some filling in the middle and a bottom piece of bread (**repeat**)! Without one of the pieces of bread, our sandwich is no longer a sandwich!

What I'm saying is...

Without **loop**, **repeat** will do nothing and vice-versa!

There is an important point when it comes to **loops**. Take a look below:

```
1. loop
2.     print( "Hello Nintendo" )
3.     update()
4. repeat
5.     print( "What about me?" )
```

See the new `print()` function on line 5? This line will never appear on screen. Why?

When **FUZE** gets to the **repeat** keyword on line 4, it returns to the last **loop** keyword. So, our program will read:

Line 1, line 2, line 3, Line 4... Line 1, line 2, line 3, Line 4... Line 1, line 2, line 3, Line 4...

FOREVER! Well... Not quite. It will run until we tell it stop. Press **+** to stop the program and return to the editor (use the **F5** key if using a USB keyboard).

Line 5 is not **in** the **loop**. To make this line happen too, our program must look like this:

```
1. loop
2.   print( "Hello Nintendo" )
3.   print( "What about me?" )
4.   update()
5. repeat
```

Now the new `print()` line is included **inside** the **loop** and is very happy indeed.

Okay. Let's get some colour going in our program. We use the `ink()` **function** to do this.

```
1. ink( white )
2. loop
3.   print( "Hello Nintendo" )
4.   print( "What about me?" )
5.   update()
6. repeat
```

We've used the *name* for the colours we want, but we can use numbers instead. Every colour is stored in a big database, each one with a number. We have 64 colours in total to choose from.

This means we can use a random selection for our colours. We'll need the `random()` **function** to achieve this!

```
1. ink( random( 64 ) )
2. loop
3.   print( "Hello Nintendo" )
4.   print( "What about me?" )
5.   update()
6. repeat
```

The brackets might look a little tricky here. Remember, the whole of `random(64)` must go in the brackets for the `ink()` line. It might look wrong at first, but we must have the same number of **open** and **close** brackets.

To change the colours, you will have to stop and start the program again. A little boring... Let's change that!

```
1. loop
2.   ink( random( 64 ) )
3.   print( "Hello Nintendo" )
4.   print( "What about me?" )
5.   update()
6. repeat
```

Now we've brought the colour line **into the loop**! Can you predict what will happen?

This way, **FUZE** will set the random colour on every repetition of the **loop**!

Let's say we wanted one colour for "Hello Nintendo" and another colour for "What about me?".

```
1. loop
2.   ink( fuzepink )
3.   print( "Hello Nintendo" )
4.   ink( fuzeblue )
5.   print( "What about me?" )
6.   update()
7. repeat
```

In the example above, line 2 sets a colour for the `print()` function on line 4. Then, line 4 sets a different colour for the `print()` function on line 5. If we want something to affect a specific line, we must put the instructions *before* the chosen line.

Finally, let's change the size of our text with the `textSize()` function:

```
1. textSize( 100 )
2. loop
3.   ink( fuzepink )
4.   print( "Hello Nintendo" )
5.   ink( fuzeblue )
6.   print( "What about me?" )
7.   update()
8. repeat
```

`textSize()` allows us to set the size of our text in pixels. With `textSize(100)` the maximum height for our letters will be 100 pixels tall.

Congratulations! You've written the best program ever. It's all downhill from here.

Functions and Keywords used in this Tutorial

`ink()`, `loop`, `print()`, `repeat`, `sleep()`, `textSize()`, `update()`

TUTORIALS

Tutorial 2: Variables

Variables are another of the most important parts of programming.

What are they? Well, put very simply a **variable** is a label. A label *you* create to store a piece of information.

Why do we need labels? Well, imagine you were trying to find a needle in a haystack. Quite the challenge, right? Not if we had a nice big signpost telling us exactly where to find it!

The main reason why we use **variables** is that they allow us to change and manipulate things in a program.

Almost every single game uses **variables** all the time, to keep track of everything from a *score* to *health* to *stats* to *screen position* to the *controller buttons* and so on.

Let's get started. I hope you like sweets!

```
1. sweets = 3
```

Type the code above into the **FUZE⁴ Nintendo Switch** code editor and press **+** to run the program.

Nothing will happen at all. In fact, you'll be taken instantly back to the editor!

However, in the computer's brain we have taken the number **3** and stored it in a place called *sweets*.

Now, change your program so that it looks like the one below and run.

```
1. sweets = 3
2. print( sweets )
3. update()
4. sleep( 2 )
```

All we are doing is instructing **FUZE** to print the contents of the *sweets* **variable** on screen. You should see the number **3** appear! **FUZE** is finding the information labeled as *sweets*, and printing whatever it finds.

We can store anything in a **variable**. The code below will also work:

```
1. sweets = "Delicious"
2. print( sweets )
3. update()
4. sleep( 2 )
```

This time though, we'll get the word "Delicious" instead of the number **3**. Make sense? Great!

Let's make this a little more complicated. We'll be using a couple of new, very important things.

Change your code in the **FUZE⁴ Nintendo Switch** editor so it looks like the program below and run it. It should count down our sweets until we have none left.

```
1. sweets = 3
2. while sweets > 0 loop
3.     clear( black )
4.     print( "I have ",sweets," sweets in my bag." )
5.     print( "If I eat one... then... " )
6.     sweets -= 1
7.     update()
8.     sleep( 1 )
9. repeat
10. print( "I have no sweets left... Oh no." )
11. update()
12. sleep( 2 )
```

First things first - we have a brand new **function** here. `clear()` is used to clear the screen with a colour. Try putting a different colour in the brackets.

Notice that on line 3, we have the **variable** name `sweets` between commas, not as part of the text. If we were to say `print("sweets")` we would get the word "sweets". See the *speech marks* in the brackets?

But remember that when we say `print(sweets)`, with *no speech marks*, we get the contents of the **variable**.

Let's talk about the new things we've introduced here. The first one is a new different type of **loop** called a **while loop**.

While

A **while loop** is a conditional loop, which repeats until a certain condition is met.

This **while loop** repeats as long as the `sweets` **variable** has a value *greater than zero* (`sweets > 0`).

For our **while loop** to actually stop and move on with the program, we must reduce the value of the `sweets` **variable** to 0.

The next tricky part is line 6.

Minus equals (-=) and Plus equals (+=)

The line of code which reduces the value of the `sweets` **variable** is line 6.

```
6. sweets -= 1
```

The sign after `sweets` is called **minus equals**. We're going to see these signs a lot as we move forward, so let's do our best to understand them. `-=` means we are subtracting from the value stored in our **variable**. `+=` or **plus equals** is adding to the value.

Really, line 6 actually reads:

```
6. sweets = sweets - 1
```

This means: Redefine the `sweets` **variable** to be equal to whatever is currently in the `sweets` **variable** *minus* 1.

Using `-=` and `+=` helps us save a lot of time because we don't need to write every **variable** name twice!

Still with us?

Awesome! Using **variables** becomes second-nature eventually, but it can be a bit strange to get used to at first.

Try to rewrite the program so that instead of taking sweets away, you begin with 0 sweets and gain them as the loop continues. You should change the **while loop** too so the program stops when you reach a certain number of sweets.

Well done! You reached the end. See you in the next tutorial where we'll talk about another one of the core programming techniques - **If Then Statements**.

Functions and Keywords used in this Tutorial

`clear()`, `loop`, `print()`, `repeat`, `sleep()`, `update()`, `while`

TUTORIALS

Tutorial 3: If Statements

Onward and upwards! This tutorial will cover another of the most important concepts in programming: **If statements**.

This one actually doesn't need much explaining... We use If Then Statements all the time in normal life!

Maybe this sounds familiar to you...

"**If** you do all of your homework, **then** you can have some extra time playing on your Nintendo Switch!"

Or perhaps...

"**If** you eat all of your vegetables, **then** you can have extra chocolate cake, or **else** you can go to bed with no dinner!"

Hopefully those sentences make some sense! An **If statement** checks **if** something is happening, and **then** does something. If it's anything **else**, we do something else!

Okay, enough rambling! Let's write a small quiz program using what we've learned so far.

```
1. print( "Welcome to the FUZE ultimate quiz of supreme difficulty \n" )
2. update()
3. sleep( 2 )
4. print( "Question 1. What is the capital of Japan? \n" )
5. update()
6. sleep( 2 )
7. answer = input( "What is the capital of Japan? \n" )
8. if answer == "Tokyo" then
9.     print( "CORRECT! Well done! \n" )
10. else
11.     print( "INCORRECT! I can't believe you didn't know that... \n" )
12. endif
13. update()
14. sleep( 3 )
```

We've got a new **function** here, take a look at line 7.

input()

On line 7 we use the `input()` **function** to allow the player to type an answer to our question. The `input()` **function** will bring up the **FUZE⁴ Nintendo Switch** keyboard on screen to type in whatever you like. Whatever the player types is stored in a **variable**. We have called our **variable** `answer`.

Something we need when using the `input()` function is to give the player a prompt message. We have put a reminder of the question: "What is the capital of Japan?", but you can use whatever you like! If you would like to display no prompt simply put empty speech marks in the brackets.

If Then Statement

Lines 8 to 12 are our If Then Statement. It begins with `if` to state the condition. In the example, we check `if` the contents of the **variable** called `answer` is *exactly equal to* the text string "Tokyo". `If` it is, `then` we print "Correct! Well done!".

We use the **keyword** `else` as part of the **if statement** to give a different result if the condition is *not* met. In this case, we know that if our condition is not met this means the answer is incorrect, so we can tell the player.

Lastly, we must use `endif` to finish the **if statement**, otherwise we will run into some problems later! Without `endif` the computer will treat any lines that may follow as part of the **if statement**.

Single Equals (=) and Double Equals (==)

Notice the double equals sign on line 8? When we are **comparing** two things, we must use **double equals** (`==`). When we are **assigning** a value to a variable we use **single equals** (`=`)

Keeping Score

Okay... Let's step this up. We can use a **variable** to keep track of the player **score** too and really turn this into a playable game.

```
1. score = 0
2. print( "Welcome to the FUZE ultimate quiz of supreme difficulty \n" )
3. update()
4. sleep( 1 )
5. print( "Question 1. What is the capital of Japan? \n" )
6. update()
7. sleep( 2 )
8. answer = input( "What is the capital of Japan?" )
9. if answer == "Tokyo" then
10.     print( "CORRECT! Well done! \n" )
11.     score += 1
12. else
13.     print( "INCORRECT! I can't believe you didn't know that... \n" )
14. endif
15. update()
16. sleep( 2 )
17. print( "You scored... ",score, "\n" )
18. update()
19. sleep( 1 )
20. if score == 1 then
21.     print( "Congratulations! Full Marks" )
22. else
23.     print( "Better luck next time." )
24. endif
```



```
25. update()  
26. sleep( 3 )
```

In the example above we have added a score **variable** to line 1. It begins at **0** because our player hasn't answered any questions yet!

Line 11 is a new line which increases the contents of the score variable by 1 **if** the player answers correctly. Remember, we use **+=** to do this.

Next, lines 20 to 24 are a new **if statement** to give the player a message depending on their score.

Using what we we have learned, can you write 2 more questions for the quiz? You should place your new questions between line 16 and line 17, because lines 17 to 26 finish the quiz and tell the player their score.

You will need to change line 20 **if score == 1 then** to be equal to your maximum total score. For example, if your quiz contains 3 questions and you score 1 point for each correct answer, the line should read **if score == 3 then**

Perhaps use the **ink()** **function** to bring your quiz into full colour!

Functions and Keywords used in this tutorial

`else`, `endif`, `if`, `input()`, `print()`, `sleep()`, `then`, `update()`

TUTORIALS

Tutorial 4: Screen

Hello again! In this tutorial we'll be looking at what the screen is, the basics of how it works, and how we can use it in our programs.

The screen is a huge collection of tiny, tiny little lights called **pixels**. There are lots and lots of them. In your **Nintendo Switch** screen, there are a whopping **921600** of them. Almost a million!

Whenever your screen is on, whenever you play a game, browse the eShop, or simply look at the HOME menu, your **Nintendo Switch** is changing these 921600 little lights at an amazing speed. About 60 times per second, actually.

Just think about that for moment - it should blow your mind!

X and Y

You're probably familiar with the **x** and **y** axis already. But, just in case you've never heard of them before let's have a quick look.

The huge collection of pixels we mentioned is arranged into an **x axis** and a **y axis**. If something moves along the **x axis**, it moves left or right. If something moves along the **y axis**, it moves up or down!

Take a look at this picture to see what we mean:



Notice the direction the arrows are pointing? This tells us the way the numbers increase along the axes.

Both axes begin at 0 and go up 1 pixel at a time, across and down the screen.

Your **Nintendo Switch** screen has 1280 pixels on the **x axis** and 720 pixels on the **y axis**. Multiply them together to get the 921600 number we mentioned earlier!

Take a look at this image to see how the numbers increase along the axes:



The zero in the top left corner tells us that 0 on the **x** and **y** axis is in the top left corner of the screen.

The maximum number for the **x axis** is the furthest right part of the screen, and the maximum number for the **y axis** is the bottom of the screen.

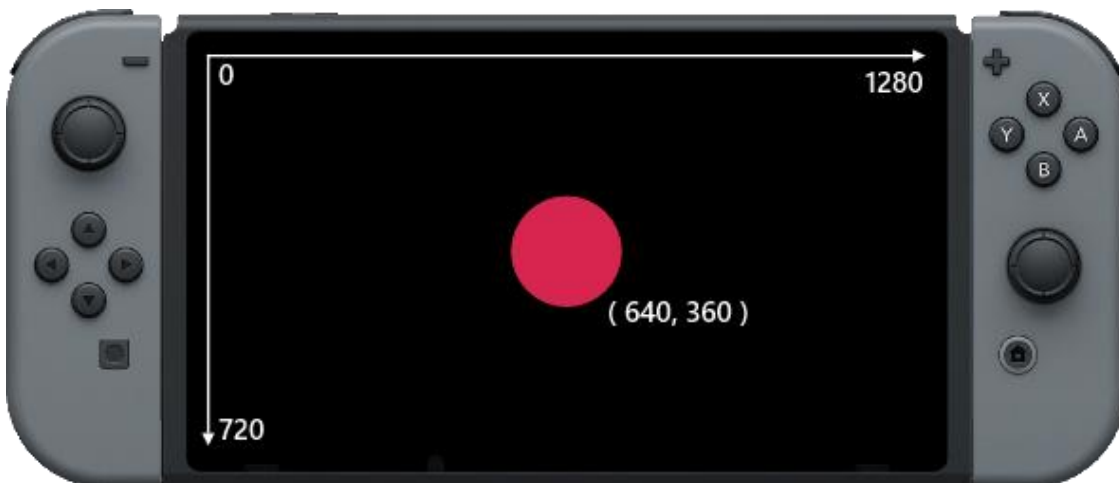
TV Mode and Handheld Mode

Something important to bear in mind is that the screen resolution (the number of pixels in a screen) changes between Handheld Mode and TV Mode. While your **Nintendo Switch Console** is in the dock, the screen resolution will be **1920** by **1080** pixels, rather than **1280** by **720**.

Using Screen Co-ordinates in a Program

Let's say we want to put a circle right in the middle of our screen. We would need to know the halfway point on both axes. Half of 1280 is 640, and half of 720 is 360.

If we use 640 and 360 as co-ordinates for an **x** and **y** position on the screen, we should get something like this:



Let's see what this would look like in code.

```
1. loop
2.   clear()
3.   circle( 640, 360, 100, 32, fuzepink, false )
4.   update()
5. repeat
```

Nice and simple. We have a **loop**, we use the `clear()` **function** to clear the screen and the `update()` **function** to send our information to the screen.

Take a look at line 3. This line is responsible for creating the circle.

The `circle()` **function** needs 6 pieces of information in brackets, separated by commas.

The first number in the brackets is the **x axis** position. The second number is of course the **y axis** position! There's our co-ordinates!

The third number (100) is the *size* of the circle. Actually, the real name for this is the *radius* of the circle - the distance in pixels from the middle of the circle to the edge. If our circle has a radius of 100 pixels, it means the circle is 200 pixels wide in total.

Next up, we have the number of sides. Yes, you read that right. This circle has 32 sides. We see it as very smooth because the sides are incredibly small. We can change this number to change how our circle looks - in fact, we can turn it into a totally different shape by changing this number! Try putting some different numbers here to see what we mean.

Moving the Circle

If we want our circle to move, we'll need some **variables** to store the **x** and **y** positions.

Change your code to look like the program below. It won't behave any differently just yet.

```
1. x = 640
2. y = 360
3. size = 100
4.
5. loop
6.   clear()
7.   circle( x, y, size, 32, fuzepink, false )
8.   update()
9. repeat
```

Now we have used **variables** to display the circle on screen. Everything is exactly the same, except we can *change* these during the program to move the circle.

Remember `+=` from the **variables** tutorial? We'll be using this here. We are only adding one line of code, check it out:

```
1. x = 640
2. y = 360
3. size = 100
4.
5. loop
6.   clear()
```

```
7.   circle( x, y, size, 32, fuzepink, false )
8.   x += 1
9.   update()
10.  repeat
```

We've added one line just after our `circle()` **function**. This line increases the value of the `x` **variable** by 1 each time the **loop** repeats. Because we are using the `x` **variable** as the circle's `x` position, this will cause the circle to move across the screen.

Run the program to see! You might notice a little problem though... The circle doesn't stop!

Making the Circle Bounce

Once the `x` **variable** becomes too big our circle is being drawn off screen. In order to make the circle bounce off the edge and come back, we'll need a couple of things.

First, we must understand that the reason the circle moves in any particular direction is because of what we **do** to the `x` or `y` position.

For now, let's just think about the `x` axis

If we *increase* the `x` position, we are moving to the right. If we *decrease* the `x` position, we move to the left.

```
8.   x += 1
```

This is the line responsible for moving the circle.

The speed the circle moves is entirely down to the number we increase the `x` **variable** by. At the moment, we are increasing it by 1 pixel each time the line is read. If we increase this number, the circle moves faster. Decrease the number and it will move slower. So far so good?

If we store this number in a **variable**, we can do some interesting things. Change your code to look like this:

```
1. x = 640
2. y = 360
3. size = 100
4. xSpeed = 3
5.
6. loop
7.   clear()
8.   circle( x, y, size, 32, fuzepink, false )
9.   x += xSpeed
10.  update()
11.  repeat
```

We've created a new variable on line 4 called `xSpeed`. This variable is used to control the speed of circle along the `x` **axis**, so it makes sense to name it something like this!

The `circle` line has changed too. We now use the variable in this line instead of just a number.

Right, now we're ready to start making the ball bounce!

The **x variable** is increasing all the time. The screen is 1280 pixels wide. When **x** becomes *bigger than* the width of the screen, we know our circle is off screen. We need to do something when that happens.

For this, we'll need an **if statement**. Take a look below:

```
1. x = 640
2. y = 360
3. size = 100
4. xSpeed = 3
5.
6. loop
7.   clear()
8.   circle( x, y, size, 32, fuzepink, false )
9.   x += xSpeed
10.  if x > gwidth() then
11.    xSpeed = -xSpeed
12.  endif
11.  update()
12. repeat
```

Our **if statement** is on line 10. We check if the **x** variable has become *bigger than* (>) `gwidth()` (the width of screen). When you run this program, the ball should bounce off the right hand side of the screen. If you look carefully, you might notice something not quite right.

The **x** variable is the middle of our circle. So, when **x** is bigger than the width of the screen, half of the circle is already gone!

To get the ball bouncing properly off the edge, we must check if **x + size** has become bigger than `gwidth()`. Remember, the size **variable** is being used as the *radius* of the circle, which is *half* of the total width. Because of this, **x + size** gives us the exact edge of the circle. Let's modify the code slightly:

```
1. x = 640
2. y = 360
3. size = 100
4. xSpeed = 3
5.
6. loop
7.   clear()
8.   circle( x, y, size, 32, fuzepink, false )
9.   x += xSpeed
10.  if x + size > gwidth() then
11.    xSpeed = -xSpeed
12.  endif
13.  update()
14. repeat
```

Alright! Now we're bouncing properly... But we have another problem. The ball just goes straight off the other side of the screen!

When x becomes too big, the circle goes off the right hand side, but if x becomes too small it goes off the left!

Before, x was becoming bigger than `gwidth()`. However, the left side of the screen is 0 pixels. So our problem is that that x has become *less than* 0.

There's a very clever solution to this problem:

```
1. x = 640
2. y = 360
3. size = 100
4. xSpeed = 3
5.
6. loop
7.   clear()
8.   circle( x, y, size, 32, fuzepink, false )
9.   x += xSpeed
10.  if x + size > gwidth() or x - size < 0 then
11.    xSpeed = -xSpeed
12.  endif
13.  update()
14. repeat
```

We can simply add an **or** to our **if statement** for this one. The right hand side of the ball is $x + \text{size}$, but the left side is $x - \text{size}$. So, we check if $x - \text{size}$ has become *less than* 0. If it does, we simply do the exact same thing as before. We make `xSpeed` negative.

But `xSpeed` is already negative! Well, because of the magic of maths, when you make a negative number negative, it becomes positive. Two wrongs do not make a right, but two negatives do make a positive!

If `xSpeed` is positive, `xSpeed = -xSpeed` makes it negative.

If `xSpeed` is negative, `xSpeed = -xSpeed` makes it positive!

This will cause our ball to bounce neatly around the **x axis**.

Okay, let's do the same with the **y axis** to finish this project. Feel free to try for yourself, but check out the code below if you need help:

```
1. x = 640
2. y = 360
3. size = 100
4. xSpeed = 3
5. ySpeed = 3
6.
7. loop
8.   clear()
9.   circle( x, y, size, 32, fuzepink, false )
10.  x += xSpeed
11.  y += ySpeed
12.  if x + size > gwidth() or x - size < 0 then
13.    xSpeed = -xSpeed
```

```

14.     endif
15.     if y + size > gheight() or y - size < 0 then
16.         ySpeed = -ySpeed
17.     endif
18.     update()
19. repeat

```

Make it Awesome!

What we've just created are the basics of the game **Pong**! All we would need now are a couple of bats and a score. We won't be adding this to the program here, but check out the demo programs for a full version of pong to play around with.

What we can do, however, is really make this program more visually exciting.

The first thing we do in the **loop** is `clear()` the screen. We must have this to see a clean moving ball, otherwise we will see all of the previous circles that have been drawn. This actually looks very cool indeed. Try deleting the `clear()` line.

```

1. x = 640
2. y = 360
3. size = 100
4. xSpeed = 3
5. ySpeed = 3
6.
7. loop
8.     circle( x, y, size, 32, fuzepink, false )
9.     x += xSpeed
10.    y += ySpeed
11.    if x + size > gwidth() or x - size < 0 then
12.        xSpeed = -xSpeed
13.    endif
14.    if y + size > gheight() or y - size < 0 then
15.        ySpeed = -ySpeed
16.    endif
17.    update()
18. repeat

```

Now we see a line being drawn all over the screen. Try changing the `false` in the `circle()` line to `true` to see a big difference.

```

8.     circle( x, y, size, 32, fuzepink, true )

```

This will give us an outline of a circle instead.

Try changing the `32` in the `circle()` line to totally change the shape. See what it looks like with a `3`!

```

8.     circle( x, y, size, 3, fuzepink, true )

```

Now let's *really* step things up.

It would be truly awesome if we could change the colours while we draw the circles for a beautiful rainbow effect.

This next piece of code is going to look quite difficult to understand, and we're certainly not going to explain it all in this tutorial as it wouldn't quite be the right place. However, the end result is so cool that we thought it would be best to include it here to give you a chance to play.

To achieve the colour changing rainbow effect we are changing the red, green and blue (RGB) values of a colour **vector**. Confused? Fear not, you'll learn all about **vectors** in the later tutorials.

All of the important settings to change in the program are in the **variables** at the start of the program. Change these settings around to see what differences you can make!

Copy the code below into the editor:

```
1. // circle properties
2. x = gwidth() / 2
3. y = gheight() / 2
4. radius = 100
5. sides = 6
6. outline = false
7.
8. // speed variables
9. xSpeed = 33
10. ySpeed = 66
11.
12. // colour variables
13. cSpeed = 0.01
14. col = { 1, 0, 0, 1 }
15.
16. loop
17.     if col.r > 0 and col.b <= 0 then
18.         col.r -= cSpeed
19.         col.g += cSpeed
20.     else
21.         if col.g > 0 then
22.             col.g -= cSpeed
23.             col.b += cSpeed
24.         else
25.             col.b -= cSpeed
26.             col.r += cSpeed
27.         endif
28.     endif
29.     circle( x, y, radius, sides, col, outline )
30.     x += xSpeed
31.     y += ySpeed
32.     if x + radius > gwidth() or x - radius < 0 then
33.         xSpeed = -xSpeed
34.     endif
35.     if y + radius > gheight() or y - radius < 0 then
36.         ySpeed = -ySpeed
37.     endif
38.     update()
39. repeat
```

There we have it! Run this program to see some rainbow patterns appear on screen. Change the radius, sides and outline **variables** to experiment with different types of shapes.

Change the xSpeed and ySpeed **variables** to change the pattern that is drawn.

Change the cSpeed **variable** to speed up or slow down the colour change effect.

Have fun, and see you in the next tutorial!

Functions and Keywords used in this tutorial

`circle()`, `clear()`, `getWidth()`, `getHeight()`, `loop`, `repeat`, `update()`

TUTORIALS

Tutorial 5: Arrays

Wow, we're really powering through! In this tutorial, we will cover **arrays**. What they are, how to use them and why we should.

Arrays are incredible and powerful tools for programming. In fact, almost every video game in the world uses many arrays to keep track of everything it needs.

An **array** is a table of **variables**. We can give each position in the table a value and use it later in the program.

Arrays are used for a huge variety of things, from inventories to maps to putting stars in your space background, but in this first tutorial we'll be creating a simple fortune teller game.

A fortune teller needs to have a selection of answers stored so we can randomly choose one to give the player.

Before we write our selection of answers, we must create our array. To do this, we must use the word **array**. Who would have thought?!

Type the following line into the **FUZE⁴ Nintendo Switch** code editor:

```
1. array answers[4]
```

This line sets up an empty **array**. Think of this like an empty chest of drawers. Notice the square brackets in this line. When we *create* or *access* an array, we always use square brackets. Because we put a **4** in the square brackets, we have **4** places to store things.

We have called our **array** "answers". This can of course be anything you like, but as always it's good to name our **variables** and **arrays** as things which make sense.

Next up, we're going to store some information in the *elements* to our **array**. In each of these elements, we can store something. We will store a variety of statements.

```
1. array answers[4]
2. answers[0] = "It is certain!"
3. answers[1] = "It does not look good..."
4. answers[2] = "You might be in luck!"
5. answers[3] = "Definitely not."
```

Edit your code so that it looks like the program above. Feel free to copy and paste the code if you don't feel like typing!

The program above now stores the four different text answers we have into the four elements of our array. These are labeled as element **0**, **1**, **2** and **3**.

The image below might help you picture this in your mind:

0	1	2	3
"It is certain!"	"It does not look good..."	"You might be in luck!"	"Definitely not."

As you can see, each position in the table is labeled with a number (0 - 3). Because of this, we can easily access any element and use the information.

There is another way to lay out our array too, and the results are exactly the same. See below:

```

1. answers = [
2.     "It is certain!",
3.     "It does not look good...",
4.     "You might be in luck!",
5.     "Definitely not."
6. ]

```

There is no difference whatsoever in the result. It's all about which ever you find easiest to understand!

Below we've made a few additions to the program. Add the lines 6 to 19 and run it.

```

1. array answers[4]
2. answers[0] = "It is certain!"
3. answers[1] = "It does not look good..."
4. answers[2] = "You might be in luck!"
5. answers[3] = "Definitely not."
6. print( "Welcome to the fortune teller, ask me a question! \n" )
7. update()
8. question = input( "Type your question here." )
9. sleep( 1 )
10. print( "Are you ready to know your fortune? \n" )
11. update()
12. sleep( 1 )
13. print( "My answer to your question is... \n" )
14. update()
15. sleep( 1 )
16. num = random( 4 )
17. print( answers[num] )
18. update()
19. sleep( 2 )

```

Now we can play! With a couple of sleep commands, we can really ramp up the tension before we get our answer!

Accessing the array

In order to print one of our answers on the screen, we must *access* the **array**. Line 17 is where we do this.

Because we want to print a random answer, we use the `random()` **function** to choose a random number. We can then use this as an *index* into our **array**.

We store the random number in a **variable** called `num` and use that **variable** to access the **array** on line 17:

```
17. print( answers[num] )
```

Challenge

Could you add some more answers to the **array**? You will need to create additional lines of code to do this. Try to add 3 more answers.

Hint: Don't forget to look at this line:

```
16. num = random( 4 )
```

Re-cap

An **array** is a table of **variables** used to store information.

Every position in the table has a number which we can use to *access* the information stored there.

Arrays are used *everywhere*! A computer controls a screen using a gigantic array, where each element of the **array** is a single pixel.

Stay tuned for the next **arrays** project in which we will throw some shapes around the screen!

Functions and Keywords used in this tutorial

`array`, `input()`, `print()`, `sleep()`, `update()`

TUTORIALS

Tutorial 6: Using Controls

Good to see you again!

In this tutorial, we'll begin to learn how to use the **Joy-Con controllers** in our programs.

If we want to start writing something like a game, we're going to have to use the Joy-Con controllers sooner or later!

To do this, we'll need our good old **if statements**. After all, we are checking **if** a button is being pressed!

Here's how we do it. As always, we start super simple.

Enter the following code into the **FUZE⁴ Nintendo Switch** code editor:

```
1. loop
2.   clear(black)
3.
4.   joy = controls(0)
5.
6.   if joy.a then
7.     print("You are pressing the A Button!")
8.   endif
9.
10.  update()
11. repeat
```

Here's a nice and simple program to get the idea across. All we want to happen is for the text "You are pressing the A Button!" to appear when we press the A button.

Notice that this program is all in a **loop**. We want our program to run continuously. The first thing we do in our **loop** is to clear the screen. We have a `clear()` and an `update()` **function** because we are changing what we want to appear on screen.

Now, onto the important part.

Take a look at line 4. Here we are *calling* a **function** called `controls()`. This **function** gives us the *current state of all of the controls* and this is exactly what we need to use the Joy-Con controllers in our program.

We store the result of the `controls()` **function** in a **variable** we've called `joy`.

Now that we've done this, we can access *any* of the buttons by using `joy.a`, `joy.b`, `joy.x` and so on.

On line 6 is our **if statement** which uses the controls data. We check **if** the A button is being pressed with **if joy.a then**.

A little more advanced...

It's important to realise that when a computer checks an **if statement**, it can only be either **true** or **false**.

When FUZE reads line 6, it checks to see if `joy.a` is **true** or **false**.

If the A button *is* being pressed, `joy.a` is **true**.

If the A button is *not* being pressed, `joy.a` is **false**.

When we write **if** `joy.a` **then**, there is a little something hidden. Really, line 6 reads: **if** `joy.a == true` **then**

Challenge

Can you add an **if statement** to this program to check another controller input? You should also write a line of text to display to the user that a button is being pressed.

Check out the user guide page for the `controls()` **function** to see *all* of the possible inputs for the Joy-Con controllers. You can find that page just [here](#).

Moving a Ball Using the Control Stick

Remember our screen tutorial? In that project, we learned how to move a circle around the screen using **variables**.

What if we wanted to move the circle using the **Joy-Con** control sticks? It's actually beautifully simple.

We'll need the `controls()` **function**. First, let's remind ourselves of the basic program template.

```
1. x = gwidth() / 2
2. y = gheight() / 2
3. radius = 100
4.
5. loop
6.     clear()
7.     circle( x, y, radius, 32, fuzepink, false )
8.     update()
9. repeat
```

We have a simple **loop** which puts a circle on screen at the **x** and **y variables** defined at the start of the program.

Let's add the `controls()` **function**, and define a **variable** to use it.

```
1. x = gwidth() / 2
2. y = gheight() / 2
3. radius = 100
4.
5. loop
6.     clear()
```

```
7. joy = controls( 0 )
8. circle( x, y, radius, 32, fuzepink, false )
9. update()
10. repeat
```

Just like before, we have created a **variable** called `joy` to store the result of the `controls()` **function**.

The left control stick is accessed with `.lx` and `.ly` for the **x** and **y** axes of the control stick.

It's important to understand exactly how the `controls()` **function** works with the control stick. Take a look at the image below:



As you can see, the value returned by the `controls()` **function** which represents the left control stick is **0** when *not being pushed* in a direction.

When pushed to any side, the value changes towards either 1 or -1. There are a lot of numbers in between! Actually, on one axis there are 65000 different positions!

If the `.lx` value is *greater than 0*, we know the control stick is being pushed *towards* a positive number, and therefore this tells us the control stick is pushed to the right. If the value is *less than 0*, we know it is being pushed towards a negative number, and therefore is being pushed to the left.

Let's modify our code with this new knowledge so we can move in both directions on the **x** axis.

```
1. x = gwidth() / 2
2. y = gheight() / 2
3. radius = 100
4.
5. loop
6.   clear()
7.   joy = controls( 0 )
8.   x += joy.lx
9.   circle( x, y, radius, 32, fuzepink, false )
10.  update()
11. repeat
```

Look at that! We've added just a single line of code.

On line 8, we simply *add* the value of the left control stick to the **x variable**. If the control stick is being pushed to the right, we have a positive number and so the circle moves to the right. If the control stick is being pushed to the left, we have a negative number and so the circle moves to the left.

Now, because these numbers are very small indeed, our circle will move incredibly slowly. Not very useful. Let's introduce a speed **variable** which we can use as a multiplier. Take a look below:

```
1. x = gwidth() / 2
2. y = gheight() / 2
3. radius = 100
4. speed = 8
5.
6. loop
7.     clear()
8.     joy = controls( 0 )
9.     x += joy.lx * speed
10.    circle( x, y, radius, 32, fuzepink, false )
11.    update()
12. repeat
```

With this change, we will see a much greater movement effect.

On line 9, we increase the **x variable** by the left control stick value *multiplied by the speed variable*. This gives us 8 times faster movement. Try changing the **speed variable** to see different results.

Okay, so let's move the circle on the **y axis** too! This one is just slightly more complicated.

```
1. x = gwidth() / 2
2. y = gheight() / 2
3. radius = 100
4. speed = 8
5.
6. loop
7.     clear()
8.     joy = controls( 0 )
9.     x += joy.lx * speed
10.    y -= joy.ly * speed
10.    circle( x, y, radius, 32, fuzepink, false )
11.    update()
12. repeat
```

Line 10 is our new line. Notice that we use **-=** instead of **+=** for the **y axis**. This is because the top of our screen is 0 on the **y axis**, but the control stick **y axis** is *positive* when being pushed upwards. Take a look at our control stick image again:



When we move the stick upwards, we receive a positive number. In order to make the circle move upwards on the screen, we need to *decrease* the **y variable**, not increase it. For this reason, we use **-=**.

Creating Boundaries

All we need now is a few **if statements** to stop our circle from moving off screen. Just like in the screen tutorial, we'll be checking the **x** and **y variables**, but instead of reversing direction, we will be redefining the **x** and **y variables**.

We'll start by stopping the circle at the edges of the **x** axis.

```
1. x = gwidth() / 2
2. y = gheight() / 2
3. radius = 100
4. speed = 8
5.
6. loop
7.   clear()
8.   joy = controls( 0 )
9.   x += joy.lx * speed
10.  y -= joy.ly * speed
11.  if x + radius > gwidth() then
12.    x = gwidth() - radius
13.  endif
14.  if x - radius < 0 then
15.    x = radius
16.  endif
17.  circle( x, y, radius, 32, fuzepink, false )
18.  update()
19. repeat
```

Lines 11 to 16 contain our first set of **if statements**. All we need to do is check whether the edge of the circle (**x + radius**) has become *greater than* the edge of the screen (**gwidth()**). If it is, we redefine **x** to be *equal to* the edge of the screen *minus* the radius (**x = gwidth() - radius**).

We do the same thing but reversed for the left side of the screen. We check if **x - radius** has become *less than 0*, and if it has, we redefine **x** to be the left side (0) *plus* the radius of the circle. We can write this simply as **x = radius**.

Easy enough! Now let's do the same for the **y** axis:

```
1. x = gwidth() / 2
2. y = gheight() / 2
3. radius = 100
4. speed = 8
5.
6. loop
7.   clear()
8.   joy = controls( 0 )
9.   x += joy.lx * speed
10.  y -= joy.ly * speed
11.  if x + radius > gwidth() then
12.    x = gwidth() - radius
13.  endif
14.  if x - radius < 0 then
```

```
15.     x = radius
16.     endif
17.     if y + radius > gheight() then
18.         y = gheight() - radius
19.     endif
20.     if y - radius < 0 then
21.         y = radius
22.     endif
23.     circle( x, y, radius, 32, fuzepink, false )
24.     update()
25. repeat
```

There we have it! This type of movement code can be applied to any program. In later tutorials we will be covering more advanced movement techniques, but just understanding this simple type of movement will open up a world of experimentation in your own projects!

See you in the next tutorial!

Functions and Keywords used in this tutorial

[clear\(\)](#), [controls\(\)](#), [endIf](#), [if](#), [input\(\)](#), [loop](#), [print\(\)](#), [repeat](#), [then](#), [update\(\)](#)

TUTORIALS

Tutorial 7: For Loops

Hello again!

Okay, we know how **loops** work and we understand **variables**. Now we can cover another incredibly useful tool for programming: **For loops**.

For loops are used all the time in programming. Sometimes used for animations, sometimes just as a counter, sometimes used to check over a whole array very quickly, **For loops** are a valuable and versatile tool.

```
1. for i = 0 to 10 loop
2.   print( i )
3.   update()
3. repeat
4. sleep( 2 )
```

The program above is a counter from 0 to 10. **For loops** are special because they *define* a **variable** within the **loop**.

In our example, we are defining a **variable** called *i*. The first time this **loop** goes around, *i* is equal to 0. As we know, when FUZE reads the **repeat** command, it returns to the last **loop** line. However, now *i* is equal to 1. Next time the **loop** goes around, *i* is equal to 2. This carries on increasing by 1 each time until we finally get to 9.

When *i* is equal to 10 the condition is completed and the **loop** finishes!

Note: The **loop** counts *up to but not including* the last value. So our example, **for i = 0 to 10 loop** will count from 0 to 9 and *will not* reach 10.

Using a For Loop to Create Sound

So why might we want to do this? Well, below is quite a cool example of how to use a **for loop** to create interesting sounds using our `playNote()` **function**. Change your code so it looks like the program below and run it. Make sure your volume is up!

```
1. for i = 0 to 900 loop
2.   playNote( 0, 1, i, 1, 10, 0.5 )
3.   update()
4.   sleep( 0.01 )
5. repeat
```

Here, we have used our variable *i* again, but this time we are counting to 900 instead of 10. Remember, we will never quite reach 900. *i* will only ever reach 899.

The `playNote()` **function** is quite a cool one to get used to because it can be used to create your own music! `playNote()` plays a note of a certain *frequency* (pitch). In this case, our frequency is determined by the *i* **variable**.

The first time around, our `playNote()` will play a frequency of 0hz. The next time around the **loop**, the frequency will be 1hz, then 2hz and so on until 899hz. This will give us a sequence of notes in a **loop**.

We are also using a `sleep()` **function** with a *very* short delay of 0.01 seconds. With a delay this short, the notes sound more like a sweep upwards, kind of like an alarm sound or a siren.

What if we wanted to play a more musical sounding series of notes? Well, we'll need two things: a longer delay, and something called a **step**.

We use a **step** in a **for loop** to count in specific amounts. Take a look below:

```
1. for i = 0 to 900 step 100 loop
2.   playNote( 0, 1, i, 1, 10, 0.5 )
3.   update()
4.   sleep( 0.1 )
5. repeat
```

In this example, our **for loop** still counts from 0 to 900, but in jumps of 100 each time. This means the first cycle of the loop is still 0, but the next is 100, then 200, 300 and so on. However, this time when the **i variable** reaches 800 the **loop** finishes.

We have used a longer delay this time to give us a longer note duration. The result will sound like a scale moving upwards. Much more musical!

Try using different steps and beginning or end values to get different results!

Using a For Loop with Shapes

Another example of how we can use a For Loop is for animation. Let's say we wanted to draw a line one pixel at a time across the screen.

```
1. for x = 0 to gwidth() loop
2.   box( x, gheight() / 2, 1, 1, white, false )
3.   update()
3. repeat
```

Notice in this example we are naming our **variable** **x**. This is because we are using it to change the *x axis position* of a pixel. You can of course call your **variables** whatever you feel like!

Our **x variable** will count from 0 to the maximum number of pixels along the *x axis* of the screen.

When we use the `box()` **function**, we must put certain pieces of information in the brackets. The first two pieces are the **x** and **y** position of where we want our box to appear on screen. Next up, we have the width and height of the box. In our example, our box has a width and height of just one pixel. A very tiny box indeed! The next piece of information is the **colour** we want our box to be. Lastly, we say whether we want an outline of a box (**true**), or a filled in box (**false**).

For the **x** position, we are using our increasing number **x**. For the **y** coordinate, we are using `gheight() / 2`, which gives us the middle point on the **y** axis of the screen.

In short, this little program will draw a line across the middle of the screen, one pixel at a time. Watch it in all its glory!

Let's do something a bit more visually exciting than just a line.

Change your code to look like the program below, but see if you can figure out what will happen before you run the program!

```
1. for y = 0 to gheight() loop
2.   circle( random( gwidth() ), y, 100, 32, fuzepink, true )
3.   update()
4. repeat
```

`gheight()` as we know is a **function** which gives us the height of the screen in pixels.

In handheld mode, that number will be 720. In TV mode, that number is 1080.

Depending on how you are using your Nintendo Switch console, you will see either 720 or 1080 circles appear on your screen, one by one, moving down the screen.

Since the `x` part of the `circle()` **function** is a random number chosen out of the total number of pixels across the width of the screen, our circles will appear in a random position along the `x` axis each time a new one is created.

Using a For Loop with an Array

One of the most useful and practical applications of a **for loop** is using it to cycle through an **array** of information.

This technique is used a lot in programming. If we have lots of things to put on screen, chances are those things are stored in an **array** and a **for loop** is used to display them.

Let's see an example of using a **for loop** with an **array** to print lots of different names on the screen.

First, we'll need to create an **array** and populate it with information.

```
1. array names[5]
2. names[0] = "Dave "
3. names[1] = "Kat "
4. names[2] = "Luke "
5. names[3] = "Jon "
6. names[4] = "Rob "
```

Before we go any further let's remind ourselves of what we've done. We have created a table of information. Each part of the table has a number.

Now let's use a **for loop** to print the names on screen.

```
1. array names[5]
2. names[0] = "Dave "
3. names[1] = "Kat "
4. names[2] = "Luke "
5. names[3] = "Jon "
6. names[4] = "Rob "
7. loop
8.   clear()
```

```
9.     for i = 0 to 5 loop
10.    print( names[i] )
11.    repeat
12.    update()
13. repeat
```

We have a **for loop** on lines 9 to 11. This is used to print all 5 names on the screen. Without using a **for loop**, we would need to use 5 different `print()` lines to achieve what we want.

The clever part of this is that we use our increasing **i variable** in the `print()` line. The first time around the loop, the line reads:

```
10. print( names[0] )
```

Next time around, **i** is equal to 1 so it reads:

```
10. print( names[1] )
```

This carries on around and around until we have printed every name from the **array**.

Recap

A **for loop** is a **loop** which repeats a set number of times.

We define a **variable** in the **loop** which increases (or decreases) on each repeat.

There are many, many applications for **for loops**. It's impossible to cover them all here, since we must start simple. However, keep your eyes peeled in the upcoming tutorials and the demo programs in **FUZE⁴ Nintendo Switch** to see how else they can be used.

Well done - you're leveling up! See you in the next tutorial!

Functions and Keywords used in this tutorial

`box()`, `circle()`, `gWidth()`, `gHeight()`, `for`, `loop`, `playNote()` `print()`, `repeat`, `sleep()`, `step`

TUTORIALS

Tutorial 8: Functions

Are you feeling funky?

All jokes aside, this is very serious business. **Functions** are incredibly important tools in programming and it really feels like a level up when you understand them!

Almost every programming language contains lots of **functions** already, so if you're not sure what they are exactly, the chances are you've used them plenty of times without realising!

What is a Function?

A **function** is like a mini-program which we can use again and again. There are two kinds of **functions**. There are **built-in functions** which already exist, and there are **user functions** which we create.

You can spot a **function** very easily because it will have a pair of brackets `()` after it. These might contain information, or they might be empty.

Perhaps the most simple **function** which we all know and love is `print`. We use `print()` all the time to put words on the screen:

```
print( "like this!" )
```

In the `print()` brackets we put the information the **function** needs.

For `print()`, it's quite useless unless we tell it *what we want to print!* So, we must **pass** our text to the **function**.

Arguments

Some **functions** need more information than just one thing. For example, `printAt()` needs not only the text you want to appear, but also the **position** of that text on the screen:

```
printAt( 0, 0, "message" )
```

The pieces of information in the brackets separated by commas are called **arguments**. The `printAt()` function **must** have its **arguments** laid out like this:

```
printAt( x, y, text )
```

Our example of `printat(0, 0, message)` would print the word "message" at **x** position 0, **y** position 0.

Are you still with us? Of course you are! Oh, this is easy you say?

Well, let's go a little bit deeper.

A function is *really* a separate piece of code which a computer has to find and use. When a computer reads `print("hello!")` it takes our text "hello!" and **passes** it to the code it must run to make the words appear on screen.

This is the same for all **functions**. For example, let's take the `box()` **function**.

```
box( x, y, width, height, colour, fill )
```

As you can see, the box function's **arguments** are laid out here. The first two numbers in the brackets are the **x** and **y** coordinates for the top left point of the box. Next, the third and fourth numbers will be used as the width and height of the box. Next up we have the colour of the box and the last argument tells the box to be filled in or just an outline.

All of these pieces of information are **passed** to the box **function** and are used to put a box on screen!

Empty Brackets

It's important to mention that *not all functions* need something **passed** to them. Sometimes they just **return** something. For example, take the `gwidth()` or `gheight()` **functions**.

Notice the empty brackets in `gwidth()` and `gheight()`. We do not need to **pass** any information to them, but they **return** a number. That number is the width or height of the screen in pixels.

It might seem a bit strange to have to include empty brackets, but remember, **functions** *always* need brackets, whether they need information or not.

User-defined Functions

Still with us?

All of that was just the beginning!

It's very common in coding to need to do the same thing multiple times. This is the perfect time for a **user-defined function**, which is a fancy way of saying your own custom **function**.

We can create a custom **function** to do just about anything. Let's start nice and simple. We want to print the word "Hello" on the screen in blue ink and a specific size.

Without our own function, we might have something like this:

```
1. textSize( 50 )
2. ink( blue )
3. print( "Hello!" )
4. update()
```

Every time we want to do this, we'll have to use these same 4 lines again and again. Unless we create a **function** to do it!

```
1. function fuzePrint()
2.   textSize( 50 )
3.   print( "Hello!" )
```

```
4.     update()
5.     return void
```

Once we've written the above section of code, we can now simply use `fuzePrint()` and we'll get the same result each time.

But we could upgrade this **function**. Let's say we wanted to print any text we like, at any size, with any colour!

By **passing** some **variables** to our **function**, we can do this very easily:

```
1. function fuzePrint( text, size, col )
2.     textSize( size )
3.     ink( col )
4.     print( text )
5.     update()
6.     return void
```

Now our `fuzePrint()` **function** has three **arguments** (pieces of information in the brackets). In our code, we could now type:

```
1. fuzePrint( "Hello!", 50, blue )
```

When we use this **function**, the information in the brackets is **passed** to the `fuzePrint()` **function** we created. "Hello!" is stored as a **variable** called `text` and used in the `print()` line. The number 50 is stored as a **variable** called `size` and used in the `textSize()` line, and the colour is stored as a **variable** called `col` and used in the `ink()` line.

We could now use our new `fuzePrint()` **function** again and again in our program, and save ourselves a lot of hassle!

Return Void

You might have noticed that strange looking line in the `fuzePrint()` **function** we just created.

```
return void
```

All **functions** *must* return something. Even if that something is technically nothing!

At the end of a **function** you create, you must specify what you would like it to return.

If your **function** does not need to return anything, simply write the `return void` line.

Sometimes we want a **function** to return something calculated in the **function** itself. For instance, here's a custom **function** which converts metres into centimetres:

```
1. function metre2cm( number )
2.     return number * 100
```

Simply put the name of the variable you'd like to return at the end of the **function**! You can also perform operations here, just like the example above.

Try writing some **functions** of your own.

See you in the next tutorial!

Functions and Keywords used in this tutorial

`box()`, `circle()`, `clear()`, `controls()`, `for`, `gWidth()`, `gHeight()`, `loop`, `random()`, `repeat`, `update()`

TUTORIALS

Tutorial 9: And, Or, Not

In this tutorial, we'll dive a little further into **if statements** and what can be done with them.

We know that if we want to check something, we can use an **if statement**. For example:

```
1. dave = true
2.
3. if dave then
4.     print( "Hurray!" )
5. endif
```

Nice and simple to start with. We have a **variable** called `dave` which we've set to `true`.

Our **if statement** on line 3 checks to see if the `dave` **variable** is `true`. If it is, we print "Hurray!". How fantastic.

Let's introduce some more complexity. It's time to talk about two words: **and** and **or**

These words are called **operators**. We can use these operations to check multiple things together in our **if statement**.

Check it out.

```
1. dave = true
2. kat = true
3.
4. if dave and kat then
5.     print( "Hurray!" )
6. else
7.     print( "Oh no..." )
8. endif
```

First of all, we've introduced a new **variable** into the mix. It's `kat`!

Now our **if statement** checks two things. We check if `dave` is `true`, and if `kat` is `true`. This **if statement** will return either `true` or `false`, and will only be `true` if *both* our variables are `true`. If we change either of the **variables** to `false`, our **if statement** will be `false` and we'll print "Oh no...".

The **and** operation works very similar to the way we use it to speak, it combines things.

Or is a little different, but again it works very similarly to the way we use it to speak. **Or** checks if *either* of the conditions are true.

See if you can predict the difference with the code below:

```
1. dave = true
2. kat = true
3.
```

```
4. if dave or kat then
5.     print( "Hurray" )
6. else
7.     print( "Oh no..." )
8. endif
```

If we run this code, we'll get "Hurray", because both the `dave` and `kat` **variables** are `true`, and the **if statement** checks if *either* are `true`. Even if we change one of them to `false`, we'll still get "Hurray".

For our **if statement** to give us "Oh no...", we would have to make *both* the `dave` and `kat` **variables** false.

Summary

Before we get into some more practical examples of using **and** and **or**, take a look below for a summary.

Try to think of the **and** and **or** operations as things which give you a true or false answer.

And

Take a look at these examples below. No need to type them into your editor as they are unfinished.

The **if statement** below will give us `true`

```
1. dave = true
2. kat = true
3.
4. if dave and kat then
```

Here, our **if statement** gives us `false`:

```
1. dave = false
2. kat = true
3.
4. if dave and kat then
```

Here, our **if statement** gives us `false`:

```
1. dave = false
2. kat = false
3.
4. if dave and kat then
```

Or

The **if statement** below will give us `true`:

```
1. dave = true
2. kat = true
3.
4. if dave or kat then
```

Here, our **if statement** gives us `true`:

```
1. dave = false
2. kat = true
3.
4. if dave or kat then
```

Here, our **if statement** gives us **false**:

```
1. dave = false
2. kat = false
3.
4. if dave or kat then
```

Not

There is also another star player here... But for some reason they're **not** here...

That's right, it's **not**!

This is getting confusing...

not does exactly what you'd think it does. It checks if something is **not** true. We can use the **!** symbol to use **not**, but it's also fine to use the word!

Let's see an example of **not** in our Dave and Kat program.

```
1. dave = false
2. kat = false
3.
4. if not dave == true or not kat == true then
```

This **if statement** will give us true, because we are checking to see if **either** dave or kat is false (not true).

To write this using the **!** symbol, the syntax is a little different, spot the difference:

```
1. dave = false
2. kat = false
3.
4. if dave != true or kat != true then
```

See how we must put the **!** just before the **=** sign? Think of this as saying "not equal to".

We can also write this another way, even more efficiently!

```
1. dave = false
2. kat = false
3.
4. if !dave or !kat then
```

Either of these ways of writing your code is perfectly fine, it's up to you to choose which you prefer!

Using And, Or and Not

Okay! Let's see a couple of examples where this might be really useful for a game project.

When we want to check the buttons on the Joy-Con's, we use **if statements** to check which buttons are being pressed. Here, **and** and **or** can help very much.

Let's say we want something to happen **only if** we are moving and pressing the A button.

```
1. loop
2.   clear()
3.
4.   j = controls( 0 )
5.
6.   if j.lx != 0 and j.a then
7.     print( "Hurray!" )
8.   endif
9.
10.  update()
11. repeat
```

As you can see we have a simple **loop**. We clear the screen at the very start, and update the screen at the very end.

Line 4 uses the `controls()` **function** which tells us the current state of all the controls. The result is stored in a **variable** called `j`.

Our **if statement** checks if the **Joy-Con** left control stick (`lx`) is being pushed in any direction `j.lx != 0`. If the control stick value is **0** it's totally still and right in the middle!

We also check if the A button is being pressed with `joy.a = true`.

Because of the **and** between these two checks, *both* must be true before our **if statement** becomes true.

Let's see an example using **or**.

```
1. loop
2.   clear()
3.
4.   joy = controls( 0 )
5.
6.   if joy.zl or joy.zr then
7.     print( "Hurray!" )
8.   endif
9.
10.  update()
11. repeat
```

This time we're checking for different buttons. We are now checking if *either* the left shoulder button `joy.zl` or the right shoulder button `joy.zr` become pressed.

This way, we could make the same thing happen for two different button presses.

Finally, let's see an example using **not**.

This time, we'll make it so that "Hurray" appears only when the button **isn't** being pressed.

```
1. loop
2.   clear()
3.
4.   j = controls( 0 )
5.
6.   if !j.z1 or !j.zr then
7.     print( "Hurray!" )
8.   endif
9.
10.  update()
11. repeat
```

We've only added the **!** symbols to our code, and now the result is totally different!

We are now checking if the shoulder buttons are false. While they are false, "Yahoo!" appears on the screen, but as soon as one becomes pressed and becomes true, our **if statement** is false and we do not see "Hurray!".

Re-cap

and, **or** and **not** are called **operators**. They perform an operation.

We mainly use them in **if statements** to make them capable of checking more complex things.

They are particularly useful when using the **controls(0) function**.

Imagine we want a character in a game to be able to perform a jumping attack. We will need **and** in this situation because we are checking if the character is jumping **and** if the attack button is pressed.

See you in the next tutorial!

Functions and Keywords used in this tutorial

and, **clear()**, **controls()**, **else**, **endif**, **if**, **loop**, **not**, **or**, **print()**, **repeat**, **then**, **update()**

TUTORIALS

Tutorial 10: Variables Extended

In this tutorial we'll be covering some of the more advanced properties of **variables**, what words like "Global" and "Local" mean and a concept called **scope**.

So far we know that we can use a **variable** to store a piece of information. This can be anything at all: we can store a number in a **variable**, we can store a piece of text (a string), we can store the result of a **function**, we can even store multiple pieces of information in a type of **variable** called an **array**.

This is all wonderful information. However, we're missing something a little important for more advanced programs.

Every **variable** in a program has something called **scope**. There are two levels of **scope**, **global** and **local**. The **scope** of a **variable** determines which parts of a program can access it.

Global variables can be accessed from **anywhere** in the program. **Local variables** can only be accessed from areas of the program with the **same scope**.

Confused?

Let's take a look at an example:

```
1. a = 10
2.
3. loop
4.     clear()
5.     print( a + modifier( a ) )
6.     update()
7. repeat
8.
9. function modifier( number )
10.     number += a
11. return number
```

Here we have a very simple program which adds two numbers together.

The **variable** at the top of the program, **a** is **global**. Once we've defined it at the top of our program, we can use it anywhere we like. In the example we have demonstrated this by using the **a variable** both in our main **loop** and in our user-defined **function** called `modifier()`.

The second **variable** in our program, **number** is **local** to the `modifier()` **function**. Because it was defined in our custom **function**, we can **only** access it within that **function**.

Data Types

When storing a piece of data in a **variable** in **FUZE⁴ Nintendo Switch**, most of the time you do not need to specify what type of data that will be. For example, let's say we want to store a whole number:

```
1. a = 10
```

There we go! Nothing more required. Let's say we want to store a **string** instead:

```
1. a = "Dave"
```

Done! What about storing an array inside our **variable**?

```
1. a = [ 0, 1, 2, 3 ]
```

Simple as that! How about storing a **structure**?

```
1. a = [ .name = "Dave", .age = 27, .interests = "Music" ]
```

"What about a **vector**!?" We hear you scream. If you don't know what a **vector** is, fear not! We'll be covering that in the upcoming tutorials.

```
1. a = { 0, 0, 0, 0 }
```

Simple as can be. We do not need to define what **type** of data will be in a **variable** to store it.

However, in **FUZE⁴ Nintendo Switch**, there is one particular time where we **must** define the **type** of data.

There is a different way to create a structure, by creating something called a "structure type":

```
1. struct person_type
2.     string name
3.     int    age
4.     array  interests[3]
5. endstruct
6.
7. person_type person[10]
```

The example above defines a **structure type** called `person_type`, which describes three properties.

Line 7 creates an array of structures called `person`. It has 10 elements, and each element now contains a structure with three properties, a string **variable** called `name`, an int **variable** called `age` and an array **variable** called `interests` with 3 elements.

When defining a structure using a structure type like the example above, we **must** define what type of data each property will be. If we do not do this, we will get an error.

This way of creating structures can be very useful for larger programs, where you might need to use a certain type of structure in multiple places in your code.

Functions and Keywords Used in this Tutorial

`array`, `clear()`, `endStruct`, `function`, `int`, `loop`, `print()`, `repeat`, `return`, `string`, `struct`, `update()`

TUTORIALS

Tutorial 11: Loading and Drawing an Image

Hello there! Great to see you again.

In this tutorial we'll be going over the basics of loading images from the **FUZE⁴ Nintendo Switch** media browser and drawing them to the screen using the built-in **functions**.

We'll also do some diving into what can be done to scale, warp and rotate an image for some customisation!

When using an image from the **FUZE⁴ Nintendo Switch** media, we must first **load** the image and assign it to a **variable**.

To do this, we use the **loadImage** function:

```
1. img = loadImage()
```

In **FUZE⁴ Nintendo Switch**, when you type the **loadImage()** function and put the cursor in the brackets, you'll notice a glowing outline around the "media" key on the on-screen keyboard. Click this to be taken to the media browser.

In the media browser you can select an asset you'd like to use and FUZE will let you paste the filename, in speech marks, into your code.

We've chosen an image for now from the artist Selavi.

```
1. img = loadImage( "Selavi Games/JapaneseSetting", false )
```

Notice the **false** in the **loadImage()** arguments. This argument tells FUZE whether or not to apply a filter to the image. We do not want a filter here, so we put a **false**.

Alright we now have the image file saved into a **variable** called **img** (short for image!). We can now use this **variable** in many other **functions**.

Let's start by simply drawing our image to the screen. We'll need the standard **clear()** and **update()** loop we all know and love!

```
1. img = loadImage( "Selavi Games/JapaneseSetting", false )
2.
3. loop
4.   clear()
5.
6.   update()
7. repeat
```

All we need now is to add the **drawImage()** function!

```
1. img = loadImage( "Selavi Games/JapaneseSetting", false )
2.
```

```
3. loop
4.     clear()
5.
6.     drawImage( img, 0, 0, 1 )
7.
8.     update()
9. repeat
```

Run the program to see the image displayed very nicely on screen. Lovely!

drawImage()

Let's take a look at that **function** in more detail and see exactly what's going on.

The first argument in the brackets is the **variable** we've stored the image file in. Easy!

The next two arguments are the **x** and **y** screen positions you'd like to draw the image to. **0, 0** is the very top left corner of the screen.

The last argument is the **scale multiplier**. This number is used to multiply the dimensions of the image when we draw it on screen.

We've used a **1** here to put the image on screen in its actual size. Let's change that to something smaller:

```
1. img = loadImage( "Selavi Games/JapaneseSetting", false )
2.
3. loop
4.     clear()
5.
6.     drawImage( img, 0, 0, 0.1 )
7.
8.     update()
9. repeat
```

With a scale of **0.1** we are making the image ten times smaller.

Let's get a little fancier and use the more complicated `drawImage()` **function** to manipulate the image.

drawImageEx

Rather than using the regular `drawImage` **function**, by using `drawImageEx` we get access to far more control over the image. There are a few more arguments needed for this **function**, take a look below:

```
1. img = loadImage( "Selavi Games/JapaneseSetting", false )
2. xPos = gwidth() / 2
3. yPos = gheight() / 2
4. rotation = 0
5. wScale = 0.7
6. hScale = 0.7
7. r = 1
8. g = 1
9. b = 1
```

```
10. a = 1
11. originX = 0
12. originY = 0
13.
14. loop
15.     clear()
16.
17.     drawImageEx( img, xPos, yPos, rotation, wScale, hScale, r, g, b, a, originX, originY )
18.
19.     update()
20. repeat
```

Wow, would you take a look at all those arguments! By using `drawImageEx()` we can control almost everything about the image.

To make the **function** a little clearer, we've used **variables** at the top of the program to show exactly what each argument is doing, laid out in the same order they appear in the **function** itself.

The first three arguments are ones we're already familiar with. As we know, `img` is the **variable** which stores the image file. The next two arguments, `xPos` and `yPos`, are the **x** and **y** positions of where we want the image to be drawn.

Rotation

The next argument is where things get a little more interesting! This argument controls the rotation angle of the image. At the moment, the `rotation` **variable** is a 0. If we were to make this 180, the image would flip upside-down.

Scale

The next two arguments are the width and height scale of the image. With both of these arguments as 1, the image will be drawn at its regular scale. Changing these numbers will give the effect of stretching the image. For example, changing the `hScale` **variable** to 0.5 will make the image twice as wide as it is tall.

RGBA

The next arguments are very cool indeed! We have separate values here for the red, blue, green and alpha elements of the image. Try changing the `r`, `g` and `b` **variables** to 0, 0 and 1 for a red tinted image. Try any numbers you like between 0 and 1 to make your own tint!

Origin

This one is a little strange to get the old head around. The origin point of an image is where the image is drawn **from**. By default, the origin is set to 0 on both the **x** and **y** axes. Remember, when we talk about the **screen**, (0, 0) refers to the top left corner. However, when it comes to images (0, 0) is the **middle** of the image. Take a look at the image below:



With an image origin of $(0, 0)$ and an x and y position of $(gwidth() / 2, gheight() / 2)$, our image will be printed right in the middle of the screen.

What happens if we change the origin, but keep the x and y positions the same?



If we set the x origin to be minus half the width of the image, but keep the x and y **location** of the image the same, our origin is now on the very leftmost side of the image, and the image will be drawn from that point on the screen.

Manipulating an Image in a Program

Let's adjust the code a little to make interesting use of these features. First, we'll rotate the image during the loop:

```

1. img = loadImage( "Selavi Games/JapaneseSetting", false )
2. xPos = gwidth() / 2
3. yPos = gheight() / 2
4. rotation = 0
5. wScale = 0.7
6. hScale = 0.7
7. r = 1
8. g = 1
9. b = 1
10. a = 1
11. originX = 0

```

```

12. originY = 0
13.
14. loop
15.   clear()
16.
17.   drawImageEx( img, xPos, yPos, rotation, wScale, hScale, r, g, b, a, originX, originY )
18.
19.   rotation += 1
20.
21.   update()
22. repeat

```

Run the program to see the image spinning on screen.

Notice that the image is rotating **around the origin**. If we were to change the origin, we also change the point of rotation. Let's give that a try:

```

1. img = loadImage( "Selavi Games/JapaneseSetting", false )
2. xPos = gwidth() / 2
3. yPos = gheight() / 2
4. rotation = 0
5. wScale = 0.7
6. hScale = 0.7
7. r = 1
8. g = 1
9. b = 1
10. a = 1
11. originX = 800
12. originY = 300
13.
14. loop
15.   clear()
16.
17.   drawImageEx( img, xPos, yPos, rotation, wScale, hScale, r, g, b, a, originX, originY )
18.
19.   rotation += 1
20.
21.   update()
22. repeat

```

All we have done is changed the value of the `originX` and `originY` **variables** to 800 and 300 respectively. Run the program to see that our rotation looks very different!

Alright. Enough spinning around. By manipulating the red, green, blue and alpha values we can control the lighting of the image in a very convincing way:

```

1. img = loadImage( "Selavi Games/JapaneseSetting", false )
2. xPos = gwidth() / 2
3. yPos = gheight() / 2
4. rotation = 0
5. wScale = 0.7
6. hScale = 0.7
7. r = 1
8. g = 1
9. b = 1
10. a = 1
11. originX = 0
12. originY = 0
13.
14. loop
15.   clear()
16.   j = controls( 0 )

```



```

17.
18.     a += j.ly / 50
19.     g += j.ly / 50
20.     r += j.ly / 50
21.
22.     if r < 0 then r = 0 endif
23.     if g < 0 then g = 0 endif
18.
19.     drawImageEx( img, xPos, yPos, rotation, wScale, hScale, r, g, b, a, originX, originY )
20.
21.     update()
22. repeat

```

Run the program and move the left control stick up and down to control the lighting. Moving the control stick down makes the image darker and more blue, whereas moving the control stick upward will make the image brighter. We've also changed our origin back to (0, 0) in this example.

The new lines added are from 16 to 23. First, we call the `controls()` function to access the Joy-Con controllers. We are using a **variable** called `j` to store the state of the controls.

On lines 18 to 20 we use the left control stick to change the values of the `a`, `g` and `b` **variables**. The `a` **variable** stores the **alpha** value (the transparency of the image). If we **reduce** this value we make the image darker and increasing it makes the image lighter. Since the control stick value can be either positive or negative we simply need to add the result to the **variable**.

We do the same with the `r` and `g` **variables** to control the amount of red and green in our image. As we move the control stick down the red and green leaves the image, leaving us with just blue. This happens as the transparency increases, allowing us to see more of the black screen behind the image. This gives us a lovely dark blue effect.

This sort of technique would be perfect for a game with lots of dialogue between characters.

Why not try changing the background image and creating different lighting effects?

See you in the next tutorial!

Functions and Keywords used in this tutorial

`clear()`, `drawImage()`, `drawImageEx()`, `else`, `endif`, `getWidth()`, `getHeight()`, `if`, `loadImage()`, `loop`, `repeat`, `then`, `update()`

TUTORIALS

Tutorial 12: Structures

Welcome back! Get your thinking hats on for this tutorial - we'll be looking at a slightly more advanced concept.

This project will be covering **structures**. What a **structure** is, how we can use one in our programs, and a couple of examples.

Structures are incredibly useful tools for programming, and they make reading code much easier.

What is a Structure?

A **structure** is a lot like an **array**. We use it to store information.

Where an array uses a *number* to access a piece of information, a **structure** uses a *name*.

We might use one to store all the information for a player character in a side-scrolling style game. A player character might have a position on screen, a health value, a speed, an attack and defense value, etc. All of this information could be stored in a **structure** which would make it very easy and convenient to access.

Let's take a step back, and create a very simple **structure** which stores some information about a person. Copy the code below into the **FUZE⁴ Nintendo Switch** code editor. Of course, feel free to change the information!

```
1. person = [  
2.     .name = "Luke ",  
3.     .likes = "Chocolate",  
4.     .dislikes = "Vegetables",  
5.     .skills = "Coding"  
6. ]
```

There we have it. This program simply sets up a structure we can use. We could access any of this information with something like: `print(person.name)` or `print(person.dislikes)`. Simple enough!

Formatting

Let's quickly address the strange way this code is laid out. To somebody new to coding, this may look quite strange.

The important parts are the square brackets and commas. First, we name the **structure**. We have called ours `person`. Then, we open square brackets to begin the **structure**, and define the properties we would like it to have. Each of these properties must be separated by a comma. Finally, we close the square brackets to finish the **structure**.

It's very important to realise that this is *not* the only way to lay out a **structure**. You might want to do it like this:

```
1. person = [.name = "Luke ",
2.         .likes = "Chocolate",
3.         .dislikes = "Vegetables",
4.         .skills = "Coding"]
```

Which would work *exactly the same*. You could even write it all on one line, like this:

```
1. person = [.name = "Luke ", .likes = "Chocolate", .dislikes = "Vegetables", .skills = "Coding"]
```

As mentioned before, the important parts are the square brackets and commas. It's *up to you* how you want to lay out your code in **FUZE⁴ Nintendo Switch**.

Creating an Array of Structures

This is all well and good, but what if you want to store the information for multiple people?

Here we might use an array of structures. This is simply an array, where each element of the array is a structure itself. Take a look below, we've added someone new to the program:

```
1. person = [
2.     [
3.         .name = "Luke",
4.         .likes = "Chocolate",
5.         .dislikes = "Vegetables",
6.         .skills = "Coding"
7.     ],
8.     [
9.         .name = "Colin",
10.        .likes = "Trains",
11.        .dislikes = "Brussels Sprouts",
12.        .skills = "Dancing"
13.    ]
14. ]
```

Now we have something a little more complicated. Try not to be put off by the square brackets!

Our first line no longer creates a structure called `person`. Now it creates an array called `person` with two elements. Each of these elements is a structure, just like the single structure before.

The first structure is stored in `person[0]` and the second in `person[1]`.

Let's say we wanted to print Colin's likes. We would do that with `print(person[1].likes)`. Let's say we wanted to print Luke's name. We could do that with `print(person[0].name)`.

Using an array of structures gives us a convenient and powerful method of storing information to use in our programs. With a combination of **structures**, **arrays** and **for loops**, you can achieve some incredible things.

Using a Structure in a simple Game

For the main part of this tutorial, we'll use an **array** of **structures** to make a simple racing game, where 3 shapes will race across the screen. It's up to us to guess who might win!

We will use three shapes, a triangle, a circle and a pentagon. Each one needs a name, an **x** and **y** position, a number of sides (more on this later), a colour and a speed to move across the screen.

Our speed will be a randomly chosen number out of 50. This way, every time we run the program we'll get a different outcome!

Let's build the **structure** first. This will be quite a lot of code, so it's recommended to *copy* and *paste* this into the **FUZE⁴ Nintendo Switch** code editor.

```
1. shapes = [  
2.   [  
3.     .name = "Triangle",  
4.     .x    = 0,  
5.     .y    = gheight() / 2 - gheight() / 3,  
6.     .sides = 3,  
7.     .col  = red,  
8.     .speed = random( 50 )  
9.   ],  
10.  [  
11.    .name = "Circle",  
12.    .x    = 0,  
13.    .y    = gheight() / 2,  
14.    .sides = 32,  
15.    .col  = green,  
16.    .speed = random( 50 )  
17.  ],  
18.  [  
19.    .name = "Pentagon",  
20.    .x    = 0,  
21.    .y    = gheight() / 2 + gheight() / 3,  
22.    .sides = 5,  
23.    .col  = blue,  
24.    .speed = random( 50 )  
25.  ]  
26. ]
```

Phew! There we go. Take a good look at this **array** of **structures**. We created an **array** called **shapes** with 3 elements. Each of these elements is a **structure** with 6 **properties**.

Just like before in our person examples, we could access any of these with a statement like `print(shapes[1].name)`, for example.

Let's now add the code which uses this information. We'll need a loop to animate the screen, and a for loop to count over our array of shapes.

```
28. loop  
29.   clear()  
30.   for i = 0 to len( shapes ) loop  
31.     circle( shapes[i].x, shapes[i].y, 100, shapes[i].sides, shapes[i].col, false )  
32.     repeat  
33.       update()  
34.     repeat
```

Here's where the usefulness of our **array of structures** really shines. Lines 30 to 32 create a **for loop** which repeats 3 times. We create a **variable** called `i` which increases from 0 to 1, to 2.

Notice we are using the `len()` **function** to give us the length of an array. Our array is 3 elements, so this gives us a 3.

This `i` **variable** is used as an index into the `shapes` **array** to draw a circle on the screen for each shape, at the `x` and `y` positions stored in the **structure**. Because of the way the `circle()` **function** works in **FUZE4 Nintendo Switch**, we can just change the number of sides and create a different shape! This is why we store the number of sides as a property of the **structure**.

Once the **for loop** is complete, we `update()` the screen, and repeat the **loop**.

Now let's move them across the screen!

By adding one more line of code into the **for loop**, we can move each shape:

```
28. loop
29.     clear()
30.     for i = 0 to len( shapes ) loop
31.         shapes[i].x += shapes[i].speed
32.         circle( shapes[i].x, shapes[i].y, 100, shapes[i].sides, shapes[i].col, false )
33.     repeat
34.     update()
35. repeat
```

Line 31 now increases the `x` position of each shape by that shape's speed. Run the program to see all the shapes move at different speeds!

All that's left is to create a victory screen when a shape wins the race!

This is the perfect time to create our own **function**! We can pass the name and colour of the winning shape to our **function** and use that information in a victory screen. Enter the code below at the *bottom* of the program, after the `repeat` on line 35.

```
37. function victoryScreen( name, col )
38.     text = name + " Wins!"
39.     tsize = 100
40.     textSize( tSize )
41.     loop
42.         clear()
43.         drawText( gwidth() / 2 - textWidth( text ) / 2, gheight() / 2, tSize, col, text )
44.         update()
45.     repeat
46. return void
```

Here we have our **user-defined function**. This is a section of code which will run when we *call* the function.

The purpose of this **function** is to print the name of the winning shape in the correct colour on the screen. For that, we need two pieces of information. The name and colour of the winning shape!

In the first line, we name the **function** as `victoryScreen()` and give it two **arguments**. The first argument is stored in a **variable** called `name` and the second is stored in a **variable** called `col`. We can now use these **variables** in our **function**.

First we define a new **variable** called `text`. This will store the entire line of text we want to print. We take the `name` **variable** from our **function's** arguments and add it to the text "Wins!". This `text` **variable** will be very useful when making our text appear in the right place on screen.

Next up we define a **variable** to store the size of the text. Doing this allows us to get the position of the text on screen perfect.

On line 40 we use the `textSize()` **function** to set the size of the text with our `tSize` **variable**.

Now on to the main part of our victory screen, the **loop**. We need a **loop** because we want our screen to display the text until the program is stopped. Of course, we'll need a `clear()` and an `update()` because we want to display something on screen.

The main instruction in our **loop** is the `drawText()` **function** which we use to display text on screen with more details than a simple `print()`. We can position the text by pixels, set a size for the text and even a colour.

As you can see, the `x` position of the `drawText()` **function** looks a little strange:

```
43.         drawText( gwidth() / 2 - textWidth( text ) / 2
```

We want the text to be exactly in the centre of the screen, no matter how long the text is. If we positioned our text at a fixed set of coordinates, we would get very different results if the console was in TV or handheld mode. We would also get different results depending on which shape won, because we get a longer or shorter victory message depending on the winning shape.

Due to these reasons, we must use adjust the position based on the length of our text. This is where our `text` **variable** comes in!

We use the `textWidth()` **function** to give us the width of a piece of text in pixels. We can then divide this number by two to give us half the length. If we subtract this from the middle of the screen (`gwidth() / 2`) we will always have our text perfectly in the middle of the screen!

It seems like a lot of trouble, but this is a very useful technique for positioning text! Be sure to remember it for your other projects!

Alright. That's our **user-defined function** completed. We end the **function** with `return void` because we do not need to return anything here.

All that's left is to actually *call* the **function** in the main **loop**!

Adjust the main **loop** to look like the one below. We're just adding a simple **if statement** to check if a shape has reached the finish line.

```
28. loop
29.     clear()
30.     for i = 0 to len( shapes ) loop
31.         shapes[i].x += shapes[i].speed
32.         if shapes[i].x > gwidth() then
33.             victoryScreen( shapes[i].name, shapes[i].col )
34.         endif
35.         circle( shapes[i].x, shapes[i].y, 100, shapes[i].sides, shapes[i].col, false )
36.     repeat
```

```
37.     update()  
38. repeat
```

Lines 32 to 34 contain our **if statement**. We simply check if the current shape in the **for loop** has reached the edge of the screen. If it has, we *call* the `victoryScreen()` **function** and *pass* it the current shape's name and colour as **arguments**.

That's it! Have fun with this one! Perhaps you could add a couple more shapes to the race? Because of how our main game is written, to achieve this you would only need to add more shapes to the **array**!

Recap

A **structure** is a tool for storing data. It is very similar to an **array**, except each element has a *name* rather than a *number*. You can freely mix and match these techniques for the task at hand. You can store **arrays** within **structures**, and store **structures** within **arrays**!

Try setting up your own and accessing the values to really get your head around how they work!

As always, well done for making it this far and see you in the next tutorial!

Functions and Keywords used in this tutorial

`else`, `endif`, `circle()`, `clear()`, `drawText()`, `for`, `if`, `loop`, `repeat`, `textSize()`, `then`, `update()`

TUTORIALS

Tutorial 13: Making Music

Musical? If not, you soon will be! In this tutorial we will be using the `playNote()` **function** to create our very own music.

We have taken a very brief look at `playNote()` before, during the **for loops** tutorial project. In this project we will going into much more detail.

playNote()

Let's take a quick look at the **function** and break down the arguments:

```
playNote( channel, waveType, frequency, volume, speed, pan )
```

The `channel` argument tells FUZE which audio channel to play our desired note on. There are 16 channels to use, giving us up to 16 sounds playing at the same time, these could be single notes, music tracks or sound effects.

The `frequency` argument is the **frequency** of the note we want to play. Another word for frequency is **pitch**. The higher the frequency, the higher the pith of the note.

Frequency is measured in something called Hertz (hz). This means **cycles per second**. If your eardrum vibrates at 440hz (a rate of 440 times a second), you will hear the note **A!** Humans can hear sounds from around 20hz to 20000hz. Anything outside of this range will not be audible to us.

The `waveType` argument is the **type of waveform** we want to use to play the note. There are 5 different types to choose from in FUZE each with a number: Square (0), sawtooth (1), triangle (2), sine (3) and noise (4). Each of these waveforms has a very different sound to our ears. Check them all out!

The `volume` argument is simply the loudness of the note. This value should be between 0 and 1, but can be pushed higher if desired.

The `speed` argument describes the envelope shape of the note. This one's a little tricky to imagine. A low number in the `speed` argument will result in a longer note duration. A higher number will result in a shorter note duration.

Finally, the `pan` argument is the stereo position of the sound. Your Nintendo Switch console has two speaker channels, a left and a right. This number is the position of the sound between these channels. A value of 0.5 is right in the centre. A number closer to 0 will result in the sound moving to the left whereas a number closer to 1 will result in the sound moving further to the right.

A Quick Example

Let's plug some values into this **function** to make a sound play:


```

1. playNote( 0, 3, 432, 1, 0.5, 0.5 )
2. loop
3.     clear()
4.     update()
5. repeat

```

Here we have a small program which plays a note then enters a loop. Notice that the `playNote()` line is not **in the loop** because we only want it to happen once. Our note will ring out for a few seconds before fading out.

Try changing these values to get different results.

note2Freq()

In FUZE we have a very clever **function** called `note2Freq()`. When making music we don't tend to think about things in terms of **frequency**, but rather in terms of the note name. For example, the note A above middle C (also known as A4) is found at the frequency of 440hz. If we were telling a musician how to play a melody, we wouldn't list the frequencies!

Some time ago, a very clever chap called Dave Smith invented a way of sending data to electronic instruments to tell them which notes to play. This is called MIDI (Musical Instrument Digital Interface). In MIDI, there are 128 notes to choose from, each with a number. The note A4 we mentioned earlier is actually the number 69.

`note2Freq()` **receives** one of these MIDI note numbers and converts it into the correct **frequency** number. This allows us to do some very helpful things when making music!

Start an empty project before moving on to this next part!

Creating an Array of Notes

If we want to use the names of notes to write music, we'll need to store the MIDI note numbers into an array. We can use a structure to do this, giving each piece of data a helpful name:

```

1. n = [
2.     .c = 60,
3.     .cs = 61,
4.     .d = 62,
5.     .ds = 63,
6.     .e = 64,
7.     .f = 65,
8.     .fs = 66,
9.     .g = 67,
10.    .gs = 68,
11.    .a = 69,
12.    .as = 70,
13.    .b = 71
14. ]

```

There we are. Now we have each note we would want to play stored in the properties of a structure called `n` for **note**. We can now use something like `n.d` to get the note D for example.

The notes with a letter “s” after them are the in-between notes called “sharps”. We could also call these “flats”, but their note would have to change. C sharp is the same pitch as D flat, but for the sake of simplicity we’ve stuck with just sharps.

So now we have a 12 note scale with all the in-between notes. This is called the **chromatic** scale and with it we can compose pretty much anything we like!

Before we compose a melody, we’ll need a couple more **variables**. Notes have not only a pitch, but a length too. There are specific names for the length values of notes in music terms, we’ll be using 4 of them: **Semiquavers, quavers, crotchets** and **minims**:

```
16. semiquaver = 0.25
17. quaver = 0.5
18. crotchet = 1
19. minim = 2
```

Here we are creating some **variables** to store the length values of each note type we will use. These are relative to one “beat” - a crotchet is one beat, a quaver is half the length of a crotchet (0.5), and a semiquaver is half the length of a quaver (0.25).

Now let’s use these variables to compose a small melody, storing the data for each note we want to play in an **array**:

```
21. melody = [
22.     [ .note = n.d, .spd = 40, .l = quaver ],
23.     [ .note = n.e, .spd = 40, .l = quaver ],
24.     [ .note = n.f, .spd = 40, .l = quaver ],
25.     [ .note = n.g, .spd = 40, .l = quaver ],
26.     [ .note = n.e, .spd = 20, .l = crotchet ],
27.     [ .note = n.c, .spd = 40, .l = quaver ],
28.     [ .note = n.d, .spd = 20, .l = minim * 4 ]
29. ]
```

Here’s our melody! As you can see, we create an **array** called `melody` which stores 7 **structures**. Each **structure** is a note in our melody, giving us a total of 7 notes. Each note has a `.note` value which stores the reference into our notes **structure**, a `.spd` value which will store the speed value for the `playNote()` **function** and finally a `.l` value which stores the note length.

Using this format you could write a piece of music very easily! Even if it is quite a bit of typing...

Before we go ahead with playing sound, we must first convert these note length values into real time values. The note length value tells us how long the note should be, not **when** in time the note should start. There is a clever way of doing this using a **for loop**:

```
31. endTime = 0
32. counter = 0
33.
34. for i = 0 to len( melody ) loop
35.     temp = melody[i].l
36.     melody[i].l = counter
37.     counter += temp
38. repeat
```

```
39.  
40. endTime = counter
```

First we create two **variables**. `endTime` will eventually store the ending time of the whole melody. `counter` will be used to keep track of the note times during the **for loop**.

Our **for loop** counts from 0 to the length of the `melody array` using an `i variable`. For each note in the melody, we store that note's length value in a **variable** called `temp`. We then set that note's length value to be equal to the `counter variable`. Since `counter` begins at 0, the first note length value becomes 0. Perfect! We want our first note to happen instantly.

We then increase the `counter variable` by the number held in the `temp variable`, giving us a new point in time which the next note should begin at. This process happens for each note in the melody, converting each note's length value into a correct start time value.

When the **for loop** is completed, our `counter variable` now stores the correct end time for the melody, so we assign the value of `counter` to the `endTime variable`.

We just need a couple more **variables** before starting the main **loop** in which the melody will play.

```
42. noteCount = 0  
43. timer = 0
```

These two **variables** are important. `noteCount` will be used as the index into the `melody array` to play each note in sequence. `timer` is a **variable** which will keep track of the amount of time which has passed. We will use the `timer variable` as a trigger to tell FUZE when to move on to the next note in the `melody array`.

Playing the Melody

Alright, let's put all of this to use:

```
45. loop  
46.   clear()  
47.  
48.   if noteCount < len( melody ) then  
49.     if timer >= melody[noteCount].l then  
50.       note = note2Freq( melody[noteCount].note )  
51.       speed = melody[noteCount].spd  
52.       playNote( 0, 3, note, 1, speed, 0.5 )  
53.       noteCount += 1  
54.     endif  
55.   endif  
56.  
57.   timer += ( 120 / 60 ) / 60  
58.  
59.   if timer >= endTime then  
60.     stopChannel( 0 )  
61.   endif  
62.
```

```
63.     update()
64. repeat
```

Run the program once you've completed the code to hear the melody in all its glory. Jazzy!

In this **loop** we have a couple of clever **if statements**. Before we look at those, take a look at line 57:

```
57.     timer += ( 120 / 60 ) / 60
```

This line of code is the part responsible for keeping time. The speed of a piece of music is called the tempo. This is measured in beats per minute, or BPM. Our melody is being played at 120 beats per minute.

Our loop is cycling 60 times per second, with the `update()` **function** happening once each time.

To get the tempo right, we must figure out not how many beats per minute must take place, but how many beats **per frame**.

To achieve this, we take the beats per minute (120) and divide it by 60 to give us the number of beats per second (`120 / 60`). We then divide that by the 60 frames happening in one second to give us beats per frame.

All of this means that our timer is now increasing at a rate that will give us 120 beats per minute at 60 frames per second. Clever!

Now let's take a look at the **if statements**:

```
48.     if noteCount < len( melody ) then
49.         if timer >= melody[noteCount].l then
50.             note = note2Freq( melody[noteCount].note )
51.             speed = melody[noteCount].spd
52.             playNote( 0, 3, note, 1, speed, 0.5 )
53.             noteCount += 1
54.         endif
55.     endif
```

First we check if the `noteCount` **variable** is less than the length of the melody. Next, we check if the `timer` **variable** has reached the start time of the current note in the melody.

If both of these conditions are true, we set a couple of local **variables** to make our `playNote()` line much easier to read. The `note` **variable** stores the frequency value of the note, calculated using the `note2Freq()` **function** we talked about earlier.

The `speed` **variable** stores the `.spd` value from the `melody` **array**.

We then use the `playNote()` **function** to play the desired note using a nice soft sine wave on channel 0 with max volume and a central stereo position.

Before we finish the **if statement**, we increase the value of `noteCount` to ensure that we play the next note in the sequence.

That's it!

Try changing the melody, adding your own notes, changing the timings and changing the beats per minute. There's nothing to get wrong here! It's all groovy!

Actually, before you do, let's apply a little **reverb** to this melody to really make it sound sweeter:

```
44. setReverb( 0, 8000, 0.5 )
```

Just before the start of the **loop**, add the line above to your program. This **function** sets an amount of reverberation, or echo, to a channel.

The first number in the brackets is a 0 for the channel. Since our melody is played using channel 0, we must make this a 0 to hear any effect!

The next number is the amount of milliseconds before we hear an echo.

The last number is the multiplier applied to the volume of the echo over time. A low number hear will cause a fast volume reduction, whereas a higher number will result in a slower volume reduction. This number can be between 0 and 1.

Now run the program and we should hear quite a difference!

Go ahead and make your own tunes!

Making Your Melody Loop

The program we have written will only play the melody once and then stop all sound from the channel:

```
59.   if timer >= endTime then
60.       stopChannel( 0 )
61.   endif
```

With this **if statement** we check if the **timer variable** has reached the value stored in the **endTime variable**. If it has, we issue a `stopChannel()` command to cease all sound from the channel. If we wanted our melody to loop, we could change this to the following:

```
59.   if timer >= endTime then
60.       noteCount = 0
61.       timer = 0
62.   endif
```

Now when our **timer** reaches **endTime**, instead the **timer** and **noteCount** are reset to 0, starting the whole thing again.

For reference, here is the complete project below, make sure yours is working properly before using it in another project!

```
1. n = [
2.     .c = 60,
3.     .cs = 61,
4.     .d = 62,
5.     .ds = 63,
6.     .e = 64,
7.     .f = 65,
```

```

8.     .fs = 66,
9.     .g  = 67,
10.    .gs = 68,
11.    .a  = 69,
12.    .as = 70,
13.    .b  = 71
14. ]
15.
16. semiquaver = 0.25
17. quaver     = 0.5
18. crotchet   = 1
19. minim      = 2
20.
21. melody = [
22.     [ .note = n.d, .spd = 40, .l = quaver   ],
23.     [ .note = n.e, .spd = 40, .l = quaver   ],
24.     [ .note = n.f, .spd = 40, .l = quaver   ],
25.     [ .note = n.g, .spd = 40, .l = quaver   ],
26.     [ .note = n.e, .spd = 20, .l = crotchet ],
27.     [ .note = n.c, .spd = 40, .l = quaver   ],
28.     [ .note = n.d, .spd = 20, .l = minim * 4 ]
29. ]
30.
31. endTime = 0
32. counter = 0
33.
34. for i = 0 to len( melody ) loop
35.     temp = melody[i].l
36.     melody[i].l = counter
37.     counter += temp
38. repeat
39.
40. endTime = counter
41.
42. noteCount = 0
43. timer = 0
44. setReverb( 0, 8000, 0.5 )
45.
46. loop
47.     clear()
48.
49.     if noteCount < len( melody ) then
50.         if timer >= melody[noteCount].l then
51.             note = note2Freq( melody[noteCount].note )
52.             speed = melody[noteCount].spd
53.             playNote( 0, 3, note, 1, speed, 0.5 )
54.             noteCount += 1
55.         endif
56.     endif
57.
58.     timer += ( 120 / 60 ) / 60
59.

```

```

60.     if timer >= endTime then
61.         stopChannel( 0 )
62.     endif
63.
64.     update()
65. repeat

```

Playing Multiple Melodies Simultaneously

Aha! So you want to harmonise?

Well, this is very possible. As we mentioned earlier, we have 16 channels to select from. This means we can have 16 melodies all playing at the same time if we want, although that might get a little bit difficult to listen to!

Start a new project file before moving on. We're going to need the same note data from before, so copy the following section into your new project:

```

1. n = [
2.     .c = 60,
3.     .cs = 61,
4.     .d = 62,
5.     .ds = 63,
6.     .e = 64,
7.     .f = 65,
8.     .fs = 66,
9.     .g = 67,
10.    .gs = 68,
11.    .a = 69,
12.    .as = 70,
13.    .b = 71
14. ]
15.
16. semiquaver = 0.25
17. quaver = 0.5
18. crotchet = 1
19. minim = 2

```

To make two melodies occur at the same time we simply need to use duplicates of our **variables**. The best way to do this is to convert the **variables** like `noteCount` and `endTime` into **arrays**. Of course, our `melody` **array** will also need to be placed inside an **array**. Confusing? Worry not:

```

21. melody = [
22.     [
23.         [ .note = n.d, .spd = 40, .l = quaver ],
24.         [ .note = n.e, .spd = 40, .l = quaver ],
25.         [ .note = n.f, .spd = 40, .l = quaver ],
26.         [ .note = n.g, .spd = 40, .l = quaver ],
27.         [ .note = n.e, .spd = 20, .l = crotchet ],
28.         [ .note = n.c, .spd = 40, .l = quaver ],
29.         [ .note = n.d, .spd = 20, .l = minim * 4 ]
30.     ],

```

```

31.     [
32.         [ .note = n.e, .spd = 10, .l = minim ],
33.         [ .note = n.a, .spd = 10, .l = minim ],
34.         [ .note = n.d, .spd = 10, .l = minim ]
35.     ]
36. ]

```

Here we've added another array of structures to our **melody array**. Let's say we wanted to access the 2nd note of the 2nd melody array. That would look something like `melody[1][2].note`. If we wanted to access the 7th note of the first melody array, that would be: `melody[0][6].note`.

Now we must modify the other parts of the code to use these **arrays** rather than single values. Since we need two separate note counters and end times, these will become small **arrays** too:

```

38. endTime = [ 0, 0 ]
39. noteCount = [ 0, 0 ]
40.
41. for i = 0 to len( melody ) loop
42.     counter = 0
43.     for j = 0 to len( melody[i] ) loop
44.         temp = melody[i][j].l
45.         melody[i][j].l = counter
46.         counter += temp
47.     repeat
48.     endTime[i] = counter
49. repeat

```

Everything about this is exactly the same as before except it happens twice, once for each melody in our array.

We might want our melodies to be played in different wave types to give the sound of different instruments playing. We could use small **arrays** just like `endTime` and `noteCount` to store information to apply to each melody:

```

51. waveType = [ 3, 1 ]
52. octave = [ 24, 12 ]
53. volume = [ 0.3, 0.2 ]

```

Here we have three small **arrays** which store the wave type, octave and volume modifier for each melody.

Before we look at the main loop, let's set our **timer variable** and the reverb for each channel:

```

55. setReverb( 0, 8000, 0.5 )
56. setReverb( 1, 8000, 0.5 )
57.
58. timer = 0

```

Now let's create the main loop.

```

60. loop
61.     clear()
62.
63.     for i = 0 to len( melody ) loop

```



```

64.     if noteCount[i] < len( melody[i] ) then
65.         if timer >= melody[i][noteCount[i]].l then
66.             note = note2Freq( melody[i][noteCount[i]].note + octave[i] )
67.             speed = melody[i][noteCount[i]].spd
68.             playNote( i, waveType[i], note, volume[i], speed, 0.5 )
69.             noteCount[i] += 1
70.         endif
71.     endif
72.     repeat
73.
74.     timer += ( 120 / 60 ) / 60
75.
76.     if timer >= endTime[0] and timer >= endTime[1] then
77.         noteCount[0] = 0
78.         noteCount[1] = 0
79.         timer = 0
80.     endif
81.
82.     update()
83.     repeat

```

There we have it! Run the program to hear out two melodies playing simultaneously. Beautiful!

This project will work in just about any scenario you can imagine. The **loop** we are using would be your main game loop and you might want it to trigger only at certain times. Feel free to use this template for your own projects!

Happy composing and see you in the next tutorial!

For reference, here is the completed project just below in case you struggled to get the sections right. Feel free to start a new project and copy the whole project below:

```

1. n = [
2.     .c = 60,
3.     .cs = 61,
4.     .d = 62,
5.     .ds = 63,
6.     .e = 64,
7.     .f = 65,
8.     .fs = 66,
9.     .g = 67,
10.    .gs = 68,
11.    .a = 69,
12.    .as = 70,
13.    .b = 71
14. ]
15.
16. semiquaver = 0.25
17. quaver = 0.5
18. crotchet = 1
19. minim = 2
20.
21. melody = [

```

```

22.     [
23.         [ .note = n.d, .spd = 40, .l = quaver ],
24.         [ .note = n.e, .spd = 40, .l = quaver ],
25.         [ .note = n.f, .spd = 40, .l = quaver ],
26.         [ .note = n.g, .spd = 40, .l = quaver ],
27.         [ .note = n.e, .spd = 20, .l = crotchet ],
28.         [ .note = n.c, .spd = 40, .l = quaver ],
29.         [ .note = n.d, .spd = 20, .l = minim * 4 ]
30.     ],
31.     [
32.         [ .note = n.e, .spd = 10, .l = minim ],
33.         [ .note = n.a, .spd = 10, .l = minim ],
34.         [ .note = n.d, .spd = 10, .l = minim ]
35.     ]
36. ]
37.
38. endTime = [ 0, 0 ]
39. noteCount = [ 0, 0 ]
40.
41. for i = 0 to len( melody ) loop
42.     counter = 0
43.     for j = 0 to len( melody[i] ) loop
44.         temp = melody[i][j].l
45.         melody[i][j].l = counter
46.         counter += temp
47.     repeat
48.         endTime[i] = counter
49. repeat
50.
51. waveType = [ 3, 1 ]
52. octave = [ 24, 12 ]
53. volume = [ 0.3, 0.2 ]
54.
55. setReverb( 0, 8000, 0.5 )
56. setReverb( 1, 8000, 0.5 )
57.
58. timer = 0
59.
60. loop
61.     clear()
62.
63.     for i = 0 to len( melody ) loop
64.         if noteCount[i] < len( melody[i] ) then
65.             if timer >= melody[i][noteCount[i]].l then
66.                 note = note2Freq( melody[i][noteCount[i]].note + octave[i] )
67.                 speed = melody[i][noteCount[i]].spd
68.                 playNote( i, waveType[i], note, volume[i], speed, 0.5 )
69.                 noteCount[i] += 1
70.             endif
71.         endif
72.     repeat
73.

```

```
74.     timer += ( 120 / 60 ) / 60
75.
76.     if timer >= endTime[0] and timer >= endTime[1] then
77.         noteCount[0] = 0
78.         noteCount[1] = 0
79.         timer = 0
80.     endif
81.
82.     update()
83. repeat
```

Functions and Keywords used in this Tutorial

`clear()`, `else`, `endif`, `for`, `if`, `loop`, `note2Freq()`, `playNote()`, `repeat`, `setReverb`, `then`, `update()`

TUTORIALS

Tutorial 14: An Introduction to Vectors

Strap your focusing hats on, because we're about to cover something quite tricky. If you can get your head around this, you've just seriously leveled up. As always, getting the hang of using a new technique means practising and writing your own programs which use them. It's **highly** recommended that you have read the [screen coordinates](#) tutorial project before trying this one. If you're already comfortable with this, please go right on ahead!

With all that said and done, we should probably introduce the topic. This tutorial is about vectors. What they are, why they're useful, and some simple things we can do to begin using them.

We will only be covering the very basic parts of using vectors in this tutorial, for more advanced information on vectors in FUZE, check out the next vector project!

Vec-what?!

What is a vector? Well... to put it as simply as possible:

A vector is *multiple numbers* treated as *one thing*.

Let's go into a little more detail.

Some things in life can be described with a single measurement. Take temperature, for example. Temperature is **only** a measurement of heat. Nothing else.

We call things like this **scalars**. Height, weight, volume (both types - loudness and quantity!) are all examples of **scalars**.

But... What if we wanted to describe something like *position*? We live in 3 dimensions, so describing the position of something by just using a single number wouldn't give us very much information!

Let's take the example of a circle right in the middle of our screen. Here's some code to make that happen:

```
1. loop
2.   clear()
3.
4.   circle( gWidth() / 2, gHeight() / 2, 100, 16, white, false )
5.
6.   update()
7. repeat
```

Simple enough! We have a single **loop** which just puts a circle on the screen in one place.

We put the circle at `gwidth() / 2, gheight() / 2` which is exactly in the middle of our screen.

Now, if somebody asked: “Hey... Where’s that circle?”, and we answered with a only single dimension: “It’s at gwidth divided by 2” ... Do you see the problem?

The circle *is* at `gwidth() / 2` on the **x** axis, but it is *also* at `gheight() / 2` on the **y** axis. It is equally at both points, so to describe its position using just one wouldn’t get the job done!

A **vector** tells us both pieces of information at the same time.

Using a Vector in a Program

We can set up a vector very easily, and use it to put our circle on screen again. Check out the code using a **vector** instead.

```
1. position = { gwidth() / 2, gheight() / 2 }
2.
3. loop
4.     clear()
5.
6.     circle( position.x, position.y, 100, 16, white, false )
7.
8.     update()
9. repeat
```

As you can see, line 1 defines a **variable** called `position`. This **variable** stores a pair of coordinates in curly brackets `{}`.

Look out for these curly brackets! If you see some curly brackets in any FUZE code, you know it’s a **vector**!

The clever part about **vectors** in **FUZE⁴ Nintendo Switch** is that we can simply define a **variable** as a **vector**, put what we want in the curly brackets, and FUZE will set up a **structure** with **x** and **y** properties. In our example, we now have `position.x` and `position.y`.

“But it does exactly the same thing!” we hear you scream. Well, yes this is true. But it’s what we can do with this that counts.

Using Vectors to Move a Circle

Remember in the screen tutorial project, we learned how to move a circle around the screen by changing its **x** and **y** location using **variables**? In this tutorial, we’ll be taking this project to the next level.

Because **vectors** are multiple numbers treated as *one thing*, we can apply *one* operation to the whole **vector** and things will work just fine. Check this out:

```
1. position = { gwidth() / 2, gheight() / 2 }
2.
3. loop
4.     clear()
5.
6.     joy = controls( 0 )
7.     position += { joy.lx, -joy.ly } * 8
8.
```

```

9.     circle( position.x, position.y, 100, 16, white, false )
10.
11.     update()
12. repeat

```

Look at that neat code!

Instead of having separate **variables** to store the **x** and **y** position and operating on them separately, we can simply *add* the value of the **Joy-Con** left control stick to the whole position **vector**. Notice we are adding *both* the `joy.lx` and `joy.ly` value together in a **vector** also.

We have multiplied this **vector** by 8 to increase the movement speed, otherwise we get a very slow movement!

Remember, because the **y** axis is 0 at the top of the screen and `gHeight()` at the bottom, we must use a minus sign (-) before `joy.ly` to make the circle move up when we push the control stick upwards.

Just to break this down even further, line 7 is taking each part of the **position vector**, adding the `joy.lx` value to `position.x` and `joy.ly` to `position.y`.

It's really two lines of code in one!

Let's make this movement slightly more advanced, since we are leveling up in this tutorial after all.

Currently, when we let go of the control stick, our circle stops immediately. This might be what we want sometimes, but it is more realistic and certainly more satisfying to have a nice slow down effect.

We can achieve this with a few more lines of code, and with **vectors** it becomes neater than ever.

To achieve a slowdown effect, we need something called *velocity*. Change your code to look like the program below:

```

1. position = { gWidth() / 2, gHeight() / 2 }
2. velocity = { 0, 0 }
3.
3. loop
4.     clear()
5.
6.     joy = controls( 0 )
7.     velocity += { joy.lx, -joy.ly } * 8
8.     position += velocity
9.     velocity *= 0.9
10.
11.     circle( position.x, position.y, 100, 16, white, false )
12.
13.     update()
14. repeat

```

We now have another **variable** at the beginning of the program called **velocity**. This **variable** stores an empty **vector** (`{0, 0}`).

On line 7, you can see that rather than increasing the **position variable** by the control stick values, we increase the **velocity vector** instead. Separating things like this allows us to control *how much* deceleration (slowdown) we want to occur when we let go of the stick.

On line 9, we apply a multiplication to the **velocity vector**. By multiplying by 0.9 every time the **loop** repeats, we are constantly making the number smaller. This only truly takes effect when we let go of the stick however, because if we are pushing the stick in a direction we are continually reading the value and applying it.

As soon as we let go, line 9 takes becomes much more visually apparent, and the amount of *velocity* applied to the circle's position begins decreasing. We see this as a slow down effect.

Decrease this number to make the slow down effect faster. The closer we get to 1, the longer it takes to stop.

Colours as Vectors

You may or may not know this, but all colours on a screen are created using a mix of red, green and blue light.

When we set up a **vector**, we know that FUZE gives us a **structure** with a *.x* and a *.y*. Actually, FUZE automatically creates a *.z* and a *.w* in addition to these, giving us 4 total values. If we do not define these values in a **vector**, they default to 0.

The cool thing about **vectors** in **FUZE⁴ Nintendo Switch** is that it *also* allows us to access these 4 numbers with *.r*, *.g*, *.b* and *.a*, standing for red, green, blue and alpha respectively.

This means we can use a **vector** as a colour! We can specify the amount of each colour, and a transparency (alpha).

These numbers are between 0 and 1, with 0 being no colour at all and 1 being maximum. Let's take the colour **red** for example.

In **FUZE⁴ Nintendo Switch**, the word **red** is really a label for the **vector** {1, 0, 0, 1}, for maximum red, no green or blue, and full opacity.

Let's say we wanted to create a colour of our very own. Check out the example below:

```
1. col = { 0.5, 0.8, 0.1, 1 }
2.
3. loop
4.     clear( col )
5.     update()
6. repeat
```

Here we set up a **vector** on line 1, then use a simple **loop** to clear the screen with our colour. A nice mint green!

If we wanted to increase the amount of blue in the colour *during* the loop, we could do something like this:

```
1. col = { 0.5, 0.8, 0.1, 1 }
2.
```

```
3. loop
4.   clear( col )
5.   col.b += 0.001
6.   update()
7. repeat
```

Run the program to see our mint green colour gently shift into a light blue. Very nice!

Recap

A **vector** is multiple numbers treated as one thing.

You can notice them easily whenever you spot curly brackets {}.

We use them for all kinds of things. Mainly, they are used for position and colour.

This tutorial covers only the very basics of how to use **vectors** in a program. Using **vectors** gives us access to some very impressive and useful maths, but this is something we'll dive into later.

If we want objects to *bounce* off each other as they would in the real world, **vectors** make this much easier. However, this is much too complicated for an introduction to **vectors**, so we'll dive into that at a later time.

Well done. We're getting advanced now! Try setting up your own **vectors** in your projects and use them to control the position of objects, or to change the colour of certain things. Getting the hang of this will really open up a world of experimentation!

See you in the next tutorial!

Functions and Keywords used in this tutorial

`circle()`, `clear()`, `loop`, `repeat`, `update()`

TUTORIALS

Tutorial 15: Sprites

In **FUZE⁴ Nintendo Switch** there are a whole load of **functions** and commands to help us use the vast amount of assets to create a game.

In this tutorial we'll be creating a simple game using the sprite system. As always we'll start simple and add complexity as we go.

Creating a Sprite

Let's get the basics up and running. We need to load an image, then use that image to create a sprite:

```
1. playerSpr = createSprite()  
2. playerImg = loadImage( "Untied Games/Player ships", false )  
3. setSpriteImage( playerSpr, playerImg )
```

Done! First we use the `createSprite()` **function** and store the result in a **variable** called `playerSpr`. Doing this creates a sprite labeled `playerSpr` which we can manipulate using the other sprite **functions**.

Before we get to that, we must assign an image file to the sprite or we won't have anything to look at!

On line 2 we use the `loadImage()` **function** to store an image file into a **variable** called `playerImg`. We then use the `setSpriteImage()` **function** to assign this image file to our sprite on line 3.

Now we can do a whole host of cool things!

setSpriteLocation()

Let's begin by simply putting the sprite on screen. First we'll need to set the location of our sprite:

```
4. setSpriteLocation( playerSpr, { gwidth() / 2, gheight() / 2 } )
```

Here we use the `setSpriteLocation()` **function** to give our player sprite a screen position. We have used `gwidth() / 2, gheight() / 2` to give us the middle of the screen.

That's really all we need to start with. Let's draw our sprite on the screen using a **loop**.

```
6. loop  
7.   clear()  
8.   drawSprites()  
9.   update()  
10. repeat
```

Run the program to see 4 tiny little ships in the middle of our screen.

setSpriteScale()

You might have noticed that these ships are a little small... We need to **scale** the image **up** to make it more usable. Add the following line before the **loop**:

```
5. setSpriteScale( playerSpr, { 4, 4 } )
6.
7. loop
8.   clear()
9.   drawSprites()
10.  update()
11. repeat
```

The `setSpriteScale()` **function** allows us to **multiply** the **x** and **y** size of a sprite. We have used the number 4 in both the **x** and the **y** to make our sprite 4 times larger. If one of these numbers is different it might look a little stretched out!

Run the program now and our sprite should be a much better size.

setSpriteAnimation()

Our image file is a collection of 4 ships for a designer to choose from. Since the sprite is created from the whole image, we have a sprite made of 4 ships. If we were to use this to make a game we only want a single ship, not all 4 at once.

We can use the `setSpriteAnimation()` **function** to do just that. This clever **function** tells FUZE which **tiles** from an image we want to display. We use it to animate a sprite, but here we will be using it to display a single tile.

Add the following line before the **loop**:

```
6. setSpriteAnimation( playerSpr, 0, 0, 0 )
7.
8. loop
9.   clear()
10.  drawSprites()
11.  update()
12. repeat
```

We should take a look at this **function** more detail. The first argument is the **variable** which stores our sprite. Nice and easy.

Next up we have three numbers. The first of these is the **starting tile** for the animation. This tells FUZE which of the tiles in the image to start with. Take a look at the graphic below:



As you can see, the tiles begin at 0 and count up. The tile we want is tile 0. This means the first number in our `setSpriteAnimation()` **function** is 0.

The second number is the **end** tile for our animation. Since we want our ship to remain the same, this number is also 0.

The last number in the **function** is the **speed** of the animation, in **frames per second**. Since our animation is only a single tile, we have a speed of 0.

Run the program and we'll have a nicely sized, single ship on screen.

What do you mean it's boring? Alright, let's move the sprite around.

setSpriteSpeed()

To actually move the sprite we'll need to use the `setSpriteSpeed()` **function**.

We can use this **function** to apply a direction and movement speed to a sprite:

```
7. setSpriteSpeed( playerSpr, { 60, 0 } )
8.
9. loop
10.   clear()
11.   updateSprites()
12.   drawSprites()
13.   update()
14. repeat
```

Notice we have added two lines this time. Since we want the position of our sprite to change as the program is running, we must use the `updateSprites()` **function** to the main **loop**. This **must** go before the `drawSprites()` **function**.

In the `setSpriteSpeed()` **function** there are two arguments. The first is of course the **variable** which stores the sprite we want to move.

The second is a two-dimensional vector which sets the movement speed in pixels for each axis. In our line we have applied a velocity of 60 to the **x** axis and a velocity of 0 to the **y** axis. This means our ship will move to the right at 60 pixels per second.

Run the program to see our ship move gracefully along the **x** axis.

Creating a Game Using the Sprite Functions

In the next tutorial, we'll be using all that we've learned to create a side-scrolling game where we must avoid meteorites.

Make sure you're familiar with the **functions** we've covered here, then we'll see you in the next project!

Functions and Keywords Used in this Tutorial

`createSprite()`, `clear()`, `drawSprites()`, `loop`, `repeat`, `setSpriteAnimation`, `setSpriteImage()`, `setSpriteSpeed()`, `update()`, `updateSprites()`

TUTORIALS

Tutorial 16: Creating a Game using the Sprite Functions

Welcome back! In this tutorial we will be creating a simple game using the sprite **functions** covered in the previous project.

We'll be using the same ship assets from the brilliant Untied Games along with some awesome meteorites.

In this game, our ship will always be falling downwards. The challenging part is that we will have to press the A button to keep our ship flying. This might remind you of a game you've played before!

Let's start just like before by creating the player sprite from an image file:

```
1. playerSpr = createSprite()  
2. playerImg = loadImage( "Untied Games/Player ships", false )  
3. setSpriteImage( playerSpr, playerImg )
```

Next we'll need to set the location of the sprite.

To help us out with this, we should create a variable which stores the width and height of the screen since we'll need these numbers a lot going forward:

Make a couple of new lines from line 1. This will move our other lines down slightly:

```
1. screen = { gwidth(), gheight() }  
2.  
3. playerSpr = createSprite()  
4. playerImg = loadImage( "Untied Games/Player ships", false )  
5. setSpriteImage( playerSpr, playerImg )
```

We have defined a **variable** called `screen` which stores a vector. This vector has an **x** property of `gwidth()` and a **y** property of `gheight()`. This means we can now access the width or height of the screen by using `screen.x` or `screen.y`. Very helpful!

With that done, we should create a **variable** which will store the **scale** multiplier for our sprites. It will save us some typing in the future to do this! Add the following new line:

```
1. screen = { gwidth(), gheight() }  
2. scale = { screen.y / 270, screen.y / 270 }  
3.  
4. playerSpr = createSprite()  
5. playerImg = loadImage( "Untied Games/Player ships", false )  
6. setSpriteImage( playerSpr, playerImg )
```

The **scale variable** stores a **vector** to use in our `setSpriteScale()` **functions**. By using `screen.y / 270` our **scale** will change depending on whether the console is docked or undocked. We use the same number for both the **x** and the **y** parts of the **vector** so that our sprites scale evenly.

Now we can create a **variable** which will store the player position for easy reference:

```
1. screen = { gwidth(), gheight() }
2. scale = { screen.y / 270, screen.y / 270 }
3.
4. playerSpr = createSprite()
5. playerImg = loadImage( "Untied Games/Player ships", false )
6. setSpriteImage( playerSpr, playerImg )
7. playerPos = { screen.x / 20, screen.y / 2 }
```

We've called the **variable** `playerPos`. It stores a **vector** with a particular location on screen. We can now access the position using `playerPos.x` and `playerPos.y`.

Lastly, we might want to store the player velocity in a **variable** too to make things easier later on in the program. Let's add an **empty vector** for now:

```
1. screen = { gwidth(), gheight() }
2. scale = { screen.y / 270, screen.y / 270 }
3.
4. playerSpr = createSprite()
5. playerImg = loadImage( "Untied Games/Player ships", false )
6. setSpriteImage( playerSpr, playerImg )
7. playerPos = { screen.x / 20, screen.y / 2 }
8. playerVel = { 0, 0 }
```

Alright, let's use the sprite **functions** to make use of all this information.

```
10. setSpriteAnimation( playerSpr, 0, 0, 0 )
11. setSpriteScale( playerSpr, scale )
12. setSpriteLocation( playerSpr, playerPos )
```

Drawing and Moving the Player

Alright! Let's create a **loop** to draw the sprite on screen.

```
14. loop
15.   clear()
16.   updateSprites()
17.   drawSprites()
18.   update()
19. repeat
```

Currently our **loop** only puts the sprite on screen. Let's add some controls to this program. First, we'll want to recalculate the `screen` and `scale` **variables** in the **loop**. This will mean that if we change from handheld mode to TV mode by placing the console into the Nintendo Switch dock, the screen and scale will be updated so it looks right:

```
14. loop
15.   clear()
16.   screen = { gwidth(), gheight() }
17.   scale = { screen.y / 270, screen.y / 270 }
18.   setSpriteScale( playerSpr, scale )
19.   updateSprites()
```

```

20. drawSprites()
21. update()
22. repeat

```

There we go. Now we can use the `screen` and `scale` **variables** in the **loop** without worrying if the console is handheld or TV mode. To make the scale actually change, we use the `setScale()` **function** in the **loop** on line 18.

Now let's apply some controls:

```

14. loop
15.   clear()
16.   screen = { gwidth(), gheight() }
17.   scale = { screen.y / 270, screen.y / 270 }
18.
19.   c = controls( 0 )
20.
21.   playerVel = { c.lx * screen.x / 4, screen.y / 3 - c.a * screen.y / 1.5 }
22.
23.   setSpriteSpeed( playerSpr, playerVel )
24.   setSpriteScale( playerSpr, scale )
25.
26.   updateSprites()
27.   drawSprites()
28.   update()
29. repeat

```

We've added some space between the lines in the program above to make things a little more clear.

There are three new lines here and they can be seen on line 19, line 21 and line 23.

First, we use the `controls()` **function** to store the state of the controls into a **variable**. We've called this **variable** `c`.

Next, we update the `playerVel` **variable**. In the curly brackets are the two ways we want to affect the velocity.

```

21.   playerVel = { c.lx * screen.x / 4, screen.y / 3 - c.a * screen.y / 1.5 }

```

We want to be able to move the ship back and forth slightly to speed up and slow down. We are using the left control stick (`c.lx`) to do this.

Since the control stick returns a value between -1 and 1, the effect of this will be very tiny indeed. We have multiplied this by `screen.x / 4` to enhance the movement speed, and by using the `{screen.x}` variable here we can make sure the speed is the same in handheld or TV mode.

Similarly, for the `y` axis velocity, we use the `screen.y` to move the ship down at a speed dependent on the screen size. We minus the value of the A button (`c.a`) to move the ship upwards. Since the A button is either 0 or 1, we multiply this by `screen.y / 1.5` to give us a stronger effect.

In order to make the movement actually take place, we need to apply the new values in the `playerVel` **variable** to the sprite velocity.

On line 23 we use the `setSpriteSpeed()` **function** to apply the values in the `playerVel` **variable** to the player sprite.

Run the program and get a feel for moving around the screen! Remember, you must hold the A button in order to move and the left control stick will speed up and slow down the ship.

Creating Boundaries

At the moment, we can move our player off screen. This isn't very good and will make the game too easy. Let's introduce some restrictions to how far the player can move.

Below, we use a new **function** called `clamp()` to restrict the values of the `playerPos` **variable**. This very useful **function** can save us lots of **if statements** when used correctly. The first value in the brackets is the value to restrict, the second is the lower limit and the third is the upper limit.

```
14. loop
15.   clear()
16.   screen = { gwidth(), gheight() }
17.   scale = { screen.y / 270, screen.y / 270 }
18.
19.   c = controls( 0 )
20.
21.   playerVel = { c.lx * screen.x / 4, screen.y / 3 - c.a * screen.y / 1.5 }
22.
23.   playerPos = getSpriteLocation( playerSpr )
24.
25.   playerPos.x = clamp( playerPos.x, screen.x / 10, screen.x - screen.x / 10 )
26.   playerPos.y = clamp( playerPos.y, screen.y / 10, screen.y - screen.y / 10 )
27.
28.   setSpriteLocation( playerSpr, playerPos )
29.   setSpriteSpeed( playerSpr, playerVel )
30.   setSpriteScale( playerSpr, scale )
31.
32.   updateSprites()
33.   drawSprites()
34.   update()
35. repeat
```

To understand why we are using the values `screen.x / 10` and `screen.x - screen.x / 10` let's visualise what we are achieving:



In the picture, the **yellow** outer box outlines the actual Nintendo Switch screen. The width and height of that screen is stored in the `screen.x` and `screen.y` **variables**.

The **white** inner box outlines the boundary we have created.

If the player position becomes more or less than the boundaries of our inside box, our `clamp()` **function** limits the values.

In order to actually place the player sprite on screen at the restricted positions, we must set the sprite location using the `setSpriteLocation()` **function**. To make this work, we use the `getSpriteLocation()` **function** on line 23 to make sure our player sprite is always placed at the location it should be.

Creating Obstacles

It's not really a very fun game unless we have something to do! Let's create a bunch of flying rocks to avoid.

First, we'll need to load the images we need and create the sprites.

We'll be using more of the brilliant Untied Games assets for this. There are 4 different asteroid assets to use. We will store all of these into an array.

We'll need to do the following changes before the main **loop**:

```
14. rockImgs = [  
15.     loadImage( "Untied Games/Asteroid A", false ),  
16.     loadImage( "Untied Games/Asteroid B", false ),  
17.     loadImage( "Untied Games/Asteroid C", false ),  
18.     loadImage( "Untied Games/Asteroid D", false )  
19. ]
```

Here is the array of images. With this, we can access any of the images with something like `rockImgs[1]`.

Next up we must create the array of information we'll use for the rocks which appear on screen. We'll call this array `rocks` and we'll be creating it just beneath the images array:

```

21. array rocks[50] = [
22.     .spr = 0,
23.     .pos = { 0, 0 },
24.     .vel = { 0, 0 }
25. ]

```

That's that! We've created an array of 50 elements, each one with three properties. Each rock will have a `.spr` property to store the sprite, a `.pos` property to store the position on screen and a `.vel` property to store the velocity.

Now let's populate this array with information it needs. We'll need a **for loop** for this:

```

27. for i = 0 to len( rocks ) loop
28.     n = random( 4 )
29.     rocks[i].spr = createSprite()
30.     setSpriteImage( rocks[i].spr, rockImgs[n] )
31.     rocks[i].pos = { screen.x + random( screen.x * 2 ), random( screen.y ) }
32.     rocks[i].vel = { random( -screen.x / 5 ), random( 32 ) - 16 }
33.     setSpriteAnimation( rocks[i].spr, 0, 39, 10 )
34.     setSpriteLocation( rocks[i].spr, rocks[i].pos )
35. repeat

```

This **for loop** counts up to the length of the `rocks` **array**. Let's look at each line in detail:

```

28.     n = random( 4 )

```

First, we create a **variable** called `n` which stores a random number out of 4 possibilities. This gives us 0, 1, 2 and 3. This random number will be used to select a rock image from the `rockImgs` **array**.

```

29.     rocks[i].spr = createSprite()
30.     setSpriteImage( rocks[i].spr, rockImgs[n] )

```

On line 29 we use the `createSprite()` **function** to create a sprite for each rock. On the next line, we use the `setSpriteImage()` **function** to set a the randomly selected rock image to the sprite.

```

31.     rocks[i].pos = { screen.x + random( screen.x * 2 ), random( screen.y ) }

```

Next, on line 31, we set the `.pos` property of each rock to store a position. We want the rocks randomly distributed over the `y` axis, so the `y` part of the vector is `random(screen.y)`. Since we want to start with no rocks on screen and have them travel towards us, the `x` part of the vector is `screen.x + random(screen.x * 2)`. This makes sure the rocks are a random amount further than the edge of the screen to begin with.

```

32.     rocks[i].vel = { random( -screen.x / 5 ), random( 32 ) - 16 }

```

On line 32 we set the `.vel` property for each rock that will be used for the velocity. The speed the rocks travel towards us needs to depend on the screen size, otherwise in handheld and TV mode the rocks will feel like they move at very different speeds. We use a random number chosen out of `-screen.x / 5`. We must put a `-` in front of `screen.x` to make sure the asteroids travel towards us instead of away!

We want the rocks to move up or down slowly to really add variety. For this we use a random number out of 32 in the `y` part of the vector, but if we minus 16 from the result we have a range of -16 to 16 giving us randomly chosen upwards or downward movement at a range of speeds.

```
33.     setSpriteAnimation( rocks[i].spr, 0, 39, random( 10 ) + 10 )
```

On line 33 we set the animation for each rock. Each asteroid image has 40 frames of animation, so our start tile is 0 and the end tile is 39. For the animation speed we have used a random number between 10 and 20 to animate the rocks at different speeds.

```
34.     setSpriteLocation( rocks[i].spr, rocks[i].pos )
```

Finally, we set the location of each rock sprite using the `setSpriteLocation()` function.

Still here? I bet you're thinking "Wow, this project rocks!"

...

What do you mean it was a terrible joke? Alright let's move on.

Drawing the Rocks on Screen

Now that we've got all the information we need in the `rocks` array, we can draw them to the screen. We'll be going back into the main `loop` for this. Because we've inserted lots of new code before the `loop`, the line numbers are a little different now. To make sure you are adding code in the right place, make sure your main loop looks like the one below before we get started:

```
37. loop
38.     clear()
39.     screen = { gwidth(), gheight() }
40.     scale = { screen.y / 270, screen.y / 270 }
41.
42.     c = controls( 0 )
43.
44.     playerVel = { c.lx * screen.x / 4, screen.y / 3 - c.a * screen.y / 1.5 }
45.
46.     playerPos = getSpriteLocation( playerSpr )
47.
48.     playerPos.x = clamp( playerPos.x, screen.x / 10, screen.x - screen.x / 10 )
49.     playerPos.y = clamp( playerPos.y, screen.y / 10, screen.y - screen.y / 10 )
50.
51.     setSpriteLocation( playerSpr, playerPos )
52.     setSpriteSpeed( playerSpr, playerVel )
53.     setSpriteScale( playerSpr, scale )
54.
55.     updateSprites()
56.     drawSprites()
57.     update()
58. repeat
```

We need to add a `for` loop before the `updateSprites()` line. Go to line 54 and create a couple of new lines:

```
53.     setSpriteScale( playerSpr, scale )
54.
```

```
55.  
56.  
57.     updateSprites()
```

We'll be adding the **for loop** starting on line 55. We must count over each rock in the `rocks` **array** and do a number of things for each one:

```
55.     for i = 0 to len( rocks ) loop  
56.         setSpriteScale( rocks[i].spr, scale )  
57.         setSpriteSpeed( rocks[i].spr, rocks[i].vel )  
58.         rocks[i].pos = getSpriteLocation( rocks[i].spr )  
59.         rockSize = getSpriteSize( rocks[i].spr )  
60.         if rocks[i].pos.x + rockSize.x / 2 < 0 then  
61.             rocks[i].pos = { screen.x + random( screen.x ), random( screen.y ) }  
62.             setSpriteLocation( rocks[i].spr, rocks[i].pos )  
63.         endif  
64.     repeat  
65.  
66.         updateSprites()  
67.         drawSprites()  
68.         update()  
69. repeat
```

Phew! That's quite a complex looking bit of code right there. Fear not, it's actually quite simple if we take it step by step.

```
56.         setSpriteScale( rocks[i].spr, scale )
```

The first thing we do in the **for loop** is to set the scale of each rock using the `setSpriteScale()` **function**.

```
57.         setSpriteSpeed( rocks[i].spr, rocks[i].vel )
```

Next, we want to apply the speed for each rock to the sprite. We use the `setSpriteSpeed()` **function**, applying the **vector** stored in `rocks[i].vel` to each sprite. This line moves each rock across the screen and up or down depending on that rock's **y** velocity.

```
58.         rocks[i].pos = getSpriteLocation( rocks[i].spr )
```

Next up we update the each rock's position variable to be the location of the sprite. Doing this allows us to write neater code later in the **for loop**. We are going to be checking each rock's position in just a minute and this gives us a neat way to check a rock's current position.

```
59.         rockSize = getSpriteSize( rocks[i].spr )
```

Similarly, here we are creating a **variable** called `rockSize` and using it to store the size of each rock on screen. This will be a very useful **variable** in just a minute!

```
60.         if rocks[i].pos.x + rockSize.x / 2 < 0 then  
61.             rocks[i].pos = { screen.x + random( screen.x ), random( screen.y ) }  
62.             setSpriteLocation( rocks[i].spr, rocks[i].pos )  
63.         endif
```

When a rock travels off the left side of the screen, we must do something to bring it back to the right hand side. Otherwise we would need a huge number of rocks! This **if statement** allows us to re-use each rock in the **array** when it has traveled off screen.

We check if the **x** position of the right side of each rock (`rocks[i].pos.x + rockSize.x / 2`) has become less than 0. If it has, we update the `rocks[i].pos` **variable** to be a random amount further than the right side of the screen on the **x** axis (`screen.x + random(screen.x)`) and a random position on the **y** axis (`random(screen.y)`). Finally we use the `setSpriteLocation()` **function** to update the location of the sprite.

Done!

Run the program to see our rocks flying across the screen towards us. When a rock travels off the left side of the screen, it will eventually come back on screen from the right!

Practise maneuvering around the rocks! We're about to add collision.

Colliding with the Rocks

Before we write the code which allows us to collide with a rock, it would be very cool if we could see some sort of explosion effect happen when we do.

Luckily, we have plenty of awesome explosion effects! You know the drill, we need to load an image and create a sprite. Add the following lines just before the main **loop** (you'll have to create a couple of lines of space):

```
37. expSpr = createSprite()  
38. expImg = loadImage( "Untied Games/Explosion 09", false )  
39. setSpriteImage( expSpr, expImg )
```

As usual, we create a **variable** to store the image file. We've called ours `expImg`. Next we create another **variable** (`expSpr` which stores the sprite).

Another thing we'll need to do is to set the visibility of this sprite to `false`. We do not want to see the explosion yet, only when we collide with a rock. Add the following line:

```
40. setSpriteVisibility( expSpr, false )
```

The `setSpriteVisibility()` **function** is incredibly useful and nice and simple. The first argument is the sprite **variable** and the second is either `true` for visible or `false` for invisible.

Lastly, we'll need some sort of flag to tell us whether the player is alive or not. This should be a `true` or `false` **variable** at the start of our program. Add the following line just before the **loop** line:

```
41. alive = true
```

Now we have everything we need to make the explosion effect happen, we just need to collide with a rock!

This next bit of code will take place in the **for loop** which draws the rocks because we must check the distance between the player and **each rock**.

Below is the whole **for loop** for clarity:

```

61.   for i = 0 to len( rocks ) loop
62.       setSpriteScale( rocks[i].spr, scale )
63.       setSpriteSpeed( rocks[i].spr, rocks[i].vel )
64.       rocks[i].pos = getSpriteLocation( rocks[i].spr )
65.       rockSize = getSpriteSize( rocks[i].spr )
66.
67.       if rocks[i].pos.x + rockSize.x / 2 < 0 then
68.           rocks[i].pos = { screen.x + random( screen.x ), random( screen.y ) }
69.           setSpriteLocation( rocks[i].spr, rocks[i].pos )
70.       endif
71.
72.       if distance( playerPos, rocks[i].pos ) < rockSize.x / 2 - 50 and alive then
73.           alive = false
74.           setSpriteAnimation( expSpr, 0, 89, 14 )
75.           setSpriteLocation( expSpr, playerPos )
76.           setSpriteScale( expSpr, scale )
77.           setSpriteVisibility( expSpr, true )
78.           setSpriteVisibility( playerSpr, false )
79.       endif
80.       repeat
81.
82.           updateSprites()
83.           drawSprites()
84.           update()
85.       repeat

```

Our new section of code is the **if statement** starting at line 79. As usual, we'll go through this line by line:

```

72.       if distance( playerPos, rocks[i].pos ) < rockSize.x / 2 - 50 and alive then

```

For us to collide with a rock our position must overlap with the rock's position. We check if the distance between the centre of our ship (`playerPos`) and the centre of each rock (`rocks[i].pos`) is less than half the size of that rock minus 50 pixels (`< rockSize.x / 2 - 50`). The 50 pixels part is really personal preference, changing this number will make the collision looser or tighter.

The second part of the **if statement** check is whether the `alive` variable is `true`. We only want to collide with a rock **once**.

```

73.           alive = false

```

The first thing we do in the **if statement** is to make the `alive` variable `false`. This means we can only collide once. Once we do, the **if statement** can no longer be `true`.

```

74.           setSpriteAnimation( expSpr, 0, 89, 14 )

```

Next up we set the animation for the explosion sprite. Our explosion sprite has 90 frames, beginning with 0 and ending at 89. A speed of 14 gives us a nice looking explosion, feel free to change this!

```

75.           setSpriteLocation( expSpr, playerPos )

```

Now we must set the location of the explosion sprite to be the same as the player position! We wouldn't want the explosion happening in a random place on screen.

```

76.           setSpriteScale( expSpr, scale )

```

Next we set the scale multiplier for the explosion sprite. Simple enough!

```
77.         setSpriteVisibility( expSpr, true )
78.         setSpriteVisibility( playerSpr, false )
```

These next two lines set the visibility for the player to **false** and the visibility for the explosion to **true**. Without these, we wouldn't see anything!

That's that. We're so close to the finish line!

We have one last thing to do. Once the explosion animation has finished we want to set the visibility back to **false**. without this the explosion animation will keep looping forever!

We'll need an **if statement** just before the `updateSprites()` function:

```
82.     if getSpriteAnimFrame( expSpr ) >= 89 then
83.         setSpriteVisibility( expSpr, false )
84.     endif
85.
86.     updateSprites()
87.     drawSprites()
88.     update()
89. repeat
```

As you can see, this part comes just before the last 4 lines of the **loop**.

We use the `getSpriteAnimFrame()` function to check which frame of animation our explosion is currently on. Since we know that the explosion's last animation frame is frame 89, we have an easy way to check if it's finished.

We use the `setSpriteVisibility()` function one last time to make the visibility **false** once it reaches the end of its animation cycle.

Complete!

Congratulations on making it through the tutorial! Now you can customise the game to your heart's content. If you unfortunately break anything at all, you can find a complete version of the program just below. Feel free to copy and paste all the code below into a new project file:

```
1. screen = { gwidth(), gheight() }
2. scale = { screen.y / 270, screen.y / 270 }
3.
4. playerSpr = createSprite()
5. playerImg = loadImage( "Untied Games/Player ships", false )
6. setSpriteImage( playerSpr, playerImg )
7. playerPos = { screen.x / 20, screen.y / 2 }
8. playerVel = { 0, 0 }
9.
10. setSpriteAnimation( playerSpr, 0, 0, 0 )
11. setSpriteScale( playerSpr, scale )
12. setSpriteLocation( playerSpr, playerPos )
13.
14. rockImgs = [
15.     loadImage( "Untied Games/Asteroid A", false ),
16.     loadImage( "Untied Games/Asteroid B", false ),
```

```

17. loadImage( "Untied Games/Asteroid C", false ),
18. loadImage( "Untied Games/Asteroid D", false )
19. ]
20.
21. array rocks[50] = [
22.     .spr = 0,
23.     .pos = { 0, 0 },
24.     .vel = { 0, 0 }
25. ]
26.
27. for i = 0 to len( rocks ) loop
28.     n = random( 4 )
29.     rocks[i].spr = createSprite()
30.     setSpriteImage( rocks[i].spr, rockImgs[n] )
31.     rocks[i].pos = { screen.x + random( screen.x * 2 ), random( screen.y ) }
32.     rocks[i].vel = { random( -screen.x / 5 ), random( 32 ) - 16 }
33.     setSpriteAnimation( rocks[i].spr, 0, 39, 10 )
34.     setSpriteLocation( rocks[i].spr, rocks[i].pos )
35. repeat
36.
37. expSpr = createSprite()
38. expImg = loadImage( "Untied Games/Explosion 09", false )
39. setSpriteImage( expSpr, expImg )
40. setSpriteVisibility( expSpr, false )
41. alive = true
42.
43. loop
44.     clear()
45.     screen = { gwidth(), gheight() }
46.     scale = { screen.y / 270, screen.y / 270 }
47.
48.     c = controls( 0 )
49.
50.     playerVel = { c.lx * screen.y / 4, screen.y / 3 - c.a * screen.y / 1.5 }
51.
52.     playerPos = getSpriteLocation( playerSpr )
53.
54.     playerPos.x = clamp( playerPos.x, screen.x / 10, screen.x - screen.x / 10 )
55.     playerPos.y = clamp( playerPos.y, screen.y / 10, screen.y - screen.y / 10 )
56.
57.     setSpriteLocation( playerSpr, playerPos )
58.     setSpriteSpeed( playerSpr, playerVel )
59.     setSpriteScale( playerSpr, scale )
60.
61.     for i = 0 to len( rocks ) loop
62.         setSpriteScale( rocks[i].spr, scale )
63.         setSpriteSpeed( rocks[i].spr, rocks[i].vel )
64.         rocks[i].pos = getSpriteLocation( rocks[i].spr )
65.         rockSize = getSpriteSize( rocks[i].spr )
66.
67.         if rocks[i].pos.x + rockSize.x / 2 < 0 then
68.             rocks[i].pos = { screen.x + random( screen.x ), random( screen.y ) }
69.             setSpriteLocation( rocks[i].spr, rocks[i].pos )
70.         endif
71.
72.         if distance( playerPos, rocks[i].pos ) < rockSize.x / 2 - 50 and alive then
73.             alive = false

```



```

74.     setSpriteAnimation( expSpr, 0, 89, 14 )
75.     setSpriteLocation( expSpr, playerPos )
76.     setSpriteScale( expSpr, scale )
77.     setSpriteVisibility( expSpr, true )
78.     setSpriteVisibility( playerSpr, false )
79.   endif
80.   repeat
81.
82.     if getSpriteAnimFrame( expSpr ) >= 89 then
83.       setSpriteVisibility( expSpr, false )
84.     endif
85.
86.     updateSprites()
87.     drawSprites()
88.     update()
89.   repeat

```

clamp(), clear(), controls(), createSprite(), distance(), drawSprites(), else, endIf, for,
 getSpriteAnimFrame(), getSpriteSize(), if, loop, repeat, setSpriteAnimation, setSpriteLocation(),
 setSpriteImage(), setSpriteScale, setSpriteSpeed(), setSpriteVisibility(), then, to, update(),
 updateSprites()

TUTORIALS

3D Tutorial 1: Simple Shapes

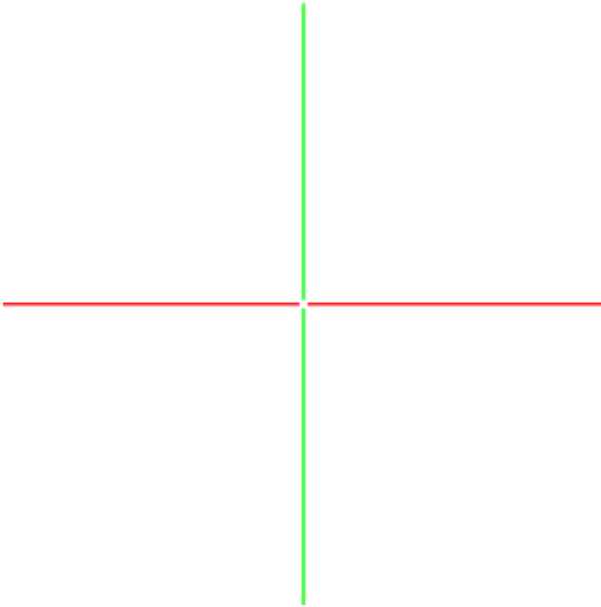
Welcome to the first of our tutorials on 3D projects!

Strap yourself in for quite the read! There's lots of information here which must be explained clearly, but once it's out of the way we can start throwing 3D objects around the screen like there's no tomorrow!

Before we get started, let's refresh a couple of important things.

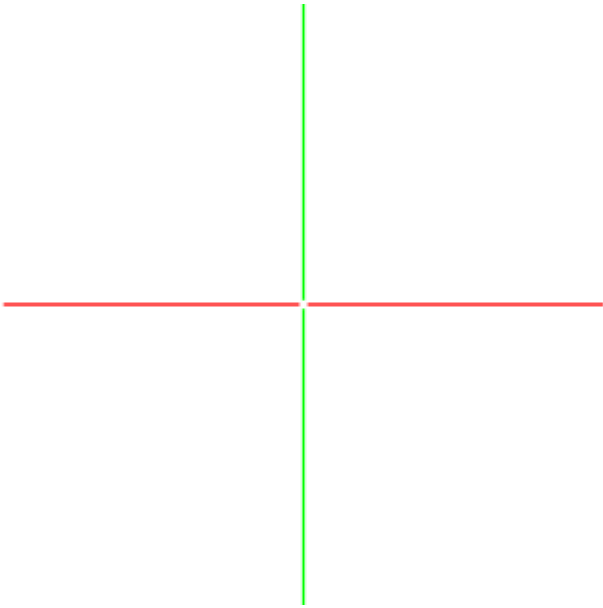
X and Y

Below is a picture of an **x** and **y** axis. Seem familiar?



Imagine we have the number **0** sat right in the middle of our axes, where the little white dot is.

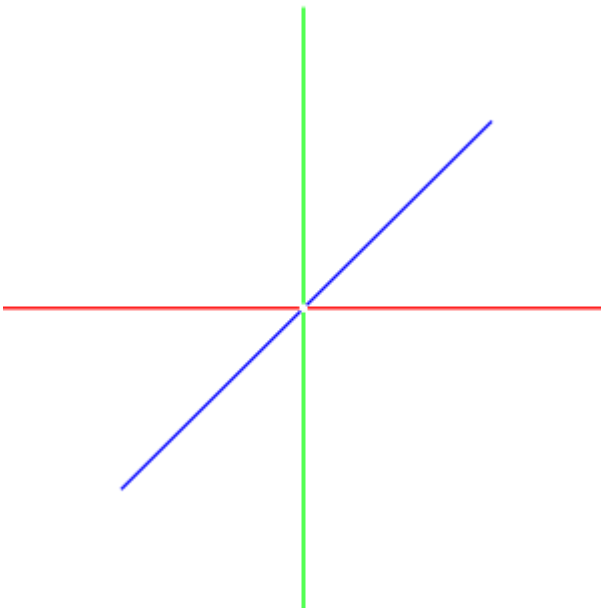
If we move along either axis, away from the centre point of **0**, we will either be *increasing* or *decreasing* along one of them. Take a look below to see what we mean!



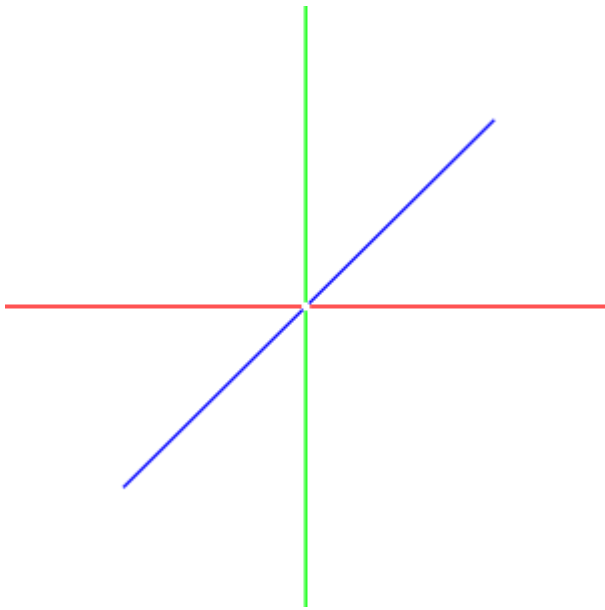
Happy with that? Excellent.

Z

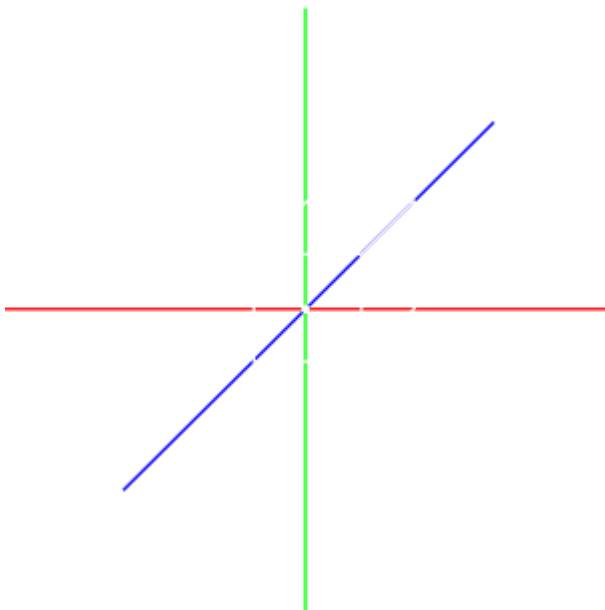
Let's throw another axis in there for a total of 3 Dimensions. We call this axis **z**.



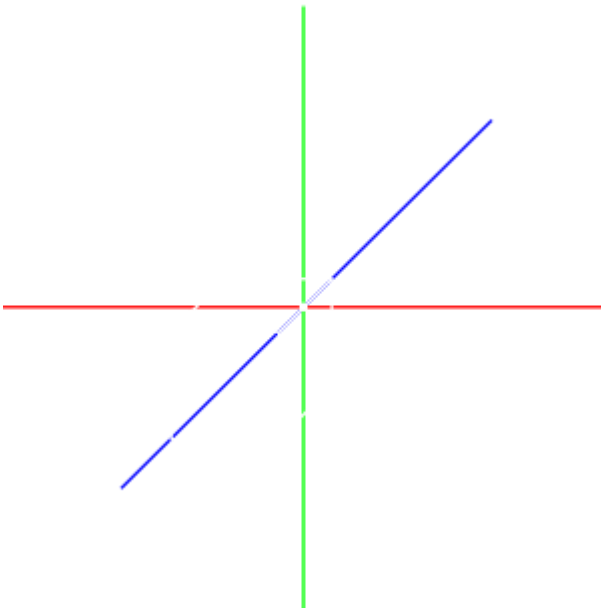
This line brings us into 3D space. It behaves just like our **x** and **y** axes - you can still increase and decrease your position:



Take a look at the cube below sitting nicely in our 3D space to picture the z axis properly.



Let's say I want to *increase* the z position of our cube. That would look something like this:



See how our cube has moved along the **z** axis? Almost as if it has gotten *closer* to us.

It's important to remember that in 3D space, things are a little different than 2D.

We must always think about the position of the **camera**. Depending on where our **camera** is, we might see the **x**, **y** and **z** axes very differently!

Way of the Cube

Time for some programming.

Let's create a simple program to make a single cube appear in 3D space.

Type (or copy and paste) the following code into the **FUZE⁴ Nintendo Switch** editor.

```
1. obj = placeObject( cube, { 0, 0, 0 }, { 1, 1, 1 } )
2. setObjectMaterial( obj, red, 1, 1 )
3. setCamera( { 0, 0, 10 }, { 0, 0, 0 } )
4.
5. loop
6.     clear()
7.     drawObjects()
8.     update()
9. repeat
```

When you run this program, don't be shocked! You won't see a cube at all. In fact, you'll see a square.

There is a very good reason for this, but before we jump right in and explain why, let's talk about these exciting new **functions** we're using.

placeObject()

```
1. obj = placeObject( cube, { 0, 0, 0 }, { 1, 1, 1 } )
```

Line 1 defines a **variable** called `obj` (short for object). In this **variable** we are storing a shape created by the `placeObject()` **function**.

The first **argument** in the `placeObject()` function is the type of object. **FUZE⁴ Nintendo Switch** knows a few basic 3D shapes by name. We have **cube**, **sphere**, **pyramid**, **cone**, **cylinder**, **wedge** and **hemisphere**.

For now, we'll keep things simple and use a **cube**.

The second argument in `placeObject()` is a **vector** which describes the position of our object. We want our cube to appear at **0** on the **x** axis, **0** on the **y** axis and **0** on the **z** axis. In other words, right in the middle of our 3D world space. Change these numbers to change the location of our cube! See what happens when you change each of the numbers in the **vector**.

Lastly, we have a **vector** to set the **scale** of our object on all 3 axes. We want our cube to be a sensible size and the same size in all directions, so we are using the number **1** for each element of the **vector**. Try changing these numbers to change the dimensions of our cube!

setObjectMaterial()

```
2. setObjectMaterial( obj, red, 1, 1 )
```

Line 2 sets the material our object is made from. This will change the way light behaves on the surface of the object.

The first **argument** in the `setObjectMaterial()` **function** is the **variable** which stores the object we are configuring. We're configuring our cube, which is stored in the `obj` **variable**.

The second **argument** is the colour we want our object to be. We can either use a colour name here, or an **RGBA (red, green, blue, alpha) vector**.

The third **argument** is a **boolean** value (either true or false) for metal or non-metal.

Finally, our last **argument** is **roughness**. This value can be anything between **0** and **1**, where **0** is totally smooth and very shiny, and **1** is full roughness, which makes the light appear smoother.

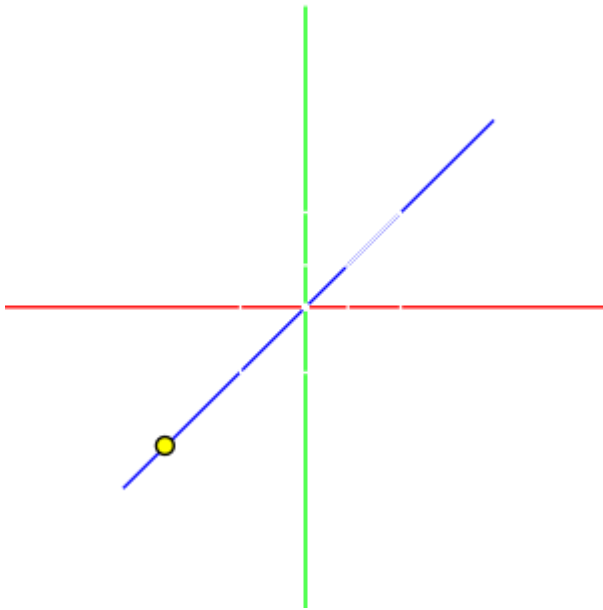
setCamera()

```
3. setCamera( { 0, 0, 10 }, { 0, 0, 0 } )
```

Without a camera, we won't be able to see anything!

The `setCamera()` **function** contains two **arguments**. Our first **argument** is a **vector** which sets the position of the camera in 3D world space.

Notice that our camera is positioned at **0** on the **x** axis, **0** on the **y** axis and **10** on the **z**. What does this mean? Take a look at the diagram below:



In **FUZE⁴ Nintendo Switch**, when working in 3D space, each unit of **1** represents **1** meter. By placing our camera at `{0, 0, 10}` we put it at 10 meters positive on the **z** axis.

The second **argument** is the position in 3D space the camera is *pointing at*. We want our camera looking right at our cube which is positioned right in the centre of world space at `{0, 0, 0}`, so we're using that!

Okay! We've got the tough parts out of the way.

drawObjects()

```
5. loop
6.   clear()
7.   drawObjects()
8.   update()
9. repeat
```

The main **loop** of our code is a simple `clear()` and `update()` with only one **function** call.

We use the `drawObjects()` **function** to prepare our 3D world to be sent to the screen, and the `update()` **function** actually sends all of this to the screen for us to see!

Well done. That was a lot to take in, but now you're equipped to move on.

So have you figured out why we only see a square?

The reason is that we are looking at our cube directly from the front!

So let's change that!

Change the `setCamera()` line in your code so it looks like this:

```
3. setCamera( { 0, 5, 10 }, { 0, 0, 0 } )
```

Notice all we have done is changed the **y** element of the position **vector** to **5** instead of **0**. This will move the camera upwards on the **y** axis by 5 metres.

Run the program to see our cube take form.

See you in the next tutorial! We'll shed some *light* on the situation.

Functions and Keywords used in this Tutorial

`clear()`, `drawObjects()`, `loop`, `placeObject()`, `repeat`, `setObjectMaterial()`, `setCamera()`, `update()`

TUTORIALS

3D Tutorial 2: Simple Lighting

In this tutorial we'll be using the same project as the previous lesson only with an extra line.

When we begin adding light to the 3D world space, things really start to take shape.

There are four functions in **FUZE⁴ Nintendo Switch** which we can use to add light. These are: `setAmbientLight()`, `worldLight()`, `spotLight()` and `pointLight()`.

We will be using `pointLight()` to start with, as it's a nice and simple one to understand.

Type, or copy and paste the following code into the **FUZE⁴ Nintendo Switch** code editor.

```
1. obj = placeObject( cube, { 0, 0, 0 }, { 1, 1, 1 } )
2. setObjectMaterial( obj, red, 1, 1 )
3. setCamera( { 0, 0, 10 }, { 0, 0, 0 } )
4. pointLight( { 0, 4, 0 }, white, 100 )
5.
6. loop
7.   clear()
8.   drawObjects()
9.   update()
10. repeat
```

As you can see, the project is exactly the same but with one extra line. Here's the line we're talking about:

```
4. pointLight( { 0, 4, 0 }, white, 100 )
```

`pointLight()`

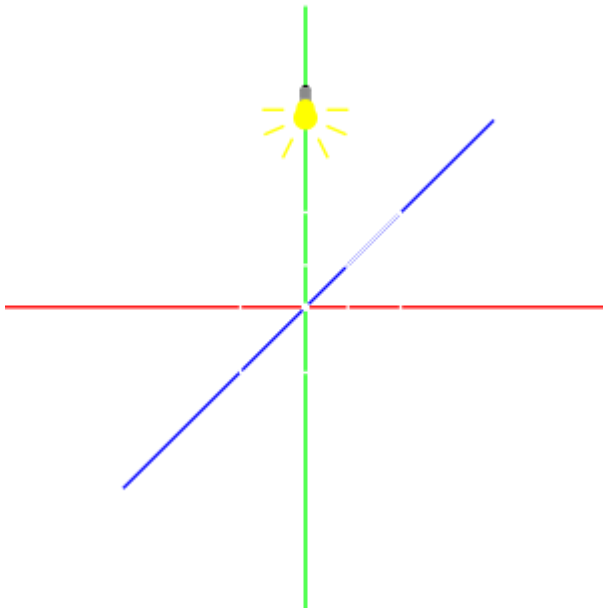
`pointLight()` creates a pinpoint light in a position, which radiates outward in all directions.

Think of it like a lightbulb!

Let's take a look at those **arguments**. The first one should be familiar to you by now! This is a **vector** to describe the position of the light in 3D world space.

Notice that the **y** element of the **vector** is set at **4**. With the camera set the way it currently is, this means our light is positioned *above* the cube on the **y** axis.

Take a quick look at the diagram below:



When you run the program, you should see a nice lit surface to the top face of the cube, with the front face slightly darker.

The second **argument** is of course the colour of our light. Nice and simple! This can either be a name of a colour, or an **RGBA vector**

Finally, we have the light intensity. We have set this to 100 for quite a bright light with a strong effect.

Let's try moving the light around a little to see the effects.

Change your `pointLight()` line to look like this:

```
4. pointLight( { 8, 4, 0 }, white, 200 )
```

Notice we have changed the **x** element of the position **vector** to **8**, meaning that from where our camera is looking, we should see the cube illuminated from the **right**.

See? Let's try it from the left, by the same amount:

```
4. pointLight( { -8, 4, 0 }, white, 200 )
```

Remember, the centre of the 3D world space is at $\{0, 0, 0\}$, so we must use a negative number if we want to move the light to the left.

To really visualise what's happening here, let's add some **Joy-Con** control to the position of the light on the **x** axis.

We'll need to make a couple of changes to do this. In order to change the position of our light during the **loop**, we must use a new **function** called `setLightPos()`. We must also use a **variable** to store the position of the light on the **x** axis, which we'll call **x**. Then it's just a couple of simple **if statements** to change the **x variable** and we'll be done!

Edit your code, or copy and paste the following into the **FUZE⁴ Nintendo Switch** code editor.

```

1. obj = placeObject( cube, { 0, 0, 0 }, { 1, 1, 1 } )
2. setObjectMaterial( obj, red, 1, 1 )
3. setCamera( { 0, 0, 10 }, { 0, 0, 0 } )
4. light = pointLight( { 0, 4, 2 }, white, 100 )
5. x = 0
6.
7. loop
8.     clear()
9.     j = controls( 0 )
10.    setLightPos( light, { x, 4, 2 } )
11.    if j.left then
11.        x -= 0.2
12.    endif
13.    if j.right then
14.        x += 0.2
15.    endif
16.    drawObjects()
17.    printat( 0, 0, x )
18.    update()
19. repeat

```

Let's take a quick look at that new function. First, check out the difference on line 4.

```

4. light = pointLight( { 0, 4, 2 }, white, 100 )

```

We are now using a **variable** to store our light. We've named this **variable** `light` for obvious reasons!

Now we have a **variable** for our light, we can use this in the `setLightPos()` **function**.

```

10.    setLightPos( light, { x, 4, 2 } )

```

As you can see, the first **argument** is the **variable** name of the light we want to move.

The second **argument** is a **vector** to describe the new position of our light.

Notice we have used the **variable** `x` in the `x` axis element of the **vector** because we want this to change during the **loop**.

We have also increased the position of our light on the `z` axis, moving the light slightly closer to the camera. This will make the effect of the light more visible, as it will also hit the front face of the cube.

Finally, line 17 is a `printAt()` **function** to show us the value of our `x` **variable**.

Using the information in this tutorial, can you add **directional button** controls to change the position on the `z` axis too?

Check out the `controls()` **function** Help Page if you need help with the names of the controls! You can find it just [here](#).

Functions and Keywords used in this Tutorial

clear(), controls(), drawObjects(), else, endif, if, loop, placeObject(), printAt(), repeat,
setLightPos(), setObjectMaterial(), setCamera(), then, update()

TUTORIALS

3D Tutorial 3: Rotation

Onwards and upwards! Well... Not upwards. As you know, upwards depends on where our camera is facing!

Before we jump into controlling a camera in 3D space, we're going to look at something really quite awesome which will bring our 3D project to life.

We're back to our original cube project again! Below is the project we'll begin with, it should look very familiar by now!

Type, or copy and paste the following into the **FUZE⁴ Nintendo Switch** code editor.

```
1. obj = placeObject( cube, { 0, 0, 0 }, { 1, 1, 1 } )
2. setObjectMaterial( obj, red, 1, 1 )
3. setCamera( { 0, 5, 10 }, { 0, 0, 0 } )
4. pointLight( { 0, 4, 0 }, white, 100 )
5.
6. loop
7.   clear()
8.   rotateObject( obj, { 0, 1, 0 }, 1 )
9.   drawObjects()
10.  update()
11. repeat
```

rotateObject()

Our new line in question is line 8.

```
8.   rotateObject( obj, { 0, 1, 0 }, 1 )
```

The `rotateObject()` **function** is used to rotate an object, just as the name implies!

The first **argument** is the name of the **variable** which stores our shape. Ours is called `obj`.

The second **argument** is a **vector** which describes the axis or axes of rotation. In our example, we are going to rotate the cube around the **y** axis only, so the **y** element of this **vector** is **1**, while the rest are **0**.

Lastly, we have the amount of rotation per frame of animation. This **argument** is an amount of **degrees**. As you can see, we have used a **1** here, meaning our cube will rotate by **1** degree every frame.

Run the program to see a lovely spinning cube.

Alright! Now we've got that, let's experiment with the other axes.

To make an object rotate in the opposite way, simply use a negative number in the **vector**.

For example:

```
8. rotateObject( obj, { -1, 1, 1 }, 1 )
```

In the above, we are rotating negatively on the **x** axis and positively on both the **y** and **z** axes.

Rotating multiple shapes

Let's adapt these concepts into a slightly more impressive project. We can store a number of shapes in an **array**, then apply these techniques to all of them using **for loops**.

Type, or copy and paste the code below into the **FUZE⁴ Nintendo Switch** code editor:

```
1. setCamera( { 0, 5, 20 }, { 0, 0, 0 } )
2. pointLight( { 0, 4, 4 }, white, 100 )
3.
4. numShapes = 6
5. array shapes[numShapes]
6.
7. for i = 0 to len( shapes ) loop
8.     shapes[i] = placeObject( pyramid, { -numShapes - 1 + i * 3, 0, 0 }, { 1, 1, 1 } )
9.     setObjectMaterial( shapes[i], fuzeblue, 0, 1 )
10. repeat
11.
12. loop
13.     clear()
14.
15.     for i = 0 to len( shapes ) loop
16.         rotateObject( shapes[i], { 1, 1, 1 }, 1 )
17.         repeat
18.
19.     drawObjects()
20.     update()
21. repeat
```

Running this program will give us **6** spinning pyramids.

We should be familiar with the first couple of lines from the previous tutorials. Let's go over creating the **array** and populating it:

```
4. numShapes = 6
5. array shapes[numShapes]
```

Here we create a **simple one-dimensional array**, with `numShapes` amount of elements. To change the number of shapes on screen, simply change the `numShapes` **variable**!

```
7. for i = 0 to len( shapes ) loop
8.     shapes[i] = placeObject( pyramid, { -numShapes - 1 + i * 3, 0, 0 }, { 1, 1, 1 } )
9.     setObjectMaterial( shapes[i], fuzeblue, 0, 1 )
10. repeat
```

Here, we use a **for loop** to populate the **array** with information. We call the `placeObject()` and `setObjectMaterial()` **functions** for *each element* of the **array**.

Notice the position **vector** on line 8: `{-numShapes - 1 + i * 3, 0, 0}`

The `-numShapes - 1 + i * 3` used in the **x** element of the **vector** will position the shapes differently depending on the number of shapes on screen.

This is designed to space **6** shapes evenly across the screen.

Next up, we have our good old `clear()` and `update()` **loop**, inside which is a **for loop** making the magic happen.

```
15.     for i = 0 to len( shapes ) loop
16.         rotateObject( shapes[i], { 1, 1, 1 }, 1 )
17.     repeat
```

For each shape in our **array**, we call a `rotateObject()` **function**, applying **1** degree of rotation on each axis per frame.

Using the Control Stick to Rotate an Object

Let's adapt this project to control the rotation using our **Joy-Con** control stick.

To achieve this, we'll only need a couple of extra lines, including the good old `controls()` **function**.

```
1.  setCamera( { 0, 5, 20 }, { 0, 0, 0 } )
2.  pointLight( { 0, 4, 4 }, white, 100 )
3.
4.  numShapes = 6
5.  array shapes[numShapes]
6.
7.  for i = 0 to len( shapes ) loop
8.      shapes[i] = placeObject( pyramid, { -numShapes - 1 + i * 3, 0, 0 }, { 1, 1, 1 } )
9.      setObjectMaterial( shapes[i], fuzeblue, 0, 1 )
10. repeat
11.
12. loop
13.     clear()
14.
15.     j = controls( 0 )
16.
17.     for i = 0 to len( shapes ) loop
18.         rotateObject( shapes[i], { 0, 1, 0 }, j.lx )
19.         rotateObject( shapes[i], { 1, 0, 0 }, -j.ry )
20.     repeat
21.
22.     drawObjects()
23.     update()
24. repeat
```

Very cool!

We simply call the `controls()` **function**, assign the result to a **variable**, then use the `j.lx` and `j.ry` values in our degrees of **rotation** argument.

Can you add a way of manually rotating the shapes on the **z** axis?

See you in the next tutorial where we'll be looking at creating camera controls!

Functions and Keywords used in this Tutorial

array, clear(), controls(), drawObjects(), loop, placeObject(), pointLight(), repeat, rotateObject(), setObjectMaterial(), setCamera(), update()

TUTORIALS

3D Tutorial 4: Camera Movement

Welcome to the last of the 3D project tutorials!

In this tutorial, we'll be looking at creating a controlled camera in a few different styles. We'll begin very simple, as always. Let's just get some camera movement going.

You know the drill, type or copy and paste the following project into the **FUZE⁴ Nintendo Switch** code editor.

```
1. obj = placeObject( cube, { 0, 5, 0 }, { 1.5, 4, 0.5 } )
2. flr = placeObject( cube, { 0, 0, 0 }, { 10, 0.1, 10 } )
3.
4. setObjectMaterial( obj, grey, 0, 1 )
5. setObjectMaterial( flr, bisque, 0, 1 )
6.
7. pointShadowLight( { 0, 7, 3 }, white, 10, 1024 )
8.
9. loop
10.   clear()
11.
12.   j = controls( 0 )
13.
14.   camPos = { j.lx * 20, j.ly * 20, 20 }
15.   setCamera( camPos, { 0, 0, 0 } )
16.
17.   drawObjects()
18.   update()
19. repeat
```

Run the program to see our rather strange scene. Move the left **Joy-Con** control stick around to change the camera angle.

As you can see, at the start of the program we're creating a simple 3D world using two objects. We have two cubes with unequal dimensions.

```
1. obj = placeObject( cube, { 0, 5, 0 }, { 1.5, 4, 0.5 } )
2. flr = placeObject( cube, { 0, 0, 0 }, { 10, 0.1, 10 } )
```

Our first cube (the floating one) is stored in the **variable** called **obj** and the flat cube object is stored in the **variable** called **flr**.

The material we use for both objects is the same as we have used previously, so no need to go into detail there!

On line 7 we have our `pointShadowLight()` **function** to cast some lovely shadows on the floor. Our light is positioned at `{0, 7, 3}`, which is 7 metres above the centre point on the **y** axis, and 3 metres towards the camera on the **z** axis.

Let's take a look at those camera controls.

First of all, since we'll be using the **Joy-Con** controls, we'll need to call the `controls()` function and assign it to a **variable**:

```
12. j = controls( 0 )
```

In this example, we've called that **variable** `j`. Nice and simple! Alright, on to the camera:

```
14. camPos = { j.lx * 10, j.ly * 10, 20 }  
15. setCamera( camPos, { 0, 0, 0 } )
```

This time around we are calling the `setCamera()` **function** *within* our main **loop**. This means our program will reset the camera position every frame.

In order to *change* the camera position during the **loop**, we must store the camera position in a **variable**. On line 14, we create a **variable** called `camPos` which stores a position **vector**. Notice that in the **x** and **y** elements of the **vector** we have used our controls **variable** to access the left **Joy-Con** control stick **x** and **y** positions. We multiply the result of this by **10** to give an increased effect.

This means our left stick now controls the **x** and **y** axis of the camera position!

You might also notice that in the `setCamera()` **function**, our second **argument** for the camera direction is always `{ 0, 0, 0 }`. The effect of this is that no matter where we move the camera, it will always be pointing at the centre of our 3D world space.

As always, try changing some of these values to see the effect it has on the program!

Making a First-Person Camera

Let's go through the process of creating a real first person camera control. We need to be able to move freely, turning the camera to point at whatever we want.

This is actually more complicated than it might sound, so get ready for some serious code!

The end result of this project will be usable in any game you might want to create, so feel free to take it for your own projects!

Our first task will be to make proper use of the right **Joy-Con** control stick so that we can look around freely.

Here's the full program below. Type or copy and paste it into the **FUZE⁴ Nintendo Switch** code editor.

```
1. obj = placeObject( cube, { 0, 5, 0 }, { 1.5, 4, 0.5 } )  
2. flr = placeObject( cube, { 0, 0, 0 }, { 10, 0.1, 10 } )  
3.  
4. setObjectMaterial( obj, grey, 0, 1 )  
5. setObjectMaterial( flr, bisque, 0, 1 )  
6.  
7. pointShadowLight( { 0, 8, 2 }, white, 10, 1024 )
```

```

8.
9. camPos = { 0, 5, 20 }
10. angle = -90
11.
12. loop
13.     clear()
14.     j = controls( 0 )
15.
16.     angle += j.rx / 4
17.     fwd = { cos( angle ), 0, sin( angle ) }
18.     target = camPos + fwd
19.     setCamera( camPos, target )
20.
21.     drawObjects()
22.     update()
23. repeat

```

As you can see, our first section of code is exactly the same as the previous one. We are creating our little 3D world of two cubes with the same material. We have our same light placement too.

```
9. camPos = { 0, 5, 20 }
```

Our camera position begins at the vector stored in the `camPos` variable.

Let's talk about the new addition to the first section, the `angle` variable.

```
10. angle = -90
```

Our `angle` variable will be used to calculate which direction we are looking. We begin at **-90**. This number is important because of the way we will use the `sin()` and `cos()` **functions** shortly.

Let's jump ahead a little to line 16 in our main **loop**.

```
16.     angle += j.rx
```

This is where we modify the `angle` **variable**. We add the current value of the the right control stick (`j.rx`) to the **variable**. Notice that we are only using the **x** axis of the right Joystick for now. We will begin by looking left and right, up and down can come later!

Now, what are we doing with that **variable**?

```
17.     fwd = { cos( angle ), 0, sin( angle ) }
```

This complex looking bit of code needs some explaining.

We are creating something called a **forward vector**. This is a special type of directional **vector** which tells us which direction is *forwards*, hence the name!

We are using the `sin()` and `cos()` functions to calculate the *direction we want the camera to point* based on the value of the `angle` **variable**.

As we change the position of the right **Joy-Con** control stick, the value of the `angle` **variable** changes, and the calculation on line 17 gives us a different result.

Notice that the **y** element of our **forward vector** is at **0** for now. This will change when we add the ability to look up and down.

Still with us? Well done!

```
19.     target = camPos + fwd
20.     setCamera( camPos, target )
```

Here we create a new **variable** called `target`. This **variable** is used in the `setCamera()` **function** on line 19. Rather than having a fixed target like before (which was `{ 0, 0, 0 }` if you remember), we are *constantly recalculating* the target by *adding* the **forward vector** to the current camera position and setting it as the new target.

Once we've done that, we call the `drawObjects()` and `update()` **functions** as usual, then close the **loop**.

Run the program! Your right control stick will now move the camera direction left and right. If that's all working nicely, let's move on to vertical movement!

Vertical Camera Movement

In order to add vertical camera movement to our project, we actually only need a couple of lines. Take a look at the lines below and add them at the designated line numbers.

First we'll need a **variable** to store the vertical angle of the camera. We'll call this `lookHeight`. It must be defined outside of the main **loop**:

```
11. lookHeight = 0
```

Now we need to modify this **variable** inside the main **loop** using the right control stick:

```
18.     lookHeight += j.ry / 50
```

We have divided the result of the right control stick by 50 to give a more manageable movement speed.

Lastly, we must use this **variable** in the calculation of our forward vector:

```
19.     fwd = { cos( angle ), lookHeight, sin( angle ) }
```

Alright! Let's take a look at the program in full so far:

```
1. obj = placeObject( cube, { 0, 5, 0 }, { 1.5, 4, 0.5 } )
2. flr = placeObject( cube, { 0, 0, 0 }, { 10, 0.1, 10 } )
3.
4. setObjectMaterial( obj, grey, 0, 1 )
5. setObjectMaterial( flr, bisque, 0, 1 )
6.
7. pointShadowLight( { 0, 7, 3 }, white, 30, 1024 )
8.
9. camPos = { 0, 5, 20 }
10. angle = -90
11. lookHeight = 0
12.
```

```

13. loop
14.     clear()
15.     j = controls( 0 )
16.
17.     angle += j.rx
18.     lookHeight += j.ry / 50
19.     fwd = { cos( angle ), lookHeight, sin( angle ) }
20.     target = camPos + fwd
21.     setCamera( camPos, target )
22.
23.     drawObjects()
24.     update()
25. repeat

```

Run the program and use the right control stick to look around freely!

Adding Walking Movement

Make sure our projects are matching and we'll add the last section of code. We need to be able to walk around!

To be able to walk around with the left control stick, we only need to add three lines of code. They must be placed **between** the `fwd = {cos(angle), lookHeight, sin(angle)}` and `target += camPos + fwd` lines. Let's go through them:

```

20.     side = cross( fwd, { 0, 1, 0 } )
21.     camPos += side * j.lx / 4

```

These two lines of code allow us to move left and right using the left control stick.

We create a **variable** called `side` which stores the result of a calculation called the cross product. The `cross()` **function** takes two **vectors** and gives an angle perpendicular to both. Notice we are using our forward vector and a vector of `{0, 1, 0}` which is directly upward. The angle perpendicular to these is horizontally across the **x** axis!

We then add the result of this calculation to our `camPos` **variable** multiplied by the left control stick x axis value, allowing us to move freely along the x axis, always keeping our camera pointed towards where we want.

Let's add the line to allow us to move along the z axis:

```

22. camPos += normalize( { fwd.x, 0, fwd.z } ) * j.ly / 4

```

Since we only want to move along the ground rather than flying, we must normalize the vector and remove the y component. We then increase the `camPos` **variable** by this result, multiplied by our left control stick y axis value. Again, we divide by 4 to create a more manageable movement speed.

Let's take a last look at the whole project including these changes for reference:

```

1. obj = placeObject( cube, { 0, 5, 0 }, { 1.5, 4, 0.5 } )
2. flr = placeObject( cube, { 0, 0, 0 }, { 10, 0.1, 10 } )
3.

```

```

4. setObjectMaterial( obj, grey, 0, 1 )
5. setObjectMaterial( flr, bisque, 0, 1 )
6.
7. pointShadowLight( { 0, 7, 3 }, white, 30, 1024 )
8.
9. camPos = { 0, 5, 20 }
10. angle = -90
11. lookHeight = 0
12.
13. loop
14.   clear()
15.   j = controls( 0 )
16.
17.   angle += j.rx
18.   lookHeight += j.ry / 50
19.   fwd = { cos( angle ), lookHeight, sin( angle ) }
20.   side = cross( fwd, { 0, 1, 0 } )
21.   camPos += side * j.lx / 4
22.   camPos += normalize( { fwd.x, 0, fwd.z } ) * j.ly / 4
23.   target = camPos + fwd
24.   setCamera( camPos, target )
25.
26.   drawObjects()
27.   update()
28. repeat

```

If you'd like to fly around in your 3D scene, it's actually a little simpler!

```
22. camPos += fwd * j.ly / 4
```

Simple as that! We do not remove the **y** component of the vector and therefore do not need to normalize. Simply add the whole **fwd vector** to the camera position.

Run the program and take a walk around! Congratulations, you've programmed a first-person camera!

Why not build the scene into something more exciting? Use more **placeObject()** **functions** just as we did at the start to create more objects, then the **drawObjects()** **function** will take care of the rest!

Well done. You've complete the 3D Tutorials!

Functions and Keywords Used in this Tutorial

[clear\(\)](#), [cos\(\)](#), [cross\(\)](#), [drawObjects\(\)](#), [loop](#), [normalize\(\)](#), [placeObject\(\)](#), [pointShadowLight\(\)](#), [repeat](#), [setCamera\(\)](#), [setObjectMaterial\(\)](#), [sin\(\)](#), [update\(\)](#)

TUTORIALS

Basic Game Tutorial: 0 - Introduction

Hello again!

In these tutorials, we'll be covering how to begin making your very own game.

It is strongly recommended that you have completed *at least* the first 5 regular tutorials before going into this project. We'll be using concepts which might seem a little tricky for an absolute beginner. If you're happy with **loops, variables, if statements, arrays, for loops, functions, structures** and how the **screen** works, then carry on!

In the tutorials so far, we've been using basic shapes to illustrate the core concepts of programming. Here, we'll be stepping things up a notch and using a small selection of the vast amount of assets **FUZE⁴ Nintendo Switch** has to offer.

Before we get started, let's quickly outline the basic steps we'll be taking in the upcoming parts to this project.

Part 1 - Drawing the Background

In the first part of the Basic Game Tutorial, we'll be covering how to draw a background image to the screen, how to make the game switch correctly between *docked* and *undocked* modes, and how to create the beginning of a 2D camera.

Part 2 - Creating a Level

In the second part, we'll be covering how to use the `drawSheet()` **function** to access a tilesheet to design and draw a level of your very own to the screen.

Part 3 - Drawing and Animating the Player

In part 3, we'll be using the `drawSheet()` **function** once again to draw and animate the player. This tutorial will introduce the concept of a *state machine* to keep track of the player characters current *state*.

Part 4 - Collision

In order to move around on our level, jump onto platforms and tragically fall down the gaps, we need to interact with our level. In part 4, we'll cover how to make the player character interact with the level we've created.

Part 5 - Movement

Once our character interacts with the level properly, we will cover adding movement controls to the game, with the ability to walk and jump.

Part 6 - Items

Now that we've got the foundations ready, we can start adding some simple items to the game!

Part 7 - Enemies

Finally, we'll be adding a simple slime enemy to our platformer. We will learn how to make enemies move around on platforms, how to animate them and most importantly, how to jump on them!

Part 8 - Customise

Now our game is complete! Or is it? It's up to you now to take this project as far as you like! Use this part of the project to see how to add your own ideas to the game.

Let's go!

Now we've outlined the structure of what we'll be learning and why, let's dive straight in to the first part. See you there!

TUTORIALS

Basic Game Tutorial 1: Drawing the Background

Hello! We meet again. In this part of the Game Tutorial, we'll be drawing a background image on screen. To achieve this, we'll be using a couple of techniques which are vital to learn for creating your own games.

By completing this tutorial, you will have learned a solid foundation to start creating your own games. Once you've got your head around this, you should try to do the same with different **FUZE⁴ Nintendo Switch** assets!

Creating the Image File

Let's get straight into this. In order to draw a background image on screen, we'll need a background image! Open a new project file and enter the single line of code below:

```
1. background = loadImage( "Kenney/backgrounds", false )
```

There we have it! We have created a **variable** called `background` which stores the image file we want. We've used the `loadImage()` **function** to do this, which has two **arguments**. The first is the location and name of the image file in speech marks. The second is a **true** or **false** switch for a filter. We will not be using a filter on our graphics, so this is set to **false**.

Now we've done this, we can freely use the **variable** called `background` in our program if we want to use the background image.

Setting up Camera Variables

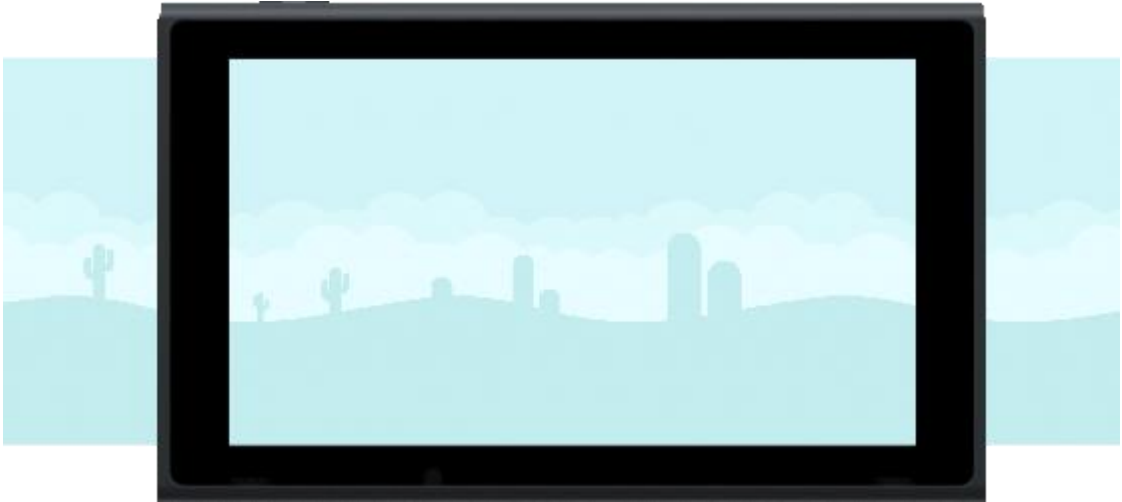
In platform games, we usually have a background image and level which is *larger than the screen*. As we move our character along the level, the screen moves with us, scrolling the background and level to the left.

Take a look at the image below to help picture this in your mind:



As you can see, we have a background image which is larger than the console screen.

If we were to move our camera (the screen) along, the background image would appear to be moving to the left:



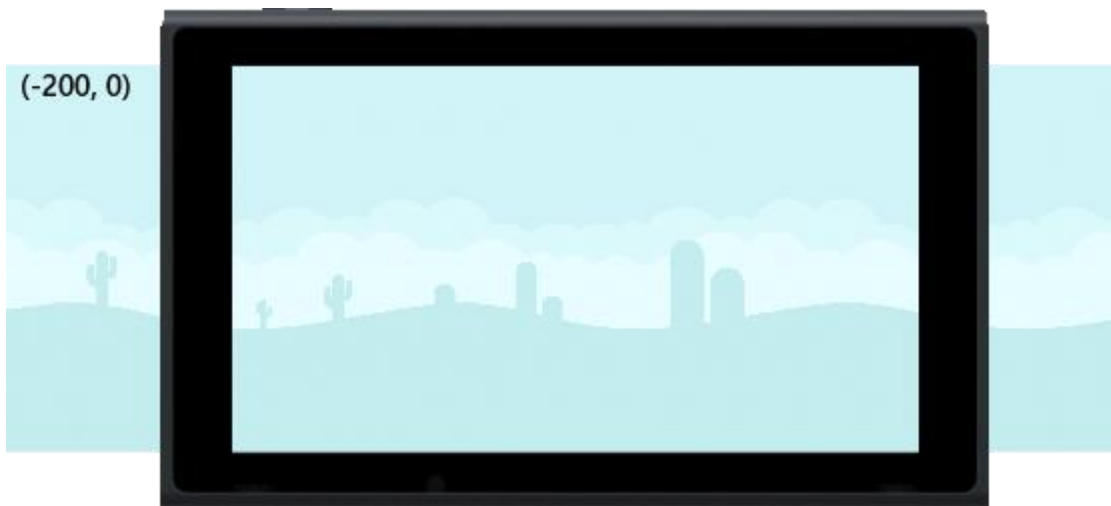
In fact, our background image is *not* moving, but our *camera* (screen) is!

Let's take a look at the first picture again, but with some coordinates. Imagine we want to draw the background image using **x** and **y** coordinates. We know the top left hand corner of the screen is **0** on the **x** axis, and **0** on the **y** axis. So our coordinates for drawing the background on screen would be **(0, 0)**:



All good?

Let's say we move the camera (screen) 200 pixels to the right. To make our background image move in the opposite direction by the same amount, we must now draw it at **(-200, 0)**:



If we want this to change while the program is running, we must use **variables** to store the camera position.

With that explained, we can add two **variables** to our program. Make the following changes to your code:

```
1. background = loadImage( "Kenney/backgrounds", false )
2.
3. screenX = 0
4. screenY = 0
```

There we go! We will use these **variables** when it's time to draw the background on screen.

Main Game Loop

Let's start our main game **loop**.

As we know, all games must use a **loop** which clears and updates the screen for us to see movement. Let's begin with that. Add the following lines to your program:

```
1. background = loadImage( "Kenney/backgrounds", false )
2.
3. screenX = 0
4. screenY = 0
5.
6. loop
7.   clear()
8.
9.   update()
10. repeat
```

Simple enough! Our **loop** doesn't do anything *just yet*, other than clearing and updating the screen, but this is where we *must* start when writing a visual program with animation.

Anything we want to happen *during* our game must now be added to the main **loop**.

Dynamic Scaling

Wow that's a tricky couple of words... What does it mean?

Well, your Nintendo Switch is very clever. When you put the console in the dock, it displays on whatever TV screen you might have it plugged into.

In the **screen** tutorial project, we mentioned that the console's screen is 1280 pixels on the **x** axis, and 720 on the **y** axis. We can access the width and height of the screen with the `gwidth()` and `gheight()` **functions**.

Those numbers might *change* when we put the console into the dock and display it on a TV screen.

Whilst we are making our game project, we want to be able to *detect* the width and height of screen *while the game is running*. This way, we can freely use the console in handheld or docked mode, and the game will *automatically scale* our graphics to look awesome no matter what!

To achieve this, we will create two **variables** *inside* the main game **loop**. Add the following changes to your code:

```
1. background = loadImage( "Kenney/backgrounds", false )
2.
3. screenX = 0
4. screenY = 0
5.
6. loop
7.   clear()
8.
9.   screenW = gwidth()
10.  screenH = gheight()
11.
12.  update()
13. repeat
```

There we go!

No we have all the information we could ever need for our screen. We have a screen position in the `screenX` and `screenY` **variables**, and screen dimensions stored in the `screenW` and `screenH` **variables**!

All that's left is to draw the background image!

Drawing the Background Image

It might seem like an awful lot of setting up just to draw an image on screen, but doing things this way will mean we won't have to come back and make lots of changes later. This way, we can move forward and our brilliant program will take care of everything for us.

All we need now is one line of code to draw the background image. Add the following change to your code:

```
1. background = loadImage( "Kenney/backgrounds", false )
2.
```

```
3. screenX = 0
4. screenY = 0
5.
6. loop
7.   clear()
8.
9.   screenW = gwidth()
10.  screenH = gheight()
11.
12.  drawImage( background, -screenX, -screenY )
11.
12.  update()
13. repeat
```

We use the `drawImage()` **function** to draw the image stored in the `background` **variable** to the `x` and `y` positions stored in `screenX` and `screenY`.

We use minus in front of the `x` and `y` positions so that our background moves to the left by the same amount that our screen moves to the right. Now we can freely move the screen any amount in any direction and our background will always move correctly!

Scaling the Background Image

Run the program to see the background image.

Ah... It appears we have a problem...

Right now, our background looks like this:



But it *should* look like this:



Our background image is quite tall. Much taller than our screen is. Because of this, we won't see our image properly. We must use a secret extra feature of the `drawImage()` function. It's an extra argument for the *scale* of the image.

We need to scale our image depending on the height of the screen.

If we divide the size of the screen by the height of the image, we will get our exact number to scale by.

We have a very clever **function** to find the size of an image, it's called `imageSize()`!

When we put an image in the brackets of the `imageSize()` function, we can access the width or height of that image with `.x` and `.y`.

With this in mind, let's make the last change to our `drawImage()` line:

```
1. background = loadImage( "Kenney/backgrounds", false )
2.
3. screenX = 0
4. screenY = 0
5.
6. loop
7.   clear()
8.
9.   screenW = gwidth()
10.  screenH = gheight()
11.
12.  drawImage( background, -screenX, -screenY, screenH / imageSize( background ).y )
11.
12.  update()
13. repeat
```

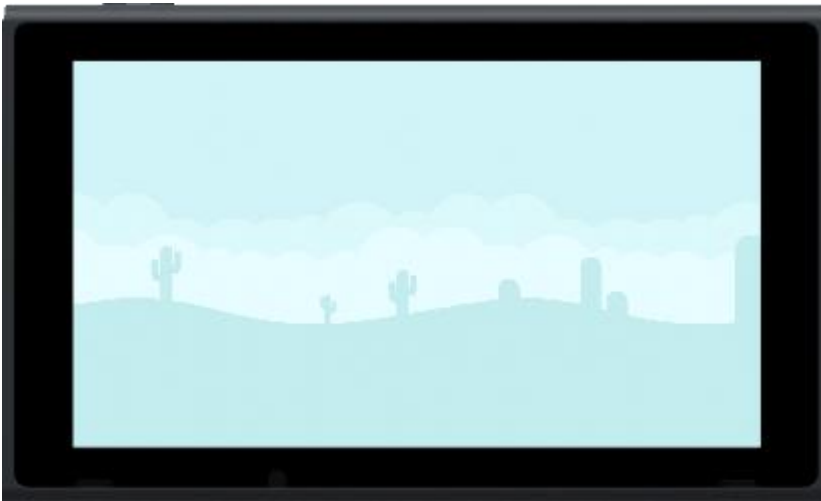
Notice that our `drawImage()` line now has an extra **argument** for a total of three. The *scale* argument reads: `screenH / imageSize(background).y`. This takes the height of the screen stored in the `screenH` variable and divides it by the height of the background image. We then use this result as a *scale* to multiply the background image size, giving us a perfectly fitting image no matter what screen we are using! Neat!

End Result

Before we move on to the next part of our Game Tutorial project, let's double check that we have everything correct so far. Your program should look exactly like this:

```
1. background = loadImage( "Kenney/backgrounds", false )
2.
3. screenX = 0
4. screenY = 0
5.
6. loop
7.   clear()
8.
9.   screenW = gwidth()
10.  screenH = gheight()
11.
12.  drawImage( background, -screenX, -screenY, screenH / imageSize( background ).y )
11.
12.  update()
13. repeat
```

When we run the program, we will just see the background image on screen in the correct dimensions. It should look something like this:



If this is how your screen looks too, awesome! We're ready to take the next step.

Functions and Keywords used in this tutorial

[clear\(\)](#), [drawImage\(\)](#), [gHeight\(\)](#), [gWidth\(\)](#), [loadImage\(\)](#), [loop](#), [repeat](#), [update\(\)](#)

TUTORIALS

Basic Game Tutorial 2: Creating a Level

Hello again! Glad to see you haven't given up yet!

We've got our background image, now it's time to create a level for our platform game. Before we go adding all kinds of complicated things, let's focus on just the basics.

Using a Tilesheet

We want a level to have something to walk on, some pits to fall into, and perhaps a couple of platforms for us to jump on.

When game designers create a level, they use something called a *tileset* or *tilesheet*. This is an image file which contains lots of different *tiles* - the building blocks for a level. We'll be using more of Kenney's awesome artwork for our level. Take a look at the image of a portion of the *tilesheet* we'll be using below:



Wow! As you can see, this tilesheet contains all of the building blocks we would need for a level. We even have lots of different themes in the same sheet! This means you could add some very different sections to your game once we're finished!

The important thing to understand here is that *each tile has a number*. Let's take a closer look at a couple of those tiles from the top left corner.



The tile numbers begin at 0 and move up as we go. Imagine a grid around each tile in the tilesheet to help picture this.

The tiles we will be using to draw the level are:



We have 5 tiles, each with different numbers. We need to use an **array** to arrange these tiles into the level we want, but before we do that, we can make our lives a little easier by storing these tiles into an **array**.

Enough talk, let's get started!

Loading the Tilesheet Image

Just like before, we need to use the `loadImage()` **function** to load and store the tilesheet in a variable. We'll be doing this right at the start of our program. Below, we've added a new line on line 2. Add this to your code:

```
2. tilesheet = loadImage( "Kenney/superPlatformPack", false )
```

We define a **variable** called `tilesheet` and use the `loadImage()` **function** to store the tilesheet. Again, we want no filter applied to this so the second argument is `false`.

Done!

Creating a Look-Up Table

Earlier, we mentioned that we could make things easier for ourselves by storing all the tiles we want in a simple **array**. Then, when we create our level **array**, rather than having a huge amount of 3 digit numbers all over the place, we will have nice single digit numbers instead.

An **array** used in this way is called a *look-up table*. It's a table that we use to look things up!

Let's create that look-up table now. We'll call it `tiles`.

We'll add this on line 7. Here's how the first part of your project should look:

```
1. background = loadImage( "Kenney/backgrounds", false )
2. tilesheet = loadImage( "Kenney/superPlatformPack", false )
3.
4. screenX = 0
5. screenY = 0
6.
7. tiles = [ 121, 138, 128, 129, 130 ]
```

That's it! We've stored the numbers for the tiles we want in a simple array.

With this, we can easily access any tile number from the table, and changing the tiles later will be a breeze!



Almost every old 2D game works in this way. You can imagine any 2D level as a grid of numbers, each number representing a different tile to draw.

With -1's in all the empty tiles, we have an easy way to tell if any particular tile is something we want to draw. It will also be used when the time comes to *collide* with the level, checking to see if a tile is empty and can be moved through.

Anyway, let's get back to some coding and create our array! It is probably a good idea to copy and paste this section of code into your project. Entering this whole array manually is quite long-winded!

```

9. level = [
10.  [ -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 ],
11.  [ -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 ],
12.  [ -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, 2, 3, 4, -1, -1, -1, -1, -1, -1, -1, -1 ],
13.  [ 1, 1, 1, 1, -1, 1, 1, 1, 1, 1, 1, 1, -1, -1, -1, 1, 1, 1, 1, 1, 1, 1 ],
14.  [ 0, 0, 0, 0, -1, 1, 0, 0, 0, 0, 0, 0, 1, -1, -1, -1, 0, 0, 0, 0, 0, 0 ],
15.  [ 0, 0, 0, 0, -1, 1, 0, 0, 0, 0, 0, 0, -1, -1, -1, -1, 0, 0, 0, 0, 0, 0 ]
16. ]

```

There we go! It's very easy to get confused by such a huge amount of numbers, but now we know exactly what they are for. You can almost *see* the level in the numbers!

Think of this **array** as 6 rows of 26 columns. This will help to understand things when we come to drawing the level.

You might notice our **array** is *longer* than the example picture. That's as it should be! We want our level to be *longer* than the screen, when we move forward in the game, the rest of the level will be revealed to us!

Before we go ahead and draw the level in our main **loop**, we need to set up a couple of very helpful **variables**. They will store the *total number of tiles* in the height of the screen, and the *offset* we will use when drawing our level. Add the following two lines to your code:

```
18. levelHeight = 12
19. levelOffset = levelHeight - len( level )
```

Our screen will be 12 tiles tall, so `levelHeight` is 12. Our level array is only 6 rows high, so without an *offset* this will result in something not quite right when we draw it. You'll see!

To store the correct *offset*, we take the `levelHeight` **variable** and subtract the number of rows in our level **array**. Rather than use the number 6, we have used the `len()` **function**. This means we can add more to the level later and everything will be taken care of!

Another very useful thing to have would be a **variable** to store the size of a single tile in pixels. We'll be using this all over the place in the program, so we should make it a **global variable** (outside of any **loops** or **functions** so it can be accessed from anywhere in the program.). Because the size of one tile in pixels will depend on whether the console is docked or undocked, we must update this **variable** in the main **loop**. For now, we can simply define it as 0.

```
20. tileSize = 0
```

Alright, enough of that. Let's draw this level. This next part will take place *inside* the main game **loop**. To begin with, we must create a **variable** to store the **scale** multiplier for drawing to the screen. The actual size of the tiles from our sheet are very small indeed! If we want them to look good on screen, we must multiply their size by a scale.

This `scale` **variable** will be used again and again throughout our program:

```
22. loop
23.     clear()
24.
25.     screenW = gwidth()
26.     screenH = gheight()
27.     scale = screenH / ( tileSize( tilesheet, 121 ).y * levelHeight )
28.
29.     drawImage( background, -screenX, -screenY, screenH / imageSize( background ).y )
30.
31.     update()
32. repeat
```

We create the `scale` **variable** on line 27 above. To calculate the scale, we take the height of the screen in pixels (`screenH`) and divide it by the height of a level tile (`tileSize(tilesheet, 121)`). This tells us how many single tiles will fit into the height of the screen. We then multiply the result by our desired level height.

Now let's use that `scale` to update the `tSize` **variable**, giving us the size of our scaled up tiles in pixels:

```
22. loop
23.     clear()
24.
25.     screenW = gwidth()
26.     screenH = gheight()
27.     scale = screenH / ( tileSize( tilesheet, 121 ).y * levelHeight )
28.     tileSize = scale * tileSize( tilesheet, 121 ).y
29.
30.     drawImage( background, -screenX, -screenY, screenH / imageSize( background ).y )
31.
```

```
32.     update()
33. repeat
```

On line 28 above we multiply the original size of a level tile (`tileSize(tilesheet, 121)`) by our **scale variable** to give us the exact height of a scaled-up tile. Very useful indeed!

Drawing the Level Using a For Loop

Here is the clever part. Rather than using a different `drawSheet()` **function** for each tile, we will use a **for loop** to count over the **level array** and call the `drawSheet()` **function** for each number.

Since we are using a *two-dimensional array*, we need something fancy called a **nested for loop**. It's really just a **for loop** inside a **for loop**!

Add the lines 32 to 40 below to your project:

```
22. loop
23.     clear()
24.
25.     screenWidth = gwidth()
26.     screenHeight = gheight()
27.     scale = screenHeight / ( tileSize( tilesheet, 121 ).y * levelHeight )
28.     tileSize = scale * tileSize( tilesheet, 121 ).y
29.
30.     drawImage( background, -screenX, -screenY, screenHeight / imageSize( background ).y )
31.
32.     for row = 0 to len( level ) loop
33.         for col = 0 to len( level[0] ) loop
34.             if level[row][col] >= 0 then
35.                 x = col * tileSize
36.                 y = ( row + levelOffset ) * tileSize
37.                 drawSheet( tilesheet, tiles[level[row][col]], x, y, scale )
38.             endif
39.         repeat
40.     repeat
41.
42.     update()
43. repeat
```

drawSheet()

The `drawSheet()` **function** is something we will be using a lot, so it's very useful to fully understand it. Let's just quickly cover what the **arguments** are:

```
drawSheet( file, tile, xPosition, yPosition, scale )
```

The first argument is the file we want to draw *from*.

The second is the *tile number* we want to draw.

Next, we have the **x** and **y** screen positions we want to draw to.

Last is the scale multiplier applied to the tile drawn.

The For Loop

Let's take a closer look at that **for loop** to really understand what's happening.

```
32.     for row = 0 to len( level ) loop
33.         for col = 0 to len( level[0] ) loop
34.             if level[row][col] >= 0 then
35.                 x = col * tsize
36.                 y = ( row + levelOffset ) * tsize
37.                 drawSheet( tilesheet, tiles[level[row][col]], x, y, scale )
38.             endif
39.         repeat
40.     repeat
```

In these **loops** we create two **variables** called `row` and `col`, to count rows and columns. They count from 0 to the *length* of the dimensions of our level **array**. Our level **array** is 6 rows of 26 columns.

The inside **for loop** (lines 33 to 39) count over each *column*, for a total of 26 repetitions. The outside **for loop** counts over each *row*, for a total of 6 repetitions. This covers every single position in our level array, no matter how long we make it.

For each repetition, we check if the position in the level array which corresponds to the number in the `row` and `col` **variables** is *greater than or equal to* 0. If it is, we use the `drawSheet()` **function** to draw the tile from the `tiles` **array** on screen.

Let's look at an example. Imagine that `row = 4` and `col = 9`.

The **if statement** on line 34 would read:

```
34. if level[4][9] >= 0 then
```

If we take a look at our level **array**, we can see that column 9 of row 4 is a 0. This is indeed *greater than or equal to* 0! So, with that check complete, let's put those values into the `drawSheet()` line:

```
37. drawSheet( tilesheet, tiles[level[9][4]], x, y, scale )
```

We know that `level[9][4]` is a 0, so really the line looks like this:

```
drawSheet( tilesheet, tiles[0], x, y, scale )
```

Looking at our `tiles` array, we can see that `tiles[0]` is the number 121. So *really*, the line looks like this:

```
drawSheet( tilesheet, 121, x, y, scale )
```

This `drawSheet()` **function** is repeated for a total of 156 times! Each time, the values in the **function** are different, drawing the correct tile from our **array** at the correct positions.

End Result

Congratulations! You've made it to the end of part 2 of the basic game tutorial. We should have our level being drawn beautifully on the screen. Try putting your Nintendo Switch into the dock while

connected to a TV screen, you'll see our program scales the level and background perfectly to look awesome no matter what screen we use!

Just to double check, your entire program should now look like this:

```
1. background = loadImage( "Kenney/backgrounds", false )
2. tilesheet = loadImage( "Kenney/superPlatformPack", false )
3.
4. screenX = 0
5. screenY = 0
6.
7. tiles = [ 121, 138, 128, 129, 130 ]
8.
9. level = [
10.  [ -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 ],
11.  [ -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 ],
12.  [ -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, 2, 3, 4, -1, -1, -1, -1, -1, -1, -1 ],
13.  [ 1, 1, 1, 1, -1, -1, 1, 1, 1, 1, 1, -1, -1, -1, 1, 1, 1, 1, 1, 1, 1, 1 ],
14.  [ 0, 0, 0, 0, -1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1, -1, -1, -1, 0, 0, 0 ],
15.  [ 0, 0, 0, 0, -1, -1, 0, 0, 0, 0, 0, 0, 0, 0, -1, -1, -1, -1, -1, 0, 0, 0 ]
16. ]
17.
18. levelHeight = 12
19. levelOffset = levelHeight - len( level )
20. tSize = 0
21.
22. loop
23.   clear()
24.
25.   screenW = gwidth()
26.   screenH = gheight()
27.   scale = screenH / ( tileSize( tilesheet, 121 ).y * levelHeight )
28.   tSize = scale * tileSize( tilesheet, 121 ).y
29.
30.   drawImage( background, -screenX, -screenY, screenH / imageSize( background ).y )
31.
32.   for row = 0 to len( level ) loop
33.     for col = 0 to len( level[0] ) loop
34.       if level[row][col] >= 0 then
35.         x = col * tSize
36.         y = ( row + levelOffset ) * tSize
37.         drawSheet( tilesheet, tiles[level[row][col]], x, y, scale )
38.       endif
39.     repeat
40.   repeat
41.
42.   update()
43. repeat
```

Make sure we're matching up to your project perfectly, and then we'll see you in the next tutorial. Let's put our character on the screen!

Functions and Keywords used in this tutorial

[clear\(\)](#), [drawImage\(\)](#), [else](#), [endif](#), [for](#), [gHeight\(\)](#), [gWidth\(\)](#), [if](#), [len\(\)](#), [loadImage\(\)](#), [loop](#), [repeat](#), [tileSize\(\)](#), [then](#), [to](#), [update\(\)](#)

TUTORIALS

Basic Game Tutorial 3: The Character

Alright! Here we go.

It's time to bring this game project to life with a character. In this tutorial, we'll be putting our character on screen and making it interact with the level. We want it to walk on the platforms and fall off the edges!

Setup

Before we get our character on screen, we need to do some setup. First, we must load a the character tilesheet:

```
1. background = loadImage( "Kenney/backgrounds", false )
2. tilesheet = loadImage( "Kenney/extraPlatformPack", false )
3. chrSheet = loadImage( "Kenney/characters", false )
```

We're using the name `chrSheet` for the **variable** which stores the image we need.

Now let's create some player position **variables**:

```
5. playerX = 0
6. playerY = 0
```

Nice and simple. We'll begin with our character being drawn at the very top left of the screen.

Let's use these **variables** in a `drawSheet()` **function** to actually get our player character appearing on screen.

We'll be adding this line just before the `update()` **function** in our main game **loop** on line 46:

```
46. drawSheet( chrSheet, 96, playerX, playerY, scale )
```

Run your program and we should see our character sitting comfortably in the top left of the screen.

Okay! We're done. Enjoy your new game.

Just kidding. There's lots more to do.

First of all let's address a slight problem. Our character tile is a **different** size to the level building tiles. Because of this, just like our `tSize` **variable**, we'll need a player size **variable** to make things easier for us later.

Add the following line to your code, just beneath the `tSize` **variable** in the main **loop**:

```
33. pSize = tileSize( chrSheet, 96 ) * scale
```


Just like with the `tSize` **variable**, we take the tile size of the character tile and multiply it by the `scale` **variable**.

Alright, let's move on!

If our game is going to be playable, we need our character to fall until he lands safely on the ground. When he's standing on a platform, he should not fall.

Achieving this is actually quite tricky. However, once complete, we will be able to freely create new parts to the level and it will work just fine!

Gravity and Velocity

You might have noticed that on Earth, when we jump into the air, we unfortunately come back down again. This is because of a rather inconvenient thing called *gravity*.

When we are in the air, gravity pulls us towards the centre of the Earth. Thankfully, there is some nice solid ground in the way to stop us going too far.

Velocity is the speed of an object in a direction.

Let's say we drop a stone from the top of a building. The force of gravity is making the stone move faster and faster towards the ground. This is called increasing in *velocity*.

When the stone reaches the ground, its *velocity* becomes 0. It has stopped. However, *gravity* has not changed.

We can simulate the effects of *gravity* and *velocity* in our game code and achieve some awesome things.

Let's create the **variables** we'll use for these effects:

```
8. gravity = 1
9. velocity = 0
```

Done! Now let's use these **variables** to affect the player. We need to add these next two lines just before the `drawSheet()` **function** used to draw the player. To make sure you've got it right, we'll show the end of the main **loop** too:

```
50.     velocity += gravity
51.     playerY += velocity
52.
53.     drawSheet( chrSheet, 96, playerX, playerY, scale )
54.
55.     update()
56. repeat
```

Run the program to see our character plummet straight past the screen! Excellent!

Okay... So we have gravity. Now let's work on actually making a working floor. Before we explain how this works, we need to delete a line of code. Line 51 to be precise. Take a look at how lines 50 onward should look:

```

50.     velocity += gravity
51.
52.     drawSheet( chrSheet, 96, playerX, playerY, scale )
53.
54.     update()
55. repeat

```

We do not always want our character's **y** position to be affected by the **velocity variable** all the time, only when there is no ground beneath them.

What exactly do we mean by ground anyway?

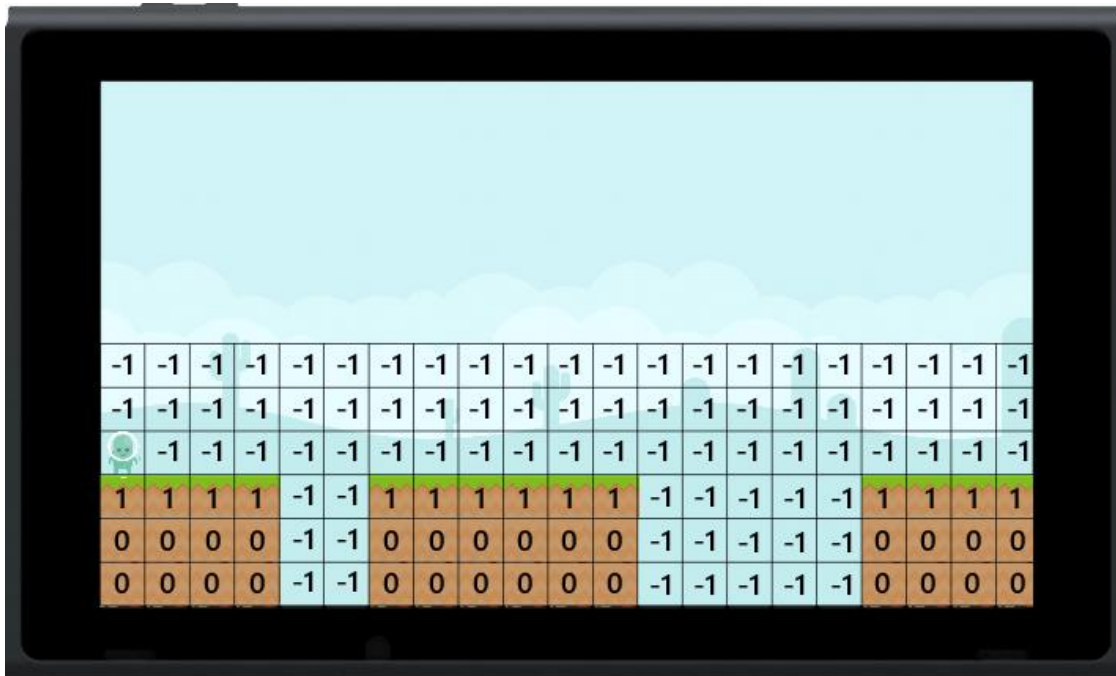
Going from Pixel Co-ordinates to Array Co-ordinates

What we need to do is quite complicated, so strap your focusing hats on.

We need to map our **level array** on to the screen so that it fits correctly in the tiles.

We must check the tile beneath the player to see if it is empty in the **level array**. As long as we have an empty tile beneath us, our **y**. position should be affected by velocity.

Let's return to our picture for a minute to illustrate what we mean:



See our cute character on the left side of the level? In the picture, he is *above* a tile indexed by the number 1. We want him *not* to fall *unless* he is above a -1 tile.

But how do we go from pixel co-ordinates to the co-ordinates of our **array**?

With maths! Hurray...

In all seriousness, this is an incredibly useful technique to learn - with it, you'll be able to create any 2D game with much more confidence.

Figuring out the surrounding tiles is something we'll want to do quite a bit in our project. Not just for falling, but for moving left and right and to interact with items too.

It's the perfect time to write our very own **function** to do just that!

A **function** which does this needs to be given an **x** and **y** position to calculate from. We will *pass* these to the **function** as **variables**.

Remember, we write custom **functions** at the very end of our program:

```
57. function collision( x, y )
58.     tileX = int( x / tileSize )
59.     tileY = int( y / tileSize ) - levelOffset
60.
61.     result = true
62.
63.     if tileY < 0 or tileY >= len( level ) or tileX < 0 or tileX >= len( level[0] ) then
64.         result = false
65.     else
66.         if level[tileY][tileX] < 0 then
67.             result = false
68.         endif
69.     endif
70. return result
```

Now that's a scary bit of code right there! This **function** receives an **x** and a **y** position of the screen, converts them into **array** co-ordinates and finally tells us whether the tile at our **x** and **y** position is something to collide with (**> 0**) or not (**< 0**).

Remember, the tiles in our **level array** are empty if they are a -1. It tells us this with a single **true** or **false variable** called **result**. We can use this **function** to check if a tile is **not** a collision tile with a statement like:

```
if !collision( playerX, playerY ) then
```

(Remember, **!** means **not**)

Here's how the **function** works:

First, we receive an **x** and **y** position. We then create two *local variables* called **tileX** and **tileY** which will be our array co-ordinates.

```
58.     tileX = int( x / tileSize )
59.     tileY = int( y / tileSize ) - levelOffset
```

We take the **x** and **y** positions passed to the **function** and divide them by the **tSize variable** to give us the **tile** coordinates.

We must use the **int()** **function** when we do this because we're looking for a whole number. If our result was a decimal, it would not work properly when used as an index into the level array.

For example, imagine our character is at the screen co-ordinates (450, 560).

We take 450 and divide it by the **tSize variable**. When undocked, the **tSize variable** is 60. 450 divided by 60 is 7.5. With the **int()** **function**, this becomes the number 7. So, our **tileX variable** now holds a 7!

Doing the same with the **y** position gives us a result of 9. We subtract the `levelOffset` **variable** and this gives us a 3. So `tileY` now stores a 3!

This operation tells us any grid position from any pair of screen co-ordinates. Now let's put those `tileX` and `tileY` **variables** to work.

The next part is a little tricky. First, we create the **variable** which will tell us whether the tile we are checking is solid (**true**) or empty (**false**). The **variable** is called `result` and at first it simply stores **true**.

```
61.     result = true
```

Next, we check if the tile in question is actually in the range of our level array. We must do this because if our character falls down a pit, or walks somewhere on screen where there is no tile data in the level array, we will get an **out of bounds** error. This is because the system is trying to check for a place in the level array which **does not exist**.

To solve this, we use an **if statement** which checks whether the `tileX` or `tileY` **variables** are in the correct range. If they are not, we simply make our `result` **variable** **false** to indicate that we **will not collide** with the tile in question.

```
63.     if tileY < 0 or tileY >= len( level ) or tileX < 0 or tileX >= len( level[0] ) then
64.         result = false
65.     else
66.         if level[tileY][tileX] < 0 then
67.             result = false
68.         endif
69.     endif
70.     return result
```

If it **is** in range, we check **if** that position in the array is solid or empty, then make the `result` **variable** **true** or **false** accordingly.

Using our Collision Function in the Program

Time to put this **function** to good use. We want to apply the gravity effect to our player *only if the tile underneath them is empty*.

Let's write this into our code. We're looking at lines 52 to 57 below:

```
52.     if !collision( playerY + pSize / 2, playerY + pSize + velocity ) then
53.         playerY += velocity
54.     else
55.         playerY = int( ( playerY + velocity + pSize.y ) / tSize ) * tSize - pSize.y
56.         velocity = 0
57.     endif
```

Here, we are using our `collision()` **function** to check if the tile below the player is **not** solid. We use an exclamation mark (!) before the **function** call to check if the result is **not true**.

Let's take a quick look at the arguments for the **function** call. We want to *pass* the player's **x** and **y** positions to the **function**, but we need a couple of other things.

First, we must add $pSize / 2$ to the x position, because we want to check the *middle* of the tile, not the corner. We also add $tSize$ to the y position to check the *bottom* of the tile rather than the top. Check out the graphics below to see what we mean by this.

The image below shows the origin point of the tile in yellow. This point is $(playerX, playerY)$.



If we add $pSize.x / 2$ to the x position, we are describing this point (shown in yellow):



Finally, we add $pSize.y$ to the y position:



That gives us the bottom of the character's feet, but we need to check *where they are going to be*, not where they currently are. For this reason, we *add* the **velocity variable** to the y position. We are checking where the character would be if velocity was added to their position.

Back to the **if statement**:

```
52.     if !collision( playerX + pSize.x / 2, playerY + pSize.y + velocity ) then
53.         playerY += velocity
```

```

54.     else
55.         playerY = int( ( playerY + velocity + pSize.y ) / tSize ) * tSize - pSize.y
56.         velocity = 0
57.     endif

```

So, if the `collision()` function returns a `false`, we know that the tile in question *is empty*, and we can apply velocity to the character's y position.

We use an `else` on line 54 to give an instruction if the tile in question is *not* empty.

On line 55 we set the y position of the player to be exactly where we want it to be and set velocity to 0.

Run the program to see the character fall perfectly on to the platform!

If you've made it this far, you deserve a huge congratulations!

Believe it or not, that's *all* of the collision code complete. This will allow us to create new parts to our level and they will work perfectly.

End Result

Let's double check we're at the same point in the project. Below is a list of exactly how the program should look. If yours is all up to scratch and works without errors then we'll see you in the next tutorial where we'll be making our player move and jump! Exciting!

```

1. background = loadImage( "Kenney/backgrounds", false )
2. tilesheet  = loadImage( "Kenney/superPlatformPack", false )
3. chrSheet   = loadImage( "Kenney/characters", false )
4.
5. playerX = 0
6. playerY = 0
7.
8. gravity = 1
9. velocity = 0
10.
11. screenX = 0
12. screenY = 0
13.
14. tiles = [ 121, 138, 128, 129, 130 ]
15.
16. level = [
17.     [ -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 ],
18.     [ -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 ],
19.     [ -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, 2, 3, 4, -1, -1, -1, -1, -1, -1 ],
20.     [ 1, 1, 1, 1, -1, -1, 1, 1, 1, 1, 1, 1, -1, -1, -1, -1, 1, 1, 1, 1, 1, 1 ],
21.     [ 0, 0, 0, 0, -1, -1, 0, 0, 0, 0, 0, 0, -1, -1, -1, -1, 0, 0, 0, 0, 0, 0 ],
22.     [ 0, 0, 0, 0, -1, -1, 0, 0, 0, 0, 0, 0, -1, -1, -1, -1, 0, 0, 0, 0, 0, 0 ]
23. ]
24.
25. levelHeight = 12
26. levelOffset = levelHeight - len( level )
27. tileSize = 0
28.
29. loop
30.     clear()
31.
32.     screenW = gwidth()
33.     screenH = gheight()
34.     scale = screenH / ( tileSize( tilesheet, 121 ).y * levelHeight )
35.     tSize = scale * tileSize( tilesheet, 121 ).y
36.     pSize = tileSize( chrSheet, 96 ) * scale
37.
38.     drawImage( background, -screenX, -screenY, screenH / imageSize( background ).y )
39.
40.     for row = 0 to len( level ) loop
41.         for col = 0 to len( level[0] ) loop
42.             if level[row][col] >= 0 then
43.                 x = col * tileSize

```

```

44.         y = ( row + levelOffset ) * tileSize
45.         drawSheet( tilesheet, tiles[level[row][col]], x, y, scale )
46.     endif
47.     repeat
48. repeat
49.
50. velocity += gravity
51.
52. if !collision( playerX + pSize.x / 2, playerY + pSize.y + velocity ) then
53.     playerY += velocity
54. else
55.     playerY = int( ( playerY + velocity + pSize.y ) / tileSize ) * tileSize - pSize.y
56.     velocity = 0
57. endif
58.
59. drawSheet( chrSheet, 96, playerX, playerY, scale )
60.
61. update()
62. repeat
63.
64. function collision( x, y )
65.     tileX = int( x / tileSize )
66.     tileY = int( y / tileSize ) - levelOffset
67.
68.     result = true
69.
70.     if tileY < 0 or tileY >= len( level ) or tileX < 0 or tileX >= len( level[0] ) then
71.         result = false
72.     else
73.         if level[tileY][tileX] < 0 then
74.             result = false
75.         endif
76.     endif
77. return result

```

Functions and Keywords used in this tutorial

clear(), drawImage(), drawSheet(), else, endif, for, function, gHeight(), gWidth(), if, int(), len(), loadImage(), loop, repeat, return, tileSize(), then, to, update()

TUTORIALS

Basic Game Tutorial 4: Character Movement

Back again are we? Good to see you!

In this part of the Basic Game Tutorials, we'll be adding perhaps the most fun part of the project. How to move the character and jump around.

Sounds simple, right? Let's see about that shall we...

Actually, this part will be much more simple than the previous one. We have already completed our collision check and we already have gravity and velocity, so moving the character should be a walk in the park!

Before we actually **animate** the character, let's just get them moving around properly.

First, we'll tackle the left and right movement. Jumping will be more fun once we can move around a little!

We'll need two **if statements** to achieve left and right movement, one to check if the left directional button has been pressed, and one for the right directional button.

Before we go ahead and write the **if statements**, it would be very useful to have a **variable** to store the character's movement speed. Let's create that first. This will be a global **variable** at the top of the program. We'll add it at line 7, moving the **gravity** and **velocity variables** down a couple of lines:

```
8. moveSpeed = 5
9.
10. gravity = 1
11. velocity = 0
```

Done!

Using the Directional Buttons to Move the Character

Before we write the **if statements** to move the character, we must call the `controls()` function to access the **Joy-Con** buttons!

At the very start of the main **loop**, add line 34 below. We have included the surrounding lines to make it clear:

```
31. loop
32.     clear()
33.
34.     c = controls(0)
35.
36.     screenW = gwidth()
37.     screenH = gheight()
```


Great! We now have access to all the **Joy-Con** buttons by using our **c variable**.

Now let's use the **moveSpeed variable** to create the movement **if statements**. We'll begin with a simple version, then we'll add some polish.

Add the following lines to your program:

```
63.     if c.right then
64.         playerX += moveSpeed
65.     endif
66.
67.     if c.left then
68.         playerX -= moveSpeed
69.     endif
```

These two very simple **if statements** do almost everything for us. They allow us to press the left and right directional buttons and increase or decrease the player's **x** position.

However, run the program and you might notice that if you walk off an edge then quickly walk backwards, you can lodge yourself firmly inside a solid block! This isn't quite what we want.

We need to use our brilliant **collision() function** in these **if statements** to check if the tile we are about to move into is solid or empty.

Make the changes below to your **if statements**:

```
63.     if c.right and !collide( playerX + tSize / 2 + moveSpeed, playerY + tSize - 1 ) then
64.         playerX += moveSpeed
65.     endif
66.
67.     if c.left and !collide( playerX + tSize / 2 - moveSpeed, playerY + tSize - 1 ) then
68.         playerX -= moveSpeed
69.     endif
```

Just like before, we are using the **collision function** to check if the position we are **about to be in** is solid or empty. This is why we must add or subtract **moveSpeed** depending on which way we are moving.

The Jump

Programming a jump is a key part of creating a platform game. There are multiple ways to achieve a good jump. Since we are already simulating gravity and velocity, our jump code will be very simple indeed and the results look fantastic!

This system of using gravity and velocity will work in any project you might want to use them in.

All we need to do is adjust the velocity **variable**. Let's add a very simple **if statement** to our program:

```
54.     if c.a then
55.         velocity -= 2
56.     endif
```

Run the program and press the A button lightly. Hopefully the character will briefly jump into the air. When A is released, we will be brought back down to the ground by gravity. Awesome!

However... This jump is a little strange. We can hold down the A button to constantly move further into the air, which isn't exactly what we'd like! It's a bit silly really.

What we need is a **timer** which tracks how long we have been in the air, and stops us from holding down the A button to keep flying upwards if the timer reaches a certain point.

First, we'll need to create the **variable** at the start of our program:

```
10. gravity = 1
11. velocity = 0
12.
13. jumpTimer = 0
```

We've shown the **gravity** and **velocity variables** here to make it clear where to put the **jumpTimer variable**.

We begin the timer at 0 and we will count **up** as we are jumping. Let's add something to the jumping **if statement**:

```
55.     if c.a and jumpTimer < 12 then
57.         jumpTimer += 1
58.         velocity -= 2
59.     endif
```

That should do it! Run the program and press the A button to see our newly limited jump.

Oh dear! You might notice that we can only jump once!

This is because when we jump, we are now increasing the **jumpTimer variable** and it must be less than 12 if we want to jump again.

In order to jump again, we must **reset** the **jumpTimer variable** when we reach the ground.

In the **if statement** which checks to see if the character will collide the floor, we must add something after the **else**. Here's how the full **if statement** should look:

```
63.     if !collision( playerX + tSize / 2, playerY + tSize + velocity ) then
64.         playerY += velocity
65.     else
66.         playerY = int( ( playerY + velocity ) / tSize ) * tSize
67.         velocity = 0
68.         jumpTimer = 0
69.     endif
```

Notice the new line on line 68. We reset the **jumpTimer variable** when we know the player has reached the floor.

A Little Extra Polish...

Our jump is pretty much complete. We can only stay in the air for a limited amount of time, we come wonderfully back down to the ground and we collide with the floor. This is really all we need, but it would be quite easy to add a couple of extra things which would really add some polish to our game.

When we jump in real life we have an initial burst of upward velocity, then as we spend more time in the air this decreases until it becomes negative, overcome by the force of gravity. We can make something very similar happen in our jump code with a very simple change:

```
56.     if c.a and jumpTimer < 12 then
57.         jumpTimer += 1
58.         velocity -= 8 / jumpTimer
59.     endif
```

We have changed line 58 so that our velocity is dependent on the `jumpTimer` variable. The longer we have been in the air, the more our velocity is divided by, therefore slowing our jump down as we reach the peak.

To make the jump work nicely we have also increased the amount we modify velocity by to 8. Changing this number will have a big effect on your jump!

You might have noticed that in the middle of your jump, you can press the A button again to pause slightly in the air. This looks a little silly and we can fix it with some very useful code. So let's do it!

We need to keep track of if the A button **has been pressed**. To do this we'll need a **variable**. Add the following global **variable** to the start of the program, just underneath the `jumpTimer` variable:

```
13. jumpTimer = 0
14. oldA = 0
```

This `oldA` variable will only be either `true` or `false`. We will store the old state of the A button in this variable, and check it against the current state of the A button using an **if statement**.

Let's put that idea to use:

```
62.     if oldA and !c.a then
63.         jumpTimer = 12
64.     endif
65.
66.     oldA = c.a
```

This tricky **if statement** resets the jump timer to the maximum value if the A button is pressed again during the jump.

The Program So Far

As always, here is a copy of the entire program so far to make sure you're up to speed. If you're having problems with your code, feel free to start a new project and copy and paste the entire program to be certain:

```
1. background = loadImage( "Kenney/backgrounds", false )
2. tilesheet  = loadImage( "Kenney/superPlatformPack", false )
3. chrSheet   = loadImage( "Kenney/characters", false )
4.
5. playerX = 0
6. playerY = 0
7.
8. moveSpeed = 5
9.
10. gravity = 1
11. velocity = 0
```

```

12.
13. jumpTimer = 0
14. oldA = 0
15.
16. screenX = 0
17. screenY = 0
18.
19. tiles = [ 121, 138, 128, 129, 130 ]
20.
21. level = [
22.   [ -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 ],
23.   [ -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 ],
24.   [ -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, 2, 3, 4, -1, -1, -1, -1, -1, -1 ],
25.   [ 1, 1, 1, 1, -1, 1, 1, 1, 1, 1, 1, 1, -1, -1, -1, 1, 1, 1, 1, 1, 1, 1 ],
26.   [ 0, 0, 0, 0, -1, -1, 0, 0, 0, 0, 0, 0, -1, -1, -1, -1, 0, 0, 0, 0, 0, 0 ],
27.   [ 0, 0, 0, 0, -1, -1, 0, 0, 0, 0, 0, 0, -1, -1, -1, -1, 0, 0, 0, 0, 0, 0 ]
28. ]
29.
30. levelHeight = 12
31. levelOffset = levelHeight - len( level )
32. tSize = 0
33.
34. loop
35.   clear()
36.
37.   c = controls( 0 )
38.
39.   screenW = gwidth()
40.   screenH = gheight()
41.   scale = screenH / ( tileSize( tilesheet, 121 ).y * levelHeight )
42.   tSize = scale * tileSize( tilesheet, 121 ).y
43.   pSize = tileSize( chrSheet, 96 ) * scale
44.
45.   drawImage( background, -screenX, -screenY, screenH / imageSize( background ).y )
46.
47.   for row = 0 to len( level ) loop
48.     for col = 0 to len( level[0] ) loop
49.       if level[row][col] >= 0 then
50.         x = col * tSize
51.         y = ( row + levelOffset ) * tSize
52.         drawSheet( tilesheet, tiles[level[row][col]], x, y, scale )
53.       endif
54.     repeat
55.   repeat
56.
57.   if c.a and jumpTimer < 12 then
58.     jumpTimer += 1
59.     velocity -= 8 / jumpTimer
60.   endif
61.
62.   if oldA and !c.a then
63.     jumpTimer = 12
64.   endif
65.
66.   oldA = c.a
67.
68.   velocity += gravity
69.
70.   if !collision( playerX + pSize.x / 2, playerY + pSize.y + velocity ) then
71.     playerY += velocity
72.   else
73.     playerY = int( ( playerY + velocity + pSize.y ) / tSize ) * tSize - pSize.y
74.     velocity = 0
75.     jumpTimer = 0
76.   endif
77.
78.   if c.right and !collision( playerX + pSize.x / 2 + moveSpeed, playerY + pSize.y - 1 ) then
79.     playerX += moveSpeed
80.   endif
81.
82.   if c.left and !collision( playerX + pSize.x / 2 - moveSpeed, playerY + pSize.y - 1 ) then
83.     playerX -= moveSpeed
84.   endif
85.
86.   drawSheet( chrSheet, 96, playerX, playerY, scale )
87.
88.   update()
89. repeat
90.
91. function collision( x, y )
92.   tileX = int( x / tSize )
93.   tileY = int( y / tSize ) - levelOffset

```

```
94.
95.     result = true
96.
97.     if tileY < 0 or tileY >= len( level ) or tileX < 0 or tileX >= len( level[0] ) then
98.         result = false
99.     else
100.         if level[tileY][tileX] < 0 then
101.             result = false
102.         endif
103.     endif
104. return result
```

Functions and Keywords used in this tutorial

`clear()`, `controls()`, `drawImage()`, `drawSheet()`, `else`, `endif`, `for`, `function`, `gHeight()`, `gWidth()`, `if`, `int()`, `len()`, `loadImage()`, `loop`, `repeat`, `return`, `tileSize()`, `then`, `to`, `update()`

TUTORIALS

Basic Game Tutorial 5: Animation

Hello yet again! Congratulations on your diligent hard work, determination and commitment to improving your skills. You're great.

In this part of the Game tutorial, we'll really be bringing this project to life. It's all well and good having a moving character jumping and walking on platforms, but without animations things look rather bland.

We'll be covering the basics of character animations in this tutorial and as always, the concepts used here can be applied to absolutely any project.

To make the character animations work smoothly and simply, we'll be creating something called a **state machine**.

What is a State Machine?

It's a cool way of saying that we're keeping track of the player's current state. At any time, the player might be idle (standing still), walking, jumping or being hit by an enemy. If we keep track of what state the player is currently in, we can use this information to make animating the player very simple.

First, as always, let's create some variables! We'll need 6 **variables** here. Add the following lines to the top of your program, just beneath the `moveSpeed` **variable**:

```
8. moveSpeed = 5
9.
10. idle = 0
11. walk = 1
12. jump = 2
13. hit = 3
14.
15. state = idle
```

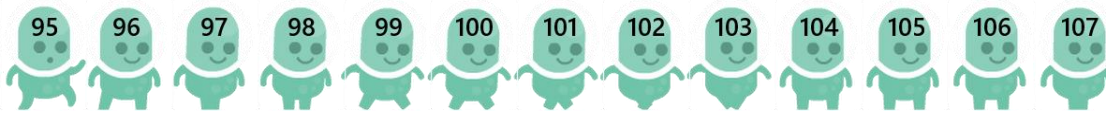
Now we have a variable for each state the player might be in. As you can see, each one holds a different number. We will use this number as an index into an array of animations.

We have also created a **variable** called `state` which we will use to store the player's state.

Now let's create the **array** of animation information, just beneath these **variables**:

```
17. anim = [
18.     [ .start = 96, .length = 1 ],
19.     [ .start = 97, .length = 11 ],
20.     [ .start = 95, .length = 1 ],
21.     [ .start = 94, .length = 1 ]
22. ]
```

There we have it! In this array of structures, we store the start tile of each state's animation. The idle animation for the player is a single frame with a tile number of 96, so the `.start` property is 96, and the `.length` property is 1. Here's an image to detail why we use these numbers in particular:



As you can see, the walk animation begins at tile 96 and lasts for 11 frames. The jump animation begins at tile 95 and lasts for only one frame.

We have created this array in the same order as the state **variables**. Because of this, we can now access any of these state animations with a statement like:

```
print( anim[walk].start )
```

Clever right!

Now let's put these to use. We have one last **variable** to create first, which will store the current animation frame. We'll call this `animationFrame` and it will be defined just after the `anim` **array**:

```
24. animationFrame = 0
```

We're all set!

Time to use these **variables** in the `drawSheet()` **function** used to draw the player. Take a look at the end of the main game **loop**:

```
102. drawSheet( chrSheet, 96, playerX, playerY, scale )
```

Currently we are using a single fixed value for the tile. Rather than the number 96, we need to use a **variable** instead in order to change this during the game. Create a line just above the `drawSheet()` **function** and define the following **variable**:

```
102. animationStart = anim[state].start
103.
104. drawSheet( chrSheet, 96, playerX, playerY, scale )
```

Our **variable** is called `animationStart` and it will store the starting **frame** of animation for our states. This **variable** isn't totally necessary as we can simply use `anim[state].start`, but it makes our code easier to read.

Remember the `animationFrame` **variable** we created earlier? It's time to put this to use:

```
104. drawSheet( chrSheet, animationStart + animationFrame, playerX, playerY, scale )
```

All we have done for this change is swapped out the 96 in our `drawSheet()` **function** for `animationStart + animationFrame`.

This is a very helpful way of changing the tile shown for the player. All we need to do now is **increase** the `animationFrame` **variable** and our tile will animate!

```
104. drawSheet( chrSheet, animationStart + animationFrame, playerX, playerY, scale )
105.
106. animationFrame += 0.2
```

Run the program to see something **very** strange!

Our character animates, but then turns into other characters and completely different tiles until we get an error. Do you understand why this is happening?

We are increasing the tile past the point we want and displaying tiles that aren't in the correct range.

With a couple of **if statements** we can solve this! Create a few lines of space above the `drawSheet()` line, and add the following **if statement**:

```
104.     if animationFrame >= anim[state].length then
105.         animationFrame = 0
109.     endif
107.
108.     drawsheet( chrSheet, animation + animationFrame, playerX, playerY, scale )
```

When we run the program our tile will no longer change. This actually means it's working correctly!

The animation for the idle state is just a single frame. In the animation array, the idle state animation has a `.start` of 96 and a `.length` of 1.

This means the `animationFrame` **variable** never gets above 1, therefore we only see a single frame.

Time to put the state machine to use! Our animation array stores the start and end tiles of each set of animations for each state of the character. All we need to do now is change the player's state!

We must set the state at various points in our program. Remember, we set the state at the start of the program as idle by default. Let's check the first place to change it:

```
73.     if c.a and jumpTimer < 12 then
74.         jumpTimer += 1
75.         velocity -= 8 / jumpTimer
76.         state = jump
77.     endif
```

Right here seems like a good place! When the character jumps into the air, we need the state to change in order to see the jump frame.

Run the program and jump to see if it works!

If it's working properly, our character should change to the jump frame but they will not change back.

To make the character go back the idle frame, we just need to switch the state back to idle **when they land**:

```
87.     if !collision( playerX + tSize / 2, playerY + tSize + velocity ) then
88.         playerY += velocity
89.     else
90.         playerY = int( ( playerY + velocity ) / tSize ) * tSize
91.         velocity = 0
```



```

92.     jumpTimer = 0
93.     state = idle
94.     endif

```

Above is the **if statement** which causes the character to fall through empty space and land on platforms. Below the **else** is what will happen when our character lands on a platform tile. Here we just need to add `state = idle` and we're done!

Run the program and see our character's glorious jump! Truly a more majestic jump has never been seen.

All that's left is to make our character walk. We already have all the data we need - we just need to change the **state variable** to `walk`. We need to add something to our **left and right movement if statements**.

```

96.     if c.right and !collision( playerX + tSize / 2 + moveSpeed, playerY + tSize - 1 ) then
97.         playerX += moveSpeed
98.         if state != jump then
99.             state = walk
100.        endif
100.    endif

```

We have added lines 98 to 100 above in the first of the movement **if statements**. We want to set the state to `walk` **when** we press the right or left directional buttons, but **only if** we are not **already jumping**. Therefore we must write:

```

if state != jump then
    state = walk
endif

```

Now let's add the exact same thing to left movement **if statement**:

```

103.    if c.left and !collision( playerX + tSize / 2 - moveSpeed, playerY + tSize - 1 ) then
104.        playerX -= moveSpeed
105.        if state != jump then
106.            state = walk
107.        endif
108.    endif

```

That's it! Our player animation is complete! Run the program and move the character around to see the results.

There is one last little task we must accomplish before moving to the next stage however. At the minute, we can move the character, jump and land on platforms, but when we travel to the right side of the screen, the camera doesn't move to reveal the rest of the level! This just won't do.

We already have the **variables** we'll need, we just need to put them to use. Add the lines below to your program:

```

61.     if playerX - screenX < screenWidth * 0.4 then
62.         screenX -= moveSpeed
63.     endif
64.     if playerX - screenX > screenWidth * 0.6 then
65.         screenX += moveSpeed
66.     endif
67.     if screenX < 0 then

```

```
68.     screenX = 0
69.     endif
```

Run the program once you've added the lines above and travel to the right side of the screen. We should see the background move, but not the level just yet!

This is because we need to modify our level drawing position to be relative to the `screenX` variable.

Go to the **for loop** which draws the level and add the change below:

```
73.     for row = 0 to len( level ) loop
74.         for col = 0 to len( level[0] ) loop
75.             if level[row][col] >= 0 then
76.                 x = col * tileSize
77.                 y = ( row + levelOffset ) * tileSize
78.                 drawSheet( tilesheet, tiles[level[row][col]], x - screenX, y, scale )
79.             endif
80.         repeat
81.     repeat
```

Can you spot the change? It's not very obvious. On line 78, we must add a `- screenX` to the `x` position argument of the `drawSheet()` function. This will cause the level to be drawn relative to the movement of our screen.

We must also do this same thing for the player or we'll encounter some strange problems. Find the `drawSheet()` line for the player and add the same change:

```
126.     drawSheet( chrSheet, animationStart + animationFrame, playerX - screenX, playerY, scale )
```

Run the program and travel to the right to see the level and background and level move with the player to reveal the rest of the level.

Wouldn't it be nice if the background image moved at a different speed than the level? This way, it would really look as though the background was in the distance! We can do this very easily. Find the `drawImage()` line which draws the background:

```
71.     drawImage( background, -screenX / 2, -screenY, screenHeight / imageSize( background ).y )
```

We simply add a `/ 2` to the `x` position! Now our background will move at **half the speed** of the foreground.

Run the program to see a wonderful moving level. All done!

The Program So Far

As always, we have a complete and up-to-date version of the whole program so far just below. If your program is not working and you cannot figure it out, feel free to copy and paste this code into a new project file:

```
1. background = loadImage( "Kenney/backgrounds", false )
2. tilesheet  = loadImage( "Kenney/superPlatformPack", false )
3. chrSheet   = loadImage( "Kenney/characters", false )
4.
5. playerX = 0
6. playerY = 0
7.
8. moveSpeed = 5
```

```

9.
10. idle = 0
11. walk = 1
12. jump = 2
13. hit = 3
14.
15. state = idle
16.
17. anim = [
18.   [ .start = 96, .length = 1 ],
19.   [ .start = 97, .length = 11 ],
20.   [ .start = 95, .length = 1 ],
21.   [ .start = 94, .length = 1 ]
22. ]
23.
24. animationFrame = 0
25.
26. gravity = 1
27. velocity = 0
28.
29. jumpTimer = 0
30. oldA = 0
31.
32. screenX = 0
33. screenY = 0
34.
35. tiles = [ 121, 138, 128, 129, 130 ]
36.
37. level = [
38.   [ -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 ],
39.   [ -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 ],
40.   [ -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, 2, 3, 4, -1, -1, -1, -1, -1, -1, -1, -1 ],
41.   [ 1, 1, 1, 1, -1, -1, 1, 1, 1, 1, 1, 1, -1, -1, -1, -1, 1, 1, 1, 1, 1, 1, 1 ],
42.   [ 0, 0, 0, 0, -1, -1, 0, 0, 0, 0, 0, 0, -1, -1, -1, -1, 0, 0, 0, 0, 0, 0, 0 ],
43.   [ 0, 0, 0, 0, -1, -1, 0, 0, 0, 0, 0, 0, -1, -1, -1, -1, 0, 0, 0, 0, 0, 0, 0 ]
44. ]
45.
46. levelHeight = 12
47. levelOffset = levelHeight - len( level )
48. tSize = 0
49.
50. loop
51.   clear()
52.
53.   c = controls( 0 )
54.
55.   screenW = gwidth()
56.   screenH = gheight()
57.   scale = screenH / ( tileSize( tilesheet, 121 ).y * levelHeight )
58.   tSize = scale * tileSize( tilesheet, 121 ).y
59.   pSize = tileSize( chrSheet, 96 ) * scale
60.
61.   if playerX - screenX < screenW * 0.4 then
62.     screenX -= moveSpeed
63.   endif
64.   if playerX - screenX > screenW * 0.6 then
65.     screenX += moveSpeed
66.   endif
67.   if screenX < 0 then
68.     screenX = 0
69.   endif
70.
71.   drawImage( background, -screenX / 2, -screenY, screenH / imageSize( background ).y )
72.
73.   for row = 0 to len( level ) loop
74.     for col = 0 to len( level[0] ) loop
75.       if level[row][col] >= 0 then
76.         x = col * tSize
77.         y = ( row + levelOffset ) * tSize
78.         drawSheet( tileSheet, tiles[level[row][col]], x - screenX, y, scale )
79.       endif
80.     repeat
81.   repeat
82.
83.   if c.a and jumpTimer < 12 then
84.     jumpTimer += 1
85.     velocity -= 8 / jumpTimer
86.     state = jump
87.   endif
88.
89.   if oldA and !c.a then
90.     jumpTimer = 12

```

```

91.     endif
92.
93.     oldA = c.a
94.
95.     velocity += gravity
96.
97.     if !collision( playerX + pSize.x / 2, playerY + pSize.y + velocity ) then
98.         playerY += velocity
99.     else
100.         playerY = int( ( playerY + velocity + pSize.y ) / tSize ) * tSize - pSize.y
101.         velocity = 0
102.         jumpTimer = 0
103.         state = idle
104.     endif
105.
106.     if c.right and !collision( playerX + pSize.x / 2 + moveSpeed, playerY + pSize.y - 1 ) then
107.         playerX += moveSpeed
108.         if state != jump then
109.             state = walk
110.         endif
111.     endif
112.
113.     if c.left and !collision( playerX + pSize.x / 2 - moveSpeed, playerY + pSize.y - 1 ) then
114.         playerX -= moveSpeed
115.         if state != jump then
116.             state = walk
117.         endif
118.     endif
119.
120.     animationStart = anim[state].start
121.
122.     if animationFrame >= anim[state].length then
123.         animationFrame = 0
124.     endif
125.
126.     drawSheet( chrSheet, animationStart + animationFrame, playerX - screenX, playerY, scale )
127.
128.     animationFrame += 0.2
129.
130.     update()
131. repeat
132.
133. function collision( x, y )
134.     tileX = int( x / tSize )
135.     tileY = int( y / tSize ) - levelOffset
136.
137.     result = true
138.
139.     if tileY < 0 or tileY >= len( level ) or tileX < 0 or tileX >= len( level[0] ) then
140.         result = false
141.     else
142.         if level[tileY][tileX] < 0 then
143.             result = false
144.         endif
145.     endif
146. return result

```

Functions and Keywords used in this tutorial

clear(), controls(), drawImage(), drawSheet(), else, endIf, for, function, gHeight(), gWidth(), if, int(), len(), loadImage(), loop, repeat, return, tileSize(), then, to, update()

TUTORIALS

Basic Game Tutorial 6: Items

Congratulations on making it to the final part of this project! Hopefully you've learned a lot throughout these tutorials and feel better about going about creating your own game.

In this part of the tutorial we'll be adding items to collect. As with the previous parts, the way we will achieve this in our program will allow you to add more items very easily.

Let's keep it classic with the good old coin. Our project will begin with just coins for our items, but adding different types will be very simple.

We'll need to begin with a few **variables** as usual. Just like with the state machine in the previous project, the items need a type and a state. Add the following lines to your program:

```
50. coin = 0
51.
52. active = 0
53. collect = 1
54. inactive = 2
```

There we go! We've got a **variable** called `coin` which stores a **0**. This will be used as an index into an array of tiles.

Similarly, we have a number of state **variables** below this which we will use to determine what happens to the coin during the game.

Now we need to create the **array** of items. Each item needs its own **structure** with a number of properties:

```
56. items = [
57.   [ .type = coin, .x = 7, .y = 1, .state = active ],
58.   [ .type = coin, .x = 8, .y = 0, .state = active ],
59.   [ .type = coin, .x = 9, .y = 0, .state = active ],
60.   [ .type = coin, .x = 10, .y = 1, .state = active ]
61. ]
```

As you can see, each item has 4 properties. We have a `.type` which stores the type of the item. We have a `.x` and `.y` which are the **level coordinates** of the item (different than the screen coordinates, these level coordinates tell us which **row** and **column** of the **level array** the item will appear in) and finally a `.state` property to store the state.

Next up we'll need the tilesheet information to animate the items, just like we needed for the player:

```
63. itemAnim = [
64.   [ .start = 154, .length = 1 ]
65. ]
```

Since we are only using coins at the moment, we don't need any more information in this array. If we were to add another **type** of item, we would need more information.

This information can be accessed with `itemAnim[0].start`, or, since we have a **coin variable** which stores a 0, we can say `itemAnim[items[0].type]`. This sort of **array** indexing, despite looking quite complex, is very useful and worth getting your head around!

We are using the `item.type` property as an index into the `itemAnim` **array**.

Lastly, we should create a **variable** to keep track of the number of coins the player has collected:

```
67. playerCoins = 0
```

Excellent. Now we have everything we need to put the items on screen. Head into the main **loop** for this next part, just after the **for loop** which draws the level.

We'll be using a **for loop** to loop over the array of items and draw each one. This **for loop** will end up being rather long and complex looking, so let's build it step by step. First we just want to actually draw the coins on screen:

```
102.     for i = 0 to len( items ) loop
103.         x = items[i].x * tSize
104.         y = ( items[i].y + levelOffset ) * tSize
105.         drawSheet( tilesheet, itemAnim[items[i].type].start, x - screenX, y, scale )
106.     repeat
```

Run the program and we should see the coins on screen. Of course, without the code to make it happen, we cannot pick the coins up yet.

Before we do that, let's make sure we understand what's happening. Our **for loop** counts using an **i variable** from 0 to the length of our items array. Our items array has 4 elements, so **i** will count from 0 to 3.

We create some local **x** and **y variables** to store the position of the item. This is just to make our code easier to read.

We take the `.x` and `.y` properties of the current item in question and multiply them by the `tSize` **variable** to give us the **screen coordinates** for the item. With the **y** position, we must add the `levelOffset` in order to put them on the correct row.

Then, on line 105, we use the `drawSheet()` **function** to draw the item. The tricky part here is the tile index:

```
itemAnim[items[i].type].start
```

This is actually one property of a structure within an array of structures indexing another array of structures to give us the correct property with which to index into a tilesheet. Try saying that three times quickly.

Since the only item type we are using is a **coin**, `items[i].type` is always a 0. If we use a 0 as an index into the `itemAnim` **array**, we get the animation tile for the coin.

As mentioned before, this might seem a little pointless since we could simply use the tile number for the coin in the tilesheet, but then when it comes to adding items we'll have a very difficult time indeed.

Collecting the Coins

If we want to be able to collect the coins, we'll have to make this **for loop** of ours a little more complicated.

First we'll wrap the calculations and the `drawSheet()` line in an **if statement**. We only want to do these things **if** the item is **not** inactive.

```
102.     for i = 0 to len( items ) loop
103.         if items[i].state != inactive then
104.             x = items[i].x * tileSize
105.             y = ( items[i].y + levelOffset ) * tileSize
106.             drawSheet( tilesheet, itemAnim[items[i].type].start, x - screenX, y, scale )
107.         endif
108.     repeat
```

Great! Now we need to add an **if statement** to check if the player has moved into the range of an item.

Collision If Statement

This **if statement** is going to be quite long indeed. When writing game code there really is no avoiding this sometimes. Ready?

```
102.     for i = 0 to len( items ) loop
103.         if items[i].state != inactive then
104.             x = items[i].x * tileSize
105.             y = ( items[i].y + levelOffset ) * tileSize
106.             if playerX + pSize.x > x and playerX < x + tileSize and
107.                playerY + pSize.y > y and playerY < y + tileSize and
108.                items[i].state == active
109.             then
110.                 drawSheet( tilesheet, itemAnim[items[i].type].start, x - screenX, y, scale )
111.             endif
112.         repeat
```

Phew, check that out for an **if statement!** It's not even finished yet, this is just the condition! It's so large that we've split it up across multiple lines to make it easier to read. Remember, you can format your code however you like! There's nothing stopping you from breaking up long lines into multiple to make things clearer.

We are checking if the right hand side of the player (`playerX + pSize.x`) is greater than the left side of the coin (`> x`), **and** the the left side of the player `playerX` is less than the right hand edge of the coin `< x + tileSize`.

We are also checking if the player's feet `playerY + pSize.y` is greater than the top of the coin `> y`, **and** that the top of the player's head (`playerY`) is less than the bottom of the coin tile (`{< y + tileSize}`).

We are **also** checking that the coin itself has to be active.

These 5 conditions must **all be true** for this **if statement** to begin. Now let's actually make something happen in it!

```
102.     for i = 0 to len( items ) loop
103.         if items[i].state != inactive then
104.             x = items[i].x * tSize
105.             y = ( items[i].y + levelOffset ) * tSize
106.             if playerX + pSize.x > x and playerX < x + tSize and
107.                 playerY + pSize.y > y and playerY < y + tSize and
108.                 items[i].state == active
109.                 then
110.                     playNote( 0, 3, 1046.50, 1, 20, 0.5 )
111.                     playNote( 1, 3, 1396.71, 1, 10, 0.5 )
112.                     playerCoins += 1
113.                     items[i].state = collect
114.                 endif
115.                 drawSheet( tilesheet, itemAnim[items[i].type].start, x - screenWidth, y, scale )
116.             endif
117.         repeat
```

There we have it. As you can see, the new lines are from 109 to 114. After the **then**, we first use two **playNote()** **functions** to play a nice coin collection sound.

We also increase the **playerCoins variable** by 1 and change the **.state** property of the item to **collect**.

By having a state other than **active** and **inactive**, we can now apply some cool things to happen before the coin vanishes.

When we pick up an item in a game we sometimes see that item shoot into the air a little before vanishing. Let's make this happen by using the **.collect** state:

```
102.     for i = 0 to len( items ) loop
103.         if items[i].state != inactive then
104.             x = items[i].x * tSize
105.             y = ( items[i].y + levelOffset ) * tSize
106.             if playerX + pSize.x > x and playerX < x + tSize and
107.                 playerY + pSize.y > y and playerY < y + tSize and
108.                 items[i].state == active
109.                 then
110.                     playNote( 0, 3, 1046.50, 1, 20, 0.5 )
111.                     playNote( 1, 3, 1396.71, 1, 10, 0.5 )
112.                     playerCoins += 1
113.                     items[i].state = collect
114.                 endif
115.                 if items[i].state == collect then
116.                     items[i].y -= 0.15
117.                     if items[i].y < -1 then
118.                         items[i].state = inactive
119.                     endif
120.                 endif
121.                 drawSheet( tilesheet, itemAnim[items[i].type].start, x - screenWidth, y, scale )
122.             endif
123.         repeat
```

There we go. That's our **for loop** all done!

Because of the **collect** state, we can make something happen to the item **before** it vanishes. On line 115 we check if the state of an item is **collect**. If it is, we reduce the **y** position of the item by a small amount. We then have another **if statement** within this to check if the **y** position has gone past a

certain number. If it has, we change the state to **inactive!** Once the state is inactive, the item is no longer drawn due to the **if statement** on line 103.

Told you it would be a rather large **for loop!**

We're still missing something... We currently have no way of telling how many coins we have! We need a couple of draw commands to display the number of coins. Let's put these just after our items **for loop**:

```
125.     drawSheet( tilesheet, 154, 10, 10, scale )
126.     drawText( 10 + tSize * 0.75 + 10, 10, tSize * 0.75, grey, playerCoins )
```

The `drawSheet()` line just above puts an image of the coin in the top left corner of our screen. The `drawText()` line simply displays the `playerCoins` **variable** next to it!

Run the program and pick up a coin! We should hear a little sound, see the coin pop into the air and our coin counter in the top left should increase. If that's all happening, excellent!

The Program So Far

Alright that's all for now. As usual, below is a copy of the whole program. Make sure we're matching and your program works as intended before moving on to the next tutorial, in which we'll be adding an enemy to the game!

```
1. background = loadImage( "Kenney/backgrounds", false )
2. tilesheet = loadImage( "Kenney/superPlatformPack", false )
3. chrSheet = loadImage( "Kenney/characters", false )
4.
5. playerX = 0
6. playerY = 0
7.
8. moveSpeed = 5
9.
10. idle = 0
11. walk = 1
12. jump = 2
13. hit = 3
14.
15. state = idle
16.
17. anim = [
18.   [ .start = 96, .length = 1 ],
19.   [ .start = 97, .length = 11 ],
20.   [ .start = 95, .length = 1 ],
21.   [ .start = 94, .length = 1 ]
22. ]
23.
24. animationFrame = 0
25.
26. gravity = 1
27. velocity = 0
28.
29. jumpTimer = 0
30. oldA = 0
31.
32. screenX = 0
33. screenY = 0
34.
35. tiles = [ 121, 138, 128, 129, 130 ]
36.
37. level = [
38.   [ -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 ],
39.   [ -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 ],
40.   [ -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, 2, 3, 4, -1, -1, -1, -1, -1, -1, -1, -1 ],
41.   [ 1, 1, 1, 1, -1, -1, 1, 1, 1, 1, 1, 1, -1, -1, -1, 1, 1, 1, 1, 1, 1, 1 ],
42.   [ 0, 0, 0, 0, -1, -1, 0, 0, 0, 0, 0, 0, -1, -1, -1, 0, 0, 0, 0, 0, 0, 0 ],
43.   [ 0, 0, 0, 0, -1, -1, 0, 0, 0, 0, 0, 0, 1, -1, -1, 0, 0, 0, 0, 0, 0, 0 ]
44. ]
45.
```

```

46. levelHeight = 12
47. levelOffset = levelHeight - len( level )
48. tSize = 0
49.
50. coin = 0
51.
52. active = 0
53. collect = 1
54. inactive = 2
55.
56. items = [
57.   [ .type = coin, .x = 7, .y = 1, .state = active ],
58.   [ .type = coin, .x = 8, .y = 0, .state = active ],
59.   [ .type = coin, .x = 9, .y = 0, .state = active ],
60.   [ .type = coin, .x = 10, .y = 1, .state = active ]
61. ]
62.
63. itemAnim = [
64.   [ .start = 154, .length = 1 ]
65. ]
66.
67. playerCoins = 0
68.
69. loop
70.   clear()
71.
72.   c = controls( 0 )
73.
74.   screenW = gwidth()
75.   screenH = gheight()
76.   scale = screenH / ( tileSize( tilesheet, 121 ).y * levelHeight )
77.   tSize = scale * tileSize( tilesheet, 121 ).y
78.   pSize = tileSize( chrSheet, 96 ) * scale
79.
80.   if playerX - screenX < screenW * 0.4 then
81.     screenX -= moveSpeed
82.   endif
83.   if playerX - screenX > screenW * 0.6 then
84.     screenX += moveSpeed
85.   endif
86.   if screenX < 0 then
87.     screenX = 0
88.   endif
89.
90.   drawImage( background, -screenX / 2, -screenY, screenH / imageSize( background ).y )
91.
92.   for row = 0 to len( level ) loop
93.     for col = 0 to len( level[0] ) loop
94.       if level[row][col] >= 0 then
95.         x = col * tSize
96.         y = ( row + levelOffset ) * tSize
97.         drawSheet( tilesheet, tiles[level[row][col]], x - screenX, y, scale )
98.       endif
99.     repeat
100.   repeat
101.
102.   for i = 0 to len( items ) loop
103.     if items[i].state != inactive then
104.       x = items[i].x * tSize
105.       y = ( items[i].y + levelOffset ) * tSize
106.       if playerX + pSize.x > x and playerX < x + tSize and
107.         playerY + pSize.y > y and playerY < y + tSize and
108.         items[i].state == active
109.       then
110.         playNote( 0, 3, 1046.50, 1, 20, 0.5 )
111.         playNote( 1, 3, 1396.71, 1, 10, 0.5 )
112.         playerCoins += 1
113.         items[i].state = collect
114.       endif
115.       if items[i].state == collect then
116.         items[i].y -= 0.15
117.         if items[i].y < -1 then
118.           items[i].state = inactive
119.         endif
120.       endif
121.       drawSheet( tilesheet, itemAnim[items[i].type].start, x - screenX, y, scale )
122.     endif
123.   repeat
124.
125.   if c.a and jumpTimer < 12 then
126.     jumpTimer += 1
127.     velocity -= 8 / jumpTimer

```

```

128.     state = jump
129. endif
130.
131. if oldA and !c.a then
132.     jumpTimer = 12
133. endif
134.
135. oldA = c.a
136.
137. velocity += gravity
138.
139. if !collision( playerX + pSize.x / 2, playerY + pSize.y + velocity ) then
140.     playerY += velocity
141. else
142.     playerY = int( ( playerY + velocity + pSize.y ) / tSize ) * tSize - pSize.y
143.     velocity = 0
144.     jumpTimer = 0
145.     state = idle
146. endif
147.
148. if c.right and !collision( playerX + pSize.x / 2 + moveSpeed, playerY + pSize.y - 1 ) then
149.     playerX += moveSpeed
150.     if state != jump then
151.         state = walk
152.     endif
153. endif
154.
155. if c.left and !collision( playerX + pSize.x / 2 - moveSpeed, playerY + pSize.y - 1 ) then
156.     playerX -= moveSpeed
157.     if state != jump then
158.         state = walk
159.     endif
160. endif
161.
162. animationStart = anim[state].start
163.
164. if animationFrame >= anim[state].length then
165.     animationFrame = 0
166. endif
167.
168. drawSheet( chrSheet, animationStart + animationFrame, playerX - screenX, playerY, scale )
169.
170. animationFrame += 0.2
171.
172. update()
173. repeat
174.
175. function collision( x, y )
176.     tileX = int( x / tSize )
177.     tileY = int( y / tSize ) - levelOffset
178.
179.     result = true
180.
181.     if tileY < 0 or tileY >= len( level ) or tileX < 0 or tileX >= len( level[0] ) then
182.         result = false
183.     else
184.         if level[tileY][tileX] < 0 then
185.             result = false
186.         endif
187.     endif
188. return result

```

Functions and Keywords used in this tutorial

clear(), controls(), drawImage(), drawSheet(), drawText(), else, endif, for, function, gHeight(), gWidth(), if, int(), len(), loadImage(), loop, playNote(), repeat, return, tileSize(), then, to, update()

TUTORIALS

Basic Game Tutorial 7: Enemies

Welcome back! We're close to completion with this last part of the tutorial!

Creating enemies is going to be very similar indeed to the way we have programmed the items. We'll be using the exact same techniques here, even re-using the state **variables** we created in the last part.

However, we will need an extra state for the enemies. This state will be `death` and we will use it when we jump on an enemy. Add the following line to the item state **variables**:

```
50. coin = 0
51.
52. active = 0
53. collect = 1
54. inactive = 2
55. death = 3
```

All done. Now we need to create a type for the enemy along with a couple of arrays just like before. This next bit of code should go just before the main **loop**:

```
70. slime = 0
```

Just like with `coin`, we create a **variable** to store an index into an array. Our enemy is going to be a slime, so that's what we've called the **variable**!

Next let's create the enemy information **array**:

```
72. enemies = [
73.   [ .type = slime, .x = 20, .y = 1, .state = active, .frame = 0, .velocity = 0, .dir = 0.05 ]
74. ]
```

You might notice that we have more properties for the enemies than the items. We want our enemy to move around and to be affected by gravity just like the player. Because of this, each enemy needs a `.velocity` and a `.dir` to store its movement speed and direction. Since the enemies have multiple frames of animation and each enemy might be animated at different times, each enemy needs its own `.frame` property too.

Now we need an **array** to store the animation details for the enemies:

```
76. enemyAnim = [
77.   [ .start = 165, .length = 2 ]
78. ]
```

Given that we only have one type of enemy (a slime), we only need one element in this **array** for now. If we added more enemy types, we would need more elements in this array with the tile information for each one.

For now we have everything we need to put a slime on screen and make it move.

Just as with the items, this will be a **for loop** which ends up being quite large. Ready? Of course you are.

This **for loop** will go after the draw commands for the items. Just like before, we'll start simple and add features as we go:

```
139.     for i = 0 to len( enemies ) loop
140.         if enemies[i].state != inactive then
141.             x = enemies[i].x * tileSize
142.             y = ( enemies[i].y + levelOffset ) * tileSize
143.             eAnimStart = enemyAnim[enemies[i].type].start
144.             eSize = tileSize( chrSheet, eAnimStart + enemies[i].frame ) * scale )
145.             drawSheet( chrSheet, eAnimStart + enemies[i].frame, x - screenX, y, scale )
146.         endif
147.     repeat
```

This **for loop** counts over the **enemies array**. First we check to see if the **.state** property is **not inactive**. If it is not, we calculate **x** and **y** positions just like with items.

On line 143 we create a **variable** to make our code easier to read. **eAnimStart** stores the starting tile of the enemy animation from the tilesheet.

We also create a **variable** to store the size of the scaled up enemy tile on line 144. Since each frame of the enemy animation might be a different size, we use **eAnimStart + enemies[i].frame** in the **tileSize()** **function** to give us the correct size for each frame.

Finally, on line 145 we use the **drawSheet()** **function** to draw the enemy at its calculated position.

Run the program to see our slimy friend sitting comfortably in mid air above the third platform of the level.

We still have some ways to go!

First, let's get the slime to animate. We'll need to adjust the **enemies[i].frame** property to achieve this:

```
139.     for i = 0 to len( enemies ) loop
140.         if enemies[i].state != inactive then
141.             x = enemies[i].x * tileSize
142.             y = ( enemies[i].y + levelOffset ) * tileSize
143.             eAnimStart = enemyAnim[enemies[i].type].start
144.             eSize = tileSize( chrSheet, eAnimStart + enemies[i].frame ) * scale )
145.             drawSheet( chrSheet, eAnimStart + enemies[i].frame, x - screenX, y, scale )
146.             enemies[i].frame += 0.1
147.             if enemies[i].frame >= enemyAnim[enemies[i].type].length then
148.                 enemies[i].frame = 0
149.             endif
150.         endif
151.     repeat
```

We have added lines 146 to 149 above. Just as we did with the player animation, we use an increasing animation frame **variable** to animate the enemy, then an **if statement** checks to see if the animation frame is greater than the length of the animation stored in the **enemyAnim array**. If it is, we reset it to 0.

Run the program to see our slime sliming about from one frame to the next. Let's get them out of the air and apply some gravity. First, we'll need an **if statement**:

```

139.     for i = 0 to len( enemies ) loop
140.         if enemies[i].state != inactive then
141.             x = enemies[i].x * tileSize
142.             y = ( enemies[i].y + levelOffset ) * tileSize
143.             eAnimStart = enemyAnim[enemies[i].type].start
144.             eSize = tileSize( chrSheet, eAnimStart + enemies[i].frame ) * scale )
145.
146.             if enemies[i].state != death then
147.                 endif
148.
149.                 drawSheet( chrSheet, eAnimStart + enemies[i].frame, x - screenX, y, scale )
150.                 enemies[i].frame += 0.05
151.                 if enemies[i].frame >= enemyAnim[enemies[i].type].length then
152.                     enemies[i].frame = 0
153.                 endif
154.             endif
155.         repeat

```

We've created some lines of space around our new **if statement** on line 146 to make things clearer. Everything we add to this from here onward will be inside this **if statement**, since we only want the enemy to move or be jumped on if they are not already in the `death` state.

The `death` state for the enemies will be very similar to the `collect` state for items, since it will be used to make something specific happen before the enemy becomes `inactive`.

Let's get gravity and velocity working first of all:

```

146.         if enemies[i].state != death then
147.             enemies[i].velocity += gravity
148.             if !collision( x + eSize.x / 2, y + eSize.y + enemies[i].velocity / eSize.y ) then
149.                 enemies[i].y += enemies[i].velocity / tileSize
150.             else
151.                 enemies[i].y = int( ( enemies[i].y + enemies[i].velocity / tileSize + eSize.y / tileSize ) - eSize.y / tileSize )
152.                 enemies[i].velocity = 0
153.             endif
154.         endif

```

There we have it! Run the program and our slimy friend should fall down to the ground and land safely.

To achieve this we are using almost exactly the same section of code as we did to make the player land safely on a platform. We use the custom `collision()` **function** again to check if the tile beneath the enemy is one to collide with. If it is not (`!`), we apply the enemy's `.velocity` to the `y` position.

Colliding with Enemies

In order to interact with the enemies, we'll need a gigantic **if statement** just like we did for the items. Again, we'll split this up across a few lines for clarity:

```

146.         if enemies[i].state != death then
147.             enemies[i].velocity += gravity
148.             if !collision( x + eSize.x / 2, y + eSize.y + enemies[i].velocity / eSize.y ) then
149.                 enemies[i].y += enemies[i].velocity / tileSize
150.             else
151.                 enemies[i].y = int( ( enemies[i].y + enemies[i].velocity / tileSize + eSize.y / tileSize ) - eSize.y / tileSize )
152.                 enemies[i].velocity = 0
153.             endif
154.             if playerX + pSize.x > x and playerX < x + eSize.x and
155.                 playerY + pSize.y > y and playerY < y + eSize.y and
156.                 enemies[i].state == active and velocity > 0
157.             then
158.                 enemies[i].state = death
159.             endif
160.         endif

```

Our monster of an **if statement** begins at line 154. Similar to the items we are checking if the **x** and **y** positions of the player are in range of the **x** and **y** positions of the enemy. We also check if the enemy is in the **active** state, since the enemy must be alive and well in order for us to collide with them.

One extra condition in the **if statement** is that we must have a velocity greater than 0 (**and velocity > 0**). This means we cannot hurt the enemy unless we are jumping, since jumping is the only way to increase our velocity!

If all of these 6 conditions are **true**, the whole **if statement** is **true** and we set the state of the enemy to **death**. Similar to the **collide function**, we will use this as a way of applying specific effects to the enemy before it becomes **inactive**.

Making the Enemy Move

An enemy which just stands still isn't very exciting! We need to make our slime move around the platform and turn around if they're about to fall off the edge.

We'll be using the `enemies[i].dir` property for this.

```
146.         if enemies[i].state != death then
147.             enemies[i].velocity += gravity
148.             if !collision( x + eSize.x / 2, y + eSize.y + enemies[i].velocity / eSize.y ) then
149.                 enemies[i].y += enemies[i].velocity / tSize
150.             else
151.                 enemies[i].y = int( ( enemies[i].y + enemies[i].velocity / tSize + eSize.y / tSize ) ) - eSize.y / tSize
152.                 enemies[i].velocity = 0
153.             endif
154.             if playerX + pSize.x > x and playerX < x + eSize.x and
155.                playerY + pSize.y > y and playerY < y + eSize.y and
156.                enemies[i].state == active and velocity > 0
157.             then
158.                 enemies[i].state = death
159.             endif
160.             if !collision( x + eSize.x / 2 + enemies[i].dir * tSize, y + eSize.y ) then
161.                 enemies[i].dir = -enemies[i].dir
162.             else
163.                 enemies[i].x += enemies[i].dir
164.             endif
165.         endif
```

Our new **if statement** begins at line 160 and ends at 164. We use the custom `collision()` **function** again to check if the tile underneath the tile the enemy is **about to walk into** is empty. If it is, we use `enemies[i].dir = -enemies[i].dir` to change the direction the enemy travels in. If the tile in question is **not** empty, we simply keep moving!

We're almost done!

We just need something to happen when the enemy enters the **death** state. Since this entire section is wrapped in an **if enemies[i].state != death**, we can simply put an **else** before the **endif** to make something happen when `enemies[i].state == death`:

```
165.         else
166.             enemies[i].y += 8 / tSize
167.             y += 8
168.             if y > screen_h then
169.                 enemies[i].state = inactive
170.             endif
171.         endif
```

This last section of the enemy code tells the enemy what to do when it enters the `death` state. We increase the `y` position of the enemy (moving it down the screen) and a simple **if statement** checks to see if the `y` position has become greater than the screen height. If it is, we set the enemy's state to `inactive`, preventing it from being drawn!

The Whole For Loop

WOW! That was a lot of code. Let's take a look at the whole enemies **for loop** to make sure we've got this right:

```
139.   for i = 0 to len( enemies ) loop
140.     if enemies[i].state != inactive then
141.       x = enemies[i].x * tileSize
142.       y = ( enemies[i].y + levelOffset ) * tileSize
143.       eAnimStart = enemyAnim[enemies[i].type].start
144.       eSize = tileSize( chrSheet, eAnimStart + enemies[i].frame ) * scale )
145.
146.     if enemies[i].state != death then
147.       enemies[i].velocity += gravity
148.       if !collision( x + eSize.x / 2, y + eSize.y + enemies[i].velocity / eSize.y ) then
149.         enemies[i].y += enemies[i].velocity / tileSize
150.       else
151.         enemies[i].y = int( ( enemies[i].y + enemies[i].velocity / tileSize + eSize.y / tileSize ) ) - eSize.y / tileSize
152.         enemies[i].velocity = 0
153.       endif
154.       if playerX + pSize.x > x and playerX < x + eSize.x and
155.         playerY + pSize.y > y and playerY < y + eSize.y and
156.         enemies[i].state == active and velocity > 0
157.       then
158.         enemies[i].state = death
159.       endif
160.       if !collision( x + eSize.x / 2 + enemies[i].dir * tileSize, y + eSize.y ) then
161.         enemies[i].dir = -enemies[i].dir
162.       else
163.         enemies[i].x += enemies[i].dir
164.       endif
165.     else
166.       enemies[i].y += 8 / tileSize
167.       y += 8
168.       if y > screen_h then
169.         enemies[i].state = inactive
170.       endif
171.     endif
172.
173.     drawSheet( chrSheet, eAnimStart + enemies[i].frame, x - screenX, y, scale )
174.     enemies[i].frame += 0.05
175.     if enemies[i].frame >= enemyAnim[enemies[i].type].length then
176.       enemies[i].frame = 0
177.     endif
178.   endif
179. repeat
```

The Program So far

Let's take a look at the entirety of the project so far. Make sure you're matching up, then in the next and final tutorial we'll cover how to add your own ideas into the project:

```
1. background = loadImage( "Kenney/backgrounds", false )
2. tilesheet = loadImage( "Kenney/superPlatformPack", false )
3. chrSheet = loadImage( "Kenney/characters", false )
4.
5. playerX = 0
6. playerY = 0
7.
8. moveSpeed = 5
9.
10. idle = 0
11. walk = 1
12. jump = 2
13. hit = 3
14.
15. state = idle
```



```

16.
17. anim = [
18.   [ .start = 96, .length = 1 ],
19.   [ .start = 97, .length = 11 ],
20.   [ .start = 95, .length = 1 ],
21.   [ .start = 94, .length = 1 ]
22. ]
23.
24. animationFrame = 0
25.
26. gravity = 1
27. velocity = 0
28.
29. jumpTimer = 0
30. oldA = 0
31.
32. screenX = 0
33. screenY = 0
34.
35. tiles = [ 121, 138, 128, 129, 130 ]
36.
37. level = [
38.   [ -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 ],
39.   [ -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 ],
40.   [ -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, 2, 3, 4, -1, -1, -1, -1, -1, -1, -1, -1 ],
41.   [ 1, 1, 1, 1, -1, -1, 1, 1, 1, 1, 1, 1, -1, -1, -1, 1, 1, 1, 1, 1, 1, 1, 1 ],
42.   [ 0, 0, 0, 0, -1, -1, 0, 0, 0, 0, 0, 0, -1, -1, -1, -1, 0, 0, 0, 0, 0, 0, 0 ],
43.   [ 0, 0, 0, 0, -1, -1, 0, 0, 0, 0, 0, 0, -1, -1, -1, -1, 0, 0, 0, 0, 0, 0, 0 ]
44. ]
45.
46. levelHeight = 12
47. levelOffset = levelHeight - len( level )
48. tsize = 0
49.
50. coin = 0
51.
52. active = 0
53. collect = 1
54. inactive = 2
55. death = 3
56.
57. items = [
58.   [ .type = coin, .x = 7, .y = 1, .state = active ],
59.   [ .type = coin, .x = 8, .y = 0, .state = active ],
60.   [ .type = coin, .x = 9, .y = 0, .state = active ],
61.   [ .type = coin, .x = 10, .y = 1, .state = active ]
62. ]
63.
64. itemAnim = [
65.   [ .start = 154, .length = 1 ]
66. ]
67.
68. playerCoins = 0
69.
70. slime = 0
71.
72. enemies = [
73.   [ .type = slime, .x = 20, .y = 1, .state = active, .frame = 0, .velocity = 0, .dir = 0.05 ]
74. ]
75.
76. enemyAnim = [
77.   [ .start = 165, .length = 2 ]
78. ]
79.
80. loop
81.   clear()
82.
83.   c = controls( 0 )
84.
85.   screen_w = gwidth()
86.   screen_h = gheight()
87.   scale = screen_h / ( tileSize( tilesheet, 121 ).y * levelHeight )
88.   tSize = scale * tileSize( tilesheet, 121 ).y
89.   pSize = tileSize( chrSheet, 96 ) * scale
90.
91.   if playerX - screenX < screen_w * 0.4 then
92.     screenX -= moveSpeed
93.   endif
94.   if playerX - screenX > screen_w * 0.6 then
95.     screenX += moveSpeed
96.   endif
97.   if screenX < 0 then

```

```

98.     screenX = 0
99. endif
100.
101. drawImage( background, -screenX / 2, -screenY, screen_h / imageSize( background ).y )
102.
103. for row = 0 to len( level ) loop
104.     for col = 0 to len( level[0] ) loop
105.         if level[row][col] >= 0 then
106.             x = col * tsize
107.             y = ( row + levelOffset ) * tsize
108.             drawSheet( tilesheet, tiles[level[row][col]], x - screenX, y, scale )
109.         endif
110.     repeat
111. repeat
112.
113.     for i = 0 to len( items ) loop
114.         if items[i].state != inactive then
115.             x = items[i].x * tSize
116.             y = ( items[i].y + levelOffset ) * tSize
117.             if playerX + pSize.x > x and playerX < x + tSize and
118.                 playerY + pSize.y > y and playerY < y + tSize and
119.                 items[i].state == active
120.             then
121.                 playNote( 0, 3, 1046.50, 1, 20, 0.5 )
122.                 playNote( 1, 3, 1396.71, 1, 10, 0.5 )
123.                 playerCoins += 1
124.                 items[i].state = collect
125.             endif
126.             if items[i].state == collect then
127.                 items[i].y -= 0.15
128.                 if items[i].y < -1 then
129.                     items[i].state = inactive
130.                 endif
131.             endif
132.             drawSheet( tilesheet, itemAnim[items[i].type].start, x - screenX, y, scale )
133.         endif
134.     repeat
135.
136. drawSheet( tilesheet, 154, 10, 10, scale )
137. drawText( 10 + tSize * 0.75 + 10, 10, tSize * 0.75, grey, playerCoins )
138.
139. for i = 0 to len( enemies ) loop
140.     if enemies[i].state != inactive then
141.         x = enemies[i].x * tSize
142.         y = ( enemies[i].y + levelOffset ) * tSize
143.         eAnimStart = enemyAnim[enemies[i].type].start
144.         eSize = tileSize( chrSheet, eAnimStart + enemies[i].frame ) * scale
145.
146.         if enemies[i].state != death then
147.             enemies[i].velocity += gravity
148.             if !collision( x + eSize.x / 2, y + eSize.y + enemies[i].velocity / eSize.y ) then
149.                 enemies[i].y += enemies[i].velocity / tSize
150.             else
151.                 enemies[i].y = int( ( enemies[i].y + enemies[i].velocity / tSize + eSize.y / tSize ) ) - eSize.y / tSize
152.                 enemies[i].velocity = 0
153.             endif
154.             if playerX + pSize.x > x and playerX < x + eSize.x and
155.                 playerY + pSize.y > y and playerY < y + eSize.y and
156.                 enemies[i].state == active and velocity > 0
157.             then
158.                 enemies[i].state = death
159.             endif
160.             if !collision( x + eSize.x / 2 + enemies[i].dir * tSize, y + eSize.y ) then
161.                 enemies[i].dir = -enemies[i].dir
162.             else
163.                 enemies[i].x += enemies[i].dir
164.             endif
165.         else
166.             enemies[i].y += 8 / tSize
167.             y += 8
168.             if y > screen_h then
169.                 enemies[i].state = inactive
170.             endif
171.         endif
172.
173.         drawSheet( chrSheet, eAnimStart + enemies[i].frame, x - screenX, y, scale )
174.         enemies[i].frame += 0.05
175.         if enemies[i].frame >= enemyAnim[enemies[i].type].length then
176.             enemies[i].frame = 0
177.         endif
178.     endif
179. repeat

```

```

180.
181.     if c.a and jumpTimer < 12 then
182.         jumpTimer += 1
183.         velocity -= 8 / jumpTimer
184.         state = jump
185.     endif
186.
187.     if oldA and !c.a then
188.         jumpTimer = 12
189.     endif
190.
191.     oldA = c.a
192.
193.     velocity += gravity
194.
195.     if !collision( playerX + psize.x / 2, playerY + pSize.y + velocity ) then
196.         playerY += velocity
197.     else
198.         playerY = int( ( playerY + velocity + pSize.y ) / tSize ) * tSize - pSize.y
199.         velocity = 0
200.         jumpTimer = 0
201.         state = idle
202.     endif
203.
204.     if c.right and !collision( playerX + pSize.x / 2 + moveSpeed, playerY + pSize.y -1 ) then
205.         playerX += moveSpeed
206.         if state != jump then
207.             state = walk
208.         endif
209.     endif
210.
211.     if c.left and !collision( playerX + pSize.x / 2 - moveSpeed, playerY + pSize.y - 1 ) then
212.         playerX -= moveSpeed
213.         if state != jump then
214.             state = walk
215.         endif
216.     endif
217.
218.     animationStart = anim[state].start
219.
220.     if animationFrame >= anim[state].length then
221.         animationFrame = 0
222.     endif
223.
224.     drawSheet( chrSheet, animationStart + animationFrame, playerX - screenX, playerY, scale )
225.
226.     animationFrame += 0.2
227.
228.     update()
229. repeat
230.
231.     function collision( x, y )
232.         tileX = int( x / tsize )
233.         tileY = int( y / tsize ) - levelOffset
234.
235.         result = true
236.
237.         if tileY < 0 or tileY >= len( level ) or tileX < 0 or tileX >= len( level[0] ) then
238.             result = false
239.         else
240.             if level[tileY][tileX] < 0 then
241.                 result = false
242.             endif
243.         endif
244.     return result

```

Functions and Keywords used in this tutorial

clear(), controls(), drawImage(), drawSheet(), drawText(), else, endif, for, function, gHeight(), gWidth(), if, int(), len(), loadImage(), loop, playNote(), repeat, return, tileSize(), then, to, update()

TUTORIALS

Basic Game Tutorial 8: Customise!

Now that we've got our completed game, it's high time you added your own features. In this final tutorial we'll cover how to add parts to the level, along with how to create more items and enemies.

The code we have written will handle everything we throw at it. This means all we need to do is add information to our arrays and we should see everything happen.

Customise the Level

Let's first take a look at how to add more parts to the level:

```
35. tiles = [ 121, 138, 128, 129, 130 ]
```

Take a look at this line in your program. This small array holds all the tiles we want to use to design our level.

The level is defined here:

```
37. level = [  
38. [ -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 ],  
39. [ -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 ],  
40. [ -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, 2, 3, 4, -1, -1, -1, -1, -1, -1, -1 ],  
41. [ 1, 1, 1, 1, -1, -1, 1, 1, 1, 1, 1, 1, -1, -1, -1, -1, 1, 1, 1, 1, 1, 1 ],  
42. [ 0, 0, 0, 0, -1, -1, 0, 0, 0, 0, 0, 0, -1, -1, -1, -1, 0, 0, 0, 0, 0, 0 ],  
43. [ 0, 0, 0, 0, -1, -1, 0, 0, 0, 0, 0, 0, -1, -1, -1, -1, 0, 0, 0, 0, 0, 0 ],  
44. ]
```

As mentioned in the earlier tutorials, these numbers are used as indexes into the **tiles array**. When the code reads a **1** in the **level array**, it finds the tile number found in **tiles[1]** to put on screen.

We can freely add numbers into our **level array** and the **for loops** which draw the level will handle the drawing for us. Let's add another platform to jump on. Take a look at the edited **level array** below:

```
37. level = [  
38. [ 1, -1, -1, -1, -1, 1, -1, -1, -1, 1, -1, -1, -1, 1, -1, -1, -1, 1, -1, -1, -1, 1, -1, -1 ],  
39. [ 1, -1, -1, -1, 1, 1, -1, -1, 1, 1, -1, -1, 1, -1, -1, 1, 1, -1, -1, -1, 1, 1, 1 ],  
40. [ 1, -1, -1, -1, 1, 1, -1, -1, 1, 1, -1, -1, 2, 3, 4, -1, -1, 1, 1, -1, -1, 1, 1, 2, 3, 4, 1, 0, 0 ],  
41. [ 1, 1, 1, 1, -1, -1, 1, 1, 1, 1, 1, 1, -1, -1, -1, -1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0 ],  
42. [ 0, 0, 0, 0, -1, -1, 0, 0, 0, 0, 0, 0, -1, -1, -1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],  
43. [ 0, 0, 0, 0, -1, -1, 0, 0, 0, 0, 0, 0, -1, -1, -1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],  
44. ]
```

Here we have added 7 numbers to the end of each row in the **level array**. The numbers we have added create a floating platform and a tall platform to jump on. We must put **-1** in every empty tile.

Run the program to see the new section of the level we built. Try and build your own section! Add numbers to the end of each row in the **array**, putting **-1** in the tiles you want to leave empty.

Why not add some more tiles to draw too? All we need to do is add a couple more to the **tiles array**:

```
35. tiles = [ 121, 138, 128, 129, 130, 78, 95 ]
```

Here we've added two more tiles to our **array**. Since they are the 5 and 6 elements of the **array**, if we use the numbers 5 and 6 in our **level array**, we'll be using our new tiles!

Feel free to completely re-design the level from scratch! No need to stick with what we've used. You might want a totally different design for your game.

Adding Items

Before we go about adding completely new items to our game, let's begin with adding another coin, since the code is already in place for this.

This part will take place in the **items array**:

```
55. coin = 0
56.
57. items = [
58.     [ .type = coin, .x = 7, .y = 1, .state = active ],
59.     [ .type = coin, .x = 8, .y = 0, .state = active ],
60.     [ .type = coin, .x = 9, .y = 0, .state = active ],
61.     [ .type = coin, .x = 10, .y = 1, .state = active ]
62. ]
```

Here we have the **items array** which stores all the information about the items we have in the game. If we want to add a coin, we simply need to create another entry in this **array**:

```
55. coin = 0
56.
57. items = [
58.     [ .type = coin, .x = 7, .y = 1, .state = active ],
59.     [ .type = coin, .x = 8, .y = 0, .state = active ],
60.     [ .type = coin, .x = 9, .y = 0, .state = active ],
61.     [ .type = coin, .x = 10, .y = 1, .state = active ],
62.     [ .type = coin, .x = 14, .y = 0, .state = active ]
63. ]
```

Line 62 contains our new coin, we have set an **x** position of 14. Remember, the **x** and **y** positions here are in **level coordinates** rather than **pixel coordinates**. The 14 really means column number 14. Experiment with different numbers here to see the effects. For the **y** position, we use a lower number to move the item higher. You can put negative numbers here to make the items even higher!

Let's create a whole new kind of item. A mushroom for example!

First we'll need to create a new item type. This will be used as an index into the **itemAnim array**:

```
55. coin = 0
56. mush = 1
```

Done. We now have a label for the mushroom item type. Now let's create an entry into the **items array** which contains the location and state of the item:

```
55. coin = 0
56. mush = 1
```

```

57.
58. items = [
59.     [ .type = coin, .x = 7, .y = 1, .state = active ],
60.     [ .type = coin, .x = 8, .y = 0, .state = active ],
61.     [ .type = coin, .x = 9, .y = 0, .state = active ],
62.     [ .type = coin, .x = 10, .y = 1, .state = active ],
63.     [ .type = coin, .x = 14, .y = 0, .state = active ],
64.     [ .type = mush, .x = 3, .y = 2, .state = active ]
65. ]

```

We're not done yet! Now that we have a new item in our **items array** the **for loop** which draws them will attempt to use the **mush variable** as an index into the **itemAnim array**. The only problem is, we don't have an entry for it in the **itemAnim array**!

Creating one is very easy. We just need the tile number of the item:

```

67. itemAnim = [
68.     [ .start = 154, .length = 1 ],
69.     [ .start = 245, .length = 1 ]
70. ]

```

As you can see, we've added an entry into the **array** above. Line 69 now contains an structure which is element [1] of the **itemAnim array**. The **.start** property contains the tile of the item to put on screen.

Try to add some more items!

Adding Enemies

Adding enemies is almost exactly the same process as adding items. Let's go to the enemy array:

```

74. slime = 0
75.
76. enemies = [
77.     [ .type = slime, .x = 20, .y = 1, .state = active, .frame = 0, .velocity = 0, .dir = 0.05 ]
78. ]

```

You know how it goes! First, let's create a new enemy type **variable**. We'll use a spider this time:

```

74. slime = 0
75. spider = 1

```

Next, we need to create the entry into the **enemies array**:

```

77. enemies = [
78.     [ .type = slime, .x = 20, .y = 1, .state = active, .frame = 0, .velocity = 0, .dir = 0.05 ],
79.     [ .type = spider, .x = 8, .y = 1, .state = active, .frame = 0, .dir = 0.05 ]
80. ]

```

Done! Now we just need to put the animation information into the **enemyAnim array**:

```

82. enemyAnim = [
83.     [ .start = 165, .length = 2 ],
84.     [ .start = 190, .length = 2 ]
85. ]

```

On line 84 we create the new entry for the spider. The tile number is 190 and lasts for two frames of animation. Once this information is in the array, our job is done!

Taking it Further

Of course, even if we add lots of new items to our game, they will only ever behave in the same way as the coins as things are currently.

If you wanted specific things to happen when we pick up various items, you will have to write this code yourself! Since we know what type of item is being drawn with the `.type` property, you can add some **if statements** to make different things happen depending on the item type.

It's a good idea to create copies of your program just in case things go wrong. You can always find a completed copy of the program in the tutorial.

You deserve a huge congratulations for making it through this project!

Keep practicing and keep improving!

FUZE⁴

 NINTENDO
SWITCH™

www.fuzearena.com

