FUZE

(teaching kids to code)

Coding Projects

Book 2 - The Next Level

Written by David Silvera

For Loops

Arrays

Loading & Drawing Images

Functions

Structures

Learn techniques that will take your programming skills to the next level!

Learn to code with FUZE^4

NINTENDO SWITCH

Introduction & Contents

Hello there! Good to see you again.

So, you want to to take your code to the next level?

Now that we've gotten used to the basics of program flow, storing data in variables and how loops work, we can talk about more advanced techniques which will give you a huge boost of confidence in your programming.

This project book is all about storing and accessing **data**. Setting up our data in the right way can save us massive amounts of time and provide opportunities for some rather impressive projects!

Over the next few pages, you'll be introduced to some new concepts - **For Loops**, **Arrays**, **Functions** and **Structures**.

Mastering these techniques takes time, patience and practise. Don't be disheartened if you find some of these tricky to understand at first - we know you can do it.

Good luck, and see you in the first project!

Contents:

FUZE

Project: 1
“For the Love of Loops”

It’s a strange name, **For Loop**, isn’t it? Doesn’t really make any sense at first. Well, by the end of this page you’ll know exactly what one is and how you can use this incredibly useful tool in your own programs!

A **For Loop** is a special type of **loop** which repeats a certain number of times. It can repeat any number of times you need it to and what’s more, it comes with a very clever feature built-in.

1

```
1.  for num = 0 to 10 loop
2.      print( num )
3.      update()
4.      sleep( 1 )
5.  repeat
```

Here’s what a **for loop** looks like. Type in the program on the left into a new **FUZE⁴** project and run. You should see the numbers 0-9 print across the screen.

Just like with a normal **loop**, the instructions between the **loop** and **repeat** keywords will keep happening. However, this time they will only happen for a total of 10 times.

So what about this **num variable**? Well, here’s the clever part.

For loops contain **variables** which act as a counter during the **loop**. You can use these **variables** within the **loop** to make clever things happen!

In the example, our **variable** is called **num** because we want to **print()** the number. This could be called anything of course!

While the **loop** goes around, this **variable** increases by **1** each time. The first time **FUZE⁴** reads line 2 our **num variable** contains a **0**. On the next time **num = 1**, then **num = 2**, and so on.

When we set up a **for loop**, we must write the start and end values on the line. However, there is a small complication. **FUZE⁴** will count *up to but not including* the last number. This is why we never see the number **10**.

2

```
1.  for num = 0 to 10 step 2 loop
2.      print( num )
3.      print()
4.      update()
5.      sleep( 0.1 )
6.  repeat
```

Sometimes you might need your **for loop** to count up in a different number than **1** each time.

For this, you’ll need to use **step**.

By adding **step 2** before the **loop** keyword, we can make our **for loop** count up in **2**’s instead!

We’ve added an empty **print()** line to the program to make our numbers print down the screen instead of across, and decreased the **sleep()** time to make the program a bit faster.

Experiment by changing the range of numbers in the first line. Get a feel for what to expect when trying different things. It might not seem like much yet, but this is one of the *most vital techniques* to understand more complex programs.

Great job! Now you understand what a **for loop** is, the next step is learning how to use them. Their most useful application is when combined with another very useful technique. See you on the next page!

HACKER CHALLENGE:

By using a negative step, can you make your For Loop count backwards instead?

Can you use our variable counter so that the sleep() command waits for longer each time?

Remember, this book is all about **data**. In programming, we can do some very impressive things by simply setting up our data in a different way.

To take our programming skills to the next level, we'll need to cover a type of data storage called an array. This is another invaluable tool for programming. Every single video game you've ever played used lots and lots of arrays.

An array is a table of **variables**. We use them to store multiple pieces of information for easy access. Think of it like a chest of drawers, where each drawer has a number. These numbers ***always start at 0***.

This project is to create a Fortune Teller game. We will set up a simple array of various sentences. Next, we will need to ask a question and receive a random answer chosen from our array. Let's get into it!

```
array fortune[4]
```

1

First order of business is to create our array.

Writing the statement above creates an empty array of 4 elements. An empty chest of drawers with 4 drawers! Each element (drawer) is given a number. So, our array looks something like this:

fortune[0]	fortune[1]	fortune[2]	fortune[3]

Of course, at the minute our array is completely empty. Let's put in some information! Below we are adding a sentence to each element:

```
array fortune[4]
fortune[0] = "It is certain! "
fortune[1] = "You might be in luck! "
fortune[2] = "It's not looking good... "
fortune[3] = "Definitely not. "
```

2

There we have it!

To store information in an array, write the element you would like to access followed by an equals and the information you want to store.

There is another way we can set this up. Arrays are very flexible in FUZE⁴.

```
fortune = [
    "It is certain! ",
    "You might be in luck! ",
    "It's not looking good... ",
    "Definitely not. "
]
```

Here is a **different** way to set up your array. Both of these ways work in the same way. Use whichever one you find most comfortable to look at!

Which one you choose depends on what you are going to use your array for.

When defining an array like this, *you must use commas to separate the information*. The last piece of information does not need a comma after it.

Either way, it's up to you which method to choose. You could even write the entire thing on one line, as long as you get the punctuation correct! This might be a bit tricky to read though...

Now that we've got our array set up, let's put it to use! Turn the page and we'll finish the program.

HACKER CHALLENGE:
Add at least 2 more sentences to your array.

```
1.  array fortune[4]
2.  fortune[0] = "It is certain! "
3.  fortune[1] = "You might be in luck! "
4.  fortune[2] = "It's not looking good... "
5.  fortune[3] = "Definitely not. "
6.
7.  loop
8.      clear()
9.      print( "Ask me anything! \n " )
10.         update()
11.         sleep( 1 )
12.         question = input( "Type your question here. " )
13.         sleep( 1 )
14.         print( "Are you ready to know your fortune? \n" )
15.         update()
16.         sleep( 1 )
17.         print( "The answer to your question is... \n" )
18.     update()
19.         sleep( 1 )
20.     num = random( 4 )
21.         print( fortune[num], "\n" )
22.     update()
23.     sleep( 1 )
24.     print( "Let's play again! " )
25.     update()
26.     sleep( 2 )
```

3

Let's go over what's happening here.

We have added a large section of code from line 7 onwards.

We are using a **loop** so that our program continues to run seamlessly until we decide to stop.

For the most part, we are using the same technique as the **"Quiz"** project - using **print()** statements followed by **update()** and **sleep()** to provide pauses between printing messages.

Just like the quiz project, we use the **input() function** to allow the user to type in a question.

We must store their question as a **variable**, even if we don't use it.

Once they type their question and press the **+** button, we print some sentences to build the tension!

The important part of our program is line 21. This is where we actually **access** the information from our **array**.

To give a random answer, we must create a random **index**. Line 20 creates a **variable** called **num** which stores a random number chosen out of **4** (0-3).

We then use the **num variable** as an **index** into our **array**!

You can think of this as rolling a dice, then choosing an answer based on the number!

Well done! You've added a crucial technique to your programming skills. It may not seem like it yet, but the concept of an **array** is immensely useful. From databases to painting software, from video channels on the internet to video games of all types - they *all* use **arrays** in various ingenius ways.

Learning this technique is going to be very important in your coding journey. Make sure you read this project carefully, try to do it from the start by yourself - but don't worry if you need some help!

HACKER CHALLENGE:
Add sentences to your array until you have at least 8. You can make them about anything you like - it doesn't even have to be a fortune teller!
Make sure that your sentences actually appear in the program! HINT: You'll need to edit line 20: num = random(4)

FUZE

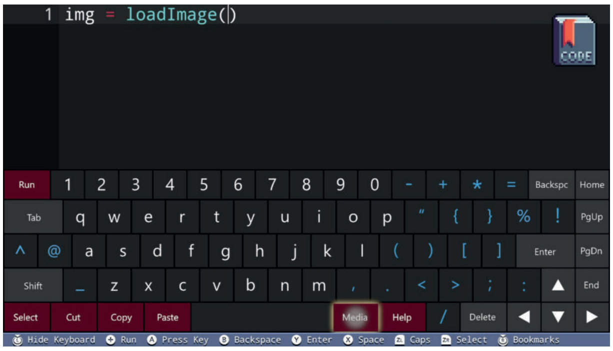
Project: 3 “Picture This…”

The **FUZE⁴** media library is full to the brim with incredible artwork to use in your creations, so let’s get using them! This project will teach you how to load an image into a program and draw it. To begin, we’ll need to load an image from the Media Library and store it in a **variable** within a program.

```
1. img = loadImage(|)
```

1 Start a new project. On line one, type the code you see on the left here. Once you’ve typed it, put your cursor *inside* the brackets.

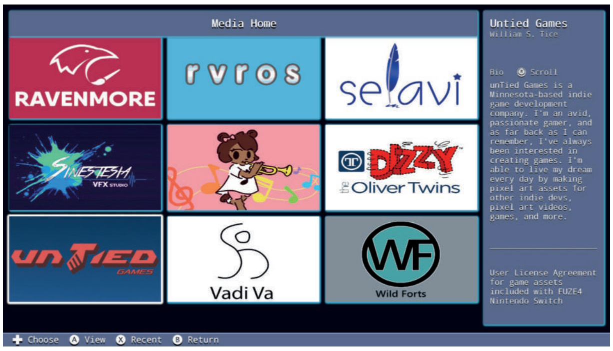
When your cursor is *inside* the brackets for **loadImage()**, you’ll see the **Media** key on the **FUZE⁴** on-screen keyboard start to flash. This means it’s ready to go into the media library for us to load an image easily!



```
1 img = loadImage()
```

CODE
Run 1 2 3 4 5 6 7 8 9 0 - + * = Backspc Home
Tab q w e r t y u i o p " { } % ! PgUp
^ @ a s d f g h j k l ( ) [ ] Enter PgDn
Shift _ z x c v b n m , . < > ; : ▲ End
Select Cut Copy Paste Media Help / Delete ◀ ▼ ▶
Hide Keyboard Run Press Key Backspace Enter Space Caps Select Bookmarks

As you can see on the left, the **Media** button on the keyboard is glowing. When this is happening, click on the button using the Joy-Con, or press F2 on the USB keyboard.



Untied Games
William S. Tice

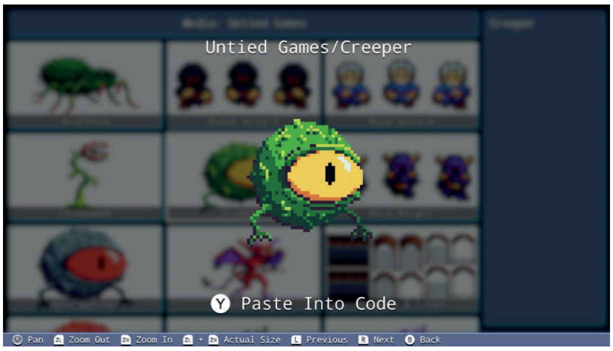
Bio Scroll

unTied Games is a Minnesota-based indie game development company. I'm an avid, passionate gamer, and as far back as I can remember, I've always been interested in creating games. I'm able to live my dream every day by making pixel art assets for other indie devs, pixel art videos, games, and more.

User License Agreement for game assets included with FUZE4 Nintendo Switch

When you go into the **Media** section, you’ll see a number of **Artist Thumbnails**. Scroll down and find **Untied Games**.

Click on their icon and find the **JRPG** folder. You’ll then see a number of awesome assets!

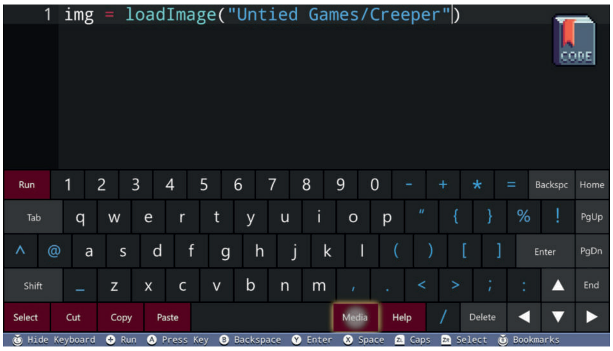


Untied Games/Creeper

Y Paste Into Code

Find the “Creeper” image and select it with the **A button**, or **Enter** on the USB keyboard.

You should see the screen on the left. At this point, notice the option underneath the image. We can now press the **Y button** to *paste* the filename into our code.



```
1 img = loadImage("Untied Games/Creeper")
```

CODE
Run 1 2 3 4 5 6 7 8 9 0 - + * = Backspc Home
Tab q w e r t y u i o p " { } % ! PgUp
^ @ a s d f g h j k l ( ) [ ] Enter PgDn
Shift _ z x c v b n m , . < > ; : ▲ End
Select Cut Copy Paste Media Help / Delete ◀ ▼ ▶
Hide Keyboard Run Press Key Backspace Enter Space Caps Select Bookmarks

Once you press the **Y button**, you will be taken back to the code editor and you’ll see the filename appear in the brackets!

This is the easiest method to get assets from the **Media** library into your code. Give it some practise and feel free to use a different image!

```
1. img = loadImage( “Untied Games/Creeper” )
```

2 Your program should look like this before continuing.

Now that we've got our image file stored in a **variable**, we can use another **function** to draw this image to the screen.

3

```
loop
    clear()
    drawImage( img, 0, 0 )
    update()
repeat
```

Add the lines on the left to your program. As you can see, we are using a simple **loop** to update the screen, with only one main command inside: **drawImage()**

In the brackets for **drawImage()**, we need the variable which stores the image and some screen co-ordinates (**x, y**). The image will be drawn from the origin point at the co-ordinate.

The origin point is the **top left** corner of the image - this includes any blank space in the picture. If we change the co-ordinate in **drawImage()**, we are changing where this **top left** corner is placed on-screen.

You might notice that the image is very small! This won't do at all. By using an extra piece of information in our **drawImage()** statement, we can change the size of our image.

4

```
loop
    clear()
    drawImage( img, 0, 0, 4 )
    update()
repeat
```

Adding an extra number after the **Y** axis position in the **drawImage()** function gives the image a scale multiplier. By putting a **4** here, we are multiplying the size of the image by **4**.

Try moving the image to somewhere else on-screen, then changing the scale multiplier.

But what if we want to do more than just move the image and change the size? We might want to spin the image around or apply a new colour to the image. Luckily, there is a function in **FUZE^4** to do just that!

5

```
loop
    clear()
    drawImageEx( img, { 0, 0 }, 90, { 4, 4 }, red )
    update()
repeat
```

Don't be scared by the strange looking brackets! These are called **curly** brackets, and we use them with something called **vectors**.

We won't be going into detail on vectors in this workbook. For now, let's explain what's going on here.

The **drawImageEx()** function allows us to do all sorts of things with an image. Here are the arguments:

drawImageEx(image, location, rotation, scale, tint)

The location of the image is just a pair of co-ordinates in curly brackets. The next value (rotation) is a number of degrees to rotate the image by.

Things are a little different for the scale - we now have separate **x** and **y** scale values in curly brackets, this gives you total control over the size of your image! Lastly, we have a tint colour to apply to the image. Try some different colours and see the effect it has!

HACKER CHALLENGE:

1. Can you add a rotation variable to your program and change it in the loop to cause your image to spin around?
2. Can you add a scale variable to your program and change it in the loop to cause your image to grow or shrink?

FUZE

Project: 4
“Out of Control”

Learning how to use the Joy-Con controllers in your own projects is a great feeling. Suddenly your programs will begin to feel like real game projects. Outside of making a game, it opens up a whole world of interactivity for any sort of program! Are you ready? It’s time to take control of this situation.

```
c = controls( 0 )
```

If we want to access the controllers, we need to use a **function** called **controls()**.

If you’ve looked at any of the **FUZE^4** demo projects, you might have seen a line of code which looks like the one just above in the grey box. The **0** in the brackets tells **FUZE^4** which pair of Joy-Con’s we are using.

Sometimes instead of a **c variable**, you’ll see a letter **j** instead. The name isn’t important, but notice that we are assigning the result of the **controls()** **function** to a **variable.**

Once we do this we can access any of the buttons, any of the control sticks and even the gyroscope for motion controls!

Programs which use the controls often need to be checking them constantly, or we wouldn’t be able to control things very well! This means we will be using a **loop**:

1

```
1. loop
2.     clear()
3.     c = controls( 0 )
4.     print( c.a )
5.     update()
6. repeat
```

If we run this program on the left, all we will see is a single **0** in the top left corner of the screen. See what happens when you press the **A button**!

Notice the strange looking **variable** on line 4. We have our **c variable**, followed by “**.a**”.

The controls **function** is special. It adds a number of *properties* to the variable we assign it to.

If we can now use **c.a** to check the **A** button, how do you think we check the **B** button? Yes. You are correct. I don’t even need to hear your answer! It really is that simple.

2

```
loop
    clear()
    c = controls( 0 )
    if c.a then
        print( “That’s the A button!” )
    endif
    update()
repeat
```

In this example we are using a simple **if statement** to display a message when we are pressing the **A button**.

Notice the **print()** statement is *inside* the **if statement** which checks **if c.a** is **true**.

When we want to check **if** the **A button** is pressed, we only need to write **if c.a then**.

If the message displays *only when the **A button** is pressed***,** your program is working perfectly!

HACKER CHALLENGE: **Can you add more If Statements to the program to display a message when the B, X and Y buttons are pressed?**

FUZE

Project: 5
“Strictly Structures”

This project will introduce you to another very useful type of data storage called **structures**. Don’t be afraid! They work very similarly to an **array**. The difference is that each “part” of the **structure** has a special name. They are called *properties*. **Arrays** have *elements*, **structures** have *properties.*

To introduce this idea properly, let’s refresh ourselves on how an **array** works. Imagine we want an **array** to store information about a person - their name, age and interests. We might set it up so it looks something like this:

```
person = [ “Dave”, 28, “Guitar” ]
```

We’ve stored the information we want in elements 0, 1 and 2 of our **array** called **person**,

Remember, we can picture this **array** as looking something like the diagram below, with each piece of information stored in its separate elements, each with its own number:

person[0]	person[1]	person[2]
“Dave”	28	“Guitar”

We could access this information with a statement like: **print(person[2])**. This is if we wanted the interests speicifcally, for example.

This is very useful but it could be even more convenient. We have to remember that the first element contains names, the second contains age and the third element contains interests. Wouldn’t it be much easier if we could *name the elements*? That’s exactly what a **structure** allows us to do! Check it out, we’ll turn our person **array** into a **structure** instead:

```
person = [ .name = “Dave”, .age = 28, .interests = “Guitar” ]
```

To make **structure** properties, we use a full stop followed by a name for the property. Our **person variable** above is now a **structure** *with three properties*: .name, .age and .interests.

We can imagine it looking something like this in the **FUZE⁴** brain:

person.name	person.age	person.interests
“Dave”	28	“Guitar”

We would now access this information with something like this: **print(person.name)**

We can upgrade this even further by combining both concepts. Below is an example of an **array** of **structures**. By adding a pair of square brackets, we can now add many **structures** into the **person array**. The **person variable** is now an **array** of 3 *elements*, each one containing a **structure** of 3 *properties*.

```
person = [
    [ .name = “Dave”, .age = 28, .interests = “Guitar” ],
    [ .name = “Luke”, .age = 21, .interests = “Robots” ],
    [ .name = “Kat”, .age = 27, .interests = “Art” ]
]
```

1 You should use information from people you know instead!

On the next page we’ll see how to use this information efficiently in a program.

HACKER CHALLENGE:
Can you add another property to your structure?

In this section, we'll look at how to use the data we've stored in our **array** of **structures**. With the latest change, we can imagine our data looking something like the diagram below:

	.name	.age	.interests
person[0]	"Dave"	28	"Guitar"
person[1]	"Luke"	21	"Robots"
person[2]	"Kat"	27	"Art"

Let's write a simple program to print the names of the people in our **array** of **structures**. Rather than writing three separate **print()** instructions, we can use a **for loop** which counts over each person, printing their name.

2

```
for i = 0 to len( person ) loop
    printAt( 0, i, person[i].name )
repeat
update()
    sleep( 3 )
```

Remember: **len(person)** is **3**, since there are 3 elements in the **person array**. The **i variable** will count from **0** up to *but not including* **3**. This means the **i variable** will be **0**, then **1**, then **2**. Using **len(person)** means we could have *any number* of people in the **array**, and the program will work!

This program is saving us time because to do this without a **loop** we would need **three** separate **printAt()** instructions! You might think "three instructions isn't many", and you'd be right - but what if our **array** contained one hundred people? This simple 3 line **for loop** would print them all!

We could improve this program to include a simple sentence which uses all of the information in our **person array**. This will be quite a long **printAt()** instruction!

3

```
for i = 0 to len( person ) loop
    printAt( 0, i, person[i].name + " is " + person[i].age + " and likes " + person[i].interests )
repeat
update()
sleep( 3 )
```

Let's break down exactly what's happening here in order to fully cement the concept into our minds.

The first time around the **loop**, our **i variable** is **0**. The **printAt()** statement will look something like this:

```
printAt( 0, 0, person[0].name + " is " + person[0].age + " and likes " + person[0].interests )
```

This will output the following sentence: "Dave is 28 and likes guitar"

The second time around the **loop**, our **i variable** is **1**. The **printAt()** statement will look something like this:

```
printAt( 0, 1, person[1].name + " is " + person[1].age + " and likes " + person[1].interests )
```

This will output the following sentence: "Luke is 21 and likes robots"

HACKER CHALLENGE:
Add more people to your array until you have a total of 5. Make people up if you can't find anyone else! Can you add more properties to the structures and include them in the sentence?

In this project we'll combine everything we've learned so far into a silly, colourful project! Using **arrays**, **structures** and **for loops**, we'll be applying these concepts into a basic template for drawing lots of images on the screen and making things happen to all of them at once using the Joy-Con controllers.

This project will be the first look at setting up a more "professional" sort of program. Remember, the theme of this workbook is **data**. Setting up our data is going to very important here, there's a lot to learn, so put your coding hats on! Start a new project and enter the code below.

We'll begin with an **array** of images to use in our program. Each element of the **array** will be a **loadImage()** instruction, containing a graphic from the artist **Untied Games**.

```
images = [
    loadImage( "Untied Games/Creeper" ),
    loadImage( "Untied Games/Man Eater" ),
    loadImage( "Untied Games/Slime" ),
    loadImage( "Untied Games/Goblin" )
]
```

1

Of course, you don't *have* to use the same images as we do! Feel free to browse the **FUZE^4** media library and paste different filenames into your code.

Important: Make sure you use single image, 2D graphics. This program won't work with 3D models or tilesheets!

Once we've completed this **array**, we'll need to create an **array** of **structures** to store all of our monsters to put on screen. We'll be setting this up a little differently this time, since we want to have *lots* of monsters on screen. Take a look at the code below:

```
numMonsters = 50
array monsters[numMonsters] = [
    .image = 0,
    .rotation = 0,
    .scale = {1, 1},
    .position = {0, 0}
]
```

2

Wow, this is already starting to look quite complex!

Don't worry - this is exactly like our **person array** from the previous project, but it's laid out a little differently.

monsters is an **array** of **50** elements, each one a **structure** containing 4 properties.

Rather than putting values in for each **structure** like we did before, we are starting with *default values* for all of the **structures**.

Now that's out of the way, we need to *populate* our **monsters array** with values. We need to set different values for all **50** elements - this will take a while if we do it manually! Instead, we will use a **for loop** and some random values to make it interesting!

3

```
for i = 0 to len( monsters ) loop
    monsters[i].image = images[random( len( images ) )]
    monsters[i].rotation = random( 360 )
    monsters[i].scale = {1, 1} * ( random( 4 ) + 1 )
    monsters[i].position = {random( gwidth() ), random( gheight() )}
repeat
```

There it is! This is quite a complicated looking **for loop**, but don't worry - it's simple when you break it down. On the next page, we'll look at each of these lines in more detail so as to understand the reasons behind them.

See you on the next page! Well done - not much more to do now!

Alright, let’s look at each of the lines in the **for loop** in more detail. What we’re doing in this **for loop** is *populating* the **array**. When we declare it on line 9, we are simply starting with default values. It’s our job now to put the things we need into our **array**!

First, we need to assign an image to the **.image** property. We have our images stored in the images **array**, so we need to select a random one then assign it to the property.

```
17.    monsters[i].image = images[random( len( images ) )]
```

It looks a bit tricky, but what we are doing is simple. We choose a random number out of the length of the images array, then use that number to index the array. We could use **random(4)** instead, but using **len(images)** allows us to *add more images* and the program will work!

Example:

len(images) = 4	How many images in the array
random(4) = 1	The random result
monsters[i].image = images[1]	The chosen image

The next two steps are more simple. We choose a random rotation angle (out of **360**) for the **.rotation** property, followed by a random scale.

```
18.    monsters[i].rotation = random( 360 )
```

We use {**1**, **1**} multiplied by a random number out of **5** (with a minimum of **1**) here. When you multiply a vector like {**1**, **1**} by a number, both of the numbers are multiplied!

```
19.    monsters[i].scale = {1, 1} * ( random( 4 ) + 1 )
```

Lastly, we give the monster a random position. The position is stored in a pair of *curly brackets* as a **vector**, since we’ll be using **drawImageEx()** to make them appear on screen.

```
20.    monsters[i].position = {random( gwidth() ), random( gheight() )}
```

Using **random(gwidth())** we are choosing a random number out of the whole screen’s **x axis**. We use **random(gheight())** for the **y axis** and there we go! We now have a random position in the property.

The next step is to build the basic program **loop**. We’ll start with just the shell, then add to it on the next page.

We simply want to **clear()** and **update()** the screen with a **controls()** check between them.

4

```
23. loop
24     clear()
25.    c = controls( 0 )
26.
27.    update()
28. repeat
```

The final stretch! We now need to write a **for loop** which will put *each* image from the **array** on screen, in the right position, scale and rotation.

The new code below is line 27 to 31. The outside loop has been left in for clarity.

5

```
23. loop
24      clear()
25.     c = controls( 0 )
26.
27.     for i = 0 to len( monsters ) loop
28.         monsters[i].rotation += c.ly * 10
29.         monsters[i].scale += c.ry
30.         img = monsters[i].image
31.         pos = monsters[i].position
32.         rot = monsters[i].rotation
33.         scl = monsters[i].scale
34.         drawImageEx( img, pos, rot, scl )
35.     repeat
36.
37.     update()
38. repeat
```

We use a **for loop** to count over each monster in the **monsters array**. This is the same technique as before, using **len(monsters)** as the upper limit.

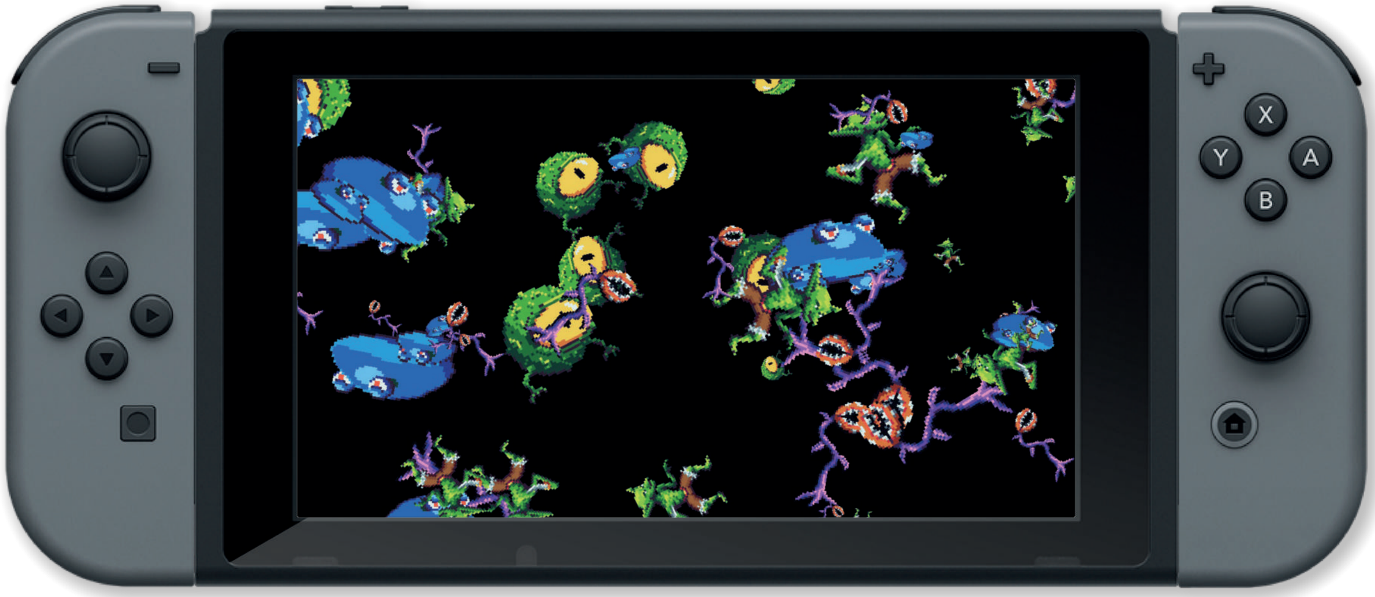
First we add the left control stick **y axis** value (**c.ly**) multiplied by 10 to the rotation. Next, we add the right control stick **y axis** value (**c.ry**) to the scale. This gives us control over the rotation and scale of the images using the control sticks!

For each monster, we want to use each of the pieces of information contained in the monster's **structure** inside a **drawImageEx()** statement.

To avoid having a very long **drawImageEx()** function, we have created some short **local variables** to hold the information from the **structure**, before using them on line 34 with the **function**.

We're finished! Press the **+** button to run your program and experience the glory of monster madness!

Move the left and right control sticks up and down to rotate and change the size of your monsters!



Well done for completing this difficult project - this might be the most technical one we've done so far. Make sure you thoroughly read through the book and understand everything we've done up to now. We'll be moving on to another tricky concept next, followed by another large project. Get ready!

HACKER CHALLENGE:
1. Add some more monster images to the images array
2. Change the number of monsters in your program

FUZE

Project: 7

"Perfectly Functional"

Time to level up your skills even further! In this project we'll be diving deeper into what exactly a **function** is, and even how to create our own. This incredibly useful technique is *guaranteed* to level you up!

We've already used functions plenty, but perhaps you don't realise it yet. Every time we write **print()** or **ink()**, we are using a **function**. We can also make our own to do anything we want!

A **function** is a series of instructions which receives an *input* and gives us an *output* - even if that input and output is nothing. Think of it like a separate little program within your main program.

Let's make a very simple example of a custom **function**. Imagine we want to convert a number of miles into kilometres. We would need to *input* a number of miles, and the function will *output* a number of kilometres. Take a look at the examples below - no need to write them.

```
function milesToKm( miles )
return miles * 1.609
```

We begin with the **function** keyword. This tells **FUZE⁴** we're about to create our own **function**. Next, we name it. Here we've called it **milesToKm**.

Next, in the brackets we tell **FUZE⁴** the information to expect. We want this **function** to convert a number, so we must provide it with one. We name this piece of information as a **variable** which is used *only within that function.* This is called a **local variable**.

On the second line, we use the **return** keyword and state the value we want to return. To convert miles into kilometres, we multiply the number of miles by **1.609**.

```
myWalk = milesToKm( 3 )
print( myWalk )
```

We could use this to figure out how many kilometres a 3 mile walk is. We create a **variable** and assign the result of our new **function** to it!

It doesn't stop there! As we said before, **functions** don't always need to **return** anything. Sometimes they can simply execute a sequence of instructions. Here's a very practical example:

```
ink( fuzePink )
print( "Hello!" )
update()
sleep( 1 )
```

Consider the code on the left - no need to write it yet.

We have a program which prints "Hello!" in **fuzePink** and sleeps for one second. These four lines are necessary if we want to make text appear with a delay afterwards. Remember the quiz game? We used this technique repeatedly to make our text appear at certain times.

We can write a **function** to use these four lines of code over and over again without having to type them each time. Pretty useful! Alright, time to actually write some code!

```
function fuzePrint( colour, text, delay )
    ink( colour )
    print( text )
    update()
    sleep( delay  )
return void
```

1 In this example, we are providing our **function** with a colour, some text, and a time delay. These **local variables** are all used in the instructions within the **function**.

With that done, we can use this as many times as we want in our program!

```
fuzePrint( fuzePink,  "Hello!", 1 )
```

2 Simply call your **function** and put the desired colour, text and delay time in the brackets!

HACKER CHALLENGE:

1. Can you add a size control to your function using textSize()?

Project: 8
“Mega Quiz”

We’re going back to our roots in this project. It’s time to use *everything* we’ve learned so far to make a program. That’s right - **loops**, **variables**, **if statements**, **arrays**, **for loops**, **structures** *and* **functions**!

In this project we’ll be making the *ultimate* quiz game. This will take the ideas we worked on in the previous book to another level. This **will** be the most impressive quiz project you’ve ever worked on, the pinnacle of interactive trivia-based question games. Are you ready?

The biggest difference between the old quiz and this one will be the way the program works. In the first book, we wrote each question as a separate list of instructions - it involved repeating ourselves a lot, and lots more typing than we really need.

There is a way to turn our entire quiz into a single **loop** - it just requires us to set up our **data** more effectively. Let’s use what we’ve learned about **structures** to set up a question and answer table.

For a bit of extra brilliance, we’ll make this a multiple-choice quiz. This means we need an **array** of **structures**, where each element of the **array** is a structure of 3 properties: the question, the possible answers, and the correct answer.

1

```
quiz = [
    [
      .question = “What is an array?”,
      .answers = [ “A table of data”, “A sausage”, “A type of small dog” ],
      .correct = 0
    ],
    [
      .question = “What is a loop?” ,
      .answers = [ “A type of cheese”, “A repeated set of instructions”, “A dance” ],
      .correct = 1
    ]
]
```

Here is what the array should look like. This one contains 2 questions, stored in **quiz[0]** and **quiz[1]**. We could access the question of either with a statement like: print(**quiz[0].question**)

The **.answers** property contains another array of 3 elements, each one a possible answer. The **.correct** property stores the number which corresponds to the correct answer. For example, the **.answers** property in question 1 contains:

.answers[0] - **“A table of data”**
.answers[1] - **“A sausage”**
.answers[2] - **“A type of small dog”**

The correct answer is of course **.answers[0]**, so the **.correct** property stores a **0**!

HACKER CHALLENGE:
Make sure to use your own questions!

With our questions and answers all set up, the next step is to write a useful **function** which we'll use again and again throughout our program. Remember the special fuzePrint **function** from the previous project? Well, here it is again! This will help us avoid repeating ourselves when we get to the main part of the program.

Our **function** is going to print text in a position on screen at a given size, in a given colour, with a given delay afterwards. Write the code below in your program:

```
function qPrint( text, x, y, size, colour, delay )
    textSize( size)
    ink( colour )
    printAt( x, y, text )
    update()
    sleep( delay  )
return void
```

2

We've called our **function** **qPrint**! I'm sure you can guess what the **q** stands for!

We provide it with 6 *parameters*:

The text we want to print, the position on screen, the size and colour for our text, and the number of seconds to wait.

With this done, we can now use the **qPrint** **function** in our program. For example, we could write:

```
qPrint( "Hello!", 0, 0, 30, bisque, 2 )
```

This would print **"Hello!"** in the top left of the screen at a text size of **30**, with bisque ink and a delay of **2** seconds.

The next step is to set up a couple of useful global **variables** for the quiz. We need to keep track of the player score and the answer they choose. Simple enough!

```
answer = 0
score = 0
```

3

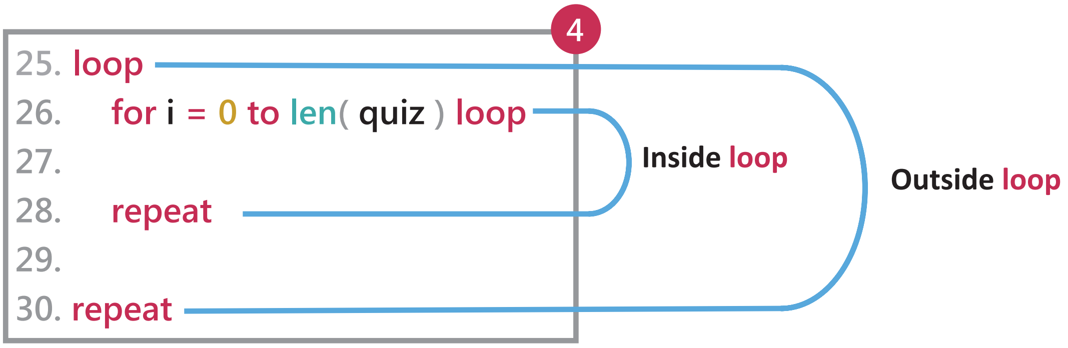
These two global **variables** are all we need for the whole program! Both of these **variables** will change as we play the game.

Nearly finished with the setup! The last step is to write the basic template for our main **loop**.

This will be a **loop** *inside* a **loop**. In fact, there will be a few **loops** going on by the time we're finished!

Below is the basic template of our main program. There are two **loops** - one *outside* infinite loop, and one *inside* **for loop**. The inside **loop** will count over each question in the **quiz** **array**.

4



```
25. loop
26.     for i = 0 to len( quiz ) loop
27.
28.     repeat
29.
30. repeat
```

Inside the **for loop** (the inside **loop**) is where the quiz itself will be taking place. We'll be working within that **loop** for the next couple of sections.

On line 29, *after* the **for loop** is where we will write the code to see how the player performed during the quiz, and let them know their score. Then, the whole quiz will **loop** around from the beginning!

Our next task is to write the playable part of the quiz. We need to use our data in the **quiz** array in this section, we’ll also be putting the **qPrint** function to use!

The first order of business is to print the question on screen, followed by the possible answers. We’ll also need a text prompt to let the player know how to select a question.

5

```
26.   for i = 0 to len( quiz ) loop
27.       clear()
28.       qPrint( quiz[i].question, 0, 0, 30, white, 2 )
```

We’ve written line 26 here as a guide - make sure you don’t type that line twice! It is there to show you where the new lines go.

We clear() the screen for a fresh page, and then use **qPrint** to print the .question *property* of the first *element* in the **quiz** array.

Feel free to try different sizes, placements and colours! We’ve also used a delay of 2 seconds to make sure the player has time to read the question.

Next, we need to print the multiple-choice answers to the question in a clear way. We’ll need to use 3 separate **qPrint** instructions here, since we want each answer to correspond to a different button:

6

```
29.       qPrint( “A: “ + quiz[i].answers[0], 0, 2, 30, white, 0.5 )
30.       qPrint( “X: “ + quiz[i].answers[1], 0, 3, 30, white, 0.5 )
31.       qPrint( “Y: “ + quiz[i].answers[2], 0, 4, 30, white, 0.5 )
```

We’ve used a very short delay time of 0.5 here to keep things moving.

Now let’s give the player a prompt so they know how to select an answer!

7

```
32.       qPrint( “Make your choice using the joy-con buttons! “, 0, 6, 40, white, 1 )
```

Done! The four lines of code we just wrote will handle *all of the printing for the entire quiz!* This is the power of for loops and good data setup. We could add as many questions as we want to our **quiz** array, and we don’t need to touch this part!

Your whole main loop should look like the code below. Double check it against yours and make sure it all matches up before we move on! The next section will be added *just before* the repeat on line 35. Give yourself a couple of lines of space like we have in the example.

```
25. loop
26.   for i = 0 to len( quiz ) loop
27.       clear()
28.       qPrint( quiz[i].question, 0, 0, 30, white, 2 )
29.       qPrint( “A: “ + quiz[i].answers[0], 0, 2, 30, white, 0.5 )
30.       qPrint( “X: “ + quiz[i].answers[1], 0, 3, 30, white, 0.5 )
31.       qPrint( “Y: “ + quiz[i].answers[2], 0, 4, 30, white, 0.5 )
32.       qPrint( “Make your choice using the Joy-Con buttons! “, 0, 6, 40, white, 1 )
33.
34.
35.   repeat
36.
37. repeat
```

Well done for making it this far! We're getting into the mechanics of how to select an answer now. For this, we'll need to write a little routine inside our **for loop**. Get ready for some **if statements**!

8

```
34.         press = false
35.         while press == false loop
36.
37.         repeat
38.     repeat
```

The new code we've added are lines 34 - 37. The **repeat** keyword on line 38 should already be there for you, this is the **repeat** which matches the **for i = 0 to len(quiz)** instruction on line 26.

Once we've added this new **while loop**, we'll be working inside it, on line 36.

But what is going on with this strange looking **while loop**?

Before we start it, we define a **local variable** called **press**. This **variable** will tell us when a button is pressed to select an answer. We want this **loop** to repeat until the player presses a button.

Because we wrote **while press == false loop**, this **loop** will *stop* when the **press variable** becomes **true**.

This will allow us to easily stop the the **loop** when the player selects a question!

9

```
34.         press = false
35.         while press == false loop
36.             c = controls( 0 )
37.             if c.a then
38.                 press = true
39.                 answer = 0
40.             endif
41.             if c.x then
42.                 press = true
43.                 answer = 1
44.             endif
45.             if c.y then
46.                 press = true
47.                 answer = 2
48.             endif
49.         repeat
```

On the left we have the code which allows the player to input an answer. The new code is from line 36 to line 48.

This **loop** will continue to run until the player selects an answer using the Joy-Con controller buttons.

Inside our **while loop**, we first call the **controls() function** and assign it to a **variable** called **c**.

We can then check the buttons we want by using three **if statements**. Since each button provides a different answer, we have used three separate **if statements** here, each one assigning a different number to the **answer variable**.

In each **if statement**, we make the **press variable true** in order to break out of the **loop**.

Give yourself one line of space, then on line 51 we must write the **if statement** which checks the answer against the correct answer for the current question. This section should be very familiar from the previous quiz game! Notice the **repeat** on line 57. This **repeat** is the one tied to the **for loop** on line 26.

10

```
51.             if answer == quiz[i].correct then
52.             qPrint( "Correct! ", 0, 8, 50, green, 1 )
53.             score += 1
54.         else
55.             qPrint( "Bad Luck... Incorrect! ", 0, 8, 50, red, 1 )
56.         endif
57.     repeat
```

We’re nearly there! All that remains to do is the ending. We now have an **array** of questions, with a **for loop** counting over each one, printing the question and its answers. The user can select an answer and we check it against the correct one, printing the result.

All we need to do is to let the player know how they’ve done and write a small section to allow them to play again, returning to the very start of our main program **loop**.

1

```
59.     clear()
60.     qPrint( “Let’s see how you did... “, 0, 0, 40, white, 1 )
61.         qPrint( “You scored: “ + score + “ out of “ + len( quiz ), 0, 2, 40, white, 2 )
```

These three lines are all we need here. First we **clear()** the screen for a fresh page, then we print two messages. The first is very simple, but the second could use some explaining - although it’s nothing new!

We create a sentence by adding **strings** together. The player **score** is displayed out of **len(quiz).** In the program so far, the length of the quiz is **2** questions - but you should add many more! By using **len(quiz)** we can add questions without worrying about our print statements being incorrect later.

Our last task is to write a routine which allows the player to try again to get a better score. We’ll need another **while loop** to do this! Give yourself another line of space to keep things neat, then type the code below:

12

```
63.     press = false
64.     while press == false loop
65.         clear()
66.         c = controls( 0 )
67.         qPrint( “Press the B button to play again! “, 0, 0, 50, white, 0 )
68.         if c.b then
69.             press = true
70.             score = 0
71.             endif
72.     repeat
73. repeat
```

This **while loop** is very similar to the one we wrote before for the player answer input. First we must reset our **press variable** before the **loop** begins. Inside the **loop**, we check the controls, print a message and use a simple **if statement** to check if the B button is pressed. If it is, we set the **press** variable to **true** to break out of the **loop** and we reset the **score variable**. Once the **loop** is broken, we move on to the final **repeat**.

The final **repeat** keyword on line 73 should already be there for you - no need to type it in. This **repeat** is tied to the main **loop** on line 25.

If you find you have the wrong number of **repeat** or **endif** keywords, don’t worry! You can compare the whole program listing on the next page.

Project: 8f
“Mega Quiz”

Congratulations! You’ve completed the FUZE Project Workbook 2. Give yourself a high five! Or a pat on the back. Even just a satisfied smile will do!

By now you should be understanding the basics more clearly, feeling confident with **loops**, **variables** and **if statements** and even beginning to set up data in more complicated and efficient ways.

Learning to code is not easy - it’s all very well and good being told what an array or a structure *is* but these ideas don’t “click” into place until you’ve had some practice. There’s the magic word - Practice!

Keep going. Play with these projects and write them in different ways, change numbers, multiply instead of add - there’s no wrong way to experiment with them! You will learn something new by simply trying things out. Don’t worry if it doesn’t work, it’s a chance to learn something new!

Below (and on the following page) is the full program listing for the quiz project with correct line numbers. Be sure that yours matches up before you start changing things. Making a copy is a great idea!

```
1.  quiz = [
2.      [
3.       .question = “What is an array?” ,
4.       .answers = [ “A table of data”, “A sausage”, “A type of small dog” ],
5.       .correct = 0
6.      ],
7.      [
8.       .question = “What is a loop?” ,
9.       .answers = [ “A type of cheese”, “A repeated set of instructions”, “A dance” ],
10.      .correct = 1
11.     ]
12.]
13.
14.     function qPrint( text, x, y, size, colour, delay )
15.         textSize( size )
16.     ink( colour )
17.     printAt( x, y, text )
18.     update()
19.         sleep( delay  )
20. return void
21.
22. answer = 0
23.  score = 0
```

HACKER CHALLENGE:

1. Check all of your code and make sure it’s in perfect working order!

2. EASY! Can you add more questions to your quiz for a total of 5?

3. TRICKY! Can you add one more potential answer to your questions? You’ll have to add quite a few things for this to work!You’ll need to modify the quiz array, add another qPrint instruction, and add another if statement to the while loop which checks the player input.

```
loop
    for i = 0 to len( quiz ) loop
        clear()
        qPrint( quiz[i].question, 0, 0, 30, white, 2 )
        qPrint( “A: “ + quiz[i].answers[0], 0, 2, 30, white, 0.5 )
        qPrint( “X: “ + quiz[i].answers[1], 0, 3, 30, white, 0.5 )
        qPrint( “Y: “ + quiz[i].answers[2], 0, 4, 30, white, 0.5 )
        qPrint( “Make your choice using the Joy-Con buttons! “, 0, 6, 40, white, 1 )

        press = false
        while press == false loop
            c = controls( 0 )
            if c.a then
                press = true
                answer = 0
            endif
            if c.x then
                press = true
                answer = 1
            endif
            if c.y then
                press = true
                answer = 2
            endif
        repeat

        if answer == quiz[i].correct then
            qPrint( “Correct! “, 0, 8, 50, green, 1 )
            score += 1
        else
            qPrint( “Bad Luck... Incorrect! “, 0, 8, 50, red, 1 )
        endif
    repeat

    clear()
    qPrint( “Let’s see how you did... “, 0, 0, 40, white, 1 )
    qPrint( “You scored: “ + score + “ out of “ + len( quiz ), 0, 2, 40, white, 2 )

    press = false
    while press == false loop
        clear()
        c = controls( 0 )
        qPrint( “Press the B button to play again! “, 0, 0, 50, white, 0 )
        if c.b then
            press = true
            score = 0
        endif
    repeat
repeat
```

Glossary of Commands Used

array

Initialises an array of integer values. Default value is 0. For example:

```
array numbers[10]
```

Creates an array of ten elements where each element contains a 0.

Extra dimensions can be specified:

```
array grid[10][10]
```

Creates a 10 by 10 two dimensional array.

```
array grid[10][10][10]
```

Creates a 10 by 10 by 10 three dimensional array.

Arrays may also be defined in the following way, with specific values:

```
myArray = [1, 2, 3, 4]
```

Furthermore, an array can be defined as *empty* and expanded later:

```
myArray = [ ]
myArray[1] = 400
```

The above creates an empty array, then defines element 1 as the number 400. Undefined elements prior to the one specified will be assigned the value 0.

drawImage() / drawImageEx()

Draws an image loaded by loadImage() to the screen. Different parameters allow for various control over the drawn image:

```
drawImage( image, x, y )
drawImage( image, x, y, scale )
```

```
drawImageEx( image, {x, y}, rotation, { xScale, yScale }, tint )
```

Data types are outlined by colour above. **Black** types are handles (variables which store the image). Yellow types are floating point numbers. Green types are colour names

for / step

Initiates a special loop which repeats a specific number of times. Requires a variable definition which acts as a counter within that loop.

```
for i = 0 to 10 loop
    print( i )
    update()
    sleep( 1 )
repeat
```

The above loop will repeat a total of 10 times. The **i** variable will begin at 0 and end at 9.

The step keyword may be used to allow the counter variable to increase at a specified rate:

```
for i = 0 to 10 step 2 loop
    print( i )
    update()
    sleep( 1 )
repeat
```

The above loop will repeat a total of 5 times. The **i** variable will begin at 0 and end at 8, counting by 2 each time.

function / return / void

Allows the user to create a custom function with specified inputs and outputs. If no output is required the **void** keyword is used

```
function divideByTen( number )
return number / 10
```

The function above receives one input (number) and returns that value divided by ten.

```
function printInGreen( text )
    ink( green )
    print( text )
return void
```

The above prints specified text in green ink.

loadImage()

Loads an image from the **FUZE^4** media Library:

```
loadImage( filename )
```

FUZE

(teaching kids to code)

FUZE is developed in the UK by:

FUZE Technologies Ltd.
15 Clearfields Farm
Wotton Underwood
Aylesbury,
Buckinghamshire
HP18 0RS
England

Contact information
email: contact@fuze.co.uk
phone: +44(0) 1844 239 432

@fuzeArena