FUZE

(teaching kids to code)

Coding Projects

Book 1 - The Basics

Written by David Silvera

Loops

Variables

If Statements

Learn the fundamentals of coding and create your own simple game!

Learn to code with FUZE4

NINTENDO SWITCH

Hello! Congratulations on your awesome decision to open this book.

FUZE⁴ makes learning **real coding** as accessible as possible.

Coding can appear pretty complicated at first. It's full of strange words, mathematical concepts and problem solving. Don't let any of this put you off as, similar to most things, once you delve a little deeper it's not as scary as it first looks.

There is one thing though… **Typing**. Lots and lots of typing.

If you want to get to grips with **real coding**, **there's no getting around this.** You will be typing lines of code and you will make mistakes. You might spend a fair amount of time having to check for tiny spelling and syntax mistakes. Not to worry - debugging your code is a very important part of the learning process. Every mistake you fix makes you a better programmer!

Over the next 16 pages, you will be taken on a journey of understanding from the simplest program imaginable to comfortably using some of the more flashy programming techniques.

When learning to code, we believe you should start with 3 things: **Loops**, **Variables** and **If Statements**.

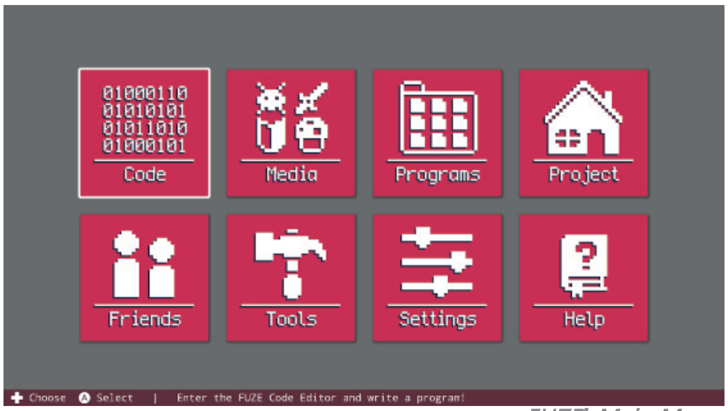
If you understand these three things even a little bit, you will begin to understand the most complicated programs out there. **ALL** programming languages rely on these fundamental concepts in some form or another, so knowing how they work is a vital step in your journey to being a coder. **Ready?**

Contents:

Use FUZE like a PRO!

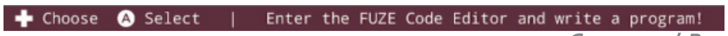
FUZE⁴ is a programming environment which puts everything you need to make your own programs and games in one easy to use and convenient place. Let's take a look at a few of the screens we'll be seeing as we go through the projects.

The first screen you see when you load **FUZE⁴** is the **Main Menu**. From here we can quickly go to any of the places we need. In this workbook we'll be mainly using the **Code** section, but take a look around and see what you can find! Feel free to change the colours of **FUZE⁴** in the **Settings** section.



FUZE⁴ Main Menu

Take a look at the bottom of the screen and you'll see the **Command Bar**. This is a very useful part of **FUZE⁴**, as it tells you all of the controls available to you.

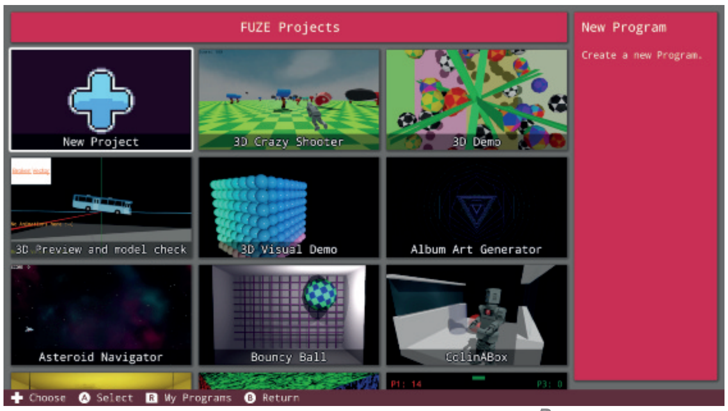


✚ Choose Ⓐ Select | Enter the FUZE Code Editor and write a program!

Command Bar

Select the **Programs** section with the **A** button.

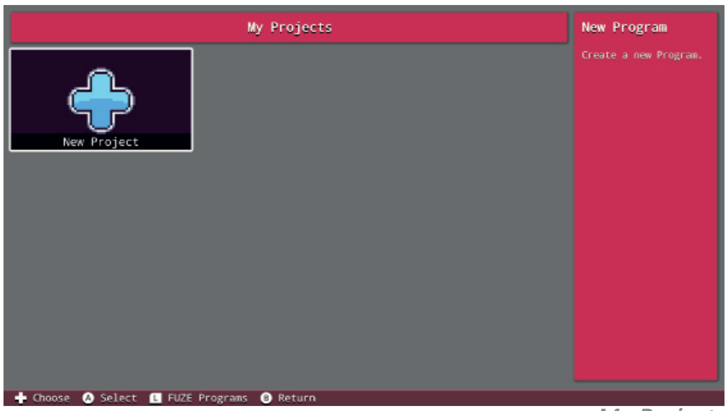
The **Programs** section is where you access all the programs in **FUZE⁴**. When you first see the screen, notice the bar at the top which says "**FUZE Projects**". These are the demo projects which are included in **FUZE⁴**. Try some of them out! There are lots of things to learn from these projects.



Programs

Pressing the **X** button on a **FUZE⁴** project will allow you to copy it into your projects to edit.

Press the **R** button to see **your** projects. Of course, right now you don't have any! Select the "**New Project**" button with **A**.

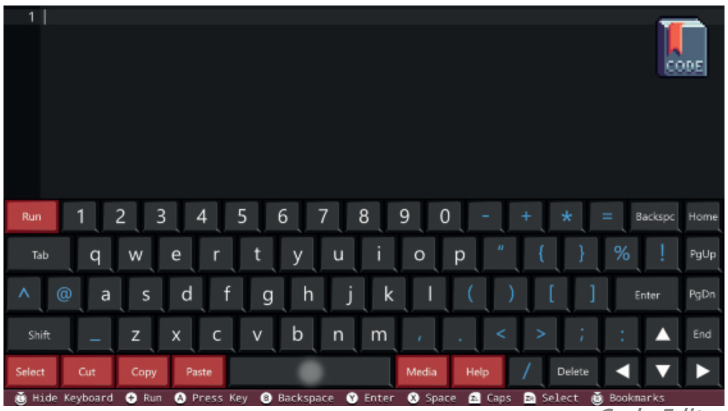


My Projects

You will be prompted to enter a project name, author and description. Name your project with your own name followed by a number. For example, my project would be called "David Silvera 01". This will make it very easy for others to find your work! Perhaps write some information about the project in the description box too. Confirm your title, author and description with the **+** button.

When you have completed the description, press the **+** button to create the project. You will be taken to the **Code Editor!**

The **Code Editor** is where the magic happens. We'll be seeing lots of this screen going forward!



Code Editor

Ready for your first computer program? I hope so! Now that you have made a project, let’s write some code! This project will teach you about what a **loop** is and why they’re important.

Loops are used all the time in programming. We use them to make a sequence of instructions happen again and again. A good example would be a video game - all video games are huge **loops** which only stop when you turn the game off!

1

```
1. print( “Hello World” )
2. update()
3. sleep( 2 )
```

Here we simply print text on the screen for 2 seconds.

We must use an **update()** function to send our text to the screen, and a **sleep()** function to wait for **2** seconds before returning to the code. **Run** the program with the **+** button.

For the next steps, **change** your code to look like the examples. You may need to delete lines of code or move them to achieve this. Keep an eye on the line numbers to make sure yours is correct.

2

```
1. loop
2.     print( “Hello World” )
3.     update()
4. repeat
```

Here we are using a **loop** to make our text print again and again. **Run** the program (+) to see our text appear across the screen. Off it goes! Press **+** again to stop the program!

All instructions between the **loop** and **repeat** keywords will repeat forever. It’s a bit like a sandwich! **loop** and **repeat** are our bread, and everything inside is the filling!

3

```
1. loop
2.     print( “Hello World” )
3.     print()
4.     update()
5. repeat
```

Let’s make a small change to make the text move **down** the screen instead of across.

By adding an empty **print()** function to our program we reset the position of the printed text to the start of the next line.

Now we have a waterfall of Hello Worlds!

4

```
1. loop
2.     print( “Hello World” )
3.     print()
4.     update()
5.     sleep( 1 )
6. repeat
```

Let’s add a small delay back into our program to really make this effect clear.

Here we have added our **sleep()** function back into the **loop**.

Think about what the computer is doing as it reads each and every line of code.

Remember: When the program reaches **repeat**, it goes back to **loop**!

Hey, look down here! This section of the page will contain challenges as we go ahead.

HACKER CHALLENGE: Can you make your program RUN Ten times faster?

"It's a beautiful World"

For the second part of this project we'll add some colour to the mix.

```
ink( fuzePink )
loop
    print( "Hello World" )
    print()
    update()
    sleep( 0.1 )
repeat
```

5

We've added a new line at the start of our program. This ink() function changes the colour of our text!

In the example we've used the colour fuzePink. FUZE4 knows the name of lots of colours, try your favourite! If FUZE4 knows a colour the name will appear green.

Alternatively, you can try a number between 0 and 115. Each colour has its own number!

Notice we have also used a 0.1 in our sleep() function to speed up the program. Now it's **ten** times faster!

```
ink( random( 116 ) )
loop
    print( "Hello World" )
    print()
    update()
    sleep( 0.1 )
repeat
```

6

In this example we have replaced our fuzePink colour with a new function: random()

This function gives us a random number out of the number we put in the brackets. Since FUZE4 knows 116 different colours, we put this number in the brackets to give a random selection out of every colour.

Notice that to change the colour, you must **run** (**+**) the program again.

This is because when the program reads the repeat line, it returns to loop. **Our ink() function is only read once!**

```
loop
    ink( random( 116 ) )
    print( "Hello World" )
    print()
    update()
    sleep( 0.1 )
repeat
```

7

Take a look at this small change, and the big difference it makes when you run the program!

By moving our ink() function inside the loop, we are now reading this line on every repetition! This means the colours will change by themselves.

This is the most important part of the project.

If you understand the difference between something being **outside** of the loop and being **inside** of the loop, then pat yourself on the back! You understand loops.

HACKER CHALLENGE: Using the textSize() function, experiment with different sizes of text. For example:

In the loop, try adding: textSize(100) to make your text much bigger.

Can you use our random() function to make the size of our text random every time?

Project: 1c

“Super Mega Hello World”

Let’s combine everything we’ve learned with some new functions to take this project to the next level.

```
loop
    ink( random( 116 ) )
    printAt( 0, 0, “Hello World” )
    update()
    sleep( 0.1 )
repeat
```

8

We’ve changed our print() function here to be printAt() instead. Notice that it looks a bit different now, we have two numbers before the text.

The printAt() function is super useful for printing text in different places! The first number in the brackets is the **column** to start printing in, and the second is the **row**.

Experiment by changing these numbers!

```
loop
    ink( random( 116 ) )
    textSize( random( 720 ) )
    printAt( 0, 0, “Hello World” )
    update()
    sleep( 0.1 )
repeat
```

9

In this example we have solved the Hacker Challenge from the previous page! By putting the textSize() function inside the loop, we can use random() in the brackets just as with we did with ink() to get random sizes!

The number in the brackets is the maximum size of your text. It is actually the height of your tallest letter in pixels! More on that later. Experiment with different values here.

Try removing the sleep() function to speed things up!

```
1. loop
2.     ink( random( 116 ) )
3.     textSize( random( 720 ) )
4.     col = random( tWidth() )
5.     row = random( tHeight() )
6.     printAt( col, row, “Hello World” )
7.     update()
8. repeat
```

1

For this example, we’ve made a couple of important changes.

We’ve added two new lines at 4 and 5. “What are those strange words, **col** and **row**?” I hear you ask.

Don’t worry! These are called **variables**, we’ll be exploring them in detail on the next page!

The “t” in the tWidth() and tHeight() functions stands for “text”. These helpful functions tell us the maximum amount of text characters we can fit on the screen.

If you’re having trouble understanding this part - imagine a piece of paper. If you write in huge letters, you won’t fit many letters on one line, but If you write in tiny letters, you can fit loads of them on one line!

This is where tWidth() and tHeight() come in handy! Depending on the size of your text, you’ll be able to fit different amounts of letters on screen. These functions tell us exactly how many.

By using random(tWidth()), we get a random text position based on the size of our text.

FUZE4

Project: 2

“How d'ya like them apples?”

Hello again! In this project we’ll be looking at another very important concept when it comes to coding. Say hello to **variables!**

1

```
1.  apples = 10
2.  print( apples  )
3.  update()
4.  sleep( 2 )
```

A **variable** is simply a piece of data which we label with a name.

Take a look at the program on the left. We begin with the statement: **apples = 10**

We are taking a value (**10**) and giving it a label.

It’s almost like putting the number **10** in a box and writing the word “apples” on it! Now that we have done this, we can use the word **apples** in our program.

On line 2 we print the value of the **apples variable**. Notice that there are no speech marks around the word apples. If we wrote **print(“apples”**) instead, we would just see the word “apples” appear on screen! We want to see the **value of the variable**, so we do not use speech marks here.

Let’s write a simple program which puts this **variable** to use. Our goal is to manipulate (change) the value of our **variable**, and print some text on the screen to show what’s happening. Feel free to change this program to say what you want to say! You don’t have to use apples. How about sweets?

2

```
1.  apples = 10
2.  ink( green  )
3.  while apples > 0 loop
4.      clear( white )
5.      print( “I have “, apples, “ apples” )
6.      print()
7.      update()
8.      sleep( 1 )
9.      print( “If I eat one then… ” )
10.         update()
11.         sleep( 1 )
12.         apples -= 1
13.     repeat
14.     print( “I have no apples left…” )
15.     update()
16.     sleep( 3 )
```

Change and add to your program so that it looks like the example on the left.

Let’s talk about the new command we are using on line 3: **while**

We can use **while** to put a condition for our **loop**. In our example, we only want the **loop** to continue if **apples** is greater than (**>**) **0**. This means when we get to zero **apples** the **loop** will stop and we will move to line 14.

What about that funny looking **-=** sign? I hear you ask. Well, that’s called a **minus equals**. We use these to reduce the value of a **variable**. To increase the value, we use **+=** (plus equals)

What a surprise! Really, **apples -= 1** is short for:

apples = apples - 1

We are redefining **apples** to be equal to itself minus one.

HACKER CHALLENGE:

1. Can you make your program eat 2 apples at once?
2. Can you change your prgram so that you begin with zero apples and gain them instead? Your program should stop when you reach ten apples.

"Let's Get Quizzical"

Who doesn't like a good Quiz? In this project we'll be making our very own quiz game which you can improve on! Test your friends and parents and feel free to make it as silly as possible.

Before we get started, there's a new concept to learn. They're called **if statements.**

```
apples = 10
if apples > 0 then
    print( "Hurray!" )
endif
update()
sleep( 2 )
```

Don't worry, **if statements** are the easiest concept to learn. Why? Because people already use them all the time!

Ever heard something like *"If you eat all your vegetables, then you can have extra dessert"*? Well, that's an **if statement**!

An **if statement** is a **condition**, and something that happens **if** that condition is **true**.

In programming, **if statements** are used to do all sorts of things. Imagine a video game controller. Each button might have a different outcome, and each one has its own **if statement!**

For our quiz, we'll use an **if statement** to do something if a player gets the answer to a question correct, and to do something different if they answer incorrectly.

Below is a small introduction to the quiz and a single question. Feel free to copy it exactly, but we encourage you to make up your own question!

1

```
1.  score = 0
2.  ink( fuzeBlue )
3.  textSize( 40 )
4.  print( "Hello and welcome to my quiz. \n" )
5.  print( "Please answer in lower case only. \n" )
6.  update()
7.  sleep( 3 )
8.  clear()
9.  print( "Q1. What is my name? \n" )
10. update()
11. sleep( 3 )
12. answer = input( "What is my name?" )
13. if answer == "david" then
14.     print( "Correct! Well done! \n" )
15.     score += 1
16. else
17.     print( "Incorrect... Better luck next time! \n" )
18. endif
19. update()
20. sleep( 3 )
```

We begin with a **score variable**, since we need to keep track of how well the player is doing.

Notice the strange looking **"\n"** at the end of the print lines? This makes **FUZE^4** start a new line to print on.

We are using the **input()** **function** to allow the player to enter some text. We store their answer as a **variable** called **answer**.

Our **if statement** is on line 13. It checks whether their answer is equal to the correct one.

Don't be scared by the **double equals** (**==**). We use this when comparing two things in programming.

If it is, we print "Correct!" and increase the **score variable** by **1**.

We use the **else** keyword to give an instruction for when the player answers incorrectly. If their answer is anything other than **"david"** they will get it wrong.

```
9.  print( "Q1. What is my name? \n" )
10.update()
11.sleep( 2 )
12. answer = input( "What is my name?" )
13. if answer == "david" or answer == "dave" then
14.     ink( green )
15.     print( "Correct! Well done! \n" )
16.     score += 1
17. else
18.     ink( red )
19.     print( "Incorrect... Better luck next time! \n" )
20. endif
21. update()
22. sleep( 2 )
23. // MORE QUESTIONS GO HERE
24. clear()
25. ink( fuzeBlue )
26. print( "Let's see how you did! \n" )
27. update()
28. sleep( 0.5 )
29. print( "You scored: ", score, " out of 1 \n" )
30. update()
31. sleep( 0.5 )
32. if score == 0 then
33.     print( "You need to do your homework! \n" )
34. endif
35. if score == 1 then
36.     print( "Wow! Full Marks! \n" )
37. endif
38. update()
39. sleep( 3 )
```

2 In this example we've added a few improvements to our quiz.

First, take a look at line 12. We now have **two** possible correct answers.

By using the **or** keyword, we can check **two** different things in a single **if statement**.

This is a very useful technique! There is no limit to how many conditions you can check in a single **if statement**.

Take a look at lines 14 and 18. We have added a change in ink colour depending on their answer.

Line 23 is simply a guide for you to add more questions! It has no effect on the program. If you add // at the start of a line, it will be ignored! These are called **comments**.

From line 24 onward is a new section. Every good quiz needs to tell the player how they did!

The important part here is line 29. Look carefully at the print() line.

When using print, you can put commas between things you'd like to print and **FUZE4** will print them all together. When we print **score,** we are printing the **value of the score variable**.

Lastly, from line 32 we have two **if statements**. These ones give the player a different sentence depending on their score!

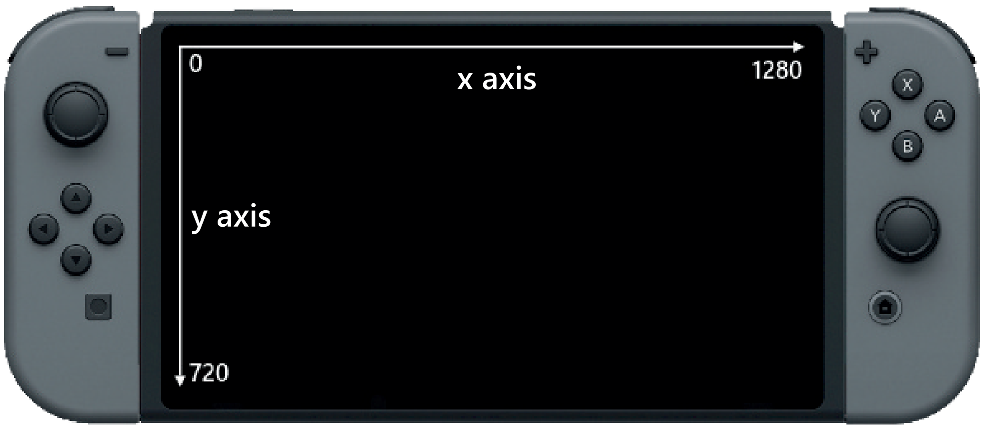
Of course, you'll need more of these **if statements** when your quiz is complete!

HACKER CHALLENGE:
Add questions to your quiz until you have at least 3. These extra questions need to go between lines 22 and 24.
From 24 onwards is our ending screen. Make sure you use update() and sleep() to give a nice delay between prints in your quiz. You should also add a different sentence for each possible score.

By now you should be comfortable with three concepts: loops, **variables** and if statements.

With just these three techniques, you can achieve truly great things! Let's combine everything we know into a more visual project. Our aim is to put a circle on the screen, make it move around and bounce off the edges of the screen. First, we must cover a few important facts about the screen.

The screen is made up of something called **pixels.** A pixel is tiny little light and there are usually lots of them. A **Nintendo Switch** screen contains **921600** pixels! These are laid out in a huge grid **1280** pixels across and **720** pixels down. We call these two measurements the **x axis** and the **y axis**.



When we see something on a screen move **horizontally** (left or right), it is moving along the **x axis**. When we see something moving **vertically** (up or down) it is moving along the **y axis**.

When we want to put something on screen, we must give it a position in **co-ordinates**. Take a look at the example below which puts a circle right in the middle of the screen.

1

```
circle( 640, 360, 100, 32, white, false )
update()
sleep( 3 )
```

Say hello to a new function! The numbers in the brackets all have a particular meaning:

circle(xPos, yPos, radius, sides, colour, outline)

How do we know this circle will appear in the middle of the screen? Well, the number we are using for the **x** position of the circle is 640. This is exactly half of 1280! The same goes for the **y axis** position, 360 is half of 720. If we change one of these numbers, the position of the circle will change.

If we wanted the circle's position to change *during the program*, we will need to use **variables**.

2

```
x = 640
y = 360
loop
    clear()
    circle( x, y, 100, 32, white, false )
    update()
repeat
```

Change your program so it looks like the example on the left.

We are now using a loop, so our program will keep running until we press the **+** button again.

Now that we are using **variables** for our circle's location, we can *change* the value of those **variables** and we will see the circle move!

Make sure you program works just fine, then turn the page and let's get this ball bouncing!

First order of business is to make the ball move around. To achieve this, we'll need to change the value of our position variables **during the program**.

3

```
1. x = 640
2. y = 360
3. loop
4.     clear()
5.     x += 1
6.     circle( x, y, 100, 32, white, false )
7.     update()
8. repeat
```

Our new line is on line 5. Remember **+=** from the last project?

All we are doing is simply adding 1 to the value of our **x variable** every single time the **loop** goes around.

Run the program with **+** to see what happens!

Your circle should move gently along the **x axis** to the right.

We could use **-=** to make our circle travel left, or we could simply change the **1** to **-1**.

To make the ball **bounce** off of the right side of the screen, we will need to change the direction from positive to negative. To bounce off of the left side, we must change the direction from negative to positive. To achieve this, we'll need our speed to be a **variable**.

4

```
1. x = 640
2. y = 360
3. xSpeed = 1
4.
5. loop
6.     clear()
7.     x += xSpeed
8.     circle( x, y, 100, 32, white, false )
9.     update()
10. repeat
```

Our new **variable** is on line 3.

We are using this **variable** as the number of pixels the ball moves along the x axis each time the loop repeats, so we have called it **xSpeed**.

Of course, it could be be called something totally different, like **giraffes,** for example. However, this wouldn't reallly help us understand what's going on and would be very confusing to anyone else!

Make sure to replace the **1** on the line **x += 1** with the **xSpeed variable**, just like in the example. If you don't, our variable isn't doing anything at all!

We're almost there! We have everything we need to bounce the ball along the **x axis**.

On the next page, we'll be adding the first set of **if statements** to make the ball bounce off of the left and right sides.

Remember: when we **add** a **negative** number, it is *exactly the same as simply subtracting*. This concept is very important as we move forward!

HACKER CHALLENGE:
Can you create more variables to store the radius and number of sides for our circle? They should be appropriately named.

To make the ball bounce off the right hand side of the screen, we'll need an **if statement** which checks the position of the ball along the x axis.

5

```
1.  x = 640
2.  y = 360
3.  xSpeed = 4
4.
5.  loop
6.      clear()
7.      x += xSpeed
8.      if x > 1280 then
9.          xSpeed = -xSpeed
10.     endif
11.         circle( x, y, 100, 32, white,
false )
12.     update()
```

The **if statement** begins on line 8 and ends on line 10.

Make the changes to your program and run it. You should see the ball bounce off the right hand side of the screen, then it will vanish off the left side!

Line 9 might look a bit confusing at first, but it's more simple than you think!

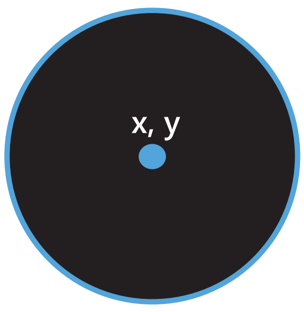
We simply take the **xSpeed** variable and make it negative. if **xSpeed** was 4, it becomes -4. If it was 20, it becomes -20.

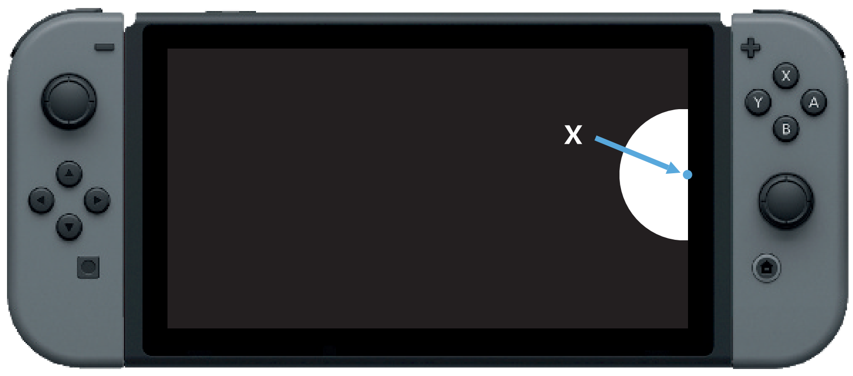
However… We have a small problem. Did you notice that the ball didn't really bounce correctly?

Take another look and you'll see that the ball bounces when *the middle touches the edge of the screen*. There is a good reason for this, and we must understand it fully! Take a look below.

When we draw a circle on screen in **FUZE**^{4}, the **x** and **y** co-ordinates describe the *centre of the circle*.

If we only check the position of the **x** variable in our **if statement**, the ball will bounce when the middle touches the edge of the screen.



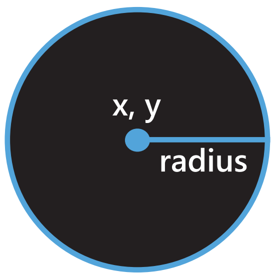


To make the ball bounce correctly, we need to understand a measurement in a circle called the *radius.*

In our line of code:

```
11.     circle( x, y, 100, 32, white, false )
```

The 100 is the *radius* of the circle. With a small adjustment to our code, we can make the ball bounce perfectly on the edge, no matter what the size! We simply need to add the radius to our **x** variable. Turn the page and let's get this fully understood!



HACKER CHALLENGE:
Take a break. You've earned it! :-)

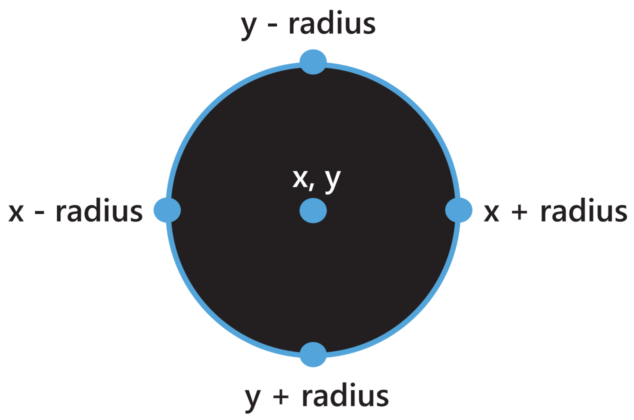
We need to include the radius as a variable in our program. Take a look below at the changes and edit your code. Be sure to read the information on the right!

6

```
1.  x = 640
2.  y = 360
3.  xSpeed = 4
4.  radius = 100
5.
6.  loop
7.      clear()
8.      x += xSpeed
9.      if x + radius > 1280 then
10.         xSpeed = -xSpeed
11.     endif
12.         circle( x, y, radius, 32, white, false )
13.     update()
```

On line 4 we've defined a **radius variable**.

Take a look at the **if statement** on line 9. We've changed it slightly to include the **radius variable**. To understand why it is **x + radius**, take a look at this diagram:



We can use either **x** or **y** and **+** or **-** **radius** to give us each edge point of the circle.

Don't forget to change the **size** part of the circle() function to radius on line 12.

Now we have a new problem. The circle bounces off the right hand side of the screen, but vanishes straight off of the left side! Remember the **or** keyword from the quiz project? This will be very useful here!

7

```
1.  x = 640
2.  y = 360
3.  xSpeed = 4
4.  radius = 100
5.
6.  loop
7.      clear()
8.      x += xSpeed
9.      if x + radius > 1280 or x - radius < 0 then
10.         xSpeed = -xSpeed
11.     endif
12.         circle( x, y, radius, 32, white, false )
13.     update()
14. repeat
```

Because of the way negative numbers work, we can simply add another condition to our **if statement**.

By adding "**or x - radius < 0**" we are also checking if the left side of the circle has touched the left side of the screen.

We can keep the **xSpeed = -xSpeed** line exactly the same, because if we make a negative number *negative again,* we get a *positive*!

This is quite a tricky thing to get your head around, but stick with it and it will soon feel natural.

Adjust the speed and watch the circle bounce!

HACKER CHALLENGE:
Test yourself and see if you can make the ball bounce and move on the y axis too! You'll need two more variables.

```
1.  x = 640
2.  y = 360
3.  xSpeed = 4
4.  ySpeed = 4
5.  radius = 100
6.
7.  loop
8.      clear()
9.      x += xSpeed
10.     y += ySpeed
11.         if x + radius > 1280 or x - radius < 0
then
12.         xSpeed = -xSpeed
13.     endif
14.     if y + radius > 720 or y - radius < 0 then
15.         ySpeed = -ySpeed
16.     endif
17.         circle( x, y, radius, 32, white, false )
18.     update()
```

8 All we need to do to complete the program is to add the same **variables** and **if statement** for the y axis.

The exact same rules apply, except the if statement must check if "**y + radius > 720**", because there are **720** pixels on the y axis!

Now we've got this set up correctly, you could freely change the radius and speed **variables** and the program will still work correctly - although, at very large numbers it might look a bit strange!

For a cool effect, try changing the **false** in the **circle()** function to **true**, and then remove the **clear()** instruction. You should see something like a tube of empty circles drawing all over the place!

With this, we've got the basic mechanics ready for a simple game. Over the next few pages, we'll look at how to turn this into a real single-player game we can customise!

Before we move on, we should change one last thing in our code. When the Nintendo Switch console is in the dock, running on a monitor or TV set, the resolution of the screen changes.

In TV mode, the x axis is **1920** pixels and the y axis is **1080**. Because of this, when we play our game on the TV, it won't quite work properly. We can fix this very easily by using some helpful functions, **gWidth()** and **gHeight()**.

```
1.  x = gWidth() / 2
2.  y = gHeight() / 2
```

9 By using **gWidth() / 2**, we get the centre of the x axis, and by using **gHeight() / 2** we get the centre of the y axis. This will give us the middle of the screen no matter how big or small our screen is!

```
11.         if x + radius > gWidth() or x - radius < 0 then
...
14.         if y + radius > gHeight() or y - radius < 0 then
```

1 We must also replace the **1280** and **720** in our **if statements** with **gWidth()** and **gHeight()** if we want our bouncing to be correct.

Project: 5a
“Tennis For One”

If we’re going to turn this concept into a game, we’ll need a player. We are going to need quite a few new things, so it is a good idea to start a new project for a fresh start. Now for lots of **variables**! Yay!

1

```
1.  gw = gWidth()
2.  gh = gHeight()
3.
4.  score = 0
5.  goal = gw / 30
6.
7.  batHeight = 200
8.  batWidth = 30
9.  batX = goal - batWidth
10.     batY = gh / 2 - batHeight / 2
11.
12.     ballX = gw / 2
13.     ballY = gh / 2
14.     ballRadius = 20
15.     ballXSpeed = -10
```

We’ll need the screen *width* and *height* many times in our program, so to avoid typing **gWidth()** and **gHeight()** every time, we’ll make them into nice short variables - **gw** and **gh**.

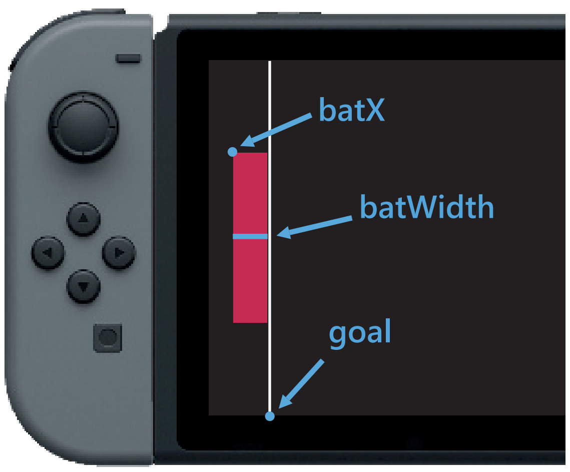
Next we will need a **score** variable to keep track of our points.

On line 5 we have a strange looking **variable**. This will be where our goal line is on screen. If the ball passes this point, we lose. It will be very helpful to have this as a variable so we can use it later in the program.

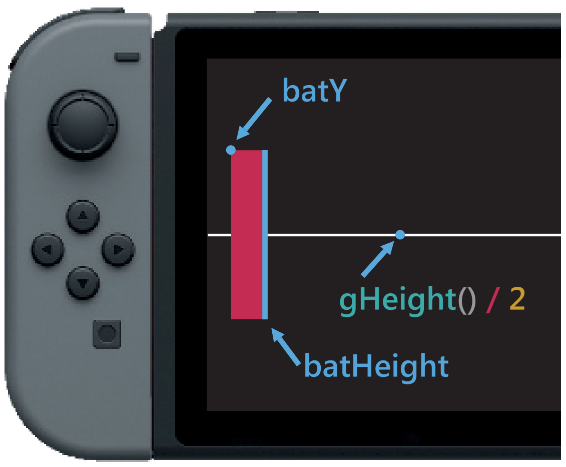
From line 7 we have the **variables** to store our player information. The height and width of the bat in pixels is very useful, followed by the position of the bat on the x and y axis. These **must** be **variables** if we want our bat to move!

Finally, we have the **variables** for the ball. These need no explanation as they are the same from the previous project. We have named them more descriptively, however.

To understand the bat **variables**, we must know that the x and y positions of a rectangle describe the top left point. When placing our bat on the screen, this is very important:



batX = goal - batWidth



batY = gHeight() / 2 - batHeight

On the next page, we will build the main game **loop** and display our bat and ball, using the **variables** we have just set up here. See you there!

Now that we understand exactly why the **variables** are what they are, let's rebuild our **loop**.

2

```
18. loop
19.         clear( green )
20.
21.         ballX += ballXSpeed
22.     ballY += ballYSpeed
23.
24.     if ballX + ballRadius > gw then
25.         ballXSpeed = -ballXSpeed
26.     endif
27.
28.     if ballY + ballRadius > gh or ballY - ballRadius < 0 then
29.         ballYSpeed = -ballYSpeed
30.     endif
31.
32.     box( batX, batY, batWidth, batHeight, red, false )
33.
34.     circle( ballX, ballY, ballRadius, 32, yellow, false )
35.
36.     box( goal, 0, 5, gh, white, false )
37.
38.     update()
39. repeat
```

On the left we have our rebuilt main game loop, with a couple of differences.

Firstly, the ball no longer needs to bounce off of the left side - this is the goal we are trying to protect now!

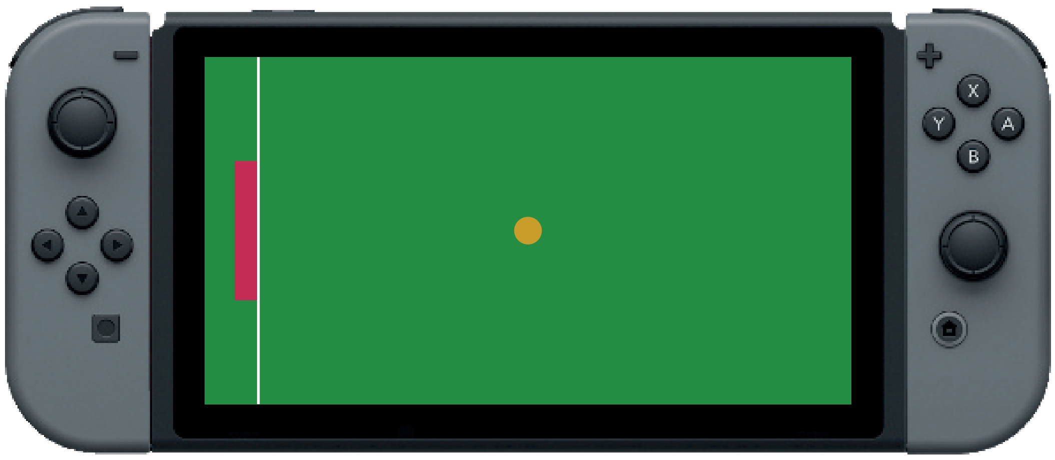
On line 32 we are drawing the player. We use a **box()** function here. As you can see, the arguments are **x position**, **y position**, **width**, **height**, **colour** and **fill**.

We draw the ball on line 34 using the **circle()** function, just like in the previous project.

Lastly, on line 36 we draw the goal line. The x position is our **goal variable**. The y position is **0**, for the top of the screen. The width is **5** pixels, giving us a nice line. Finally, the height is our screen height **gh variable**, since we want the line to cover the whole screen height.

Run your program and make sure it all looks good!

Your program should look something like this (*don't worry if your bat and ball aren't quite the size here*):



HACKER CHALLENGE:
Feel free to change the colours of everything! Make your project look unique, no need to copy us!

Time to take control! This part of the project will teach you how to access the Joy-Con Controllers.

3

```
18.  loop
19.  clear( green )
20.
21.  c = controls( 0 )
22.  batY -= c.ly * 12
23.
24.  ballX += ballXSpeed
```

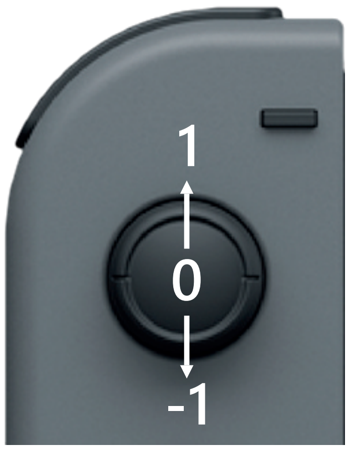
We have added only two new lines *before* the "**ballX += ballXSpeed**" line.

In **FUZE^4**, we access the controls with the **controls()** function. In this project we assign it to a **variable** called **c**.

Once assigned, the **c variable** now contains the state of all of the controls in the Joy-Con controllers! We can access them by using a dot "." followed by the name of the button or stick we want to read. But what about the **(0)**?

1 - controls(0)　2 - controls(1)　3 - controls(2)　4 - controls(3)

Each pair of Joy-Cons linked to the console is accessed by a different number in the **controls()** function



Take a look at line 22 - "**batY -= c.ly * 12**" What on earth does that mean?

In our game, we want to control the player using the *left control stick*. Since we only want to move up and down, we need to know the *y axis position* of the stick. That's why we use **c.ly**! The **l** stands for **left**, and the **y** stands for **y axis**.

The diagram on the left shows the values we get from **c.ly**. If the stick is not being pushed, **c.ly** is **0**. As we move the stick up or down, this **0** gets closer to **1** (pushed up) or **-1** (pushed down).

So, line 22 really says: subtract the value of **c.ly** from the bat y position. We also multiply **c.ly** by **12** to give us higher movement speed.

Run the program and move the left stick up and down to move the bat. Hurray! We have controls. However... Have you noticed that we can move the bat off-screen? Keep holding up or down to see.

4

```
18.  loop
19.  clear( green )
20.
21.  c = controls( 0 )
22.  batY -= c.ly * 12
23.  batY = clamp( batY, 0, gh - batHeight )
24.
25.  ballX += ballXSpeed
```

To fix this, we must *restrict* the **y position** of our player. We can use a really helpful function called **clamp()** to do this.

The **clamp()** function forces a number into a particular range. The first argument is the value we want to restrict.

The second argument (**0**) is the *minimum* possible value. The last argument is the *maximum*. Using **gh - batHeight** means the bat cannot move past the bottom of the screen.

In this section, we'll add the ability to hit the ball and score points. Without this, it's not a very fun game!

```
32.  if ballY + ballRadius > gh or ballY - ballRadius < 0 then
33.      ballYSpeed = -ballYSpeed
34.  endif
35.
36.  if ballX - ballRadius < goal then
37.      if ballY > batY and ballY < batY + batHeight then
38.          ballXSpeed = -ballXSpeed
39.          score += 1
40.          ballXSpeed *= 1.2
41.          ballYSpeed *= 1.2
42.      else
43.          score = 0
44.          ballXSpeed = -10
45.          ballYSpeed = 10
46.          ballX = gw / 2
47.          ballY = gh / 2
48.      endif
49.  endif
50.
51.  box( batX, batY, batWidth, batHeight, red, false )
```

5

Here it is! Note that this **if statement** has been added at line 36, and ends at line 49. The previous **if statements** and **box**() command after have been included for clarity.

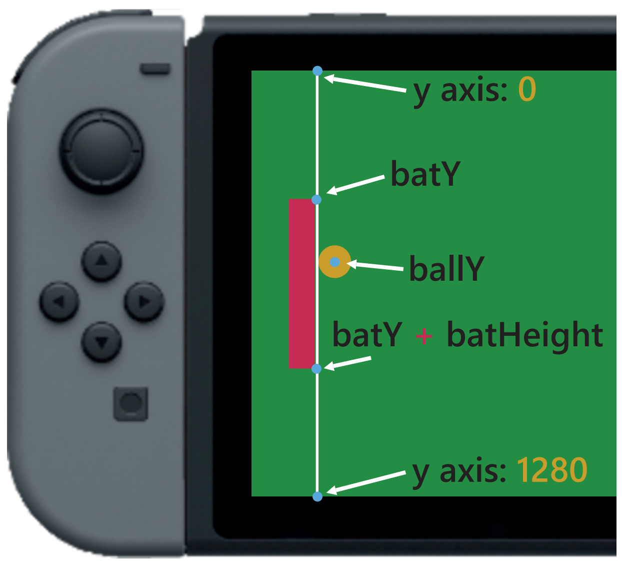
First, we check if the left edge of the ball has moved past the goal (line 36).

Then, inside that **if statement**, we have another **if statement** to check if the player is in the right place to bounce the ball back.

See the diagram below for a visual explanation of this condition.

If the bat is in the right place, we reverse the x axis direction, add a point to our **score variable**, and increase the speed of the ball!

Using an **else**, we can provide an alternative set of instructions which happens if we miss. We reset the score, speed and position of the ball.



To properly understand the **if statement** on line **37**:

```
if ballY > batY and ballY < batY + batHeight then
```

Take a look at the diagram on the left. All of the key points are highlighted with a **blue circle**, with an arrow labelling each.

The ball's y axis position must be **greater than** (further *down* the screen) the top corner of the bat (**batY**), and *also* be **less than** (further *up* the screen) than the bottom corner of the bat (**batY + batHeight**).

The key here is to remember that the y axis *begins* at **0** in the top left corner of the screen, and *increases* as we move down the screen toward the bottom.

HACKER CHALLENGE:
Can you make the ball speed increase even faster when we hit it successfully?

We’re almost there! Let’s display the score and add some polish to the program in this final section.

```
55.     box( goal, 0, 5, gh, white, false )
56.
57.     box( 0, 0, gw, 10, grey, false )
58.     box( 0, gh - 10, gw, 10, grey, false )
59.     box( gw - 10, 0, 10, gh, grey, false )
60.
61.     textSize( 35 )
62.     ink( white )
63.     printAt( 30, 1, “Score: “, score )
64.
65.     update()
66.  repeat
```

6

The newly added code on the left begins at line 57 and ends at line 63. The previous box() line and the end of the main loop has been included for clarity.

The three box() functions simply draw thin rectangles around around the screen, leaving the left side blank. This is only cosmetic - it looks nice, but it doesn’t change how our program works.

From line 61, we have our score display. We define a size for our text followed by the ink colour, then simply print “Score: “ followed by the score **variable**. Easy as that!

Congratulations! You now have a fully working game in your **FUZE^4** projects - made by you! Now comes the most fun part - see how high you can score!

If you feel confused about any of the parts of your code, take another read of the particular page which explains the section. Making a game is not easy, and this is just the beginning!

Most professionally made video games are tens or hundreds of thousands (sometimes even *millions*!) of lines of code. Of course, they usually also require huge teams and years of hard work!

Practise, apply yourself, experiment and learn. Before you know it you’ll be a coding wizard!

HACKER CHALLENGE:
Time for the ultimate Hacker Challenge!

Now that your program is complete, it’s time to get creative. You should change anything about the program to be the way you want.

Here are some ideas:

- Change the background colour using the clear() command
- Change the colour of the player
- Change the colour of the ball
- Change the size of the player
- Change the size of the ball
- Change the movement speed of the player
- TRICKY: Could you change the rate of increase in the ball speed when we score?
- TRICKY: Could you increase the size of the ball when we score?
- TRICKY: Could you increase the size of the player when we score?
- TRICKY: Could you change the size of the player when we miss?
- TRICKY: Could you change the size of the ball when we miss?

Glossary of Commands Used

box()
Prepares a box to be drawn with update(). The parameters for the box function are:

```
box( xPos, yPos, width, height, colour, outline )
```

circle()
Prepares a circle to be drawn with update(). The parameters for the circle function are:

```
circle( xPos, yPos, radius, sides, colour, outline )
```

clamp()
Restricts a supplied value to a specific range

```
output  = clamp( input, min, max )
```

clear()
This function clears the screen. By default, this will clear with a black colour. However, any colour can be put in the clear() brackets to change the background colour for the whole screen. This can be a colour name or an RGBA vector. For example:

```
clear( fuzePink )
clear( { 1, 0, 1, 1 } )
```

controls()
Reads the state of the specified pair (0, 1, 2 or 3) of **Joy-Con Controllers**. Should be assigned to a variable.

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

The user can now access any of the controller buttons/control sticks with .name:

Reading the "**X**" button: **c.x**
Reading the right control stick y axis: **c.ry**

All button values are either 0 or 1.

Control stick values are between -1 and 1.

gWidth() / gHeight()
Returns the screen width/height in pixels

```
screenWidth  = gWidth()
```

if / then / else / endif
Creates a conditional statement which allows subsequent lines to be executed provided the condition is met. else allows for an alternative condition should the original condition **not** be met.

Conditional statements are closed with endif.

ink()
Changes the colour of printed text. You can put a colour name, an integer number between 0 and 115 (116 colours), or an RGBA (red, blue green, alpha) vector describing any colour. For example:

```
ink( bisque )
ink( 42 )
ink( { 1, 0.2, 0.8, 1 } )
```

input()
Allows user to input text using the keyboard. Entered text is stored in a variable. Prompt text may also be entered. For example:

```
savedText = input( "Enter prompt text here" )
```

loop / while / repeat
Initiates a series of instructions to be executed repeatedly. A loop begins with the **loop** command and ends with the **repeat** command.
The **while** command provides a condition to a loop. For example:

```
while condition == true loop
```

print()
Prepares text to be be drawn with the update() function. Can be used to print strings (information in speech marks), numeric values, arrays and structures (like controls(0)).

printAt()
Similar to print(), except it allows the user to print text at a specific text co-ordinate. Text co-ordinates change with text size.

```
printAt( 3, 4, "Hello!" )
```

sleep()
Instructs **FUZE⁴** to do nothing for a given number of seconds. To sleep for 1 second, for example:

```
sleep( 1 )
```

textSize()
Sets the size of displayed text. The number supplied is the maximum height of letters in pixels.

```
textSize( 100 )
```

update()
Sends the framebuffer memory to be drawn to the screen. This is a necessary function to display anything on the screen.

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