"I thought programming would have been really hard, but this wasn’t."

(Darren, 14 year old high school student)
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Foreword</td>
<td>3</td>
</tr>
<tr>
<td>Session 1 – Creating your first application</td>
<td>4</td>
</tr>
<tr>
<td>When finished, students will be able to:</td>
<td>4</td>
</tr>
<tr>
<td>When finished, students should have a basic understanding of:</td>
<td>4</td>
</tr>
<tr>
<td>Introduction</td>
<td>4</td>
</tr>
<tr>
<td>Activity 1a – Creating a new script</td>
<td>5</td>
</tr>
<tr>
<td>Activity 1b – Creating a game script</td>
<td>6</td>
</tr>
<tr>
<td>Session 2 – Creating your first game</td>
<td>12</td>
</tr>
<tr>
<td>When finished, students will be able to:</td>
<td>12</td>
</tr>
<tr>
<td>When finished, students should have a basic understanding of:</td>
<td>12</td>
</tr>
<tr>
<td>Cloning and publishing scripts</td>
<td>12</td>
</tr>
<tr>
<td>Random numbers</td>
<td>12</td>
</tr>
<tr>
<td>Conditions</td>
<td>13</td>
</tr>
<tr>
<td>Logical operators in TouchDevelop</td>
<td>14</td>
</tr>
<tr>
<td>Arithmetic operators in TouchDevelop</td>
<td>14</td>
</tr>
<tr>
<td>Actions</td>
<td>14</td>
</tr>
<tr>
<td>Collisions</td>
<td>15</td>
</tr>
<tr>
<td>Activity 2a – Creating a Fruit Ninja clone</td>
<td>16</td>
</tr>
<tr>
<td>Activity 2b – Creating a Pong clone</td>
<td>27</td>
</tr>
<tr>
<td>Challenges</td>
<td>34</td>
</tr>
<tr>
<td>Session 3 – Creating games with multiple sprites of the same type</td>
<td>35</td>
</tr>
<tr>
<td>When finished, students will be able to:</td>
<td>35</td>
</tr>
<tr>
<td>When finished, students should have a basic understanding of:</td>
<td>35</td>
</tr>
<tr>
<td>Collections</td>
<td>35</td>
</tr>
<tr>
<td>FOR Loops</td>
<td>36</td>
</tr>
<tr>
<td>FOR EACH Loops</td>
<td>36</td>
</tr>
<tr>
<td>WHILE Loops</td>
<td>37</td>
</tr>
<tr>
<td>Activity 3 – Creating a Breakout clone</td>
<td>37</td>
</tr>
<tr>
<td>Challenges</td>
<td>48</td>
</tr>
<tr>
<td>Session 4 – Creating games using the accelerometer</td>
<td>49</td>
</tr>
<tr>
<td>When finished, students will be able to:</td>
<td>49</td>
</tr>
</tbody>
</table>
When finished, students should have a basic understanding of:

- Accelerometer ................................. 49
- 2D Animation .................................. 50
- Activity 4a – Creating a basic accelerometer game ........................................ 52
- Challenges ...................................... 53
- Activity 4b – Creating a scrolling shooter game .............................................. 54
- Challenges ...................................... 65
- Activity 4c – Creating a Space Invaders clone .................................................. 66
- Challenges ...................................... 79
TouchDevelop Curriculum

Foreword

TouchDevelop is an amazing new platform from Microsoft which allows anyone to create applications for mobile devices on mobile devices. When I was a child I taught myself programming in BASIC on an 8-bit computer and the games I created were developed on the same computer that I then played them on. This is a return to that concept as TouchDevelop is a free web/cloud based platform, which runs on any web browser with full HTML5 support. So anyone can create/edit applications on a pc or mobile device and then test that application on the same device. To access TouchDevelop just point your HTML5 web browser to www.touchdevelop.com/app and log in via a Microsoft live, Facebook or Google account.

TouchDevelop can be used to create any type of application, however in this curriculum we will be focusing mainly on creating gaming applications. The purpose of this curriculum will be to introduce basic programming concepts, such as variables, conditions, loops, functions, while students learn to create fun games. The final applications they create can even be published on the Windows Store.

The TouchDevelop interface is designed to be usable with a touch screen tablet or mobile phone, but also with a standard keyboard and mouse on a personal computer or laptop. So in this curriculum I will use the term “select” when I want the student to click on or touch a button in the TouchDevelop interface.

You will find video tutorials for each of the activities in this curriculum on my youtube channel, follow the URL below to my TouchDevelop playlist.


You can also download the latest version of my curriculum from the TouchDevelop website by going to https://www.touchdevelop.com/courses
Session 1 – Creating your first application

When finished, students will be able to:

- Draw text on the wall
- Create a script
- Setup a game board
- Set background colours and art
- Create ellipse and rectangular sprites
- Create sprites from graphic images
- Set game physics
- Set the speed and rotation of a sprite

When finished, students should have a basic understanding of:

- Local and global variables
- Variable types
- Assigning values to variables
- Game physics

Introduction

A script in TouchDevelop is basically your application; it contains the code, the art and anything else which makes up the application. The wall is your screen; you can post things to it so that when you run your script you can see them. In game applications you will create a game board, which is posted to the wall and has a specific size and orientation, for the majority of things we will do in this curriculum it will be the board we are working on and we will display it on the wall.

In TouchDevelop functions are called actions and come under the code grouping. Actions are bits of code which perform a specific task, which are usually things that you will want to do in your application more than once. Putting these tasks into actions, means we can call that action whenever we want to do that task, rather than having to repeat the code again and again.

We are going to be encountering variables, in programming a variable is a storage location which holds some information or a value. We give it a name so that we can reference it. For instance a popular variable in computer games is often named SCORE, which (as you might guess) stores the score. Game scores are nearly always numbers, so we want a variable which is a number; this is called the “variable type”. When a variable is created, we need to assign it a variable type, which sets the type of data it can hold. Other popular variable types are Strings (for holding text) and Booleans for holding true or false values. A lot of games have a Boolean variable called GAME OVER, which can be true or false. The other variable type we will use a lot in games is SPRITE, which can be defined as a graphical image which can move.

Variables can also be local or global. A local variable means that the variable can only be accessed or referenced from within the action that it has been creating (declared) within. It is preferably whenever possible to use a local variable, however many variables must be global which means that you can access those variables from anywhere in your game or application. Things like score and lives would generally be set as global.
When working with variables we will often assign values to variables and sometimes we need to check the value contained within a variable.

To assign a value to a variable the symbol we use is $\ := \$\$.

To check for a numeric value contained with a variable the symbol is $\ = \$$.

To check for a specific value contained within a string you use the equals command.

For instance $\text{score} \ := \ 10$ assigns the score variable a value of 10.

While $\text{score} \ = \ 10$ means is score equal to 10, which results in a TRUE or FALSE value.

When you create your first script, you should notice that an action script is created by default called MAIN. Most programming languages have something like this, which is a main function which executes first when your application runs; this is the case in TouchDevelop.

Activity 1a – Creating a new script

- Go to [www.touchdevelop.com](http://www.touchdevelop.com)
- Login using your Microsoft Live account, Google account or Facebook account.
- From the HUB which appears, select “CREATE SCRIPT” and select blank.
- Enter a name for the script in the box which appears, I suggest you call it “Activity1a”.
- Select the create button.
- You should now see the main action script, select the first line in the main action where it says “do nothing”.
- Select the string button and type in “Hello World!” and select the “Post to Wall” button.
- Try running the script by selecting the RUN button; the wall should be displayed with the message “Hello world!” on it.
- Select the back button to return to editing the script.
- Select the existing line of code and select the part which says “Hello world!”.
- A box should appear allowing you to edit the text. Change it to say something else and run the script again to test it works.
- Select the back button twice and you will be back at the hub. TouchDevelop automatically syncs your script and you should see a message in the top left hand corner saying syncing done. Syncing basically means your script has been saved to the cloud. If you log into TouchDevelop on another device you should find that the script you just created is there as well.
Activity 1b – Creating a game script

We are going to create a script for making a game this time, but we will do it step by step, so we won’t use the “Physics Game Starter” button which creates a partially created game.

- Select “Create Script” & select blank. Give the script a name like “My first game”.
- Now let’s set a background colour on the wall. If you are using a keyboard you can type this code in, if you are using a touch device then use the buttons at the bottom right of the screen. The line of code we want to enter is shown below, which will set the background to red. If you are typing on a keyboard you should notice that as you start to type each command a list of matches will appear on the left. When you see the command you are looking for, you can use the UP and DOWN cursor keys to move onto it and then press ENTER to select it. Each part that comes before an arrow must be confirmed by pressing enter or by selecting it from the buttons at the bottom, before starting on the next part. The “set background” command will default at first to colors -> random which, as you might have guessed, sets a different colour on the background each time you run the script. Change this to red so that your background is always set to red.

  wall → set background(colors → red)

- Run the script to make use it works (the screen should go red).
- Next we want to add a game board to our script. A game board is a 2D game engine that comes with TouchDevelop. This will be our first use of variables, the variable we need is of type BOARD. First select the ADD (+) icon to create a new line underneath the last one.
- Select or type the keyword VAR (which stands for variable). After selecting VAR the line of code should read var x :=
- Press Enter or touch to the right of the equals symbol.
- Enter media -> create landscape board. You should notice after selecting this command that it defaults to 800 by 480. This is the width and height of the game board in pixels; 800 by 480 is the default resolution for windows phone games. If you want the game to be portrait the command becomes “create portrait board” and it will default to 480 by 800. If you are developing a Windows 8 application the resolution is 1366 by 768. However when you are testing your script, TouchDevelop will automatically rescale the board for you to fit within your browser.

  var board := media → create landscape board(800, 480)

- TouchDevelop assigns a default variable name of board to a variable of type game board. You can change this by selecting board and using the RENAME button. However I would just leave it as board, which is a good enough name for the game board.
- The board variable really needs to be global, as we will need to redraw it on the wall as the game progresses. So select board and then select the “promote to data” button. The keyword VAR should disappear to be replaced by a symbol which means global variable.
• Add a new line and select the DATA button (which lists all the global variables) and choose board. Then enter the command shown below after board, which will set the board to black.

```
board → set background(colors → black)
```

• We need to draw our game board onto the wall, so add the line below to do this.

```
board → post to wall
```

• We need this to be drawn repeatedly as the game progresses and as things change on the game board. So let’s add a GAMELOOP event. An event is much like an ACTION in TouchDevelop except it is triggered by an EVENT. In this case the EVENT that triggers it is just a timer which triggers the gameloop approximately every 50 milliseconds. This is similar to cartoons where animations are created by moving the characters a little in each frame. The gameloop is the place where your game comes to life.

• Select the “add new action, event” button from the panel at the left hand side, then select EVENT & then GAMELOOP from the list that pops up.

• Select the new gameloop event and inside it enter two lines shown opposite. The first line evolves the gameboard, which basically means it updates the positions of sprites or anything else on the board. The second line updates the board, which means it re-draws the board on the wall.

• Run your script and you should now have a black background board in the middle with a red wall behind it. Imagine that the black box is the screen on your phone.

• Now let’s add our first sprite to our game. Add the following lines of code into the MAIN action script. The first line adds an ellipse shaped sprite into the game board, with a width of 100 and a height of 100. The second line sets its colour to orange.

```
var sprite := board → create ellipse(100, 100)
sprite → set color(colors → orange)
```

• Run the script and you should see an orange ball in the middle of the screen.
- We can add some basic physics into the game by setting gravity. Add the line below, which applies a constant downwards force of 200, simulating gravity which is also a constant downwards force.

\[ \text{board} \rightarrow \text{set gravity}(0, 200) \]

- Run the script and the sprite should now fall out the bottom of the game board. Below is a diagram demonstrating how the positions work in a TouchDevelop game board, like the one we have been using. This is why setting the Y value of gravity to 200, forces sprites to move downwards, because the bigger the Y position of a sprite gets, the farther down the screen it will appear.

- Add the line below which creates a boundary around the game board using the “create boundary” command. Setting the boundary to 0 puts the boundary right around the edge of the board, you can bring this boundary in by setting a negative value or out by giving it a positive value. The sprites hitting the boundary will automatically bounce on it.

\[ \text{board} \rightarrow \text{create boundary}(0) \]

- Run the script and you should notice that the sprite no longer falls out the bottom, but bounces on the bottom.
- Try experimenting with the boundary value to understand what I meant by bringing the boundary in and out. Try 100, then -100 and then experiment with other numbers.
- Try changing the gravity, for instance setting gravity to \((10, 200)\) and you will find that the sprite will move to the right as well as down. This could be used to simulate wind blowing from left to right. Try also \((0, -200)\) or \((200, 0)\) and take notice of what happens.
- Our final physics lesson for this activity is friction. Pretty much any object in the real world has some sort of resistance acting on it, which acts to slow that object down when it is in motion. This resistance could be caused by air resistance or by friction between an object and the surface it is moving on. Apply a default friction to all objects on our game board by entering the line below. You can try setting the friction to values between 0.0001 and 1.

\[ \text{board} \rightarrow \text{set friction}(0.01) \]
• Run your script again and you should notice that the sprite now comes to a stop after bouncing a number of times.
• Another property we can set on a sprite is its speed. The speed is really it’s velocity as it gives the sprite speed on both the x axis and the y axis, which means we are also defining the direction it is moving in as well as the speed (which is velocity).

```
sprite → set speed(400, 0)
```

• Run the script and the sprite should now move initially quite quickly from left to right, but again gravity and friction take effect pulling it down and eventually bringing it to a full stop.
• Another property we can set for a sprite is its position. Enter the line below which will start the ball in the top left corner of the screen.

```
sprite → set pos(0, 0)
```

• Run the script and the sprite should come from the top left hand corner of the screen and move across the board in a projectile fashion.
• Try setting the sprite’s initial position to (0, 480) and the speed to (400,-200) and see what happens. We have basically simulated projectile physics in this example, much in the vein of games such as Angry Birds.
• Experiment with different settings for speed, friction, gravity etc. You could also try using “CREATE RECTANGLE” instead of “CREATE ELLIPSE” which will create a rectangular shape for the sprite instead of a circular one.
• Our final activities for this session will involve making our game look nicer. We will replace the sprite ellipse with a proper PNG graphic image of a soccer ball and we will add a nice background image to the game board.
• First of all we need to choose a picture. Select the “add new action, event” button & then select pic.
• Give the new piece of art the name ball.
• You have four choices for choosing the art for your games. The first is to enter a URL to an image (this could be found using your search engine of choice). The 2nd method is to “search online art pictures” and choose a piece of art already uploaded by another TouchDevelop user. The 3rd is to search for an image using “search Bing images” and lastly you can upload your own image.
• Probably the simplest method is to “search online art pictures” and use one of the 3 images I have uploaded for this activity. Select the button to search online and then type in a description of the art you are looking for and then select it from the matches which appear.

If you are uploading a picture for the ball sprite, select the UPLOAD button. When you go into the UPLOAD window, you must select an image from your hard drive, give it a name and select publish. The image should be a PNG and the white space around the sprite image should be set transparent, something jpgs can’t do.

• Repeat this process for another art resource which we want to name “game background”. Search for a suitable image from the online art pictures or upload a suitable background image, either in jpg or png format. Preferably its size should match the game board resolution we have been using of 800 by 480.

• Go back into the MAIN action script and alter the 2 lines which set the board background colour & creates the sprite, by changing it to match the lines below. To select the your art, you must select the art button and then choose the name of your art from the buttons that will appear on the bottom right. You can also delete the line which sets the colour of the sprite as it doesn’t need coloured now that it is a picture.

```
board -> set background picture("game background")
var sprite := \board -> create picture("ball")
```

If the picture is too big then try entering the following line of code to resize the sprite, by specifying a new width for it.

```
sprite -> set width(100)
```
Run the script again and you should now have a much nicer looking version, which should look something like the one below.

One final trick before we end this session, let’s make the ball spin based on its horizontal movement. To do this we are going to have to set its rotational speed in the game loop. To access the sprite in the game loop it needs to be set to global. So go back into the MAIN action script and PROMOTE the sprite variable to DATA, so that it becomes global.

Next go into the game loop script and add the following line, which will set the rotation speed of the ball based on the current speed of the ball. If you want to just set the angle of a sprite, rather than the angular speed, use the “set angle” command instead of the one below.

Run the script and you should have a nice simulation of a ball being thrown.

Back out of your game script to the HUB so that it syncs.

End of session 1
Session 2 – Creating your first game

When finished, students will be able to:

- Use random numbers
- Use IF conditions and logical operators
- Add actions and events
- Add comments
- Add and subtract numeric variables
- Create and display text sprites
- Add time limits
- Add sound effects
- Clone and publish scripts
- Create high score tables
- Add and use libraries
- Check for collisions between sprites

When finished, students should have a basic understanding of:

- Conditions
- Logical operators
- Arithmetic operators
- Functions (Actions and Events)
- Parameter passing

Cloning and publishing scripts

TouchDevelop allows you to re-use existing scripts, by simply creating a copy of an existing script. This script may be one you created or it may be one created by another TouchDevelop user. This feature is called “clone script”, which is very useful if you are making a game which shares a lot of similarities with an existing game script and it also allows you to try different things in a script without worrying about messing up the original.

When you publish a script (Unless you publish it as hidden) it allows other TouchDevelop users around the world to run your script and make changes to the code, by installing it into their account. This is something that you can take advantage of and learn from other users by installing their scripts into your account and having a look at the way they have done things. Publishing a script in TouchDevelop is not the same as commercially publishing your application to the Windows store.

Random numbers

Random numbers are used extensively in almost all computer games, as without some element of randomness in a game they quickly become very predictable and boring. They can be used to set the speed or direction that an enemy is moving in or used to randomise choices made by the artificial intelligence (AI) elements in a game such as which enemy should appear and how skilled the enemy character should be and so on.
Conditions

Programmers need to think in a logical manner and a massive part of that are the decision points in code, known as conditions. All high level programming languages have some form of IF THEN ELSE statement, where you can make choices based on logical conditions, a pseudo-code example would be: IF LIVES EQUALS ZERO THEN GAME IS OVER

In TouchDevelop the code for this would look as shown opposite. The ELSE condition occurs whenever the condition set in the IF part is false, so whenever lives is not equal to zero the ELSE condition runs.

In the example opposite the line which calls the game over action is indented, that is the way you know in TouchDevelop if code is part of the IF condition or not. If you add a line inside an IF condition or indeed a LOOP (which we will look at later on) you can choose whether to make it part of it or not part of it. Once you add a line you will notice a button with an arrow inside it called “move left” (as shown opposite). If you select “move left” then the line will no longer be part of the condition.

In the 2 examples below there is a subtle difference in the 2nd example where the last line is not indented. In the example on the left, if lives equals zero it runs the game over action, if lives is not equal to zero it adds 5 to score and then adds another 1 to score. In the example on the right the difference is that 1 is added to score regardless of what the result of the IF statement is.

Sometimes you need multiple conditions, for instance the game should end if lives reaches zero or the time limit runs out. This is where logical operators come into play (AND, OR and NOT). In the example below, if the gametime is less than or equal to 0 OR lives is equal to zero then the game over action is called. You can use the AND operator to check if two conditions are true, for instance in a 2 player game you might check if player 1 lives equals zero and player 2 lives equals zero before ending the game. I’ve included a list of all the logical operators on the next page.
Logical operators in TouchDevelop

<table>
<thead>
<tr>
<th>Description</th>
<th>Symbol</th>
<th>Usage example</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQUAL TO</td>
<td>=</td>
<td>score := score + 10</td>
<td>Adds 10 to score</td>
</tr>
<tr>
<td>NOT EQUAL TO</td>
<td>≠</td>
<td>lives := lives – 1</td>
<td>Subtracts 1 from lives</td>
</tr>
<tr>
<td>GREATER THAN</td>
<td>&gt;</td>
<td>score := timeleft * 5</td>
<td>Sets score to 5 times the value of timeleft</td>
</tr>
<tr>
<td>LESS THAN</td>
<td>&lt;</td>
<td>timeleft := timeleft / 2</td>
<td>Divides timeleft by 2</td>
</tr>
<tr>
<td>GREATER THAN OR EQUAL TO</td>
<td>≥</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LESS THAN OR EQUAL TO</td>
<td>≤</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOGICAL NOT</td>
<td>not</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOGICAL AND</td>
<td>and</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOGICAL OR</td>
<td>or</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

As you will probably have noticed in the previous examples where we added numbers to the score variable, TouchDevelop also has a number of arithmetic operators.

Arithmetic operators in TouchDevelop

<table>
<thead>
<tr>
<th>Description</th>
<th>Symbol</th>
<th>Usage example</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>+</td>
<td>score := score + 10</td>
<td>Adds 10 to score</td>
</tr>
<tr>
<td>SUBTRACT</td>
<td>-</td>
<td>lives := lives – 1</td>
<td>Subtracts 1 from lives</td>
</tr>
<tr>
<td>MULTIPLE</td>
<td>*</td>
<td>score := timeleft * 5</td>
<td>Sets score to 5 times the value of timeleft</td>
</tr>
<tr>
<td>DIVIDE</td>
<td>/</td>
<td>timeleft := timeleft / 2</td>
<td>Divides timeleft by 2</td>
</tr>
</tbody>
</table>

Actions

Functions are called Actions in TouchDevelop and I will refer to them as Actions from this point forward. TouchDevelop also has events, which are Actions which are triggered by an event which occurs. The event could be that the user touches the screen on a sprite or presses the camera button on their device.

Actions can have parameters which you pass in and out. Below is a very simple example of this to try and get the principal across.

The action on the left has 2 input parameters called A & B; these are local & exist temporarily within the action. The same goes for the output parameter (answer) it is also local & temporary. If you look at the picture above on the right, it shows how to call these actions. I have demonstrated two different ways; in the first I have declared a variable called result and I have assigned to it the result from calling “add numbers” and passing in the numbers 1 and 2. The outcome of this is that the
result variable will now hold the value 13, because inside the action b has 10 added to it and then a is added to b, so basically it does result = 1 + 2 + 10 (which adds up to 13)

In the second example I have declared two variables, assigned values to them and then passed them into the same function. The result of calling it this way is that result2 will be equal to 22 (10+5+7), however x and x2 will still be equal to 5 & 7, despite the fact that x2 is passed into B and B has 10 added to it, the original x2 is unchanged as B is just a temporary copy of it. In TouchDevelop variables of most data types, when passed in as a parameter, remain unaffected by what happens inside the action which they were passed into and inside that action a temporary copy of the original is made. However, if you pass a sprite in, the local variable points to the original sprite and any changes made to the sprite will affect the original.

Some actions may be useful in more than one game script; in this case you may want to put them into a library which allows you to use that action again in another script without having to re-create it. Libraries can be added into any script you create and then used in the same way as any other actions. If you publish your library other TouchDevelop users can also make use of them. In the exercises which follow, I will get you to make use of a library that I wrote, which contains some useful actions that come in handy when creating game scripts.

**Collisions**

The game board in TouchDevelop has built in collision detection, although it doesn’t actually check for sprites touching each other, it checks for them overlapping, a subtle difference which is important to take account of when creating a game script. In the example below we are checking if bat1 sprite overlaps with ball sprite and if it does we then set the ball’s horizontal speed on the x-axis to positive to reverses its direction and simulate the ball bouncing off the bat. The math->abs function is a maths function to set any number to positive, whether it was already positive or not.

```plaintext
if  □ bat1 → overlaps with(□ ball) then
    □ ball → set speed x(math → abs(□ ball → speed x))
else do nothing
```
Activity 2a – Creating a Fruit Ninja clone

- Login to your TouchDevelop account and open up the script that we created in Activity1b.
- Select EDIT and then select where it says the name of your game.

- Select “clone script” and a copy of your game script will be created.

- In the new script go into the same “script name” screen and change the name of the script to something new, if you can’t think of anything just call it “Fruit Ninja Clone”.
- Select the main action and delete out the lines which set the position of the sprite, the speed of the sprite and the boundary around the game board as shown below.
Let’s make use of random numbers to generate the position of the sprite and the speed of the sprite. When you use the math->random function you pass in a number to tell it how many random numbers to generate. So for instance math->random(10) returns a number between 0 and 9, which is 10 different numbers starting at 0.

Enter the lines of code below at the end of main action and test that it works by running it.

This first line above sets the Y position of the sprite to below the bottom of the board, by adding the height of the sprite to the height of the board. The second line generates a random X position for the sprite of between 0 and the board width, putting it somewhere between the left and right sides of the board. The third line generates a negative vertical speed for the sprite of between -300 and -699, to make the sprite move upwards. The IF condition, generates a horizontal speed of between 0 and 399, if the sprite starts on the left hand side of the board. It generates a speed of between -399 to 0 if the sprite starts on the right hand side of the board. This is so that the sprite doesn’t fly out of the board right away, but moves left if it spawns on the right and right if it spawns on the left.

The code above basically generates a position and a speed for the sprite, which is something we need to do over and over again, so this code should really be in its own action. So select the button and highlight the section of code as shown below. Name the new action “fire sprite” and select extract.

You should find you have a new action in your script and a new line in your main action which calls the new “fire sprite” action.
Now we want the sprite to fire again and again each time it falls out of the screen, so add the following code into the gameloop action, which will re-fire the sprite when it leaves the game board at the bottom.

```plaintext
event gameloop ()
| sprite → set angular speed(square sprite → speed x)
| board → evolve
| board → update on wall

if (square sprite → y > square board → height + square sprite → height) then
| ▶ fire sprite
| else do nothing
```

Add a new event for “tap sprite: sprite” which will be called when the player taps on the sprite.

Open the new tap sprite event and add in a call to the fire sprite action. So that the sprite re-fires when it is tapped on.

Run the script now and check it works.
Next we need to add score into the game. Go into the main action, add a global variable called score and give it an initial value of 0. Remember the process in TouchDevelop for declaring (creating) a variable. First you must select or type VAR, then if you wish, change the variable name, in this case to score. Then enter an initial value for the variable on the right-hand side of the equals’ symbol, in this case the number 0. Lastly select the new variable name (score) and select the “promote to data” button.

Next, in the main action, create a global text sprite variable called scoreboard, by entering the lines of code below. Notice that I have suggested that you set friction to 1, which stops gravity from moving this new sprite. The “create text” function has 4 parameters passed into it; the 1st value is the width of the text box, the 2nd is the height of the text box, the 3rd is the font size and the last bit is the string you want to draw. Feel free to change the colour of the text and change the position that I have suggested.

Add the following line into the gameloop action which sets the string for the scoreboard by converting the score numeric value into a string and concatenating it to the string “Score”.

In the tap sprite event, add the following line to add 10 to the score each time a sprite is touched.

Run the script and test that it works. You almost have a game, except that it never ends.
• Before we go any further let’s make the game more fruit ninja style, so go into the ART group and change the pictures you are using for the BALL and the GAME BACKGROUND. Feel free to go with any theme you want with regards to the art.

• Now let’s add a timer and a text sprite to display the timer. The time->now function returns the current system time, we will use this to store the time when the game starts. Enter the code below at the end of the main action script.

```plaintext
start time := time -> now
timeboard := board -> create text(200, 100, 40, "")
timeboard -> set friction(1)
timeboard -> set pos(400, board -> height - 20)
timeboard -> set color(colors -> black)
```

• In the gameloop action add the code marked in red. The first line works out the time remaining in the game and stores it into a variable called timeleft. The second line updates the timeboard text sprite, so that the time left is displayed on the screen.

```plaintext
event gameloop ()

scoreboard -> set text("Score " || score)

var timeleft := 60 - math -> floor(time -> now -> subtract(start time))
timeboard -> set text("Time Left " || timeleft)
sprite -> set angular speed(sprite -> speed x)
```
• Add the following lines at the end of gameloop to check for the game ending when time left is less than or equal to 0. When it does reach zero the next 2 lines of code use the Bazaar library to create an online leaderboard for your game. Once you publish your game, people around the world will be able to play your game and get on your leaderboard. The first line posts the score variable to the online leaderboard and the second line draws the leaderboard to the wall. Time->stop stops the script from running and the game is over.

```
if timeleft <= 0 then
    bazaar -> post leaderboard score(score)
    bazaar -> post leaderboard to wall
    time -> stop
else do nothing
```

• Run your game and check the game ends when the time runs out.

• In fruit ninja, part of the fun is that you swipe your finger through the fruit, not tap on it, so let’s change that. There isn’t a built in function in TouchDevelop as of yet to do this, but there is a library which you can use which allows you to do a “swipe through” event on a sprite. Libraries are made up of actions and variables, which combine to provide commonly used functions. These can be created by any TouchDevelop user, so perhaps one day soon you will create a library that does a specific task that you think other TouchDevelop users could make use of. The first thing we need to do is add the library, select “add new action, event” and then select lib.

• Search for the “game events” library and select it. It should then be added to your libraries.

• Delete the tap sprite event action and add a “swipe board” event instead.
In the swipe board event add the lines below, which do the same thing we were doing in the tap sprite event, but it runs when a swipe through the sprite event occurs. To bring up the “game events” library select the library button ⚪️. Note that x & y form the start position of the users swipe and “delta x” and “delta y” gives the vector length of the swipe. The last parameter you pass into the “check swipe” action is the actual sprite that you want to check for a swipe against.

```javascript
event swipe board: board (  
  x : Number,
  y : Number,
  delta x : Number,
  delta y : Number)

  do
  if ⚪️ game events → check swipe(x, y, delta x, delta y, ⚪️ board, ⚪️ sprite)
  then
    ⚫️ fire sprite
    ⬜️ score := ⬜️ score + 10
  else do nothing
```

Run the script and test that you can swipe through the sprites correctly. If you are using a mouse, you need to click & hold the LMB while moving the mouse to swipe and then release the LMB when the swipe is over.

Before we finish the game and publish it, let’s make things a little more interesting by adding a second sprite which the player must try to avoid swiping. To help with adding a second sprite, let’s change the “fire sprite” action so that we can use it to fire more than one sprite. To do this we will pass in the sprite that we want to fire. So change it to match the one shown on the next page, by adding an input parameter for sprite. You do this by selecting the main name for the action where it says “private action fire sprite” and then selecting “add input parameter”. By default it creates a number parameter called p, select this and an option should appear on the left saying “of type” with Number underneath. Select Number and a window should appear which allows you to search for and select a variable type. Select Sprite from this list.
• Next go through the fire sprite event action changing all the global references to the new local sprite variable (you need to delete the sprite references and replace them with just sprite).

```c
private action fire sprite (  
  | sprite : Sprite)  
  do  
  | sprite  y(|board  height  sprite  height)  
  | sprite  x(math  random(|board  width))  
  | sprite  set speed y(-300  math  random(400))  
  | if sprite  x < |board  width  2 then  
  |   | sprite  set speed x(math  random(400))  
  | else  
  |   | sprite  set speed x(-math  random(400))  
```

• Once you do this you will need to go through the main, swipe board and gameloop actions, changing fire sprite to fire sprite(sprite). Once you have done this, run the script and make sure it still works as it did before.

• Let’s also extract the 3 lines of code which happen at the end of the game into their own action, so that we can call it whenever we want. So extract the 3 lines of code shown below into a new action called gameover.

```c
private action endgame ()  
  | bazaar  post leaderboard score(score)  
  | bazaar  post leaderboard to wall  
  | time  stop  
```
Now we can add a second sprite. So add the 2 lines shown below to the main action around the same place where you declared the original sprite. Note that the order that you declare sprites in the code, determine which is drawn first and therefore which sprite will appear on top if two sprites are drawn on top of each other, the sprite declared last is last drawn and therefore on top of the rest. You will also need to add another piece of art for the badsprite, the picture I am using is called “Brussel Sprout” and the art resource name I used was “not fruit”, but feel free to choose a different image.

```plaintext
badsprite := board -> create picture( apple not fruit)
fire sprite( badsprite)
```

Add the following lines of code (marked in red) into the gameloop action, the first line makes the new badsprite spin based on its horizontal speed. The IF statement checks for the new sprite leaving the bottom of the screen and re-fires when it does.

```plaintext
sprite -> set angular speed( sprite -> speed x)
badsprite -> set angular speed( badsprite -> speed x)
if (sprite -> y > board -> height + sprite -> height) then
    fire sprite( sprite)
else do nothing

if (badsprite -> y > board -> height + badsprite -> height) then
    fire sprite( badsprite)
else do nothing
```

To check to see if the player swipes through badsprite, we need to add the IF statement below into the swipe board event, so that when it detects a swipe through badsprite, it calls the endgame action.

```plaintext
if game events -> check swipe( x, y, delta x, delta y, board, badsprite) then
    endgame
else do nothing
```
• Lastly, let’s add a sound effect when the fruit is swiped. Go to the Art group and add a new art resource, which we will name swipesound. Select the “of type” button and change it from Picture to Sound. The sound I would suggest using can be found by selecting “search online art sounds” and searching for fireball.

![Image](fireball)

• Add the following line by selecting the ART button and choose swipesound and then play. You should now find that when you swipe the fruit it will play a sound effect.

```plaintext
if game events → swipe through(board, sprite) then
  fire sprite(sprite)
  score := score + 10
  swipesound → play
```

• Run your script again and test that the sound plays.
• Now that your game is complete, let’s publish it so that other users can play it. Publishing it is not putting it into the Windows Store to sell, it’s simply allowing other TouchDevelop users to try out your script and make use of your code. As soon as you publish it other TouchDevelop users can play your game and get entries on your high score table. To publish select the PUBLISH button beside the name of your script and select PUBLISH on the window that pops up.
- Once you have published it, try your game again and set a high score for others to try and beat.

![Score 70](image)

- You can also tweet, email and facebook like your script, so friends of yours can log into TouchDevelop and try it out. To do this, select the SHARE button from the main screen of your script as shown below.

![Share Button](image)

- Finally hit the back arrow and return to your HUB.
Activity 2b – Creating a Pong clone

- Login to your TouchDevelop account.
- Select the “Create Script” button & select “physics game starter” which creates a basic game script that we can use as a template to save a bit of time.
- Name the script “PONG” and select Create.
- Go into the Main action and set the gravity to (0, 0).
- Add a line of code to set the game board to have a black background (as shown opposite).

- Next we will create a ball for the game. Enter the lines of code below into the main action, which will create a white ball in the middle of the screen and assign it a random speed. Use the comment button to enter the comment on the first line. In any program language it’s important to comment your code. Comments are just descriptions to explain what different bits of code do, to make it easier for someone else looking at the code to understand it. Comments in touchdevelop are colour coded in grey. Remember when you want to create a global variable such as the ball below, you must first create a variable, rename it and then promote it to data. In the first line of code below we create a global variable called ball and create a 20 by 20 ellipse (filled in circle) for the ball. Next we set the position of the ball to the middle of the game board, by dividing the game board width & height by 2 and assigning those values to the x & y positions of ball. Then we randomly create the x & y speed for the ball, by setting the x speed to 100 + a random value between 0 & 99 (100 to 199) and the y speed to 50 + a random value between 0 & 99 (50 to 149). Then we use an IF statement to set the x speed randomly to negative, by checking if the result of a random command gives us 1. We repeat this process for the y speed.
• Run the script and check that a white ball appears in the middle of a black board and that the ball moves in a random direction.
• In pong the ball bounces back off the top and bottom walls and goes through the left and right sides and reappears at the opposite side. We need to check for the ball hitting the four walls and make it respond accordingly. We will use IF statements for this in the gameloop, however when doing this we must consider that in sprites the position is the centre of the sprite, as shown in the diagram opposite. If we take this into consideration, then depending on which side of the sprite we want to check, we will need to add or subtract half of the width or half of the height, from the centre point to get to the edge.
• We will need to re-fire the ball sprite when it leaves the left or right sides of the screen, so let’s extract the code which does this in the main action, into its own action called `fireball`.
• Enter the following lines of code below into the gameloop event.

```plaintext
Check for ball hitting the top & bottom and reverse speed on the y if it does
if ball → y - ball → height / 2 ≤ 0 then
    ball → set speed y(math → abs(ball → speed y))
else do nothing
if ball → y + ball → height / 2 ≥ board → height then
    ball → set speed y( - math → abs(ball → speed y))
else do nothing
Check for the ball leaving the left or right sides
if ball → x + ball → width / 2 < 0 or ball → x - ball → width / 2 > board → width then
    fireball
else do nothing
```

• The first IF statements checks for the ball hitting the top of the screen by subtracting half of the ball height from the ball y position (to get the top of the ball) and then checks if this position is less than or equal to zero. If it has hit, we use a Math absolute function to set the y speed to positive, which will make the ball move downwards. The reason we don’t just reverse the speed is because this sometimes causes a bug in pong where the ball gets jammed against the top or bottom of the screen, as the speed keeps reversing back and forward.

• The second IF statement checks for the ball hitting the bottom of the screen by adding half of the ball height onto the ball y position (to get the bottom of the ball) and checks if this position is greater than or equal to the board height. If it has hit, we negate the absolute value of the y speed to force it to be negative, which moves the ball upwards.

• The third IF checks if the ball has fully left the left hand side of the screen, by checking if the ball x position plus half of the ball width is less than zero. So basically we are checking if the right hand side of the ball leaves the screen on the left. If this occurs we re-fire the ball from the centre of the screen with a new random speed.

• Run the script and make sure the ball bounces off the top and bottom and re-fires from the middle when it leaves either side of the screen.
• Go back into the Main action and add two bats, as shown below. We are setting one bat to red and placing it 80 pixels in from the left side and putting it at half the board height vertically. We are making the other bat yellow, bringing it in 80 from the right hand side of the board and also placing it at vertically in the middle. I’m suggesting we bring the bats in 80 from each side to give room for a player to put their finger behind the bat when playing on a touch screen device.

bat1 := board -> create rectangle(10, 100)
bat1 -> set pos(80, board -> height / 2)
bat1 -> set color(colors -> red)
bat2 := board -> create rectangle(10, 100)
bat2 -> set pos(board -> width - 80, board -> height / 2)
bat2 -> set color(colors -> yellow)

• In gameloop add the following lines of code which check for the ball hitting either bat. If the ball does hit a bat then this will change the x speed to make the ball move in the correct direction, so that it appears to bounce off the bat. We are basically forcing the x speed of the ball to be positive when it hits the left hand side bat and negative if it hits the right bat.

Check if the ball hits the bats and reverse the x speed if it does

if bat1 -> overlaps with(ball) then
    ball -> set speed x(math -> abs(ball -> speed x))
else do nothing
if bat2 -> overlaps with(ball) then
    ball -> set speed x( - math -> abs(ball -> speed x))
else do nothing
• Also in gameloop add the following code, which allows the user to control the first bat by touching the screen and sets the yellow bat to move up and down itself depending on whether the bat is above or below the bat.

```
if board -> touched then
    | bat1 -> set y(board -> touch current -> y)
else do nothing

Move the second bat based on the position of the ball

var ai_batspeed := 200
if (bat2 -> y < ball -> y) then
    | bat2 -> set speed y(ai_batspeed)
else
    | bat2 -> set speed y(-ai_batspeed)
```

• Run the script and check that it works. If you are using a mouse, simulate touch by holding the LMB down and moving the mouse up and down.

• To make the game more interesting I suggest you add the following lines of code, which gradually increase the speed of the ball over time and will make the game more competitive. We do this by multiplying the x and y speed properties of the ball by a new variable called acceleration. Put this into gameloop and run the script and test it.

```
Gradually increase the ball speed

var acceleration := 1.001

ball -> set speed(ball -> speed x * acceleration, ball -> speed y * acceleration)
```
We need to add score and lives to make our script into a game. Go back into main and declare the following global variables for score and lives, also declare 2 text sprites to display the score and lives.

### Variables to store score and lives

- `score := 0`
- `lives := 3`

Add text sprites for score and lives

- `scoreboard := □ board → create text(200, 100, 40, "")`
- `livesboard := □ board → create text(200, 100, 40, ")")`
- `scoreboard → set pos(160, 50)`
- `livesboard → set pos(600, 50)`

At the top of gameloop add in the following 2 lines to update the text sprites with the current score and lives.

- `scoreboard → set text("Score " || □ score)`
- `livesboard → set text("Lives " || □ lives)`

Let’s add in an action called ENDGAME like we did in the previous game we created.

```plaintext
private action endgame ()

"Game Over" → post to wall
bazaar → post leaderboard score(□ score)
bazaar → post leaderboard to wall
time → stop
```
• Change the 2 lines of code which check for the ball leaving the left and right sides, into 2 separate IF statements, which also assign points if the player scores a goal and subtract lives if the computer controlled bat scores a goal. It also contains an IF which checks for lives reaching zero and calls the endgame action when it does.

```plaintext
if □ ball → x + □ ball → width / 2 < 0 then
  ▷ fireball
  □ lives := □ lives - 1
  if (□ lives = 0) then
    ▷ endgame
  else do nothing
else do nothing

if □ ball → x - □ ball → width / 2 > □ board → width then
  ▷ fireball
  □ score := □ score + 5
else do nothing
```

• Run the script and test the game as you should now have a working game.
• The original pong had special “game physics” which weren’t really representative of real world physics, but nevertheless make the game more interesting, as it allowed you to aim the ball. Basically when the ball hit near the top of the bat the ball it went upwards and when the ball hit near the bottom of the bat it went downwards. The farther away from the centre of the bat the ball hit, the more it would change direction vertically. Apply the following lines of code marked in red to do an approximation of this effect. The first line uses maths to calculate the magnitude of the speed. We then work out the difference between the ball position and the bat position when they collide and multiple this difference by the magnitude of the speed to set the new speed on the Y axis.

```plaintext
var ball speed := math → sqrt(□ ball → speed x * □ ball → speed x + □ ball → speed y * □ ball → speed y) / 50

if □ bat1 → overlaps with(□ ball) then
  □ ball → set speed(math → abs(□ ball → speed x), □ ball → y - □ bat1 → y) * ball speed
else do nothing

if □ bat2 → overlaps with(□ ball) then
  □ ball → set speed(- math → abs(□ ball → speed x), □ ball → y - □ bat2 → y) * ball speed
```

TouchDevelop Curriculum

Author: David Renton
TouchDevelop Curriculum

- Run the script and check that it works.

Challenges
Making use of what you have learnt in the previous exercises, try to complete the following challenges.

- Set a new background using a picture (use board->set background picture).
- Change the appearance of the ball and the two bats using graphics (use board -> create picture).
- Make the ball spin based on the x speed of the ball.
- Add sound effects for when the ball hits the bats and perhaps for when a goal is scored.
- Adjust the numeric values being randomly generated in the fireball action, to increase or decrease the speed that the ball moves at initially.
- Adjust the AI bat speed to increase or decrease how difficult it is to beat the AI controlled bat.
- Publish the script.

End of session 2
Session 3 – Creating games with multiple sprites of the same type

When finished, students will be able to:

- Add obstacles
- Resize picture sprites
- Create sprite sets
- Add and remove sprites from a sprite set
- Delete sprites
- Use FOR and FOR EACH loops
- Check for collisions between multiple sprites and handle collision response

When finished, students should have a basic understanding of:

- Collections (arrays/lists)
- Loops

Collections

Most programming languages have some way of creating a collection of variables, usually in the form of an Array or Linked list. In TouchDevelop they are simply called collections and you can have collections of a number of different variable types, such as a collection of numbers, a collection of strings and even a collection of sprites. The one that we will make use of most in games, is the collection of sprites, which is referred to in TouchDevelop as a Sprite Set.

Collections allow you to create multiple instances of the same variable type, linked to a single variable name. Each instance is sometimes referred to as an element. You can add or remove elements from the collection. You reference each separate element in the collection by using an index number. For instance we could create a sprite set called aliens, by entering the line of code shown below.

```
aliens := board -> create sprite set
```

We could then create a local sprite variable and then add it into the sprite set, as shown opposite. The first 3 lines just create a sprite and set its position and speed. The last line adds that specific sprite as an element of the sprite set aliens.

```
var sprite := board -> create picture(❤️ monster)
sprite -> set pos(100, 200)
sprite -> set speed(50, 100)
aliens -> add(sprite)
```

If this is the first sprite you have added into the collection, you could reference it as shown below. In this example, we are setting the height, for the same sprite that we declared above.

```
aliens -> at(0) -> set height(20)
```

The reason it says at(0) is because 0 is the first element in the collection, the second would be 1 and so on.
You can also remove elements from collections; the example below show the two different methods. The first line would look through the collection for an element matching sprite and remove it. The second line would remove the sprite at position 2 in the sprite set, position 2 being the third element as the index starts at 0.

```
aliens → remove(sprite)
aliens → remove(aliens → at(2))
```

**FOR Loops**

We have already touched on loops a little, by using a gameloop in our previous games. However an important part of any programming language is the ability to create a loop to repeat a certain bit of code multiple times. In a lot of cases we want to repeat some code, a specific amount of times, which is where the FOR Loop comes in. In TouchDevelop, FOR loops repeat a certain amount of times, counting up a local number variable from 0 until a limit is reached (which you can set). In the example below the counter variable “i” starts at 0 on the first loop, then on the second loop it goes to 1 and so on until it hits 5 and at that point since “i” is no longer less than 5 it jumps out of the loop. So when it hits 5 it jumps out before executing the loop, so inside the loop “i” will never equal 5, however the loop is repeated a total of 5 times. Inside the loop “i” will be 0 on the first loop and 4 on the final loop.

```
for 0 ≤ i < 5 do
    var sprite := board → create picture(monster)
    sprite → set pos(100 + (i * 50), 200)
    sprite → set speed(50, 100)
    aliens → add(sprite)
```

This example adds 5 sprite elements to the aliens sprite set, each one with a slightly different x position which I am setting it to 100+(i*50). So the first element will be at 100, the second element at 150, the third at 200 and so on.

**FOR EACH Loops**

Another type of loop is the FOR EACH loop, which is used specifically with collections. This loop does not repeat a set amount of times; instead it repeats the exact amount of times needed to match the number of elements in the collection that has been specified. Each element in the collection is temporarily stored into a local variable (e in the example opposite), which can be used to access each individual element in the collection. In the example opposite, every element in the aliens sprite set is assigned a new width of 80. Notice the “where true” statement, you can change this to only select elements from the collection that match a certain criteria, leaving it as it is means all elements will be selected.

```
for each e in aliens
    where true
do
    e → set width(80)
```
WHILE Loops
WHILE loops repeat a section of code until a condition becomes false, so there is not a set amount of times that it will repeat, it just depends on when the condition is broken. In the example below the loop repeats as long as ballsprite has a “speed x” of less than or equal to 50. Inside the WHILE loop a random number of between 0 and 199 is being assigned into the “speed x”, so the loop will repeat until it generates a random number of greater than 50.

```
while ballsprite -> speed x <= 50 do
    ballsprite -> set speed x(math -> random(200))
```

Activity 3 – Creating a Breakout clone
- In TouchDevelop create a new “physics game starter” script and name it Breakout or something similar, as we are going to recreate the classic arcade game of the same name.
- Go into the main action and set the gravity to (0,0) effectively turning it off.
- Add the line of code shown opposite to set the game board background colour to black.
- Add a new picture to use in your game as a sprite. I suggest you search the online art pictures and choose a picture of a ball. Change the name the new art resource to ball.
- In the main action create a new global variable called ballsprite and load the new art resource into it. The second line below, uses a command called “set width”, which as you might imagine, sets the width of the sprite in pixels. This basically resizes the graphic, but it does it proportionally, so you don’t need to resize the height as well, but if you do want to there is a command for “set height” also.

```
ballsprite := board -> create picture(ball)
ballsprite -> set width(25)
```
Next let’s create a new action for firing the ball. Notice that I’ve added two input parameters, which I’ve named x and y, this is so we can pass in the position of where we want the ball to start at. To add a parameter you click on the name of the action, where it says “private action fireball” and then select the “add input parameter” option. The action simply sets the position of the ball to the x and y values that are passed in and then generates a random speed on the x axis and a negative random speed on the y axis to make sure the ball travels upwards first.

In Pong we used IF statements to check for the ball hitting the top and bottom of the screen, but there is another way to do it in TouchDevelop, we can use something called obstacles. In Breakout the ball bounces off the left, top and right sides of the screen, but not the bottom, so we want to create single line obstacles around those 3 sides of the screen. Add the 4 lines of code below into the MAIN action. The first line fires the ball from the bottom middle of the screen. The 2nd line creates an obstacle across the top of the screen, the 3rd line creates an obstacle on the left hand side and the 4th line creates an obstacle on the right hand side. You specify the obstacle by giving it a start position (x & y) and then setting the x segment which is the width & the y segment which is the height, lastly the final parameter is how much of a bounce the obstacle should cause, which is a value between 0 and 1 (try setting it to 0.5 to see the difference it makes).
• Let’s add in the user controlled bat, much in the same way as we did for Pong.
• First go to the art group and add in a suitable graphic for the bat. You can search online art pictures, search Bing images or upload your own. You can even go onto the World Wide Web and find a free image using your favourite search engine and copy and paste the URL in. Name the art resource as bat.
• Next add in a global sprite variable called userbat and load the bat graphic into it as shown below. Notice that we are also setting the size of the bat by setting its width to 100 and we are setting the position of the bat to near the bottom middle of the screen. I have suggested that you move the bat up 80 pixels from the bottom of the screen, this is because we are going to use touch control for moving the bat in the same way as we did in Pong and we need to leave room for the user to put their finger on the screen while still seeing the bat. Notice that I have also changed the position of where the ball fires from so that it now fires from the top of the bat.

```plaintext
| board → post to wall |
| userbat := board → create picture(bat) |
| userbat → set width(100) |
| userbat → set pos(board → width / 2, board → height - 80) |
| ballsprite := board → create picture(ball) |
| ballsprite → set width(25) |
| fireball(userbat → x, userbat → y - userbat → height / 2 - ballsprite → height / 2) |
```

To set the ball so that it starts on top of the bat, we first set the X positions of the 2 sprites the same; this is because the position of a sprite is the centre point of the sprite as shown in the diagram below. To move the ball to sit on top of the bat, we take the Y position of the bat and subtract half of the height of bat from it and also subtract half of the ball height. The diagram below illustrates this.

![Diagram showing the positioning of the ball and bat](image-url)
- Run the script and you should find that the bat is displayed and the ball fires from on top of the bat.
- Add the following lines of code (within the red box) which will check for the user touching the screen. When this occurs it will set the x position of the bat based on where the user touches the screen.

```
set the position of the bat based on where the player touches the screen
if board -> touched then
  userbat -> set x(board -> touch current - x)
else do nothing
```

The code above works perfectly on a touch screen device, but you may have a bug when using it with a mouse on a pc as you will be able to click outside the screen and put the bat off screen. If you want to stop this happening, replace the IF statement above with the one below, which will only move the bat if the user touches within the board area.

```
set the position of the bat based on where the player touches the screen
if board -> touched and board -> touch current - x > 0 and board -> touch current - x < board -> width then
  userbat -> set x(board -> touch current - x)
else do nothing
```

- Run the script and make sure you can control the bat by touching the screen or by holding down the LMB on the mouse and moving your mouse (if you are using a non-touch screen computer).
- Next let’s re-spawn the ball when it leaves the bottom of the screen. To do this we use an IF statement, with the condition being that the ball position on the Y is less than the bottom of the screen. So enter the following code into the `gameloop` action.

```
refire the ball if it leaves the bottom of the screen
if ballsprite -> y > board -> height + ballsprite -> height then
  fireball(userbat -> x, userbat -> y - userbat -> height / 2 - ballsprite -> height / 2)
else do nothing
```

- Run the script and test that the ball re-spawns when it leaves the bottom of the screen.
Next we need to make the ball bounce off the bat. To do this we use an IF statement with the condition being does the ball overlap with the bat. If it does then we set the speed of the ball on the Y axis to negative to make it go back up the way. Add the code below into the `gameloop` action.

```
if userbat -> overlaps with(ballsprite) then
    ballsprite -> set speed y(- math -> abs(ballsprite -> speed y))
```

Run the script and check that the ball bounces off the bat.

Breakout uses some special game physics, much like in Pong, to allow the player to aim the ball depending on the part of the bat the ball hits off. If the ball hits to the left of the bat, it sends the ball to the left, if it hits to the right, the ball bounces upwards and to the right. To do this correctly, involves some Vector Maths, which I’ve done for you in a small library. Add the “game events” library to your script, the same one we used in Activity 1b.

```
library reference

binds to script: game events
```

Go back into gameloop and change what happens when the ball overlaps with the bat, by deleting the original line of code and replacing it with the 2 lines shown below. We pass the speed (x and y) of the ball and the x positions of the bat & the ball into the “bat collisions” action and it returns a new x and y speed for the ball.

```
if userbat -> overlaps with(ballsprite) then
    var newx, newy := game events -> bat collisions(ballsprite -> speed x, ballsprite -> speed y, userbat -> x)
    ballsprite -> set speed(newx, newy)
```

Run the script and test that the collisions work properly and that you can aim the ball depending on the part of the bat it connects with.
Next we need to add in our bricks. First let’s add a sprite set for our bricks as we will need a lot of them, which means we will be making use of collections for the first time. Add the line below at the end of the MAIN action to create a sprite set.

```
bricks := board → create sprite set
```

Go into the art group and add a suitable brick graphic and call it brick.

Next we will add a new action called “add bricks” which will add the bricks and set the individual positions of each brick. So create the action shown below.

```
private action add bricks (
    rows : Number,
    columns : Number)

do
    | Add rows and columns of brick sprites into the game
    | bricks → clear
    | var firstx := 108
    | var firsty := 80
    | var xspacing := 5
    | var yspacing := 20
    | for 0 ≤ i < rows do
    |     for 0 ≤ j < columns do
    |         var brick sprite := board → create picture(brick)
    |         brick sprite → set width(60)
    |         brick sprite → set x(firstx + (j * (brick sprite → width + xspacing)))
    |         brick sprite → set y(firsty + (i * (brick sprite → height + yspacing)))
    |         bricks → add(brick sprite)
```
The action you just entered to create the bricks may seem a quite complicated action if you are new to programming. The action takes in two parameters which set the number of rows and columns of bricks. We issue the bricks->clear command to make sure the sprite set is empty. We declare variables called firstx and firsty, which will be used to set the position of the first brick in the set. We declare variables called xspacing and yspacing which will be used to set the gaps between the bricks. Feel free to change these values and see what happens. Next we create 2 FOR loops, one which loops from 0 up to the number of rows and then another FOR loop inside the first FOR loop which loops from 0 up to the number of columns. This means that if we pass in 4 rows and 10 columns, the code inside the two FOR loops will repeat a total of 40 times. Inside the FOR loops we create a brick sprite and load the brick picture into it. We set the width of the brick to 60 (again feel free to alter this) and then we set the x and y positions of each bricks to lay them out in rows and columns, making use of the first bricks position and the spacing variables. Lastly we add the new sprite into our bricks sprite set. This repeats until both FOR loops have completed.

Go back into MAIN and add in the following line to call this new action and create a sprite set of bricks.

```
bricks := board -> create sprite set
>add bricks(4, 10)
```

Run the script and you should have rows and columns of bricks on the screen as shown below.
• One major thing is missing; the ball doesn’t break the bricks! So let’s make this happen. Add a new action into the code group called “check bricks”. Then add the following code which uses a FOR EACH loop to loop through all the bricks and check for collisions between them and the ball. If a collision occurs the brick is removed from the sprite set and the sprite is deleted.

```csharp
private action check bricks ()
|
| Check for the bricks getting hit by the ball
|
| for each e in bricks
| |
| where true
| |
| do
| |
| if e → overlaps with(ballsprite) then
| |
| bricks ← remove(e)
| |
| 'remove' returns a 'Boolean'; insert 'post to wall' if you want to display it
| |
| e ← delete
| |
| else do nothing
```

• Call the “check bricks” action from the end of the gameloop as shown opposite.

• Run the script and check that the bricks get destroyed when they are hit by the ball.

• One glitch with our script is that the ball destroys the bricks and keeps going right through the space where the brick was and does not bounce off them, which is not meant to happen in breakout. Since TouchDevelop checks for sprites overlapping rather than touching, we need to create 2 new variables to hold the x and y positions of the ball sprite just before it overlapped with the brick. So we will declare 2 new global variables to do this by adding the following 2 lines at the bottom of the main action.

```csharp
ball oldx := ballsprite → x
ball oldy := ballsprite → y
```

• Add the code on the next page into the “check bricks” action. It will check which side of the bricks has been hit and make the ball respond accordingly. The first IF checks for the ball hitting the left hand side of the bricks and sends the ball left if it does. The second IF checks for the ball hitting the right hand side of the bricks and sends the ball right if it does. The third IF checks for the ball hitting the top of the bricks and sends the ball upwards if it does. The fourth IF checks for the ball hitting the bottom of the bricks and sends the ball downwards if it does. The local hit Boolean variable is used to store whether a collision has occurred with any of the bricks and if none have occurred the IF statement at the end (which is outside of the FOR loop) just stores the current position of the ball sprite into the “ball oldx” and “ball oldy” variables that we created.
• Run the script and check that the ball bounces correctly off each side of the bricks. Carry out some further play testing to make sure that the ball responds correctly when it collides with the bricks.
• You might be wondering why I have divided the height and width of the ball by 3 instead of 2, in the conditions for the IF statements; this is because we are dealing with a circular object not a square object. Using a third of the width and height rather than a half, works better, especially when the ball hits the corner of a brick as shown in the diagram above.

• Now that we have the game mechanics working, we just need to add score, lives and a counter so we know when the bricks are cleared, so we can spawn a new level. Since we will need to spawn a new level, let’s extract the line which creates the bricks into a new action called “new level”.

    private action new level ()
    ► add bricks(4, 10)

• Add the following lines of code at the bottom of the main action to create 4 new variables to store the lives, score and to display lives and score on the screen. Also make sure you still call the “new level” action to create the bricks.

    lives := 3
    lives display := □ board → create text(100, 20, 40, "")
    lives display → set pos(645, 10)
    score := 0
    score display := □ board → create text(100, 20, 40, "")
    score display → set pos(125, 10)
    ► new level

• Add the code marked in red opposite, into the “check bricks” action. It will assign 5 points for each brick destroyed. It will also check for when all the bricks have been destroyed by checking the number of sprites left in the bricks sprite set, using the count command which returns the number of items like in a collection. When this happens it will call the “new level” action to re-spawn all the bricks for a new level.

    bricks → remove(e)
    'remove' returns a 'Boolean'; insert e → delete
    hit := true
    if bricks → count = 0 then
    ► new level
    else do nothing
• Add the following 2 lines (marked in red) at the top of the gameloop action to update the text which displays the current score and lives, before the board is updated on the wall.

```
event gameloop ()

[Red text]
- score display → set text("Score " || score)
- lives display → set text("Lives " || lives)

Apply physics and redraw.
board → evolve
board → update on wall
```

• Create the following new action called endgame, which will post scores to the global leaderboard, display a “Game Over” message and then stop the script.

```
private action endgame ()

wall → clear
bazaar → post leaderboard score(score)
bazaar → post leaderboard to wall
"Game Over" → post to wall
time → stop
```

• Add the following code (marked in red) into the gameloop action, which will subtract one from lives when the ball leaves the screen and end the game when lives becomes 0.

```
if ballsprite → y > board → height + ballsprite → height then

▷ fireball(userbat → x, userbat → y - userbat → height / 2 - ballsprite → height / 2)
▷ lives := lives - 1
if lives = 0 then
▷ endgame
else do nothing
else do nothing
```
Run the script and check that the game functions correctly; in that score is added, lives are lost, bricks re-spawn if they are all cleared and that the game ends correctly.

Finally let’s add a couple extra things to make the game interesting. Add the code below at the end of the gameloop. It will gradually increase the speed of the ball to make the game more difficult as time goes on and it will spin the ball depending on the “speed x” property of the ball sprite.

```
Gradually increase the ball speed

var acceleration := 1.0005

ballsprite → set speed([ballsprite → speed x * acceleration], [ballsprite → speed y * acceleration])

ballsprite → set angular speed([ballsprite → speed x])
```

Run the script and check that everything works.

**Challenges**

Making use of what you have learnt so far, try and complete the following challenges in your game.

- Set a new game board background using a picture.
- Add sound effects for when the ball hits the bats and for when a brick is destroyed.
- Use different brick graphics for each row of bricks.
- Add a power-up sprite which falls from one of the bricks when it is destroyed, which can be collected by the bat and which will then increase the bat size until the end of the level.
- Publish your final game script.

End of session 3.
Session 4 – Creating games using the accelerometer

When finished, students will be able to:

- Create a scrolling background
- Rotate picture sprites
- Make use of the tilt function of the accelerometer
- Move multiple sprites at the same time
- Shoot a sprite from another sprite based on a touch event
- Use prompts to inform the player of something
- Prompt the player for a yes/no decision
- Use a sprite sheet to create a 2D animated sprite
- Create on-screen buttons inside a game for controls

When finished, students should have a basic understanding of:

- The Accelerometer
- 2D animation using sprite sheets

Accelerometer

Most mobile devices and tablet devices now contain an accelerometer and a gyroscope. The accelerometer can be used in conjunction with the gyroscope (which reports orientation of the device) to give a reading back on how much a device has been tilted in different directions.

The example below demonstrates how to read these values into a variable. The command I am using below “senses->acceleration quick” gives pretty much instant feedback when the user tilts their device, by returning a Vector3. Vector3 variables contain x, y and z properties and in this case x is the amount of tilt left to right, while y is the amount of tilt forward and back, z is to do with the orientation of the device which is not something we will be using in my exercises (basically whether your device is upside down or not). There are two other commands you may want to experiment with “senses->acceleration smooth” and “senses->acceleration stable”. These commands give feedback over a longer period of time, smooth over a medium time period and stable over a longer time period. I would suggest for games that “acceleration quick” is best as we usually want instant feedback. The value of x will range from -1 to 1, where -1 means the device is tilted all the way to the left, 0 means the device is not tilted at all on the x-axis and 1 means the device is tilted all the way to the right. The value of y will range from -1 to 1, where -1 means the device is tilted all the way forward, 0 means the device is not tilted at all on the y-axis and 1 means the device is tilted all the way back. For both of these the value will usually be somewhere between -1 and 1, for instance x=0.5 would mean that the device is tilted half way to the right.

\[
\text{var} \ \text{tilt} := \text{senses} \rightarrow \text{acceleration quick} \rightarrow \text{scale(360)}
\]

\[
\text{ship} \rightarrow \text{set speed} \ x(\text{ship} \rightarrow \text{speed} \ x + \text{tilt} \rightarrow \text{x})
\]

In the example above I have applied ->scale(360) which basically multiples the x, y and z properties of the vector3 by 360. So instead of getting a value between -1 to 1 it becomes a value between -360
and 360. In this example I am adjusting the x axis speed of a ship sprite by adding the x value accelerometer reading to it, which has the effect of moving the ship sprite to the left as you tilt your device to the left and moving it to the right as you tilt right. Adjusting the scale value will adjust how fast the sprite moves. You could also do this on the y axis using the y reading from the accelerometer, to allow you to move a sprite in any direction just by tilting. If you are using a pc without an accelerometer, most HTML5 browsers allow you to simulate the tilt effect by moving your mouse left, right, up or down.

2D Animation
Most video games make use of some form of 2D animation; one method is to use a sprite sheet, some examples of which are shown below.

TouchDevelop sprites have a “set clip” command which allows you to select just part of a sprite to display; this can be used to create an animation sequence from a sprite sheet. Luckily my friends at Microsoft Research have written a library for doing animations which uses “set clip”, but makes it easy to do and allows you to use multiple sprite sheets.

To use this library first search for “sprite sheet” and add it into libraries in the script you want to use it in.

Then you must initialise it in your main action, as shown below.
You must add the sprite sheet you want to use into the art section of your script. Sprite sheets should generally be png files so that the background can be set transparent. To initialise a sprite sheet and make it ready for use, use the library and issue the set sheet action as shown below. This command gets “cuteman sprite sheet” from art and gives it the identifier Cuteman, it also sets the number of rows to 2 and columns to 8. The example below is for the sprite sheet shown underneath.

We can use the sprite sheet library to create sprites from the sheet, by issuing the command shown below and selecting the row and column numbers of the sprite we want. The command below creates a sprite containing the sprite at row 2 and column 1 from the sprite sheet shown above.

If we want to create an animation which loops through each frame of animation in a sprite sheet, we need to first initialise the sprite sheet in the same way as I demonstrated above, then we issue the “create full animation” action to create an animated sprite. In the example below we have set the duration it takes to play through the animation once to 1 second, if you increase this value the animation will play slower, decrease it and the animation will play faster. The last parameter is how many times the animation should be repeated, if we set it to 3, for instance, it would play through the animation 3 times, in the example below it has been set to -1, which indicates that the animation should repeat indefinitely.
Lastly to make the animation actually update and move through the different frames you must add the line shown below into the gameloop.

Once you have created an animated sprite, you can do all the things to it that you would do to a normal sprite, such as setting friction, speed, position etc.

**Activity 4a – Creating a basic accelerometer game**

- In TouchDevelop create a new “physics game starter” script (name it whatever you want).
- Add a ball picture into the art section of the new script (I named mine “Soccer ball”).
- Make the following changes and additions to the main action, which will set the gravity to 0, set the background to green, set friction at 0.03 and create a boundary around the game board.

Add the lines below into main, to create a global variable for the ball and set its height.
In gameloop add the following 3 lines of code. The first line reads the accelerometer and scales the result up by a multiple of 60. The second line adds the x and y tilt values from the accelerometer to the current speed of the ball, which has the effect of allowing the player to control the balls movement by tilting his or her device. The last 2 lines sets the angular speed (spin) of the ball based on the speed of the ball; this is just to give the effect of the ball spinning faster as it moves faster. We do this by creating a Vector variable based on the ball speed & using its length to set the speed the ball spins at, because the vector length is the speed that the ball is travelling at, no matter what direction it is going in.

```plaintext
event gameloop ()
  board → evolve
  board → update on wall
  var tilt := senses → acceleration quick → scale(60)
  ball → set speed(ball → speed x + tilt → x, ball → speed y + tilt → y)
  var ball velocity := math → create vector3(ball → speed x, ball → speed y, 0)
  ball → set angular speed(ball velocity → length)
```

Run the script and if possible test it on a device with an accelerometer. If you are using a pc simulate the accelerometer using the mouse.

Experiment with adding some obstacles in. To create your first obstacle, add the following line into the MAIN action.

```plaintext
  board → create obstacle(300, 200, 0, 200, 1)
```

Test the script with at least one obstacle and see if it works.

**Challenges**

Making use of what you have learnt so far, try and complete the following challenges in your game.

- Add multiple obstacles to create a maze to navigate the ball through.
- Set the position of the ball to be at the start of the maze.
- Add an ellipse sprite as a hole in the maze for the ball to fall into to complete the level.
- Add code into the gameloop to check when the ball touches the hole and display a message to say the level is complete. You can use the wall->prompt command as shown above to do display a message.
Activity 4b – Creating a scrolling shooter game

- In TouchDevelop create a new “physics game starter” script (name it whatever you want). If you remember games like Defender, R-Type, Xenon, 1942; this is the type of game we are going to create.

- First let’s select our scrolling background. Start by loading a background image into ART. I suggest you choose one of the following 2 pictures (from search online art pictures) as they are both almost seamless so you won’t be able to easily see the join when we repeat the images as we scroll them, plus they both work well horizontally and vertically.

- Add the following changes and additions to the code (marked in red) into the main action to create a vertically scrolling background. The code creates obstacle up the left and right sides of the game board to stop sprites leaving either side. It adds two sprites called back1 and back2 which you must set as global. They each hold the same background picture and it positions the 2nd background immediately above the 1st background by subtracting the height of the 1st background from its y position to give the y position for the 2nd background. It sets both backgrounds to have the same positive speed vertically on y, which basically moves the two backgrounds downwards to give the impression that we are flying upwards.

```action main()```

```wall → set background(colors → black)```
```wall → set foreground(colors → white)```
```board := media → create portrait board(480, 800)```
```board → set gravity(0, 0)```
```board → create obstacle(0, 0, 0, board → height, 0.5)```
```board → create obstacle(board → width, 0, 0, board → height, 0.5)```
```board → post to wall```

```Back1 := board → create picture(space background)```
```Back1 → set height(board → height)```
```Back2 := board → create picture(space background)```
```Back2 → set height(Back1 → height)```
```Back2 → set y(Back1 → y - Back1 → height)```
```var scroll speed := 100```
```Back1 → set speed y(scroll speed)```
```Back2 → set speed y(scroll speed)```
• If instead you want to create a horizontally scrolling game, use the following code which has minor changes. The minor differences in the code are marked in blue. The main differences are that we create a landscape rather than portrait board. We create obstacles along the top and bottom of the board, instead of the left and right sides. We set the width of the backgrounds to the width of the game board, rather than its height and we set the second background position on the x rather than the y and use the width of the 1st board, rather than its height to calculate its position. Also we scroll the backgrounds by setting the speed on the x axis, rather than the y axis. The scroll speed is a negative value as we are moving the backgrounds from right to left.

```action
main ()

wall -> set background(colors -> black)
wall -> set foreground(colors -> white)

board := media -> create landscape board(800, 480)
board -> set gravity(0, 0)

board -> create obstacle(0, 0, board -> width, 0, 0.5)
board -> create obstacle(0, board -> height, board -> width, 0, 0.5)

board -> post to wall

Back1 := board -> create picture(Space background)
Back1 -> set width(board -> width)
Back2 := board -> create picture(Space background)
Back2 -> set width(Back1 -> width)

Back2 -> set x(Back1 -> x + Back1 -> width)

var scroll speed := -100
Back1 -> set speed x(scroll speed)
Back2 -> set speed x(scroll speed)
```

• Run the script you have created and you should have a background that scrolls, but after the 2 backgrounds have left the screen, they disappear and it stops scrolling. What we need to do is that when either background leaves the screen, we need to put it back at the top or to the right, of the other background to give the impression of a never ending scrolling background.
Add the following action and call it from the gameloop event. The first version swaps the backgrounds around for a vertically scrolling game and the second version swaps the backgrounds around for a horizontally scrolling game (only add the one you need). The code basically just checks if either background has left the screen and if so it sets a new position by adding the width or height of the other background to its position.

```plaintext
private action check backgrounds ()
    if Back1 -> y - Back1 -> height / 2 > board -> height then
        Back1 -> set y(Back2 -> y - Back2 -> height)
    else do nothing
    if Back2 -> y - Back2 -> height / 2 > board -> height then
        Back2 -> set y(Back1 -> y - Back1 -> height)
    else do nothing
```

Remember that the position of a sprite is the middle of it, which is why I am subtracting half of the height to calculate the top of the background in the first example and adding half of the width in the second example to calculate the right hand side of the background. Don’t forget to call the action as shown below from the gameloop.

```plaintext
event gameloop ()
    board -> evolve
    board -> update on wall
    check backgrounds
```

Run the script and make sure the backgrounds scroll indefinitely. Feel free to adjust the scroll speed as you see fit.
Next let’s add the user controlled spaceship to the game. Create an art resource called spaceship and choose or upload a suitable image for it. The one I choose was the third image shown opposite, but feel free to choose your own graphic.

Add the code below after the previous code in the main action, to initialise a sprite variable for the ship. The left hand side code is for the vertical scrolling version and the right hand side is for the horizontally scrolling version. The only difference is the starting positions of the ship and on the horizontal version we rotate the ship sprite 90 degrees so it is facing to the right. This is only necessary if you are using an image where the ship faces upwards, if your image is already facing to the right you won’t need to do this.

Add the following code to the gameloop event which will allow you to move the spaceship using tilt. The code on the left hand side is again for the vertical version and the right hand side is for the horizontal version. Feel free to adjust the scale as need be. In the vertical version we are allowing the player to move the ship left and right, whereas in the horizontal version we are allowing the player to move the ship up and down. You could put both set speed lines in and allow the player to move the ship in all directions.

Run the script and test that it works correctly.

Next we will add multiple alien ships using a sprite set, but first we need to add an art resource for the aliens. The one I choose is shown opposite.

Declare a sprite set for the aliens in main as shown below.
TouchDevelop Curriculum

- Add the following code into an action to spawn alien enemies (left is vertical and right is horizontal). It creates a sprite using an art resource that I’ve just named alien. The methods set the width of the sprite and randomly generate a position for the sprite off screen at the top or to the right, depending on which version of the game you are creating. It also randomly generates a speed for the enemy ship, on either the y axis or the x axis depending again on whether it’s horizontal or vertical. Lastly it adds the sprite into the enemies sprite set.

```plaintext
private action spawn enemy ()

var sprite := □ board → create picture(alien)
        sprite → set width(50)
        sprite → set pos(math → random(□ board → width - sprite → width) +
                     sprite → width / 2, 0 - 300 - math → random(200))
        sprite → set speed y(200 + math → random(200))
        □ enemies ← add(sprite)
```

- Next add the action shown opposite which will use the action above to spawn multiple aliens. We will use this action to spawn extra waves of aliens and also when the game restarts. The first thing we do in it, is to delete any existing alien sprites from the game board using a FOR EACH loop and the delete command. Then we issue a clear command for the enemies sprite set to remove all sprites from it. Lastly we use a FOR loop to spawn multiple aliens by repeatedly calling the “spawn enemy” action. The number of aliens we want is passed into the action as a parameter called “number to spawn”.

```plaintext
private action spawn alien wave ()

number to spawn : Number

do
    for each e in □ enemies
        where true
            do
                e → delete
                □ enemies ← clear
            for 0 ≤ i < number to spawn do
                □ spawn enemy
```

- Add one more action called “Reset game”, in which we will initialise everything that needs reset at the beginning of each new game. So we declare global variables for score and lives and assign initial values. We will also set the initial number of aliens and call the “spawn alien wave” action to create the sprite set. Make sure you call this new action from the main action as shown opposite.

```plaintext
private action Reset game ()

□ score := 0
□ lives := 5
□ number of aliens := 10
□ spawn alien wave(□ number of aliens)
□ board → post to wall

▷ Reset game
```
• Run this script and the aliens should appear and move. However you will find that they don’t re-spawn after they leave the screen.

• So let’s add another action called “check aliens” which will use a FOR EACH loop to go through each alien and check if it leaves the screen. If it does leave the screen we will remove it from the sprite set, delete it from the game board and spawn a new alien in its place by calling “spawn enemy”. If it doesn’t leave the screen we check to see if it has hit the ship, in this case we remove it from the sprite set and delete it from the board, but we do not re-spawn it, instead we subtract a live from the player. The code shown below is for the vertical version. Make sure you call this action from inside gameloop.

```plaintext
private action check aliens ()
    for each e in enemies
        where true
        do
            if e -> y - e -> height > board -> height then
                enemies -> remove(e)
                'remove' returns a 'Boolean'; insert 'post to wall' if you want to stop it
                e -> delete
                ▷ spawn enemy
            else if e -> overlaps with(ship) then
                enemies -> remove(e)
                'remove' returns a 'Boolean'; insert 'post to wall' if you want to stop it
                e -> delete
                □ lives := □ lives - 1
```

If you are creating the horizontal version then just change the initial IF condition to the one shown below, to check for them leaving the left hand side rather than the bottom of the screen, the rest of the code is the same as shown above.

```plaintext
if e -> x + e -> width < 0 then
```

• Run the script again and make sure that the aliens re-spawn when they leave the screen.

• Next we want to add the ability to shoot at the aliens and for them to shoot back. In this game we are just going to add one rocket for the player and one for the aliens, but you could easily add multiple rockets using a sprite set. Add at least 2 art resources to your game for the rockets for both sides to fire.
• Add the following lines of code into the main action to create sprites for each rocket. The reason we are hiding the sprites, is because we don’t want them displayed before they have been fired. Depending on the art you chose, you may need to use the “set angle” command to make sure the rockets are pointing in the correct directions. I named my art resources “Rocket” and “Enemy fire” so make sure to choose the correct name for the art you added.

```
□ rocket := □ board -> create picture(Rocket)
□ rocket -> set width(20)
□ rocket -> hide
□ Enemy rocket := □ board -> create picture(Enemy fire)
□ Enemy rocket -> set width(10)
□ Enemy rocket -> hide
```

• Next let’s add an action to shoot the rocket sprites, which can be used for both sides. We pass into this action the sprite we want to shoot, a speed for the rocket and x and y co-ordinates for the position to fire the sprite from. The IF condition checks to see that the sprite is NOT already visible on the screen before trying to shoot it. It then sets the sprite to visible by using the SHOW command, sets the initial position of the sprite and assigns speed to it. If you are making the horizontally scrolling version change the line which sets the speed on the x axis rather than the y as shown below.

```
private action shoot (  
  sprite : Sprite,
  speed : Number,
  X : Number,
  Y : Number)
  do
  if not sprite -> is visible then
    sprite -> show
    sprite -> set pos(X, Y)
    sprite -> set speed y(speed)
  else do nothing
```

• We need to have a way of firing the rocket from the player’s spaceship, let’s make it so that when the player touches the screen that the rocket fires (you can simulate this on a pc by clicking the LMB). We can do this by checking board->touched in an IF statement as shown below and then we call the shoot action passing in the rocket sprite, a speed of -450 (change it to 450 for horizontal) and the x & y positions of the ship so the rocket will fire from the current position of the player’s spaceship. The second IF statement checks for the rocket leaving the screen at the top and hides it when it does so it can be re-fired. Add this code into gameloop.

```
if □ board -> touched then
  □ shoot(□ rocket, -450, □ ship -> x, □ ship -> y)
else do nothing
if □ rocket -> y + □ rocket -> height < 0 then
  □ rocket -> hide
```
If you are creating the horizontal version change the IF statement to check for the rocket leaving the right-hand side of the screen as shown below.

```
if □ rocket ← x + □ rocket ← width > □ board ← width then
□ rocket ← hide
```

Run the script and check that you can fire the rocket from the spaceship and when it leaves the screen that you can fire it again.

Go into the “check aliens” action and add in the following IF statement against the ELSE part of the last IF statement. This code checks if an alien is hit by a rocket which is visible on the game board. If this is the case it then hides the rocket, removes the alien from the sprite set, deletes the alien sprite from the game board and adds 5 points to the score for destroying an alien ship.

```
else if e ← overlaps with(□ rocket) and □ rocket ← is visible then
□ rocket ← hide
□ enemies ← remove(e)
% 'remove' returns a 'Boolean'; insert 'post to wall' if you want to display it
e ← delete
□ score := □ score + 5
```

Run the script now and check that the player’s rocket can destroy the alien ships.

Before we go much further it would be useful if score and lives were displayed on the screen. Go into the main action and declare a text sprite as shown below. I am setting the width of the sprite to the full width of the game board and then setting the position of the sprite to be in the middle of the screen on the x-axis, by setting the x position to the board width divided by 2.

```
□ score board := □ board ← create text(□ board ← width, 20, 40, "")
□ score board ← set pos(□ board ← width / 2, 20)
```

Next we need to update the score board text sprite in the gameloop so that it displays the current score and lives. Do this by adding the line below into gameloop.

```
□ score board ← set text("Score " || □ score || " Lives " || □ lives)
```

Run the script and make sure score and lives appear. Shoot some aliens and check that score is assigned and run into some aliens to make sure that you lose a life.
Next we are going to add code into the “check aliens” action which will fire a rocket from the alien ships when they are in line with the player’s spaceship. So add the code below at the end of that action.

```plaintext
else if e → x - e → width / 2 < □ ship → x and e → x + e → width / 2 > □ ship → x and e → y > 0 then
  ▶ shoot(□ Enemy rocket, 500, e → x, e → y)
```

Make sure that the IF above is being added against the previous ELSE statement, this is because if any of the previous IF conditions are true then the alien ship will not be in a position to fire a rocket, since it will either have left the screen or been destroyed. Due to this statement being added into a FOR EACH loop which loops through all the aliens, this code will check if any of the aliens are in line with your spaceship and shoot a rocket if it is. It does this by checking if the left hand side of the alien is less than the centre position of the spaceship and checking that the right hand side of the alien is greater than the centre position of the spaceship, which means the alien is directly above the spaceship.

If you are creating the horizontally scrolling version of the game, use the code below instead. Notice the minor differences, which check if the aliens are in line with the spaceship horizontally rather than vertically.

```plaintext
else if e → y - e → height / 2 < □ ship → y and e → y + e → height / 2 > □ ship → y and e → x < □ board → width then
  ▶ shoot(□ Enemy laser, - 500, e → x, e → y)
```

We also need to check if the alien rocket leaves the screen and if the alien rocket hits the spaceship. So add the following code into either the gameloop or the “check alien” action, either place is fine (I went for putting it into the “check alien” action). If you are adding it into the “check alien” action be careful that you don’t put it inside the FOR EACH loop, as that would be very wasteful. Use the diagonal arrow button to make sure the new line you add for the code is as far to the left as you can make it, so that it is not contained within a previous condition.

```plaintext
if □ Enemy rocket → y - □ Enemy rocket → height > □ board → height then
  □ Enemy rocket → hide

else if □ Enemy rocket → is visible and □ Enemy rocket → overlaps with □ (□ ship) then
  □ Enemy rocket → hide
  □ lives := □ lives - 1
else do nothing
```
• Go through the 2 IF statements you just added and make sure you understand fully what they do, check with your tutor if you don’t. An important part of learning to code, is being able to read code, much like when you are learning a language like English or French, it helps to be able to read it; being able to read code helps you write it.

• If you are making the horizontal version of the game, then change the first IF statement above to check for the rocket leaving the left hand side of the screen rather than the bottom of the screen; as shown below.

```plaintext
if [Enemy rocket] x + [Enemy rocket] width < 0 then
    [Enemy rocket] hide
```

• Add the following code into the gameloop or into the “check aliens” action, either place will work. This code checks if all the sprites have been removed from the enemies sprite set, and if the number of sprites left is 0, then we use the wall->prompt command to inform the player that a new wave of aliens is about to begin. We also increase the “number of aliens” variable by 5 for the next wave and then call the action to spawn the next wave.

```plaintext
if [enemies] count = 0 then
    wall -> prompt("New wave about to begin")
    [number of aliens] := [number of aliens] + 5
    spawn alien wave([number of aliens])
else do nothing
```

• The game is almost complete; however we will also need to have an action to call whenever the game is over, to deal with it. So add in the action shown below. This action prompts the player to tell them the game is over, it then clears the wall and posts the leaderboard to it. It then pauses for 5 seconds before prompting the player to see if they want to play again or not.

```plaintext
private action Endgame ()
    wall -> prompt("Game Over")
    wall -> clear
    bazaar -> post leaderboard score(score)
    bazaar -> post leaderboard to wall
    time -> sleep(5)
    if wall -> ask boolean("Play again?", "") then
        > Reset game
    else
        time -> stop
```
Lastly we just need to check for lives reaching zero and call the action we just added if it does. So add in the code below into the gameloop.

```plaintext
if lives <= 0 then
    Endgame
```

Run the script and check that the game is now fully working. Debug any errors that you find. The game should look something like the screenshot opposite.

To finish the game off let’s add some 2D animated explosions, for when the player destroys an alien ship. Add an art resource by searching the online art pictures for the 2D sprite sheet shown below; name it “explosion sprite”.

Add the library called “sprite sheet” to your game script.

Go into MAIN and add the following 2 lines at the end. The first line just initialises a sprite sheet for your game board. The second line loads in the explosion sprite into a sheet that I have called “explosion”. The numbers being passed in at the end are the number of rows and columns in the sprite sheet, it is vital to get this correct. So if you choose a different animation then make sure you change these numbers to match.
• In gameloop add the following line to evolve (update) the sprite sheet as and when the animation is used.

```
    event gameloop()
    sprite sheet → evolve
    score board → set text(""
```

• Lastly go into the “check aliens” action and add the last 2 lines shown below, inside the IF statement that checks for aliens being hit by the player’s rocket. The first line creates a sprite containing the full animation, the first number being passed in sets the length the animation takes to play through once to 1 second and the second number passed in is the number of times to play the animation, which is this case is once. The second line of code just sets the position of the animation sprite to the last position of the rocket, so that the explosion will appear where the rocket hits the alien ship.

```
else if e → overlaps with(rocket) and rocket → is visible then
    rocket → hide
    enemies → remove(e)
    'remove' returns a 'Boolean'; insert 'post to wall' if you want to display it
    e → delete
    score := score + 5
    var sprite := sprite sheet → create full animation("explosion", 1, 1)
    sprite → set pos(rocket → x, rocket → y)
```

• Run the game and you should have an animated explosion each time you destroy an alien with a rocket.

**Challenges**

Making use of what you have learnt so far, try and complete the following challenges in your game.

• Add explosions when the player’s spaceship is hit by alien rockets, either using the same animation or a different one.

• Add explosions when the player’s spaceship collides with an alien ship.

• Add sound effects for when rockets are fired and when explosions occur.

• If you have a device on which you can test the tilt function, try it out and alter the tilt scale until you are happy with how the spaceship moves.

• Go through the code and add comments to explain the main parts.

• Publish your script.
Activity 4c – Creating a Space Invaders clone

In our last activity for this section, we are going to re-create one of the all-time classic arcade games, space invaders.

- Create a new “physics game starter” script and name it whatever you want, I called my version TouchInvaders, but feel free to be more creative with the name.
- Add a picture for the game background and call it “Space Background”. I would suggest the one shown below.

- Add a picture for the player controlled base / tank and name it “player tank”.

- Go into the main action and make the following changes and additions to the code, which will setup the game board and declare a global sprite variable for the player’s base. It also creates a global number variable for the base speed, which we will use later on. It is optional whether you want to add the comments or not, although it wouldn’t hurt.

```
ACTION main ()

  wall ← set background(colors ← black)
  wall ← set foreground(colors ← white)

  Setup the game board
  □ board := media ← create landscape board(800, 480)
  □ board ← set gravity(0, 0)
  □ board ← set background picture(☆ Space background)
  □ board ← post to wall

  Create game sprites
  □ base := □ board ← create picture(☆ player tank)
  □ base ← set width(100)
  □ base ← set friction(0.05)
  □ base speed := 30
```
Create a new action called resetgame as shown below. Inside it we are creating 3 global numbers variables to hold the values for lives, score and the number of alien waves. We are also setting the initial position of the player’s base. Then we prompt the player for a YES / NO response as to whether they want to use tilt screen controls for the base, the result of this prompt is a Boolean TRUE or FALSE which will be stored into the global Boolean variable “Tilt on”. Lastly we re-post the board to the wall (this is necessary later on when we will allow the player to play again without quitting out).

```csharp
private action resetgame ()

lives := 3
score := 0
wave := 1
base ← set pos(board ← width / 2, 370)
Tilt on := wall ← ask boolean("Tilt screen controls on?", "Accelerometer")
board ← post to wall
```

Call this new action from the main action and remove the line “board→post to wall” from the main action.

Add the following code into the gameloop, which will move the player’s base left or right if the player touches the game board. If the player touches to the left of the base it subtracts the base speed variable from the current base speed x value, which will alter the direction of the base to the left. If the player hasn’t touched to the left (therefore he must have touched to the right) it adds this value instead of subtracting to alter the direction to the right.

```csharp
if board ← touched then
	if board ← touch current ← x < base ← x then
		base ← set speed x(base ← speed x - base speed)
	else
		base ← set speed x(base ← speed x + base speed)
```

Add the following code to add tilt screen controls to the base, this is the same method we used in the previous activity. You may alter the scale value of 100 to change the movement speed. Note that the code to move the base only runs if the “Tilt on” variable is set to true, which is set in the resetgame action we created previously.

```csharp
if Tilt on then
	var tilt := senses ← acceleration quick ← scale(100)
	base ← set speed x(base ← speed x + tilt ← x)
else
do nothing
```
• Extract both bits of code above which move the player base to an action called “move base” and make sure it is called from the gameloop. "move base"

• Run the script and checks that the controls work. You will probably find that the base can move outside of the limits of the screen. Let’s fix this by adding in the following IF statements to stop it from going out by reversing the speed when it hits the sides. The reason I didn’t use obstacles at either side, as we did in the previous activity, is because it would cause problems with the aliens who are meant to move as one when any one of them hits the sides.

```plaintext
if base → x < width / 2 then
    base → set speed x(math → abs(base → speed x / 2))
else
do nothing

if base → x > board → width - base → width / 2 then
    base → set speed x(- math → abs(base → speed x / 2))
else
do nothing
```

• Extract the IF statements above into a separate action called “check limits” and call it from the gameloop. "check limits"

• Next let’s add in our aliens. We need to add multiple rows of aliens, a bit like we did with the bricks in our breakout clone. First of all declare a global sprite set in main, as shown below.

```plaintext
aliens := board → create sprite set
```

• Next add an alien graphic picture into art resources and name it alien.

• Create a new action naming it addinvaders; it will be used to add and position the alien sprites in rows and columns. Add the parameters opposite to the action. The reason that we are passing the sprite set and game board in, which might seem a bit unnecessary, is because it means there are no direct calls to global variables from this action, which means it could be put into a library and used as a template for building space invaders or similar games which require rows of sprites. Actions that do not directly access global variables should be considered good practice and something you should do whenever possible.
• Add the following code into the new addinvaders action we just created. It first clears all sprites from the sprite set and then adds the correct numbers of aliens, by adding one alien each time the code inside the 2 FOR loops (rows and columns) repeat. It sets the size (width) and initial horizontal speed of each sprite. It uses the variables we passed in along with the row and column numbers from the 2 FOR loops to space out the positions of the sprites. Then finally it adds the new sprite to the sprite set.

```
do
  sprite set ← clear
  This creates rows and columns of alien invaders
  for 0 ≤ i < rows do
    for 0 ≤ j < columns do
      var alien sprite := board ← create picture(alien picture)
      alien sprite ← set width(alien width)
      alien sprite ← set speed x(alienspeed)
      alien sprite ← set x(firstx + (j * (alien sprite ← width + xspacing)))
      alien sprite ← set y(firsty + (i * (alien sprite ← height + yspacing)))
      sprite set ← add(alien sprite)
```

• Go into the resetgame action and add the following line of code, which calls the new addinvaders action, passing in the correct parameters to set-up 4 rows with 10 columns of alien sprites. Feel free to change the values of any of the parameters to see what happens.

```
addinvaders(50, 50, 4, 10, 5, 50, □ aliens, ⭐ alien, □ board, 40)
```

You will find that as you enter or change each parameter that a box appears below, with the name and variable type of the current parameter you are entering highlighted, this is so that you know what the parameter is for. For instance in the example below I am entering the value for firsty which is a number, this is the y position value for the first alien in the set and changing this effects the positions of all the aliens in the sprite set.

```
addinvaders(50, 50, 4, 10, 5, 50, □ aliens, ⭐ alien, □ board, 40)
```

```
addinvaders(firstx : Number, firsty : Number, rows : Number, columns : Number, xspacing : Number, yspacing : Number, alienspeed : Number, sprite)
```
Add the following action which checks for the aliens hitting either side of the screen and then moves them all down a little when they do. It also changes the horizontal speed and therefore the direction that they are all moving in when this occurs. It is a little bit tricky to rename the local sprite variable inside a “for each” loop, it will default to sprite. However if you then use sprite within the “for each” loop you can then select it and select the RENAME button to change it. You will need to do this to set the variable names of alien and alien2 for the local sprite variables inside the 2 for each loops shown below.

```action move aliens (sprite set : Sprite Set, pixels to jump down : Number, left limit : Number, right limit : Number)
do
    This moves the aliens left and right and then down when they hit either side.
    var hitsides := 0
    for each alien in sprite set
        where true
        do
            if alien → x < left limit + alien → width / 2 then
                hitsides := 1
            else do nothing
            if alien → x > right limit - alien → width / 2 then
                hitsides := - 1
            else do nothing
        if hitsides ≠ 0 then
            for each alien2 in sprite set
                where true
                do
                    alien2 → set y(alien2 → y + pixels to jump down)
                    alien2 → set speed x(hitsides * math → abs(alien2 → speed x))
            else do nothing
    end for each
end do
```

Now we just need to call the above method from the gameloop to make the aliens move. The number 5 we are passing in is how many pixels to move the alien sprites down by when it hits the sides, while the last 2 parameters are the left and right limits between which the aliens move, which I’ve set to the limits of the game board.

```move aliens(alien, 5, 0, board → width)```
Run the script and you should now have aliens moving left to right, down & then right to left and so on.

Next let’s add in rows and columns of shields in a very similar fashion to the way we created the aliens. To do this we first need to create a global sprite set for the shields, do this in the MAIN action as shown below. I would suggest you add this line after the line where you declared the alien sprite set. Remember the order that you declare sprites in, is also the order they are drawn in, so if you declare it before the background then the shields would be hidden by the background being drawn on top of it.

```
shields := board -> create sprite set
```

We could probably have used the addinvaders action to do this, however we are going to do it a slightly different way and create layers of overlapping filled rectangles to create the shields. This means that as the top layers are destroyed the aliens still have to shoot through another couple of layers before they can shoot at you. I am using the set color command and using numeric RGB values to assign slightly different shades to each layer. However apart from the way I am creating the shields, the rest of the code is very similar. So first of all add in the following action as shown below.

```plaintext
private action addsShields (  
  firstx : Number,  
  firsty : Number,  
  shieldwidth : Number,  
  shieldheight : Number,  
  rows : Number,  
  columns : Number,  
  spriteset : Sprite Set,  
  xspacing : Number,  
  yspacing : Number,  
  board : Board)
  do
    spriteset -> clear
    for 0 ≤ i < rows do
      for 0 ≤ j < columns do
        var sprite := board -> create rectangle(shieldwidth, shieldheight)
        sprite -> set pos(firstx + (j * xspacing), firsty + (i * yspacing))
        sprite -> set color(colors -> from rgb(1, 1 - (i / 10), 0))
        spriteset -> add(sprite)
  
```
• Call this action from the resetgame action as shown below. The shields should now be visible if you run the script.

```csharp
addshields(100, board → height - 180, 50, 10, 3, 9, shields, 75, 5, board)
```

• Add in the following action to delete all alien & shield sprites from the board, this is so that when we start a new game sprites aren’t left behind on the board from the previous game.

```csharp
private action clear sprites ()
    for each sprite in shields
        where true
            do
                sprite → delete
    for each sprite2 in aliens
        where true
            do
                sprite2 → delete
```

• Call it from the resetgame action. It is important that you add it before you call addinvaders and addshields, as shown below.

```csharp
addinvaders(50, 50, 4, 10, 5, 5, 50, aliens, star, alien, board, 40)
addshields(100, board → height - 180, 50, 10, 3, 9, shields, 75, 5, board)
```

• Go into art resources and add in the rest of the graphics that we need for the game. We need a rocket or laser graphic for both the aliens & the player to fire. So add graphics into art resources for this; name one alienrocket and one rocket. You can search online media or upload your own graphics. We also need a graphic for a shoot button, so find one and name it shootbutton in art resources.

• Go into main and declare the following global variables, as shown below, for the rockets and for text sprites to display lives and score.

```csharp
alien rocket := board → create picture(star, alienrocket)
player rocket := board → create picture(star, rocket)
player rocket → set width(50)
shoot button := board → create picture(star, shootbutton)
shoot button → set pos(board → width - 120 - shoot button → width / 2, board → height - shoot button → height / 2)
livesboard := board → create text(100, 20, 30, "")
scoreboard := board → create text(100, 20, 30, "")
livesboard → set pos(740, board → height - 35)
scoreboard → set pos(80, board → height - 35)
```
• Go into the resetgame action and add in the 2 lines opposite to hide both rockets at the start of the game.

• Add in the 2 lines following into the gameloop to update the text sprites with the current score and lives.

  □ scoreboard ← set text("Score " || □ score)
  □ livesboard ← set text("Lives " || □ lives)

• Add in the 2 libraries shown below to your game script.

• Create a new action and call it shoot, this action will be used to shoot both the alien and player rockets. We pass in a velocity for the rocket, the sprite we are shooting from and the sprite we are going to shoot. The action then sets the position of the rocket based on the position of the sprite we are firing from, it gives it a vertical speed and then it issues the show command on the rocket sprite to make it visible on the screen.

  action shoot ( y velocity : Number, sprite shooting from : Sprite, sprite to shoot : Sprite)
  do
  | sprite to shoot ← set pos(sprite shooting from → x, sprite shooting from → y)
  | sprite to shoot ← set speed y(y velocity)
  | sprite to shoot ← show

• Add the IF statement marked in red on the following page, into the gameloop action so that the main game code only runs when the player has more than zero lives. Make sure you indent the 3 lines of code below the IF statement, as shown in the picture, as those 3 lines call the actions which move both the player sprite and the alien sprites and we only want them to move when the game is running.
• Make the following changes, marked in red below, to the “move base” action, which check for the user touching and/or holding down the shoot button on the screen. When this occurs we fire the player’s rocket if it is not already visible (which we are taking to mean that it is not currently being fired). The second bit of code marked in red, checks for the rocket leaving the top of the screen and sets it to be hidden if it does, so we can re-fire it if the player presses shoot again. The middle bit of code, marked in blue, doesn’t need entered again, it just needs moved so that it only runs against the ELSE from the new IF statement. This is so that the game only moves the player’s base when they touch the screen if they aren’t pressing the shoot button.

```plaintext
event gameloop ()
| sprite sheet -> evolve
| scoreboard -> set text("Score " || score)
| livesboard -> set text("Lives " || lives)
| board -> evolve
| board -> update on wall

Check if the game is running

if lives > 0 then
| move base
| check limits
| move aliens(aliens, 5, 0, board -> width)
```

```plaintext
private action move base ()

if game events -> touch_hold(board, shoot button) then
| if not player rocket -> is visible then
| | shoot(-300, base, player rocket)
| else do nothing

else if board -> touched then
| if board -> touch current -> x < base -> x then
| | base -> set speed x(base -> speed x - base speed)
| else
| | base -> set speed x(base -> speed x + base speed)

else do nothing

if Tilt on then
| var tilt := senses -> acceleration quick -> scale(100)
| base -> set speed x(base -> speed x + tilt -> x)
else do nothing

if player rocket -> y < - player rocket -> height then
| player rocket -> hide
else do nothing
```
• Try running the script and make sure you can now move and shoot. However you will find that you can’t destroy the aliens. If you find that the player’s base moves towards the shoot button when you press it, have a look at the code on the previous page again and make sure you have indented it properly.

• We need to add a few actions for events that will occur in the game, such as the base being hit by a rocket, aliens being destroyed and the shields being hit. Let’s start by adding an action called “Base Hit” for when the player’s base is hit by the alien rocket. This action basically just subtracts a life and hides the alien rocket. This would be a good place to start an explosion animation and/or add an explosion sound effect (this may be in the challenges later).

  ```c
  private action Base Hit ()
  | lives := lives - 1
  | alien rocket → hide
  ```

• Next let’s add the following action which we will call when a shield has been destroyed. This action just hides the alien rocket, removes the shield from its sprite set and then deletes it from the game board. This is another good place for an explosion sound and animation.

  ```c
  private action shield hit (sprite : Sprite)
  do
  | alien rocket → hide
  | shields → remove(sprite)
  | sprite → delete
  ```

• Add the following action which we will call when an alien has been hit by a rocket. This action hides the rocket, removes the alien sprite from its sprite set, deletes the alien sprite from the game board and finally adds 5 onto the score. Once again this is another great place for an explosion sound and animation.

  ```c
  private action kill alien (alien sprite : Sprite)
  do
  | player rocket → hide
  | aliens → remove(alien sprite)
  | alien sprite → delete
  | score := score + 5
  ```
• Add the following action to handle collisions between the alien rockets and the shields, the alien rockets and the player’s base, as well between the player’s rocket and the aliens.

```csharp
private action check for collisions ()
    Check if alien rocket hits user controlled base
    if □ alien rocket → overlaps with(□ base) and □ alien rocket → is visible
        ▷ base hit
    else do nothing
    Check if the alien rocket has left the game board & hide it if it has
    if □ alien rocket → y - □ alien rocket → height / 2 > □ board → height
        □ alien rocket → hide
    else do nothing
    Check if alien rocket hits the shields and remove the shield if it does
    for each sprite in □ shields
        where true
        do
            if □ alien rocket → is visible and □ alien rocket → overlaps with(sprite)
                ▷ shield hit(sprite)
            else do nothing
    for each sprite2 in □ aliens
        where true
        do
            If alien rocket is NOT being fired & the alien ship is above the base then FIRE
            if not □ alien rocket → is visible and sprite2 → x - sprite2 → width / 2 < □ base → x and sprite2 → x + sprite2 → width / 2 > □ base → x
                ▷ shoot(300, sprite2, □ alien rocket)
            else do nothing
            Check if alien is hit by the player’s rocket and destroy it if it has
            if □ player rocket → is visible and □ player rocket → overlaps with (sprite2) then
                ▷ kill alien(sprite2)
            else do nothing
            If all the aliens have been destroyed then spawn a new wave
            if □ aliens → count = 0 then
                ▷ addinvaders(50, 50, 4, 10, 5, 5, 50 + (□ wave * 10), □ aliens, □ alien, □ board, 40)
                □ wave := □ wave + 1
            else do nothing
```
• Go through this new action and make sure you understand what each bit does, the comments should help explain what each main part does. Basically though, it checks for collisions between sprites and calls the respective action when they do collide. For instance if the rocket hits an alien, it calls “kill alien” and passes in the sprite of the alien that was hit. Take special note of the part which fires the alien rocket, this part basically checks every alien to see if it is directly above the player’s base and if it is and if the alien rocket is not visible (therefore not already fired), it fires the rocket. The very last IF statement in the action checks for the alien sprite set count reaching zero, basically meaning all the alien sprites have been destroyed, at which point a new wave is created by calling addinvaders. Notice though that we are altering one thing in the next wave, which is the speed the aliens move at, so that as the player progresses through wave after wave of aliens it becomes faster and therefore harder & harder.

• Add one final action (shown below), which will only be called when the game ends (lives reaches 0). This action posts the player’s score to the bazaar high score table and displays it for 5 seconds, along with a “Game Over” message. It then prompts the player to ask if they want to play again and then either calls the resetgame action or stops the script, depending on the response.

```plaintext
private action endgame ()
   | wall → clear
   | bazaar → post leaderboard score(score)
   | bazaar → post leaderboard to wall
   | "Game Over" → post to wall
   | time → sleep(5)
   | if wall → ask boolean("Play again?", "") then
      | ▶ resetgame
   | else
      | time → stop
```
To finish the game all we need to do now is to call the two actions we just created from within the gameloop, do this as shown below (marked in red). Make sure the endgame action is called against the final ELSE, which if you look carefully should mean it will only be called when \( \text{lives} > 0 \) is not true, therefore when \( \text{lives} \) is equal to or less than 0.

```
event gameloop ()
|| sprite sheet → evolve
|| scoreboard → set text("Score " || □ score)
|| livesboard → set text("Lives " || □ lives)
|| board → evolve
|| board → update on wall

Check if the game is running
if □ lives > 0 then
▷ move base
▷ check limits
▷ move aliens(□ aliens, 5, 0, □ board → width)
▷ check for collisions
else
▷ endgame
```

Run your game script and check everything works. It should look something like the screen capture shown below.
Challenges
Making use of what you have learnt so far, try and complete the following challenges in your game.

- Add explosion animations and explosion sound effects for when an alien ship is destroyed, for when the base is hit by an alien rocket and for when a shield is destroyed. You can use more than one sprite sheet animation graphic in the same game, as shown below.

  ```
  sprite sheet → initialize(board)
  sprite sheet → set sheet("good explosion", explosion, 4, 5)
  sprite sheet → set sheet("bad explosion", base explosion, 5, 5)
  ```

- Add sound effects for when both rockets are fired.
- Add an alien mothership that appears every so often, which the player can shoot for bonus points.
- Change the graphics for the aliens after each wave is destroyed.
- Publish your script.

End of session 4.