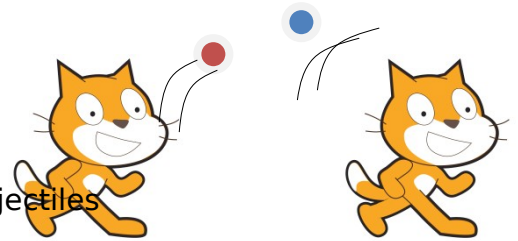


Scratch Workshop - Intermediate

In this workshop you will learn how to:

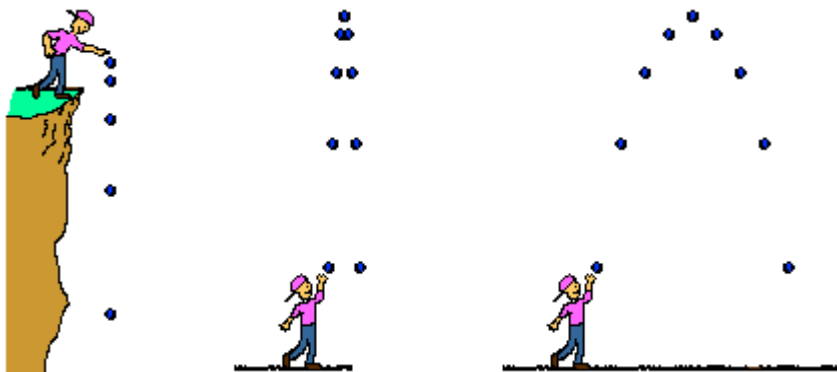
- Model projectiles in Scratch
- Create a two-player game in Scratch using projectiles



Introduction to Projectiles

Projectiles are objects that move through the air due to some force acting on it.

For example, when you throw a ball up into the air, the ball is the projectile, and the movement of your arm provides a force.



While a projectile is in the air, there is another main force acting upon it - gravity! (let's not worry about air resistance for now) Otherwise, it would just float off into space!

Projectiles in 2D

Scratch exists inside a 2-Dimensional world. This means that we have to worry about two axes of motion - x (left and right) and y (up and down).

This effectively means that the projectile will have a certain **velocity** in both directions. Think of velocity as speed, but velocity relates to a direction. You can have a negative velocity but not a negative speed!

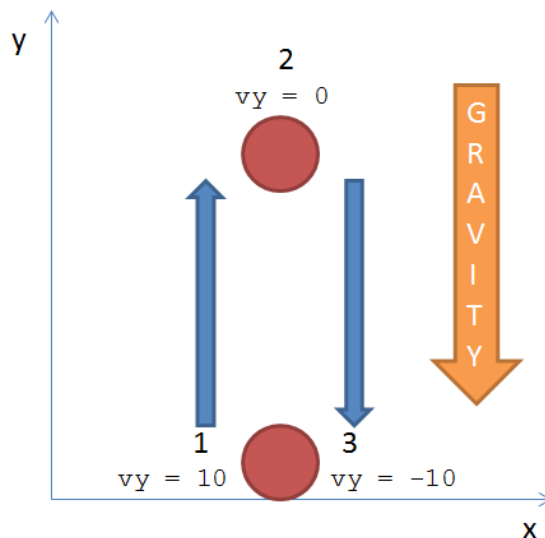
This means that each projectile has two **variables** we need to keep track of:

1. x velocity (v_x)
2. y velocity (v_y)

Let's take a look at the y velocity (up and down) first.

Projectile Travelling Directly Upwards

Imagine you have a ball and you throw it directly upwards (in line with your



body).

The only movement is in the y-direction (up and down), so we don't have to worry about the movement in the x-direction. Hence, in our 2D world, v_x is set to 0 and doesn't change.








v_y on the other hand does have a value, and it will be constantly changing due to **gravity**.

Gravity is modelled by decreasing v_y over time. In Scratch, this can be performed by using a 'forever' loop.

Without further ado, let's do some Scratch!

Projectile Modelling in y-Direction

1. Open up Scratch and either make your own projectile sprite or choose a sprite from the default sprites folder (any one without an attached script will do)
2. With your sprite selected, click on the **Variables** tab and then click 'Make a variable'
3. Call the variable name v_y , and ensure that 'For this sprite only' is selected, then click 'OK'
4. Make another variable and call it $gravity$, this time ensure that 'For all sprites' is selected, then click 'OK'
5. With the sprite selected, click the **Control** tab and drag across the 'when [flag] clicked' method to the script panel
6. Go to the **Motion** tab and connect a 'go to x:[] y:[]' under the 'when [flag] clicked' method to initialise the projectile's start position (you need to set the x and y values to something appropriate)




7. Next we need to set the initial y-velocity. In the  **Variables** tab, drag the 'set [variable] to []' method onto the script, ensuring that `vy` is selected on the variable drop-down menu, then enter an initial value (e.g. 20)
8. Begin a loop statement by dragging a 'forever' method from the  **Control** tab and connecting it to the script
9. Now we want to move the ball. Find and drag the 'change y by []' method inside the 'forever' loop
10. Instead of entering a number, we want to use our variable `vy`. Drag the variable from the  **Variables** tab and drop it into the [] section of the 'change y by []' method
11. Now we need to set the `gravity` variable's value. Click on the Stage and create a script which sets a value for the `gravity` variable when the green flag is clicked, just like how we did it for the `vy` variable in the projectile sprite script. Set the variable's value to -1
12. Select the projectile sprite again. Now add a 'change [variable] by []' method from the  **Variables** tab, within the 'forever' loop and underneath the 'change y by `vy`' method. Select the `vy` variable from the drop-down menu, and drag the `gravity` variable to the [] box
13. Now we want to stop the ball when it reaches its starting position again. Drag an 'if' from the  **Control** tab and place a 'stop all' method inside of it. The 'if' method should be inside the 'forever' loop
14. We want the 'if' statement to fire when the sprite's current y position is below its starting y position value, so begin by fetching a '[] < []' expression from the  **Operators** tab, placing it in the gap of the 'if' statement. Then drag 'y position' from  **Motion** and place it in the first box of the expression. Finally enter the starting y value into the second box (refer to your 'go to x: [] y: []' method)
15. Press the green flag!

Extra Exercises:

1. Add another projectile to the scene and add the same script as above, with a different start position so you can see it. What happens when you change `vy` for one of them?
2. Change the value of `gravity` and observe the results on your projectile(s)
3. Change the script so that the projectile bounces off the bottom of the screen and comes to a stop

Making a Two Player Game

Now we're going to game-ify what we've got so far. Here's where it gets really interesting!

1. The first thing we need to do is add the moveable object (e.g. person, slingshot) for each player. Choose or create an appropriate sprite for one of the players, giving it a name, e.g. player1.
2. With the player1 sprite selected, set an initial starting position using the 'when [flag] clicked' and 'go to x: [] y: []' methods
3. Within the  **Control** tab, find the method which is activated by a key press, and add it to the script
4. Choose a key for moving left in the drop-down menu for the method you just found, and drag an 'if' method underneath it
5. Set the condition within the 'if' statement such that the methods within it only get processed if the player can move left without going off the screen (**Hint:** there is more than one way of doing this. Try using a 'if ... else' method with a condition set via the  **Sensing** tab)
6. Now initiate movement by dragging a 'change x by []' method into the script, then set an appropriate value
7. Right-click on the top method of the block of methods you just created, then click 'duplicate' to generate a copy of the block. Now you just have to change the key, 'if' condition, and the value for the change in x
8. Now we turn our attention to the projectile. Click the projectile sprite for player1. We only want to make the projectile appear when a button is pressed by the person controlling player1, so the first thing you need to do is only set the initial `vy` value and drag a 'hide' method from the  **Looks** tab, placing them both underneath a 'when [flag] clicked' control method. Try not to delete the other code just yet, just drag it outside of the block!
9. Next, we want the projectile to respond to a keyboard input by appearing at player1, travelling upwards, then back down, then hiding again once it stops. You should have enough knowledge to figure this part out for yourself (**Hint:** you should only be adding methods to the projectile sprite's script, don't forget to reset `vy` when you hide it again)
10. Once you can move your player sprite and fire a projectile sprite from it, you're ready to add the other player. Right click on each sprite and then click 'duplicate'. Rename the sprites, and edit their costume so that they are distinguishable (e.g. change the colour using the paint bucket tool)
11. If we try playing the game now, only player2's sprite will show, because it has exactly the same code as player1! Edit the appropriate values such as key presses and start position, in order to fix this.
12. We need to keep a track of the number of lives each player has, so create variables called `Player 1 Lives` and `Player 2 Lives`. Think about whether the variables are available to all sprites or just the player (**Hint:** Do both players need to use the variables?). Choose the best place (**Hint:** sprites or stage?) to give a starting value for them, then add the relevant methods to the script
13. Now we need to enable players to take damage. Within each projectile sprite, change the script underneath the 'when [key] pressed' block to

change the other player's lives variable by -1 (**Hint:** use the 'touching

[sprite]' condition within the  tab)

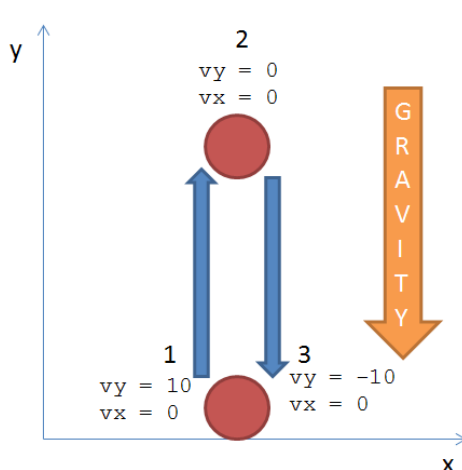
14. One last thing to do - an end-game screen! (**Hint:** Create a block of code in an appropriate place, which shows an end-game sprite when lives variables reach a certain value)

Extra Exercises:

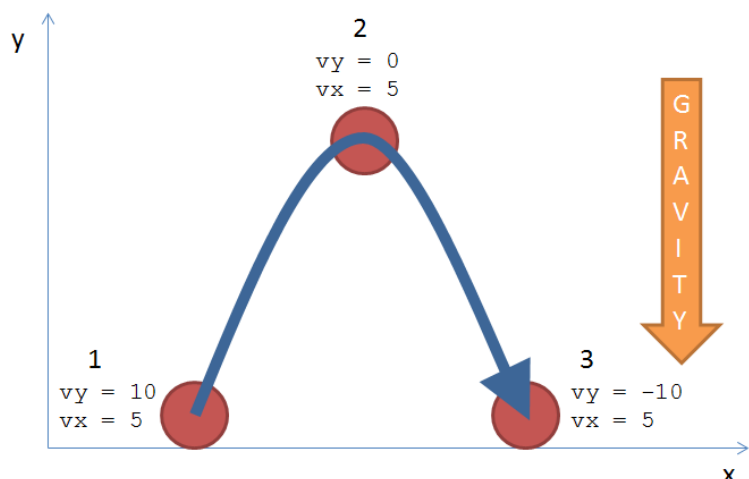
1. When a player reaches 0 lives, change sprite costumes to indicate winning/losing
2. As a player moves off the side of the screen, make them reappear on the opposite side
3. Introduce the idea of 'catching' in that each player can press a button to catch the projectile being thrown at them, then they do not lost a life

Adding Movement in x Direction

Gravity only acts in the y direction, so what does this mean about the value of v_x ? - It doesn't change while the projectile is in flight! (remember, we're ignoring



air resistance)



We can quickly add this to our game by doing the following:

1. Create a new variable for each projectile sprite called v_x , and only make it available for the sprite; not all sprites should have access to it
2. Wherever you have set the value for v_y , set a value for v_x
3. Underneath changing the y position of the sprite ('change y by [v_y]'), change the x position of the sprite to v_x
4. Test it out!

Extra Exercises:

1. Add control of the v_x variable by using keyboard inputs, for each player
2. Can you add an indicator for the direction of the projectile, for each player (e.g. an arrow pointer)?
3. Experiment with changing the starting y position of a player, observing the trajectory

Adding Power and Angle of Projection

We can use slightly more advanced maths to give more control over the projectile's trajectory, given a starting angle and power value. This will involve some trigonometry functions. If you haven't come across these in your maths lessons then don't worry, all you need to know is written below:

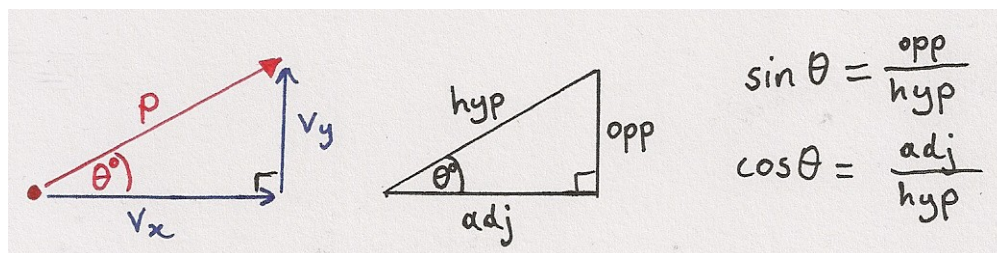
$$v_y = \text{power} * \sin(\text{angle})$$



$$v_x = \text{power} * \cos(\text{angle})$$



Here's a diagram explaining how we figure this out, but don't worry if you don't fully understand it.



Let's get started!

1. Begin by creating new variables, `power` and `angle`, for each projectile sprite
2. Set an initial starting value for the two new variables
3. Use the 'set' methods above to set the values for v_x and v_y based on `power` and `angle`. Do this everywhere in the script where v_x and v_y are set
4. Add blocks of code which change the value of `angle` when specific keys are pressed (think about setting restrictions on values)
5. Add an angle indicator using a sprite (**Hint:** think of which variables the new sprite needs access to), for each player. Things you need to consider are:
 - a. Which variables the new sprite needs access to (or how you can get around that)
 - b. Possibly changing the centre point of the costume (in the edit costume window)
 - c. How to ensure the indicator moves with the player sprite

Extra Exercises:

1. Allow each player to control the power of the force applied to the projectile
2. Try adding a randomly-changing force of wind
3. Allow the velocity of the projectile to have an effect on the amount of damage dealt to the other player
4. Think of something else you could add to the game to enhance it

Now you're an expert at 2D projectile modelling using Scratch! Remember, Scratch is available on all platforms, not just the Raspberry Pi, so you can transfer the files you make using your Pi's SD card onto another computer, and share with your friends! You can also share to the Scratch community at <http://scratch.mit.edu> by registering a (free!) account and uploading your work.

I hope you enjoyed this workshop. See you at the next DigiMakers event!