# Tutorial Assignment

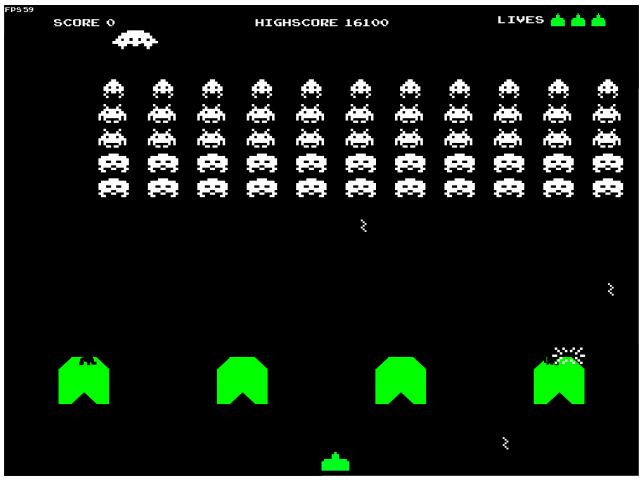
**By Cameron BLACK**

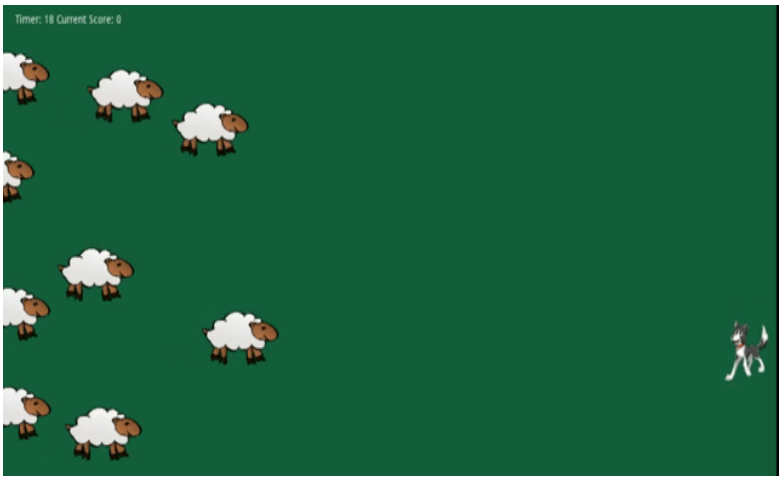
# “Making your own Space Invaders”

---

This tutorial for Game Development demonstrates how to create your own version of a side scroller shooter. In this tutorial, you will learn how to use the Godot coding language ‘GD Script’ and create an interactive shooter experience called ‘Border Collie Hunt’, a game based on the concept of Space Invaders. A game in which you play as a dog, and bark (shoot bullets essentially) at sheep (the enemies) to herd them together.

While the concept of a dog herding sheep game may not be appealing, this tutorial includes the core fundamentals of a shooter scroller game.




Blackstorm18. (2019). *Sion invader.jpg* [Photograph]. Wikimedia Commons. https://commons.wikimedia.org/wiki/File:Sion_invader.jpg CC BY-SA 4.0.

To the left details the Space Invaders experience, and to the right is the game you will develop.

The following concepts you will learn include (in no particular order):

- Version Control
  - How to save your work and properly use the ‘Git’ system for your project. Essentially moving your project to a storage space on Github.
- Player Movement / Shooting
  - Giving the player their own inputs/actions is quite literally game-changing.
- Countdown Timer / Global Variables
  - Adding elements of game design to your version of Space Invaders, such as your score, or a timer (as most games require).
- High Score
  - The use of a filing system to save the top scores in a leaderboard environment, saving the top 5 scores.

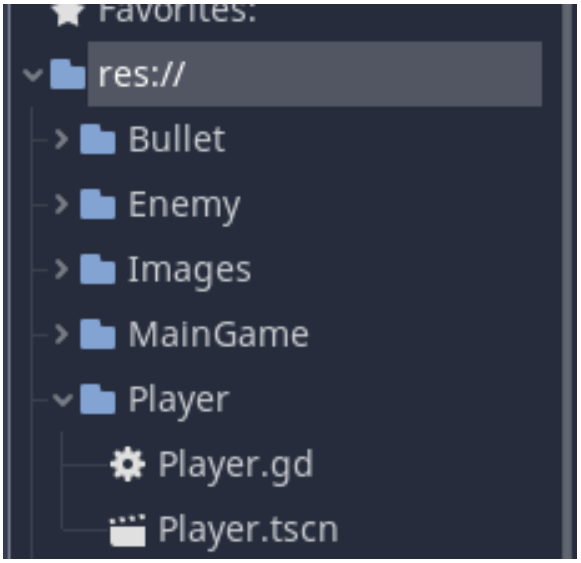
This tutorial provides a step-by-step guide to game development but is also designed to help improve your understanding of the concepts. There will be exercises at the end of each section to test your knowledge, and it is very important that you do them to develop your project.

# Player Movement:

**2 EXERCISES WITHIN**

Kaiser, K. (2021). *Warrior Right Walk* [Gif]. Wikimedia Commons. https://commons.wikimedia.org/wiki/File:Warrior_Right_Walk.gif CC0.
Puny Human. (2014). *Blade Symphony - Ryoku breakdance.gif* [Gif]. Wikimedia Commons. https://commons.wikimedia.org/wiki/File:Blade_Symphony_-_Ryoku_breakdance.gif CC BY-SA.

Once you have set up your project on Github and Godot (https://github.com/LTC-IT/BorderCollieHunt), the next step is to look into how to give the player some action. Because the end product is said to be a side scroller shooter game, there needs to be elements of shooting and moving from enemies!




On the bottom left of your godot screen (once you have opened the editing portion of the project), you will see a `res://` area (within the FileSystem tab), containing all the images and code for the project. These can also be seen within the natural files of the project when you look at it in File Manager.

Going into the player.gd “script”, this is where you will start the development of your player’s actions.

A script is a general file for an object’s code (what it can do). Say, for example, the player.gd script would hold player movement, but the bullet.gd script would kill the enemies. The player.gd script is also connected to the Player file, linking the player’s character (the dog) with their own script. Script doesn’t just appear in the game, scripts must be linked to scenes or objects.

Once you go into the file, there should not be much there to begin with. As you may see, the `func` lines (otherwise known as functions), do not have any use at the moment. Functions are used specifically to group together many lines of code, and is generally the more difficult aspect of development. Functions are utilised for the purpose of enacting a task.

```
Filter scripts
MainGame.gd
Player.gd

1  extends KinematicBody2D
2
3  var movement_speed = 200
4  var bulletSource = preload ("res://Bullet/Bullet.tscn")
5
6  func _ready():
7      set_process(true)
8      set_physics_process(true)
9
10 func _physics_process(delta):
11     pass
12
13 func _process(delta):
14     pass
15

Player.gd
```

But you should note that the **`bulletSource and movement_speed`** variables are already set. These **`var`** lines help identify lines of code simplified into a 'variable', which is then usable in later code. Variables are used to identify things and simplify them for use in later tasks.

Imagine you are part of a team of biology workers, in order to categorise people's characteristics, you'd need to put eye colours under 'eyes', heights and weight under 'body shape', hair colour under 'hair' and so on. Categorising important data is how variables work!

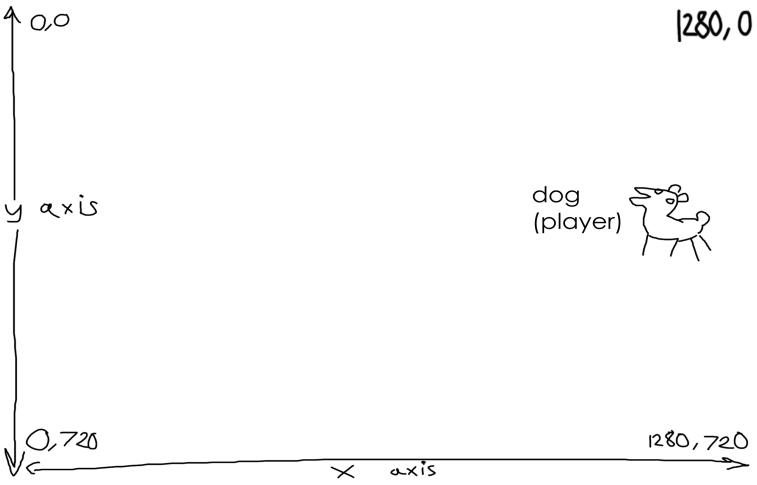
By identifying the two previous variables, we can use them to go towards setting the player's movement speed and the preset code to use for the bullets. So how do we get the player to move? Godot specifically uses a 'physics process' to enable physics. However, when you deal with movement in video games, most coding platforms like Unity or Unreal Engine will have their own terms for physics.

```
6  func _ready():
7      set_process(true)
8      set_physics_process(true)
9
10 func _physics_process(delta):
```

This will let the player's character move when given the appropriate inputs.
Given that this **`set_physics_process`** is the Godot version of enabling physics for our game.

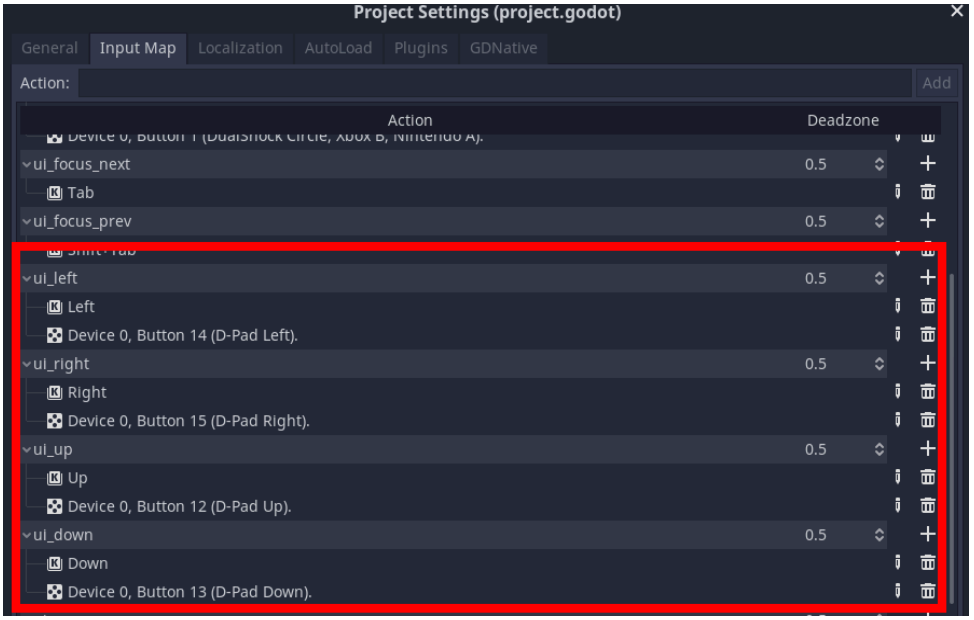
## The X and Y values:

However, to move your character, first you need to look at the game. The environment in Godot may be confusing, but the basis is that Godot's y axis is flipped.




So you'd need to increase the y value of the object's position (controlling the player) to go down, and decrease the y value to go up.

Noting this, we can now understand how we move the player in the game to move them up and down.




If you go into 'Project' in the very top left, you may see 'Project Settings'. This is where you can see the inputs that players have in game. These have not been implemented yet, but note that these can be changed.

The input for left (as an example) uses the left button on the keyboard, so you do not need to specify the left button as it is already categorised here, alongside other buttons.

---

## EXERCISE 1: WHAT GOES UP MUST COME DOWN

In order to get the player to do things, we need to give the game 'if' statements. This will say that when one thing happens (such as pressing a button on the keyboard), then a second thing will happen (like the main character moving).

For 'if' statements, these are called decisions. In everyday use, you could use decisions to plan your work schedule, or times to water your plants. For your project, decisions are used to determine how your player can move. **If** your player presses up **then** the player moves up.

### **Delta**:

Delta is a very important part of character movement as it moderates how much a player moves, specifically to moderate the movement speed, which is primarily based on your computer processing speed. "People may have differently running computers, so "Delta helps moderate this inconsistency by putting the movement speed into 'Units per second', allowing the movement of a character to be more exact" (CodeNMore, 2020)

CodeNMore. (2020, July 19). *Godot Basics: Delta* [Video]. YouTube. https://youtu.be/cikN_k_dm7I

```
func _physics_process(delta):
    if Input.is_action_pressed("ui_up"):
        if position.y > 40:
            move_and_collide(Vector2(0, -movement_speed * delta))
    if Input.is_action_pressed("ui_down"):
        if position.y < 680:
            move_and_collide(Vector2(0, movement_speed * delta))
```

Sometimes as well, you will have to add onto your if statements, to give a more exact detail.
How this code works is that **if the up input is pressed, and if the y value of the player's position is more than 40** (40 being close to the top of the game window) **then the character lowers their y value to go up** (and increases their y value to go down).

The reason why the y value (of the character's position) has to be more than 40 y and less than 680 y is so that the character does not go outside of the game window when moving. You can also see that in the `move_and_collide` lines that the x value (zero), doesn't change. This is because **the character is supposed to go up and down the screen**, not the usual left and right (that being the x axis).


You should have something like this.

Where you are right now:
The player can press either up or down and the character will respond by going either up or down.

However we aren't quite done with our player. There's no element of a game here. All that happens right now is sheep just moving across the screen… where's the action? The shooting?

---

## EXERCISE 2: BARKING BULLETS


Dick, I. (2006). *Image from Halo 3* [Photograph]. Flickr. https://www.flickr.com/photos/ian_d/298225136 CC BY 2.0.

The player can move up and down, a key aspect of sorts… but can the player shoot?
While this exercise follows similar 'if statement' instructions, this tutorial focuses on the use of variables. During the game, the player cannot control the movement of the bullets, nor are they affected by the movement of the player, so this calls for the use of `func _process`.
`_process` is used to control other aspects of the game excluding movement. For the player, it would be to spawn bullets.

Considering the following terms of 'process' and 'physics process' you may see in the `func _ready` Area, that it always has lines of `set_process(true)` and `set_physics_process(true)`. These lines allow the script to enable 'process' and 'physics process' to become functions in code.
Whenever the game is run, the `_ready` function is enabled, as it runs when the game is 'ready'. If 'process' and 'physics process' are enabled by the 'ready' function, then they are enabled as well.

```
extends KinematicBody2D

var movement_speed = 200
var bulletSource = preload ("res://Bullet/Bullet.tscn")

func _ready():
```

1

Looking from the first specified line, the player.gd script already has a preset asset for the bullet (barking, because it's a dog the player controls). The `bulletSource` is set as a **.tscn** right now, meaning "scene", which describes the game's images and game screens (such as when you go to the main menu into a game).

```
func _process(delta):
    if Input.is_action_just_pressed("fire"):
        var bulletInstance = bulletSource.instance()
        bulletInstance.position = Vector2(position.x-40, position.y)
        get_tree().get_root().add_child(bulletInstance)
```

2

In order to use the bullet code, it needs to be converted from a scene to an instance, something we can use in the main game.
Of course the bullet needs a direction to go, so if you give the `bulletInstance` a position of -40x and keep the y the same, the bullet should go left (negative x according to earlier x/y graph) towards enemies.

Now that you can summon bullets as their own entities, time to herd some sheep! Wait… no the bullets don't move.

Well that's because the code that is being run just summons bullets, it doesn't allow them to move. Like the character moving, the bullets need their own `_physics_process`.
In order to fix this, enter the 'bullet.gd' script in the Bullet file.

When you're there, you shouldn't see anything underneath `func _physics_process`, which explains why the bullets don't move! Because they aren't attributed with the power of Physics!

```
func _physics_process(delta):
    pass
```

```
var speed = 500
```

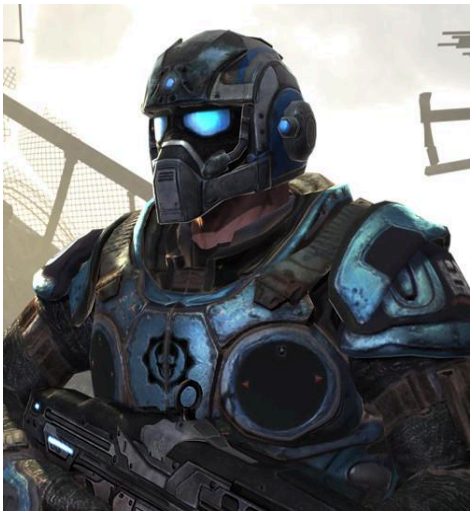
```
func _ready():
    set_physics_process(true)
```

However, as you can see above `func _physics_process`, there is a `set_physics_process(true)` statement (implying there is physics needing to be applied here), but most importantly a variable called 'speed'. This variable is similar to the player's 'movement_speed' variable, but is not the same. These two variables are in two different scripts and cannot overlap.

---

First, in order to move the bullets we need to assign a '`move_and_collide`' line, allowing the bullet to move across the 2 Dimensional plane, also known as **Vector2** (we are not dealing with 3D games so you don't need to worry about 2D or 3D stuff).

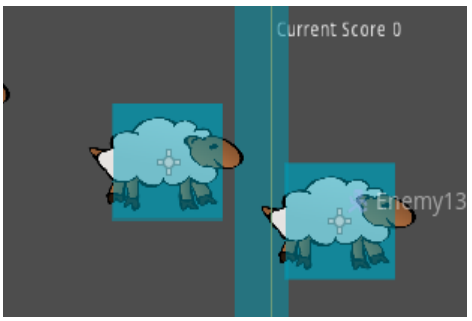
```
func _physics_process(delta):
    var collidedObject = move_and_collide(Vector2(-speed*delta, 0))
```



Gamerscore Blog. (2008). *Gears of War 2 - Carmine* [Photograph]. Flickr. https://www.flickr.com/photos/gamerscore/3022163073 CC BY-SA 2.0.

But now we have a newer problem, the bullets bounce off the enemies! How is that possible? Are they BULLETPROOF?

Well actually no, we didn't code the bullets to do anything to the sheep. We actually have to manually set the bullets to "herd" the sheep using a thing called '**Colliders**'!




You may have noticed on the main screen of your game, that every sheep has their own '`EnemyNumber`' tag, these are each sheep's nodes (including their collision boxes within their character).
In the following image, we use an 'if' statement to check **if** the sheep has "Enemy" in their name **then** using `.collider.name` we can make it do something when the bullets come into contact!

The line `collidedObject.get_collider().queue_free()` is used mainly to make the sheep disappear, because like in Space Invaders you shouldn't see the enemy once you kill it! Make sure to implement the below code into your project (in the bullet.gd file).

```
func _physics_process(delta):
    var collidedObject = move_and_collide(Vector2(-speed*delta, 0))
    if (collidedObject):
        print(collidedObject.collider.name)
        if "Enemy" in collidedObject.collider.name:
            collidedObject.get_collider().queue_free()
        queue_free()
```

After adding the code to the bullet.gd file and playtesting the game, you should have something like this!

By following these instructions, you've made the basis of a side scroller shooter game, shooting enemies and designing movement primarily!

**Now it is here that we have our `INTERMISSION`. We will not be designing something new, but it is a reminder to save your work and upload to Github. You might have been asking how to save this project from the beginning, and it's here that you'll learn how to.**

---

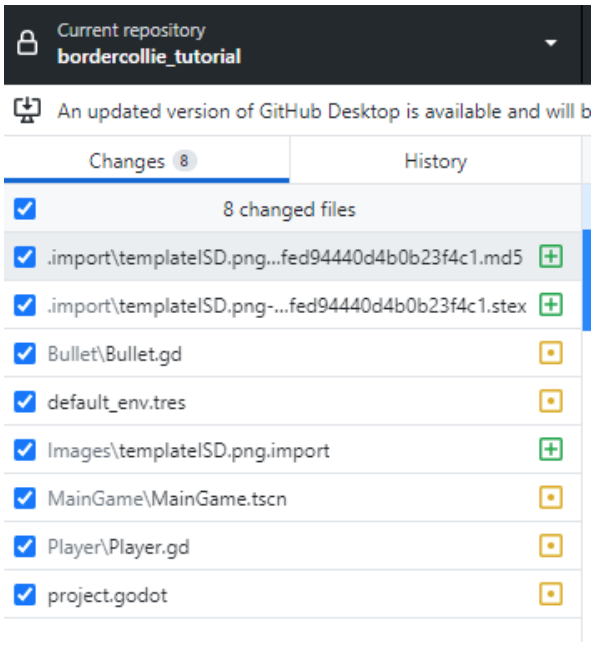
## Version Control:

**0 EXERCISES WITHIN (INTERMISSION)**

Yunus, G. (n.d.). *Save Icon in Colored Outline Style* [Photograph]. Iconscout. https://iconscout.com/icon/save-1769151 CC BY 4.0.

Now that you can shoot enemies and move the player, it is time to save your project. Don't worry, you will be reminded later to save as well.

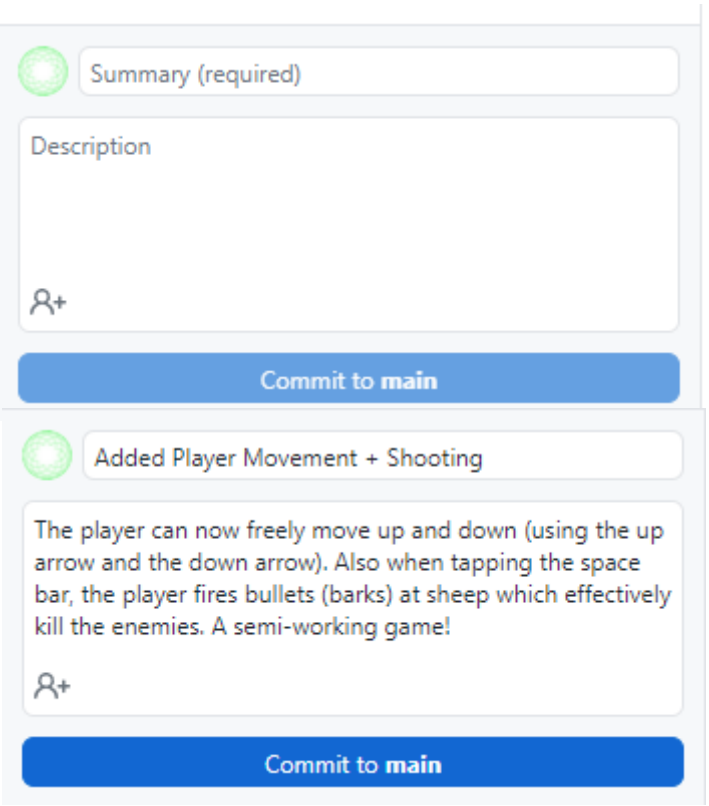
Before leaving your project in Godot, remember to use **Ctrl + S** as it saves your project to your **COMPUTER**. The main use of this portion in the tutorial is how to upload / save your project to Github (as either a backup spare or a resource for your coworkers if you need it for a group project of some kind).




When you exit Godot and enter Github Desktop, all of your changes that you made to the game's files will appear here. Github Desktop will give you an overview of what files you've changed or added.

You do not need to worry about the consistency of the left example, but this is a basis of the overview you will receive when you get to this part in the tutorial.

While your repository is located on your computer, it's important to note that the repository can also be found on your Github Account. This is useful for people such as your teacher or your coworkers to access your Godot projects and add to it themselves.

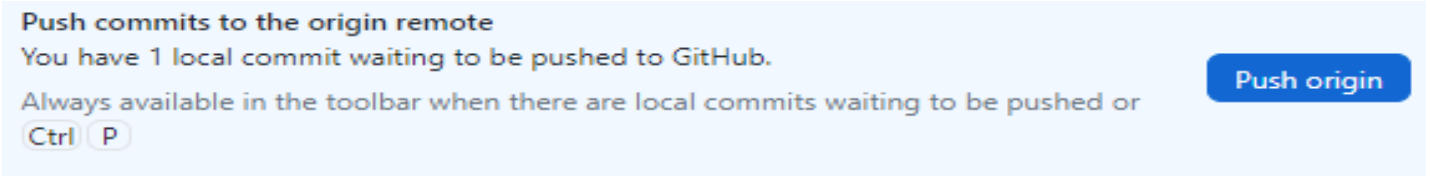


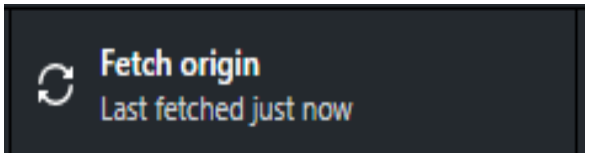

In the bottom left corner, you should see a 'commit to main' option with a few text boxes. In order for you to submit your work to Github (for safekeeping :D) is to write an overview of your changes (to find out what you changed and when you changed it!).

When you fill out the text boxes, the 'commit to main' option will appear. The '**main**' being your Github Repository Storage on your main Github Account.

But also, if you have used a repository (doesn't have to be your current project, it could be any) on any other device with the same account, you will need to verify your 'committed' changes by '**pushing**' the project into Github.

Lastly, if you wanted to access your project's changed files from a new computer, you would need to 'pull' the project from Github, as it would have the latest version.




Be careful with **pulling** however, as it can override uncommitted work. It is much safer to '**fetch origin**' most of the time, as it will download the committed files from the github website (which **pulling** also does), but it can overlap with current changes from the current computer's files. "Fetch will never manipulate, destroy, or screw up anything. This means you can never fetch often enough." (Git Tower, n.d.)

*What's the difference between git fetch and git pull?* (n.d.). Git Tower. https://www.git-tower.com/learn/git/faq/difference-between-git-fetch-git-pull

## Countdown Timer:

**1 EXERCISE WITHIN**


Brx0. (2011). *Countdown clock* [Photograph]. Flickr. https://www.flickr.com/photos/atul666/6535101111 CC BY-SA 2.0.

Now, you're ready for the next part of the planning process. Adding game elements such as a timer, scoreboard, etc. In this section we will be focusing on the timer, an important part of shooter games.


Because of the use of a timer, we'd be using a thing called `int` otherwise known as Integers. Integers are numerical values that are primarily used to calculate mathematical equations or functions in code. However, we will use `int` to determine the countdown timer for our game. We use `int` for a mathematical purpose because we want the countdown timer to *count down* during the game.
Other uses of `int` may include: turning statements into values, maths equations of adding and subtracting, and listing a numerical value set.

Phin, J. (2021). *A series of numbers in an animated gif* [Gif]. Wikimedia Commons. https://commons.wikimedia.org/wiki/File:Numbers.gif CC BY-SA 4.0.

We've looked at integers, but now we look at Strings. Strings are how we use basic words in our programming, such as printing a phrase like 'Today is Friday' or 'Cool Hat!'. Strings are also used to turn integers into words, as integers on their own would not be seen in the normal game. Strings help with this by putting integers into their numerical symbols (0, 1, 2 and so on…).

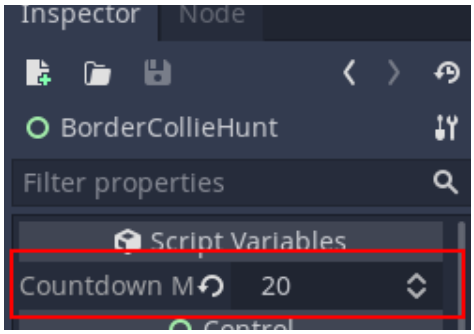

Bent, D. (2012). *Hello, World!* [Photograph]. Flickr. https://www.flickr.com/photos/zengei/7317420838 CC BY-SA 2.0.

## EXERCISE 3: TIME IS ON YOUR SIDE

Starting the development of a countdown timer, we need to set a basis of how long this timer will be. To do this, we need to use a variable called **`countdownMax`** which we can 'export' directly into the game's nodes. Please put the below code into the 'mainGame.gd' script.

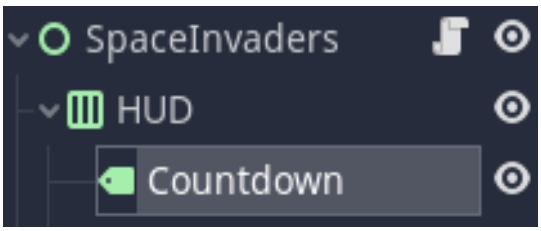
```
extends Control


export(int) var countdownMax
```




By exporting this variable, we can freely change the value of the countdown's maximum length. So now we can decide the length of the game.

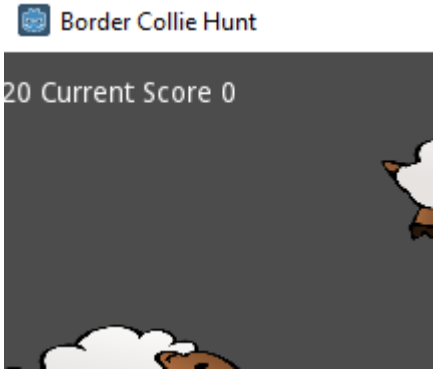
It would roughly take around 15 seconds to kill all the enemies, but for a good standard timer for new players, let's say the countdown is 20 seconds long.




```
export(int) var countdownMax
var currentTimer
func _ready():
    currentTimer = countdownMax
    $HUD/Countdown.text = str(currentTimer)
    
    set_process(true)
```

For the countdown timer to actually appear within the game you need to use the HUD to display text. This is commonly used for keeping track of the player's score ingame, but is also used for the countdown timer, as it is crucial that the player must know these things ingame.

The reason why **`countdownMax`** has been converted to **`countdownTimer`** is because we need a way to *change* the timer's value every second. **`$HUD/Countdown.text`** uses the countdown 'label' (text on the game screen) to show the game's timer, and yet the timer is still an integer… so turn that into a string statement. This set of code should allow us to see this timer ingame.




Indeed it does. But however, it doesn't move. This is because all that is happening is **`countdownMax`** uses a singular value (20) and just shows that value, with nothing changing it.

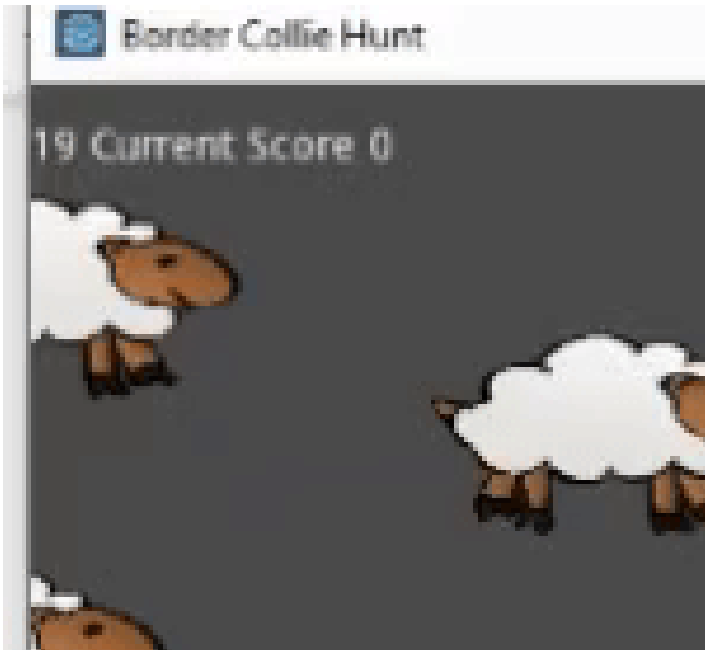
To make a functioning timer, you just have to decrease the current value by 1 every second. This is where '**`While`**' statements come in.

`While` statements use an almost always running group of code dependent on one line of code to preface (mainly a variable). For real life examples you could say: “While I watch the news, I am going to eat my breakfast.” or “while I'm writing up my homework, I will listen to music.” `While` statements are supposed to be for doing something based on the result of another thing (in our case the timer variable).

Our statement that we have here “`while currentTimer is greater than 0:`” shows that while the game is still running, this is how the timer goes down.
First of all, there is a specified timer option called ‘`get_tree().create_timer(1.0)`’, which goes by every second, with a phrase “timeout” when it reaches zero (stopping the timer entirely)

An important aspect to the timer is ‘`currentTimer = currentTimer - 1`’, which details how every second the timer goes down by one. Like a normal timer should. Whilst including the $HUD/Countdown.text to show it ingame, we can see this timer more clearly in the output menu. Be sure to add a “Game Over” ending text when the timer does hit zero.

```
while currentTimer > 0:
    yield(get_tree().create_timer(1.0), "timeout")
    $HUD/Countdown.text = str(currentTimer)
    currentTimer = currentTimer - 1
    print(currentTimer)
print("Game Over")⇐APPEARS IN OUTPUT
```


The timer in output shows how the timer system works, similarly how the HUD text represents the timer the same.
Overall, you’ve finished this exercise with a crucial game element under your belt, a game timer. Now the game has evolved from “Herd the sheep.” to “Can you herd all the sheep in 20 seconds?”, with much more to go! Pretty impressive stuff there.