Introduction to GameMaker: Workshop 1
ITCS 4230/5230

Learning Outcomes

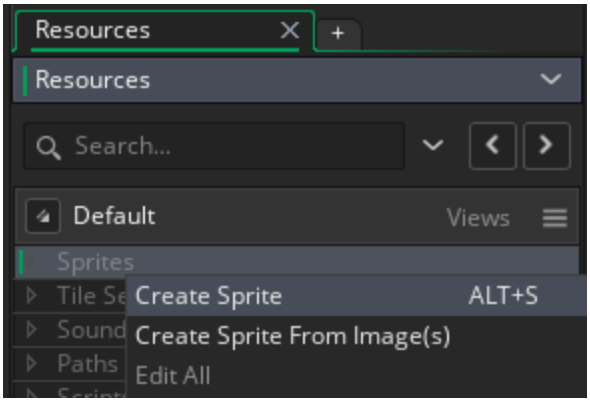
By the end of the workshop the student will be able to:

1. Use events & actions to manipulate object properties, including:
    a. x & y coordinates
    b. direction & speed
    c. health & score
2. Use the GM sprite editor to create & modify game sprites.
3. Understand and apply the use of collision events.
4. Understand how to use conditionals to implement game functionality.
5. Apply blocks to execute groups of actions in conjunction with conditionals.
6. Use the draw event to display specific elements on the screen.
7. Understand and apply the concept of object inheritance.
    a. Create objects that use inheritance to reuse functionality.
8. Understand how to use alarm events to control game behavior.

Prologue: Setup

1. Create a new game project in *GameMaker Studio*.
    a. Choose **Drag and Drop**
    b. Name your project *ScrollingShooter_studentid*, where *studentid* should be replaced with your UNC Charlotte “800” number.
    c. Make sure to save your files in a place where you can easily find them. We recommend a folder dedicated to this course, which is in your *Documents* folder.
2. Download the *Game Resource*s zip file from canvas.
    a. Unzip the file
    b. Copy the file’s contents to the folder created by GameMaker for you project
3. Create sprites for the player plane and for the background
    a. Navigate to the Resources panel, usually located on the far right of the program window.

b. Right-click on Sprites, then select Create Sprite



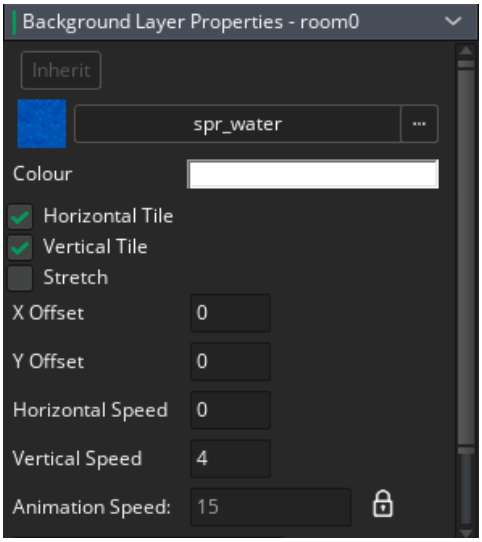
c. Rename this sprite as *spr_plane*. This will hold the graphics for our player.
d. Below the name field, click Import, and navigate to the *"myplane_strip3.png"* image from the resources zip file.
    i. Because of the *"_strip3"* portion of the filename, GameMaker automatically tries to separate this image into 3 subimages horizontally. Since the width of the image is divisible by 3, this will be successful.

    ii. If you are curious, try renaming the image file as "myplane_strip5.png" and importing it. 5 subimages will be created, though this certainly does not produce desirable results.

e. Repeat this process a second time, creating *spr_water* as a sprite using the *water.png* image. This is just a static image of water, so it will not be separated into subimages.
f. Save your project.

You can read more about the GameMaker Sprite editor here.

4. There should already be a single room present in the **Resources** panel, room0. Selecting this will open the **Room Editor**, which you will use to manipulate various elements of your game environments.
    a. The room starts out with two layers (found on the left edge of the window), *Instances* and *Background*. Instances is where the player and other objects will go, but for now let's deal with the Background.
    b. Selecting the background layer, we can configure a few things. Set the sprite to spr_water, then check Horizontal Tile & Vertical Tile. The layer will now display the water sprite in a repeating pattern across the screen.



    c. Below Horizontal Tile & Vertical Tile are a few more settings. To indicate movement, set Vertical Speed to 4. The y-axis is top-to-bottom in GameMaker, so this will cause the layer to scroll downwards at 4 pixels per game update. (One update in GameMaker is one frame, so at a default of 30 fps that's 30 updates per second)
5. From the Resources panel, right click Objects to create obj_player, the player object.
    a. For now, just assign spr_plane to the player object. We will work on making it functional shortly.
    b. You can now drag obj_player from the Resources panel into room0. Make sure the *Instances* layer is selected!

6. Save your project
7. Pressing the **Play** button (sideways triangle near the top of the window) or **F5**, GameMaker will build the game and play it.

Prologue Checklist

- ❏ Two sprites, one for the player & one for the water
- ❏ A background layer that covers the screen with water & scrolls downwards
- ❏ A player object that displays the player sprite, which looks like a plane flying over water

---

Part 0.5: Drag ‘n Drop & Game Maker Language

For writing the actual code of your game, GameMaker provides two options: A Visual Scripting interface, and a programming language loosely based on C/C++. YoYoGames provides a good explanation of each within their own documentation.

Drag & Drop Overview
Game Maker Language Overview

It is worth mentioning here that GameMaker Studio has a robust and thorough documentation, available both in a web browser and through the program itself. When working through this workshop, if a function or action is unclear in how it works, we highly recommend checking the documentation.

When possible, instructions for this workshop will be given in both GML and Drag & Drop. Understand, however, that there is not a Drag & Drop action for every function in GameMaker. At least some amount of GML will be necessary. As you work through the workshop, **when instructions for the same functionality are provided both using Drag and Drop and GML, you should only implement one of these instructions**.

Part 1: Player Movement

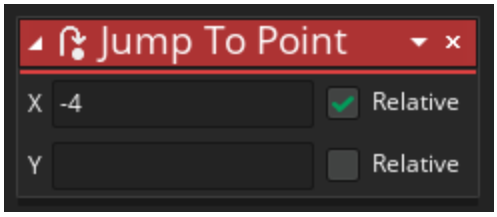
1-A: Keyboard Controls

Stop the game and return to obj_player. We can add **events** to this object that will react to specific inputs and other occurrences, such as pressing keys on the keyboard.

Method 1: Keyboard Events & Drag 'n Drop

This method uses events that react directly to keyboard buttons. **Key Down** events are what we'll use, as they are called as long as the respective key is held down, but there are also events for when a key is pressed or when it is released.

1. Add a **Left Key Down** event
2. To move to the left, we can use the Jump to Point action.
    a. Jump to Point has two fields for x and y coordinates. If you leave a field blank, it won't change the position on that axis. Set the x field to -4, and check the box labeled "Relative".
    b. The result is that obj_player will move 4 pixels to the left of its current position.



3. Repeat this process for the 3 other directional inputs. Remember that the y axis is top-to-bottom, so up would be -4 and down would be +4.

Method 2: Step Event & GML

This method uses the **Step Event**, which will inevitably see use no matter which method you go for. The step event is an event that runs once every time the game is updated (by default, that is 30 times per second). Using GML, it is then very easy to group together all of the keyboard inputs together into one place in the step event.

1. Add the **Step** event to *obj_player* and the **Execute Code** action
2. The function keyboard_check() mirrors the Key Down event, returning true if the specified key is pressed.
3. Moving to the left can be accomplished the same way you'd manipulate number variables in other languages. x -= 4 would move the object 4 pixels to the left.
4. A full statement to move the player when the left key is pressed can be written as follows:

    if keyboard_check(vk_left) x -= 4

5. Repeat this process for the 3 other directional inputs.

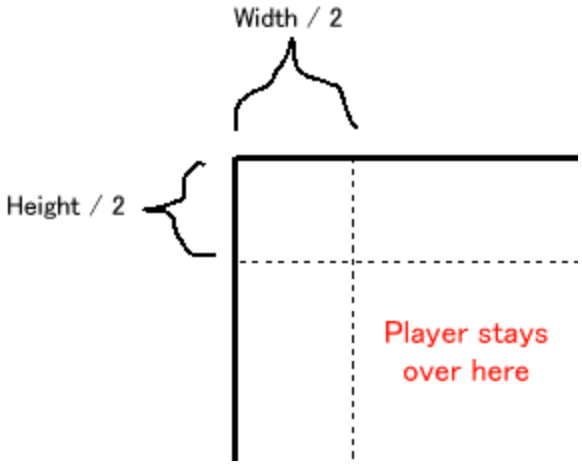
```
1 //Movement Inputs
2 if keyboard_check(vk_left) x -= 4
3 if keyboard_check(vk_right) x += 4
4 if keyboard_check(vk_up) y -= 4
5 if keyboard_check(vk_down) y += 4
```

Regardless of which method you used, you should now have a player object that moves up, down, left & right with your keyboard inputs. *Save and run your game.*

1-B: Staying within screen bounds

Currently, the player is capable of leaving the screen from any direction. Let's add some constraints to keep them within bounds.

1. To make this easier, start by setting the origin of the player sprite to "Middle Centre" (check here if you're confused about how to do this). Centering the sprite's origin will make it easier to write code involving the player's position relative to other locations.



2. With the sprite's origin centered, we now know that the distance from the player's (x,y) position to the edges of the sprite is equal to half the sprite's width horizontally, and half the sprite's height vertically. In order to prevent the player from leaving the screen, we then can constrain their (x,y) position so that they stay farther than that distance from each edge of the screen.
    a. The built-in variables we can use to facilitate this constraint are sprite_width, sprite_height, room_width, and room_height. They're fairly straightforward in how they work - they're values based on the current sprite of the object and the current active room.
    b. To actually restrain the x and y positions, we can make use of the clamp() function.
    c. Here's the horizontal position, as a freebie: the left bounds are "sprite_width / 2", and the right bounds are "room_width - (sprite_width / 2)". From there, constraining the horizontal position can be done with: **x = clamp(x, sprite_width/2, room_width-sprite_width/2)**. It is up to you to devise a similar statement for the vertical position.

3. A good place to enact these constraints is in the step event. By placing this in the step event, the game will check that the player does not leave the bounds we define each time an in-game update occurs. As a refresher, the update happens 30 times per second by default.
    a. This can either be done by writing the statements as shown above in GML, or using the **Jump to Point** action in Drag & Drop, followed by an **Assign Variable** action that includes the *clamp()* function.
4. When you run the game, the player should now be unable to leave the screen.

1-C: Vertical Momentum

Now, let's slightly alter how our vertical movement works. Instead of moving at a static 4 pixels up or down at a time, we could instead use a system of acceleration and deceleration. This can be achieved with the variable vspeed, which will automatically move an object's y position every game update.

1. Starting this change is simple. Where you would move the y position upwards, instead set vspeed -= 1, and do the opposite instead of moving downwards. Pressing up and down will now increase or decrease vspeed, which will cause the player to move at ever faster speeds the longer a button is held down.
2. What do you notice about this? Two problems should be clear - the player quickly reaches speeds that make maneuvering difficult, and it is equally difficult to come to a complete stop. Vspeed is allowed to increase or decrease indefinitely with no limit so long as the key is held down. Additionally, vspeed only changes when the player is holding a key; therefore, it will not come to a stop on its own.
3. We can address the first problem by clamping vspeed in a manner similar to how we previously clamped the x and y positions (vspeed = clamp(vspeed, min, max)).
    a. The minimum value allowed is the maximum upwards speed. From the plane's perspective, this is how fast it can move **forward.** Play around with different values for this, but remember that this value must be negative (-y is up)
    b. Let's do something different for downwards momentum. Planes can't necessarily move *backwards*, so we can limit their maximum speed to the background scrolling speed. This can be obtained via layer_get_vspeed(). Provide the background layer's name as a string, & it'll return its vspeed.
    c. Given a negative number **u**, which equals the maximum upwards speed you've chosen, the equation for clamping vertical momentum can be written as:

        vspeed = clamp(vspeed, u, layer_get_vspeed("Background"))

    d. Similar to the last bit, this can be done in GML directly or in Drag & Drop, in the Step event.
4. For the second problem, the player's vspeed should decelerate back to 0 when up and down are not pressed.
    a. Fortunately, there *is* a Drag & Drop action to check if a key is / is not pressed, so you have that option. Otherwise the GML keyboard_check() function will suffice. These checks need to be performed each game update by using the step event.
    b. When the plane is moving upwards, we would add 1 to vspeed to decelerate. When moving downwards, we would add -1. An easy way to accomplish this is subtracting the sign() of vspeed. As long as your vspeed only consists of whole numbers (no decimals), this will consistently bring it down to 0.

Part 1 Checklist

- ❏ The player's plane can be moved around with the arrow keys
- ❏ The player will always remain entirely on screen
- ❏ Moving up and down involves accelerating & decelerating
- ❏ The player has a max vertical speed in each direction
- ❏ Downward vertical speed does not exceed the scrolling speed of the background
- ❏ Vertical speed decelerates to 0 when neither up nor down are pressed

---

Part 2: Islands

Now, we're going to create island objects that scroll down with the background and reappear at the top of the screen after scrolling off of the bottom.

2-A: Screen Wrapping

1. Start by importing the *island1, island2* & *island3* images as sprites. Go ahead & center their origins like we did with the player.
2. Make an object out of island 1. To scroll with the background, we only need to set its vspeed once, in the **Create** event. This event runs when an object is first created. Since these islands will exist at the start of the game, their create events run as soon as the game begins.
    a. We can use the layer_get_vspeed() method to set the island's vspeed.
    b. The vertical speed can be set with a **Set Speed** action if using Drag and Drop, or by using vspeed in GML.
3. To determine when the island moves off of the bottom of the screen, we can employ a method similar to how we clamped the player's position. In this case, we'll want to check that the island's sprite has entirely left the visible screen. (This will be implemented most effectively in the island's **Step** event)
    a. You can start with the statement "if y > room_height then y = 0". It's simple, straightforward & mostly functional. You'll see, though, that the island will move from the bottom to the top of the screen while it's still visible, which looks a bit unprofessional.
    b. To alleviate this, we can make use of sprite_yoffset. It's the y coordinate of the sprite's origin. This returns a number representing the distance between the y value at the origin of the sprite and the y value at the top of the sprite. Go ahead and click on the link for a visual representation of this. We can factor this into our previous statement to ensure the jump from bottom to top is not seen on screen.
    c. room_height + sprite_yoffset is just below the bottom of the screen, while -sprite_yoffset is just above the top (Note that this particular relationship only works because the origin is at the exact center of the sprite). If the sprite is 64x64 and the origin is at the center (32,32), the yoffset of the sprite would be 32. Each game update, the code is checking to see if the y value of the sprite's origin is 32 pixels greater than the room's height.
4. It's worth noting at this point that entity load order determines which objects get drawn first/last. That is, islands placed after your player will likely appear to be above it, rather than on the ocean below. This can be avoided by putting islands on a separate **Instance** layer below the player's. (Likely between that layer & the background layer)

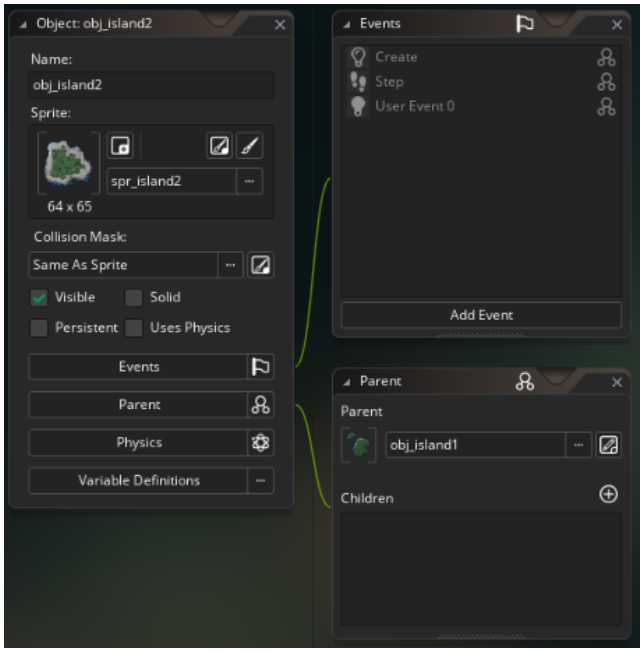
2-B: Random Variation

1. As it stands, it still looks a bit strange that the island appears at the same horizontal position each time. When we reset the island to the top of the screen, we can randomize its x position to add some variation.
    a. irandom_range() is a function suitable for this purpose. It generates a random whole number between the min & max values you provide it. This allows us to randomize the position while still remaining within the horizontal bounds of the screen.
    b. Since we're already working with offsets, sprite_xoffset will fit here. Similarly to how we used sprite_yoffset to loop seamlessly from the bottom to top of the screen, we can employ sprite_xoffset to define minimum & maximum values for the island's random x position.

    c. In the same place you had previously set the y position, add a new statement that randomizes the x position with irandom_range().

2-C: Inheritance

1. Now we have a fully functional island. However, there are still two other island types. Instead of duplicating/rewriting the code for these two, we can create objects that inherit the first island's behavior.
    a. Create objects for the second and third islands.
    b. Set a sprite for each object.
    c. Press the "**Parent**" button, and use the resulting window to set both new islands' parent to island 1. You'll see that they gain island 1's events without the need to write any more code.



Part 2 Checklist

- ❏ Island objects scroll downwards with the background
- ❏ When an island fully leaves the screen, it will reappear at the top of the screen.
    - ❏ When this happens, its horizontal position is randomized.
    - ❏ Despite the randomized positions, islands will always remain fully within the horizontal bounds of the screen.
- ❏ There are 3 different types of island, two of which inherit the behavior of the first.

Part 3: Enemy Planes

Time to add a challenge to the game. Enemy planes will actually behave very similarly to the islands we already implemented, but will have a handful of interactions with the player.

3-A: Initial Setup

1. Import *enemy1_strip3.png*, and create an object for it. *Remember, make sure to center its sprite origin!* Have this object be a child of island 1, much like the 2nd and 3rd islands. You should again not have to change/add any code for it to function just like the islands do.
2. Now, we can start making modifications to enemy 1. It'd make more sense for the enemy to move downwards faster than the background scrolls, so we have to change how its vspeed is set in its create event. We have the option of overriding individual events inherited from an object's parent, and we'll do just that for the create event.
    a. From the Events window, you can *inherit* or *override* each event. Right-click on the event to see a menu from which to choose. In Drag & Drop, *inherit* and *override* both do the same thing, whereas in GML *inherit* will start you with an event_inherited() statement, which will call the parent's create event. The equivalent of this in Drag & Drop is Call Parent Event. Using event_inherited() will execute the code in the parent's event so that the child's event can be continued.
    b. The parent's create event already sets vspeed based on the background scroll speed, so we can inherit that and add an extra value to vspeed directly afterwards. This would ensure that no matter what speed the background scrolls at, the enemy planes will always move faster.

```
1 event_inherited();
2 vspeed += 2
```

3-B: Player Collision

1. The default planes will simply charge downwards and explode on contact with the player. Part one of this behavior is complete - now to handle the contact. On enemy 1, add a collision event that occurs against the player. This event will trigger whenever an enemy touches the player.
2. What to do here is fairly straightforward: simply destroy the enemy on contact with the player. instance_destroy() in GML, or Destroy Instance in Drag & Drop.
    a. From here, it may become difficult to continuously debug the enemy's behavior. After all, it's capable of permanently removing itself from the game. You may find it beneficial to include a quick way to reset the game. We recommend adding a **Key Pressed** (not Key Down) event for some key of your choosing that restarts the game. **game_restart()** or **Restart Game** are your options for GML and Drag and Drop, respectively.

3-C: Cool Explosion

1. Right now the enemy's destruction is rather underwhelming. We just so happen to have an explosion graphic we could use for this. Import *explosion1_strip6* and make an object out of it. Don't forget to set the origin!
2. Notice that this explosion sprite has a clear start & end, and isn't intended to loop. Therefore, we need to get rid of it once its animation has concluded. Fortunately, the Animation End event exists for purposes such as this. In the explosion's Animation End event, destroy it.
3. Now we need a way to create an explosion whenever an enemy is destroyed. We'll just use a Destroy event. The destroy event runs once the object it is attached to is destroyed.
    a. Within enemy 1's destroy event, your options are the **Create Instance** action, as well as the instance_create_layer() & instance_create_depth() functions. Just create an explosion at the enemy's position (0,0 relative), on the enemy's layer.

Part 3 Checklist

- ❏ The new enemy object inherits the movement behavior from the islands, but changes it to travel *faster* than the background scroll speed
- ❏ Enemies are destroyed on contact with the player, creating an explosion object that itself gets destroyed once its animation concludes

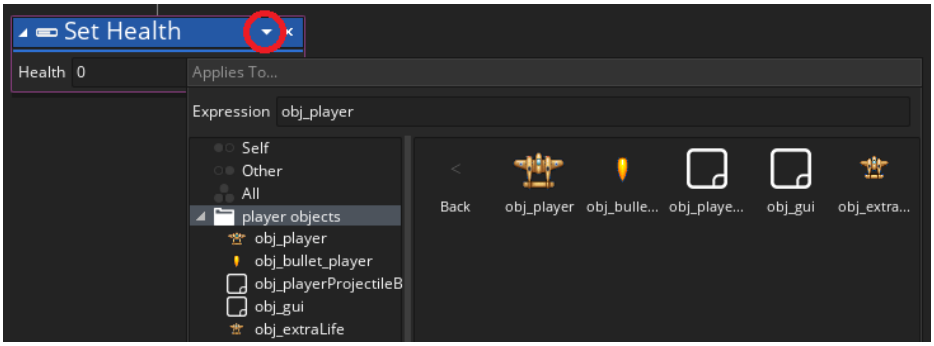
---

Part 4: Health & Damage

Currently, there is not a proper consequence for colliding with another plane. Yours is apparently far sturdier, and can ram other planes without a scratch. To change that, we'll implement health. The player will take damage every time they collide with a plane, and if their health reaches 0 they'll explode, just like the enemies.

4-A: A Talk on Scope

1. Before we get started with player health, let's talk about scope.
2. GameMaker has three built-in global variables: health, lives & score (if you reference these in GML they will turn green). They have no inherent meaning, but do know that they are **global** - every object has access to the same health, lives & score values. For our purposes, we will instead be using instance variables - whose values are unique to the object instance using it - contained within the player. If obj_enemy1 had an instance variable named hp, two instances of obj_enemy1 would possess unique values for that variable.
    a. GameMaker's Drag & Drop actions regarding these 3 properties use an instance variable. These include the Get/Set Health, Draw Score and others like that.
    b. To define your own instance variables (in GML or Drag & Drop), do so in the object's create event.
3. When accessing/changing an instance variable in a different object, we'll need to preface the variable with the name of that variable's object. For example, to access the player's hp, we can write *obj_player.hp* (substitute "hp" for whatever variable you've defined).
    a. When using Drag & Drop actions, such as Get/Set Health, the action itself must be pointed at the object you are trying to access. See the screenshot below for an example.



4. More importantly, to access an instance variable, that instance must exist. At a certain point, our player object will be destroyed and replaced by an explosion. **If any object tries to access our player's health after that point, the game will crash.** In the context of the enemy's collision event with the player, this won't be

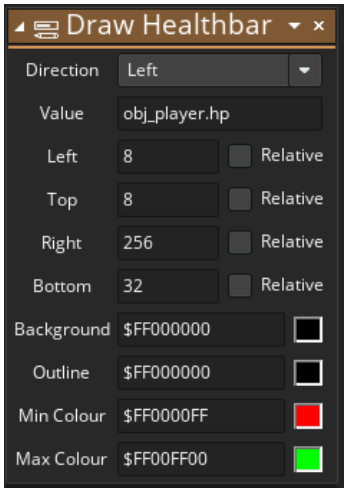
an issue, as we can assume the player exists if we're colliding with them. However, other situations won't allow us to make that same assumption.

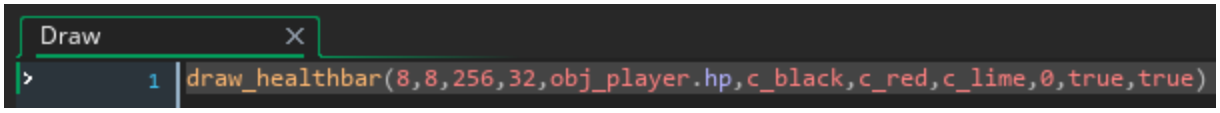
4-B: Inflicting Damage

1. In the player's Create event, initialize your health variable to 100.
2. **Return to obj_enemy1's collision event with the player**. In it, subtract 30 from the player's health. Then, if the player's health is <= 0, destroy the player.
3. Go ahead & test this out. To deal enough damage for the player to be destroyed, you'll need to place at least 4 enemies in the room to collide with.

4-C: Displaying Health

1. Currently, there's no way of knowing how much health your player has left. We can render a health bar to show us this information.
2. Start by creating a separate object for displaying health & other information, **obj_scoreboard**. This object doesn't need its own sprite, but you will need to place one somewhere in the room.
3. *obj_scoreboard* will use a **Draw** event to display our health bar. **Any functions/actions that deal with rendering graphics must be used within a Draw event - otherwise, they do nothing**.
4. Refer to the following screenshots for implementing a health bar. Figuring out good screen coordinates for GUI elements can get tedious, so this is to save you some time.





5. Think back to what we said regarding the variable scope earlier. Because your player health value is an instance variable, you always need to be sure the player exists whenever you try to access it. Currently, our scoreboard *assumes* the player exists, which is not always guaranteed. With an instance variable for health, if the player gets destroyed your game will likely crash.
    a. The easiest way to fix this is to manually check if the player exists before drawing the health bar. Use instance_exists(obj_player) or the **If Instance**

Exists action to properly verify before accessing the player's health, or any other instance variable.

4-D: Exploding & Restarting

1. Currently, once the player runs out of health they just disappear, and the game just continues on without them. We'll add a new explosion object for the player to use, which will deal with this problem.
2. Import *"explosion2_strip7"* and use it to make *obj_playerExplosion*. This will function just like the enemy explosion, so you can inherit its behavior.
3. The one addition you need to make is to give it a destroy event. When obj_playerExplosion is destroyed, restart the game.
4. Then, give the player a destroy event in which it creates obj_playerExplosion.

Part 4 Checklist

- ❏ The player has 100 health points, displayed in a health bar
- ❏ Crashing into an enemy plane deals 30 damage to the player
- ❏ When the player runs out of health, they explode (with a bigger explosion than the enemies). Once this explosion concludes, the game automatically restarts

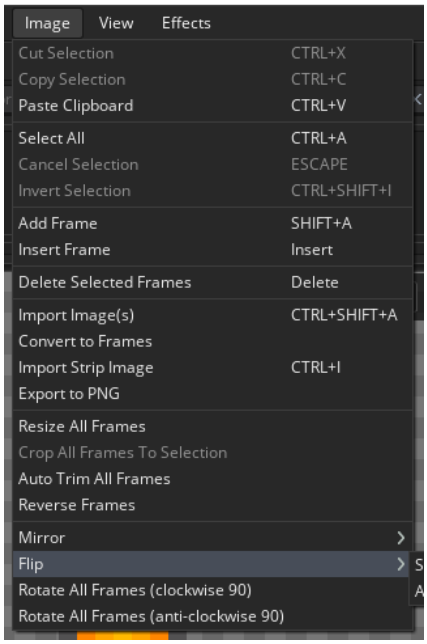
---

Part 5: Combat & Score

Now that the player has been made vulnerable, it'd be nice to be able to fight back. In this part, we'll give the player the ability to fire bullets that can dispatch enemies from a safe distance, as well as provide us points for our score.

5-A: Creating projectiles

1. Import *bullet.png* from the external resources. Before you make an object out of it, note that the sprite looks like it's upside down…
    a. With the sprite open, clicking "Edit Image" will bring you to the engine's built-in sprite editor.
    b. From here Image -> Flip will flip the sprite so that it's correctly pointing upwards.



2. *Now* let's make an object. *obj_playerBullet* will have a few simple jobs
    a. Travel upwards
    b. Upon collision with an enemy, destroy both objects.
    c. Additionally, you'd want the bullet to destroy itself once it's left the room. We actually have an event for this, **Outside Room**.

5-B: Firing projectiles

1. To actually fire these bullets, we'll implement a control scheme in obj_player as follows: As long as the spacebar is held, fire bullets at a regular interval.
2. The important component here is the delay between shots. We don't want to allow the player to fire a bullet every frame (that'd be way too many bullets), so we must define when the player can & cannot fire, as well as for how long.

3. Let's define a variable called canShoot in obj_player that defaults to true. From there, if canShoot = true and the spacebar is pressed, fire a bullet and set canShoot = false.
4. From here, we'll also need a way of setting canShoot back to true afterwards. We can make use of an alarm.
    a. If using GML, see here - the example code provided on the alarms page is actually exactly what we're going to implement. When the player fires a bullet, set an alarm to a reasonable value (15 is a good number), and after that many game updates the player will trigger a corresponding alarm event. That's where we can reset canShoot back to true.
    b. If using Drag and Drop, use the Set Alarm Countdown action.
    c. Note that the alarm index in the action must match the index of the Alarm event.
5. If done properly, your player should fire off 2 bullets per second as long as you hold down the spacebar.
    a. If you set the alarm to a different value, it will fire at a different rate. For example, if the alarm is set to 10 game updates, it will fire 3 bullets per second. (30 game updates per second, bullet fires every 10 updates.)
    b. A better practice would be to access a built in variable called room_speed. This variable returns the number of game updates that GameMaker will perform per second. To fire at a rate of twice per second, the alarm's countdown would be set to room_speed / 2. This is useful in case the amount of game updates per second is changed later, as well as making the specific timing more readable for the coder.
    c. If for whatever reason you want to manually trigger an alarm, look into the event_perform() function. (Note: setting an alarm to 0 will not trigger it instantly, instead it won't trigger it at all.)

5-C: Scoring Points

1. Score is in a similar situation as health. There's a builtin global variable intended to use for score, but for now we will define our own variable for it as well. Drag & Drop actions use an instance variable for score. Make sure that whichever object keeps track of the score should be around for most of the game. It is a good idea for your obj_scoreboard to store your score, as it will always be available.

2. Let's start by drawing score on the scoreboard. If you used our previous set of coordinates, this should fit in nicely with it (the parameters should transfer over just fine to the Drag & Drop action as well).

```
//health
draw_healthbar(16,16,144,32,obj_player.hp,c_black,c_red,c_lime,0,true,true)
//score
draw_text(16,64,"Score: "+string(points))
```

   a. Note the quirk here within the draw_text() function: GML is a language in which you need to manually convert non-character values to a string if you want them to be treated as such.

3. Now to actually change the score, we can return to obj_playerBullet. In its collision event, simply add to the score. Whenever the player destroys an enemy with a bullet, the score should now increase (e.g. +10 points), while the same will not occur if the player & enemy crash into each other.

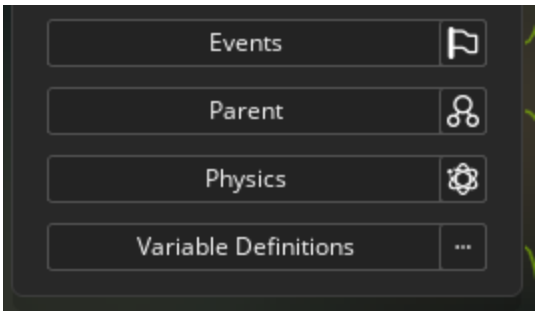
Part 5 Checklist

- ❑ As long as the spacebar is held down, the player will fire bullets at regular intervals
- ❑ Player bullets destroy enemy planes on contact and add to the player's score.

Part 6: Enemy Variants

Currently, there's only one type of enemy to fight. This ultimately limits the types of challenges that could be provided to the player, so let's add more types. Though, instead of just making new enemies that inherit from our existing one, let's do a bit of future-proofing…

6-A: Variable Definitions



1. Chances are you've seen the "Variable Definitions" button in the object window, under Events & Parent. The Variable Definitions window allows you to… *define variables* for later use in actions and GML code. The immediate difference here is that they're organized in a convenient interface, but more importantly it allows a child object to easily change the value of a parent's variable.
    a. For example, we have our enemies set up to travel slightly faster than the background scrolls, but chances are that's a hard-coded value. Using a variable definition, we could define an *extraSpeed* variable that holds that value, allowing us to easily change the speed of individual enemies.
2. Let's set up two Variable Definitions for our enemy object:
    a. *extraSpeed*, an Integer (or Real number)
    b. *scoreValue*, an Integer
3. Then, we'll make some adjustments to our code to use these new variables:
    a. In the enemy's create event, use *extraSpeed* to increase the instance's vspeed
    b. In the player bullet's collision event, increase the score by the enemy's *scoreValue*
        i. Getting this one right's important. Since we now have the possibility of different enemies having different *scoreValues*, we want to get the value from the specific enemy the bullet collided with. In a collision event, the "other" keyword can be used in place of an object name to provide the specific object instance involved in the collision.
        ii. This could be written as: score += other.scoreValue

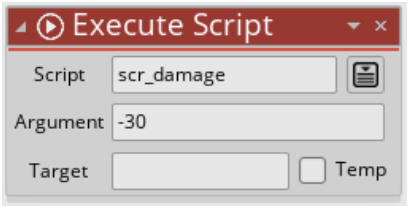
6-B: Orange Enemy Plane

1. Now that the setup is done, let's *finally* create a second enemy. Import *enemy2_strip3.png*, and follow the usual procedures to make a new child of enemy1

2. As a start, change around the declared variables to distinguish our two enemies:
    a. Enemy 1 will move faster, and is worth an average amount of points
    b. Enemy 2 will move slower, and is worth more points
3. Enemy 2 is currently the easier target, but we can make it more dangerous by having it fire bullets of its own.

6-C: Multiple Damage Sources

1. Create a new bullet object based on *enemybullet1.png* that moves downwards & destroys itself on contact with the player. It will also destroy itself if it's outside the room.
2. To actually damage the player, this presents a new problem. Every time we want to inflict damage, we need to do the following:
    a. Decrease player health by an amount
    b. Check player health's new value
    c. If player health <= 0, destroy the player
3. It's not a lot of code but it adds up, and you have to do it correctly every time. If we, perhaps, had a function we could call to inflict damage and do the related health check afterwards, that'd be great. We can get exactly that by using a **Script**.
    a. Create a new script, with a name you'll remember (scr_damage?).
    b. Transfer your existing damage code from the enemy1 collision into this script
    c. Your existing code should've decreased player health by 30. Replace this 30 with *argument0*. This allows the script to accept one parameter.
    d. In the enemy1 collision event, call your damage script instead of running the damage code. Be sure to pass in 30 as a parameter! (And make sure this still works like it used to before continuing).
4. Now that we have a simple script we can call to inflict damage, use it to inflict 5 damage in the enemy bullet's collision event.
    a. If using Drag and Drop, call the script using the **Execute Script** action.



For details, see https://docs2.yoyogames.com/source/_build/3_scripting/1_drag_and_drop_overview/action_scripts.html

b. Also, the arguments passed to the script will need to be -30 and -5, respectively.

6-D: Orange Enemy Plane Fires Bullet

1. The firing mechanism we'll use is similar to how the player character shoots. In this case, however, we'll set up a *looping* alarm, and fire a bullet every time it triggers. For an alarm to run continuously, just reset the alarm to its starting value when the alarm's event occurs. ***In Alarm 0 Event, set Alarm 0 to 15***, etc. You can set the first alarm for the loop in the **Create Event**. *Remember that when we need to inherit a parent's event, we will use event_inherited() or Call Parent Event.*\

2. Another thing we can do here is use variable definitions to set up some more variation:
    a. Add a *shotInterval* number that defines what the enemy's alarm is set to each time
    b. Add a *shotType* Resource (use the options to restrict it to only objects) that defines what type of bullet object gets created
3. With these variable definitions, we can have enemies that fire more or less often, and with different types of bullets

6-E: White Enemy Plane

1. Let's now make a third enemy based on the second. Import *enemy3_strip3* and set up an object as a child of enemy 2. This third and final enemy will be the slowest, worth the most points, and fire bullets aimed in the player's direction, rather than straight down.
    a. Since we've done all this setup, the only part you should have to change about enemy3 are the values of its Variable Definitions.
2. We *will* need a new bullet object - one which will aim towards the player. Import *enemybullet2.png* and use it to create this new bullet object, inheriting its behavior from the original.
3. This new bullet will have a unique Create event, completely overriding the original. In it, we'll set the direction of the bullet using the point_direction() function or **Set Point Direction** action. The parameters used will be the x and y coordinates of the player.
    a. If you think back to the Scoreboard, you may predict a potential problem here. If one of these bullets is created while there isn't a player around, the game will crash. Therefore, if the player doesn't exist (check for that), you need to instead set direction manually.

    b. Direction is a builtin variable that uses degrees and rotates counter-clockwise, with 0 pointing right, 90 pointing up, 180 pointing left & 270 pointing down. Therefore, when the player doesn't exist, just set direction to 270 to send the bullet downwards.
4. Setting direction alone will not cause the bullet to move. You'll also need to set the bullet's speed. The two pairs *hspeed & vspeed* and *direction & speed* work parallel to each other, where changing one pair will affect the other.
5. One final thing we can do for the aiming bullet is clamp its direction. Currently the plane is capable of firing in all 360 degrees around it, which can be disorienting to deal with. Knowing that a direction of 270 points directly downwards, clamp the bullet's direction between 240 and 300.
6. Now that the new bullet's all set up, just make sure *obj_enemy3* is actually firing it.

Part 6 Checklist

- ❏ There are now 2 new enemy types
- ❏ Enemy 2 shoots bullets straight downwards, moves slower and is worth more points than enemy 1
- ❏ Enemy 3 shoots bullets that aim towards the player within 240 & 300 degrees, moves the slowest and is worth the most points.
- ❏ Variable Definitions are used to easily configure enemy speeds, point values, shot intervals

---

Part 7: Spawning

Up until now, it's been up to you to manually place enemies in the room. Now let's create a spawner object to deal with automatically and periodically creating enemies.

7-A: Enemy Spawning

1. Before we start with the spawner, there's something we can do with our enemies first. Ideally, when creating new enemies, we don't want them to immediately pop onto the screen. It'd work well to have them scroll onto the screen from the top, much like they do after they leave the screen from the bottom. If we could reuse the code that places them at a random horizontal position at the top of the screen, that'd work great.
    a. You may see what I'm getting at here. Go all the way back to our original island object, take the code that modifies its *x* & *y* positions when it leaves the screen, and make a script out of that.
    b. If you call this script in the create event for enemy1, other enemies will inherit that event so all our spawner would have to do is create them. Afterwards, they'll reposition themselves so that they naturally enter the screen.
2. For the actual spawner, we'll use a setup nearly identical to how enemy2 is set up to fire bullets. Set up interval & object type Variable Definitions, and use them to set up an alarm loop that creates the desired object.
    a. Spawn the enemy instance at a random x coordinate. Set the y coordinate taking into account the sprite's height to position the enemy at a negative location on the y axis, so it looks as if the plane is arriving from the top of the screen.
3. There is another important feature of variable definitions that we can use here. Once an individual instance of an object is placed in a room, the values in variable definitions will be set to the values that we already defined inside of the object. These values can be edited for each individual instance once placed inside a room. For example, it would be possible to create a single instance of enemy2 that fires 15 times per second by placing it in the room and editing its variable definitions. (Feel free to try that if you want! You're about to clear the room of enemies anyway.)
    a. This way, you can drop 3 spawners into the room, then configure them to spawn *enemy1, enemy2* & *enemy3* respectively, with different spawn intervals.

    b. Double-click on the spawner object (question mark) to get a dialog box where the variables can be set.
4. Clear out any pre-existing enemies and run the game with just the three spawners. Play around with the spawn intervals until you get a balance that feels good to you.

7-B: Health Pickups

1. With this spawner system, it'd be really easy to add helpful items to be periodically spawned as well. Import *life.png* for a new health pickup. This new object is different enough that there's nothing in particular it can easily inherit from, but it's mostly things we've done already.
    a. In the create event, use our randomizer script to place it above the screen. Then, set its vspeed equal to the background scrolling speed.
    b. When the object leaves the bottom of the screen, destroy it.
    c. When the object collides with the player, reset the player's health to 100 and destroy the pickup.
2. Now, all you need to do is add an additional spawner configured to create health pickups. They're very helpful, so you probably shouldn't create them *too* often.

Part 7 Checklist

- ❏ A spawner object can be configured to create a particular type of object at a set interval. Multiple of these can now be used to spawn the three enemy types, as well as the new health pickup.
- ❏ The health pickup behaves similarly to other scrolling objects, but destroys itself upon leaving the bottom of the screen rather than looping to the top.
- ❏ If the player collects the health pickup, their health will be restored to 100

Part 8: Sounds

Waiting until the very end to add sound effects & music was a deliberate choice. Sound files can add a significant amount of compile time when they are in a project. Hence, you should wait as long as you can afford to when implementing sounds!

1. We have two sound effects, *snd_explosion1* & *snd_explosion2*, as well as *Richard Wagner's Ride of the Valkyries* as background music. Import all of these sounds.
    a. You may want to adjust the volume of these sounds once they're imported - they are very loud.
2. You can use audio_play_sound() or the **Play Audio** action to play sound effects & music.
3. Implement these three sounds as follows:
    a. *Ride of the Valkyries* should play when the game starts. There are a number of objects active at the start of the game, but we recommend using the Scoreboard. Make sure that the music doesn't play over itself when the player dies! Either stop the music on death and start over or keep the music playing, but prevent the song from starting over from the beginning. If you restart the room, sounds will persist, but if you restart the entire game then all sounds will stop.
    b. *snd_explosion1* should play when an enemy explosion is created, and *snd_explosion2* should play when the player's explosion is created.

Part 8 Checklist

- ❏ Background music is played when the game is running
- ❏ An explosion sound is played when an enemy plane is destroyed
- ❏ An explosion sound is played when the player is destroyed

---

Final Checklist

Make sure each of these features appear in your submitted game!

- ❑ The Player
  - ❑ The player moves left & right at a fixed rate
  - ❑ The player accelerates & decelerates in the up & down directions
  - ❑ The player cannot move backwards faster than the background scrolls
  - ❑ When up & down are not pressed, the player's vertical momentum decelerates back to 0
  - ❑ The player's full sprite will always remain entirely in view
  - ❑ While the spacebar is held down, the player fires bullets at regular intervals
  - ❑ The player starts with 100 health. If that health reaches 0, the player is destroyed
- ❑ Player bullets
  - ❑ They travel upwards & are destroyed if they leave the room
  - ❑ On collision with an enemy, both objects are destroyed and points are awarded
- ❑ Player Explosion
  - ❑ Plays a very loud explosion sound
  - ❑ When the animation ends, the object is destroyed and the game is reset
- ❑ Scoreboard
  - ❑ Renders the player's health bar & score
  - ❑ Plays *Ride of the Valkyries* when the game starts
- ❑ Islands
  - ❑ Scroll down the screen at a rate equal to the background scrolling speed
  - ❑ When they leave the bottom of the screen, their horizontal position is randomized and they reappear at the top
- ❑ Enemies (general)
  - ❑ Spawns at a random horizontal position
  - ❑ Scrolls down the screen at a configurable rate faster than the background
  - ❑ When they leave the bottom of the screen, their horizontal position is randomized and they reappear at the top
  - ❑ If they collide with the player, they are destroyed and the player takes 30 damage
  - ❑ When destroyed, they create an explosion

- ❏ Enemies (shooters)
  - ❏ Creates a bullet object at regular intervals
  - ❏ the type of bullet and duration of interval can be configured for different enemy types
- ❏ Enemy bullets (straight)
  - ❏ Travels downwards & are destroyed if they leave the room
  - ❏ On collision with the player, the bullet is destroyed and the player takes 5 damage
- ❏ Enemy bullets (aimed)
  - ❏ Are fired in the player's direction & are destroyed if they leave the room
  - ❏ Direction is limited to 30 degrees clockwise/counter-clockwise from directly downwards
  - ❏ On collision with the player, the bullet is destroyed and the player take 5 damage
- ❏ Enemy explosion
  - ❏ Plays a moderately loud explosion sound
  - ❏ Destroys itself once its animation ends
- ❏ Health pickups
  - ❏ Spawn in a random horizontal position above the screen
  - ❏ Scroll downwards & are destroyed once they leave the bottom of the screen
  - ❏ Can be collected by the player to reset their health to 100
- ❏ Spawners
  - ❏ Create an object at a given interval
  - ❏ Object type and interval length can be configured
  - ❏ Are used to spawn 3 enemy types and health pickups
- ❏ Sounds
  - ❏ Background music is played when the game is running
  - ❏ An explosion sound is played when an enemy plane is destroyed
  - ❏ An explosion sound is played when the player is destroyed