

Introduction to GameMaker: Workshop 1

ITCS 4230/5230

Learning Outcomes

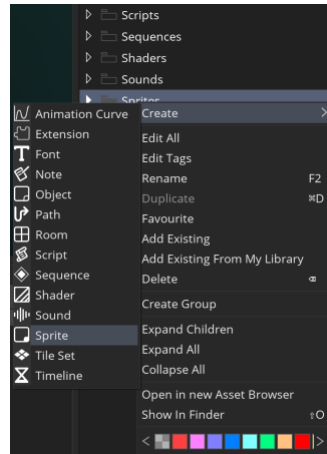
By the end of the workshop students will be able to:

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

Graduate students (only): Please note that there are **additional features** that you need to implement. **See section 9** at the end of this document.

Prologue: Project Setup

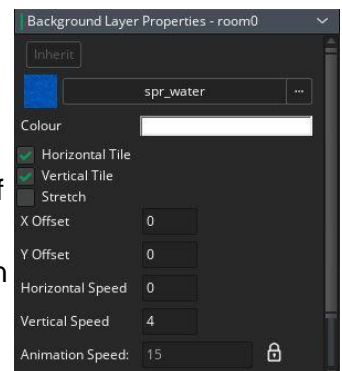
1. Create a new game project in *GameMaker*.
 - a. Choose **GameMaker Language (GML)**
 - b. Name your project *ScrollingShooter_studentid*, where *studentid* should be replaced with your UNC Charlotte "800" number.
 - c. Make sure to save your files in a place where you can easily find them. We recommend a folder dedicated to this course, which is in your *Documents* folder.
2. Download the **Game Resources** zip file from *Canvas*.
 - a. Unzip the file.
 - b. Copy the file's contents to the folder created by GameMaker for your project. Put them in a folder named *Scrolling_Shooter_Resources*
3. Create sprites for the player plane and for the background
 - a. Navigate to the **Assets** 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 sprite 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 sub images, horizontally. Since the width of the image is divisible by 3, this will be successful.



- ii. Click on the *Play* button to watch the propeller animation.
 - 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 sub images.
 - f. Save your project.
 - g. You should **read more about the Sprite editor at the [YoYo Games website](#)**.
4. There should already be a single room present in the **Assets** panel, **Room1**. Double-clicking on 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 focus on 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 there 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 a new object named *obj_player*, the player object.
 - a. For now, assign *spr_plane* to the player object. We will work on making it functional shortly.
 - b. You can now **add an instance of *obj_player***
 - i. **Select** the *Instances* layer.
 - ii. Drag and drop ***obj_player*** from the Assets panel into *Room1*.
 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 and Drop & Game Maker Language

GameMaker provides two options to implement games: A visual scripting interface, and a programming language (GML) loosely based on C/C++. YoYo Games provides a good explanation of each in their [documentation](#). This documentation is available both in a web browser and through the development environment. When working through this workshop, if a function or action is unclear, we highly recommend checking the documentation.

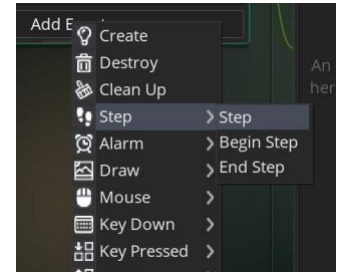
Instructions for this workshop will use GML code. Whenever possible, hyperlinks to the documentation are included in the instructions.

Part 1: Player Movement

1-A: Keyboard Controls

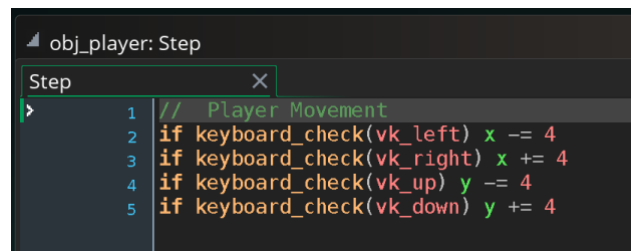
Stop the game and return to `obj_player`. We can add **events** to this object that will react to specific inputs and other occurrences, such as pressing keys on the keyboard.

To implement basic keyboard controls, we will use the **Step Event**. The step event runs once every time the game is updated (by default, that is 30 times per second). Using GML, we can group all of the keyboard inputs into one place in the step event.



1. Add the **Step** event to `obj_player`.
2. The function `keyboard_check()` returns `true` if the specified key is pressed.
3. Moving to the left is done the same way we manipulate number variables in other programming languages. For example, `x -= 4` moves the object 4 pixels to the left.
4. The code 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. Note that in GameMaker the values on the y-axis start at zero at the top and increase as we move down the screen.



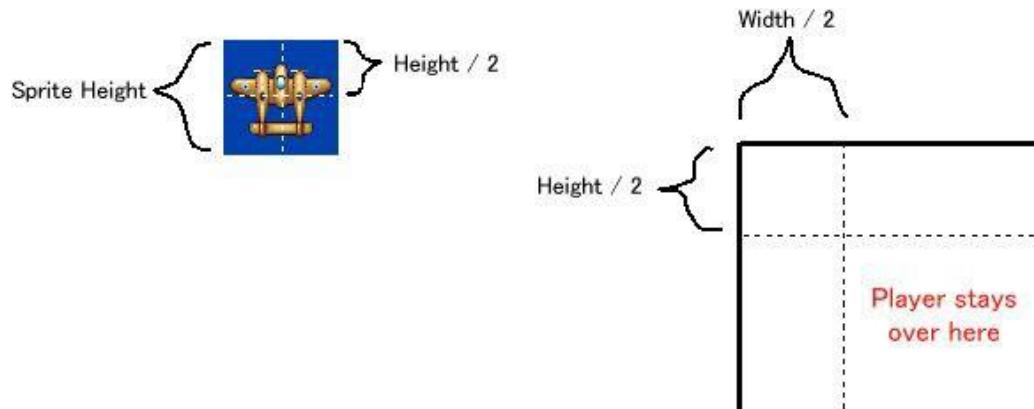
You should now have a player object that moves up, down, left & right with your keyboard inputs. **Save, run and test your game.**

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. Start by setting the **sprite origin** of the player sprite to *Middle Centre* ([check here](#) for documentation on 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. If the sprite's origin is centered, 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. We can use this distance to constrain the player's (x, y) position and prevent them from going beyond the edge of the screen, as follows:



- a. We will use the following built-in variables: `sprite width`, `sprite height`, `room width`, and `room height`.
 - b. To limit the player's (x, y) position, we can use the `clamp()` function.
 - c. The left bounds are `sprite_width / 2`, and the right bounds are `room_width - (sprite_width / 2)`. Hence, limiting the player's horizontal position can be achieved with:


```
x = clamp(x, sprite_width/2, room_width-sprite_width/2)
```
 - d. You will need similar code for the Y-axis
3. The recommended place to implement this code is in the **Step Event**.
 - a. This way, the game will check that the player does not leave the bounds we define each time an in-game update occurs (30 times per second by default).
 4. Run the game. The player should now be unable to leave the screen.

When typing function or variable names, GameMaker will provide [intellisense](#). You can choose a variable or function name from the list that is displayed in the code editor, instead of typing the entire word. This can help reduce syntax errors in your project.

Events, Actions and Instances can be moved around using *cut-and-paste*.

1-C: Vertical Momentum

Next we will modify how vertical movement works. Instead of moving at a static 4 pixels up or down at a time, we will use a system of acceleration and deceleration. This can be achieved with the built-in variable `vspeed`, which changes an object's y position every game update.

1. Modify the code in the **step event** to set `vspeed -= 1` to move upward and set `vspeed += 1` to move downward, instead of directly setting the value of y .

2. 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.
3. You should notice two problems when you run the game. First, the player quickly reaches speeds that make maneuvering difficult, and it is equally difficult to come to a complete stop, because **vspeed** is allowed to increase or decrease indefinitely with no limit so long as the key is held down. Second, **vspeed** only changes when the player is holding a key; therefore, the plane will not come to a stop on its own.

4. We can solve 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 **min** value sets the highest upward speed, this is how fast the player can move **forward**. Play around with different values for this, but remember that this value needs to be negative because **-y is up**.
- b. We need to do something different for downward momentum. Most planes cannot move *backward*, so we can limit their maximum speed to the background scrolling speed. We can use the `layer_get_vspeed()` function, with the background layer's name as the parameter, to get this value.
- c. Given a negative number **min_val**, which equals the highest upward speed you have chosen, the code to clamp vertical speed is:

```
vspeed = clamp(vspeed, min_val, layer_get_vspeed("Background"))
```

- d. This should be done **in the Step event**.
5. To address the second problem, the player's **vspeed** should decelerate back to **0** when neither up **nor** down is not pressed.

- a. We can use the `keyboard_check()` function, as follows:
 - i. This needs to be done each game update by using the **Step** event.
 - ii. We need to check for both keys, Up and Down, as shown below.
- b. When the plane is moving upward, add **1** to **vspeed** to decelerate. When moving downward, subtract **1**. To accomplish this, **subtract the value returned by `sign()` of `vspeed`**. As long as your **vspeed** only consists of whole numbers (no decimals), this will consistently bring it down to **0**.

```
// If neither UP or Down is pressed, slow down
if not keyboard_check(vk_down) and not keyboard_check(vk_up)
  vspeed -= sign(vspeed)
```

Part 1 Checklist

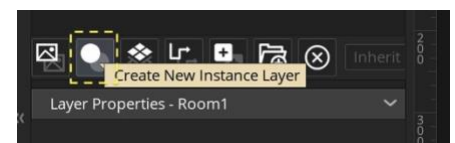
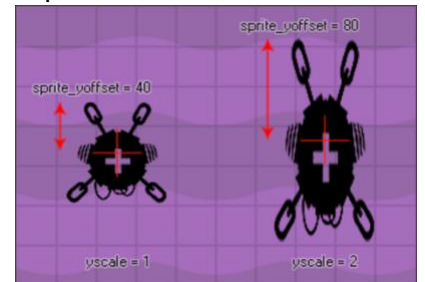
- ☐ The player's plane can be moved around with the arrow keys
 - ☐ The player will always remain entirely on screen
 - ☐ Moving up and down involves accelerating & decelerating
 - ☐ The player has a max vertical speed in each direction
 - ☐ Downward vertical speed does not exceed the scrolling speed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

Part 2: Islands

In this part we are going to create island objects that scroll down with the background and reappear at the top of the screen after scrolling off the bottom.

2-A: Screen Wrapping

- Start by importing the *island1*, *island2* & *island3* images from the resources folder as sprites.
 - Center their origins like we did with the player.
 - Name them *spr_island1*, *spr_island2* and *spr_island3* respectively.
- Make an object out of *island1*, named *obj_island1*. To make it scroll with the background, set its **vspeed** once to match the background speed, 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.
 - We can use the **layer_get_vspeed()** method to get the speed of the background.
 - The vertical speed can be set by setting the value of the **vspeed** variable.
- To determine when the island moves off of the bottom of the screen, we can use a method similar to how we clamped the player's position. In this case, check that the island's sprite has entirely left the visible screen. Do this in the island's **Step event**.
 - You can start with the statement **if y > room_height then y = 0**. However, we will improve it later.
 - Add a couple of islands to the room's **Instances** layer.
 - Save and run your game.
 - Notice that the islands move from the bottom to the top of the screen while it is still visible, which looks a bit messy.
 - To address this, we can use the **sprite_yoffset** built-in variable, which returns the distance between the **y** value at the origin of the sprite and the **y** value at the top of the sprite. We can use this value in the code to ensure the jump from bottom to top is not seen on screen.
 - room_height + sprite_yoffset** is just below the bottom of the screen, while **room_height - sprite_yoffset** is just above the top (note that this only works if 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.
- Note 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 layer. (Likely between that layer & the background



layer). To do this, create a new layer using one of the buttons just below the **Layers** pane. Layers can be reordered by dragging them to the desired location.

2-B: Random Variation

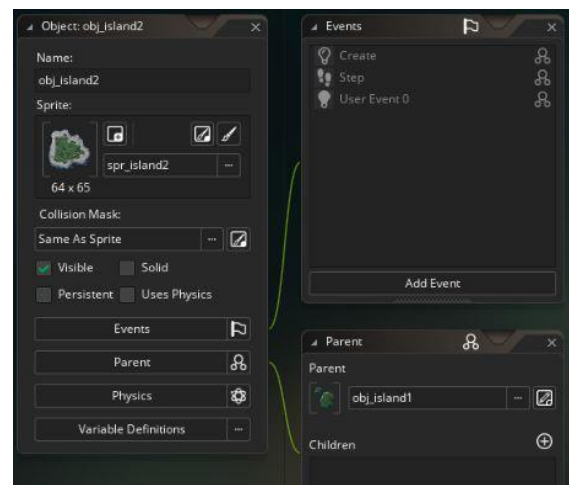
1. When you run the game, note that the island appears at the same horizontal position each time. When we reset the island to the top of the screen, we can randomize its **x** position to add some variation.
 - a. We can use the [irandom_range\(\)](#) function to generate a random whole number between a min & max value. This allows us to randomize the position while still remaining within the horizontal bounds of the screen.
 - b. Since we are already working with offsets, we will use [sprite_xoffset](#). This is similar to how we used [sprite_yoffset](#) to loop seamlessly from the bottom to the top of the screen, we can use this built-in variable to set 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 using the value returned by [irandom_range\(\)](#).
2. Run and test your game. You should see the islands' position change.

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, 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 dialog to set both new islands' parent to **obj_island1**. Notice that they inherit island 1's events.
 - d. Add a few instances of these new islands to the room.
2. Run and test your game.



Part 2 Checklist

- ☐ Island objects scroll downward with the background
 - ☐ When an island fully leaves the screen, it reappears at the top of the screen.
 - ☐ When this happens, the island's 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 islands, two of which inherit the behavior of the first.
-

Part 3: Enemy Planes

Next, we will add a challenge to the game. Enemy planes will 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.
 - a. Make sure to center the sprite's origin.
 - b. Set this object be a child of *obj_island1* and name it *obj_enemy1*.
2. Next, we will make some changes to enemy 1. It makes more sense for the enemy to move downward faster than the background scrolls, so we need 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 will 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 **GML** *inherit* will use an **event_inherited()** statement by default to call the parent's event code.
 - 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 afterward, to ensure that no matter what speed the background scrolls at, the enemy planes will always move faster.
3. Go ahead and add a couple of enemies to the room's **Instances** layer.
4. Run and test your game.

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

3-B: Player Collision

1. The planes move down and explode on contact with the player. To implement the explosion, add a **Collision event** in *obj_enemy1*, add that occurs against the player. This event will trigger whenever an enemy touches the player.
2. In this event, destroy the enemy on contact with the player, using the **instance_destroy()** function. (Click on the hyperlink for more details.)
3. To make it easier to test and debug the enemy's behavior. We will implement a quick way to reset the game, as follows:
 - a. Add a **Key Pressed** (not Key Down) event for some key of your choosing that restarts the game (e.g., 'R'). The function **game_restart()** provides the necessary functionality.
 - b. This event can be added to any object that will still exist after the collision. The *obj_player* object is a good choice.
 - c. Run and test your game, after adding the Collision event.

3-C: Explosion Animation

1. Right now, the enemy's destruction is rather underwhelming; however, we have an explosion resource we could use for this. `Import explosion1_strip6` and create an object named `obj_explosion1` for it. *Don't forget to set the origin.*
2. The explosion sprite has a clear start & end and isn't intended to loop. Therefore, we need to get rid of it once the animation has concluded. `We can do this in the Animation End event.` In the event code, destroy the explosion object.
3. Now we need a way to create an explosion whenever an enemy is destroyed. `We can do this in the Destroy event`, which occurs when the object it is attached to is destroyed.
 - a. In `obj_enemy1`'s **Destroy** event, use the `instance create layer()` function to create an instance of the `obj_explosion1` object at the enemy's position `(x, y)`.
 - b. Do this on the **Instances** layer (the same as the enemy plane).
 - c. Use the enemy object's location for the position of the explosion.
4. Run and test your game. *Make sure to run into some enemy planes.*

Part 3 Checklist

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

Part 4: Health & Damage

Currently, there are no consequences for colliding with another plane. To address this, we need to implement **health**. The player will take damage every time they collide with a plane, and if their health reaches 0 they will 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**. Since they are global, every object has access to the same data. In this game, we will use **instance variables** instead. Data stored in instance variables can only be accessed by the object instance in which they are declared. For example, *obj_enemy1* had an instance variable named **hp**, two instances of *obj_enemy1* would possess unique values for that variable.
3. Usually, we define instance variables in the object's **Create** event.
4. When accessing/changing an instance variable in a different object, we will need to preface the variable with the name of object to which it belongs. For example, to access the player's **hp**, we can write **obj_player.hp**
5. Additionally, **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 the player's health after that point, the game will crash**. In the context of the enemy's collision event with the player, this will not be an issue, as we can assume the player exists if we are colliding with them. However, in other situations we will need additional code to make sure that the instance exists.

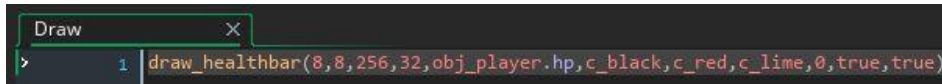
4-B: Inflicting Damage

1. In the player's **Create** event, **declare a health variable and initialize it 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. Make sure to do this before destroying the enemy instance.
3. Go ahead and test this out. To deal enough damage for the player to be destroyed, you will need to place at least 4 enemies in the room to collide with.

4-C: Displaying Health

1. Currently, there is 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, name it **obj_scoreboard**. This object doesn't need its own sprite, but you need to place one somewhere in the room.
3. *obj_scoreboard* will use a **Draw** event to display the health bar. **Any code that renders graphics must be implemented in a Draw event; otherwise, it does not nothing**.

4. Refer to the following screenshot for the code to implement a health bar.



```

Draw
> 1 draw_healthbar(8,8,256,32,obj_player.hp,c_black,c_red,c_lime,0,true,true)

```

5. Recall our earlier discussion about scope. Because the player's health value is an instance variable, we need to make sure an instance of the `obj_player` object exists whenever we try to access it. Currently, the scoreboard code assumes the player exists, which is not always guaranteed. If any code tries to access an instance that does not exist, the game will likely crash.
 - a. We need code that checks if the player exists before drawing the health bar.
 - b. Use the `instance_exists()` function to address this, with `obj_player` as the input parameter.
6. Run and test your game. If the health bar does not appear, make sure that you added an instance of `obj_scoreboard` to the room.

4-D: Exploding & Restarting

1. Currently, once the player runs out of health they just disappear and the game keeps running without them. We need to create a new explosion object for the player to use, which will deal with this problem.
2. Import `explosion2_strip7` and create an object named `obj_playerExplosion`. This object 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`.
5. Run and test your game.

Part 4 Checklist

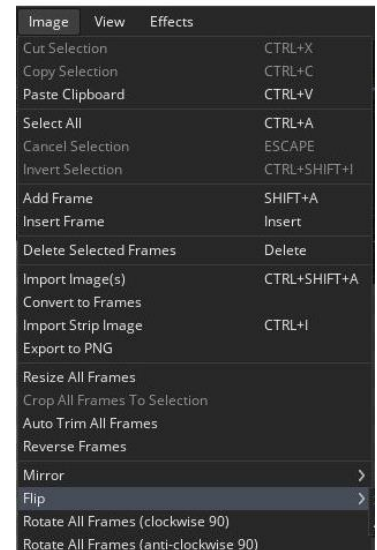
- ☐ The player has 100 health points, displayed in a health bar
- ☐ Crashing into an enemy plane deals 30 damage to the player
- ☐ When the player runs out of health, they explode (with a bigger explosion than the enemies). Once this explosion concludes, the game automatically restarts

Part 5: Combat & Score

In this part, we give the player the ability to fire bullets that can destroy 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 use it to create an object, note that the sprite is upside down.
 - a. With the sprite open, clicking **Edit Image** will bring you to the engine's built-in sprite editor.
 - b. From here select **Image** → **Flip** on the top menu to rotate the sprite so that it is correctly pointing upwards.



2. Next let's make an object named **obj_playerBullet**, which does the following:
 - a. Moves upward.
 - b. Upon collision with an enemy, it destroys both objects.
 - c. We need the bullet to destroy itself once it leaves the room. GameMaker has a special event for this named **Outside Room**. *For more details on this and other events, see the [GameMaker manual](#).*

5-B: Firing projectiles

1. To fire bullets, we will implement a control scheme in **obj_player** as follows:
 - a. **As long as the spacebar is held, fire bullets at a regular interval.** There needs to be a delay between shots. **Players cannot fire bullets continuously.**
 - b. We need to define when the player can and cannot fire and how often.
2. First, declare an instance variable named **canShoot** in **obj_player**, initialised to **true**.
3. Next, add code to check whether **canShoot == true** and the **spacebar** is pressed. If **true**, create an instance of **obj_playerBullet** at the same location as the plane and set **canShoot = false**. Do this in either the **Step** event or in a **Key Down** event.
4. Run and test the game. *Notice that we can only fire one bullet. We will fix that next.*
 - a. We need to set **canShoot** back to **true** after a given time interval. To do this, we can use an **Alarm**.
 - b. The example code provided in the GameMaker manual is very close to what we are 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 game will trigger a corresponding alarm event. When the Alarm event fires, we can reset **canShoot** back to **true**.

- c. Make sure to add an **Alarm** event to `obj_player`. In it set `canShoot = true`.
5. The player should fire 2 bullets per sec. (approx.) when the spacebar is held down.
 - a. If you set the alarm to a different value, the player 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 is to use the `game_get_speed()` built in function to get the game's update rate in frames per second. To fire at a rate of twice per second (approx.), set the alarm countdown to `game_get_speed(gamespeed_fps) / 2`. This will also make the code easier to understand and maintain.
 - c. We can manually trigger an alarm (or any event). *For more details on how to do this, see the `event_perform()` function in the GameMaker manual. (Note: setting an alarm to 0 will not trigger it instantly, instead it will not trigger it at all.)*
6. Run and test the game.

5-C: Scoring Points

1. The implementation of **Score** will be similar to what we did for health, i.e., instead of using the built in global variable we will define our own.
2. It is important to use an object that is always available in the game. In this game, we can use the **`obj_scoreboard`** object to store and maintain the player's score.
3. Start by drawing the score on the scoreboard. If you used our previous set of coordinates, this should fit in nicely with it. Note that when using the `draw_text()` function it is necessary to manually convert non-character values to a string.

```
//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))
```

4. We can update the score in the **Collision** event of the `obj_playerBullet` object. Whenever the player destroys an enemy with a bullet, we add to the score (e.g., +10 points), while the same will not occur if the player & enemy crash into each other.

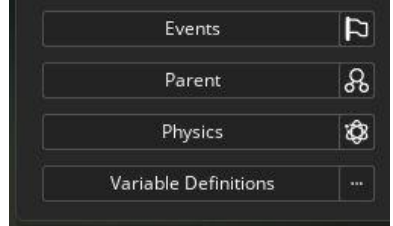
Part 5 Checklist

- ☐ 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 is 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.

6-A: Variable Definitions

1. You may have seen the **Variable Definitions** button in the object window, under **Events** and **Parent**. This dialog allows us to declare and initialize variables for later use in **GML** code. The immediate difference here is that they are 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 instead of using a hard-coded value 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. Next, we will modify the 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. Since the *scoreValue* will depend on the type of enemy that was destroyed, **we need to get such the value from the specific enemy the bullet collided with**. In collision events, the **other** keyword can be used to get the specific object instance involved in the collision, as follows:
`score += other.scoreValue`
4. Run and test your game after making these changes.

6-B: Orange Enemy Plane

1. Now that the setup is done, let's create a second enemy type. **Import** *enemy2_strip3.png*, and create a new object that is a child of *obj_enemy1*.
2. Change the values of the Variable Definitions to distinguish the enemies, as follows:
 - a. Enemy 1 will move faster, and is worth the default point value.
 - b. Enemy 2 will move slower, and is worth more points. To do this, click on the pencil icon to override the variable's values.
3. Add some instances of enemy 2 to the room.
4. Run and test the game.
5. Note that 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 downward and self-destructs on contact with the player and when it is outside the room.
2. Every time we want to inflict damage on the player, we need to do the following:
 - a. Decrease player health by a given amount.
 - b. Check player health's new value.
 - c. If player health ≤ 0 , destroy the player.
3. It would be better to have a user-defined function we could call to inflict damage and do the related health check afterwards. [We can do this with a Script](#).
 - a. Create a new script, with a descriptive name, such as `scr_damage`.
 - b. Add a function named `inflict_damage`.
 - c. Move your existing damage code from the enemy1 **Collision** event into this script function.
 - d. The function should have one argument, named `damage`.
 - e. Your existing code should have decreased player health by 30. Replace this number with `damage` to use the input given when the script function is called.
 - f. In the enemy1 collision event, call the `inflict_damage` script function instead of running the damage code: `inflict_damage(30)`
 - g. Run and test your game. *Make sure the game still works like it used to.*
4. Next, call the `inflict_damage` script function in the enemy bullet's collision event to inflict 5 worth of damage: `inflict_damage(5)`
5. Make sure to do this before destroying the instance of the bullet.

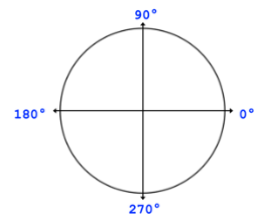
6-D: Orange Enemy Plane Fires Bullet

1. The firing mechanism we will use is similar to how the player character shoots. However, in this case, **we are going to set up a looping alarm**, and fire a bullet every time the alarm goes off. For an alarm to run continuously, we need to reset the alarm when the alarm's event occurs. **In the Alarm 0 Event, set Alarm 0 to 15**, etc. You can set the first alarm for the loop in the **Create Event**. *Remember that to inherit a parent's event, we will use `event_inherited()` in the code.*
2. We can use **Variable Definitions** in `obj_enemy2` to allow for more variation:
 - a. Add a `shotInterval` variable (type Integer) that defines what the enemy's alarm is set to each time. Set the default value to 30.
 - b. Add a `shotType` variable, set the type to **Asset** and the **Resource filter** (click on the gear icon) to **restrict it to game objects**. Default to `obj_enemyBullet`. We will use this variable to set the type of bullet object fired.
3. With these variable definitions, we can have enemies that fire more/less often, and with different types of bullets.
4. To practice the skills you have been working on, figuring out the complete **GML** code to implement the functionality to fire bullets is left as an exercise to the reader. **Make**

sure to do this and test your game **before** proceeding.

6-E: White Enemy Plane

1. Create a third enemy type based on `obj_enemy2`. Import `enemy3_strip3` and create a new object that is a child of `obj_enemy2`. This enemy will be the slowest, worth the most points, and fire bullets aimed in the player's direction, rather than straight down.
 - a. 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 that aims 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, set the `direction` of the bullet using the `point direction()` built-in function. The parameters used will be the `(x, y)` coordinates of the player.
 - a. Note that if one of these bullets is created when 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 the direction manually.
 - b. Direction is a built-in variable that uses degrees and rotates counter-clockwise, with 0 pointing right, 90 pointing up, 180 pointing left & 270 pointing down. When the player doesn't exist, set direction to 270 to send the bullet downward.
4. Setting direction alone will not cause the bullet to move. You also need to set the bullet's `speed`. Do not use `hspeed` or `vspeed` in this code.
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 could be disorienting. Since a direction of 270 is directly downward, clamp the bullet's direction between 240 and 300.
6. Run and test your game to make sure `obj_enemy3` is correctly firing its bullets.



Part 6 Checklist

- ☐ There are now 2 new enemy types
- ☐ Enemy 2 shoots bullets straight downward, 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 has 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 is something we can do with our enemies first. Ideally, when creating new enemies, we do not want them to immediately pop onto the screen. We need them to scroll onto the screen from the top, much like they do after they leave the screen from the bottom. We should reuse the code that places them at a random horizontal position at the top of the screen, as follows:
 - a. Open the **original island object**, take the code that modifies its *x and y* position when it leaves the screen, and make a script out of it.
 - b. If you call this script in the create event of *obj_enemy1*, other enemies will inherit the functionality so all the spawner has to do is create them. They will reposition themselves so they naturally enter the screen.
2. For the spawner, **we will use a setup similar to how enemy2 is set up to fire bullets**. Set up *spawnInterval* & *objectType* **Variable Definitions**, and use them to implement an alarm loop that creates the desired object.
 - a. Spawn the enemy instance at a random *x* coordinate.
 - b. 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 arrives from the top of the screen.
3. When an individual instance of an object is placed in a room, the values in variable definitions are set to the values already defined inside of the object. However, 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 *obj_enemy2* that fires 15 times per second by placing it in the room and editing its variable definitions.
 - a. This way, you can drop 3 spawners into the room, then configure them to spawn *enemy1*, *enemy2* & *enemy3* respectively, with different spawn intervals.
4. Double-click on the spawner object (question mark) to get a dialog box where the variables can be set.
5. 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, we can easily 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 is no need for it to inherit from other objects, i.e., it does not have a parent object.
 - a. In the create event, use the 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. Next, add an additional spawner configured to create health pickups. They are 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 instances of this object are 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 and 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 in the game resources folder, *snd_explosion1* & *snd_explosion2*, as well as *Richard Wagner's Ride of the Valkyries* as background music. **Import all of these sounds.**
 - a. **Create a sound object for each one of these.**
 - b. You may want to adjust the volume of these sounds once they are imported (they are very loud).
2. You can use `audio play sound()` to play sound effects & music.
3. Implement the game's background music and sound effects as follows:
 - a. *Ride of the Valkyries* plays 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.
 - c. *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.
-

Part 9: Additional Features

This part is only for the **Graduate students**, i.e., all students registered in **ITCS 5230**.

Undergraduate students do not need to implement any of the features listed on this section.

The following directions are not very specific. This is on purpose, to encourage you to read the documentation and explore more advanced features of GameMaker. You are expected to give it your best effort; however, if you get stuck, please ask questions to the course staff.

9-1: High Score Table

Implement a high score table, similar to the one in the image.

1. The game keeps track of the players with the best 10 scores.
2. Players need to be able to enter their initials or name (your choice).
3. The table needs to be persistent, i.e., the high scores are kept even after the game is shut down.

RANK	SCORE	NAME
1ST	10000	BOB
2ND	10000	JMC
3RD	10000	SKT
4TH	10000	TBS
5TH	10000	MNM
6TH	10000	WKJ
7TH	10000	SVD
8TH	10000	WHO
9TH	10000	TRN
10TH	10000	JMC

9-2: Difficulty Level

Implement a dialog that allows players to choose a difficulty level at the start of the game.

1. At least three levels should be available: *Easy*, *Hard* and *Insane*.
2. Changing the difficulty level affects the following:
 - a. How often enemies are spawned.
 - b. How often extra life power ups appear.
3. How fast bullets fly.
4. Players can only set the difficulty after a game restart.
5. The game displays the dialog when it first starts and after players press the restart key.

9-3: Pause Functionality

Implement a keyboard event that allows players to pause/resume the game at any time by pressing ALT+P.

Part 9 Checklist

- ☐ The game has a persistent high-score table
- ☐ Players can set the game's difficulty level
- ☐ The game can be paused and resumed

Part 9 is for graduate students (only).

Final Checklist

Make sure all the following features exist in the game you submit for grading.

- ☐ 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
- ☐ **Graduate students (only)**
 - ☐ The game has a persistent high-score table
 - ☐ Players can set the game's difficulty level
 - ☐ The game can be paused and resumed

Images used in this document were either created by the course staff or from the GameMaker documentation, found at <https://manual.yoyogames.com/>