Introduction to GameMaker: Workshop 2
ITCS 4230/5230

Learning Outcomes

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

1. Implement the game objects and mechanics necessary to change rooms
2. Write Cheat codes
3. Implement functionality to enable player characters to move objects
4. Use gravity
5. Use tiles to create background effects
6. Implement collisions that take into account continuous movement
7. Apply basic AI concepts such as
    a. Enemies that have a patrol path
    b. Enemies that follow the player
8. Modify an object's appearance to reflect what it is doing

Setup

1. Navigate to the ***GameMaker: Platform Games*** Module in *Canvas*
2. Download the activity file named ***Simple_Platformer.zip*** from the assignment page. This file contains a partially completed game that you will use to learn about platformer games and use as the basis to implement a simple platformer that you will submit.
3. Unzip the file to an appropriate location on your computer.
4. Start GameMaker and open the project (Simple_Platformer).
5. Make sure the game builds and runs.

Introduction

In this workshop, you will be using a partial game. This partial game is the basis to implement a simple platformer. The game you will submit is similar to games like *Super Mario Brothers* or *Metroid*. However, what you will produce is much simpler.

As you look at the project code, keep in mind that GameMaker Studio uses the Cartesian plane for coordinates, as well as degrees (0-360) for most anything labeled “direction.” A quick reference for these is shown below.

Direction:
0 to the right
90 straight upwards
180 to the left
270 straight down

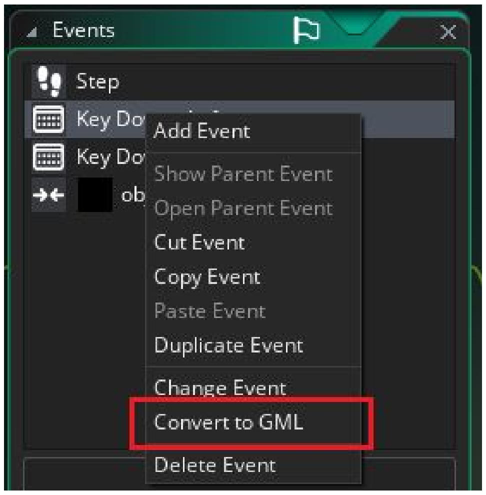
Coordinates:
x+ moves to the right
x- moves to the left
y+ moves down
y- moves up

(Note that since -y is the upward direction, the traversal from 0° to 360° is counter-clockwise.)

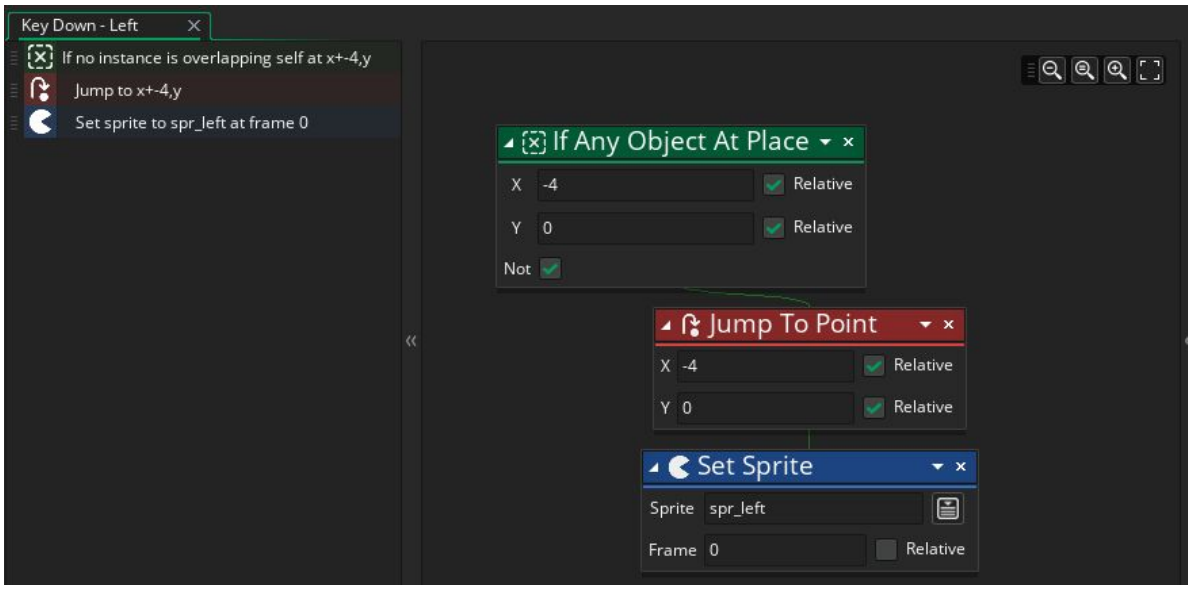
Part 0 (Optional): GameMaker Language

Even though GameMaker's drag and drop (DnD) functionality may be sufficient for many games, you may want to use programming for your group projects. Hence, it is greatly recommended that you familiarize yourself with GameMaker Language (GML).

A good way to get started is to right click an event from any object and select the option, ***Convert to GML***. This will take the Drag & Drop blocks in the event and convert them into code, which gives you a quick & easy look at GML syntax, as well as what functions GameMaker has to implement the functionality you need.



For instance, the Step Event in *obj_character* from *Simple_Platformer* converts from the following Drag and Drop actions:



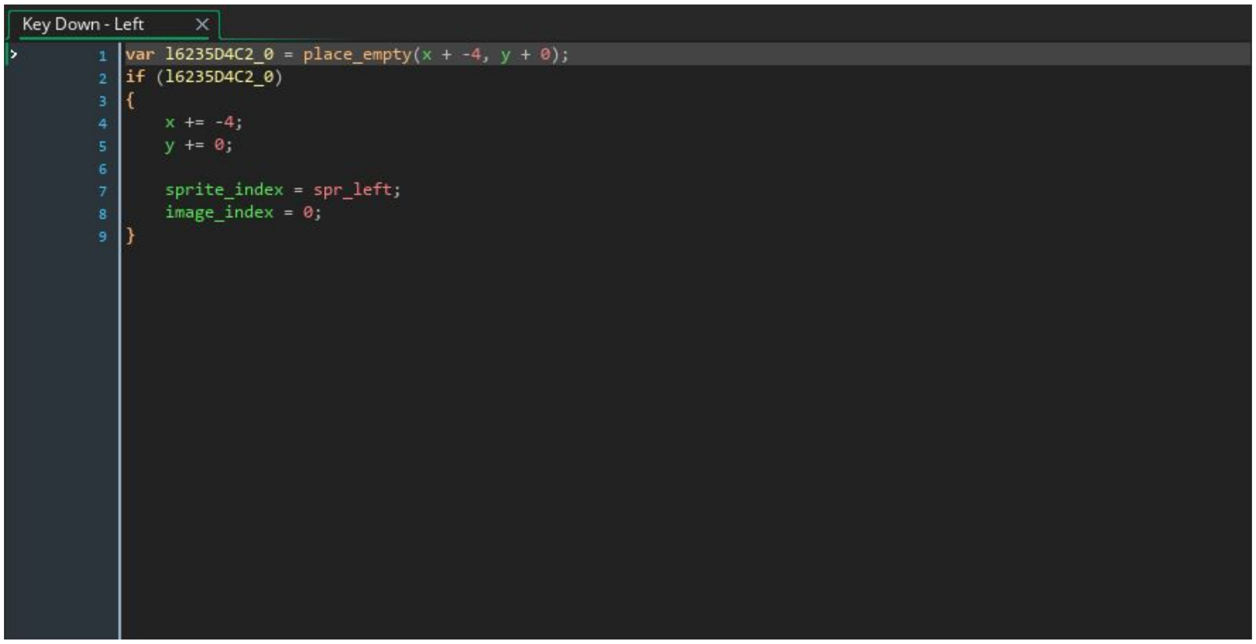
If no instance is overlapping self at x+-4,y
Jump to x+-4,y
Set sprite to spr_left at frame 0

If Any Object At Place
X -4 ✓ Relative
Y 0 ✓ Relative
Not ✓

Jump To Point
X -4 ✓ Relative
Y 0 ✓ Relative

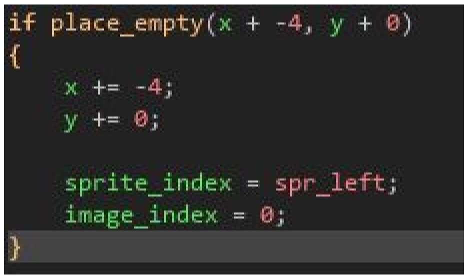
Set Sprite
Sprite spr_left
Frame 0 Relative

To the following GML code:



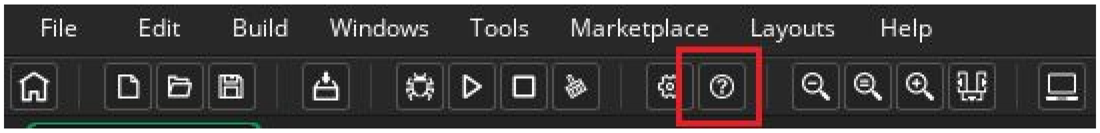
```
var l6235D4C2_0 = place_empty(x + -4, y + 0);
if (l6235D4C2_0)
{
    x += -4;
    y += 0;

    sprite_index = spr_left;
    image_index = 0;
}
```

Note that the variable name produced at line 1 is merely a consequence of how GameMaker converts from Drag & Drop to GML. The code can be modified to improve clarity and maintainability, as shown below:



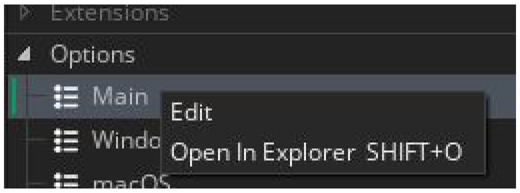
```
if place_empty(x + -4, y + 0)
{
    x += -4;
    y += 0;

    sprite_index = spr_left;
    image_index = 0;
}
```

For these GameMaker Workshops, your only requirement is that your submitted game has all of the functionality that has been described. How you create that functionality is entirely up to you (though, following the instructions certainly helps). You can use Drag & Drop, GML or even a combination of the two. If you do choose to work with GML, it's recommended that you use this "Convert to GML" technique to get started and refer to GameMaker's built-in manual. The manual can be accessed via ***Help -> Open Manual*** or by clicking the Question Mark icon near the top of the program.

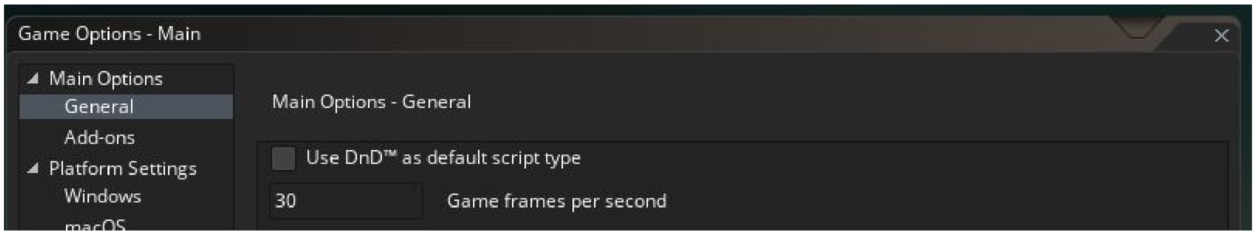


If GameMaker runs slow on your computer, you may have better luck using the online manual, found at https://docs2.yoyogames.com/

You may decide that you are more comfortable with GML overall than you are with Drag & Drop. If you are confident in using GML, another way you can convert to GML is to change **default script type** in the **Options** dialog. In the **Resources** menu, expand **Options**, right click on **Main** and select **Edit**.



From here, **uncheck** ***Use DnD as the default script type***, then select apply. This means that when you add a new event, Game Maker will automatically open the event in GML. Note: This will not automatically convert old Drag & Drop boxes to GML, it will only convert new events.


Game Options - Main

Main Options
General
Add-ons
Platform Settings
Windows
macOS

Main Options - General

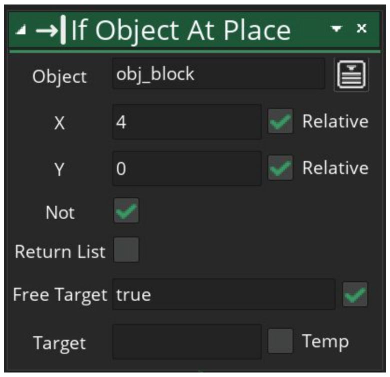
☐ Use DnD™ as default script type

30 Game frames per second

Part 1: Player Movement in a Platform Game

1-A: Walking left & right

- **All the code will be written in *obj_character.***
- All of the sprites needed have been provided.
- Motion in the horizontal direction has already been implemented. Note that the character's sprite always faces right, regardless of the direction of motion. You need to implement the code to correct this, i.e., **the sprite should change to match the direction in which the character is moving**.
- Additionally, we want to make sure that the character can only move if there are no obstacles in its way



If Object At Place

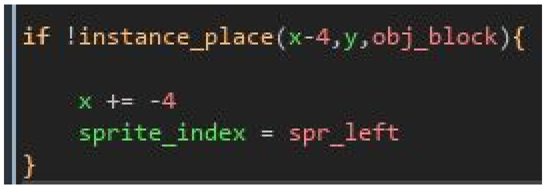
Object	obj_block	
X	4	✓ Relative
Y	0	✓ Relative
Not	✓	
Return List		
Free Target	true	✓
Target		Temp

Method 1: Drag 'n Drop

- Change the *Key Down - Left* and *Key Down Right* events to include a **Set Sprite** action that updates the player's sprite to either ***spr_left*** or ***spr_right*** depending on the direction of motion.
- Use the If Object At Place action to check before updating the value of the character's **x** coordinate. In other words, always check to see if the position is empty before jumping to it!.

Method 2: GML

- Right click the *Key Down - Left* and *Key Down Right* events and select "Convert to GML."
- To change the sprite, Game Maker has a built-in variable called *sprite_index*. This needs to be set equal to either ***spr_left*** or ***spr_right*** depending on the direction of motion.
- Use instance_place() to check before updating the value of the character's **x** coordinate. In other words, always check to see if the position is empty before jumping to it!.



```
if !instance_place(x-4,y,obj_block){

    x += -4
    sprite_index = spr_left
}
```

1-B: Jumping

- Jumping applies vertical momentum - you need to set ***vspeed*** instead of using jump-to-position. (Remember, upward is negative in the y-direction!)
- Run the game and try to jump while still in the air.

- The character should only be able to jump while they are standing on the ground or on a platform. To address this, add code to check if there is ground *directly below* the player `(x+0, y+1, relative)`. This needs to happen before setting the vertical speed.
  - Useful DnD actions: ***Any Object at Place*** or ***Object at Place***
  - Useful GML functions: ***place_meeting()*** or ***instance_place()***
- You may notice that once obj_character jumps it never comes back down…

1-C: Gravity

- **A Crash-Course on Gravity:**
  - ***gravity*** is a built-in variable present in all *GameMaker* objects. It has an accompanying variable, ***gravity_direction***
  - Every game tick, ***gravity*** adds speed in a specific direction, specified by ***gravity_direction***
  - The amount of change enacted by gravity is dependent on ***gravity****'s* value - if an object's ***gravity*** = 0, nothing will happen.
  - **To create a force that pushes obj_character downwards, set *gravity_direction* = 270.**
  - In Drag and Drop, gravity is handled by the **Set Gravity Direction** and **Set Gravity Force** actions
- ***gravity_direction*** only needs to be set once - that can be done in *obj_character*'s create event
- Gravity is usually handled in the ***Step*** event, which runs every frame
- Similar to jumping, we need to check whether there is solid ground below *obj_character*. If there is, ***gravity*** should be 0, otherwise it should be a positive number (0.5 or 1 works pretty well)
- After implementing gravity, you should see that after jumping *obj_character* falls back down, *and then proceeds to fall through the floor…*

1-D: Colliding with walls

- When walking left/right, we avoid colliding with walls entirely using the conditional check; We are working with ***vspeed*** to facilitate jumping, which makes *avoiding* collisions much harder. Instead, we can use a collision event to react when a collision occurs.
- Note that there are 3 wall objects: ***obj_block***, ***obj_blockv*** and ***obj_blockh***.
  - ***obj_blockh*** and ***obj_blockv*** are children of ***obj_block***, so if ***obj_character*** has a collision event that activates with ***obj_block***, it will activate with either of them as well.
- What we need to do here is to set ***vspeed*** to 0 after colliding with a block, and **obj_character** will stop falling once they hit the ground.

- **Additional Note:** if you check **obj_block**, you will see that it is flagged as *solid*. An easy way to understand what this means is that when **obj_character** collides with a block, it will be pushed out (this ONLY happens if the object has a collision event). Using a little more detail, it works like this: GameMaker will detect if the character's sprite overlaps the block's sprite. When this occurs, the character's position will be returned to its original position before the collision event occurs. Say the character was at (0,0) and there was a wall at (5,0). If the character moves horizontally by 10 pixels, GameMaker will detect the collision, then return the player to (0,0). If you turn off solid on **obj_block**, you will see that upon falling back down to the ground, **obj_character** will often embed itself into the ground, making left/right movement impossible.

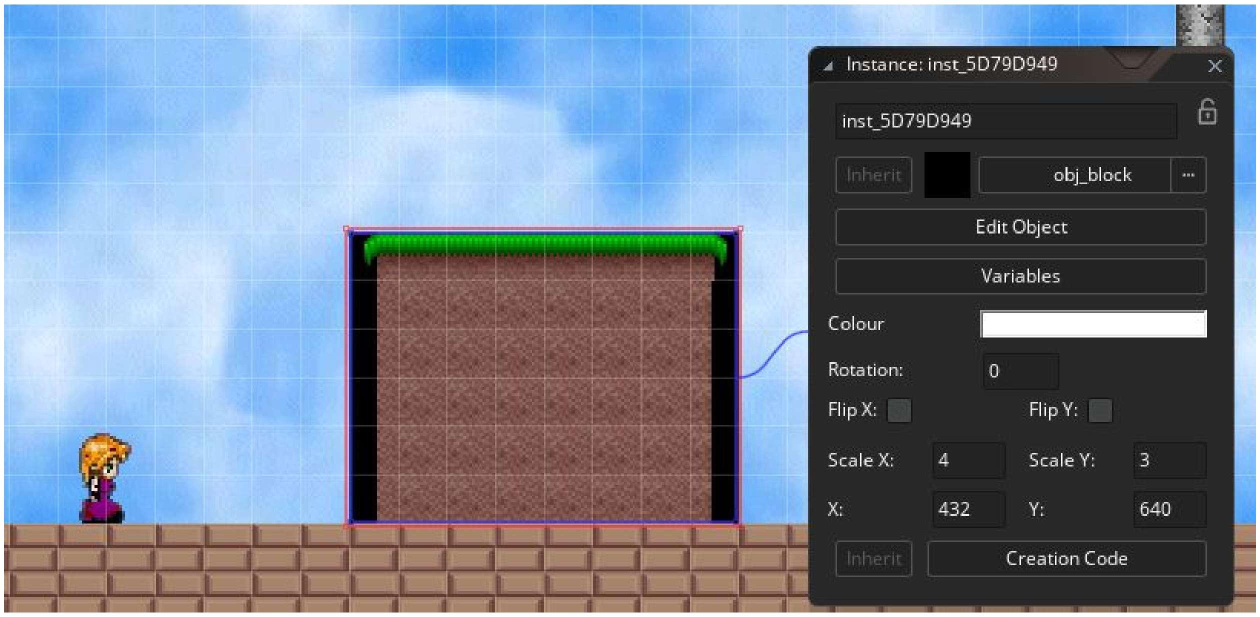
- **Additional Note:** Unfortunately, GameMaker's default collision system has a tendency to produce some unforeseen bugs. Because of this, we will be providing code to circumvent some of these issues at the end of this workshop. The code might be a little more involved that what you are comfortable with right now so implementing it is completely optional, but it might help for future assignments.

1-E: Limiting Vertical Speed

- If ***obj_character*** falls for a long time, their ***vspeed*** can get to be pretty fast. This can create problems in your game; at best, it might prove detrimental to how your character controls midair (a requirement for platformers), while at worst your character might fall so fast they bypass the floor's hitbox entirely! The solution is an upper limit to ***vspeed***.
- To address this, add a conditional statement in the ***Step*** event, which checks if ***vspeed*** is greater than a certain number (12 should work, but we recommend testing).
  - If ***vspeed*** is greater than 12, set it to 12.
  - If you are using GML, GameMaker provides a few handy math functions such as min(), which returns the smallest value out of the ones you provide. It might be easy to see how that could be applied here…

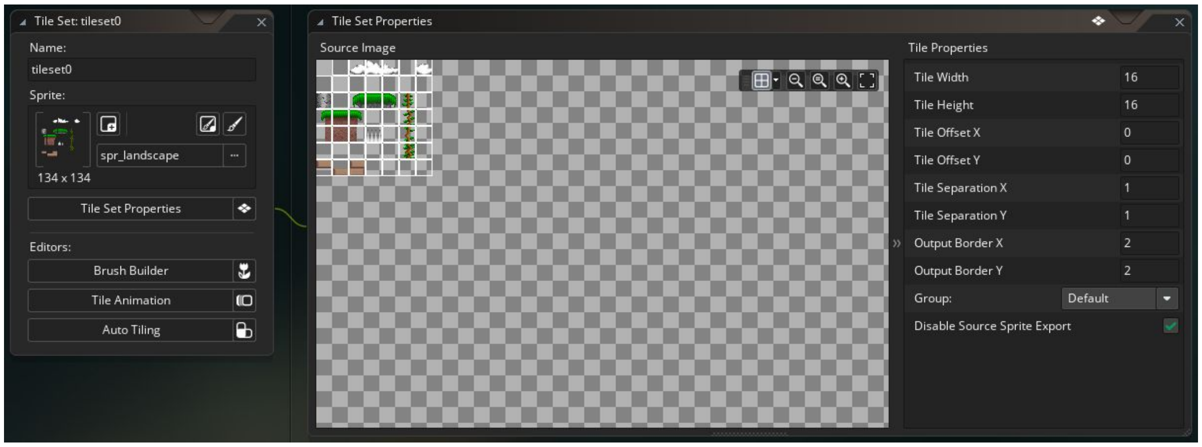
Part 2: Level Creation

- The default size of **obj_block** is determined by its 32x32 sprite. However, GameMaker allows you to resize instances once they are placed in the room editor. This allows you to cover large stretches of level (room) using just one wall object.
  - Scaling an instance in the room editor will alter its builtin variables **image_xscale** and **image_yscale**. Since collisions take these variables into account, colliding with a scaled object functions as usual.
  - These changes to **image_xscale** and **image_yscale** are made before the object's create event is called. You may be able to find some use for this…



inst_5D79D949
obj_block
Colour
Rotation: 0
Flip X:
Flip Y:
Scale X: 4
Scale Y: 3
X: 432
Y: 640

- You may have noticed that while **obj_block** and its children have been used as colliders for our level, they are not visible when the game is running. Their **Visible** property is set to *false*, as we have a *tileset layer* providing the visual aspect of our level.
- *Tilesets in GameMaker* are sprites broken up into equal-size sections, which can be placed on a tileset layer. By default they are purely visual, though there are some advanced functions that can make clever use of them.



Name: tileset0
Sprite: spr_landscape
134 x 134

Tile Properties

Tile Width	16
Tile Height	16
Tile Offset X	0
Tile Offset Y	0
Tile Separation X	1
Tile Separation Y	1
Output Border X	2
Output Border Y	2
Group:	Default
Disable Source Sprite Export	✓

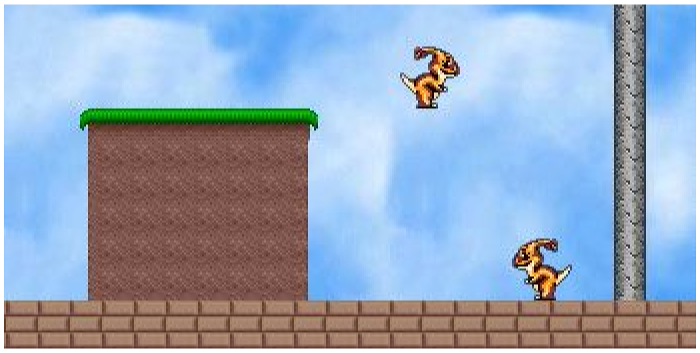
- Open the resource for **tileset0** and examine its properties. The sprite it uses is spr_landscape, and it splits the sprite up into 16x16 tiles. *It is recommended that you read about tilesets' other properties in the GameMaker manual.*
- **Note: The top-left tile in a tileset will always be the "empty" tile. If you make a sprite for the purposes of a tileset, know that any image data in the top-left tile space will not be used.**

Part 3: Enemies

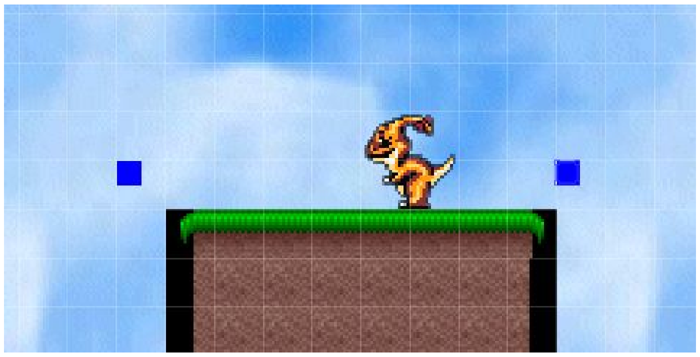
- This part of the project focuses on adding the enemy characters obj_monster & obj_flyer. Initially, we will be taking a look at some rudimentary enemy behaviors, as well as some potential roadblocks when using inheritance. After that we will sprinkle in some additional trinkets and hazards that may appear in a platformer.

3-A: Giving obj_monster Boundaries

- Take a look at **obj_monster** and you should notice that it already has some basic functionality. In its **Create** event, its **hspeed** is set to 2, and upon collision with any of our wall objects, the monster turns around.
    - The effect: our monster walks to the right, and upon hitting a wall it walks to the left. Repeat.

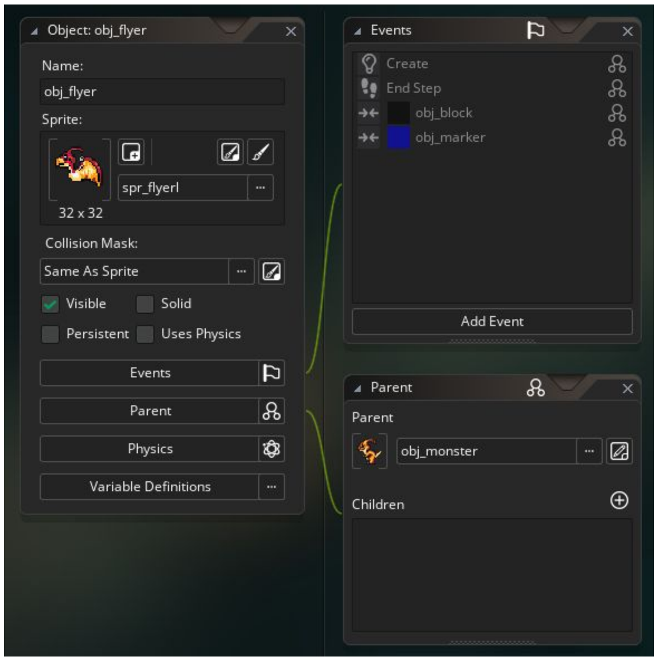


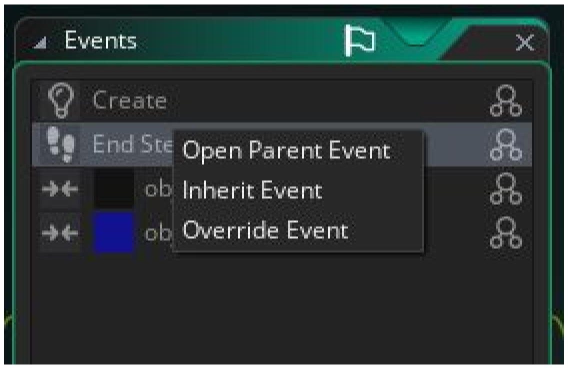
- You will see, however, that nothing stops this monster from walking right off of platforms (and nothing makes them fall down). For now, rather than dealing with gravity we will just prevent **obj_monster** from walking off the platforms.
- A simple fix for this comes in the aptly-named ***obj_marker*** that already exists in the project. This marker is already set to be invisible, so it is perfect to set boundaries.
    - **Duplicate the event that makes our monster turn around on walls, and have it do the same upon collision with *obj_marker*.** *Right-click on the obj_block collision event to bring up the context-sensitive menu.*
    - Place one of these markers at any position where the monster should turn around.



3-B: Creating obj_flyer

- We have in our possession a second enemy character: **obj_flyer**. This flyer should behave much the same as our prior monster, but it is expected that it will be able to fly up in the air without question. Still, it would be *really convenient* if we could just inherit **obj_monster**'s code.
- Much like Workshop 1, we can set our flyer's parent to be **obj_monster**, and it will instantly get (inherit) all the functionality of monster. Run the game, however, and a problem immediately presents itself: all of our flyers are using monster's sprites!
- Check the **End Step** of either enemy. This is where we are setting their sprite, based on their **hspeed**. Since the flyer is inheriting monster's event, the sprites being set are also monster's sprites.
- The most direct way to solve this having **obj_flyer** *override* the **End Step** event (which can be done by right-clicking on the event) with a version that uses its own sprites.


Object: obj_flyer
Name: obj_flyer
Sprite: spr_flyerl
32 x 32
Collision Mask: Same As Sprite
Visible ✓ Solid
Persistent Uses Physics

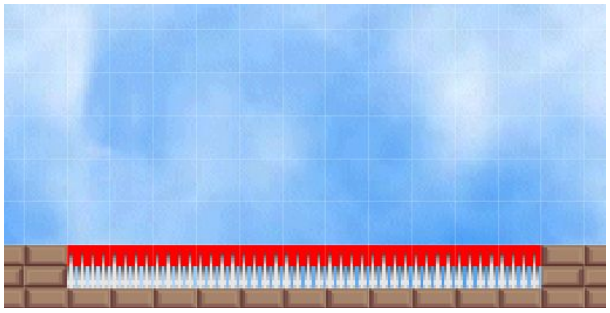


3-C: Squashing Enemies

- If you look in **obj_character**, you will see that we already have code that defeats the player upon contact with **obj_monster** (it plays a sound and restarts the room). We are going to add to this collision event to allow the player to defeat enemies by jumping on them.
- **Let's think about the conditions required to be considered jumping on an enemy**: our character would need to be falling downward (**vspeed > 0**) and be <u>above</u> the enemy.
    - Coincidentally, the collision event provides the reference variable **other**, which provides access to the *other object* involved in the collision. In the case of **obj_character**, **other** provides a reference to the enemy we collided with.
    - Comparing the two instances' vertical positions would then require **y** and **other.y + *some offset*** (e.g. 8). *Note that if using DnD actions, you should use the* ***If Expression*** *action. The syntax allowed in the Expression field is very similar to the expressions you can write in Java or C++, including relational and logical operators.*

- There is already code that plays the player kill sound and restarts the room.
    - Add code that checks our two conditions.
    - If the jump is successful, kill the enemy instead of killing the character.
        - There is an appropriate **snd_kill_monster** resource you can use.
        - Replace the instance of the monster with an instance of **obj_monster_dead**, which is a gruesome depiction of a flattened monster. *Make sure you replace the enemy, not the player.*
        - Similar to the explosions created by the planes in Workshop 1, **obj_monster_dead** displays its sprite briefly, then destroys itself. *Hint: Do this using alarms.*
    - Add 50 points to the player's score.
- Fortunately, since we have **obj_flyer** inherit **obj_monster**, we only have to write this once.
- *Super Mario* doesn't just fall through enemies when jumping on them, he bounces up and off of them. Try setting your character's **vspeed** when successfully landing on an enemy to facilitate that bounce.

3-D: Death Spikes



- There's already one way to kill the player, let's add some more.
- Check the game sprites you should find one named **spr_death**. Similar to the marker object we used previously, this can be used to define an area that kills the player upon contact. The actual death object should be invisible, but it can be given a visual representation using the spikes present in our tileset. Use the trench on the right side of the room for this.
- At this point, we now have two ways in which the player can die.

Recommendations:

- To avoid repeating code, such as the player's death, we should put it in a single event that can be triggered in either case.
    - Putting the code in the **Destroy** event always works, though you could also try using the **User-Defined** events as well (check the manual).
- Our game can be a very dangerous place for the character! To address this, note that we have given our character some **extra lives** to work with. This can be handled with a separate controller object or within the character itself.

Part 3.5: Some Detail and Challenges

Below is a quick reference for the three (3) different types of ***Step*** events. See the GM documentation for full details.

Begin Step occurs at the beginning of each step. (Disregarding create events, room start events, game start events, etc.)

Normal Step occurs after alarms, keyboard events, and mouse events, and before collision events.

End Step occurs after collision events and after all objects are repositioned with *hspeed* and *vspeed*.

The following challenges are optional, but completing them will further your understanding of how different facets of GML can be applied.

- **Challenge 1** - Using Collision-Checking to Detect Ledges
  - Using markers is all well and good, but if your enemy characters can automatically detect when to turn around to avoid ledges, that will save you some work in the room editor. That will add up when your projects get bigger!
  - We already used collision checking to detect if there’s a wall in the direction a character is currently walking, and we have used it to detect if the character is/isn’t standing on ground. *Try using it to detect if there is ground in the direction that a character is walking.*
  - As **obj_monster** will be moving every step, you need to place this check in an event that runs every step.
  - Ensure that **obj_flyer** does not perform this check, however, as it does not care whether or not it is in the air. You will need to override the event in which the check is made.

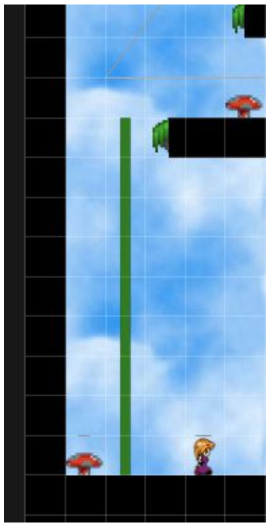
- **Challenge 2** - Using Instance Variables to Keep Track of Sprites
  - Given that monster and flyer both only have two sprites, it was fairly simple to make the change for flyer to use its own sprites. In a larger project, however, a character might have significantly more sprites unique to them, and it could be time-consuming to manually edit each individual change.
  - Do some reading about how variables can be used in GameMaker (the manual is very helpful). Variables can be assigned references to particular resources (including sprites), and the same variable in **obj_monster** could hold a completely different reference in **obj_flyer**. If you had 10 different places where monsters needed to be

assigned **spr_monsterl**, wouldn't it be great if you only had to change a reference variable *once* to assign **spr_flyerl** instead?

Part 4: Bonus Mushrooms

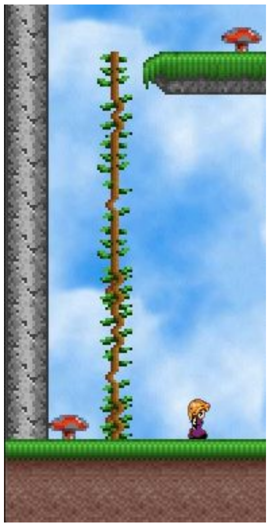
- We need more ways for the player to earn points.
- Add a few mushroom objects (obj_mushroom) to the room.
- Implement functionality to do the following:
    - Collect mushrooms (use collisions).
    - Play a sound when a mushroom is collected.
    - Increase the score when a mushroom is collected.
- Examine the mushroom sprite (spr_mushroom). Note that there are many different mushroom pictures; however, all the mushrooms in our game look the same.
- In the **Create** event of **obj_mushroom**, use Random Number Generation to assign each mushroom a random sprite image.
    - *Alternatively, make a single mushroom sprite with each mushroom as its own frame, then assign the sprite to a random frame.*
    - Be careful if you are using the ***If Any Object At Place*** DnD action or the **place_empty()** function in GML. This method might make it difficult to walk into a powerup!

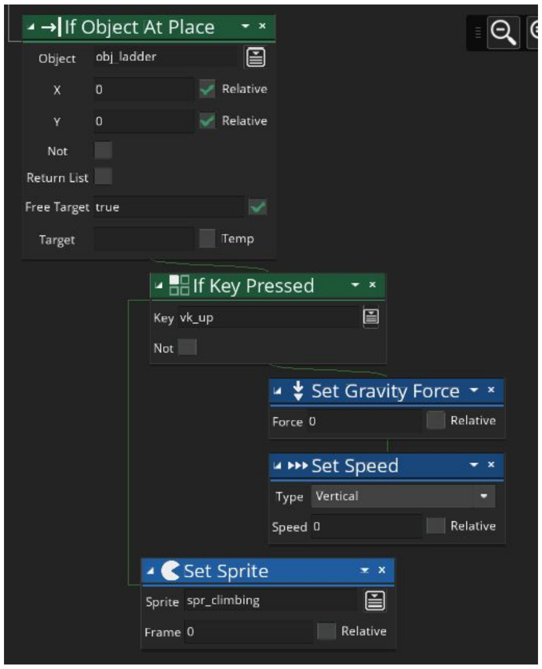
Part 5: Ladder Climbing



- Start by using the tileset to give an appropriate look to the ladder, which is the green bar on the lower-left area of the room. *This is an invisible object, so you need to be in the room editor to be able to see it.*

- Use the vine tiles to create a look like the one in the following image:





```
If Object At Place
  Object       obj_ladder
  X            0      ✓ Relative
  Y            0      ✓ Relative
  Not
  Return List
  Free Target  true   ✓
  Target              Temp
    If Key Pressed
      Key  vk_up
      Not
        Set Gravity Force
          Force 0       Relative
        Set Speed
          Type  Vertical
          Speed 0       Relative
    Set Sprite
      Sprite spr_climbing
      Frame  0          Relative
```

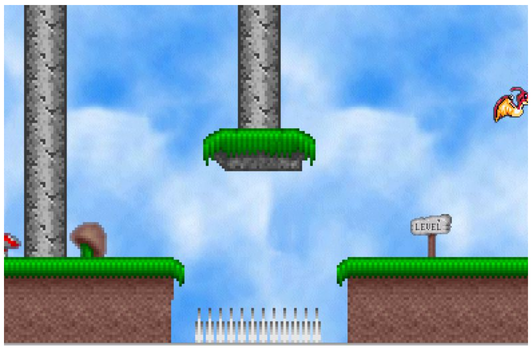
- Each step, our character should check if it is overlapping with a ladder. Use [instance_place()]() or the **If Object At Place** action to do this.
- To initiate the act of climbing, **obj_character** must both be on a ladder and pressing either the **up** or **down** keys
  - Recall that in GML, **keyboard_check(vk_up)** and **keyboard_check(vk_down)** can be used for this purpose.

- Note that the player character will behave very differently when climbing. **They won't be able to move left & right, and instead of jumping when up is pressed they climb up on the ladder.** To enforce this drastic change in behavior, we need a variable that keeps track of whether or not **obj_character** is climbing.
- For the purposes of this tutorial, we shall call the variable **`climbing`**. To initialize it, assign it the value **`false`** in the create event. This will ensure that **`climbing`** has a value before any other event attempts to check it.

- With our variable `climbing` declared, we need to make some edits:
    - In the condition that checks if we are pressing up or down on a ladder, set `climbing` to `true`.
    - Also set **vspeed** to 0.
    - On the other hand, set `climbing` to `false` if our character is not colliding with a ladder.
- In our left & right movement events, check that `climbing` is `false` before moving.
- Implement functionality in the up & down events similar to our left & right movements, but check that `climbing` is `true` instead. Alternatively, disable the effects of the **If Any Object At** action in the Left and Right Key Downevents.
- We should not be applying `gravity` if `climbing` is `true`.
- There is a sprite named ***spr_climbing***. Make sure that it is displayed when appropriate.

Part 6: Multiple Levels

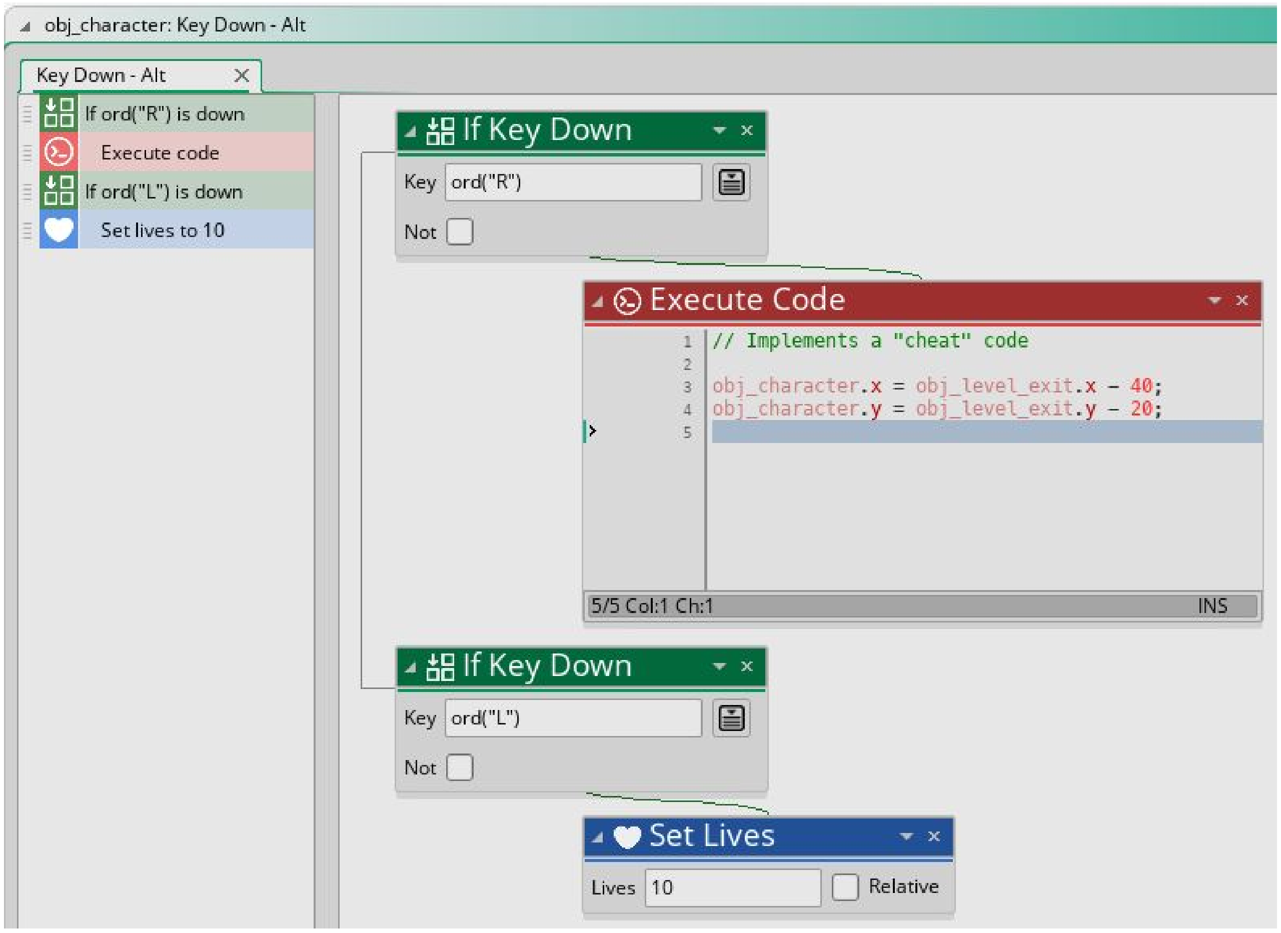
- The game resources include an object named obj_level_exit that we can use to depict the end of a level. When **obj_character** collides with this object, the current room changes to a new one.
    - There are functions that can be used to either **Go to the Next Room** or **Go to [a] Room** of Your Choosing in Drag and Drop or room_goto_next() or room_goto() in GML. Any of these can be used for these purposes, but it's recommended to check the manual to see how they work.



- In order to test this, you need to create an additional room **(You must have at least 2 levels total).** A good, easy way to achieve this for now is just duplicating the existing room and making some changes to it.
    - To duplicate a room, right-click on the room's name in the resource pane and select *Duplicate* from the context-sensitive menu.

Part 7: Cheat Codes

- **Games should have cheat codes that facilitate testing.**
- Cheat codes perform actions such as:
  - Grant unlimited lives
  - Advance to the next level
  - Restart the game
- The following image depicts a typical way of implementing such cheat codes:



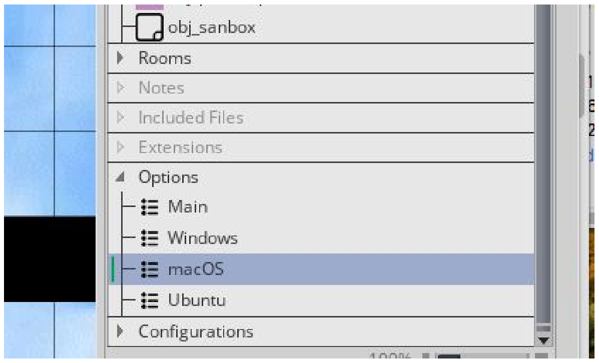
Key: ord("R")

```
// Implements a "cheat" code

obj_character.x = obj_level_exit.x - 40;
obj_character.y = obj_level_exit.y - 20;
```

Key: ord("L")

Lives: 10

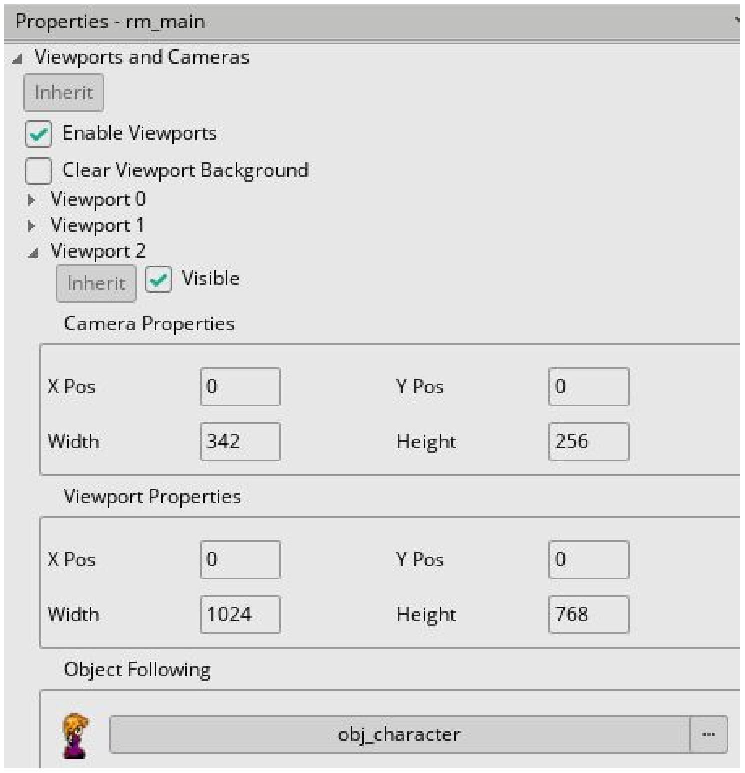
Part 8: Maximizing the Screen



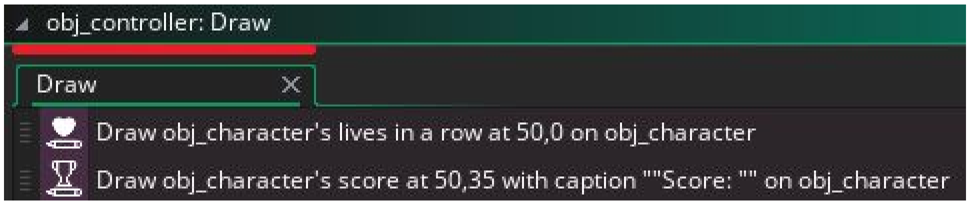
- To enable full screen switching, you need to go into the **Options** dialog for each target platform. The Instructor and TA's use both Windows and MacOS, please enable fullscreen on both of these operating systems.
- Edit the object named **obj_controller** to hold the functionality to toggle between full-screen and window mode.
- Implement two keyboard events as follows:
    - **F4 toggles fullscreen**
        - Look in the manual for a function that sets whether the game is run in fullscreen or not.
        - If using Drag & Drop, you will need to implement this in an **Execute Code** action.
    - **F10 exits the game**

Part 9: Using views

- GameMaker Studio provides built-in functionality to enable multiple different views of the same room.
- These are configured per room, using the ***Viewports and Cameras*** section of the ***Room Properties*** pane in the ***Room Editor***
- We will use views to make the game more challenging by using a viewport that restricts what the player can see.
    - Make sure that the camera area is smaller than the Viewport
    - Make sure to set the *Object Following* property to use the player's character.
- We can also create a simple mini-map, by using a camera that covers the entire screen area, but uses a small viewport.
    - *Hint: You will need to enable a different viewport*

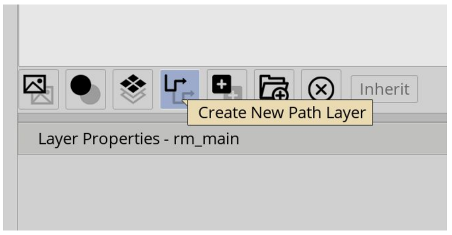

Properties - rm_main
Viewports and Cameras
Inherit
✓ Enable Viewports
☐ Clear Viewport Background
Viewport 0
Viewport 1
Viewport 2
Inherit ✓ Visible
Camera Properties
X Pos 0 Y Pos 0
Width 342 Height 256
Viewport Properties
X Pos 0 Y Pos 0
Width 1024 Height 768
Object Following
obj_character

Note: When using viewports, the x and y position will need to be fixed onto the character's x and y position.

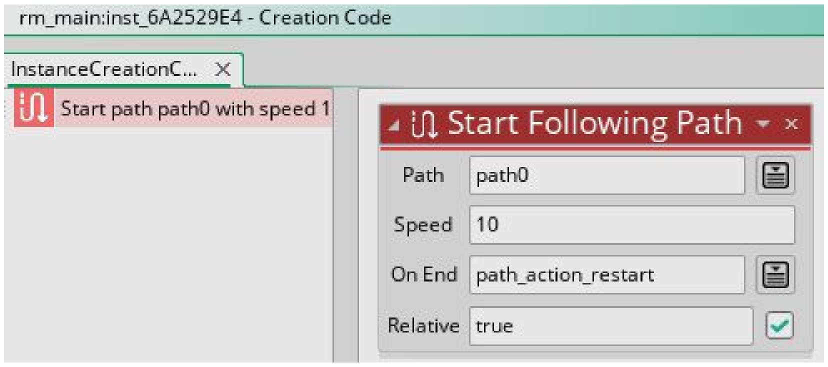

obj_controller: Draw
Draw
Draw obj_character's lives in a row at 50,0 on obj_character
Draw obj_character's score at 50,35 with caption ""Score: "" on obj_character

Part 10: Firing a bullet based on an object's sprite

- Create a new path layer and apply it to one of the flying monsters
  - Using the room toolbox, create a new **Path Layer** (see image)
  - Select *Path* and *Create New path*
  - You can now click on the room to create a path
  - Create a new path by selecting multiple waypoints.
    - Make sure that the path is navigable, i.e., objects should not run into obstacles and get stuck while following the path.



- Create a new object named *obj_flyer_with_path*
  - On **Create** event, set the path.
  - This can be done for all instances (on object) or for a single instance in room.
  - You can also have a dedicated type of object that has a path assigned.



rm_main:inst_6A2529E4 - Creation Code

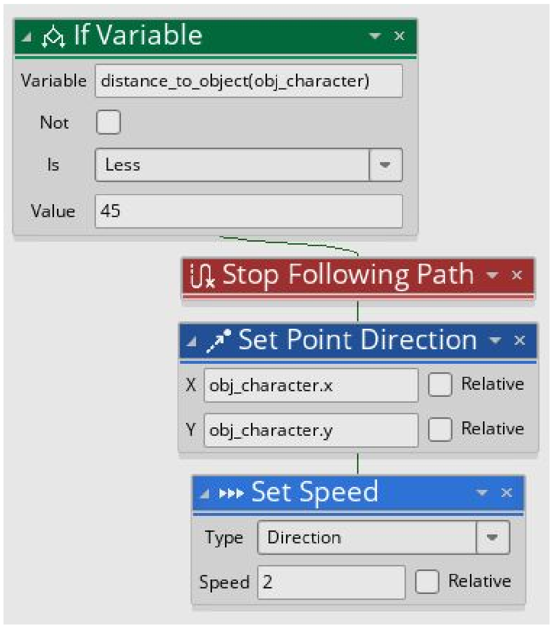
InstanceCreationC...

Start path path0 with speed 1

Start Following Path

Path: path0
Speed: 10
On End: path_action_restart
Relative: true

- Check for proximity
  - Applies to the flyer monster with path
  - **Step** event
  - Use 2 for the speed



If Variable
Variable: distance_to_object(obj_character)
Not: ☐
Is: Less
Value: 45

Stop Following Path

Set Point Direction
X: obj_character.x ☐ Relative
Y: obj_character.y ☐ Relative

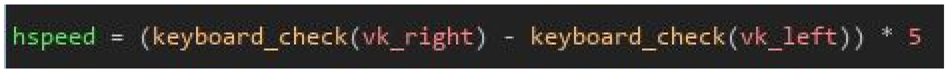
Set Speed
Type: Direction
Speed: 2 ☐ Relative

Additional Items NOT Required But Worth Considering*:*

- Middle click on a function name brings context-sensitive help
- Use of /// comments to add JSDoc Script comments
  e.g. /// @description My Function
- Custom-built text boxes
- Variables can store numbers or strings
  - Use *var* for local variables, i.e., variables used only in specific events
- When drawing text, use hashtags (#) to create new lines
- Objects will snap to the grid when placed in a room. To change this and do exact placement, hold CTRL while dragging them.
- To change the alignment of an object with respect to the grid, use the ***Flip X*** and/or ***Flip Y*** buttons.
  - In GM 2.x object properties appear after double-clicking on the object.
- Reminder: The order of layers in the room editor determines whether a sprite is obfuscated by another. For example, drag the tile layer to the top and watch what happens when the princess character climbs.
- For sprites with multiple frames. It is necessary to set the speed to zero if only one frame will be displayed.
- Make sure to always set what draw and set DnD actions are relative to.
- Path are set in the object creation code. Double click on instance.
  - Check precision.
  - Check relative = true
  - Be careful with solids!
- When a room restarts or the player changes rooms, non-persistent objects will respawn.
- DnD Draw Value cannot be left blank.
- GameMaker Studio has an exhaustive built-in manual, which includes sample code.

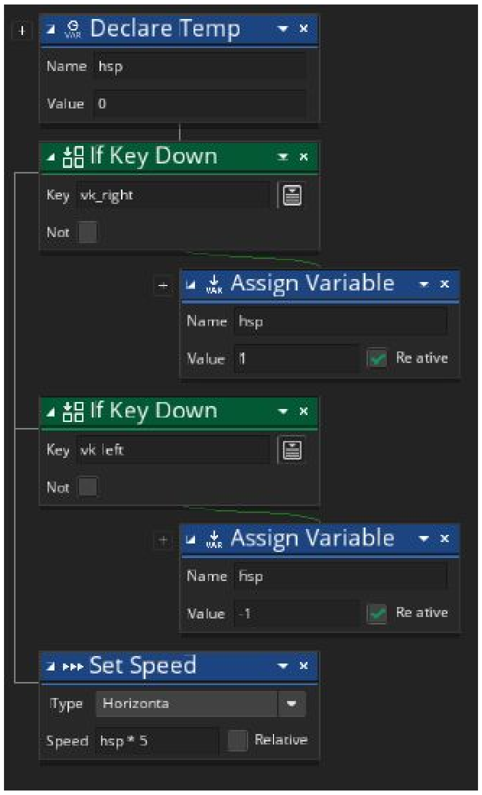
Part 11: Optional Collision System

- Before we explain the collision system, one important detail is that this system requires the player to be using *hspeed* for movement, rather than Jump to Point or x +=.
- Also, this system will not work with every type of project. This code will help your game if you are using a movement system dependent on *hspeed* and *vspeed*. However, even if you are planning to do something else, it would be useful practice to go through this. It could help to understand collisions on a more conceptual level.
- First, try changing the *Key Down - Left* and *Key Down - Right* events so they set *hspeed* rather than directly move the object, then run the game to see what happens.
- You will notice that hspeed never gets set to zero this way. The best way to do this would be to set *hspeed* in the player's step event like follows (DnD NOTE: It's hard to see, but the value of the second assign variable action is -1):



```
hspeed = (keyboard_check(vk_right) - keyboard_check(vk_left)) * 5
```

- So how does this work? The first part is getting the direction of movement.
- In GML: What is going on is that **keyboard_check()** actually returns a 0 for false and a 1 for true. If the right arrow key is being held, **keyboard_check(vk_right)** returns 1, else it returns 0. By subtracting left from right, we get the direction that we want the player to move.
- In DnD: First, we create a temporary variable and set it to 0, in this case we can call it hsp. If right is being held down, add 1 and if left is being held down, we subtract 1. This gets us the direction that we want the player to move.
- Then we can multiply the direction by the actual speed we want the player to move at.
- We can visualize this by using a table.



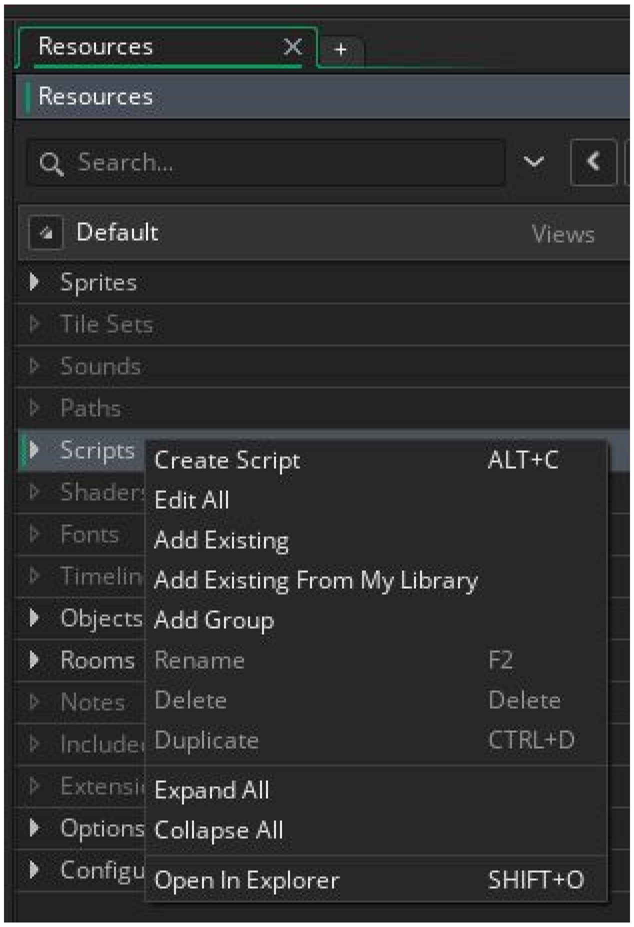
```
Declare Temp
  Name   hsp
  Value  0
If Key Down
  Key    vk_right
  Not
    Assign Variable
      Name   hsp
      Value  1        ✓ Relative
If Key Down
  Key    vk_left
  Not
    Assign Variable
      Name   hsp
      Value  -1       ✓ Relative
Set Speed
  Type   Horizonta
  Speed  hsp * 5      Relative
```

Holding Right?	Holding Left?	Right - Left	Final hspeed
0 (f)	0	0	0
0 (f)	1	-1	-5
1 (t)	0	1	5
1 (t)	1	0	0

- This fixes the prior issue, since *hspeed* will be set to zero each frame that the player is not holding left or right.

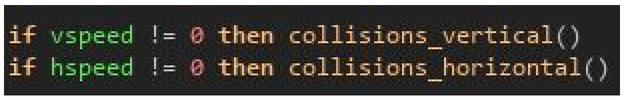
11-A: Creating the Scripts

- This system itself is split between two scripts, vertical collisions and horizontal collisions. A script is a resource which contains a series of programming instructions (code). Scripts can be run from inside instances to help avoid duplicating the same code repeatedly.
- An important function of scripts is that any code that gets used will be run from the perspective of the instance calling it. For example, suppose that there was a script that increases *hspeed* by one when it is called. If the player calls it, it will increase the player's *hspeed*. If an enemy runs the same script, it will increase that enemy's hspeed.
- To create a script, right click the "Scripts" folder on the resource pane. Then select **Create Script.**
- You will need to create two scripts, go ahead and name them "collisions_vertical" and "collisions_horizontal"

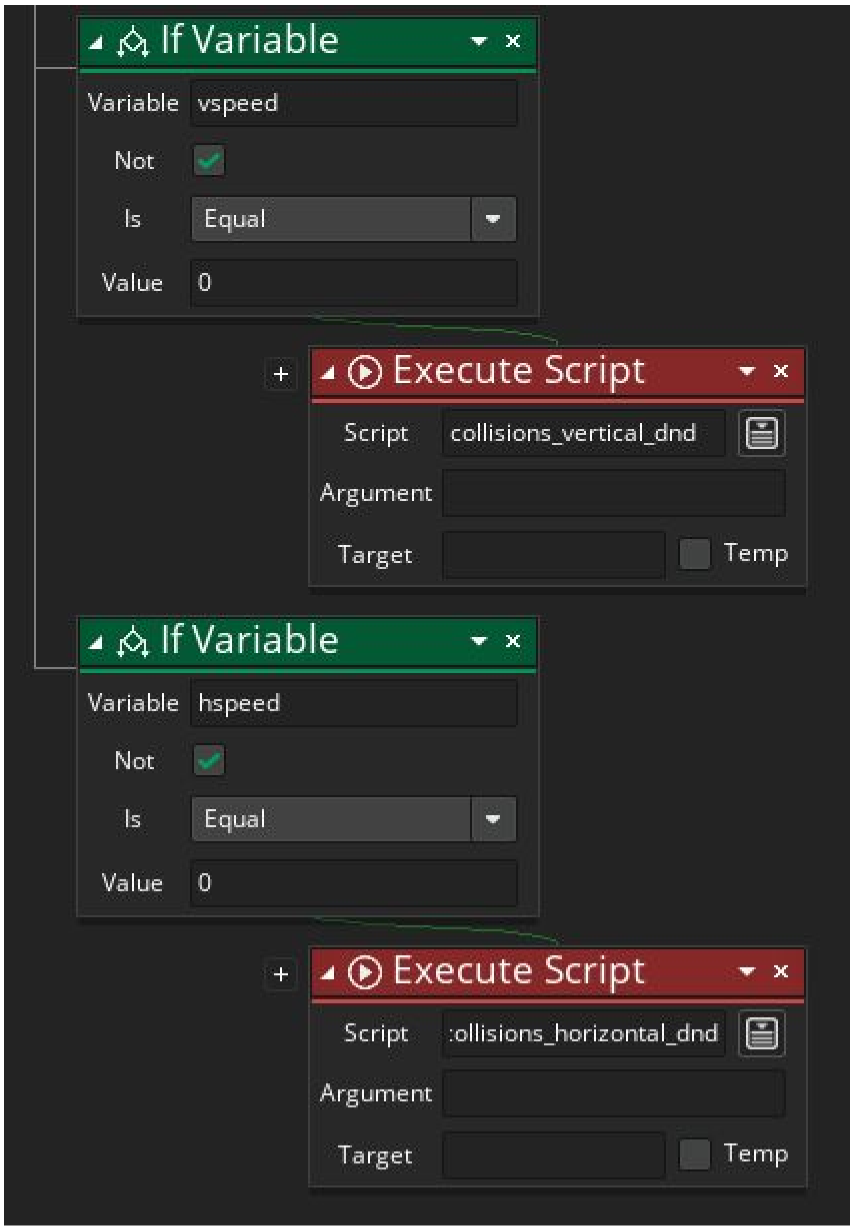


11-B: Calling the Scripts

- We only want to check the collisions if the player character is moving. Implement a check for each script that checks the player's speed for the specified direction.
- To call a script, all you need to do is use the Execute Script action in Drag and Drop or simply call it as if you were calling a regular method in GML. Put these at the very end of your step event so that they activate after gravity and vspeed are set.



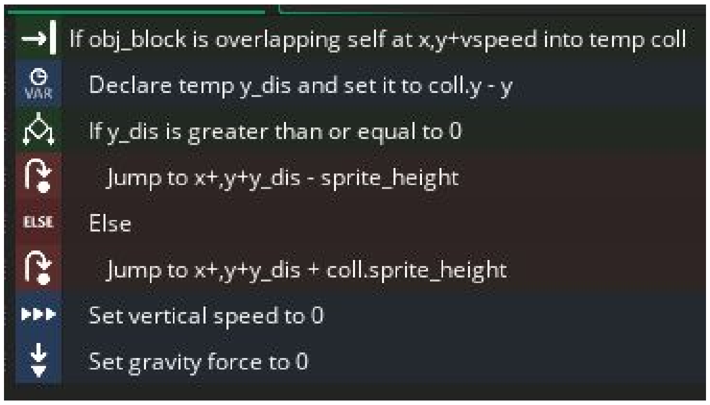
```
if vspeed != 0 then collisions_vertical()
if hspeed != 0 then collisions_horizontal()
```



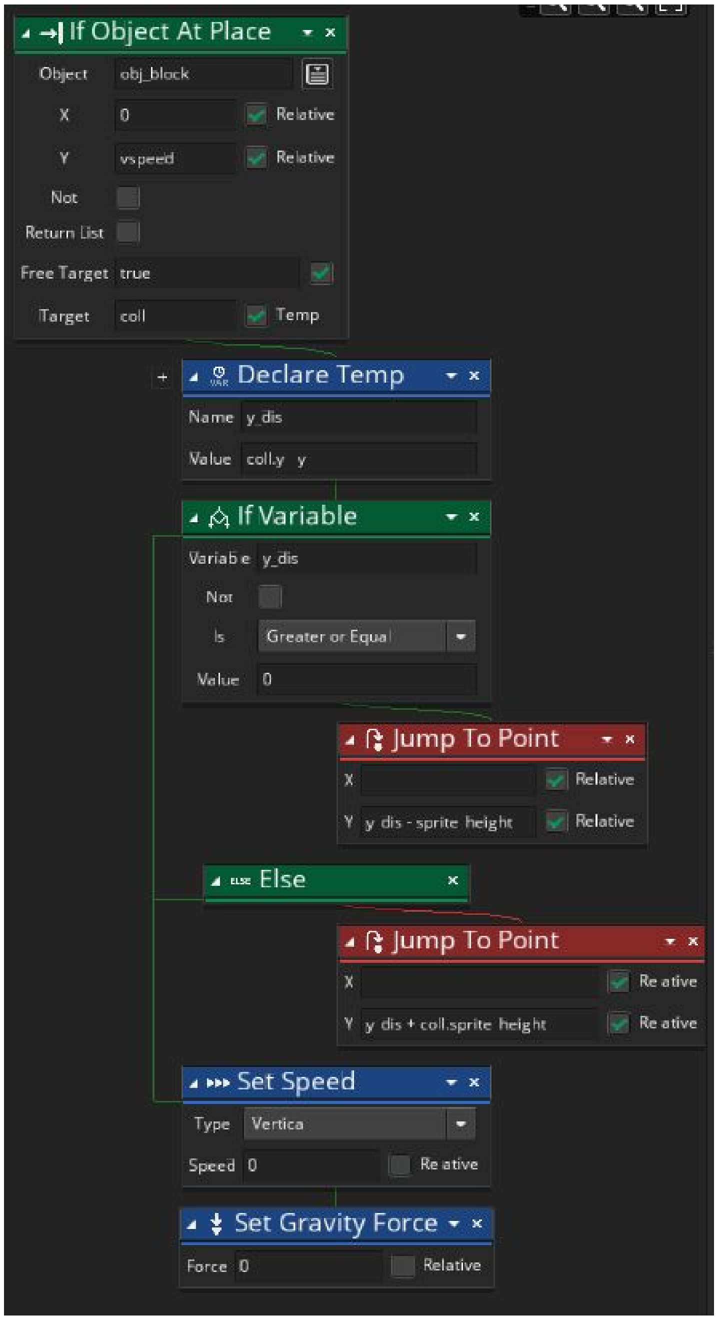
```
If Variable
Variable  vspeed
Not       ✓
Is        Equal
Value     0

    Execute Script
    Script    collisions_vertical_dnd
    Argument
    Target              Temp

If Variable
Variable  hspeed
Not       ✓
Is        Equal
Value     0

    Execute Script
    Script    :ollisions_horizontal_dnd
    Argument
    Target              Temp
```

- Now, onto the script code. This tutorial will give you the script for the vertical collisions, break down in detail how it works, then challenge you to complete the code for horizontal collisions. To prevent there being any confusing overlap, the rest of this tutorial will be split into two sections. The first will cover the Drag and Drop implementation of the system and the second will cover the GML implementation.

11-C: Drag and Drop Implementation



```
If obj_block is overlapping self at x,y+vspeed into temp coll
    Declare temp y_dis and set it to coll.y - y
    If y_dis is greater than or equal to 0
        Jump to x+,y+y_dis - sprite_height
    Else
        Jump to x+,y+y_dis + coll.sprite_height
    Set vertical speed to 0
    Set gravity force to 0
```



If Object At Place
Object: obj_block
X: 0 ✓ Relative
Y: vspeed ✓ Relative
Not: ☐
Return List: ☐
Free Target: true ✓
Target: coll ✓ Temp

Declare Temp
Name: y_dis
Value: coll.y - y

If Variable
Variable: y_dis
Not: ☐
Is: Greater or Equal
Value: 0

Jump To Point
X: ✓ Relative
Y: y_dis - sprite_height ✓ Relative

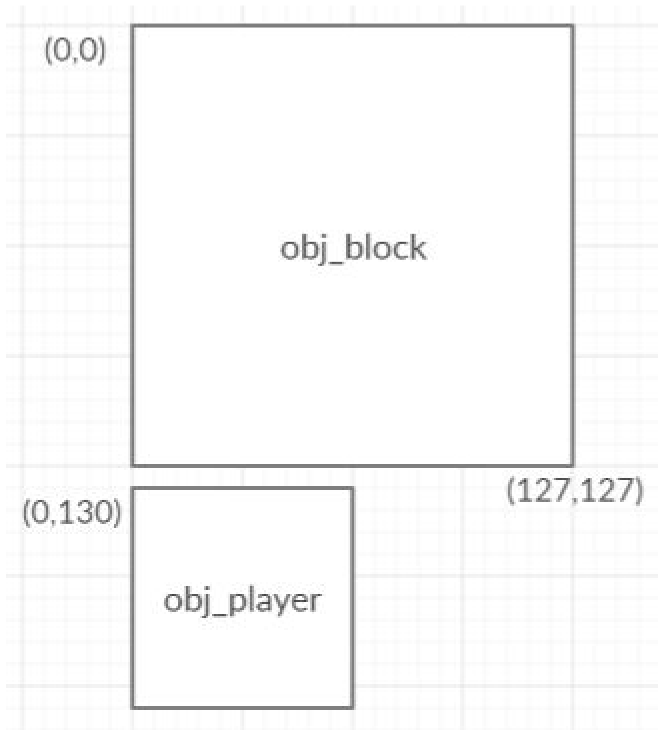
Else

Jump To Point
X: ✓ Relative
Y: y_dis + coll.sprite_height ✓ Relative

Set Speed
Type: Vertical
Speed: 0 ☐ Relative

Set Gravity Force
Force: 0 ☐ Relative

- Okay. Now, let's break down each of the individual actions.
- We will walk through this with a player that has a 64x64 sprite and is at position (0, 130).
    - It is travelling toward the top of the screen with a vspeed of -5.
    - The wall has a 128x128 sprite at position (0, 0).
        - This will fill up all the space from its sprite origin through position (127, 127).

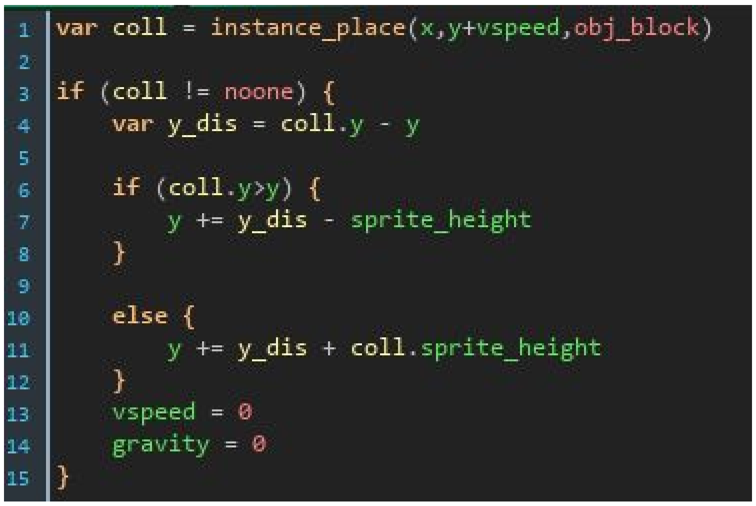


- **If Object At Place**: This checks the position that the player is going to move into. If that position is taken by a block, store a reference to that block into a temporary variable "coll." If that position is not taken, that means that the player will be free to move there, and the script will end.
    - In our example, GameMaker checks the spot (0, 125). The block is filling up that coordinate, so it continues with the code. It also stores a temporary reference to that block as coll.
- **Declare Temp**: This variable is called "y_dis" because it's holding the distance between the block's y value and the player's y value.
    - `coll.y - y` in this case would be `0 - 130 = -130`. Note that y is *not* 125, since the player object has not actually moved yet.
- **If Variable**: Checks if the distance from the coll's y is greater than the the player's y. If coll's y value were greater than the player's y value, that'd mean coll would be lower on the screen than the player.
    - `y_dis ≥ 0` = `-130 ≥ 0` = `false`. For our instance, we'd skip this and move on.
    - If you want to work out what would happen, start with a different obj_player at (0,0), the obj_block at (0, 66) and vspeed was +5.
- **Jump To Point 1**: This will move the sprite by the remainder of the player sprite's height subtracted from the y distance.
    - We know that y_dis must be greater than the player's height since if it were closer, a collision would be caused. Here, we need to compare the bottom of the player sprite with the top of the block. To get the bottom of the player sprite, we would need to add the player's y value and the player's sprite height.
        - In other words, `coll.y - (y + sprite_height)`

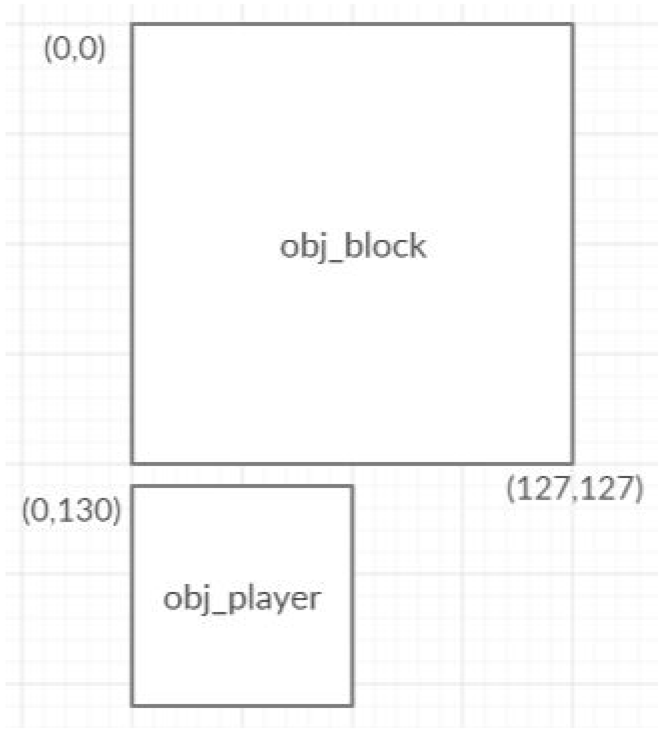
    - If the *If Variable 1* scenario was worked all the way through, the player will be at (0,2).
- **Else**: At this point we know that there must be a collision. If we are not colliding with the top of the block, we must be colliding with the bottom.
- **Jump To Point 2**: This will move the sprite the result of the distance added to the coll's sprite's height.
    - We know that the player must be lower than the block. Unlike *Jump to Point 1,* we need to compare y_dis to the bottom of the block. To compare this, we would add the block sprite's height to the block's y value, then subtract the player's y.
    - In other words, `coll.y + coll.sprite_height - y`. We can rearrange the equation to be `(coll.y - y) + coll.sprite_height`.
    - `(coll.y - y)` is y_dis, so we just need to do `y_dis - coll.sprite_height`, or `-130 - 128 =` **`-2.`**
    - All we have left for our player is to move it by -2 on the y axis, bringing it from (0,130) to (0,128). Notice that this is exactly one pixel away from the bottom of our obj_block, lining up perfectly but not colliding!
- **Set Speed / Set Gravity Force**: What we want to do here is halt momentum.
    - In our scenario, the player was moving at a *vspeed* of -5. If we do not set vspeed = 0, GameMaker will still attempt to move the player -5 pixels on the y axis. This will cause them to collide with the block anyway, even after all this work.
    - We also have to set *gravity* = 0 because if the player was moving downwards, gravity will add to the vspeed and cause the same issue.

If you have gotten this far, then go ahead and see if you can implement this with horizontal checking as well!

11-D: GML Implementation



```
var coll = instance_place(x,y+vspeed,obj_block)

if (coll != noone) {
    var y_dis = coll.y - y

    if (coll.y>y) {
        y += y_dis - sprite_height
    }

    else {
        y += y_dis + coll.sprite_height
    }
    vspeed = 0
    gravity = 0
}
```



- Now, let's break down each of the individual lines of code.
- We will walk through this with a player that has a 64x64 sprite and is at position (0, 130).
    - It is travelling toward the top of the screen with a vspeed of -5.
    - The wall has a 128x128 sprite at position (0, 0).
        - This will fill up all the space from its sprite origin through position (127, 127).

- **var coll = instance_place(x,y+vspeed,obj_block)**
    - instance_place() returns a reference to the specified object at specified (x,y) coordinates. Since it returns a reference to that object, we need to create a temporary variable to hold on to it.
        - In our example, GameMaker checks the spot (0, 125). The block is filling up that coordinate, so it continues with the code. It also stores a temporary reference to that block as coll.
- **if (coll != noone) {**
    - noone is a GameMaker keyword that represents "no instance at place." If there is nothing in the way of obj_player's movement, then we can skip the rest of this code.
        - In our case, coll is holding a reference to an obj_block. Thus, coll != noone and we can continue

- **var y_dis = coll.y - y**
    - This variable is called "y_dis" because it's holding the distance between the block's y value and the player's y value.
        - `coll.y - y` in this case would be `0 - 130 = -130`. Note that y is *not* 125, since the player object has not actually moved yet.
- **if (coll.y>y) {**
    - Checks if the distance from the coll's y is greater than the the player's y. If coll's y value were greater than the player's y value, that'd mean coll would be lower on the screen than the player.
        - `y_dis ≥ 0` = `-130 ≥ 0` = `false`. For our instance, we'd skip this and move on.
        - If you want to work out what would happen, start with a different obj_player at (0,0), the obj_block at (0, 66) and vspeed was +5.
- **y += y_dis - sprite_height**
    - This will move the sprite by the remainder of the player sprite's height subtracted from the y distance.
        - We know that y_dis must be greater than player's height since if it were closer, a collision would be caused. Here, we need to compare the bottom of the player sprite with the top of the block. To get the bottom of the player sprite, we would need to add the player's y value and the player's sprite height.
            - In other words, `coll.y - (y + sprite_height)`
        - If the *If Variable 1* scenario was worked all the way through, the player will be at (0,2).
- **else {**
    - At this point we know that there must be a collision. If we are not colliding with the top of the block, we must be colliding with the bottom.
- **y += y_dis + coll.sprite_height**
    - This will move the sprite the result of the distance added to the coll's sprite's height.
        - We know that the player must be lower than the block. Unlike *Jump to Point 1,* we need to compare y_dis to the bottom of the block. To compare this, we would add the block sprite's height to the block's y value, then subtract the player's y.
        - In other words, `coll.y + coll.sprite_height - y`. We can rearrange the equation to be `(coll.y - y) + coll.sprite_height`.
        - `(coll.y - y)` is y_dis, so we just need to do `y_dis - coll.sprite_height`, or `-130 - 128 =` **`-2.`**
        - All we have left for our player is to move it by -2 on the y axis, bringing it from (0,130) to (0,128). Notice that this is exactly one pixel away from the bottom of our obj_block, lining up perfectly but not colliding!

- **vspeed = 0 / gravity = 0**
    - What we want to do here is halt momentum.
        - In our scenario, the player was moving at a vspeed of -5. If we do not set vspeed = 0, GameMaker will still attempt to move the player -5 pixels on the y axis. This will cause them to collide with the block anyway, even after all this work.
        - We also have to set gravity = 0 because if the player was moving downwards, gravity will add to the vspeed and cause the same issue.

If you have gotten this far, then go ahead and see if you can implement this with horizontal checking as well!