Introduction to GameMaker: Workshop 2
ITCS 4230/5230

Learning Outcomes

By the end of the workshop students will be able to use *GameMaker* to do the following:

1. Implement functionality to move a character in a 2D platformer, including:
    a. Using **collisions** to pick up items and fight enemies.
    b. **Dynamically modify an object's appearance** to reflect what it is doing.
    c. Use **gravity**.
    d. Implement code that uses the different key **events**.
    e. Implement a **melee attack**.
2. Implement a simple **finite state machine** for the player's character.
3. Use **tiles** to create background effects.
4. Apply **basic AI** concepts such as:
    a. Enemies that have **patrol paths**.
    b. Enemies that **follow** the player.
5. Use **viewports** for game user interface functionality.
6. Implement the game objects and mechanics necessary to **change rooms**.
7. Write **cheat codes**.

Setup

1. Download the activity file named ***`Simple_Platformer_GML.zip`*** from the assignment page. This file contains the skeleton code (starting point) for the simple platformer that you will submit.
2. Unzip the file to an appropriate location on your computer.
3. Start GameMaker and open the project (***`Simple_Platformer_GML.yyp`***).
4. Make sure the game builds and runs.

Copyright © 2019 - 2023, Julio C. Bahamón, UNC Charlotte
All Rights Reserved.

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.

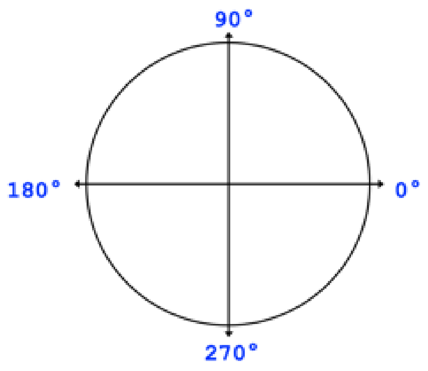
As you look at the project code, keep in mind that GameMaker Studio uses the Cartesian plane for coordinates, as well as degrees (0 – 360) for the value of ***`direction`***. A quick reference for these is shown below.

Direction:

- `0` to the right
- `90` straight and up
- `180` to the left
- `270` straight and 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.*

Important:

- Whenever possible, **hyperlinks to the documentation are included in the instructions**. You should click on the link for more details.

- We highly recommend using the GameMaker manual (https://manual.yoyogames.com/#t=Content.htm) as a reference as you work on this assignment and other game projects.

Copyright © 2019 - 2023, Julio C. Bahamón, UNC Charlotte
All Rights Reserved.

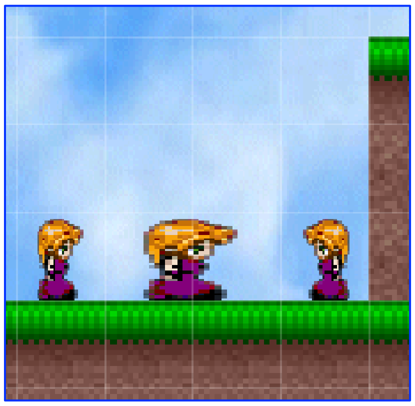
Part 1: Player Movement in a Platform Game

1-A: Walking left & right

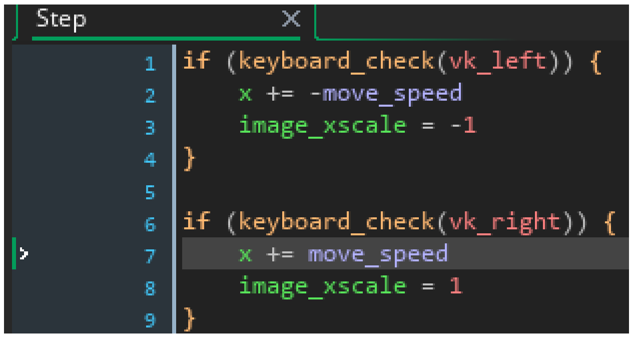
- All the code will be written in `obj_player`.
- All of the sprites needed have been provided.
- Horizontal movement has already been implemented. The default code uses a variable called `move_speed`. This is set as a variable definition.
- Note that he character's sprite always faces right, regardless 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 their way

image_xscale

- To change the sprite's direction, you will be using a built in variable called `image_xscale`.
- *`image_xscale`* will resize an object's sprite on the `x` axis. Changing it to 2 will double the width of the sprite, and setting it to be -1 will invert the sprite on it's x axis. See the image for an example.
- In the **Step event**, add a line of code reading `image_xscale = -1` within the left keyboard check. Do the same in the right check, except set it equal to `1`.
- Run the game and see what happens!



Left: `image_xscale = -1`
Middle: `image_xscale = 2`
Right: `image_xscale = 1`

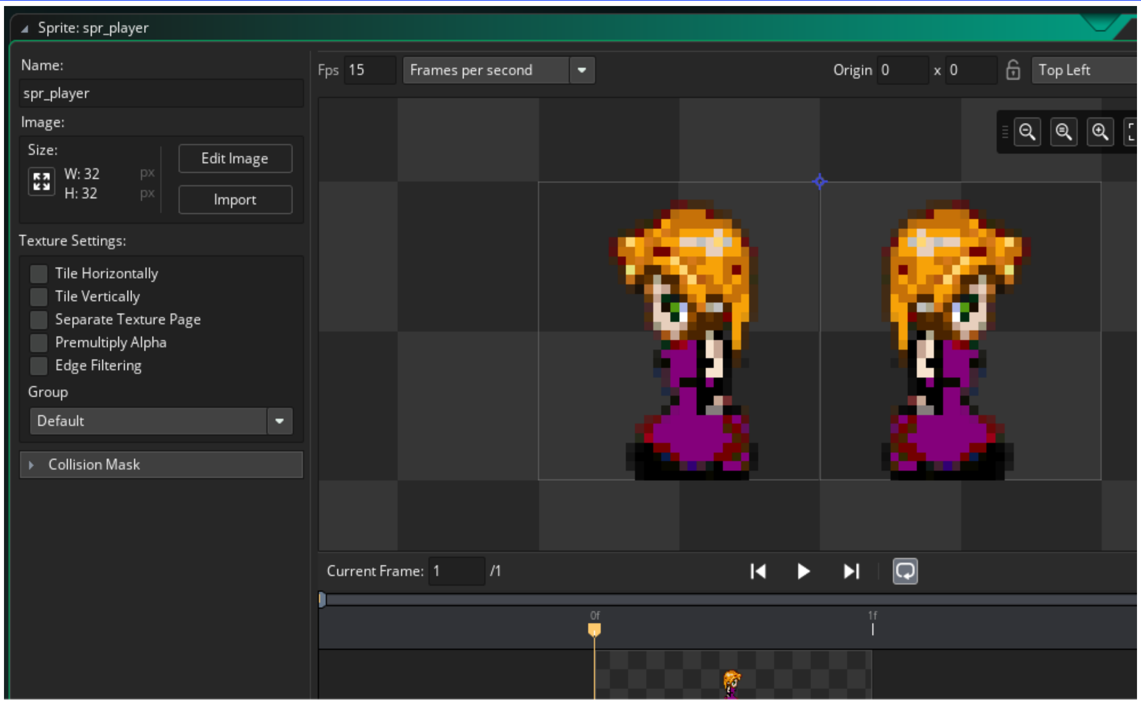


```
if (keyboard_check(vk_left)) {
    x += -move_speed
    image_xscale = -1
}

if (keyboard_check(vk_right)) {
    x += move_speed
    image_xscale = 1
}
```

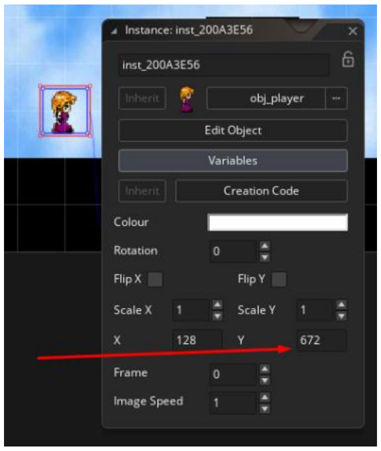
- Notice that the sprite blinks forwards and backwards when you move left and right. This is due to the sprite's origin, which needs to be corrected. See the image below.

Copyright © 2019 - 2023, Julio C. Bahamón, UNC Charlotte
All Rights Reserved.



The image on the left of the crosshairs is an edited representation of what the sprite *would* look like if `image_xscale` were set to `-1`. The image on the right is what the sprite looks like normally, with an `image_xscale` of `1`.

- Each GameMaker sprite is assigned an origin. In the last workshop, we used `sprite_xoffset` to calculate the distance from a sprite's origin to the edge of the sprite itself. `image_xscale` works using a similar principle. When it is set to `-1`, it flips the image along the origin's `x` coordinate.
- To correct the problem, **change the origin to *Middle Centre***.
- Nex, open **rm_main**, and notice that your character is now hovering above the ground. This will be inconvenient later, so the character needs to be moved down a bit. Double click on the player sprite and add 16 pixels to the y value (it should be `688`).
- Run the game, and see your character turn left and right properly!
- Next, we need to check for collisions. This needs to be done before the player moves, so you will not be able to use the



Copyright © 2019 - 2023, Julio C. Bahamón, UNC Charlotte
All Rights Reserved.

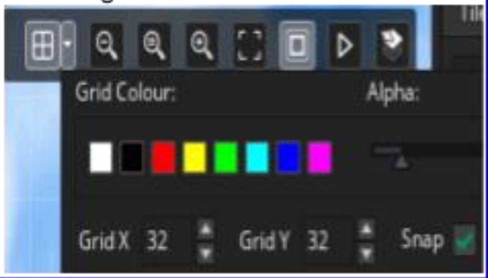
collision function. For this, you will be using GameMaker's `instance_place()` function.

- The `instance_place()` function takes in 3 variables: `x, y` and the object type it is going to check for. It checks if moving to those `x` and `y` values would cause a collision with the specified object. If there is a collision, `instance_place()` will return the **instance** it is colliding with. If there is no collision, `instance_place()` returns `noone`. See [Instance Keywords]() in the GameMaker manual.
    - You will be using this to check if the next movement is a valid move. Add a `and` clause to the `if` statement for *left* motion in the **Step** event, then write `!instance_place(x-move_speed, y, obj_block)`.
    - Do the same for the *right* motion if statement, except use `x+move_speed`.
    - **Make sure that there is an `!` before** `instance_place()`, to check if the position is empty before moving to it. In GML `!` is the operator for logical NOT.
    - See the image below for the complete code.
- Run and test the game. The character should be able to move left and right, but only if there is nothing blocking their path.

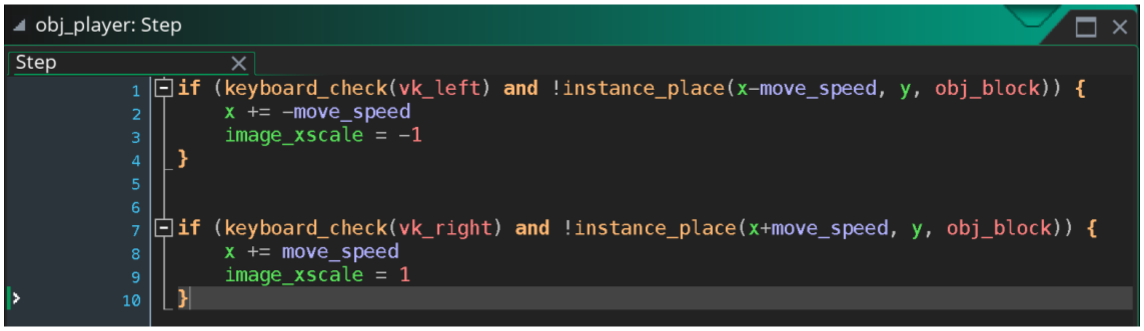
A Quick Note: The Grid

By default, each new room will cause all objects to snap to the grid. This only applies when you are placing objects in the room and obviously does not apply while the game is running. As you just saw, this could be inconvenient when using sprites that do not perfectly align to the grid. This can be remedied by editing the *y* value of each sprite, but a faster way could be to just change the grid to 16x16.

Snapping to grid is generally a powerful tool, especially for a game like a platformer. It allows a developer to easily align objects that would require pixel precision. For example, click around on some of the blocks in the room. If those blocks were off by even a single pixel, it could potentially cause issues when handling collisions.



Grid X 32
Grid Y 32



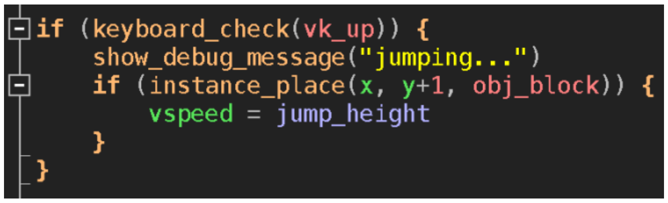
```
if (keyboard_check(vk_left) and !instance_place(x-move_speed, y, obj_block)) {
    x += -move_speed
    image_xscale = -1
}


if (keyboard_check(vk_right) and !instance_place(x+move_speed, y, obj_block)) {
    x += move_speed
    image_xscale = 1
}
```

Copyright © 2019 - 2023, Julio C. Bahamón, UNC Charlotte
All Rights Reserved.

1-B: Jumping

- Jumping applies vertical momentum. We need to set `vspeed` instead of setting the character's `y` position.
- First, create a new variable definition called `jump_height` and set it to a negative number (e.g., `-8`).
- In the **Step** event, add a new `if` statement, this time vhecking `if(keyboard_check(vk_up))`. Within this `if` statement, set `vspeed` equal to `jump_height`.
- 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, y+1)`. Use a **nested `if` statement with `instance_place()` inside** of the keyboard check to do this.



```
if (keyboard_check(vk_up)) {
    show_debug_message("jumping...")
    if (instance_place(x, y+1, obj_block)) {
        vspeed = jump_height
    }
}
```

- Run the game. You should notice that once `obj_player` jumps it never comes back down…

A Quick Note: Variable Definitions

You saw in the last workshop that variable definitions are powerful tools for inheritance. So why are we using them in the player? Simply put, it provides an easy way to change multiple values at once. Say you wanted to change the movement speed to 6. If you had just hard coded 4 each time movement was referenced, you would have to search through the code for any instance of 4, making sure you did not miss any. Here, all you have to do is change the value of

1-C: Gravity

- `gravity` is a built-in variable present in all *GameMaker* objects. It has an accompanying variable, `gravity_direction`
  - Every game update, `gravity` adds speed in a given 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_player` downward, set `gravity_direction = 270`.**
  - `gravity_direction` only needs to be set once, usually in the **Create** event.
- Gravity is usually handled in the **Step** event, which runs every frame.
- Similar to jumping, you need to check whether there is solid ground below `obj_player`. If there is, `gravity` should be 0, otherwise it should be a positive number (0.25 should work, but we recommend testing).
- After implementing gravity, run the game. You should see that after jumping, `obj_player` falls back down *and then proceeds to fall through the floor…*

Copyright © 2019 - 2023, Julio C. Bahamón, UNC Charlotte
All Rights Reserved.

1-D: Colliding with walls

- While walking left/right, you avoid colliding with walls entirely using the conditional check. You are working with `vspeed` to facilitate jumping, which makes *avoiding* collisions much harder. Instead, you can use a **Collision** event with `obj_block` to react when a collision occurs.
- What you need to do here is set `vspeed` to `0` after colliding with a block, and `obj_player` will stop falling once they hit the ground.
- Once you have a jump working, look to the raised block directly next to the player. You might not be able to jump over it at this time. Before continuing, play with the values for ***jump_height*** and ***gravity***. Make sure that you can jump high enough to land on that platform.

A Quick Note: Solid Objects

If you check `obj_block`, you will see that it is flagged as solid. 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). 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 occurred. Suppose 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.

1-E: Limiting Vertical Speed

- If `obj_player` 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 movement 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`.
    - You may use the min() math function to accomplish this.

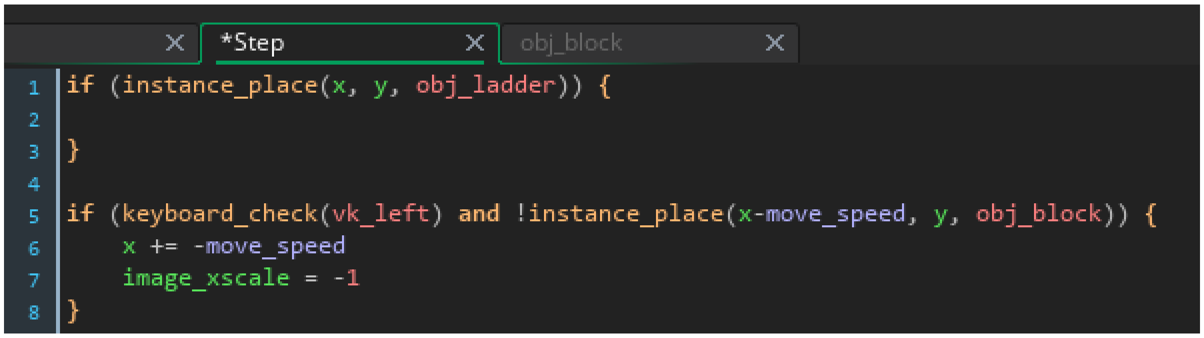
Copyright © 2019 - 2023, Julio C. Bahamón, UNC Charlotte
All Rights Reserved.

Part 2: Ladder Climbing

You may have noticed the green lines that are in the room by default. These are going to be ladders for the player to climb. You will implement a **finite state machine** to control movement while climbing. You will also use **tilesets** and the **visible checkmark** to create the ladder look.

2-A: Climbing

- In **`obj_player`**, create a new variable definition called **`climbing`**. Set its type to **`Boolean`** and its value to **`false`** (unchecked) by default.
- Create a new variable to control climb speed called **`climb_speed`**.
- Your character will have two states: climbing and not climbing. When the character is climbing, you want it to perform a certain set of actions (i.e., climb up or down). When it is not climbing, you want it to perform its other actions, like running and jumping.
- Each step, your character should check if it is overlapping with a ladder. Go to the **Step** event of **`obj_player`** and enter a few empty lines at the top of your code. On line 1, use the **`instance_place()`** function to check if the player is colliding with the ladder.



```
if (instance_place(x, y, obj_ladder)) {

}

if (keyboard_check(vk_left) and !instance_place(x-move_speed, y, obj_block)) {
    x += -move_speed
    image_xscale = -1
}
```

- Inside this **`if`** block, write another **`if`** statement that checks if the player is holding up or down. If either is true, set **`climbing`** to **`true`**, **`vspeed`** to **`0`** and **`gravity`** to **`0`**.
- You will also need to set the player's sprite to **`spr_climbing`**.
  - Use the built in variable **`sprite_index`** for this.
- Next, you need the player to detach when they are not colliding with the ladder. After the **`if(`*`instance_place`*`())`** block, write an **`else`** statement. Inside that **`else`** statement, set **`climbing`** to **`false`**. You also need the player's sprite to go back to normal.
- Next, right after the **`else`** statement, write another **`if`** statement reading **`if (climbing)`**. This will control the player's behavior when they are climbing. You will need the player to
  - Move up while pressing up. Use the **`climb_speed`** variable.
  - Move down while pressing down. Use the **`climb_speed`** variable.
  - Detach from the ladder while touching the ground. Note that using **`y+1`** may not always work here.
    - Recall the earlier discussion about collisions with solid objects work. What variable can be added to **`y`** that would work here?
- Finally, you need to make sure that the player character is not able to move left or right, or jump while on the ladder. You also need to disable gravity.

Copyright © 2019 - 2023, Julio C. Bahamón, UNC Charlotte
All Rights Reserved.

- Highlight all the code below the `if (climbing)` block, then hit tab.
- Wrap the newly indented code in an `else` statement.
- Run the project and try climbing the ladder.

A Quick Note: Finite State Machines

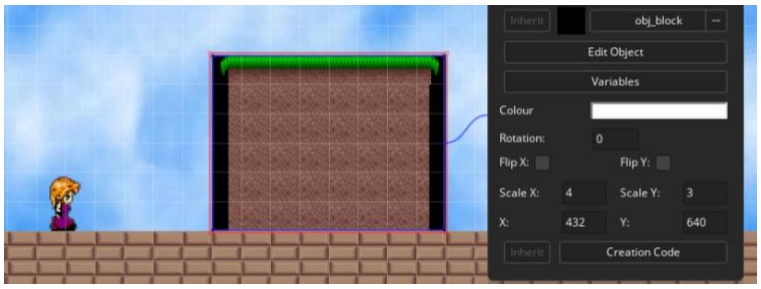
Finite State Machines (FSM) will be discussed in more detail in Module 8: Rules on Three Levels. As you will see, they are a very important element of game design that helps organize aspects of the game into individual ***states***. The playable character implements a very simple state machine, which can be broken down as follows:

A set of states: The character uses two states, 'climbing' and 'not climbing'.
A set of inputs: So far, we have reactions for each of the arrow keys. These inputs control behavior differently, depending on which state the player is in. For example, pressing up while 'not climbing' will attempt a jump, where pressing up while 'climbing' will climb upwards.
A set of transition arcs: A transition arc is a specific input that transitions from one state to another. In this instance, an example would be pressing up while colliding with a ladder. This changes the player's state from 'not climbing' to 'climbing.'

We can extend this concept, for example you could make jumping its own state. If in the air, control differently than if on the ground. Later in the workshop, you will implement a melee attack. You could have the player act differently while attacking, maybe by forcing them to be stationary. This could also be its own state. However, for the purposes of this workshop, the functionality will not be that detailed. For now, you will only use an FSM for climbing.

2-B: Tilesets

- 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 a level (room) using just one wall object.
  - Scaling an instance in the room editor will alter its built-in variables `image_xscale` and `image_yscale`. Since collisions take these variables into account, colliding with a scaled object functions as usual.

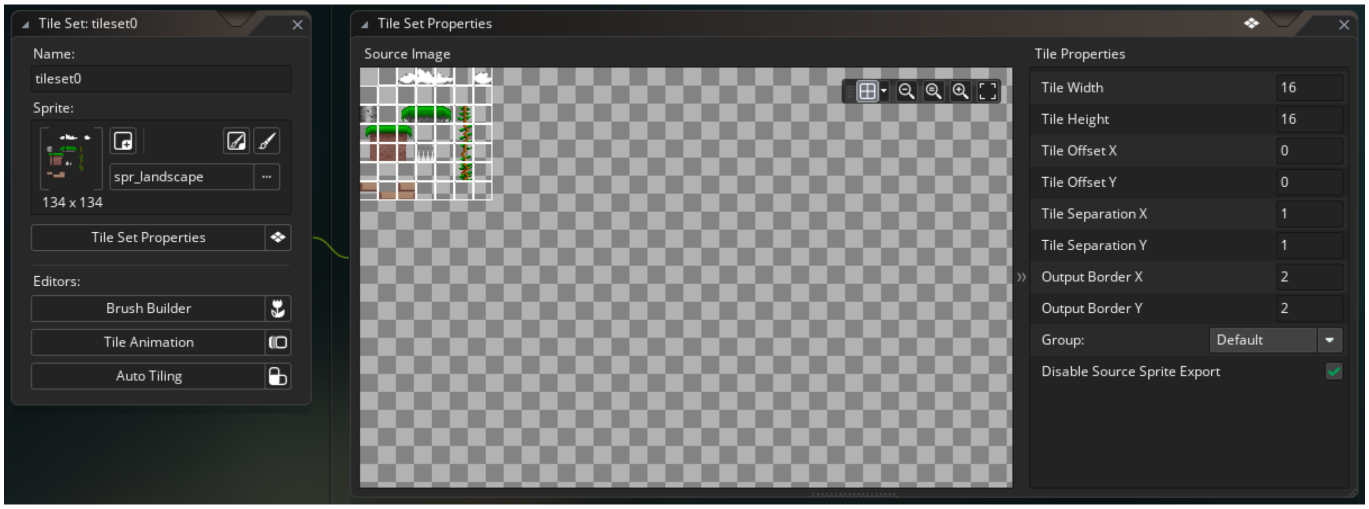


obj_block
Colour
Rotation: 0
Flip X: Flip Y:
Scale X: 4 Scale Y: 3
X: 432 Y: 640

  - You may have noticed that while objects of the type `obj_block` have been used as colliders for our level, they are not visible when the game is running. Their `Visible` property is set to `false`, because there is a *tileset layer* providing the visual aspect of the 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.

Copyright © 2019 - 2023, Julio C. Bahamón, UNC Charlotte
All Rights Reserved.

- Open the resource **`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** in the GameMaker manual.*



Tile Set: tileset0

Name: tileset0

Sprite: spr_landscape

134 x 134

Tile Set Properties

Editors: Brush Builder, Tile Animation, Auto Tiling

Tile Set Properties

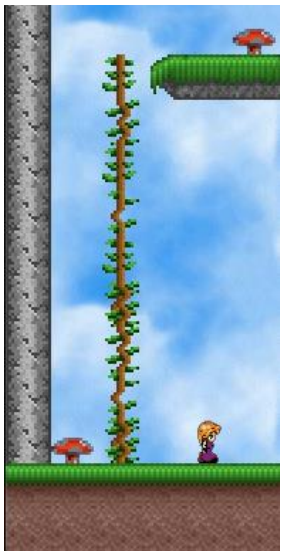
Source Image

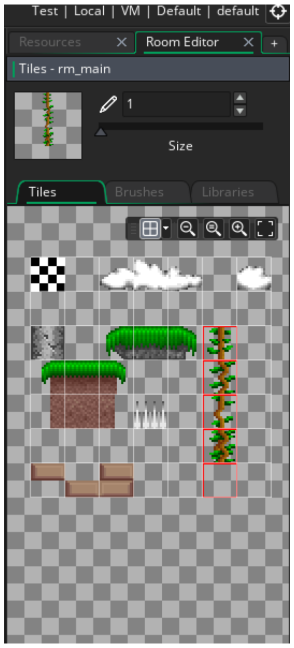
Tile Properties

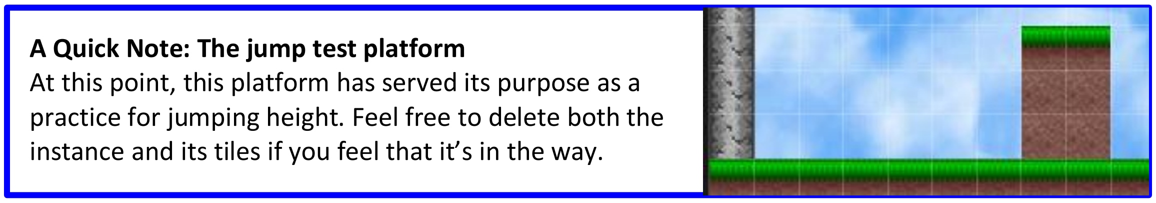
Tile Width	16
Tile Height	16
Tile Offset X	0
Tile Offset Y	0
Tile Separation X	1
Tile Separation Y	1
Output Border X	2
Output Border Y	2
Group:	Default
Disable Source Sprite Export	✓

- **The top-left tile in a tileset will always be the "empty" tile. If you make a sprite for the purposes of a tileset, remember that any image data in the top-left tile space will not be used.**

- Start by going to **`rm_main`**. Make sure you are working on the tiles layer by opening Room Editor on the left side of the screen and clicking on "Tiles_1".
- Next, click on the Room Editor on the right side of the screen. Click and drag so that all the vine tiles are selected.
- Use the vine tiles to create a look like the one in the image:
- Once you have done that, open **`obj_ladder`** and uncheck the **`Visible`** checkbox.
- Run the game. You should see the ladder without seeing the green line.







A Quick Note: The jump test platform
At this point, this platform has served its purpose as a practice for jumping height. Feel free to delete both the instance and its tiles if you feel that it's in the way.

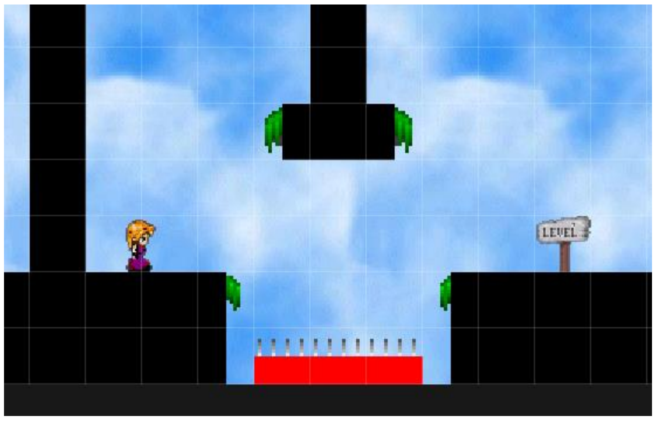
Copyright © 2019 - 2023, Julio C. Bahamón, UNC Charlotte
All Rights Reserved.

Part 3: Danger

It's time to introduce a **challenge** to the player. This section will introduce two primary dangers to the player: spikes and enemies. You will need to use the **Destroy** event and also be able to interact with the enemies.

Part 3-A: Spikes

- In the resources list, you will see an object called `obj_death`. Similarly to `obj_block` and `obj_ladder`, this object has `Visible` set to false and will be used to define an area on screen that interacts with the player.
- Place `obj_death` inside the trench on the right side of the room, then tile it over using the spikes in your tileset. Note that it might be useful to move the player character over so that it will be easier to test. (Just make sure to move them back when you are done.)
- Next, go back to `obj_player` and add a **Collision** event with `obj_death`. Call the `instance_destroy()` function to destroy the player's instance.



- Add a **Destroy** event in `obj_player`. This event will be triggered whenever `instance_destroy()` is called on the player.
- The player needs to play a death sound. For this, you will use the built in function `audio_play_sound()`. This function takes in three variables, the audio source, a priority integer and a boolean to determine if the sound should loop. For now, you do not need to worry about priority, as long as it is a positive number. For the audio source, use `snd_kill_character` from the resources menu. Make sure that *loop* is set to `false`.
- You also need to restart the room when the character dies. Call the `room_restart()` function to do this.
- Run the game, and jump on the spikes. **You should be able to hear the character say "ow!" The room should also restart.**

Part 3-B: Flyers

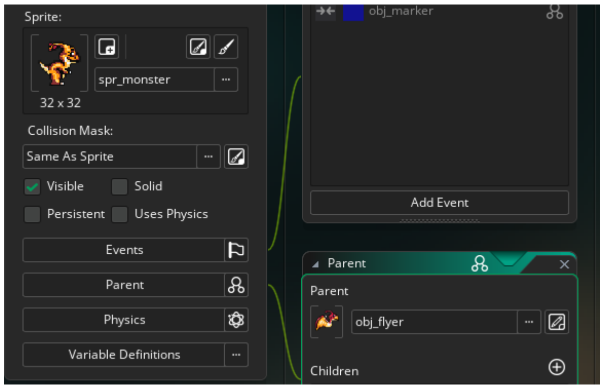
- Now onto our first enemy, `obj_flyer`. You will need this enemy to perform two simple actions, move and turn around.
- Create a variable definition in `obj_flyer` and call it `hsp`. Set it equal to `2`.
- In `obj_flyer`'s **Create** event, set its `hspeed` equal to `hsp`.
- Next, the enemy needs to turn around once it has hit a wall. Add a **Collision** event in `obj_flyer` that triggers with `obj_block`.
- In that event, you need to reverse the flyer's `hspeed`. The easiest way to do that is to set `hspeed` to itself, multiplied by `-1`.

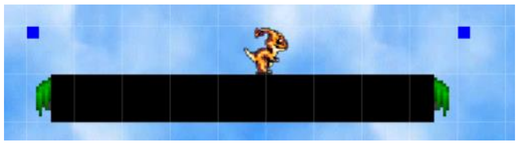
Copyright © 2019 - 2023, Julio C. Bahamón, UNC Charlotte
All Rights Reserved.

- Next, the flyer's sprite needs to reflect the direction it is moving. You will use `image_xscale` for this, each time the flyer changes directions. Do the same here that was done for `hspeed`, set it to itself times `-1`.
- Run the game. **You should see the fliers moving to the right, hitting a wall, and turning around**.
- One more thing. You may want the flyers to patrol a section of the sky that is not blocked off by `obj_block`. For this, you can use an object called `obj_marker`. This is another invisible object, but you will not be tiling on this one.
    - Make another **Collision event**, this time with `obj_marker`.
    - Copy and paste the code from the collision with `obj_block`.
    - Put at least one marker in the room that blocks the path of a flyer.
- Run the game. You should see the flyer moving until it hits that invisible point, then turning around.

Part 3-C: Monsters

- Next, we need land monsters. These should behave nearly identically to the flyers, except that they stay on the ground, and turn around after hitting the edge.
- Open `obj_monster` and select the **Parent** menu. Select `obj_flyer` as its parent.
- Run the game. Note that `obj_monster` has inherited `obj_flyer`'s behavior, including the monster on the platform hovering off the edge and continuing its momentum.
- There is a very simple fix to this. To avoid having to deal with gravity, put markers at the edges of the platform that your monster is standing on (see image below).
- Run the game. The monster should hit the edge of the platform (actually hitting the marker) and turn around.


Sprite:
spr_monster
32 x 32
Collision Mask:
Same As Sprite
Visible: ✓
Solid
Persistent
Uses Physics
Parent
Parent: obj_flyer
Children



A Quick Note: Scope

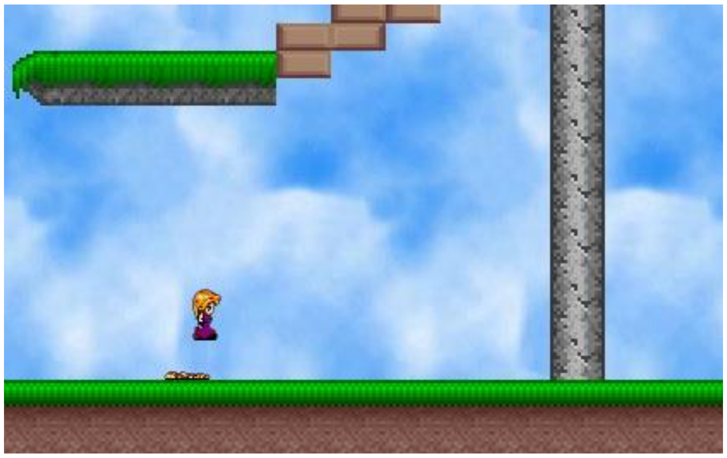
The marker method for ledges works well for a small project. However, when projects start to get bigger in scope, it could be tedious to place markers on each and every ledge an enemy could exist on. For bigger projects, it will often be faster to automate these tasks. Following is an optional challenge you can do to practice this mindset by setting `obj_monster` to detect ledges and turn around on its own.

- Delete the markers from the edges of the platform.
- When the character moves, check if the next step will be standing on a block. If not, turn around. *Hint*: use `instance_place(x+?, y+?, obj_block)`. You will need to place this in an event that triggers each game update.

Copyright © 2019 - 2023, Julio C. Bahamón, UNC Charlotte
All Rights Reserved.

3-D: Squashing Enemies

- Go back to `obj_player` and create a **Collision** event for `obj_flyer`. During this collision, you need to determine if the player is safely jumping on an enemy or should be destroyed by the enemy.
- Consider what would be required to recognize a successful jump on an enemy.
  - First, the player needs to be moving downward.
  - Second, the player needs to be physically above the enemy. Remember that `y` increases from top to bottom (`y = 0` at the top of the screen).
  - If both of these conditions are met, then a successful jump has been made and the enemy should be destroyed.
- For the first condition, we need to check if `vspeed` is positive.
- For the second condition:
  - We need to use the built in GameMaker variable named `other`. Recall from the previous workshop that `other` is a **Collision** event specific variable that holds a reference to the instance that is being collided with.
  - Check if the player's `y` value is higher on the screen than `other.y`
- Write an `if` statement to check the conditions. If both conditions are true, call `instance_destroy(other)`, otherwise call it on the player in the `else` block.
- In many platformer games with stomping on enemies, the character will bounce off of the enemy. On a successful jump, set `vspeed` directly to facilitate this.
- Add a **Destroy** event in `obj_flyer`. In it, use instance_create_layer() to create an instance of `obj_monster_dead`, to represent it being squashed.
- Next, play `snd_kill_monster`, just as we did for the player's death.
- Finally, open `obj_monster_dead`. This should work similarly to the explosions in the previous workshop. It should exist on screen for a moment, then destroy itself.
- Run the game. **You should be able to squash enemies now.**



Copyright © 2019 - 2023, Julio C. Bahamón, UNC Charlotte
All Rights Reserved.

Part 4: Lives

Next, we are going to implement multiple lives for the player. A **persistent** controller object will be needed to keep track of lives when the room restarts. You will need to use **global variables**. You will also need to **draw lives** on the screen.

4-A: The `obj_controller` Object

- Go to `rm_main` and add `obj_controller` into the room. It does not matter where, just make sure you can find it later.
- Open `obj_controller`. This object has some default functionality already implemented.
  - **Persistent**: The controller is marked **Persistent**. Persistent objects do not get destroyed when the room changes or gets reset. Furthermore, when the room gets reset, new versions of persistent objects do not get created. That means that when the player gets destroyed and the room resets, the controller will not reset its progress. This is very useful for tracking things like lives. Note that persistent objects will still get overwritten and deleted if the game is reset.
  - `global.game_over`: In the **Game Start** event, a global variable named `game_over` is instantiated. Global variables can be called from any object once they are instantiated. More on global variables can be found in the GameMaker manual.
  - ***lives/score***: GameMaker has three built in global variables, `lives`, `score` and `health`. You use them just typing `score`, `global.score` is not necessary.
  - **Draw GUI**: Inside the Draw GUI event is some basic GUI code, as well as a spot for you to draw lives (see `TODO` comment). You can use this code for inspiration for your own projects. You can also find more drawing functions in the GameMaker manual.

- **Run the game and take a look at the top left corner of the window**.

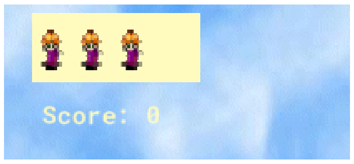
- Your goal is to draw `spr_life` for each life the player has. This can be accomplished using iteration (a for loop) and draw functions.
  - `spr_life` is a `24 x 24` pixel sprite, centered at the top left.
  - Create a `for` loop that starts `i` at `0`, and runs as long as `i < lives`.
    - Inside the loop, call the `draw_sprite()` function.
    - Four parameters are required, the sprite to draw, the frame of that sprite and the `x` and `y` values.
      - For the sprite, use `spr_life` and `0` for `subimg`, since `spr_life` only has one frame.
      - For `x`, the first life needs to be drawn at `x` position `50` (so that it fits inside the rectangle,) and each subsequent sprite needs to be drawn `24` pixels to the right of the one before it. To do this, use `(50 + 24 * i)` for the `x` parameter. The first life will draw at `50`, the next at `74` and the last at `98`.

Copyright © 2019 - 2023, Julio C. Bahamón, UNC Charlotte
All Rights Reserved.

- For **y**, use **15**, since you want to draw the sprite about **15** pixels down from the top of the screen. The code should look as below:

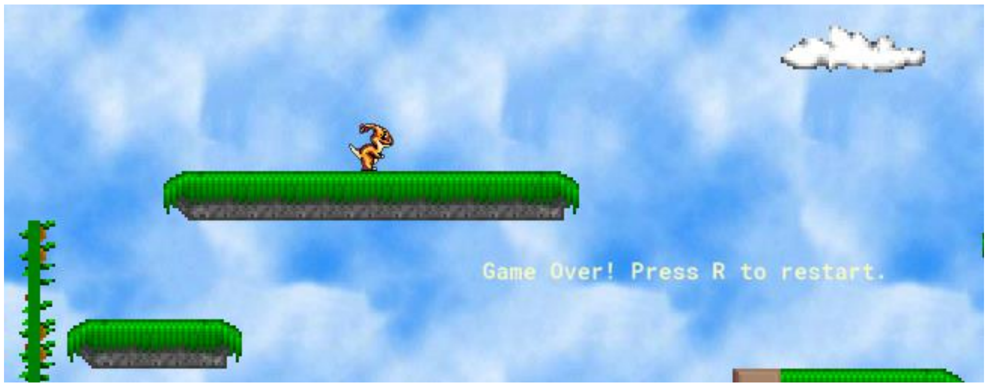
```
for (var i = 0; i < lives; i++) {
    draw_sprite(spr_life, 0, (50 + 24 * i), 15)
}
```

- Run the game. You should see 3 lives at the top of your screen, as well as an indicator for score.



4-B: Updating Lives

- Open **obj_player** and go to the **Destroy** event.
- Decrease **lives** by one. Remember that **lives** is a global variable, so you should **not** call **obj_controller.lives**.
- If **lives > 0**, restart the room. Else, set ***global.game_over*** to **true**.
- Finally, go back to **obj_controller** and create a new event, **Key Down - R**.
  - In the code for this event, if **global.game_over** is **true**, call **game_restart()**.
- Run the game.
  - Let your character be destroyed three times, making sure that a life is removed each time.
  - After losing three lives, you should be presented with a message asking you to press **R** to restart.
  - Pressing **R** should restart the game.



Copyright © 2019 - 2023, Julio C. Bahamón, UNC Charlotte
All Rights Reserved.

Part 5: Score

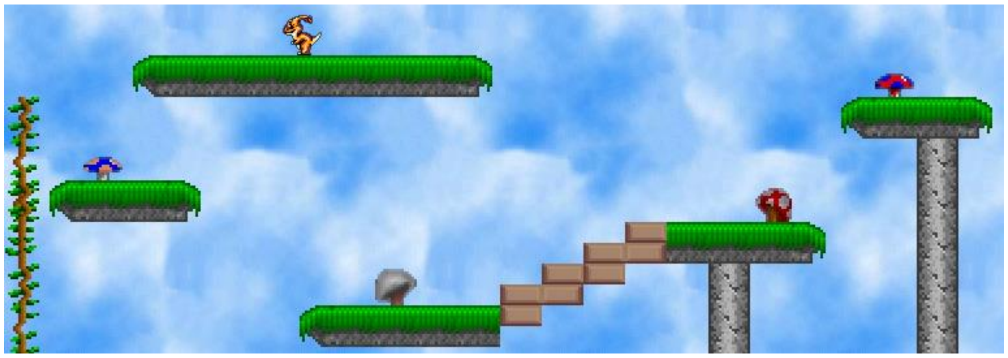
We left **scoring** alone in the last section, but now it is time to implement it. There will be two ways for the player to score points: defeating enemies and collecting mushrooms. We will also need to implement **ima** for the mushrooms.

5-A: Defeating Enemies

- Open `obj_flyer` and go to the **Destroy** event.
- Increase `score` by `10` points.
- Run the game. Each time you jump on an enemy, score should go up by 10.

5-B: Collecting Mushrooms

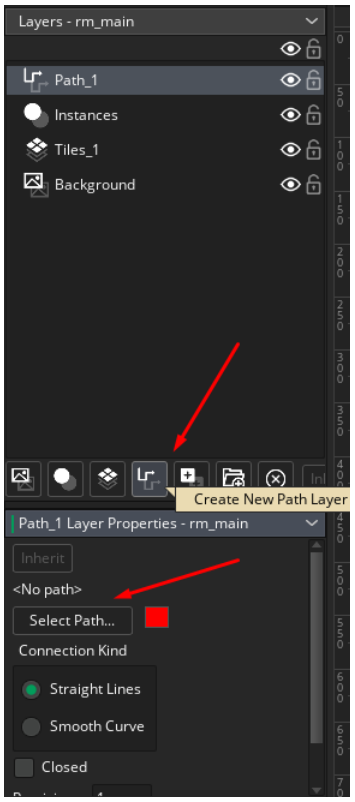
- Open `spr_mushroom`. Notice that the mushroom has many different mushroom sub-images, each being represented by a different frame of animation.
- Open `obj_mushroom` and add a **Create** event. Set `image_speed` to 0. This will prevent the mushroom sprite from animating.
- Each mushroom needs to be represented by a random frame.
    - To do this, we can use the `irandom()` function. `irandom(9)` selects a random integer between 0 and 9.
        - There are 10 mushroom sprites.
        - Sprite sub-images are zero-indexed.
    - Set `image_index` equal to the value returned by the function call.
- Go to `obj_player` and add a **Collision** event with `obj_mushroom`.
    - Destroy the other object
    - Play `snd_get_mushroom`
    - Add `5` to the `score`.
- Place at least 5 mushrooms in the room, anywhere you want, then run the game.
    - Each mushroom should be a different color.
    - Make sure that colliding with a mushroom updates the score.



Copyright © 2019 - 2023, Julio C. Bahamón, UNC Charlotte
All Rights Reserved.

Part 6: Paths

One more enemy is required. This enemy will **patrol a path** looking for the player, then lunge at the player when it gets too close.



6-A: Creating the Path

- Right click on Paths in the resources menu and select ***Create Path***. Name the new path `path_flyer`.
- Add a few points to the path, the location doesn't matter yet.
- On the left pane, check **Closed** in the properties, so the path will loop.
- Go to `rm_main`. In the Room Editor, on the left side click "Create New Path Layer" in the toolbar that is just under the list of laters.
- Next, go to the new properties window and select `path_flyer`.
- Select the line to add a new point and select a point to drag it around. Clicking on empty space will also create a point, but you have less control over which other points it is attached to. Deleting a point will bridge its two adjacent points.
- Draw a path that avoids any obstacles in the way. Make sure to select another layer when you're done, or else you will end up accidentally creating extra points.

6-B: Attaching Enemy To Path

- One more enemy is required.
  - Go to the resources menu and create a new object named `obj_patroller`.
  - Set its parent as `obj_flyer` and set its sprite to be `spr_flyer`.
- Create a new variable definition called `fly_path`. Change its type to ***Asset*** → ***Paths*** and set its value to `path_flyer`.
- Right click on the **Create** event and both collision events, and select Override Event for each of them.
- Open the **Create** event. Use the `path_start()` function to attach the new enemy to our path. This function takes in 4 variables: the path in question, the path speed, ending action and whether the enemy should follow the path absolute or relative to its position. Set path to `fly_path`, speed to `hsp`, end action to `path_action_restart` and absolute to `true`.
- Drag a patroller into the room and run the game.
  - **You should see the enemy move along the path you had laid out for it.**
  - If the enemy gets stuck, you can adjust the path in the room editor.

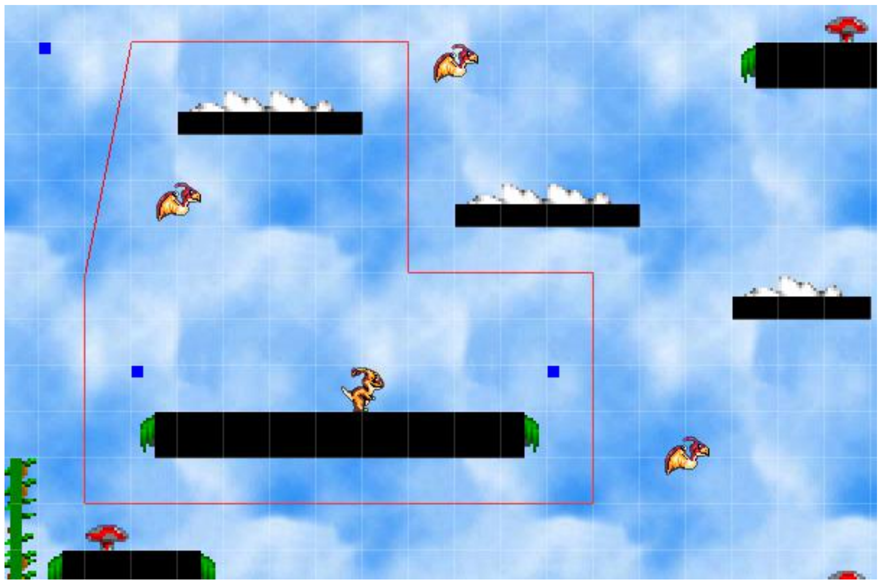
A Quick Note: Absolute

As you can see, setting absolute to `true` is very useful for having an enemy patrol a specific area. However, setting it to `false` is often useful for more general tasks. For example, suppose you wanted a flying enemy that would move back and forth in a zigzag pattern. You could create a zigzag path and add it in the room wherever. Then, set relative to `false`, and enemies would be able to move in a zigzag fashion wherever they are.

Copyright © 2019 - 2023, Julio C. Bahamón, UNC Charlotte
All Rights Reserved.

6-C: React to Player

- Go back to `obj_patroller` and add a new variable definition called `attack_range`. Set its value to `100`.
- Add a new **Step** Event. In it, we want to check if the player is too close. If `true`, detach from the path and start chasing the player.
  - First, check if the player exists. If `true`, use the `distance_to_object()` function to find the distance to the player object. If that distance is less than `attack_range`, proceed.
  - Next, call `path_end()` to allow the patroller to break from the path.
  - Set `direction` equal to `point_direction()` to aim at the player. This function expects four (4) parameters, use the current object's `(x,y)` for the first two and the player's `(x,y)` for the last two.
  - Finally, set `speed` to be equal to `hsp`.
- Run the game. **When the player gets close to your patroller, it should leave the path to start chasing after the player.**



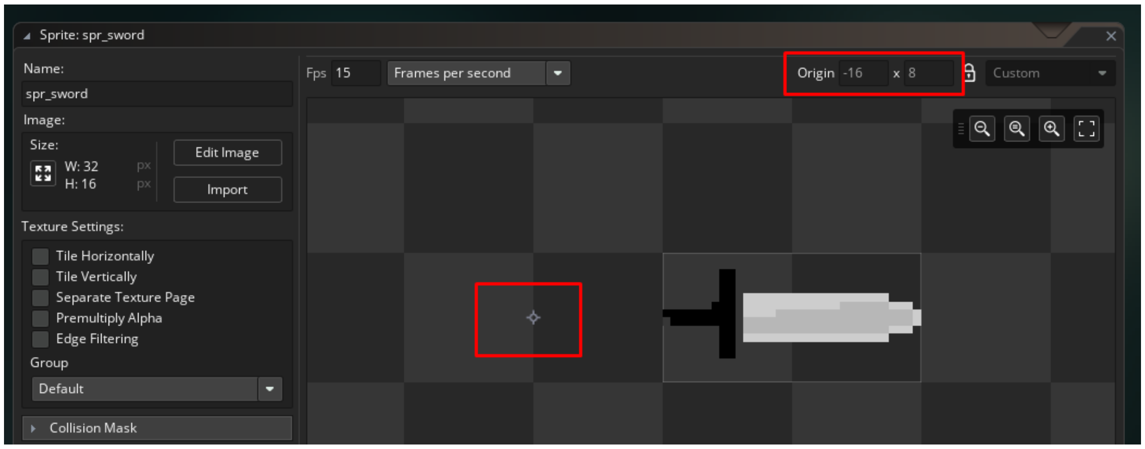
Copyright © 2019 - 2023, Julio C. Bahamón, UNC Charlotte
All Rights Reserved.

Part 7: Melee

Many projects will incorporate **melee** as a primary attack mechanism. There are many ways to do this, but for this part, you will be implementing a basic sword collision system.

7-A: Creating the sword

- Go to Sprites and open `spr_sword`.
    - Note that the origin is at a custom point, set to `(-16, 8)`.
    - **Do NOT to not change this**.
- Create a new object called `obj_sword`, then set its sprite to `spr_sword`.



- Open `obj_player` and go to the **Step** event.
    - Within the 'not climbing' else block, you will need to add a new keyboard check, except this time you will use the `keyboard_check_pressed()` function.
    - Inside the parentheses, write `ord("Z")`.
    - A quick rundown on keyboard input can be found in the GameMaker manual.
        - ***Note****: The official documentation uses apostrophes for chars, but that has been deprecated in recent versions of GameMaker. Make sure to use quotes.*
- `keyboard_check_pressed()` works the like `keyboard_check()`, except that it only returns `true` on the first frame that the user pressed a key. This is used for any inputs where you would not want the result to be triggered each frame.
- Inside the input block, write `instance_create_layer()` to create an `obj_sword` and fill in the necessary parameters.

```
if (keyboard_check_pressed(ord("Z"))) {
    instance_create_layer(x, y, "Instances", obj_sword);
}
```

Copyright © 2019 - 2023, Julio C. Bahamón, UNC Charlotte
All Rights Reserved.

7-B: Using the sword

- Open the `obj_sword` object.
- Create a new Variable Definition called `swing_timer`.
  - Set its value to `room_speed / 4`.
- Add a **Create** event. In it, you'll need to do two things
  - Set `alarm[0]` equal to `swing_timer`.
  - Set `image_xscale` equal to your `obj_player`'s `image_xscale`.
    - This will make the sword face in whichever direction your player is facing.
- Add an **Alarm 0 event**.
  - When the event is triggered, have the sword destroy itself.
- Add a **Collision** event with `obj_flyer`.
  - When this gets triggered, set the other object to be destroyed.
- Finally, add a **Step** event. Check if the player exists, then set the sword's `x` and `y` values to be equal to the player's `(x, y)` coordinates. This will make sure that the sword moves around with the player.
- Run the game and test using the sword.
  - **The sword should be a bit ahead of the player at all times and follow the player as they move.**
  - **It should also disappear after a little bit of being on screen and destroy any enemies it makes contact with.**
- At the beginning of the part, the origin was mapped to be outside of the sprite itself. You may have noticed that it lines up with the player's origin so that when its x and y were set, it would protrude just a bit away from the player. Lining up the origin like that is a useful trick for any time you want two sprites to link up.

A Quick Note: Melee

If you wanted to include melee combat in your projects, what you just learned should be easy to extend. For example, if you want the sword to swing, you can use `image_angle` to add rotation. You could animate the player's sprite to do a custom attack animation, while spawning the sword and unchecking `Visible` to use as a hitbox. You can also create a custom swing effect sprite and use that as the hitbox. If you have your own system, you are free to do use that system instead. Whatever you decide to do, it might be helpful to keep these basics in mind.

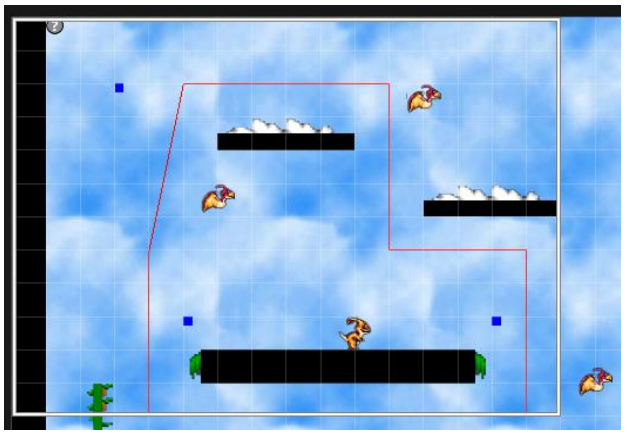
Copyright © 2019 - 2023, Julio C. Bahamón, UNC Charlotte
All Rights Reserved.

Part 8: Viewports

So far, you have been able to view the entire screen whenever the game is running. Next, you will use **views** to make the game more challenging by restricting what the player can see. You will also use views to create a simple **minimap**.

8-A: Setting the Viewport Size

- Open `rm_main` and go to the Room Editor on the left side of the screen. (If this is not visible, on the top menu select *Window* → *Inspector*.)
- Under ***Room Settings***, note that the current room size is `1024x768`.
- Open the drop down menu that says *Viewports and Cameras*, then check ***`Enable Viewports`***.
- Open **Viewport 0** and check ***Visible***.
  - You will see two sets of properties, **Camera Properties** and **Viewport Properties**.
  - **Camera Properties** will determine the amount of the room that will be visible to the player, while **Viewport Properties** will determine the size that the player will see it. For example, if the **Camera Properties** were `100x100` and the **Viewport Properties** were `200x200`, the camera would pick up a `100x100` block inside the room, and the Viewport would blow that up to `200x200`.
- Set the **Camera Properties** to `512 x 384`. Note that this is half of the room size.
- You should see a white box appear in the room. This represents the camera size of the viewport within the room.
- Run the game. **You should see the top right corner of the room**, but this would render the game unplayable!



8-B: Following the Player

- Go back to Viewport 0 and in Viewport Properties, find the **Object Following** property.
  - Set it from No Object to `obj_player`.
- Run the game again. **It should now follow the player around, but it only moves when the player is very close to the edges of the screen**.
- Go to **Horizontal Border** and **Vertical Border** and set them to `128`. This will move the edge of the screen whenever the player is within `128` pixels of the edge. However, the screen will only move to the edges of the room and not go beyond that. (Feel free to adjust this if you want to make it more comfortable.)
- Run the game. You should see the player being given a more comfortable view to move around with.

Copyright © 2019 - 2023, Julio C. Bahamón, UNC Charlotte
All Rights Reserved.

8-C: Creating a Minimap

- Close **Viewport 0**, then open **Viewport 1** and check Visible.
- Viewports draw on top of each other in the order that they are numbered. That means **Viewport 0** draws first, then **Viewport 1** draws on top of 0 and so on.
- For this, you will want to capture the entire room, then display it using a small portion of the screen.
    - Go to **Viewport Properties** and set the viewport dimensions for Viewport 1 to `128x96`.
    - Leave the camera properties as is.
- Run the game. **You should see a copy of the screen being shown, but it's drawn behind the lives and score indicators**.
- Go back to *Viewport Properties* and find `X Pos` and `Y Pos`. Set them to `32` and `96`, respectively.
- Run the game. **You should see an updating representation of the full room, now set in a more comfortable position**.

A Quick Note: Viewports

The primary purpose of a viewport is to implement screen scrolling, as you saw in 8-B. On the surface, there does not seem to be much more than that. However, the point of you making a minimap is to show an example of how viewports can be used in interesting ways. One common use would be to put something in the room that is inaccessible to the player, then using a viewport to display that part of the room. You could also use the viewports for any UI elements that would be tricky to put in a GUI event; place them in a hidden part of the room, then display them to the player using a viewport.

Copyright © 2019 - 2023, Julio C. Bahamón, UNC Charlotte
All Rights Reserved.

Part 9: Changing Rooms

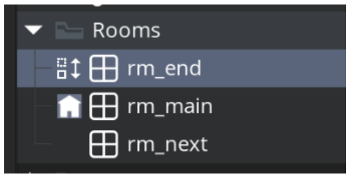
In this section, we will cover the necessary concepts to be able to **change rooms**, including room changing functions and **persistence**.

9-A: Creating a New Room

- For this part, you will need to create a new room. This new room will need to have a player in it and a level exit, as well as some tiling, background and other objects to interact with. Further, viewports and room size are room specific, so the second room would need to be manually lined up with the first room. You have two options for this.
- **Option 1**: Creating a new room
    - In the ***Resources* menu**, right click on Rooms and select "Create Room." Rename the new room to `rm_next`. Go to Room Order in Quick Access, then click and drag `rm_next` so that it fits in between `rm_main` and `rm_end`.
    - You will need to include all the features mentioned above, but you otherwise have free reign to design a level.
- **Option 2**: Duplicating `rm_main`
    - In the resources menu, right click `rm_main` and select "Duplicate."
        - Rename the new room to `rm_next`.
    - Delete `obj_controller` in `rm_next`.
    - The newly created room will have everything you need to continue. However, the room needs to be visibly different to making testing the next part clearer. Edit the room so that it becomes clear which room you are working in at a glance.

9-B: Moving Between Rooms

- Open up `obj_player` and create a new **Collision** Event with `obj_level_exit`.
- Use room_goto_next() to move to the next room on collision.
    - This will cause the next room in the Room Order to be loaded. (`rm_main` → `rm_next` → `rm_end`).
    - To change the room order, hover over one of the rooms in the asset browser and click on the icon that appears to the left of the room name.
    - For more detail on room changes, see the GameMaker manual.



- Run the game. **Colliding with the level exit should start the next room**.
- Notice that changing rooms does not wipe out `obj_controller`, like it does with the other objects in the room, maintaining the GUI.
    - Anything that has **Persistent** set to true will not change in between rooms.

Copyright © 2019 - 2023, Julio C. Bahamón, UNC Charlotte
All Rights Reserved.

Part 10: Cheat Codes

If you designed your `rm_next` to be a little difficult, you may have found it tricky to test if `obj_level_exit`'s collision worked. In this section, you will be implementing codes that make it easier to test the game. You may also use them for commands available to the player. For this part, all the functionality will be implemented inside `obj_controller`. After each section, run the game to ensure that the code works.

10-A: Quitting the Game

- Create a new **Key Pressed - Escape** Event.
- Use `game_end()` to close the game. (If you already have something like this implemented, please make this event and write a comment that points us to it.)

10-B: Cheat Codes

- Create a new **Key Down - Alt** Event.
- For each code, make sure to use `keyboard_check_pressed()`. You will only want each code to trigger once per press, not once per frame. This way, you can hold `Alt` and press a key to trigger the codes.
- Set the following codes:
  - Press `Alt - R` to restart the room
  - Press `Alt - G` to restart the game
  - Press `Alt – L` to increase lives by 5
  - Press `Alt – N` to move to the next room
- There are plenty more ways that cheat codes can be useful. Add at least one more code of your own that you could use to test things.

Copyright © 2019 - 2023, Julio C. Bahamón, UNC Charlotte
All Rights Reserved.

Additional Items Worth Considering:

- While in the code editor…
  - Middle click on a function name opens the GameMaker manual entry for it.
  - Right-click on a function name displays a context-sensitive menu.

- 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.

- 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.

- Paths 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.

- *GameMaker* has an exhaustive built-in manual, which includes sample code.
  - You may use any code found in the manual as part of your projects.
    - Make sure to modify it as needed.
    - Make sure to include a citation to the source.
  - To access the manual, go to: https://manual.yoyogames.com/#t=Content.htm

Copyright © 2019 - 2023, Julio C. Bahamón, UNC Charlotte
All Rights Reserved.