Make a Platformer Using Godot

Based on Brackeys' Godot Tutorial

Go here: https://www.youtube.com/watch?v=LOhfqjmasi0&t=401s for the full video with visuals. This is intended as a supplement to help with following the tutorial.

Setup

Note: Some of this is covered in the video, but I'm customizing it for our school computers.

1. Go to https://godotengine.org and download the latest version of Godot.
2. Go to https://brackeysgames.itch.io/brackeys-platformer-bundle to download the assets for the tutorial.

- These are free, so when we click the *Download Now* button, we can then click *No thanks, just take me to the downloads*.
- Then click the *Download* button.

3. Go to our **Downloads** folder in Windows Explorer.
4. Extract the Godot program ZIP folder.

- Right-click the file.
- Choose *Extract all...*
- Do the same for the assets we downloaded.

Keep the window with Godot open. We need to move the Godot program to a spot where it is easily accessible.

5. Move Godot: I'm going to put Godot into the **Documents** folder. we can also put it in the **Desktop** folder if we wish.

- Single-click (left mouse button) on the Godot program so that it is selected. (Don't run the program yet.)
- Use CTRL + X to cut the file.
- Go to our Documents folder. (Look on the left-side of the Windows Explorer window.)
- Make a new folder called *Godot*.
- Use CTRL + P to paste the file in the new location.

6. Make a desktop shortcut for Godot:

- Right-click on the Godot program.
- Choose *Show more options*.
- Find **Send to...**
- Choose *Desktop (Shortcut)* to make shortcut to Godot. we can now double-click this shortcut to start the program.

Getting Started

1. Start Godot if we haven't already (double-click the new shortcut we created on our Desktop).
2. Godot begins with a start screen when we first launch it. Once we have projects here, they'll be available for we to open. For now, click *Create New Project* or the *Create* button in the upper left.
3. In the **Create New Project** window, we need to do a few things:
    1. Give our project a name. For this example, I'm using **BrackeysPlatformerTutorial**.
    2. Make sure **Create Folder** is selected. (The button should be blue.)
    3. Under **Project Path**, we need to tell Godot where to save our project. Click *Browse*.
    4. Go to either our **Documents** or our **Desktop** and create a new folder called **GodotProjects**.
    5. Open the folder and click *Select Current Folder*.
    6. Make sure **Project Path** looks right. For example, mine says at the end **Documents\GodotProjects\brackeys-platformer-tutorial**.
    7. Under **Renderer** click *Compatibility*. This will make sure our game runs smoothly on any computer, as well as mobile and web platforms.
    8. Click *Create* and wait for Godot to load. It is usually very fast to start.
4. Once the project is loaded, we'll be in the 3D view. Look at the middle of the top of the window, and we'll see 2D, 3D, Script, Game, and AssetLib. Click *2D* to see the 2D perspective.

Importing Assets

Assets are the sprites, textures, fonts, sounds, and other elements that go into making our game. We need to import these from the download we did earlier. Later, we can also change these assets, modify them, or create our own.

It's important to keep our project organized with folders. This way, we can easily find assets, scripts, scenes, and other files our game needs.

1. Find the FileSystem window. This is usually at the bottom left of the screen.

2. Right-click over **res://**.

3. Move our mouse over *Create new* and choose *Folder* from the menu.

4. Name this folder **assets**.

    1. Open up Windows Explorer and find the folder we downloaded called **brackeys_platformer_assets**.
    2. Select all of the folders inside this folder.
    3. Drag these folders into our Godot window and right on top of the **assets** folder in the FileSystem. Make sure our mouse is actually on top of **assets** before letting the mouse button go. All of the folders will be imported into our project. A down-arrow will appear to the left of the **assets** folder name, and we can click on this to view the folders and files inside of it.

5. Create two other folders in the FileSystem the same way we created the **assets** folder: one called **scenes** and one called **scripts**.

- Make sure to right-click on **res://** each time; otherwise, we might create these folders inside of another one.

How Projects Are Structured in Godot

Godot uses **nodes** and **scenes** to organize projects.

- Each node takes care of a specific part of the program, such as a player's sprite, movement, and collisions.
- Nodes are arranged into scenes, so we might have a player scene with all of the nodes controlling the player, and another scene with the nodes controlling the enemy.
- Scenes can be reused in other scenes (such as Coin scene being used in Level 1, Level 2, and Level 3 scenes).
- Nodes and scenes create a **Scene Tree** with a **root** node at the start of everything.
    - For example:

```
├── Game (ROOT)
|   ├── Player
|   |   ├── Graphics
|   |   ├── Collider
|   ├── Enemy
|   |   ├── Graphics
└── └── └── Collider
```

Creating the Player Character (Player 1.0)

1. In the upper left, under **Create Root Node:** click *2D Scene*.
2. Name this **Game**
3. Use *CTRL + S* to save this scene.
    - In the **Save Scene As...** window, double-click the **scenes** folder to open it.
    - Click *Save* at the bottom right.
    - Note that scenes are saved as .tscn files.
4. Press *F5* to preview the game.
    - Click *Select Current* to make the Game scene the main scene.
    - The preview window will be grey since we haven't actually done anything yet.
5. Press *F8* or click the Stop icon in the upper right to stop the preview.

---

6. Press *CTRL + N* to create a new scene and then press *CTRL + A* to create new node for our player.
7. Search for *CharacterBody2D* and select it.
8. Click *Create*.

---

9. Press *CTRL + A* again to add a child node.
10. Search for **Sprite**.
11. Choose *AnimatedSprite2D* and click *Create*.

12. Find the **Inspector** on the right side of the screen. This is where we change the properties for each node.
13. Click < *empty* > next to **Sprite Frames**.
14. Under **New** choose **Sprite Frames** and then click on *SpriteFrames* again.
15. At the bottom middle of the screen find **Animation Frames:** and click the icon that looks like a box with lines. This allows us to add animation frames from a *sprite sheet* (a collection of sprites in one image file).
16. Open **assets**.
17. Open **sprites**.
18. Choose **knight.png** and click *Open*.
19. We can click the **plus sign** in the upper right of the **Select Frames** window to zoom in on the sprite sheet. Since all of these sprites are in a single image, we need to tell Godot how to divide them up into separate pieces for our animations.
20. On the left, make **Horizontal** be 8 and **Vertical** be 8. Each sprite should now be in its own box.
21. Click the four frames next **IDLE** from left to right.
    - The order we click these will determine the order of the animation frames. we'll see a number appear next to each sprite (starting with **0**) as we click them.
22. Now click *Add 4 Frame(s)* at the bottom.

- The sprite will appear very small in the 2D window. To zoom in, place our mouse cursor over the character and use the mouse wheel to zoom. The window will zoom to wherever we have our mouse cursor.
- We can move the window around by holding the **Space Bar** and dragging with the mouse.
- The sprite will be blurry when we zoom in on it. This is because Godot's trying to make it look smoother, but pixel art needs hard lines to look good.

23. Click on *Project* at the top of the screen.
24. Choose *Project Settings...*
25. Scroll down and look for **Rendering** and click on *Textures*.
26. Click next to **Default Texture Filter** and choose *Nearest*.
27. Click *Close*.

- The character should look clear now.

---

28. Make sure **AnimatedSprite2D** is selected on the left.
29. Go down to **Animation Frames:** and click on the play button to test the animation.
30. Increase the FPS (it currently says **5.0 FPS**) to **10.0**.
31. Under **Animations:** double-click on **default** and rename this to **idle**.

- Also click on button that has an **A+** inside of a tag. This is the **Autoplay on Load** button, and will start the animation when the game starts.

32. Click on the character in the main window.
33. Drag the character so that the bottom of the sprite is on top of the horizontal line and centered on the vertical line.

---

34. Click on *CharacterBody2D* on the left.
35. Click the plus sign above it.
36. Search for **CollisionShape2D** and *Create* it.
37. In the **Inspector** find **Shape** and click on < *empty* >.
38. Choose *CircleShape2D*.
39. Click and drag on the orange point on the right of the blue circle (i.e. the **hit box**); this marks the radius of the circle.
    - Dragging to the left makes it smaller, and dragging to the right makes it bigger.
    - Make it smaller to fit just inside of the character's sprite.
    - Make sure the bottom of the circle is aligned with the bottom of the character, on the horizontal line.
    - our hit boxes / collision shapes don't have to be perfect. we'll want to test them later to see if they work correctly and give the player a sense of fairness.
    - If we want our collision to be exactly as I have mine, we can actually specify the exact values:
        1. Click on *CircleShape2D* next to **Shape** in the **Inspector**.
        2. Change the **Radius** to **5.0 px**.
        3. Click on the arrow next to **Transform**.
        4. Next to **Position** change **y** to **-5.0 px**.

---

40. Rename the **CharacterBody2D** to **Player**.
    1. Click on *CharacterBody2D* on the left.
    2. Press *F2*.
41. Save the scene.
    1. Press *CTRL* + *S*.
    2. Godot usually opens the last folder we opened. Open up the **scenes** folder if it's not already opened.
    3. Click *Save*.

---

42. Open up the **Game** scene by clicking on the tab at the top of the main window that sames **game**.
43. Go to the **FileSystem** and expand the **scenes** folder by click the arrow.
44. Drag the **player.tscn** into the main window.

- The player will be very small, so zoom in on it.

- We now need a **camera** to follow our player.

45. Click on the **Game** node.
46. Add a **Camera** node.
    1. Press the plus sign under **Scene**.
    2. Search for **Camera**.
    3. Choose *Camera2D* and *Create* it.

- The camera viewport (a light-purple line) will be too big for our game.

47. In the **Inspector** find the **Zoom** and change this to **4 x 4** (i.e. x = 4.0 and y = 4). we only need to change the x to **4.0** and the y will change to match.

---

48. Go back to the **Player** scene (click on the **player** tab in the main window or double-click the **player.tscn** file in the **FileSystem**.)
49. Click on the *Attach Node Script* button in the Scene window on the left; it looks like a scoll with a plus sign.

- We're going to use a template to fill in some basic movement for us, and it's already chosen **CharacterBody2D: Basic Movement** for us.

50. In **Path** choose the file folder and change it to the **scripts** folder.
    1. Click the up arrow next to the word **Path**. we may have to click it more than once to see the **scripts** folder.
    2. Open **scripts** (double-click the folder).
    3. Click *Open*.
51. Click *Create*.

- This will open up the **Script** window.

52. Play the game with *F5*.

- The player immediately falls because it has nothing to stand on.

---

53. Go the **game** tab.
54. Go back the **2D** view.
55. Click the *Game* node.
56. Click the plus sign to add a new node.
57. Search for **StaticBody2D** and *Create* it.
58. Click on the node *StaticBody2D*.
59. Press *CTRL + A* to add a child node.
60. Search for **CollisionShape2D** and *Create* it.
61. In the **Inspector** click on < *empty* > next to **Shape**.
62. Choose *WorldBoundaryShape2D*.
63. Move the collider under the player:
    1. Click *StaticBody2D* (we need to move the entire StaticBody and not just the collision mask.)
    2. Click the **Move** tool in the main window. It is next to the **Select** tool (the mouse arrow) and looks like a dot with four arrows pointing out from it. We can also press *W* on the keyboard to select it.
    3. Drag the collider under the player.
64. Switch back to the **Select** tool by clicking it or pressing *Q* on the keyboard.
65. Test the movement and jumping by pressing *F5*. Use the **arrow** keys to move and the **space bar** to jump.

- The player's movement will be very fast and the jump higher than we might want.

- We can modify the **player.gd** script to change this.

66. Switch to the **player** tab in the main window.
67. Click *Script* and make sure **player.gd** is selected (it's to the left of the main script window).
68. Change the **const SPEED** to something lower, such as **130.0** and the **const JUMP_VELOCITY** to **-300.0**
69. Press *CTRL + S* to save.

70. Press *F5* to test the movement speed and jumping.

- We will next add the world to give us something to look at.

Worldbuilding 1.0

---

1. Remove the **StaticBody2D** node from the game:
    1. Right-click it and choose *Delete*
    2. Click *OK*
    - We'll be replacing it with actual platforms.
2. Click the **Game** node.

---

3. Add a new node and search for **TileMapLayer**.
    - Note: Brackeys' tutorial uses the **TileMap** node, which is older and now *deprecated*, meaning Godot suggests we not use it. If we come across other tutorials that still use the **TileMap** node, sometimes it's better to use it rather than the new one; we'll have to experiment to see (which is how I know).
4. *Create* the node.
5. In the **Inspector** find **Tile Set** and click < *empty* >.
6. Under **New** choose *Tile Set*.
7. At the bottom of the window, click on *TileSet*.
8. In the **FileSystem**, expand the **sprites** folder.
9. Drag **world_tileset.png** into the **Tile Sources** window to the right.
10. Click *Yes* to **automatically create tiles in the atlas**.

- This will draw boxes around all of the tiles. If we had a **Tile Set** that was not **16x16** pixels, we could change that in the **Texture Region** setting in the middle-bottom of the screen.
- We'll probably need to zoom in to the tile set by hovering the mouse over the tile set and using the scroll wheel. we can also click the + next to the zoom percentage.
- We can also center our view of the tile set by click the icon in the upper right corver of the **TileSet** window that looks like a point with arrows facing inward.

11. Check to make sure each of the tiles in the tile set is surrounded by a box.

- These boxes can be changed if necessary. For example, we need to change the top of the palm tree that that Godot knows it's all one piece instead of 9 separate pieces:
    1. Click the *eraser* tool or press *E* on the keyboard.
    2. Click on each of the 9 tiles or click and drag our mouse around on them. If we go too far, use *CTRL + Z* to **undo** any mistakes.
    3. Click *E* again to unselect the **eraser** tool or click the tool.
    4. Hold down the *Shift* key and while holding it, click the upper-left box of the palm tree top (it will be greyed out).
    5. Drag our mouse to the bottom-right corner of the 3x3 grid that we erased earlier.
    - This will make Godot treat the top of the palm tree as 1 piece instead of 9.
    - The other trees should be separate pieces so that we can make trees of different heights.

---

12. Click the *2D* view.

- We'll now see a grid laid over the view port.

13. Click on *TileMap* at the bottom of the window.
14. Click on a tile in the **TileMap** window.

- Remember can zoom in and out and move the window around as needed to pick tiles.

15. Click the *Paint* tool or press *D* on the keyboard.
16. Move our mouse over to one of the grid blocks under our character; we'll see a blurry version of the tile.
17. Left click to add a tile.

- There are several different tools for drawing tiles:
    - The **Paint** tool adds a single tile or a group of tiles.
        - we can drag the mouse to add a bunch of the same tile.
    - The **Line** tool can paint a single straight line of the same tile.
        - Just click and drag to make a line.
        - Lines can be diagonal as well as horizontal and vertical.
        - Use this when we need a very straight line, as if we use the **Paint** tool it's easy to accidentally add tiles we don't want.
    - The **Rect** tool makes a rectangle-shaped block.
        - Just click and drag to make a rectangle.
    - The **Bucket** tool fills in an enclosed area with a bunch of the same tile or replaces all of same tiles with another tile.
    - The **Dropper** tool picks a tile that we've already drawn from the 2D window so we don't have to find it in the TileMap.
    - The **Eraser** tool can be used with the paint, line, rect, and bucket tools to erase any tiles in various combinations.
        - For example, if we click on the **Line** tool and then the **Eraser** tool, we can erase straight lines.
        - If we click the **Bucket** tool and then the **Eraser** tool (or while the eraser is already selected), we can erase all of the tiles of a single type if they're touching each other.
        - Don't forget to turn off the **Eraser** tool when we want to start drawing again.

18. It's now up to we to draw our own level.

- Once our level is complete, we can test the game again.
- The player will fall through the level again, as none of these tiles has a collider on them.
- So we'll need to add a **Physics Layer** to the tiles we want the player to collide with.

19. In the **Inspector** find **Tile Set**.
20. Click on *TileSet* next to that.
21. Click on *Physics Layers*.
22. Click on + *Add Element*.

- We'll now see options for **Collision Layer** and **Collision Mask**. These are used to determine what game elements actually interact with each other. For now, we'll leave them on their defaults.

- We now need to determine which tiles the player will collide with and which it won't. For example, we want the player to collide with the ground, but not with decorative elments such as trees and bushes.

24. Click on *TileSet* at the bottom.
25. Click on *Paint* inside the **TileSet** window.
26. Click on *Select a property editor*.
27. Choose *Physics Layer 0*.
28. We can now paint the tiles to apply the **Physics Layer 0** to those tiles.
    - The tiles will be shaded to show the collision area; by default, it selects the entire tile.
    - If we accidentally paint a tile we didn't want to:
        1. Click the 3 dots menu under **Painting:**.
        2. Select *Clear*.
        3. Click the tile we want to clear.
        4. Click the 3 dots again.
        5. Click *Reset to default shape* to keep painting more tiles.
    - We can hold down **Shift** and click and drag while holding to select multiple tiles at a time.
    - We can adjust the shape of the collider to specific tiles.
        1. Click on a tile that we want to adjust, such as the bridges (which don't fill the entire block).
        2. Under **Painting**, we drag the white diamonds (points) to make the collider fit around the bridges.
        3. Make them fit as best we can, but be careful where the tiles join each other.
            - For example, on the bridges, we don't want the tops of the bridges to be different heights, or the player will get stuck on the edges.
        4. Click the tile again to apply the new collider shape to the tile.
            - We can click on more than one tile with the new collider shape if they're similar (such as the middles of the three different bridges).
            - We can use the **Dropper** tool to select a previous collider, and then apply it to other shapes without having to redraw it.

- Once we have our colliders painted on the appropriate tiles, run the game and make sure the player doesn't fall.

- We now need to change the **Camera** so that it follows the player as it moves through the level.

29. Click and drag the node *Camera2D* to the **Player** instance.
    - When you let go, it should appear underneath the **Player**.

- We can also make the camera smoother as it follows the player.

30. With the **Camera2D** node selected, go over to the **Inspector** and click *Position Smoothing* on.
    - The speed will be set to **5.0 pixels per second**; we can adjust this as needed to make it feel different, but we'll leave it for now.
    - We can also test the camera without the smoothing to see which setting we like better.
31. Press *F5* to test the game and make sure the camera follows the player.

Platforms

- We'll now add moving platforms to our game to make it more dynamic.

1. Click on the plus sign next to the **player** tab in the main window to make a new scene.
2. Click on the plus sign in the **Scene** window (or press *CTRL + A) to add a new node.
3. Search for **AnimatableBody2D** and *Create* it.
4. Press *CTRL + A* to add a child node.
5. Search for *Sprite2D* and *Create* it.
6. In the **Inspector** find **Texture** property.
7. In the **FileSystem**, go to **sprites** and find **platforms.png**.
8. Drag **platforms.png** over to the **Texture** property and < *empty* >.

- Because this is a sprite sheet, it shows all of the platform sprites, so we'll need to tell it which we want to display.

9. In the **Inspector** turn on **Region**.
10. Click *Edit Region*.
11. Change the *Snap Mode* to *Pixel Snap*.
12. Drag the points until the platform we want is selected, such as the grass platform at the top.
    - It's helpful to zoom in so that you can align the boundaries exactly with the pixels of the platform; otherwise, we might crop out the outlines or other parts of the platform.

- We now need to add a collider to our platform.

13. Click on the **AnimatableBody2D** node.
14. Press *CTRL + A* or the plus sign to add a new child node.
15. Search for CollisionShape2D.
16. In the **Inspector**, find **Shape** and click < *empty* >.
17. Click *RectangleShape2D*.
18. In the main window, drag the points on the collision shape to match the platform.
19. Rename the top node to **Platform**.
    1. Click on the **AnimatableBody2D** node.
    2. Press *F2* on the keyboard.
20. Save the new scene (*CTRL + S*).
    - Make sure we're still in the **scenes** folder.
21. Go to the **Game** scene.
22. From the **FileSystem**, drag the **platform.tscn** file into the game world.
    - Put the platform somewhere where you player can jump on it from both the side and the bottom.

- Test the platform to make sure it works (*F5*).

- For the platforms, we want to be able to jump up to them from the bottom, but we currently can't.
- To fix this, we'll enable **One Way Collision**.

23. Return to the **Platform** scene (click the **platform** tab if it's still open).
24. Select the *CollisionShape2D* node.

25. In the **Inspector** turn on **One Way Collsion**.

- In the main window, an arrow will appear pointing down.
- The arrow points in the direction through which the player can pass (i.e. it will pass through the bottom but not the top).
- Test our platform again to see if we can jump through it from below.

- If our player node is before the platform node in the tree, the player will jump behind the platform.
    - Objects in Godot are drawn in their order in the **Scene Tree**, so the last objects are usually on top of the first ones.
    - We can use this to provide depth to our games, but we can also override this by changing the **Z Index** of each node.
- For now, let's make sure the player is always in front of the other game elements.

26. Open the **Player** scene.
    - (Note, don't make this change in the **Game** scene, as that will only affect the player **instance** and not the actual player.)
27. In the **Inspector** find **CanvasItem** and then **Ordering**.
28. Expand **Ordering** and change the **Z Index** to a number higher than 0, such as 5.
    - Items with a higher Z Index will appear in front of those with a lower Z Index.
    - The default Z Index is 0.

- Let's now use animation to make some moving platforms.

29. Drag in a new platform instance to the **Game** scene.

- If we want a specific instance to have a specific behavior (such as a unique animation), then we ***can*** edit the specific instance in the **Game** scene.
    - We can then copy and paste a unique instance to make more of it, such as a moving platform versus a static platform.

30. This new instance will be called **Platform2** by default.
31. Select it if it's not already, and press *CTRL + A* to add a child node to it.
32. Search for **AnimationPlayer** and *Create* it.

- The **Animation** window should now be shown at the bottom.

33. In the **Animation** window, click on *Animation*.
34. Choose *New*.
35. Name this animation **move**.
36. Select the **Platform2** node.
37. In the **Inspector** find the **Transform** property under **Node2D**.
38. Click the *key* icon next to **Position** to add a **key frame**.
    - **Key frames** are the main points of any animation.
    - For example, if we put one key frame at one position, then move the object and add a second key frame, the object will move back and forth between those two positions.
39. Click *Create*.

- A new animation track was automatically added, with a key frame at the 0 time position (indicated by a white diamond).

40. In the **Animation** window, move the blue line to the 1 time position.
41. In the main window, switch to **Move** mode (press *W*)
42. Move the platform horizontally by holding down the **Shift** key and dragging the platform to its end position.
43. Click the *key* icon again (the one under **Transform** and next to **Position**).

- We can test the animation by clicking the *play* button in the **Animation** window.

- Right now, the animation only plays once. We'll need to make it loop to keep it moving.

44. At the right side of the **Animation** window is the **Animation Looping** button; it looks like two arrows in an ellipse.
45. Click the **Animation Looping** button.
46. Test the animation by click the play button.

- Notice that the animation snaps back to the original position rather than moving back and forth.

47. Click the **Animation Looping** button a second time.

- The icon changes to a double-headed arrow between two lines, indicating **Bounce** mode.
- We can adjust the speed of the animation to make the platform move faster or slower.

48. Next to the **Animation Looping** button is the time of the animation. Change it to **1.5** and test the animation again.

- The animation is longer, but the platform hangs for a second before moving back.

49. Drag the second white diamond on the **position** track to the end of the animation.

- The platform should now move back and forth a little slower.

50. Set the platform to **Autoplay on Load** (look next to the **Edit** button on the right).

- We can repeat this process to add more moving platforms throughout our level.
    - It might be a good idea to rename the platform nodes to specify which are static and which are moving, but this is up to the creator.

Pickups

- We'll create a simple coin pickup, but the principles will work for other things as well.

1. Create a new scene with *CTRL + N*.
2. Add a new node with *CTRL + A*.
3. Search for **Area2D** and *Create* it.
    - This is a node for objects that don't need to physically collide (such as we did for the platforms) but that need to detect collisions, such if the player enters the area.

4. Add graphics:
    1. Press *CTRL* + *A*.
    2. search for **AnimatedSprite2D** and *Create* it.
    3. In the **Inspector** find **Sprite Frames** and click < *empty* >.
    4. Add new **SpriteFrames** and select it.
    5. Click the **Sprite Sheet** icon (the box with grids).
    6. Open the **assets** and then **sprites** folders.
    7. Select *coin.png* and *Open* it.
    8. Use the plus sign at the top right corner of the **Select Frames** window to zoom in.
    9. On the right, change **Horizontal** to **12** and **Vertical** to **1**.
    10. Click each from from left to right, or click and drag from the left to select all of the frames.
    11. Click *Add 12 Frame(s)*.
    12. Press *F* on the keyboard to center the view and zoom in on the coin in the main window.
    13. Change the FPS in the **Animations:** window to **10**.
    14. Click the *play* button to test the animation.
    15. Set it to *Autoplay on Load* (the button to the right of the trashcan icon in the **Animations:** window).
    16. Stop the animation preview by clicking the *pause/stop* button or pressing *S* on the keyboard.

---

5. Add a collision shape:
    1. Make sure the root node is selected.
    2. Press *CTRL* + *A*.
    3. Search for **CollisionShape2D** and *Create* it.
    4. In the **Inspector** find **Shape** and add *CircleShape2D*.
    5. Resize the circle to fit the coin (radius of 5.0 px).
6. Save the Scene.
    1. Rename the root node to **Coin**.
    2. Press *CTRL* + *S*.
    3. Make sure you're in the **scenes** folder.
    4. Click *Save*.

- We can now add some coins to our game.

7. Select the **game** scene (clickt tab in the main window).
    1. Make sure the **2D** view is showing (i.e. you see your level).
    2. Make sure the **Game** node (the top, root node) is selected.
    3. From the **FileSystem**, drag in several instances of **coin.tscn** into your level wherever you would like them to be.
    4. Press *F5* to test the game.
    - Nothing should happen when we interact with the coins yet; we need to add some code to make this happen.

---

8. Create new script for the coin:
    1. With the **Coin** node selected, click the *Attach a new script...* button.
    2. For **Template** choose *Node: Default*.

3. For **Path** makes ure it's saved to the **scripts** folder.
4. Click *Create*.

- We're now in the **Script** window.
- This default template has some basic functions, but note that they all have **pass** under them, which means they won't do anything until we remove this and add the code.
- On line 1 we read:

```
extends Area2D
```

- This tells us what type of node this code is modifying.
- On lines 4-6 we see the **_ready** function:

```
# Called when a node enters the scene tree for the first time.
func _ready() -> void:
    pass # Replace with function body.
```

- We can test out this ready function by deleting the **pass** line and adding a **print** command with a message string. This will show up in the **Output** at the bottom of the screen:

9. In the **_ready** function,
    1. change:

        ```
        pass # Replace with function body.
        ```

        to

        ```
        print("I'm a coin.")
        ```

    2. Make sure that the tabbing is correct. GDScript won't work without correct tabbing. You'll see a tab icon like this >| to indicate a tab.
    3. Run the game with *F5*. In the **Output**, you should see the message **"I'm a coin."** for each coin you added.

- For this game, we actually don't need the **_ready()** and **_process** functions.

10. Remove the code except for **extends Area2D** from the coin script:
    1. Go back to the **coin** scene (click the tab in the main window).
    2. Change the view to **Script**.
    3. In the script window, select everything but **extends Area2D** and delete it.
    - We should be left with just:

```
extends Area2D
```

- What we want is for the coin to recognize when the player has entered its collision area.
- To do this, we're going to use **Signals**.
- Godot has a lot of built-in signals to use.

11. Click the **Coin** root node.
    1. On the right, next to the **Inspector** tab, choose *Node*.
    - We'll now be able to see the built-in signals for an **Area2D** node.
    1. Find the **body_entered** signal and double-click it.
    2. Click *Connect*.

- A new function has been added to our script:

```
func _on_body_entered(body: Node2D) -> void:
    pass # Replace with function body.
```

- We can also see that this is a signal because on the same line as the function is a "green arrow/right bracket" icon.

12. Add a **print** function to make sure the signal works:
    1. Delete the entire **pass** line.
    2. Add the following code, making sure it's tabbed under the function:

    ```
    print("+1 coin!")
    ```

        - This will print the message **+1 coin!** each time our player interacts with a coin.
        - Printing messages like this is a good way to make sure something works before adding in the actual code, which might be more complicated to debug.
    3. Test the game to see if the signal works.

- The message will also be triggered when a platform interacts with a coin.
- We can change this in two ways.
    1. We could check to see which body has entered the coin.
    2. We can put the Player on a different **Physics Layer**, so that only objects on that physics layer interact with the coin.
- We'll try the second method:

13. Change the **Physics Layer** of the Player:
    1. Switch to the **player** scene.
    2. In the **Inspector**, find the **Collision** property.
    - If the tab is still on **Node**, choose **Inspector**.
    1. Under **Layer** click on *1* to deselect it.

    2. Click *2* to change the physics layer.
    3. Press *CTRL + S* to save.
14. Change the **Physics Layer** of the Coin:
    1. Switch to the **coin** scene.
    2. In the **Inspector**, find the **Collision** property.
    - We don't have to change the coin's **Layer** at this point, but the **Mask**, which determines which layers the coin interacts with.
    1. Click *1* to deselect it and click *2* to select it.

- We can now also edit our coin script so that the coins disappear when the Player interacts with them.

15. Make sure we're in the **coin** scene and the **Script** view is showing.
    1. Under the **print** line (at the same tab point) add the following code:

```
queue_free()
```

    - The **queue_free()** function will remove the coin instance that we interact with as the Player.
    1. Test the game by pressing *F5* on the keyboard.

- Your code should now look something like this:

```
extends Area2D

func _on_body_entered(body: Node2D) -> void:
    print("+1 coin!")
    queue_free()
```

- For readability, I like to delete extra lines if they're empty, but keep a line between different parts of the code to make them more visible.

Dying 1.0

- Our player still just falls if it misses a jump, so let's add dying and restarting

1. Limit the Camera:
    1. Switch to the **game** scene.
    2. Select the **Camera2D** node under the player.
    3. In the **Inspector** find the **Limit** property.
    - We'll need to change the **Bottom** limit, but we'll need to measure how many pixels to limit it to.
        1. Press *M* to select the **Ruler** tool. We can also click the **Ruler** tool icon in the main **2D** window. The icon looks like a right triangle.
        2. With our mouse cursor at the baseline (the horizontal purple like running through the entire game world), click and drag the mouse to the bottom-most part of your level (i.e.

the bottom of the last rock).
3. Look for the length measurement in pixels; mine was 220 px, but yours might be different.
4. Input this value into the **Bottom** property under **Limit**.
- We should see a yellow line after we change this
1. We can also turn **Smoothed** on so the camera doesn't just suddenly stop.
- Now, if the character falls, the camera will only follow to the bottom of the level.
- We could repeat this process if we wanted to limit the camera on the left, top, and right as well, but for now we'll leave it as is.
- Don't forget to switch back to the **Select** mode in the main window (*Q* on the keyboard or the mouse cursor icon).

- We now need to add a **killzone** so that the player dies and the game resets when it falls off into the abyss.
- We'll use a method that can be reused for other types of killzones, such as enemies, traps, and other dangers.

2. Create a killzone:
    1. Make a new scene (click the plus tab at the top of the main window).
    2. Add an **Area2D** node. (Note that you can see your recent nodes in the **Create New Node** window and choose from here to make things faster.)
    3. Change the **Collision Mask** to *2* in the **Inspector**.
    - We're not going to add a collision shape to this node, as we want it to work for different shapes and sizes of things.
    1. Rename the root node to **Killzone**.
    2. Save the scene in our **scenes** folder.
3. Add the new **killzone** scene as an instance to the **game** scene.
    1. Switch to **game**.
    2. We can drag in the scene as we've done, or we can click the **link** icon under the **Scene** window to add an instance.
    - Now, if we add a collision shape here, it will only apply to this particular instance.
4. With the new **Killzone** instance selected in our **game** scene, add a **CollisionShape2D** as a child.
    1. In the **Inspector**, set the **Shape** to the **WorldBoundaryShape2D** that we tested earlier.
    2. Select the **Killzone** instance.
    3. Switch to the **Move** tool.
    4. Move the Killzone to the bottom of the level, somewhere under the last tiles.

- We can now make a script to kill the Player and restart the game when it hits the killzone.

5. Switch back to the main **killzone** scene (tab above the main window).
    - We need this script to apply to all killzone instances.
    1. Attach a new script to the **Killzone**.
    2. Choose *Empty* for the **Template**.
    3. Make sure the **Path** uses the **scripts** folder to save the file.
    4. Click *Create*.
    - Our code should now just have **extends Area2D** on the first line.
    1. Use the **body_entered** signal as we did for the coins:

1. Make sure the **Killzone** node is selected under **Scene** on the left.
2. On the right, choose the **Node** tab.
3. Double-click the **body_entered** signal.
4. Click *Connect*.

2. To test this, replace the **pass** line with **print("You died!")**
    - The function should now look like this:

```
func _on_body_entered(body: Node2D) -> void:
    print("You died!")
```

- We can now test the game and see that the signal and collision work.

- We can now work on restarting the game after we fall.
- Rather than immediately restarting, we can add a delay with a **Timer** node.

6. Add a **Timer** node:
    1. Press *CTRL* + *A* to add a new child node to the **Killzone**.
    2. Search for **Timer** and *Create* it.
    3. In the **Inspector** find the **Wait Time** and change it to **0.6 seconds**.
    4. Turn **One Shot** on so that the timer doesn't repeat.

- We'll now start the **Timer** in our code.

7. Add the **Timer** to the script:
    1. Click and drag timer into the Script window just below **extends Area2D**.
    2. Hold *CTRL* while releasing the mouse button.
    - This should add this line to the code:

```
@onready var timer: Timer = $Timer
```

    - The **dollar sign** next to **Timer** indicates that this is a path to the **Timer** node.
    - This also creates a variable called **timer** (note the lowercase) that we can use in our script.
    - **@onready** means this will be ready to go when the node starts.
    1. In the script, add the following code under the **print** function:

```
timer.start()
```

- We'll also need to check for when the timer ends.
- We can use a signal to do this.

8. Add a **timeout** signal:
    1. Select the **Timer** node.
    2. On the right, choose the **Node** tab.

3. Double-click **timeout()**.
4. Click *Connect*. -We should now have a function for the timeout:

```
func _on_timer_timeout() -> void:
    pass # Replace with function body.
```

1. Replace the **pass** line with:

```
get_tree().reload_current_scene
```

- This will access the Scene Tree and reload the current scene when the timer runs out.

1. Test this by pressing *F5* and fall off the edge of a platform. The game should restart after about half a second.

Worldbuilding 2.0

- We'll now add some more features to make the game more interesting.
- Before continuing, however, we need to organize our code.
- If we switch back to the **game** scene, we have a bunch of nodes in the Scene Tree, and the list is getting long.
- We can use a base **Node** to organize the parts so that we only see them when we need to.

1. Reorganize the Scene Tree:
    1. With the **Game** node (root) selected, add a child node (*CTRL + A*).
    2. Search for **Node** and *Create* it.
    3. Rename this to **Coins**.
    4. Select all of the Coin instances and drag them to the **Coins** node.
        - There are several ways to do this, but I prefer to click the first or last node I want to select, hold down **Shift** key, and click the last or first node I want to select. This should select all of the nodes in a row.
    - Once this is done, we can click on the arrow next to **Coins** to compress or expand the list.
    - We can do the same thing for any other instances that we have multiples of, such as platforms (if you added more than one).

- We can now add some more elements to our game.
- We can add more to our level, either higher or to the right as we wish.

- We will also add a background to our level.

1. Add a background:
    1. Make sure we're in the **game** scene and the **2D** window.
        - Note: this method will be different than the one Brackeys uses. I'm using the new **TileMapLayer** node, whereas his tutorial uses the deprecated **TileMap** node. I think for

this one there isn't much difference between the two.

2. Find the **TileMapLayer** and rename it to **Mid**.
    - We need to save our **TileSet** as a resource so we can reuse it:
    1. With **Mid** selected, go over to the **Inspector**.
    2. Click the down arrow next to **TileSet**.
    3. Choose *Save As...*
    4. In this window, name the file **world_tileset.tres**.
        - This saves it as a resource that can be reused.
    5. Save it to the **assets** and **sprites** folder.
        - We'll probably need to click the up arrow next to **Path** to go up a level, and then open up the **assets** and **sprites** folders.
3. Select the **Game** node and add a new **TileMapLayer** to it.
4. Rename this new Tile Map Layer to **Background**.
5. With the new **Background** layer selected, go to the **Inspector** and click the down arrow next **Tile Set**.
    1. Choose *Quick Load*.
    2. Click on the new *world_tileset.tres* resource.
6. Using the same method as we drew the level, draw some background tiles into the world.
    - Remember that the order of the nodes determines which are draw first, so move the **Background** node above the **Mid** node.
    - The background tiles in this tileset are the ones at the bottom.
    - Use any combination you wish to create your background.

Enemy

- Our enemy is going to use the Killzone for collisions, so we can use a basic Node2D as the base for this enemy.

1. Create an enemy:
    1. Add a new scene.
    2. Add a **Node2D**.
    3. Add an **AnimatedSprite2D** node as a child.
    4. Add new **Sprite Frames** in the **Inspector**.
    5. Click on *SpriteFrames*.
    6. Add the **slime_green.png** sprite sheet to the **Animation Frames:** window.
        1. Zoom in on the sprite sheet and resize the boxes (4 for horizontal and 3 for vertical).
        2. Select the middle row of four sprites; this is the **Idle** animation for the slime.
        3. Click *Add 4 Frame(s).
    7. Center the slime in the main window:
        1. Press *F* on the keyboard to center.
        2. Zoom in so you can see the slime.
        3. Move the slime up so that it's resting on the baseline.
            - Remember if you hold **Shift** and then click and drag, you can drag the sprite straight on a particular axis. This keeps the sprite from moving left or right if you want to drag it up or down, and vice versa if you want to drag it left or right.
        4. Change the FPS of the animation to 10 FPS.

5. Set the animation to **Autoplay on Load**
6. Play the animation to test it. Stop it once done.

- We can now reuse our **Killzone** scene as an instance here to give the slime a killzone collider.

2. Add the Killzone scene as an instance:
    1. Click the *Node2D* (root).
    2. Click the *link* icon.
    3. Choose *killzone.tscn*.
    4. Add a collider to the killzone instance:
        1. Right-click the Killzone instance.
        2. Choose *Add child node*.
        3. Search for a **CollisionShape2D** and *Create* it.
        4. In the **Inspector** add a **RectangleShape2D** to the **Shape** property.
        5. Resize the box to fit the slime.
            - It can be slightly inside of the slime, but not at all outside.
            - (Players will think this is unfair if they die without actually hitting the slime.)
            - If you hold **Alt** on the keyboard before clicking on the organge points and then drag them, the box will maintain its aspect ratio.
3. Rename the **Node2D** to **Slime** and save the scene in the scenes folder.

- We can now add our slime to the game scene

4. Add Slime to the Game:
    1. Switch to the **Game** scene.
    2. From the **FileSystem** drag the **slime.tscn** file into the 2D window.
    3. Place the slime in game.
        - Note: if you can't see anything because the **Background** is still select, click on the *Game* node before draggin in the slime.
5. Test the game and make sure the game restarts when the player hits the slime.
    - Remember that there is a 0.6 second delay on the restart, so the effect won't be instant.

- We can now work on moving the enemy.

6. Add a new script to the Slime scene. Make sure it saves in the **scripts** folder.
    1. Remove the **func _ready()** function but keep the **func _process()** function. We'll use this to control the movement of the slime.
        - In the video, Brackeys explains how delta and movement work and writes and deletes some code to show this. I'm going to skip that part and explain the result.
    2. After **extends Node2D**
        1. Add a **constant** called SPEED and give it a value of 60.
        2. Add a **variable** called direction and give it a value of 1 (to represent moving right)

```
const SPEED = 60

var direction = 1
```

- If we wanted the SPEED to change for some reason, we would type **var speed = 60** instead.
- Also note that it's standard practice in gdscript for contants to be in all upper case and variables to be all lower case.

3. Under the **func _process()** Replace the **pass** line with the following code:

```
position.x += direction + SPEED * delta
```

- This will do several things:
    1. We change the x position of the slime (its horizontal movement).
    2. We use **1** and **-1** to move the slime right and left.
        - Remember that right movement is positive and left is negative.
        - In most game engines, up is negative and down is positive.
    3. We multiply the SPEED constant first by direction and then by **delta**.
        - Multiplying by **delta** makes the movement uniform over time, even if the frame rate increases or decreases.
        - If the frame rate increases, it lowers the amount of movement.
        - If the frame rate decreases, it raises the amount of movement.

- If we test the game now, the slimes will move towards the right forever.
- We want them to change direction if they encounter an obstacle (such as a wall) or the edge of a platform (so they don't float in the air).
- Brackeys keeps his slime between two walls, and his instructions depend on that; however, I didn't draw my level with any parts surrounded by walls on both sides.
- I'm going to adapt Brackeys instructions so that the slime will not only react when reaching a wall, but also notice when it reaches the end of a platform; this still uses **Raycasting** as Brackeys shows, but adapts it for a different situation.

7. Make sure we're in the Slime scene and in the 2D view.
    1. Add a new child node to the slime.
    2. Search for **RayCast2D** and *Create* it.
        - Raycasts can be used to detect collisions.
        - They are represented by an arrow showing the direction of the detection.
        - We can use these to detect collisions with the walls as well as they platforms and ground.
    3. Move the origin of the Raycast to the center of the slime (as close as we can get it).
        1. Change the direction of the Raycast by clicking and dragging the orange point at the tip of the arrow; make sure it faces **right**.
        2. We can also shrink the size of the Raycast; it just has to reach just past the edge of the slime.
    4. Duplicate this RayCast2D node (*CTRL + D*) and make this one face left.
    5. Rename both of these the **RayCastRight** and **RayCastLeft**.
        1. I'm going to duplicate one more of these and rename it to **RayCastDown**.
        2. This one will just face a little bit down from the center of the slime.

        - I originally started with two, but ended up changing this to just one facing down after experimentation.
8. We can reference these nodes in our script as we did with the timer:
    1. Switch to the **Script** view.
    2. Select all of our RayCast nodes.
        - (We can hold *CTRL* and click each one to select them all, or use the **Shift** example we did earlier).
    3. Drag them into the script below **var direction = 1**.
    4. Hold *CTRL* before release the left moust button, and Godot will create variables for each of our nodes.
        - Here's what it should look like:

```
@onready var ray_cast_right: RayCast2D = $RayCastRight
@onready var ray_cast_left: RayCast2D = $RayCastLeft
@onready var ray_cast_down: RayCast2D = $RayCastDown
```

9. Add a variable under **var direction = 1** to check if the slime is still on the ground:

```
var on_ground = true
```

    - We use a boolean (true or false statement) here, and if it returns **false** we'll flip the direction of the sprite.
10. Check each frame if the slime is colliding with a wall or the ground:
    1. Under the **_process** function and before **position.x** 1, type the following code:

```
if ray_cast_down.is_colliding():
    on_ground = true
else:
    on_ground = false
if on_ground == true:
    if ray_cast_right.is_colliding():
        direction = -1
    if ray_cast_left.is_colliding():
        direction = 1
elif on_ground == false:
    direction = -(direction)
```

    - This code does several things:
        1. If RayCastDown is colliding with the ground, it sets the variable **on_ground** to **true**; otherwise, it sets it to **false**.
        2. If **on_ground** is **true**, then check to see if RayCastRight or RayCastLeft are colliding with anything, and if so, change the direction.
        3. If **on_ground** is **false**, then flip the direction.

- Since we can't know if the direction is -1 or 1, we just multiply **direction** by -1 to flip it.
- If direction == -1, this would make it 1 and vice versa.

11. Test the code (*F5*). If the slimes move too fast, lower the SPEED constant. I changed my to **40** after testing.

- We can now work on flipping the slime sprite so that it faces the correct direction.

11. Drag the **AnimatedSprite2D** node into the script and hold *CTRL* before releasing to create a new **@onready var animated_sprite_2D**.
    1. Place it just under the onther **@onready** variables.
    2. We can delete the **_2D** from the name to make it shorter.
        - Here's what our code looks like:

```
@onready var animated_sprite: AnimatedSprite2D = $AnimatedSprite2D
```

12. Add the following code to flip the sprite:
    1. Under **if ray_cast_right.is_colliding():** add:

```
animated_sprite.flip_h = true
```

        - This flips the sprite when it begins moving left.
    2. Under **if ray_cast_left.is_colliding():** add:

```
animated_sprite.flip_h = false
```

        - This turns off the flip_h when the sprite begins moving right.
    3. Under **elif on_ground == false:** and **direction = -(direction)** add:

```
if animated_sprite.flip_h == true:
    animated_sprite.flip_h = false
elif animated_sprite.flip_h == false:
    animated_sprite.flip_h = true
```

        - This is necessary because we're also checking if the slime is colliding with the ground below it. This checks which way the slime is facing and flips it the other direction when the RayCastDown is not colliding with the ground.
        - Note that **elif** is short for **else if**. We use this when we want to set another condition for an if statement.

Dying 2.0

- Now we want to make dying to the slimes more interesting.

1. Add a slow-motion effect when dying:
    1. Switch to the **killzone** scene.
    2. Switch to the **killzone.gd** script.
    3. In the **func _on_body_entered(body):** function and under the **print("You died!")** add:

```
Engine.time_scale = 0.5
```

        - This slows the game down by half when the player enters the killzone.
    4. In the **func _on_time_timeout():** function, we need to reset the timescale back to one:

```
Engine.time_scale = 1.0
```

        - This also goes above **get_tree().reload_current_scene()**.

- Next, we can add a death effect, such as having the character fall off the screen:
    - Notice in the **func _on_body_entered** there is a reference to **(body: Node2D)**.
    - This is a reference to the Player (or whatever body entered the killzone, which is just the Player at this point).
    - We can use this to remove the collider from the Player, and it will fall off the screen when it dies.

2. In **func _on_body_entered** and after **Engine.time_scale = 0.5** add:

```
body.get_node("CollisionShape2D").queue_free()
```

    - This looks for the **CollisionShape2D** node on the body and uses **queue_free()** to remove it from the Scene Tree.
    - This means the Player will no longer be colliding with the ground, and it will fall off the screen.

Player 2.0

- Here we'll add animations to the Player as well as customize the keybindings used to move.

1. Switch to the **player** scene and the **player.gd** script.
    - Note that the player script uses the **_physics_process()** instead of the **_process()** function.
    - The _physics_process() runs at a fixed rate (always 60 times per second) in order to process physics interactions (such as the CharacterBody2D), while the _process function is variable.
    - Also note that if you're following the tutorial video, Godot has changed some things since it was made. When these changes matter, I'll point them out as we go.

- One in particular that Brackeys mentions is the **gravity** variable. The current version of Godot has a built-in function called **get_gravity()**, so we don't need to declare that variable separately.

2. Godot Action System
    - We need to bind our actions to specific inputs. While Godot has some defaults that do work, we often need to refer to specific actions that we've defined to make our game more interesting.
    1. Go to **Project** at the top left of the screen.
    2. Open **Project Settings...**
    3. Open the **Input Map** tab at the top of the **Project Settings** window.
    4. Click in the **Add New Action** box.
        1. Name the action **jump**.
        2. Click *Add*.
        3. Add two more actions: **move_left** and **move_right**
    5. Use the **plus sign** to the right of the action to bind a key to that action.
        1. Click the plus sign for **jump**.
        2. This window listens for our input, so press the **Space Bar**. This binds the space bar to our jump action.
        3. Click *OK*.
    6. We can bind multiple keys to the same action to give players a choice. For example, we can bind both the left arrow and the a key to move left:
        1. Click the plus sign for **move_left**.
        2. Press the **left arrow** key on the keyboard.
        3. Click *OK*.
        4. Repeat the same steps, but press the **a** key instead.
        5. Repeat the whole process for **move_right**, using the **right arrow** and **d** as the key binds.
        6. Click *Close* when done.
3. We can now replace Godot's default actions with our own. In the **player.gd** script, make the following changes:
    1. Change **ui_accept** to **jump**. (Keep the quotes around all of these.)
    2. Change **ui_left** to **move_left**.
    3. Change **ui_right** to **move_right**.

- Now we need to have the game change our player's animations when performing actions.

4. Flip the sprite when moving left or right:
    1. Drag the **AnimatedSprite2D** node into the script, underneath the **constants** and hold *CTRL* while letting go of the left mouse button.
    2. Remove the **_2D** from the variable name to make it shorter. (Note: don't remove the other two **2D**s from the code, just the one in the variable name.)
5. Find the **direction** variable in the **_physics_process** function. We can use this to determine left (**-1**) or right (**1**) or no movement (**0**).
6. Modify the code as follows:
    1. Remove Godot's default comments above **var direction**. (Note that they even tell us here to change the UI actions to custom ones.)
    2. Above **var direction**, type the following comment (comments are denoted in GDScript with a **#**). This will help us remember the values for direction:

```
# Get the input direction: -1, 0, 1
```

3. Make some extra space under **var direction** and add the following:

```
# Flip the Sprite
if direction > 0:
    animated_sprite.flip_h = false
elif direction < 0:
    animated_sprite.flip_h = true
```

- Note: GDScript is really picky about tabs, so you may need to retab this code if you copy and paste it. -Let's add some more animations to our player.

7. Add run and jump animations:
    1. Switch to the **player** scene if you need to.
    2. Select the **AnimatedSprite2D** node.
    3. If the **Sprite Frames** window isn't showing, click *SpriteFrames* at the bottom of the screen.
    4. Add a new animations:
        1. Click the *Add Animation* button (looks like a piece of paper with a plus sign).
        2. Click *new_animation* and rename it to **jump**.
        3. Do the same thing to add a **run** animation.
    5. Add animation frames from the **knight.png** sprite sheet like we did for the **idle** animation.
        1. Click the *Add frames from sprite sheet* button (the square with a grid).
        2. Change both the horizontal and vertical to **8**.
        3. For the run animation, select 16 frames in the middle of the sprite sheet: start at the left and click on each of the frames in the first row, and then move to the second row and start again on the left.
        4. Do the same for **jump**.
            - There isn't a specific jump animation in the sprite sheet.
            - Brackeys chooses a single frame under the Roll animation (the 3rd from the left), and we can do the same.
            - We can also choose a longer animation, such as the entire Roll animation.
            - Feel free to experiment.
8. Add code to make the animations play:
    1. For the **run** animation, add the following code after the code flipping the sprite:

```
# Play animations
if direction == 0:
    animated_sprite.play("idle")
else:
    animated_sprite.play("run")
```

    2. For the **jump** animation, we're going to check if we're on the ground, and if not, we'll play the animation.

1. Add this code just above the code we just wrote:

```
if is_on_floor():
```

    - This will only play the running animation if we're on the floor.
2. Retab the idle and run animation code so that it's nested under the **if** statement we just wrote.
3. After the code for the run animation, add this code (make sure it's in line with the **if is_on_floor()**):

```
else:
    animated_sprite.play("jump")
```

3. When we're done, this block of code should look like this:

```
# Play animations
if is_on_floor():
    if direction == 0:
        animated_sprite.play("idle")
    else:
        animated_sprite.play("run")
else:
    animated_sprite.play("jump")
```

-Remember to make sure the tabs are correct!

- Test the game; feel free to adjust or change things to see how they make a difference.
- I used the **ROLL** sprites to make the knight do a flip when jumping, and deleted frames to make it feel right to me.

Text

- We can tell our story and add other elements through text.
- Text nodes are called **Labels**.

1. Add some instructions:
    1. Add a **Label** node to the main **game** scene:
        1. Return to the **game** scene.
        2. Select the **Game** node.
        3. Add a child node.
        4. Search for **Label** and *Create* it.
    2. Move the Label:
        1. Make sure to switch to the **2D** view.

2. Zoom in to the Label and move it over to the left or anywhere the player will see it when the game starts.
3. Add text:
    1. In the **Inspector** find the **Text** property.
    2. Click in the box under **Text** and type the text: **Space to jump.**
    - Note: if our background is white, we won't be able to see the white text when we type it.

- The font may be too big for the space and/or appear blurry.
- This is because we're actually zoomed in on the game since it's a pixel-art game.
- We can fix this by adding a pixel font with sharp edges.

2. Change the font:
    1. In the Inspector, find **Control**.
    2. Expand **Theme Overrides**.
    3. Expand **Fonts**.
    4. From the **FileSystem** window on the left, expand the **fonts** folder, and drag the **PixelOperator8.ttf** font to the **Inspector**.
    5. Drop it on the font (it might say < *empty* > or have a font name).
    6. Change the **Font Sizes** to **8 px**. (Use multiples of 8 for this font to make it appear crisp.)
    7. Find **Colors** and change the font color so we can read it on the background.
        - I changed mine to black since my text is on a white background.
3. We can duplicate this Label, change the text as necessary, and move them around to provide more gameplay hints.
4. As we did for the other instances, we need to create a basic **Node** and collect all of the labels under it.
    1. With **Game** selected, add a child node.
    2. Search for **Node** and *Create* it.
    3. Select all of the **Label** nodes and drag them into the new **Node**.
    4. Rename the Node **Labels**.

Score

- We can use these text elements to display our current score.
- We need two things: a score script and a label.
- It's common pratice to keep game-wide variables in a **Game Manager** node or scene. 1. For this game, we'll just make a **GameManager** node.
    1. With the **Game** node selected, create a child node
    2. Find the **Node** (just the plain one) and *Create* it.
    3. Name it **GameManager** and drag it to just under the **Game** node.
    4. Attach a script to the **GameManager** node (make sure it's still selected).
    5. Change the **Template** do **Object: Empty**.
    6. Make sure the **Path** saves it to the **scripts** folder and make sure the name is **game_manager.gd**.
    7. Click *Create*.

2. In the **Script** window:
    1. create a variable for score and set it to zero:

```
var score = 0
```

2. Create a function to add points to our score:

```
func add_point():
    score += 1
    print(score)
```

- Now that we have a function to add points, we need to call it every time we pick up a coin.

3. Pickup Coins:
    1. Open the **coin.gd** script.
    - We need to access the **GameManager** node, but we don't want to do it like we did with the others.
    - Generally, it's bad practice to refer to nodes further up the tree, as things might break as we edit the game.
    - Because the GameManager node is unique, however, we can assign it a special property.
    2. Make the GameManager node unique:
        1. Right-click on the **GameManager** node.
        2. Choose *% Access as Unique Name*
        3. This adds a **%** icon next to the node, which tells us this is a unique name (unlike things like Label or Slime, etc.)
    3. We can now click and drag the GameManager node into our script like we did before, and hold *CTRL* before we let go of the left mouse button.
    - Note that unique names can only be accessed within the same scene, so this wouldn't work if the coin instances were in a different scene.
    4. Replace the line **print("+1 coin")** with the following code:

```
game_manager.add_point()
```

        - Test the game; we should see the points add up in the **Output** window as we pick up coins.
4. Create a Label to display the Score:
    1. Duplicate one of the previous Labels.
    2. Use the Move tool to move it to the end of your level.
    3. Change the text to something like "You collected X coins!"
    4. Change the **Horizontal Alightment** to **Center**.
    5. Change the **Autowrap Mode** to **Word**. (This way, we can resize the box.)
    6. We can also use a bolder font
        1. Find **PixelOperator8-Bold.ttf** in the **FileSystem**.
        2. Drag it over to the **Font** property under **Theme Overrides** in the **Inspector**.
    7. Rename it **ScoreLabel**.

8. Because this Label is specific to the Score and the GameManager, we'll drag it under the **GameManager** node.

5. Return to the **game_manager.gd** script and add the following:
    1. Add the ScoreLabel node as a variable:
        1. Drag it into the script window.
        2. Hold *CTRL* before releasing the left mouse button.
        3. Let it go under the **var score = 0** code.
            - It should now read:

```
@onready var score_label: Label = $ScoreLabel
```

    2. Replace the **print(score)** line with:

```
score_label.text = "You collected " + str(score) + " coins!"
```

        - This does two things:
            1. It replaces the ScoreLabel's text with new text.
            2. It converts the integer of the **score** variable to a string that can be printed.

- Now, if we play our game and collect coins, when we get to the message, it should say the number of coins we've collected, and change if we collect more.

Audio

- We can now add sounds to make the game more interesting.
- We'll use a new node called **AudioStreamPlayer2D** to play sounds and music.

1. Create an **AudioStreamPlayer2D** node:
    1. With the **Game** node selected, add a child node.
    2. Search for **AudioStreamPlayer2D** and *Create* it.
    3. Rename it to **Music**.
2. Add a music file to the **Music** node:
    1. Find the **music** folder in the **FileSystem** and expand it.
    2. Drag **time_for_adventure.mp3** into the **Stream** property in the **Inspector**.
    3. Set the **Autoplay** property to **On**.
    4. Double-click on **time_for_adventure.mp3** to open the **Audio Stream Importer**.
    5. Under **Loop:** check the box next to **Enable**.
    6. Click *Reimport*.
3. Add two audio buses to control the audio tracks:
    1. At the bottom of the screen, click *Audio*. This opens the **Audio Mixer**.
    2. Click *Add Bus* twice to add two buses.
    3. Name one **Music** and the other **SFX**. This will let us easily adjust the music and sound effects.
4. Route the Music audio stream to the Music bus:
    1. In the **Inspector** find the **Bus** property.

2. Change it from **Master** to **Music**.
3. Change the volume of the **Music** bus to something like -12db. Sounds tend to play loud, and we don't want the background music to overwhelm the player.

- Test the scene and make sure the music volume is good.
- At the moment, it will restart when we die, so we can fix this so that it continues to play as the game restarts.

5. Make the **Music** node an autoload scene:
    1. Make the **Music** node its own scene:
        1. Drag the **Music** node into the **scenes** folder in the **FileSystem**.
        2. Let it go and click *Save*.
    - An autoload scene is a global scene that persists while our game is running. This is good for anything that we need to keep track of even if the character dies.
    2. Make the **Music** scene an autoload scene:
        1. Remove the **Music** node from the game.
        2. Click *Project* at the top of the screen.
            1. Click *Project Settings...*.
            2. Click the *Globals* tab.
            3. Click the file folder icon next to **Path**.
            4. Open the **scenes** folder.
            5. Click the *music.tscn* file and *Open* it.
            6. Click the + *Add* button.
            7. Click *Close*.

- Test the game again and now the music will continue playing while the game resets if we die.

- Now we can add sound effects.

6. Add a sound effect to the coin pickup:
    1. Open the **coin** scene.
    2. Add a child node.
    3. Search **AudioStreamPlayer2D** and *Create* it.
    4. Rename it to **PickupSound**.
    5. Drag in the **coin.wav** sound into the **Stream** property in the **Inspector**.
    6. Set the **Bus** to **SFX**.

- While we could play the sound using scripting, this can create issues.
- Right now, when we pickup a coin, it is removed, so the sound wouldn't have a chance to play.
- Adding a delay could fix this, but could create weird gameplay bugs where a coin is still visible while the sound effect is playing.
- We can avoid all of this by using the **AnimationPlayer** to do all of this rather than coding it by hand.

7. Coin pickups with the **AnimationPlayer**:
    1. Setup the **AnimationPlayer** node:
        1. With the **Coin** node selected, add a child node.

    2. Search for **AnimationPlayer** and *Create* it.
    3. Click *Animation* and choose *New*.
    4. Name this **pickup**.
2. Make the coin disappear:
    1. In the **Animation** window, click the **plus sign** to add a new track.
    2. Choose *Property Track*.
    3. Choose *AnimatedSprite2D* and *OK*.
    4. Find and click on *visible* under **CanvasItem**.
    5. Right-click at **0** seconds and *Insert Key*.
    6. Click the new key frame.
    7. In the **Inspector** uncheck the **On** box next to **Value**. This will make the coin disappear.
3. Turn off the collision so the Player can't interact with the coin after picking it up:
    1. Add a new animation track by clicking the **plus sign**.
    2. Choose *Property Track*.
    3. Choose *CollisionShape2D* and *OK*.
    4. Find and click on *disabled*.
    5. Insert a new key frame at 0 seconds.
    6. Click the new key frame.
    7. In the **Inspector** check the box **On** next to **Value**. This will disable the collider.
4. Play the PickupSound:
    1. Add a new animation track by clicking the **plus sign**.
    2. Choose *Property Track*.
    3. Choose *PickupSound* and *OK*.
    4. Find and click on *playing*.
    5. Insert a new key frame at 0 seconds.
    6. Click the new key frame.
    7. In the **Inspector** check the box **On** next to **Value**. This will disable the collider.
5. Run the **queue_free()** function after the sound plays:
    1. Add a new animation track by clicking the **plus sign**.
    2. Choose *Call Method Track*. This lets us call a function.
    3. Choose *Coin* and *OK*.
    4. Move the play head to 1 second by click next to the **1** in the timeline.
    5. Insert a new key frame at 1 seconds.
    6. Search for **queue_free()** and click it.

---

8. Code the **AnimationPlayer** to run the animation and reset:
    1. In the **Script** window, open the **coin.gd** script.
    2. Drag in the **AnimationPlayer** node under **@onready var game_manager** and hold *CTRL* before releasing the left mouse button.
        - We should see this code:

```
@onready var animation_player: AnimationPlayer = $AnimationPlayer
```

    3. Replace **queue_free()** with the following code:

```
animation_player.play("pickup")
```