How To Make a Simple 3D Game in Unity

My name is Sophia, Ian, and Seth and we are the Team Lead 2's. Today we will be showing you guys how to make a simple 3D game in unity. This will be following a tutorial, and we condensed the instructions to make them a bit easier to understand and you guys can follow along with the step-by-step guide that has been provided. We'll be covering the basics, as well as some key features like a start and pause menu, nav mesh, score keeping, audio, and light, that may be beneficial to your video games that you'll be developing- as well as showing the use of super and subclasses as well as going over the differences between private and public variables.

Intro through Step 6: Sophia
Step 7 through Step 9: Ian
Step 10 through Step 12: Seth

<u>Step 1:</u> Create a new project.
- Open Unity.
- Click File -> New Project
- Select the location for your project.
- Name your project.
- Click Create.

<u>Step 2:</u> Customize the layout.
- Manipulate the 5 main windows (Scene, Game, Hierarchy, Project, and Inspector) as you would like.
- Going over these windows: the scene window is where the game-making happens. It shows what elements you have in your game and where they are relative to each other. The game window shows the view that the main camera sees when the game is playing. This is where you can test out your game by clicking the play button at the top of the screen. The hierarchy screen lists all of the elements you have added to the scene- it's set to the main camera by default. You can create new elements by clicking Create and selecting the type of object you want. (DEMO THIS). This can also be done by using the GameObject dropdown menu at the top of the screen. The project window shows the files being used for the game. You can create folders, scripts, and anything else you need here by clicking create under the project window. And the last window, the inspector window, is where you customize aspects of each element that is in the scene. Just select an object in the Hierarchy window or double-click on an object in the Scene window to show its attributes in the Inspector panel.

<u>Step 3:</u> Save the scene & set up the build
- Click File -> Save Scene. Save the scene under the folder [Project Name] - Assets.

- Assets is a pre-made folder into which you will want to store your scenes and scripts. You may want to create a folder called Scenes within Assets because the Assets folder can get messy quickly, as you may have learned during the Pong assignment.
- Save the scene as Scene or Main.
- Click File -> Build Settings.
- Add the current scene to build.
- Select desired platform.
- There are lots of options, including computers, game systems, and smartphones, but we will just be using PC/Mac/Linux Standalone for this demonstration.
- Click Player Settings at the bottom of the Build Settings window. This opens the Player Settings options in the inspector. Here, you can change the company name, the game name, default icon, etc.
- Close out of the Build Settings window. You'll come back to this when you're ready to finish your game.

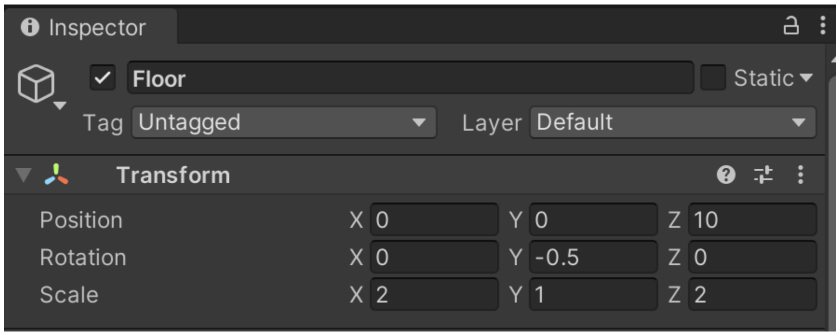
Step 4: Create the stage

- The simplest way to create a stage in Unity is by adding cubes. To add cubes, go to Game Object -> 3D Object -> Cube, or use the Create menu in the hierarchy window.
- Add a cube.
- Reset the cube's transform by right-clicking "Transform" in the Inspector panel. It's good practice to do this whenever you create a new GameObject.
- Select the cube in the Hierarchy. Rename it "Wall" by double-clicking its name in Hierarchy or using the Inspector panel.
- Scale the cube in the X direction to make it long and wall-like. We changed the X to 20.
- Right-click "Wall" in the Hierarchy panel, and duplicate it three times so you have four walls. It will look like you only have one wall because they are identical and therefore occupy the same point in space. Drag them into position and/or use the transform options for each cube to make an arrangement that looks like an arena.

	position:		
	X	Y	Z
Wall:	0	0	.5
Wall(1):	-10.5	0	10 (rotate Y: 90)
Wall(2):	0	0	19.5
Wall(3):	10.5	0	10 (rotate Y: 90)

- Create an empty GameObject, using the GameObject dropdown. (Create Empty) at the top of the screen.
- Call it "Stage".
- Reset its transform.
- Select all four "Walls" and drag them under the "Stage" GameObject.

- Add a plane GameObject by selecting Create in the Hierarchy panel and use it for the floor. Rename it “Floor”, and drag it under Stage in the Hierarchy. As a reminder, you need to hit enter after renaming or else the change might not get saved.
- Give the floor a -.5 transform in the Y-direction to ensure it lines up neatly with the four walls.
- Make the floor’s scale in the X, Y, and Z directions 1/10 of the scale you used to size the walls.



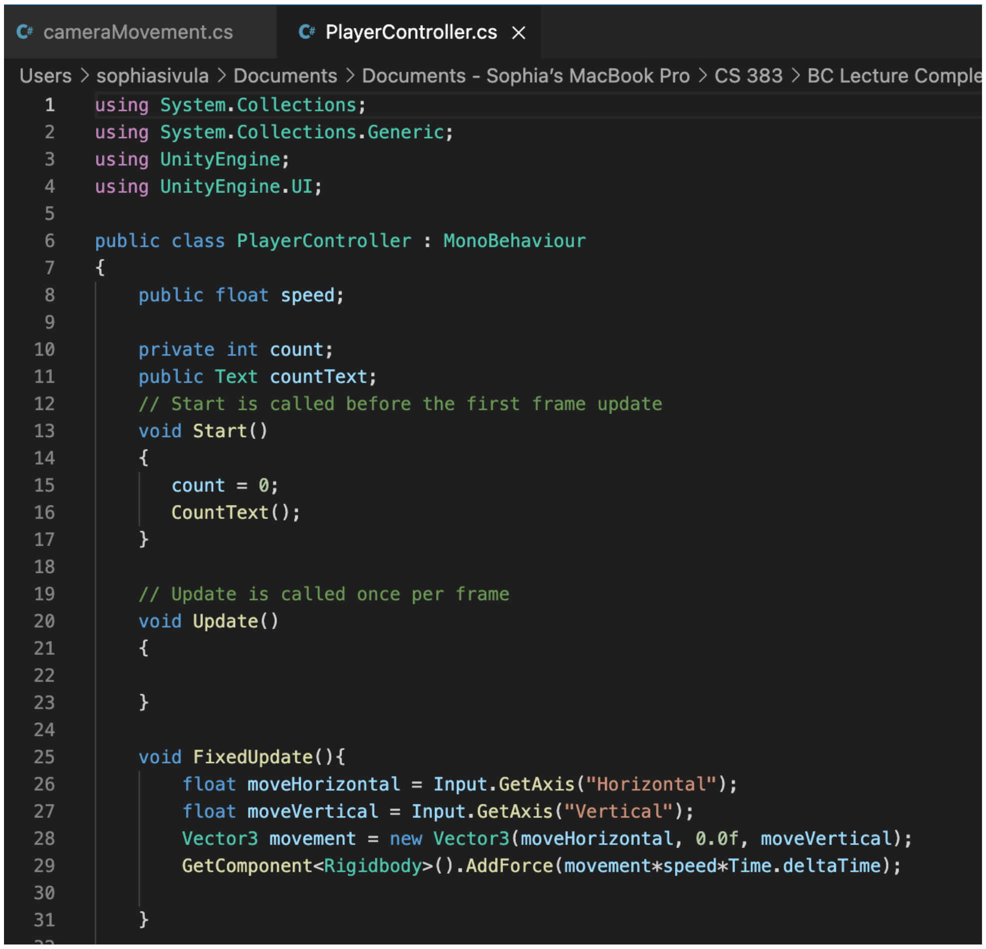
Floor
Tag Untagged
Layer Default

	X	Y	Z
Position	0	0	10
Rotation	0	-0.5	0
Scale	2	1	2

<u>Step 5:</u> Create the player
- Go to GameObjects -> 3D Object -> Sphere
- Select the sphere in the Hierarchy, and rename it “Player”. Reset its transform.
- Make the player subject to the laws of physics by clicking Add Component at the bottom of the Inspector panel with the player selected.
- Add Physics-> Rigidbody. Leave all the default settings. If you’re creating a 2D game, then youll set it to Rigidbody2D.
- You will notice that each object comes with a variety of “components” added to it that you can see in the Inspector. Each cube, sphere, etc. has a component called a “collider.” This is the physical area of the screen where that object is considered to take up space. If you turn off a collider, than the object becomes like a ghost, able to pass through other objects. You can turn components on and off by checking and unchecking the box next to the component’s name.
- We want to add a color to the player so we can see it better: In the Projects panel, right-click and go to Create-> Material and name this new material PlayerMaterial.
- Drag PlayerMaterial onto Player in the Hierarchy.
- Make a new folder “Materials” in Project Panel and move the material into it.

<u>Step 6:</u> Making the player move around
- Select the player in the Hierarchy.
- Minimize the components that you don’t want to see open in the Inspector by clicking the down arrows to the left of the name of each component.
- Click Add Component at the bottom of the Inspector window. Select New Script, name the script something like “PlayerController”.
- Click Create and Add.

- To keep organized, open the Assets folder in the Project window and create a folder called Scripts.
- Put your new script in this folder.
- Double click the script's name in the Inspector, or open it from the Project window.This will open the script in an editor, generally Visual Studios if you have that installed. Opening the script opens a programming environment called MonoDevelop.
- There will be 2 default functions in your code: void Start() and and void Update(). Start runs as soon as the object comes into the game, and update runs continuously while the object is in the game. Add a function "void FixedUpdate()" in order to handle physics-related protocols.
- Declare a "public float speed;" before start() that we can adjusts to determine the speed at which our character moves around the arena. It is important to put public before declaring the variable or else you will not be able to access the variable within Unity. This is also applicable to classes and functions: by putting public before the class or function name you will be able to access it by GameObjects within Unity.
- Under FixedUpdate, declare two more floats: moveHorizontal and moveVertical. These take on values depending on the user's keyboard commands, and FixedUpdate updates them every frame.
- Add: float moveHorizontal = Input.GetAxis("Horizontal");
- Add: float moveVertical = Input.GetAxis("Vertical");
- This will coordinate the keyboard movements to the horizontal and vertical movements.
- Create a new Vector3: Vector3 movement = new Vector3(moveHorizontal, 0.0f, moveVertical); Vector3 is a type of variable with three dimensions useful for moving objects around in 3D space. This will take on the value of the user's input for horizontal and vertical movement, and will be zero in the up/down direction because in this game, the player can only move in two dimensions.
- Input a force on the player to move it around, using rigidbody: GetComponent<Rigidbody>().AddForce(movement*speed*Time.deltaTime);
- AddForce is a protocol built in to the player's rigibody component. DeltaTime is used to make movement smoother. We will adjust the speed variable later, in Unity.
- Save the C# file and go back to Unity.
- Go to the Inspector panel for the player, and look at the movement script you just created. There will be a box for your public variable, speed. You can change the value of the public variables using the Inspector.
- Make speed equal a number between 100-1000.
- Click the play button at the top of the screen.
- Test out moving the ball using Unity's default movement keys: either ASWD or the arrow keys.
- Click the play button again to exit out of testing mode.


cameraMovement.cs
PlayerController.cs
Users > sophiasivula > Documents > Documents - Sophia's MacBook Pro > CS 383 > BC Lecture Comple

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class PlayerController : MonoBehaviour
{
    public float speed;

    private int count;
    public Text countText;
    // Start is called before the first frame update
    void Start()
    {
        count = 0;
        CountText();
    }

    // Update is called once per frame
    void Update()
    {

    }

    void FixedUpdate(){
        float moveHorizontal = Input.GetAxis("Horizontal");
        float moveVertical = Input.GetAxis("Vertical");
        Vector3 movement = new Vector3(moveHorizontal, 0.0f, moveVertical);
        GetComponent<Rigidbody>().AddForce(movement*speed*Time.deltaTime);

    }
```

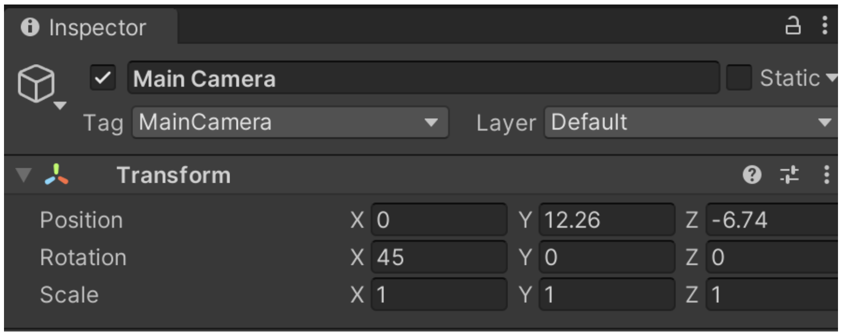
<u>Step 7:</u> Add lighting.

- Create an empty GameObject and call it "Lights".
- Create a directional light by selecting the option from "create" toolbar in the Hierarchy panel.
- Name it "Main Light".
- Make it a child object of Lights by dragging it in the Hierarchy onto the Lights game object.
- With Main Light Selected, change the light settings in the Inspector panel by changing Shadow Type to "Soft Shadows" and Resolution to "Very High Resolution".
- In the Inspector panel, change the main light's transform properties to the following:
    - Position: X: 16, Y: 60, Z: -60
    - Rotation: X: 37, Y: -34, Z: 0
    - Scale: Default
- Right click the Main Light in the Hierarchy panel to duplicate it.
- Name the duplicate "Fill Light" and child it under Lights.
- Dampen the intensity of the Fill Light by changing the color to a light blue tint and reducing the Intensity field to 0.1 in the Inspector.
- Change Shadows to "No Shadows".

- Angle the Fill Light the opposite direction of the main light. For us, this was:
    - Position: X: 56, Y: 60, Z: -10
    - Rotation: X: 37, Y: 146, Z: 0
    - Scale: Default

Step 8: Fine-tune the camera angle.

- We want the camera to be angled down over the arena, so select the Main Camera in the Hierarchy, and adjust its transform until the image in camera preview (the bottom right of the Scene panel, with the camera selected) looks good.



Main Camera

Tag MainCamera Layer Default

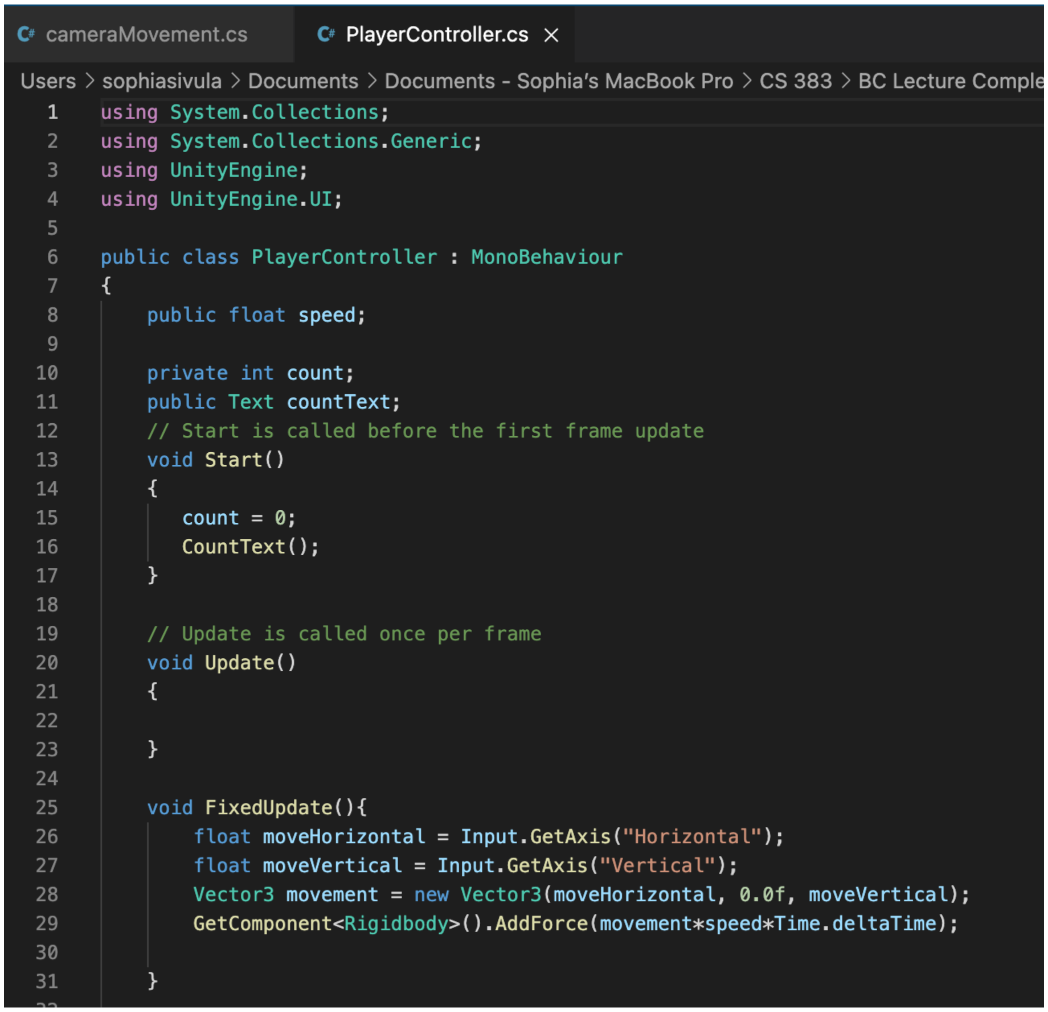
Transform

Position	X 0	Y 12.26	Z -6.74
Rotation	X 45	Y 0	Z 0
Scale	X 1	Y 1	Z 1

Step 9: Make the camera follow the player.

- We want the camera to follow the player around the screen as it moves. Create a script called "cameraMovement" by adding a new script component to the Main Camera in the Inspector panel.
- Double click the script to open it in MonoDevelop.
- This script will access another GameObject, the player, so you must declare this before the script's Start() function by writing: public GameObject player;
- Create a Vector3 called "offset" by writing: private Vector3 offset;
- Create a Vector3 called "fov" by writing: public Vector3 fov;
    - Must be public so we can edit this value in the editor.
- Under the Start() function, assign the value of offset to be: offset = transform.position;
- This is the (x,y,z) position of the camera.
- Under a function called LateUpdate(), define the camera's position as the player's position plus some offset: void LateUpdate(){ transform.position = player.transform.position+offset+fov;}
- Save the script and go back to Unity.
- We need to assign a GameObject to the "player" we defined in the cameraMovement script.
- Select the Main Camera and look at the Inspector panel.
- Under the cameraMovement script, there should be a box called "Player". It is currently assigned md to None (GameObject).
- Drag the Player from the Hierarchy into this box to assign the player game object to the cameraMovement script.
- Be sure to drag the new script into the scripts folder (in the Project panel).

- Try out the game by clicking the play button at the top of the screen. You should be able to move the player around with WASD or the arrow keys and the camera will follow your movement.
- As you are testing the game, you can change the fov value to whatever you please and see how it changes the camera view. We opted for (0,0,-5).
- Save the scene and save the project.



cameraMovement.cs | PlayerController.cs

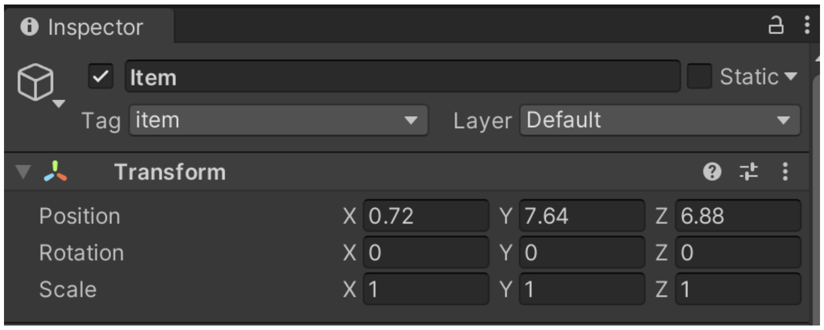
Users > sophiasivula > Documents > Documents - Sophia's MacBook Pro > CS 383 > BC Lecture Comple

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class PlayerController : MonoBehaviour
{
    public float speed;

    private int count;
    public Text countText;
    // Start is called before the first frame update
    void Start()
    {
       count = 0;
       CountText();
    }

    // Update is called once per frame
    void Update()
    {

    }

    void FixedUpdate(){
        float moveHorizontal = Input.GetAxis("Horizontal");
        float moveVertical = Input.GetAxis("Vertical");
        Vector3 movement = new Vector3(moveHorizontal, 0.0f, moveVertical);
        GetComponent<Rigidbody>().AddForce(movement*speed*Time.deltaTime);

    }
```

Step 10: Make items.

- Create a new GameObject. It can be a sphere, cube, capsule, or a cylinder. We used a cube.
- Call it “Item”.
- Tag the Item as “item” by selecting Tags, and creating a new tag called “item”, them going back to Tags for that game object and selecting the new “item” tag that you created.
- Tag all your items as items. Make sure you match the spelling and capitalization exactly.
- Place the Item into an empty GameObject called “Items”.
- Reset their transforms.

- Add a rigidbody to the Item.
- Create an ItemMaterial in the Project panel. Choose a color, and drag this onto Item in the Hierarchy.



Inspector

Item — Static

Tag: item — Layer: Default

Transform

	X	Y	Z
Position	0.72	7.64	6.88
Rotation	0	0	0
Scale	1	1	1

- Move ItemMaterial into the Materials folder.
- Duplicate the Item a bunch of times and place the copies around the arena.
  - I decided to make it into a prefab before duplication
  - Easier to change them all later on

<u>Step 11:</u> Make the player collect the items & display the score.

- Open the player movement script from the Inspector panel with the Player gameObject selected, and modify the script to allow the player to collect, and keep track of items it has collected.
- Add "using UnityEngine.UI;" to the top of the script.
  - I decided to use 'TMPro' instead, for me it was:
    using TMPro;

- Make two declarations: one is a variable that keeps track of your score, and other is a GUI text that will display your score on the scene view:

```
private int count;
Public Text countText;
```

  - For me it was:

```
private int count;
Public TMP_Text countText;
```

- Under the function void Start(), initialize count and CountText (a function we will write later):

```
count = 0;
CountText();
```

- Write a new function for what happens when the Player collides with the Items. This should be its own function:

```
void OnTriggerEnter(Collider other){
        if(other.gameObject.tag == "item"){
                other.gameObject.SetActive(false);
```

```
                        count = count + 1;
                        CountText();
                    }
                }
```

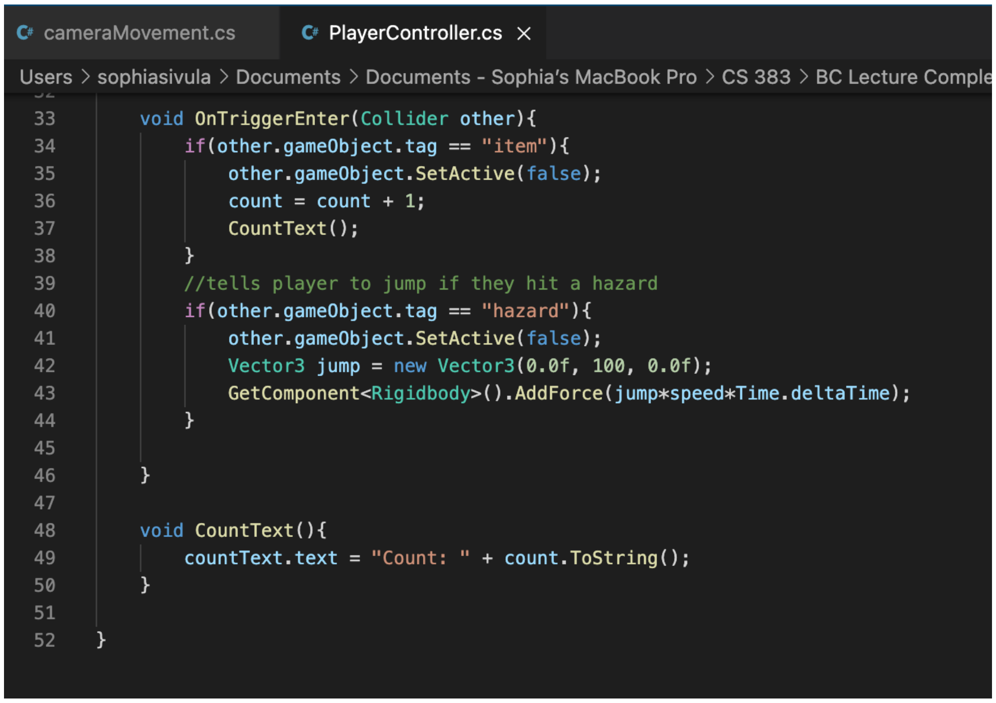
- Write the CountText function, which will update the score on the GUI display:

```
    void CountText(){
        countText.text="Count: " + count.ToString();
    }
```

- Save the code and switch back to Unity.
- Select all your items, make sure they're all tagged as items, and check the button "Is Trigger" in the Box Collider component of the Inspector.
- Check the "Is Kinematic" button under rigidbody. This prevents your items from falling through the floor by turning off gravity.
- For the countText, create a new GUI (graphic user interface) Text using the Create option under Hierarchy.
- Create a UI-> Text in the Hierarchy. Adjust so it's visible within the camera.

I did:

- Create a UI-> Text - TextMeshPro in the Hierarchy.
    - It should create Canvas and EventSystem gameobjects as well
    - DO NOT get rid of either
- Adjust so it's visible within the camera by:
    - Going to to the Canvas gameobject, then the Canvas component, and finally under Render Mode and changing it to 'Screen Space - Camera'
    - Drag and drop our camera Gameobject into the camera field
    - Decrease the 'Plane Distance' until the text is visible in the Game view, I chose 5
    - Select the 'Text' element in the hierarchy and click on the square
    - Hold 'alt' and select the top left corner to lock our text element there
    - Adjust font size as desired, I chose 100
    - Adjust the rectangle for the text in the scene view as desired
- With the player selected, drag the UI Text into the Count Text box on the Inspector.



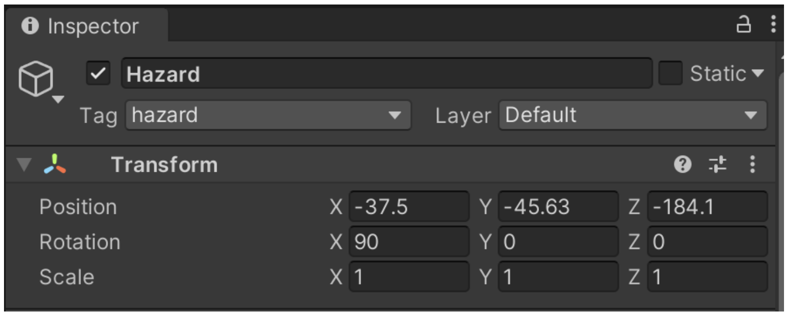
cameraMovement.cs PlayerController.cs

Users > sophiasivula > Documents > Documents - Sophia's MacBook Pro > CS 383 > BC Lecture Comple

```
    void OnTriggerEnter(Collider other){
        if(other.gameObject.tag == "item"){
            other.gameObject.SetActive(false);
            count = count + 1;
            CountText();
        }
        //tells player to jump if they hit a hazard
        if(other.gameObject.tag == "hazard"){
            other.gameObject.SetActive(false);
            Vector3 jump = new Vector3(0.0f, 100, 0.0f);
            GetComponent<Rigidbody>().AddForce(jump*speed*Time.deltaTime);
        }

    }

    void CountText(){
        countText.text = "Count: " + count.ToString();
    }

}
```

<u>Step 12:</u> Making hazards.

- These hard-to-see panels will launch the player into the air, and possibly over the edge of the arena, in which case it will be game over. Making hazards is a similar process to making items.
- Create a new empty gameObject called “Hazards”.
- Create a new Quad and call it “Hazard”.
- Tag it as hazard, and check “Is Trigger”.
- In the Project panel, create a new material and name it HazardsMaterial. Drag this onto Hazards in the Hierarchy, and move the new material into the Materials folder.
- Change the hazard’s rotation to 90 about the X axis and lower its Y height to -0.4 so it is a small square lying just over the floor of the arena.



Inspector

Hazard ✓ Static

Tag hazard Layer Default

Transform

	X	Y	Z
Position	-37.5	-45.63	-184.1
Rotation	90	0	0
Scale	1	1	1

- Edit the Player script, under the OnTriggerEnter() function, so that it accounts for the possibility that the object the player runs into is a hazard and not an item. Tell the player to jump if it hits the hazard:

    void OnTriggerEnter(Collider other){

        if(other.gameObject.tag == “item”){

```
                    other.gameObject.SetActive(false);
                    count = count + 1;
                    CountText();
            }
            //tells player to jump if they hit a hazard
            if(other.gameObject.tag == “hazard”){
                    other.gameObject.SetActive(false);
                    Vector3 jump = new Vector3(0.0f, 30 , 0.0f);
            GetComponent<Rigidbody>().AddForce(jump*speed*Time.deltaTime);
            }
    }
```

- Save the code, go back to Unity, and duplicate the hazard a few times.
- Position the hazards around the arena.
- Try out the game by going to Build & Run, save the game, and run it.
- You’ve successfully created a simple 3D game in Unity with C#!

Link to this tutorial: https://www.instructables.com/How-to-make-a-simple-game-in-Unity-3D/

How to Add Sound Effects:

Step 1: Add Sound Effects

- Create a new gameobject under ‘Audio’ called an ‘AudioSource’
- Now, make a new ‘Sounds’ folder under ‘Assets’
- Navigate into that sounds folder
- Now follow this link: https://online-voice-recorder.com/
- Make and trim down your very own “BOING” sound effect
- Open this up then drag and drop it into our Unity Sounds folder
- Now, go into our Player Controller script
- Add the following code at the top:

  ```
  public AudioSource boingClip;
  ```

  - And within the ‘OnTriggerEnter()’ method under the ‘hazard’ if statement, add:

    ```
    boingClip.Play();
    ```
- This should play the audio source’s audio clip everytime a hazard is stepped on
- Go back to Unity and drop our audio clip onto the audio source
- Go into the audio source and disable ‘Play on Awake’
- Go to our player and drop the audio source into its field
- Test it out!
- Repeat the process for a cube disappearing sound effect, but make a “BLIP” noise for this one

<u>Step 1:</u> Add Sound Effects

- Create a new gameobject under 'Audio' called an 'AudioSource'
- Now, make a new 'Sounds' folder under 'Assets'
- Navigate into that sounds folder
- Now follow this link: https://online-voice-recorder.com/
- Make and trim down your very own "BOING" sound effect
- Open this up then drag and drop it into our Unity Sounds folder
- Now, go into our Player Controller script
- Add the following code at the top:

```
public AudioSource boingClip;
```

  - And within the 'OnTriggerEnter()' method under the 'hazard' if statement, add:
    `boingClip.Play();`
- This should play the audio source's audio clip everytime a hazard is stepped on
- Go back to Unity and drop our audio clip onto the audio source
- Go into the audio source and disable 'Play on Awake'
- Go to our player and drop the audio source into its field
- Test it out!
- Repeat the process for a cube disappearing sound effect, but make a "BLIP" noise for this one
- Create a 'Sound Effects' empty gameobject in unity and put your audio sources underneath there

<u>How to Create a Superclass and Subclasses:</u>

<u>Step 1:</u> Superclass and Subclasses

- We're going to make different kinds of hazards
- We'll have to alter the player movement script and create some new scripts to go on the hazards
- Call one of the new hazard scripts "HazardAct" and attach it to the hazard gameobjects
- Create a public void method in this script called "DoSomething()"
  - Make sure to specify it as virtual so we can override it later

```
virtual public void DoSomething()
```

- Make it print out "Hazard tripped" for now
- Now, open up the player controller script, and down in the OnTriggerEnter() method, add this within the hazard if statement:

```
other.gameObject.GetComponent<HazardAct>().DoSomething();
```

- Let's walk through this slowly.
- So, we access the game object that has a trigger with the tag "hazard"

    - On that gameobject, we assume it had a component called “HazardAct”, which it should because we attached this script to all our hazards
    - Within that “HazardAct” script, we call the DoSomething() method
- Let’s test it out!
    - Make sure to look at the ‘console’ window to see the output
- Now that we’ve verified it works, let’s make something happen
- Create two new scripts: “HazardGameover” and “HazardSpawn”
    - Open both of them up
    - Change inheritance from ‘Monobehaviour’ to ‘HazardAct’
    - Now they’re child classes of our Hazard Act script
    - In both of them, override the original ‘DoSomething()’ method:
      override public void DoSomething()
- Within both overridden methods, call the ‘base’ method
- This allows us to access the original method even though we overrode it
- In the ‘HazardGameOver’ overridden method, add this code to make the editor and the actual game quit:

```
Application.Quit();
EditorApplication.isPlaying = false; //for editor only
```

- In ‘HazardSpawn’, add this code to have the tripped hazard spawn a duplicate of itself:

```
Vector3 spawn;
GameObject spawnedPlat;

spawn = transform.position - Vector3.forward;
spawnedPlat = Instantiate(gameObject, spawn, Quaternion.identity);
spawnedPlat.transform.rotation = gameObject.transform.rotation;
spawnedPlat.SetActive(true);
```

- We instantiate the gameobject this script is attached to, at its current location offset by 1 unit, then set the spawned gameobjects rotation back to the original’s and set it active
- To spawn more, just copy and paste the second section of the code and change ‘forward’ to ‘right’ and ‘left’ and add forward for all four directions
- Finally, go back to Unity and remove ‘HazardAct’ from our hazard prefab, and replace it with our new script
- Test it out!
- When satisfied:
    - duplicate our hazard prefab
    - rename it to ‘logoutHazard’
    - change its material
    - make sure the only script attached is ‘HazardGameOver’
    - Then place it into the scene

- Test it ouuuut
- Note: remove 'EditorApplication.isPlaying = false;' before building the unity project

<u>How to Create a Start Screen:</u>

<u>Step 1:</u> Create the Start Screen.

- Create a new scene "Menu" in the Project panel.
- In the Hierarchy, create a UI->Panel.
- Change the Scene to 2D mode, and press F to focus on the canvas.
- Scale the screen to Screen Size.
- Choose a color in the Inspector for your background.
- Click on Panel, and under Color drag the alpha channel to 255. (Alpha channel is the A)
- Rename Panel to "Background".
- Right-click on Canvas, click UI->Text Mesh Pro - Text.
- Change text to "PLAY"
- Change font size to 84 in Inspector.
- Align text to center, vertically and horizontally.
- Make the text bold.
- Enable Underlay in order to create a shadow. Offset X by 1, Y by -1. Increase softness to .6.
- Under Font Settings, enable a color gradient.
- Create a TextMeshPro-> Color Gradient Asset named Gold in the Project panel.
- Set top left and right to FFC757.
- Set bottom left and right to FF9C31.
- Select Text object, and drag the Gold gradient into the Gradient (Preset) under font settings in the Inspector.
- Change "Text" to "Play" in the Hierarchy.
- Right-click on Canvas, go to UI-> Button.
- Scale the button, and move it up.
- For the Image, make the color completely black.
- Disable the Image for now.
- In the Hierarchy, under Button, delete Text and drag Play into Button.
- Rename "Play" to "Text".
- Rename "Button" to "PlayButton".
- Select Text, change anchor presets by holding down ALT (or Option on Mac) and clicking the bottom right option.
- With PlayButton selected, re-enable image.
- Under Normal Color, change Alpha to 0.
- Under Highlighted Color, decrease Alpha to 60.
- Under Pressed Color, change Alpha to 115.

- Change Navigation to “None”.
- After pressing play on the game you can see now that if you hover over the button, the black box appears and becomes darker when you click on it.
- Duplicate the PlayButton.
- Change its name to “QuitButton”, and within this button change the text to say “QUIT”.
- In the Hierarchy, right-click on canvas and create a new Canvas and name it MainMenu.
- Rescale the canvas.
- Select the buttons and drag them into the MainMenu.
- To create a game title, right-click on menu, go to UI-> Text, move the title where you’d like, and change the text.

Step 2: Create the script for the play button.

- With MainMenu selected in the Hierarchy, click Add Component->New Script called MainMenu.
- Double-click to open it. Remove start and update functions.
- Add “using UnityEngine.SceneManagement;” in order to access the scene management features.
- Create a new function:

```
public void PlayGame(){
        //to load the next scene in the queue
        SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex +
1);
}
```

- Save the script, and go back into Unity.
- Go to File -> Build Settings, and Add Open Scene.
- Make sure the StartScreen scene is listed first.
- Select PlayButton, and in the Inspector scroll down to On Click().
- Hit the + button, drag the MainMenu object into it, and choose MainMenu->PlayGame().

Step 3: Create the script for the quit button.

- Open the MainMenu script from earlier, and create a new function under PlayGame():

```
public void QuitGame(){
        Debug.Log(“QUIT!”); //so we can see this works within the editor
        Application.Quit();
}
```

- With QuitButton selected, scroll down to OnClick(), hit the + button, drag MainMenu, and choose MainMenu->QuitGame().
- Now you have a functioning start screen!

Link to this tutorial: https://www.youtube.com/watch?v=zc8ac_qUXQY

How to Create a Pause Menu:

- In the Main scene, create a Canvas in the Hierarchy.
- Make sure the Render Mode is set to Screen Space- Overlay.
- Scale the screen to Screen Size.
- Select Pixel Perfect.
- Go to Scene, 2D mode, and click F to focus on the Canvas.
- Right-click on canvas, go to UI-> Panel.
- Change the color on the panel to a transparent black.
- Change the source image from Background to None.
- Rename Panel to PauseMenu. We will put all of our UI into this object.
- Right click PauseMenu, go to UI-> Button.
- Scale the button while holding down ALT (or OPTION on Mac).
- Disable Image.
- Select Text under button.
- Make the font size 65, and change the text to "RESUME".
- Make the font style bold.
- To add a hard shadow, go to add component->Shadow.
- Change Effect Distance to X: 4 and Y: -4
- Select button, switch into scene view and move the button to where you'd like.
- Rename the button "ResumeButton".
- Re-enable the image.
- Change the color to completely black.
- Change the Normal Color alpha to 0.
- Change the Highlighted Color alpha to 60.
- Change the Pressed Color alpha to 105.
- Change Navigation to "None".
- Duplicate the button and move it down. This will be the Menu Button.
- Rename it MenuButton.
- Change the font size to 35, change the text to "MENU".
- Resize the button to fit it.
- Duplicate the ResumeButton, move it down, and change the text to "QUIT".
- Rename the new button to QuitButton.

<u>Step 2:</u> Add scripts to the buttons.

- Disable the PauseMenu
- Go to Canvas, Add Component -> New Script -> PauseMenu.
- Add "using UnityEngine.SceneManagement;"
- Delete the start method, and create a variable to keep track of if the game is on play or paused:

```
public static bool GameIsPaused = false;
```

- Within Update(), add:

```
if (Input.GetKeyDown(KeyCode.Space)){
		if(GameIsPaused){
			Resume();
		}else{
		Pause();
		}
	}
```

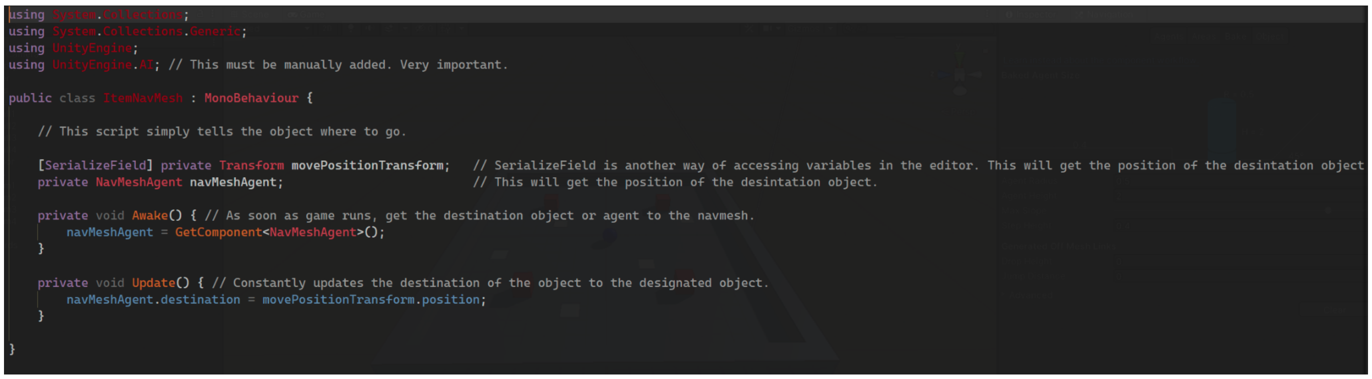
- Create a public GameObject at the top:
  ```
  Public GameObject pauseMenuUI;
  ```
- Create the Resume() and Pause() methods:
  ```
  public void Resume(){ //this will be public so we can access it from our buttons
  	pauseMenuUI.SetActive(false);
  	Time.timeScale = 1f; //speed passes at a normal rate
  	GameIsPaused = false;
  }
  void Pause(){
  	pauseMenuUI.SetActive(true);
  	Time.timeScale = 0f; // freezes the game
  	GameIsPaused = true;
  }
  ```
- Save this script and return to Unity.
- Drag PauseMenu into Pause Menu UI.
- Notice that you can now press space to pause and resume the game!
- Go back into the script, and add methods to load the menu and to quit the game:
  ```
  public void LoadMenu(){
  	Time.timeScale = 1f; //so time runs properly on the menu
  	SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex -
  1);
  }

  public void QuitGame(){
  	Debug.Log("Quitting game…");
  	Application.Quit();
  }
  ```
- Save and go back to Unity.
- Select ResumeButton, scroll to On Click (), add an action, drag Canvas into the object, go to PauseMenu->Resume().
- Repeat this action for the MenuButton (putting LoadMenu) and for the QuitButton (putting QuitGame).
- Make sure in Build Settings that all scenes are added.
- Now you have a functioning pause menu!

Link to this tutorial: https://www.youtube.com/watch?v=JivuXdrIHK0

<u>How to Create a Navigation Mesh:</u>

Definition: an abstract data structure used in artificial intelligence applications to aid agents in pathfinding through complicated spaces.
Synopsis: At this point of the guide, we have successfully created a functional 3D game with basic functions and operations. This guide will be adding a navigation mesh to one of the square item game objects. After the navigation meshes' implementation to the game, it will enable our item to follow the player around the arena with the other items serving as obstacles.

1. In the Unity Editor, select Window > AI > Navigation. This will open a Navigation tab next to the Inspector tab.
2. Select the Navigation Tab and select the Object category.
3. Under the Hierarchy. Select all the game objects (Ctrl + Select to individually select the items) under Stage and all the items under Items but Item1.
4. After selecting the objects in the Hierarchy, tick the Navigation Static option in the Navigation pane under the Object category.
5. Select the Bake category and click the Bake button.
6. Under the Scene tab, you will now be able to view the navigation mesh that was generated automatically. That in blue is the area the object can traverse, the objects in gray are inaccessible areas or obstacles for the object.
7. Create a C# script. Name it ItemNavMesh.cs
8. Open the script in a file editor.
9. Implement this code.



```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.AI; // This must be manually added. Very important.

public class ItemNavMesh : MonoBehaviour {

    // This script simply tells the object where to go.

    [SerializeField] private Transform movePositionTransform;   // SerializeField is another way of accessing variables in the editor. This will get the position of the desintation object
    private NavMeshAgent navMeshAgent;                          // This will get the position of the desintation object.

    private void Awake() { // As soon as game runs, get the destination object or agent to the navmesh.
        navMeshAgent = GetComponent<NavMeshAgent>();
    }

    private void Update() { // Constantly updates the destination of the object to the designated object.
        navMeshAgent.destination = movePositionTransform.position;
    }

}
```

10. Add this script as a component to Item1.
11. Select the Player game object and drag it to the Move Position Transform.
    a. This will ensure that Item1 will be going to the destination of the player based on the transform values of the player.
12. Test the game and watch as the item follows the player.