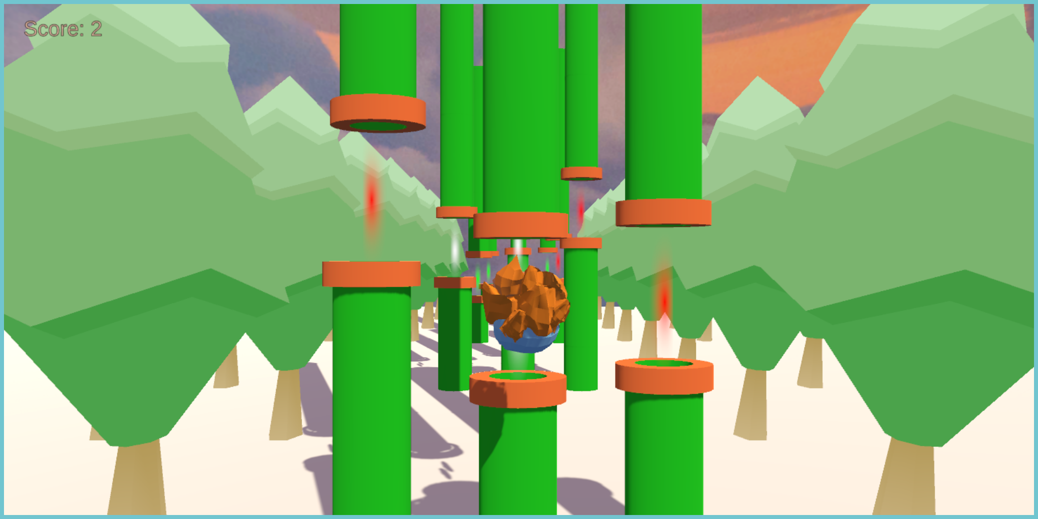
FLAPPY TMOC

A 3D Version of Flappy Bird

Thinh Pham (35%)
Robert Zhou (20%)
Charles Bacani (20%)
Trent Mellor (20%)
Haley Wheatley (5%)

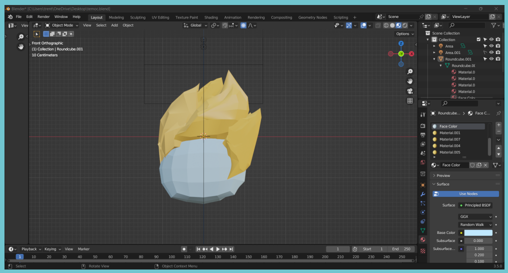
Project Description

For our project, we created an interactive game that allows a user to control a central figure and move through objects while collecting points. The base of this project is similar to the well known game, Flappy Bird. Upon creating it, we first focused on getting some simple game mechanics through by making the 2D version that is way closer to the original game. Moving on from making the simpler version of our intended game, we then incorporated methods and elaborated upon the project with multiple sets of pipes. We decided to go with a 3D approach where the player looks at the central object from behind, and navigates through pipes that spawn in front of it. There would be three different pipes for the controlled object to choose from. Furthermore, we decided to add a point system where the game would reward the player with more points if the player chose a certain pipe. A green beam would add more points, white being neutral, and red being subtractive. The dome of this game would take place at the pond of UTD, which includes a long, narrow pool with trees surrounding it. The pipes would spawn from both the floor and the top of this dome as the central object moves forward. The game would end when the central object collides with one of the pipes. For the central object, we created a sprite of TMOC using blender, which depicts the mascot of UTD. Thus, we named our game "Flappy TMOC".



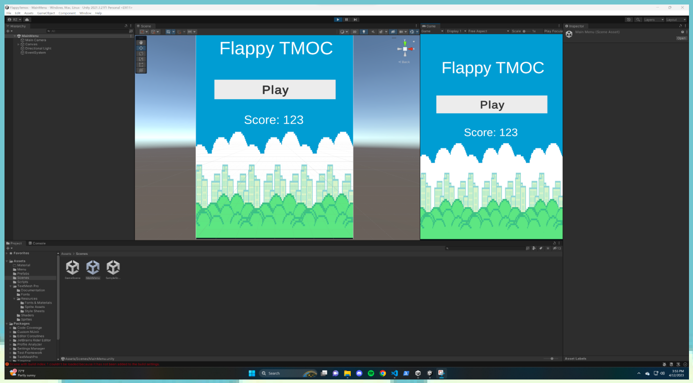
Software Tools

The main software tools we utilized to create "Flappy TMOC" was a combination of the Unity UI and C# scripts. Unity is a well known software for game creation, which is used to develop popular titles such as "Rust", "Hollow Knight", and "Monument Valley". It is known as "arguably the most robust, flexible and easy-to-use game development interface"("Choosing the Right Programming Language for Video Game Development"). Therefore, Unity provides a broad set of tools to create our game. To control the game structure and objects, C# was utilized to model the behavior of game logic and object movement. In addition to C#'s well known compatibility with Unity, it is "easy to learn, efficient and features reusable code"("Choosing the Right Programming Language for Video Game Development"). Since C# is an object oriented programming language, it makes it easier to divide the pipes, central object, and other assets to objects and control its behavior through these scripts. Furthermore, C# compiles in a just in time manner, meaning that the game would utilize significantly less memory and other resources to run compared to other languages. The last software tool that we used was Blender, which is an open source 3D computer graphics software commonly used to create animations, motion graphics, and other 3D applications. For our game, we specifically used Blender to create 3D-printed models such as TMOC. After creating it in blender, we were able to import it into Unity as sprites and other assets.



Design and Implementation

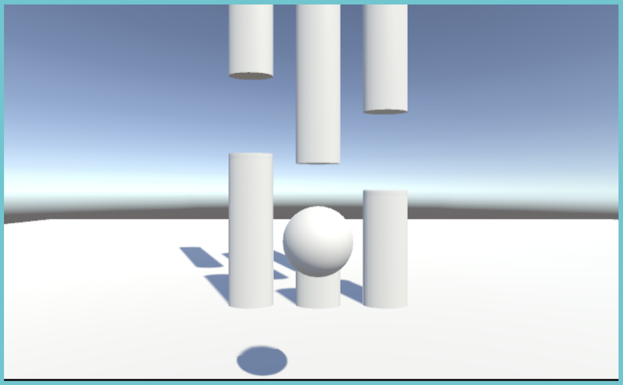
The first part of the implementation was to create the main menu of the game. Here, we would depict the high score that the current user has, which is defaulted to zero at the beginning. The main menu has a play button, taking the user to the next scene of the game which is the main scene. In Unity, the main menu scene includes a camera pointing forward to a solid image. There is no lighting option for the menu. The play option would be a button on the image which we positioned in the middle. The title and score would be texts, which are positioned both above and below the play button. In order for the play button to be more interactive, we highlighted the button when the user clicks on it by increasing the transparency of the box of the button. After that we created a script called MainMenu.cs and that allows the transition from menu to game scene once the user clicks on the “Play” button.



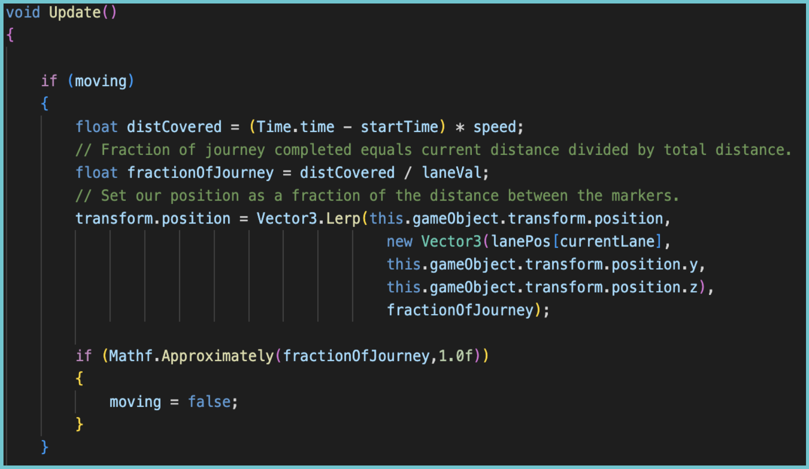
For the foundation of the game, we created a set of three pipes and a ball as a starting point as pictured below. We created an EventManager.cs to spawn a ball from the start and not be destroyed on load and destroyed once the scene is not there. EventManager is an important part to start with this project because it is responsible for the functionality and the existence of the objects in the game. The movement of the ball, which will later be the TMOC's movement, is implemented in PlayerController.cs that includes Left, Right, and Jump functions that take in the user input of left arrow key, right arrow key, and spacebar accordingly which allow the ball to move in 3 dimensions.

To give the illusion of the ball moving forward, we implemented the pipes to be moving towards the point of view of the player. To do that we created PipeMovement.cs and in the Update function, we set a fixed speed of the pipes moving towards the camera. All the trees and pipes used in the run of the game are initialized at the start of the game. In order to make them appear to go on indefinitely, they are teleported to the very beginning of the play area once they go behind the view of the camera. To the player, the obstacles appear to go on endlessly.

In order to randomize the gap of pipes at each level, we created a prefab of the pipe and implemented a script called PipeSpawner.cs that spawns 3 pipes at once at a random y value. This script will also spawn multiple pipes instantaneously at a fixed delay so that players can be able to jump through different levels and earn more points. In addition to creating the ball and pipes, we implemented the colliders that are built-in functionality in Unity for those objects. Different tags were assigned to each game object so we could easily create statements for each collision event.



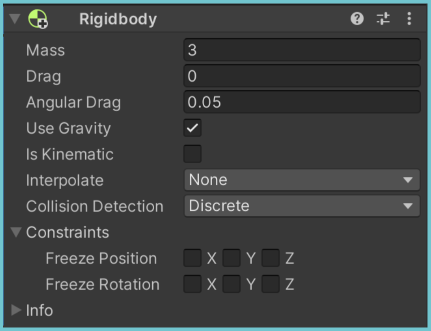
To enhance the ball movement's graphics, we added a smoother transition of the ball's movement by adding a segment of code (shown below) in the PlayerController.cs that allows the ball to transition from left to right and vice versa at a certain speed. We also added a Rigidbody that is a built-in Unity function to make the ball slightly float every time it jumps with a fixed gravity and mass values.



```
void Update()
{

    if (moving)
    {
        float distCovered = (Time.time - startTime) * speed;
        // Fraction of journey completed equals current distance divided by total distance.
        float fractionOfJourney = distCovered / laneVal;
        // Set our position as a fraction of the distance between the markers.
        transform.position = Vector3.Lerp(this.gameObject.transform.position,
                                          new Vector3(lanePos[currentLane],
                                          this.gameObject.transform.position.y,
                                          this.gameObject.transform.position.z),
                                          fractionOfJourney);

        if (Mathf.Approximately(fractionOfJourney,1.0f))
        {
            moving = false;
        }
    }
```



Rigidbody

Mass	3
Drag	0
Angular Drag	0.05
Use Gravity	✓
Is Kinematic	
Interpolate	None
Collision Detection	Discrete
Constraints	
Freeze Position	X Y Z
Freeze Rotation	X Y Z
Info	

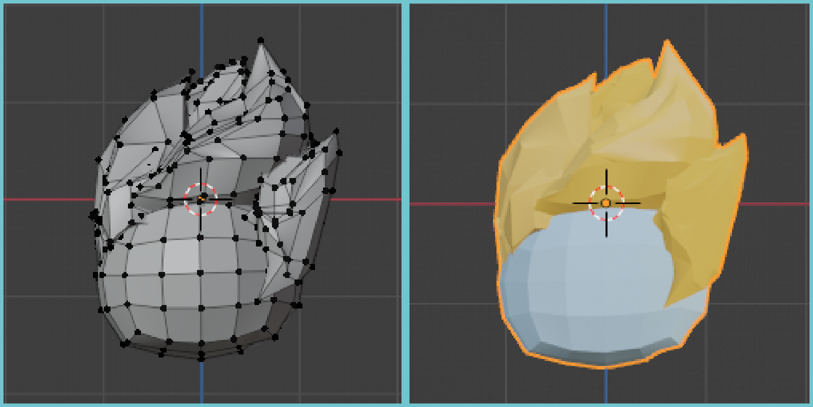
As you can see in the code segment, we calculated the distance moved by the ball with a fixed speed. Then we calculated the fraction of distance between the markers, in this case, the distance between two pipes. Finally, we set the ball's position using the fraction that we just calculated, making the ball appear to be moving between the distance of two pipes at a certain speed and that is how we were able to make the ball slide left and right smoothly. You can increase or decrease the speed of how fast it would move by changing the variable "speed". Without this implementation, the ball would just teleport left and right according to the user inputs.

For the reward and penalty system, a simple randomizer is applied to each pipe object that is created in the game scene. A normal reward has equal the chance of spawning as the penalty and extra reward chances combined together. Since the randomizer is implemented for each independent pipe game object, there are occasional instances where a set of three pipes to all have penalty gates attached to them, hurting the player's current score regardless of which pipe they go through.

Regarding the scoring system, there are different scenes at play and saving values into files was necessary since referencing game objects and values from different scenes proved to be very difficult to implement. Utilizing the already built in method PlayerPrefs(), we were able to smoothly implement score transfers whenever the player object died and displayed the highest score ever achieved locally.

Modeling Temoc in Blender:

Creating the Temoc model in Blender was a surprisingly simple task even having no prior knowledge of how to use the software. The model is composed of multiple roundcube wireframes with their vertices transformed to roughly shape Temoc's hair and head. Using a reference image, color and textures were applied to the parts of the model.



Importing the model into unity took some additional steps, rather than just putting the blender file into the Unity engine. The model had to be converted to an .FBX for it to be properly available in Unity. In addition, using the Blender file created problems with the placement of the origin of the model making it very difficult to properly orient it. Converting the model was relatively easy, and mostly only required setting the origin and scale. Once this was done, the origin was correctly placed in the center of mass and the model was accessible without requiring blender to be installed on the device.

A minor issue that we ran into when doing this conversion was with the size of the .FBX file. The file was originally about 118 kb. While this didn’t necessarily cause issues with running the game, it did mean performance would be slightly lower and it could not be uploaded to GitHub. After some research, we found that we can use the “Decimate” modifier in Blender to decrease the face count of the model enough to get it to at least push to the Github repository.

Trees and Backgrounds:

The trees used in the game were downloaded free from the Unity Asset store. They were chosen because they looked similar to the magnolia trees on either side of the reflection pools on campus. The arrangement of the trees in the prefab was also set to mirror this setting. The current state of the game is missing the actual reflection pools themselves as time did not allow for the creation of the assets. However, their movement and spawning logic would be the same as the trees and pipes.

Lessons Learned

One lesson we learned while figuring out who to optimize game states was creating different scenes instead of pausing and resuming the same scene when a different game state needed to be enforced. In our 2D version of our game, we were easily able to just work with only one scene even if we needed to change the UI whenever the game state changed. Objects could easily be destroyed out of the camera view and the resetting of positions was also implemented with little issues. Problems occurred with the 3D aspect of the game because even more objects had to be dealt with and resetting, pausing, and resuming during one scene proved to be problematic since destroying objects could not happen fast enough. Furthermore, position changes could not be consistently successful between game state transitions. We ended up creating different scenes instead to solve the issue and transitioning between scenes instead of within one scene proved to be more stable.

Another lesson we learned is that the addition, removal, or just modification of any prefabs does not always update the references to methods in the code attached to the game objects. Creating a prefab of an object but later needing to add another object to that prefab can occasionally prevent that new object from being found with certain methods. The method OnCollisionEnter() was exactly that type of method that would always read that newly added as a null reference instead of a valid game object. We needed to use OnTriggerEnter() and tested with a new prefab to fix that issue. The implementation of the random point system into the pipes would not be possible if this issue was not fixed.

Future Improvements

Some improvements upon the current working version of the game include finishing the map by including more detail and possibly higher resolution models, bug fixes, and the addition of new objectives and terrain generation. Currently, there is a bug in the spawning for the reward system which causes every pipe space to spawn a red light, giving the user no choice but to be penalized. In addition, some pipes could spawn completely closed off to further incentivize lateral movement.

Citations

“Choosing the Right Programming Language for Video Game Development.” *Beamable*, 16 Mar. 2022, beamable.com/blog/choosing-the-right-programming-language-for-video-game-development#:~:text=Today%2C%20C%23%20is%20widely%20regarded,efficient%20and%20features%20reusable%20code.