Fruit Ninja in C# and Unity

Harrison Zhang

1 Introduction

In this final project for CS 175: Computer Graphics, we replicated some key aspects of the hit mobile game Fruit Ninja, a fruit slicing video game developed by Halfbrick and released on Apr 21, 2010, using C# and Unity. We added materials and colors for each fruit variety and their insides and outsides. For instance, if we have an apple, then we added some instance of red for the outside and a yellow-ish beige color for the inside core. Perhaps most important, we implemented Fruit_Whole and Fruit_Sliced models, which represent the fruit states before and after being sliced by the blade respectively. We also introduced prefabs for each variety of fruit, including apple, kiwi, lemon, orange, and watermelon. A fruit base, Fruit_Base served as the foundation of the fruit behaviors we defined throughout the game. In this writeup, we outline steps and configurations that we took to reproduce the key aspects of Fruit Ninja and how these aspects played into the overall game design, including graphical interface and physical mechanics.

2 Main Camera

For the transform section, we positioned the main camera at coordinates $(X : 0, Y : 1, Z : -10)$ as we found that this provided the best distance and shadow rendering for the fruits against the wooden dojo-esque backdrop we chose. We did not alter rotation or scale. They are set at $(X : 0, Y : 0, Z : 0)$ and $(X : 1, Y : 1, Z : 1)$ respectively. In the Camera section, we set the Clear Flags as Solid Color brown (hexadecimal: 302118) to match the backdrop (although it does not really matter because the backdrop obscures this anyway). We used an orthographic projection of size 10 and clipping planes of Near: 0.3 and Far: 1000 to maintain aesthetics as discussed before. The viewport rectangle was left as default, that is, we center on the origin, and set a depth of -1. We also use occlusion culling for the main camera, and set the tag to Main Camera.

3 Directional Light

The directional light was the main tool controlling the shadow aesthetic and configuration, where we can see the better aesthetic effects of positioning the main camera back by a factor of -10 in the Z direction. For the directional light, we positioned it at $(X : 0, Y : 3, Z : 0)$, rotated it with coordinates $(X : 10, Y : -5, Z : 0)$, and kept the original scaling at

$(X : 1, Y : 1, Z : 1)$. We used a softer yellow-ish beige light to complement the background (hexadecimal: FFF4D6). Moreover, we set the light to mixed mode, kept the intensity and indirect multiplier at the default setting of 1, and opted for soft shadows. To configure the shadows, we used a strength of 1, a bias of 0.05, a normal bias of 0.4, and a near plane setting of 0.2.

4 Plane

For the backdrop plane, the coordinates at which the origin lies is $(X : 0, Y : 0, Z : 5)$, the rotation is $(X : -90, Y : 0, Z : 0)$, and the scale is $(X : 10, Y : 10, Z : 10)$. We opted for a plane mesh. In the configurations for a mesh renderer, we set the material to be the same as our Clear Flags (hexadecimal: 302118) in order to match the backdrop. For the lighting configurations in this plane, we turned on cast shadows and the option to receive shadows. For the light probes and reflection probes, we opted for blend probes. For motion vectors, we use per object motion and turn on dynamic occlusion. This way, HDRP calculates motion vectors for the fruit if the fruit moves and the camera does not, which is as desired in our replication. Dynamic occlusion is mainly for performance boosting. We also introduce a mesh collider, which we set to be concave. The backdrop (background) is set to the dojo-esque wood material described earlier, which we (obviously) render as opaque. We also set the metallic and smoothness factors to 0 because wood in Fruit Ninja is not metallic or smooth. Further, we set the emission tiling to $(X : 3, Y : 3)$ and default the offset to $(X : 0, Y : 0)$. For secondary maps, we defaulted the tiling to $(X : 1, Y : 1)$ and the offset to $(X : 0, Y : 0)$.

5 Spawner

5.1 Configuration

For the spawner, we set the origin of the spawn box (as we will discuss later) to $(X : 0, Y : -15, Z : 0)$. We set the rotation and scale to default settings at $(X : 0, Y : 0, Z : 0)$ and $(X : 1, Y : 1, Z : 1)$ respectively. We did so because we want the fruit to spawn randomly within a given box of fixed size (note that we also care about the depth, or Z direction at which the fruit spawns), but we do not want the action of spawning to be visible to the player. Hence, we augment the Y coordinate for the origin of the spawn box to $Y : -15$, which is below the cutoff for the viewport. As discussed earlier, we also introduce a box collider. The size of the fruit spawn box is set to $(X : 10, Y : 1, Z : 2)$ because it fits the relative size of the viewport. That is, the fruit would spawn in the center of the viewport and not at the edges. We set the minimum spawn delay to 0.25 second and the maximum spawn delay to 1 second to ensure a continuous, but still relatively random stream of fruit spawned and pushed upwards from the spawn box. For the box collider (i.e. the spawn box itself), we turn off Is Trigger and default the material to None. We set the minimum angle to -15 and the maximum angle to 15, and set the minimum and maximum forces to 15 and 18 respectively. The force metrics control how much the fruit is accelerated upwards when it spawns in the box collider below the viewport. We also set max lifetime to be 5 seconds

to be safe. That is, it should take at most 5 seconds for the fruit to spawn, be propelled upwards, and fall back down (regardless of whether it is sliced), after which we can delete the object. This frees memory and prevents lag as the player progresses throughout the game.

5.2 Script

For the fruit spawner, we essentially wrote two complementary C# scripts, Spawner.cs and Fruit.cs, to configure where and how the fruit would spawn within the box collider discussed before. First, we set the fruit prefabs as (E0: Apple, E1: Kiwi, E2: Lemon, E3: Orange, E4: Watermelon) where Ei, $i \in \{0, 1, \ldots, 4\}$ corresponds to the element indexing in a fruit array we utilize in the scripts. For the Spawner.cs script, we intialize a Collider object for the spawn area box collider and a GameObjects array for the fruit prefabs. We also initialize and fine tuned the variables that control the fruit physics. In particular, we set

minimum spawn delay: 0.25
maximum spawn delay: 1
minimum angle: -15
maximum angle: 15
minimum force: 18
maximum force: 22
maximum lifetime: 5

where units are in seconds and degrees as categorically appropriate. There are four main functions that control the fruit spawner. First, the Awake() function initializes our collider for the spawn area. Next, the OnEnable() function starts the Spawn() coroutine. Similarly, the OnDisable() function stops all (Spawn() function) coroutines. In the Spawn() function, we intiallly wait for two seconds at the start of the game before spawning any fruit. Then, we continuously do the following while the game is ongoing. We select a random fruit (prefab) from the array we initialized, and choose a random spawn positon within the spawn area box collider. Then, we choose a random rotation (with Euler angles) in the Z direction. That is, we set the rotation to $(X : 0, Y : 0, Z : R)$, where $R \sim \text{Unif}(\text{minimum angle}, \text{maximum angle})$. Next, we destroy any fruit that has lived past its maximum lifetime of 5 seconds. Finally, we generate a uniformly random force on the interval [minimum force, maximum force] and apply it to the fruit we just spawned. We wait for a uniformly random time on the interval [minimum spawn delay, maximum spawn delay] before continuing in the loop and repeating this process.

In the script Fruit.cs we introduce two GameObjects for the abstract whole and sliced versions of the fruits, which correspond to the fruit before and after being sliced respectively. We also have a Rigidbody for the abstract fruit itself, and a Collider object for its collider. We also have a ParticleSystem object for fresh juice squirts, which occurs when the player slices the fruit with their blade. We have three main functions, which are Awake(), Slice(Vector3 direction, Vector3 position, float force), and OnTriggerEnter(Collider other). The first function initializes the objects from NULL. The second controls the fruit behavior

while and after it is being sliced. That is, we first disable the "whole" (non-sliced) fruit object and set its status to inactive. Next, we enable the sliced fruit object and play the juice squirt effect. After this step and as the juice is squirting from the fruit, we deflect the fruit based on which direction the blade hit the fruit initially. We calculate the angle of deflection for each slice of the two half slices, and apply force by applying an impulse for each slice. The end result shown is a fruit that is sliced, then squirts juice, and finally deflects each of its two slices in a sensical direction, with sensical velocity and acceleration. The last function essentially detects if and when the fruit is being sliced (i.e. hit by the blade) by comparing tags, since the tag for the blade is set to the Player tag (recall that we intiially set every other part to either Main Camera, or not tagged at all if it was not discussed). After getting information from the blade, this function calls the Slice function discussed above in order to initiate the behaviors we defined for fruit slicing.

5.3 Particles

Briefly, the particle system defines the juice squirt effect when the blade comes into contact with the fruit object and effectively slices it in half. We configured spherical particles because they best resemble juice droplets. Also, consider that the squirting effect plays immediately after the blade hits the fruit and the fruit splits in half, and that we immediately clear the particles. During this process, we also composed droplets that would decrease in diameter linearly over time, which is a sensical method of making the juice droplets disappear so that they do not clutter the viewport.

6 Blade

6.1 Configuration

The origin coordinates of the blade is defaulted to $(X : 0, Y : 0, Z : 0)$, its rotation is defaulted to $(X : 0, Y : 0, Z : 0)$, and the scale is defaulted to $(X : 1, Y : 1, Z : 1)$. Afterwards, we introduced a sphere collider (since our blade is really just another ball that we drag around, and the "slicing" is defined as the blade ball hitting the fruit ball). We set Is Trigger for this collider, and default the material to None. The center of the blade ball is $(X : 0, Y : 0, Z : 0)$, and the radius is set to 1. The majority of the blade behavior is defined in the script we wrote, which is outlined below.

6.2 Script

There are four main functions we utilize to define the blade behavior, which are Update(), StartSlice(), StopSlice(), and ContinueSlice() respectively. In Update(), we call the latter three functions. Namely, if our left click button is down, then we start slicing (so we call the StartSlice() function), if the left click button is up, then we stop slicing (so we call the

StopSlice() function), but if we are slicing, then we continue slicing (given that the left click button is down), so we call the ContinueSlice() function. In StartSlice(), we get the position of the mouse in world space coordinates and set the Z-coordinate to 0. Then, we set our slicing status to "true", enable our collider, enable our blade trail (which is the fancy white slash when the player drags the blade around in the game, whether they are slicing a fruit or not). Immediately, after enabling our trail, we clear it, which means that the trail disappears in a reasonable amount of time starting from when the player began dragging to the current state of the game. The StopSlicing() function simply sets our slicing status to "false", disables our slicing collider, and disables the blade trail.

The ContinueSlice() function is similar to the StartSlice() function. In ContinueSlice(), we find the new position of the camera in world space coordinates as we did earlier in StartSlice(), and set its Z-coordinate to 0. We set the direction to this new position to the originally recorded position. Then, we calculate the velocity at which the blade hits the fruit, which is simply the magnitude of the direction vector divided by the change in time δt. Also, if the velocity is greater than the minimum threshold velocity for a slice, we enable the blade collider so the player is able to slice the fruit of interest. Otherwise, we do not enable it, that is, we disable it. Finally, we set the recorded position to the new position of the mouse cursor in world space coordinates we calculated earlier. The slice force is set to 5 as a default constant, and the minimum slice velocity is set to 0.01, because they yielded the most sensical graphical and physical results (that is, the fruit objects obey the laws of physics).

7 Event System

The event system has default position, rotation, and scale, which are $(X : 0, Y : 0, Z : 0)$, $(X : 0, Y : 0, Z : 0)$ for each property respectively. It is controlled by a default script that we do not alter in any way. As for the properties we do modify, we set the drag threshold to 10, set input actions per second to 10, and set repeat delay to 0.5.

8 Game Manager

The game manager is a script we wrote to define how the game operates. This controls the fruit and blade spawning, what happens to the sliced fruit after they have been sliced or any whole fruit that have not yet been sliced, etc. In GameManager.cs, we defined an Image object to fade in the game at the start, a Blade object that corresponds to our blade, and a Spawner object to spawn (whole) fruits. There are four main functions we will discuss, which are Awake(), Start(), NewGame(), and ClearScene(). The Awake() function initializes the blade object to the current one in the game, and does the same for the spawner we have in the game. The Start() function calls NewGame(). The NewGame() function sets the time scale to 1, calls ClearScene(), enables the blade, and enables the spawner. Finally, the ClearScene() function initializes an array of fruits with elements that are the fruits in the current game. Then, we iterate through this array and destroy every fruit in order to free memory as discussed above.

9 Conclusion

In our replication of Fruit Ninja in C# and Unity, we applied several concepts we learned in class, such as rigid body transforms, world space coordinates, euler angles, and directional lighting. While we worked with these aspects at a high-level in Unity, since the software intentionally makes game creation more beginner-friendly, applying these concepts at a low-level in C^{++} in class served as a nice foundation that helped us understand the consequences of the configurations we selected in Unity. We also learned basic C# as well as Unity from scratch in order to complete this project, which took several days, but was invaluable in helping us see how the low-level notions we discussed and implemented in class and on the problem sets applied to real-world game design in the form of a fruitful replication of Fruit Ninja.

10 Appendix

This project is available at https://github.com/itsharrisonzhang/fruit-ninja. View some screenshots of the game below.