81368384 and 31317932

CS175 Final Project: Blob's Great Adventure

Description

Our game is an extension of a 3D platformer. You control a character (Blob Jr) and explore different biomes! There are three biomes: beach, snow, and forest, each with their own flavor. This was our first time using Unity and we learned a great deal about using Unity as well as implementing more complex, interactive graphics.

Basic Game Mechanics

You can move Blob Jr using WASD or the arrow keys and jump using the spacebar. Move the mouse to change the direction Blob Jr is facing, Blob Jr will move forward in the direction it faces!

Initially, we tried having Blob Jr move according to the global coordinate system (i.e. the up arrow key always moves you north). However, we found that to be counterintuitive and that, without being able to rotate the camera, it was to difficult to see around obstacles. We chose to implement ego motion since we are viewing the world from the perspective of a character.

We also tried several different techniques to control the characters movement and collisions. Although we initially used RBTs which allow you to easily apply physics within unity, we eventually divided to use the character controller. Character controllers don't experience forces such as gravity, instead you simulate any forces within the character scripts. We initially made this decision believing we would make our character fly as part of a gameplay element, but eventually ended up removing this feature due to map constraints and focusing on the graphical elements of the game. Using a character controller also allowed us to fine tune the character's motion. Now, when pressing multiple arrow keys at once (e.g pressing left and up to travel diagonally) the character will travel at the same speed as when one arrow key is pressed.

Although we initially began with a static camera, we realized that it was impractical for navigation. The camera is currently set to follow Blob Jr around from a set distance, rotating to match Blob Jr as it turns. After making this change, Blob Jr's movements appeared jagged or laggy. To resolve this, we set the camera controller to update after the player controller. Since the camera's position and orientation is based on Blob Jr's, the camera controller must update after the player controller to use the Blob Jr's new position.

Since most of our terrains were created using planes, we also encountered an issue with the camera flipping when looking down from too high an angle and clipping below the world when viewed from too low an angle. To fix this, we limited the cameras vertical range of motion from 45 to 65 degrees and -45 to -60 degrees.

Another camera viewing issue we encountered was the issue of the camera clipping through walls. Initially, since we positioned the camera a fixed distance behind Blob Jr, this led to an issue where if the character was positioned close to a wall and rotated the camera, the player would see the other side of the wall, blocking their view of Blob Jr. To resolve this, we used raycasting, or specifically, Linecast in Unity, to draw a straight line from the center of the player's position to the position of the camera. We also gave the player a Unity attribute tag called "Player." Whenever the raycast function is broken by an object (not tagged "Player") between the player and the camera, we positioned the camera so that the camera was placed at the position of where the ray was broken , plus a slight offset.

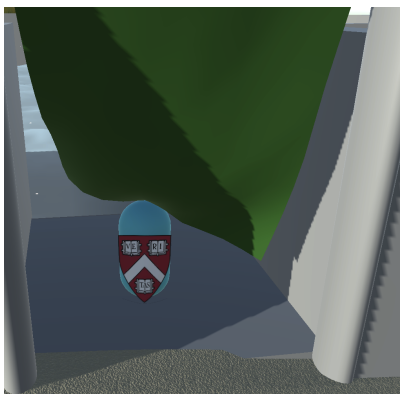
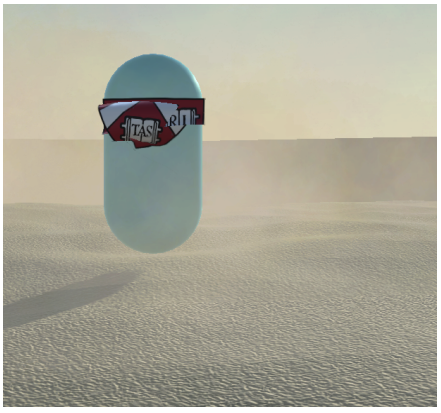
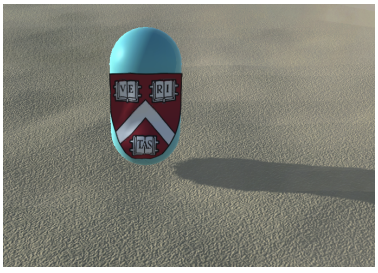
This had the effect of preventing camera clipping by making the camera appear close to the player (giving a zoomed-in effect), instead of putting the camera behind a wall.

Here we learned a lot about camera control. Instead of having a camera you can control like in the psets, we allowed the player to move the character on screen and pegged the camera to the Blob Jr's motion instead. Fine tuning the camera control (e.g no clipping through walls, no flipping, minimizing lag) was the most challenging aspect of creating the basic game mechanics.

Blob Jr

Blob Jr. has a cape attached to him that simulates cloth physics and moves along with his motion. This cape was created through attaching a transparent texture to a plane, and adding a cloth component to the plane. Then, we added anchor points for the cape to Blob Jr. to give the cape an axis to flutter around, while adding Blob Jr. as a capsule collider to the cloth component to prevent the cape from clipping through the character.

For the cloth cape, we experimented with different values on the Cloth component to change the feel of the cape, such as the bending stiffness, damping, world acceleration scale, and friction. The two most important values that we changed were increasing the damping, to decrease the fluttering of the cape over time, as well as decreasing the world acceleration scale, which prevented the cape from swinging wildly from side to side or flying above Blob Jr.'s head when the character jumps.

Cloth Door

To transition between each zone, we added door with fabric coverings over them that the player can push through to enter the new zone. These doors were constructed by creating a doorframe and anchoring the cloth object to the top of the doorframe. However, deciding the method of creating a two-sided cloth door with textures on both sides was slightly trickier than anticipated.

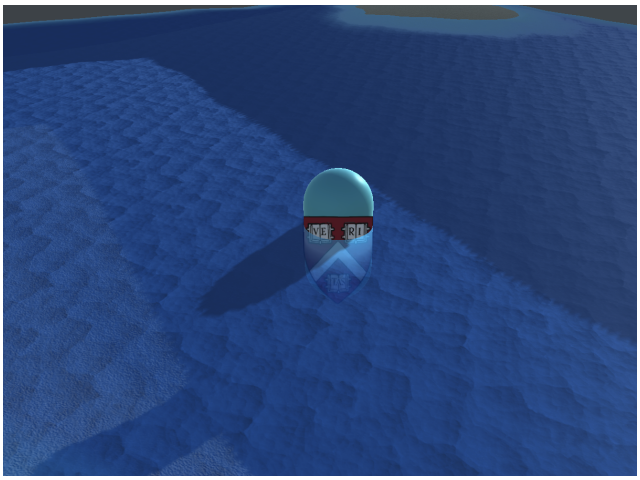
Initially, like the cape, we implemented the cloth for the door as a plane in Unity. The downside for this approach, though, is that Unity shaders automatically have backface culling enabled, which does not display textures on the back side of the plane. For a door, where players are allowed to enter from both sides, this is obviously problematic - when going from the beach to the snow, there is a cloth, but when going from the snow to the beach zone, there is no cloth to be seen!

The first way we attempted to resolve this was creating a 3D cube object, then stretching it out so it had very small thickness (around .01) to resemble a plane. Since cubes are visible from all sides, we anticipated that we could simply attach a cloth component and then have a double-sided door. However, we encountered an issue with the cloth constraints - the cube "cloth" could only be constrained at its vertices, which created a very stiff cloth that reacted similarly to a plank of wood when interacted with. Despite modifying the stiffness variables, this still persisted.

Our final solution involved creating two separate planes directly on top of each other with their front faces facing outwards. This enabled us to have accurate cloth physics and to display textures for entering and leaving the same zone.

Beach Biome

Our starting area is the beach biome. In the beach biome we have a hilly beach and a shallow ocean. Although you start in a dust storm, after walking forward you will arrive at the ocean! The ocean is dynamic — lighter where the water is shallower and darker where the water is deeper. The shoreline ebbs and flows and there are waves on the surface of the ocean that move slowly as well!

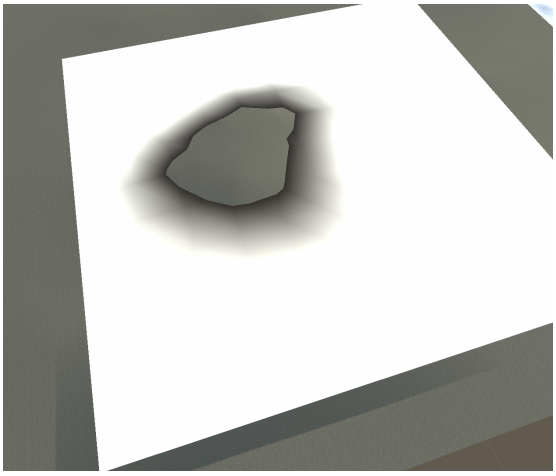


Blob Jr standing in the ocean

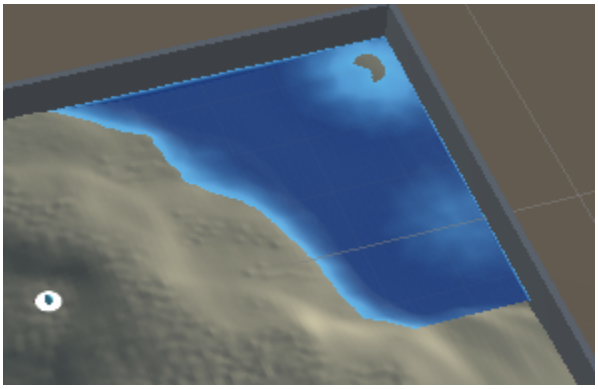
The beach was created using Unity's terrain tools. We originally created a hilly environment to explore how shadows would change based on the position of our directional light (we experimented with mornings, noons, and sunsets). Ultimately, we decided to keep the beach well lit to maximize reflections coming from the water surface.

Even though we only implemented an ocean surface, it was surprisingly challenging. Implementing a realistic ocean had many components and we struggled to get balance the quality of the ocean with the strength of the normal map and reflectiveness of the surface.

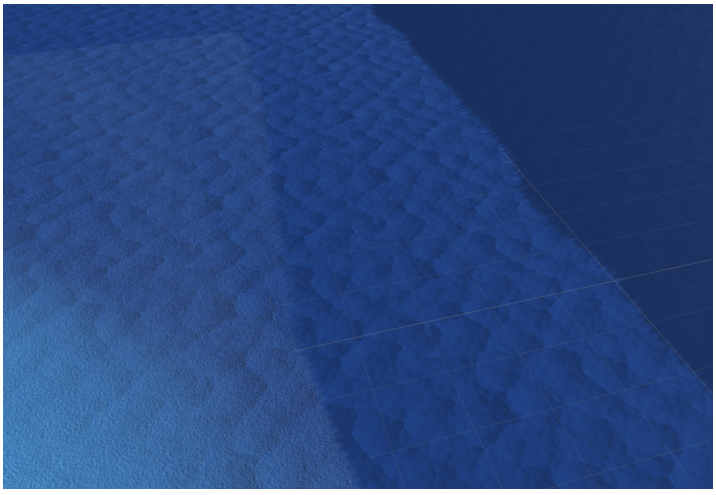
To create the ocean, we began by using Blender to subdivide a plane and create a mesh that could be applied to our water plane. Next, we wanted to modify our water's base color based on its depth. We created a shader graph and set the surface type to transparent. We then subtracted each fragment's raw screen position (not divided by clip coordinate) from the camera's far plane to calculate its depth. This created a gradient where the darkest point is the shallowest.



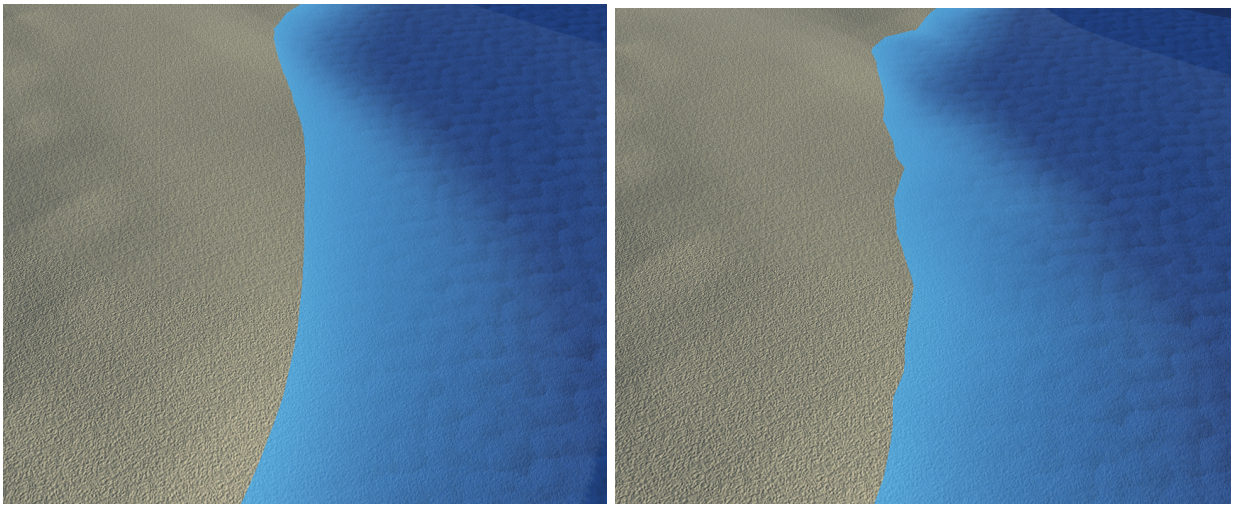
We then transformed this gradient into a gradient of blues by linearly interpolating between the RGBA values of a light and dark blue.



Next, we wanted to add waves to the surface of our water. Since the ocean is merely a plane, we simulated the appearance of waves using bump maps, similar to the strategy used for the stone floor in our pset. We used two slightly different bump maps to create the appearance of natural waves and, by changing the sample offset of both relative to the game time, we were able to create the appearance of moving waves.



Finally, to create a more realistic ocean edge, we offset the y-value of our vertices based on a randomly generated noise values. Our current mesh didn't have a high enough polygon count, as such the edges of the water now looked a little jagged (see below for a before and after). We couldn't upgrade to a mesh with more triangles as that cause performance issues on our computers.



However, we decided to keep this new jagged edge because the smooth water edge looked less realistic when combined with a moving shoreline. By changing the offset of our noise sample based on time, we were able to create the appearance of a shoreline that ebbs and flows.

We also added a dust storm that swirls around the beach with a particle effect. We decided to prewarm the particle system, which means all the particles generate before the game starts to simulate the effect of the player being immediately dropped in a sandy environment.

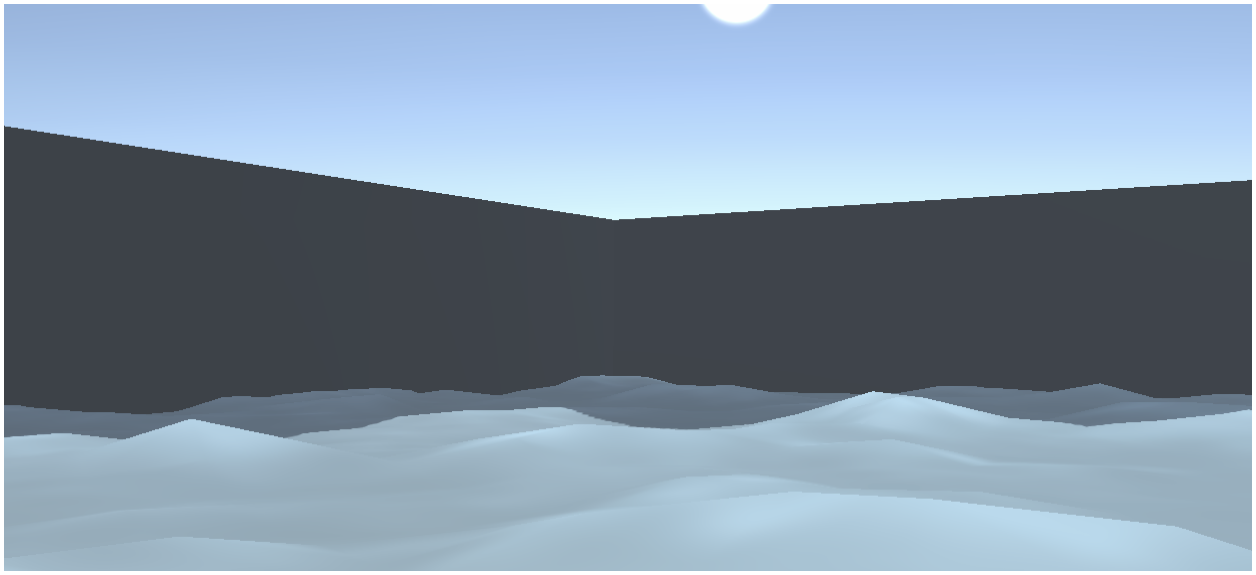
As another feature in the Unity Terrain tool, we simulated wind erosion on the beach terrain to flatten the terrain and give some geological accuracy to having a dust storm.

We learned many things from implementing the beach biome. Firstly, to implement the water we had to upgrade our render pipeline to a universal render pipeline. This required us to also learn how to upgrade all of our existing assets. We learned how to use blender and import assets into unity. We learned how to use a scene graph for the first time and the many different attributes and properties that you can access within it. We also created several variables to allow us to easily fine tune various parameters of the water (e.g depth, smoothness, normal strength, color, and displacement). Finally, we learnt several how to create more realistic looking water (this included preventing the water mesh from casting shadows and colliding — thus allowing the player to enter the water, adding bump maps together to create more realistic textures, and creating the appearance of motion by modifying sample offsets).

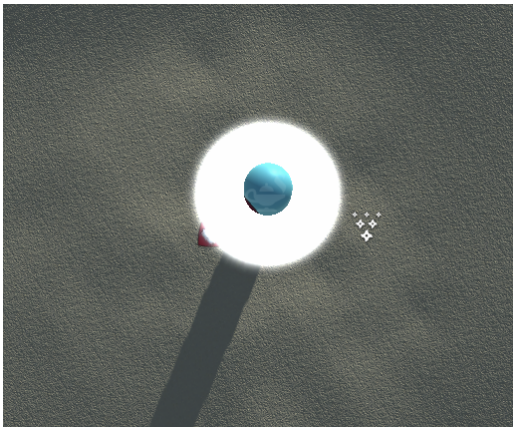
Snow Biome

Through the hallway on your right you'll find the snow biome! The snow biome features falling snow flakes and interactive snow. We implemented this biome because we wanted to learn more about interactive graphics. When Blob Jr walks through the snow, it will leave tracks behind, the tracks last for 30 seconds before disappearing.

Unlike the beach biome, which used a sand texture painted on to the terrain, we wanted our snow to have volume. We thus offset the y-positions of the vertices by random values generated by Unity's simple noise function. This created a slightly uneven texture that we associate with more realistic snow.

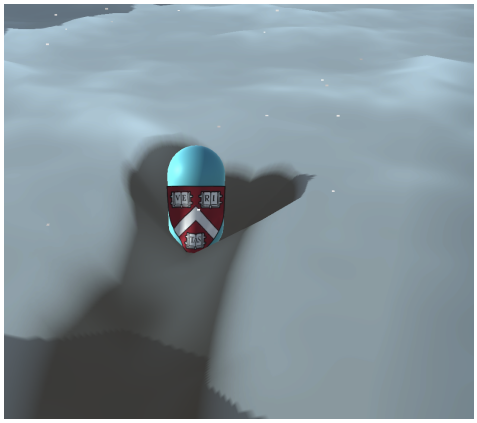


We then focused on creating tracks. To register the player's movement, we set up an orthographic camera over the entire snow biome. We then created a visual effects graph and set it to emit particles of constant radius centered around Blob Jr.



We then set these particles to be invisible in-game by creating a particle layer and removing it from the culling mask. The camera then picks up these particles and outputs the image to a render texture, effectively drawing the path of the player. By adding this render texture to the noise, we create an effect where the player appears to leave tracks in the snow because those vertices have a y-offset of 0. The size of the particles emitted corresponds to the size of the track

left by the player and spawn rate corresponds to the speed at which the player creates those tracks. Since the particles have a lifetime of 30 seconds, after 30 seconds the tracks fade away.



Finally, the color of the snow was set by multiplying a light blue color with the noise and render texture combination. This created light snow and dark tracks.

As a finishing touch, we used Unity's particle effects to create falling snow. By setting velocity and color as uniform random variables we were able to create a more natural falling snow appearance with some snowflakes being darker than others and with all snowflakes falling in slightly different directions.

This was the most difficult section of our game to make and we learned a lot about creating the appearance of volume and how to use cameras and render textures. The most challenging part of this biome was understanding how to translate a particle effect into a change in the vertices' y-position. However, by the end of this biome we had a much stronger understanding of color in Unity as well as the relationship between vertex positions and what is shown on screen. Using a plane was also conceptually tricky as we envisioned the snow to be solid as we carved a track into it. However, in reality we were merely deforming a surface.

Forest Biome

After Blob Jr. treks through the snow biome and goes through the door, the player will find the forest biome, which consists of grass, fog, trees, and mountainous terrain. Like the beach biome, we used Unity's Terrain tools to create and shape the terrain - we used this biome to show the effect of terrain on shadows with the inclusion of the high terrain and stone bridge that cuts across the landscape. However, the forest biome made more extensive use of the terrain tools than the beach - one aspect was in painting terrain. While the beach used one texture, to simulate rock covered by grass, we used two textures that we blended together - first painting the grass texture then adding the rock texture on top.

Another aspect of the terrain tools we utilized was Unity's paint trees feature, which we used to generate trees around the biome. One issue we encountered with the trees was they would not render if the camera was positioned a certain distance away from the trees, causing the trees to suddenly pop into view as the player moved through the snow biome. To resolve this problem, we changed the tree render distance in the inspector. Another issue we encountered was changing the shader of the imported materials to URP, which we had to do for the imported tree prefabs.

In addition to the terrain, we included fog generated through a particle system while changing the duration and start speed of the particle system to have the fog be strongest at the bottom of the biome (the valley) and gradually dissipate as the player climbs to the top of the biome (the mountain).

The most challenging element of the forest biome was creating large quantities of grass without manually dragging several hundred objects into the scene. To approach this, we used a grass painting script to add generated grass to the surface of a terrain, then utilized another script to blend the grass into the terrain by giving it a similar color. Additionally, each clump of grass has its own separate physics - we simulated the effect of wind on the blades of grass, causing them to individually sway. One feature we attempted to implement was interactive grass, where the player's movement through the grass would trample blades of grass directly underneath them and push aside nearby grass. We attempted this by attaching a script to the player that would constantly track the player's position, and modify grass that was within a certain radius from the player's position. We are unsure why this interaction did not occur, but ended up not implementing it.

From the Forest biome, we learned about many additional features of the terrain tool in Unity, as well as some of its limitations - because the terrain tool is based off a heightmap, we could not make features like overhands or arches that have negative space below terrain - custom terrain has to be built with Blender or other tools. We also learned about different uses of scripts, such as how scripts could be used to generate hundreds of grass clumps as well as physics simulations with the effect of wind.

Resources

1. Snow biome
    a. Interactive snow: Unity Shader Graph - Snow Interactive Effect Tutorial
    b. Falling snow: How to Make a Falling Snow in Unity
2. Beach biome
    a. Introduction to terrain tools: The Ultimate BEGINNERS GUIDE To Terrain in Unity!

    b. Water shader: How to make a Water Shader In Unity with URP! (Tutorial)
3. Forest biome
    a. Fog: Volumetric Fog in Unity using Particles (Any Rendering Pipeline)
    b. Grass: Unity | I made a better Interactive Grass Shader + Tool
4. Universal render pipeline
    a. Upgrading materials to URP: How to fix Pink Materials in Unity
    b. Installing URP: Installing the Universal Render Pipeline into an existing Project
5. Intro to unity/creating a 3D platformer
    a. Make A 3D Platformer in Unity #1: The Setup
    b. Make A 3D Platformer in Unity #2: Moving The Player
    c. Make A 3D Platformer in Unity #3: Better Movement
    d. Make A 3D Platformer in Unity #4: Moving the Camera & Smooth Jumps
    e. Make A 3D Platformer in Unity #5: Rotating the Camera
    f. Make A 3D Platformer in Unity #6: Moving With Camera Rotation
    g. Make A 3D Platformer in Unity #7 - Stop Camera Flip & Inverting Camera
6. Cloth
    a. Unity's Cloth Tutorial