# Gradual Gravity Gauntlet

Team: Zhongqiao Gao, Nicholas Liu, Vinay Ramesh

**Abstract**

Gradual Gravity Gauntlet is an immersive platforming game with one singular goal: Fall onto the ground at a low enough velocity so that you can survive the fall! This threshold velocity and current velocity will always be displayed on the screen for the player's convenience so they always know how they are doing. The game uses several aspects of physics such as gravity and collisions in order to simulate a real life skydive. Many realistic objects such as birds and clouds will be in the way in order to slow the diver's fall, and several rings will be in the way to speed the diver up. The rings will cause a lot of stress, so please avoid those rings! Have fun diving and good luck!

## Introduction

Our goal was to create a game which mimics a person skydiving through the air, trying to hit objects along the way in order to slow their fall. We wanted to add certain elements of realism to the project such as birds with flapping wings, gravity such that the diver accelerates through the sky, and elements of air resistance such that the diver hits a certain terminal velocity. This terminal velocity means that due to air resistance, the diver can't go faster than a certain speed from gravity's acceleration. This was possible to implement with the addition of a physics engine into our system, namely the Cannon.js module.

The future COS 426 students wanting to know how to create projects with physics engines would benefit from this project. The physics engine is a bit tricky to integrate into the

game. One can create an object in the game, but it is not necessarily a part of the physics world yet, and so wouldn't be influenced by gravity in the way that we want. Therefore, in the code it is important to, in every time frame, to set the object's visual position to the object's physics world position.

Gradual Gravity Gauntlet will also benefit gamers who enjoy playing challenging games with nice visual and sound effects. At its core, the game is challenging and fun to play, and so those people that enjoy platform-type games would benefit by having a good time. The players enter an alternate world, often more fantastical or appealing than their own, and with the suspension of disbelief that one is falling high from the sky. Obviously, in real life, one can't fall from the sky attempting to hit objects along the way to slow their fall, and so this game allows them to live this fantasy.

Some previous work that people have done in this realm includes Wingsuit Simulator 3, and the video game Sky Diver. Wingsuit Simulator 3 provides very great visuals and scenery with which to fall down to. However, this game does not provide a mechanism with which to slow your fall. No matter what happens in the game, you are guaranteed to land on the ground safely, as you are wearing a wingsuit. Additionally, there are many buildings in the scenery which could clutter up the screen while you are playing the game. The game Sky Diver has pretty bad visuals (albeit it is an arcade game from the 1970s), and unrealistics physics (the diver falls at a constant rate rather than accelerating). Additionally, the camera angle was 2-dimensional, such that you cannot see the fall from the diver's point of view.

We as a group felt that this was a major issue. We wanted to provide a game such that there is a 3D world and a camera angle that shows the perspective of the diver falling through the

sky. Since the camera is fixated on the diver (the diver always remains centered on the screen), we are confident that the 3-dimensional, diver point of view aspect of the game will indeed work very well. Additionally, we believe our players will appreciate the level of realism incorporated into the video game, as there are birds with flapping wings, textured grass and trees, as well as a textured diver falling through the sky. Above all, the game has a level of difficulty to it, as one needs to try and hit objects along the way to slow down their fall, which will prove to be a challenge. However, you can also choose easy and hard modes depending on your skill in gaming.

**Methodology: Physics**

Simulating realistic physics interactions was a major concern of our methodology. At the beginning of our development process we planned to use simple location comparisons between objects' x, y, and z positions to check for collisions. For example, for checking for collisions with the ground we used a y-height comparison between the player character and the y-level of the ground plane. If the player was below the y-level at any given time, indicating that they had reached the ground, then their position would be set back up to the ground level. This approach was similar to the one used in the cloth simulation assignment. However, for our game we decided that this was not realistic enough graphically. Notably, resetting the position of the player to the ground level appears to instantly stop them, regardless of whether they were falling fast or slow when they hit the ground. Therefore we decided to integrate a third party physics library into three.js to provide more realistic collisions.

It was difficult to determine the correct physics library to choose. The more popular and more recently maintained option, ammo.js (https://github.com/kripken/ammo.js/) , is a JavaScript port of the widely used C++ physics engine, Bullet. However, ammo.js has limited documentation and difficult integration with three.js. On the other end of the spectrum, we have Physiji, (https://chandlerprall.github.io/Physijs/), which was especially designed for tight integration with three.js. However, it was several years out of date, and we weren't sure if it would work well with the latest version of three.js. This caution was out of experience after dealing with old three.js versions inside assignments that reacted poorly to functions from the latest three.js documentation.

In the end we landed on a middle ground with active maintenance and ease of use with the slightly out-of-date Cannon.js project, which additionally offers the best documentation out of the three physics libraries mentioned (https://schteppe.github.io/cannon.js/). To integrate Cannon.js with our project, we created a physics world, abstracted as a Cannon.js World object, that would simulate physics objects. These Cannon.js objects would provided position information to the rendered three.js objects in every frame of the game; this was achieved by copying over the physics object position vectors and advancing the state of the physics world inside of our render loop in parallel with updating the camera and scene belonging to three.js.

**Methodology: Diver**

The diver was created out of several gltf meshes that were obtained off of TurboSquid. TurboSquid gave these meshes as a .fbx file. It proved a bit difficult to locate an FBXLoader on JavaScript, so we decided to convert these files into a .gltf format file. There indeed is a GLTFLoader in JavaScript, and so this file format worked out.

Next, we imported the .gltf mesh into an application called Blender in order to edit the mesh itself. After importing the mesh, we had to edit the mesh in the following two ways: first, we had to adjust the arms such that they were spread out, thus making the diver look more like he was actually skydiving. Secondly, we had to orient the mesh such that it was facing downward. Clearly, we wouldn't want the mesh to be standing up while it is diving, so this orientation was necessary.

Our game has two possible divers you can use: a Minecraft character or a US Marine. Selecting a character to dive will also dynamically change the difficulty of the game, as the minecraft character, Steve, survives with a velocity threshold of 90 while the Marine can only handle 40.

**Methodology: Bird**

For the Bird object, we hope to render rigged Birds with animation for the player to hit during the fall. We first found an Eagle .blend file with rigged animation on the Turbo Free 3D Model Website. However, when converting the blend file to a .glb file, the animation information was damaged, as we couldn't render the animation on the online gltf viewer. Finally, we found an open source Parrot .glb file in the three.js example directory. With the Parrot .glb file, we were able to create an animation "mixers" array to store all the animation frames and render all the mixers in the renderer loop for the  rigged movement. In order for the player to hit multiple Bird objects along the journey, we randomly generated Bird objects around the diver in the update loop. We also have a flyDirection variable to enable Birds to fly to a certain direction based on their current randomly generated positions. All the locations updates are implemented

on top of the physics engine in the renderer loop and then copied to the object position for them to show in the rendering Canvas.

**Methodology: Cloud**

Our Cloud implementation is very similar to the Bird. We found a Cloud .gltf file on the TurboSquid website. However, it had three clouds in the.gltf file, which made it hard for us to render the collision. Therefore, we deleted the other two clouds objects in the file, computed the bounding box's center to recenter the object, then randomly generated Cloud around the diver's position. We had a problem when trying to pass the center value to the physics engine in another file. The center value is successfully stored inside the glTF loader function, yet the value disappears after the function. We found out there is a scope problem, as memory will be cleaned up after the function. Thus, we declared the bounding box space outside of the function for passing value purposes. Then in SeedScene.js, we copied the bounding box center value to the physics engine for them to render the collision, and copied the value back in the render loop for every Cloud object.

**Methodology: Snow**

We hope to elicit the feelings of  falling through the sky. Despite all the clouds and birds objects that we passed through, small Snow objects everywhere will further improve the atmosphere of the fall. They give a general impression of descending while passing through many snow particles. We find a tutorial in the three.js examples page to render small particles, and find an open source snow image file for texture mapping. As we are rendering huge numbers of snow at one time, we choose to store vertex information in a BufferGeometry for faster and

efficient rendering, so that we can store all the location values at one time and in one pass for rendering.

**Methodology: Ground**

Because of the nature of our game, we needed a ground surface that extended far enough into any direction so that the camera would not be able to view any of the edges, giving the illusion of an infinite world. The code to set up the ground object is inside of Land.js, and sets up the PlaneGeometry and MeshStandardMaterial that comprise the land. Using webpack to handle the assets, we import a grassy land texture for the land, making use of three.js parameters to use anisotropic filtering on the ground texture as well as enable shadows cast on the ground.

**Methodology: Tree**

Before, we tried to render a pure object tree file. However, object files are inefficient for storing color values, as we need color values for every vertex. A more efficient and common way to render a Tree is to use an .obj file for the vertex value, .mtl file for texture mapping information, and different texture files for specific texture mapping. Therefore, we used an .obj loader inside a .mtl loader. For every child object in the object file, we implemented a specific texture mapping based on .mtl information. We also enabled shadows for the trees themselves and its shadow on the Ground by tweaking the Light object. Finally, based on the num_tree variable, we randomly generated trees with different texture mapping for tree branches on the ground.

**Methodology: Camera**

The starter code included a basic implementation of initializing a three.js camera object, however, we had to extend it to include player tracking, which was no mean feat as result of the

unexpected interaction between the physics engine and the three.js camera's LookAt method. As a first step, we had to disable the OrbitControls found in the starter code, as we didn't want the player to change the view and instead focus on the game.

When we were trying to change velocity in the physics engine for the player control, there was a discrepancy between the actual object vertex value and object position value. Simply calling camera.LookAt saw the player drifting outside the visible screen as it moved. Therefore, we manually computed the bounding box center for the diver object, in order to locate its actual rendering location. Then, we used the center variable instead of the player's location for the camera to look at. In this way, we are able to always keep the view center at the player, creating a third person camera view for this game.

**Methodology: Lighting and Effects**

We were using the default light in the starter kit and it worked fine until we hoped to add shadow for the trees for them to look more realistic. However, the lights are too far away to create tree shadows. Therefore, we configured another directional light on top. Then, we set the light's shadow map size to be 512 and the shadow camera size to be 200,  to increase the directional light range for the tree shadow. In this way, there is a larger range and nearer impact of the directional light on trees to map the shadow on top of the ground.

We also added a fog effect that decreases in intensity as the player nears the ground, to give more visual feedback as the player approaches the ground. We also experimented with three.js's post processing pipeline, adding a motion trail effect to the diver and objects. This required using three.js' EffectComposer and RenderPass features. However, we found that this slowed down the game too much for machines without a graphics card; uncommenting lines

87-88, 111 inside app.js and recommenting line 109 will put the feature back in for the curious tester.

**Methodology: Renderer**

We basically keep the provided rendering loop structure inside the request animation frame. Inside the render loop, we rescaled the time stamp 1/10, as the animation mixers are updating too fast. For the bird animations, we loop through the bird animation mixers array to update the Bird rigged movement. Then, we update the camera LookAt target at the real diver's position, by computing the diver's bounding box's center. After the update, we render the scene and the camera. For the physics engine, we update the step by 1/50 seconds. With physics engine update, we copy the values from the physics engine to update the diver, clouds, and birds positions for rendering purposes. Finally, as response to Phillip's feedback of adding current player's velocity on screen, we update the current diver's velocity, and show on the upper left corner of the game screen for the player to track its progress towards winning the game.

**Methodology: Game Loop/ Interface**

The game loop is a feature in the game that allows a user to start, win/lose, and restart the game. In order to do this, we had to create a landing page where we could choose a diver with which to dive with, and clicking the button starts the game. We used the open-source front-end framework React with CSS for styling, and created our own win, loss, and game logo images using PhotoShop, including a favicon. There were multiple ways in which this could have been done, but we figured this would be the cleanest solution, as there is a landing page with start buttons. Another way to do this is to simply start the game with no landing page, but we felt that a proper landing page would really improve the gaming experience.

For better interaction with users, we add some hover and active effects on buttons as well for users to better engage with the game.

In order to handle the win/loss conditions of the games, the current velocity of the diver must always be known (this is important to the user, and not the program as it provides feedback as to how they are doing in the game). The actual value of this velocity is important at ground impact time (to both the user and the program), as it will be compared to the threshold velocity. If the impact velocity is less than the threshold velocity, then the player wins the game. Otherwise, they lose the game. This seems to be the only way we could have implemented this feature of the game. Note that the win/loss velocity is set based on the player's choice of difficulty level.

In any instance, after the end of the game the player simply needs to click anywhere on the screen in order to restart the game. This seemed to be the best option for us, as the player can restart the game very easily through this mechanism.

**Results**

We measure success based on the diver's velocity on hitting the ground. If the diver's velocity hitting the ground is lower than 90 on easy or 40 on hard, it means the diver successfully survived and won the game. The diver loses the game otherwise.

We received feedback from both TAs and friends. Phillip suggested we can add instructions, a start page, and a loss and a win page. Also, he suggested that we can add a live velocity value on the screen while playing the game for the player to track their progress towards success. We added all the features that he suggested, such as a start page, a button to show instructions, win and loss pages after hitting the ground, and also added the current velocity of

the player on the upper left corner of the canvas. Our friends all liked our game for detailed rendering and dynamic animations. They suggested that using controls to manipulate the velocity direction is a bit hard, but a relative difficulty makes it a fun game to play with. We choose the threshold of winning velocity to be 40 so that this game is above-average difficulty that player has relatively the same tendency to win as to lose. WE also included an easy threshold of 90 that allows the player to win almost every time. We experimented with the velocity threshold ourselves many times and then passed it on for friends to play and further tweak the game based on their feedback.

**Discussion**

Overall, we believe that we took a promising approach to our goal of creating a fun and accessible platforming minigame. The use of three.js allows for anyone to open up and play the game in the browser without the need for downloads or external plugins like Flash. Educational purposes aside, a full-featured WebGL enabled game engine such as Unity may be better for creating a longer or more detailed game, and would also allow for playing in the browser. Longer-term follow-up work would probably see a port of this game to a Unity-like game engine. In our conclusion, we discuss some short-term improvements that could have been made to our project without time constraints.

We learned a lot over the course of building this small game. Most of the lessons we learned had to with the implementation of the game. These were physics engine details with Cannon.js; loading models, audio, textures in three.js; controlling a player character smoothly with user inputs; tracking a player character with a camera, generating and moving 3D objects in the scene “as needed” as opposed to statically generated objects inside assignments, and

dynamically creating sound effects. Other lessons were about the development process: building to production with all assets with Webpack as well as loading and synchronizing packages across team members with npm and git. Finally, we dipped our toes into React, CSS, and integrated interactive UI elements to create the game's menus and interface.

**Conclusion**

In conclusion, we set out to make a short, fun,  falling-centric minigame. We mostly achieved our goal, however, there is always space for improvement to improve the game. Although we continually tweaked the parameters for generating objects into the game world as well as how the player controls handle, there could be a more fun and addicting balance struck with those parameters. For example, the obstacle generation and controls in Flappy Bird are frustrating and fair. However, we also had to balance our goals of making an accessible game.

Although we did not run into problems with performance on our machines, we believe there is room for improvement for optimizations, such as by deleting objects from both the rendering and physics engines when they are no longer necessary. There could also be room to improve the game graphically, for example by generating a non-flat landscape, or adding additional models and obstacles that can slow down or speed up the player.

**Contributions**

- Vinay: I worked on the diver meshes, and integrated them within the physics world. The diver meshes needed to have their arms correctly oriented, as well as the orientation of the entire body such that it faces down. I implemented the bird animations for multiple rendered birds in the scene as well, along with their motion across the scene. I also implemented the win/loss condition for the game, showing the current velocity and goal

velocity so that the user has a good idea of where they stand in the game. I also implemented the logic of the rings: behavior when the diver collides with a ring, the increase in speed when the diver goes through it, and the collision detection between the diver and the ring. Lastly, I also implemented the buttons on the landing page, making sure that the start of the game had a clean looking interface and an easy way to view the instructions/start playing the game.

- Nicholas: I worked on researching and integrating the physics engine, as well as adding the physics materials for the diver and ground. For example, the player experiences less friction with clouds then with the ground, so they don't get stuck. This required evaluating several open-source projects, and then reworking our initial physics implementation to use the parigdm of rendered objects linked to physics objects. For example, this required reworking the diver controls to be physics-based instead of position based. For visual effects, I added and tweaked the fog effect, as well as added an after-image post-processing effect, although this ultimately did not make it into the game for performance reasons. I also created the image assets for the win, loss, and game screens with free images, as well as the favicon. I added easy and hard difficulty selection handling to the game to accomodate a broader range of players.
- Zhongqiao(Olivia): I worked on bird meshes, and enabled bird rigged animations. For the cloud, I randomly generated clouds inside the update function and integrated them into the physics world. Moreover, I handled the massive randomly generated Snow and Tree objects, Ground, Light objects generation and their according different texture mappings based on MTL file information. To improve the rendering, I added shadow by tweaking

the light object. In order for users to have a third person view with the camera center focusing on the diver, regardless of any collision that diver encountered, I handled the camera refocusing and the update. For the rendering of the game, I created a draft starter page and reloading game logics inside the init function for every initial set up of the game. I also tweaked the interactive buttons, instruction renderings and front image rendering, set up the background sound with play and stop logics inside the game loop.

## Works Cited

TurboSquid (downloaded diver meshes from this source).

Free images from Pexel.

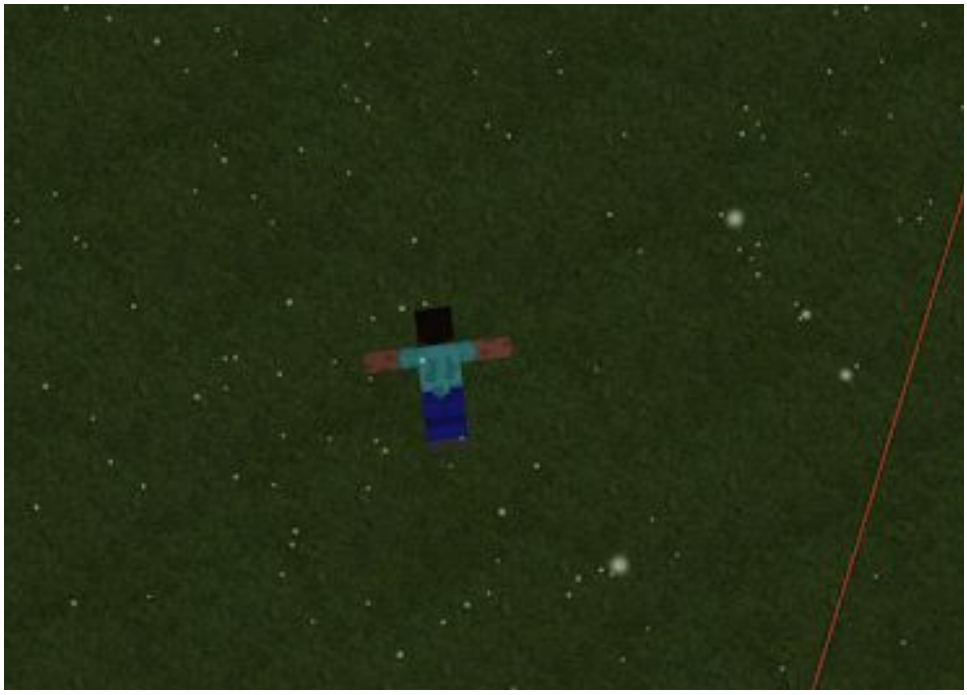
"Three.js/license". github.com/mrdoob. Retrieved 20 May2012.

Background music, Knowing Nothing {140 & 4/4} by Mid-Air Machine is licensed under a Attribution-ShareAlike License.

## Appendix

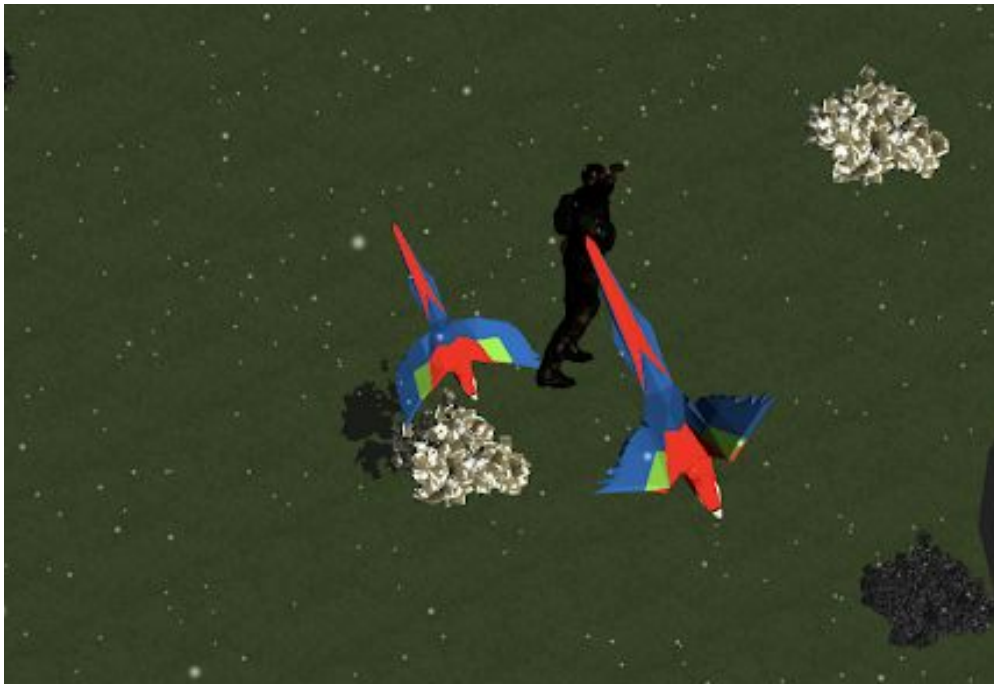
### Methodology: Diver

**Minecraft Diver:**

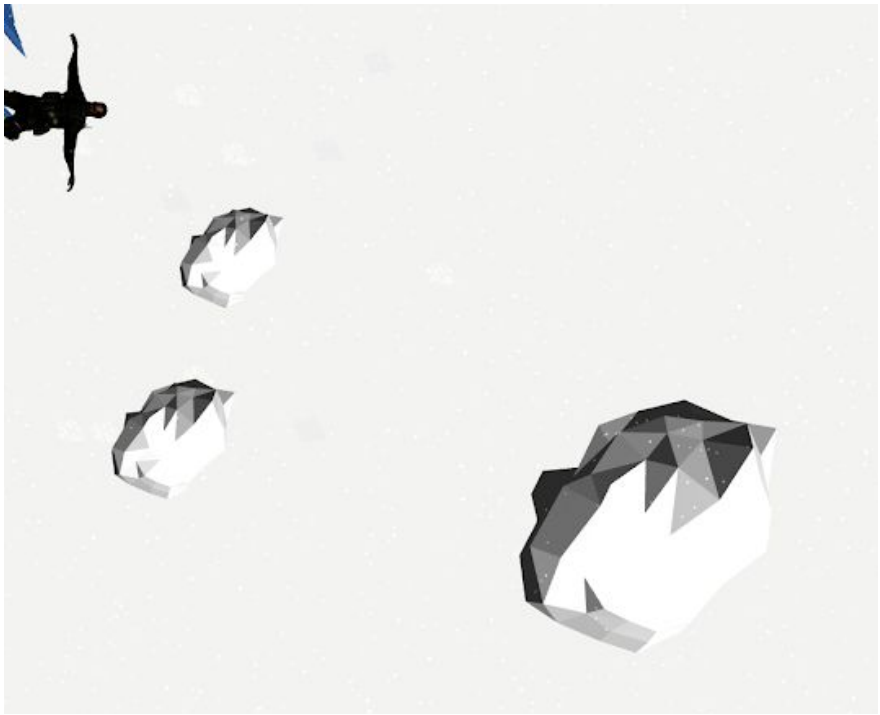


**Marine Diver:**

**Methodology: Bird**

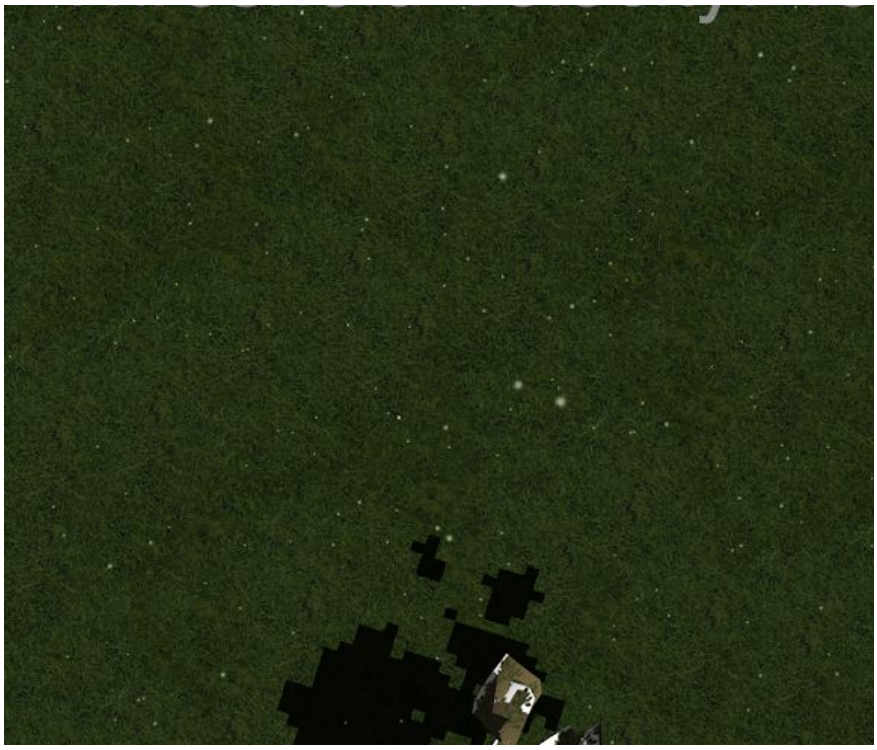


**Methodology: Cloud**



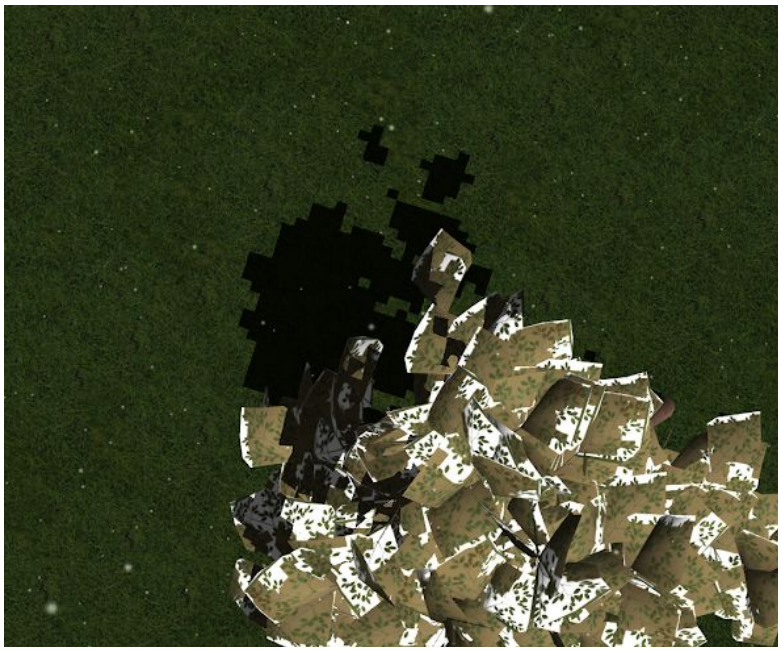
**Methodology: Snow**

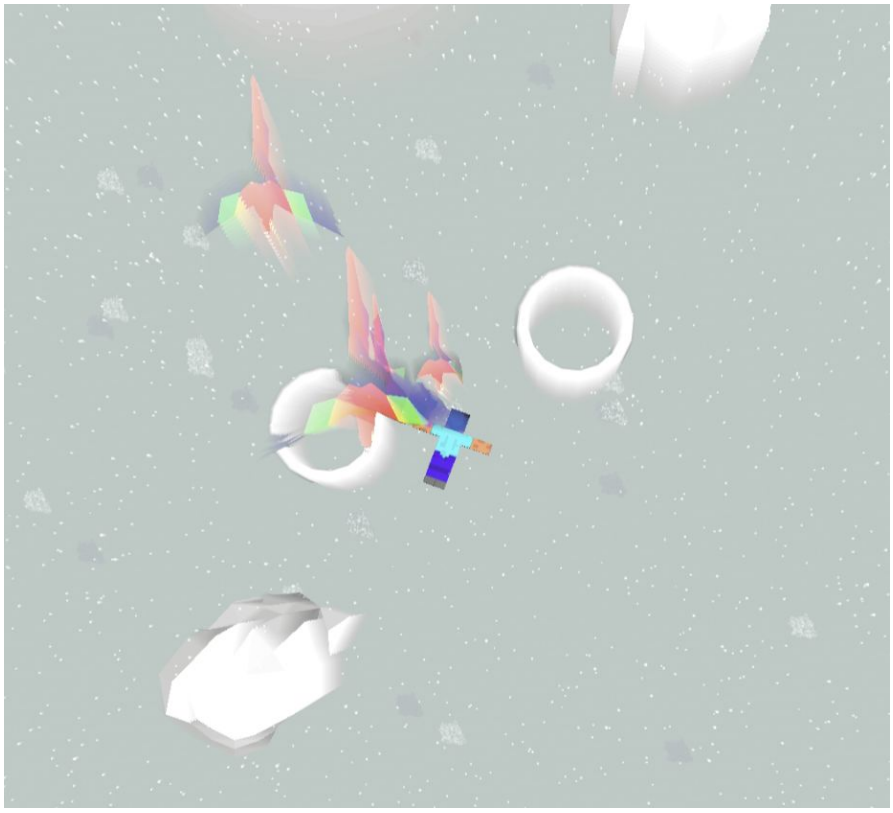
**Methodology: Ground**



**Methodology: Tree**

**Methodology: Lighting, Effects and Shadow**



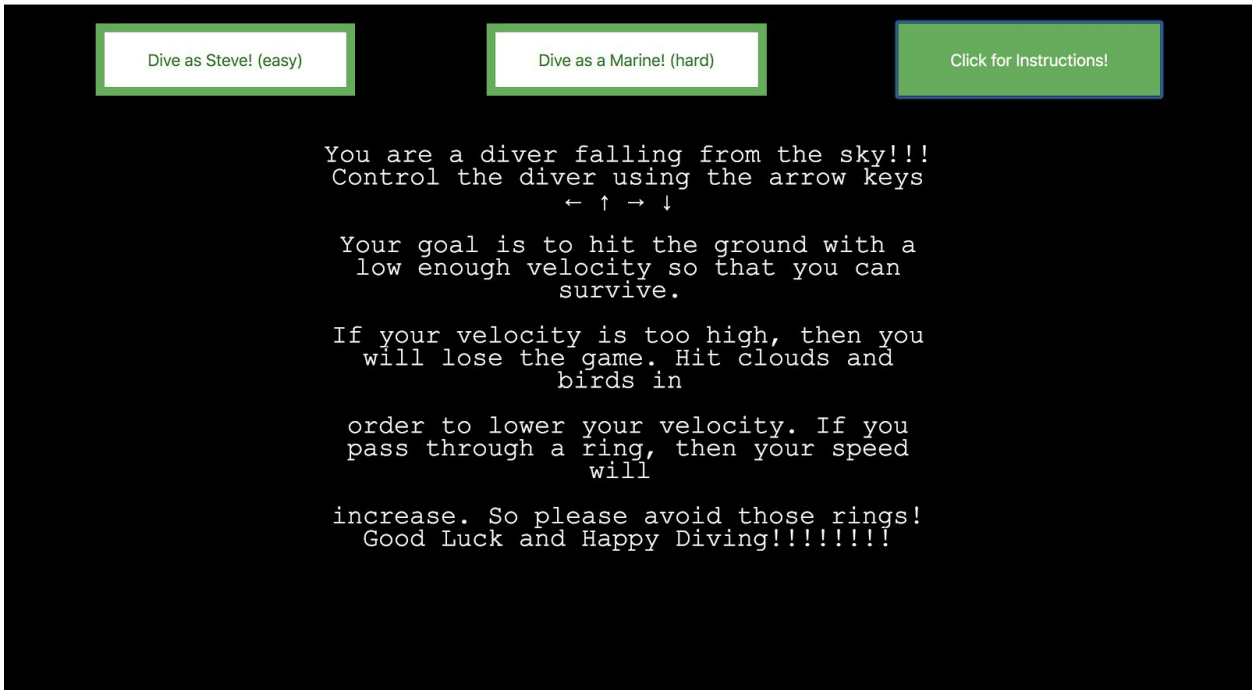


(note: motion trails not in production build for performance purposes)

**Methodology: Game Loop**

**Start Page:**

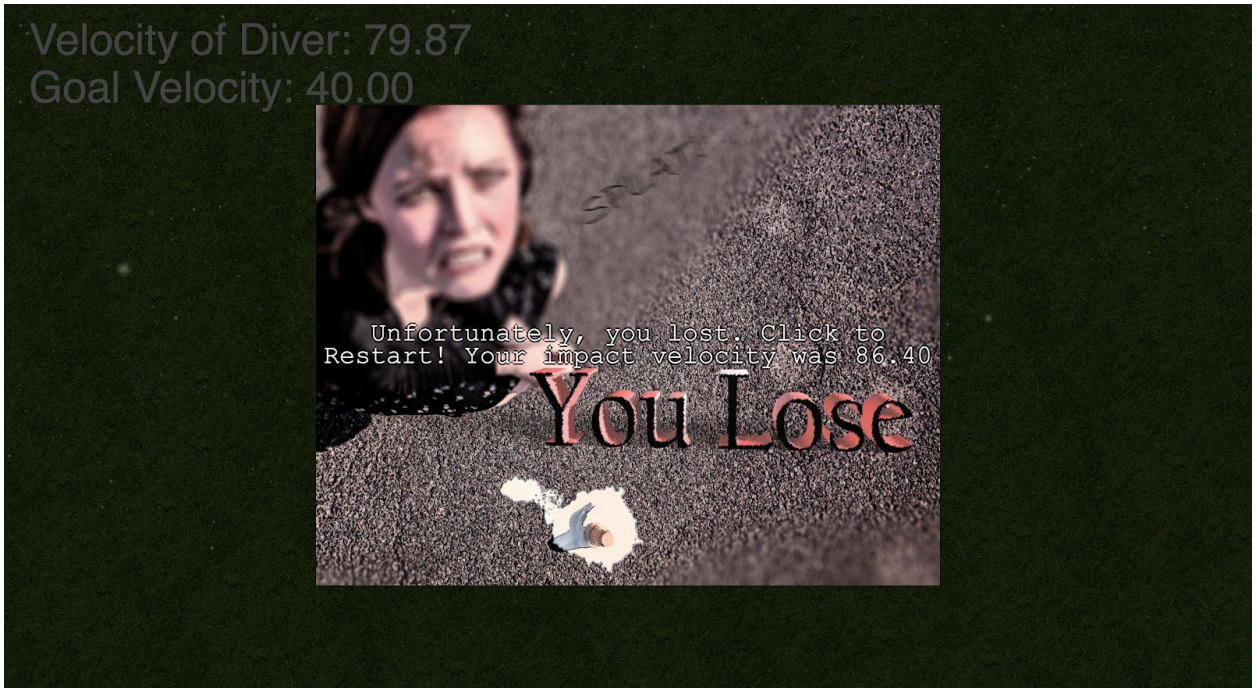

**Instruction Button:**



**Resume Page:**


**Win Page:**

**Lose Page:**


Velocity of Diver: 79.87
Goal Velocity: 40.00
SPLAT
Unfortunately, you lost. Click to Restart! Your impact velocity was 86.40
You Lose