Glider: An Infinite Runner Game

Andy Wang, Harvey Wang, Richard Cheng, Sophie Li

Abstract: Glider is a third-person infinite runner game built with Three.js. The player controls an airplane that glides through multiple terrains, slowly losing altitude over time. The goal of the game is to stay in the air for as long as possible, as crashing into the terrain will end the game. Colliding with obstacles lowers the player's elevation, while flying into boost power-ups increases the player's elevation. This deceptively simple game puts the player's reflexes to the test.

1. Introduction

Infinite runner games are simple yet challenging games that task players with staying alive for as long as possible. Inspired by the infamous glider game in the notorious Nintendo title *Takeshi's Challenge*, Glider is a third-person infinite runner game built using Three.js where the player controls an airplane that slowly loses elevation over time, bound to eventually crash into the terrain and end the game. The player controls the airplane using the arrow keys, but there's a catch: the up arrow key only helps the player fight against gravity but does not allow the player to gain elevation. Instead, players must avoid colliding into obstacles, which lower their elevation, and collect boosts, which increase their elevation. The game becomes progressively more difficult with time, increasing the speed and rate of elevation loss. Our goal is to create a fun, fast-paced game that invites players to try to beat their personal best and compete with others to obtain the highest score.

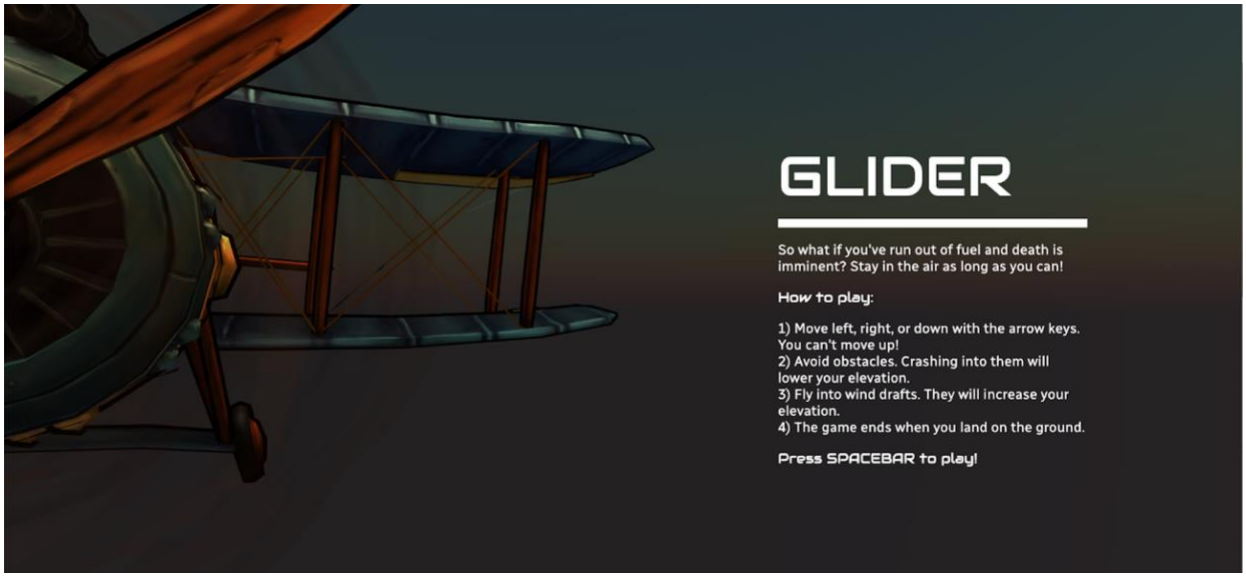

Figure 1. Title Screen.

We have two major sources of inspiration for our ideation:

Takeshi's Challenge
Glider is inspired by the glider mini-game in the Nintendo game *Takeshi's Challenge*. The mini-game is a 2D side-scroller where the player controls a hang-glider that can only move down. Crashing into obstacles lowers the hang-glider's elevation, and flying into wind gusts increases the hang-glider's elevation. The winner wins when they reach a destination island.

We largely adapt the game concept for Glider, except converting it into a 3D infinite runner which also allows for lateral movement. We also make it less infuriating so that players would be incentivized to play rather than give up in frustration.

Dream World
We were inspired by the aesthetic direction of *Dream World*, one of the selections from COS426 Spring '21 Hall of Fame. Particularly, we wanted to emulate the infinite terrain generation from the game, as well as adopt the low-poly aesthetic. However, unlike *Dream World*, we wanted to create a more interactive app in the form of a game. We also wanted to implement complex features like collision detection, which *Dream World* lacks.

Our Approach
We use the starter code provided to us by COS426 staff and build upon it using Three.js. We use Perlin noise for infinite terrain generation and JavaScript event handlers for user controls. We expected this approach to work well as there are a wealth of built-in Three.js functions and objects that we could make use of, like raycasting.

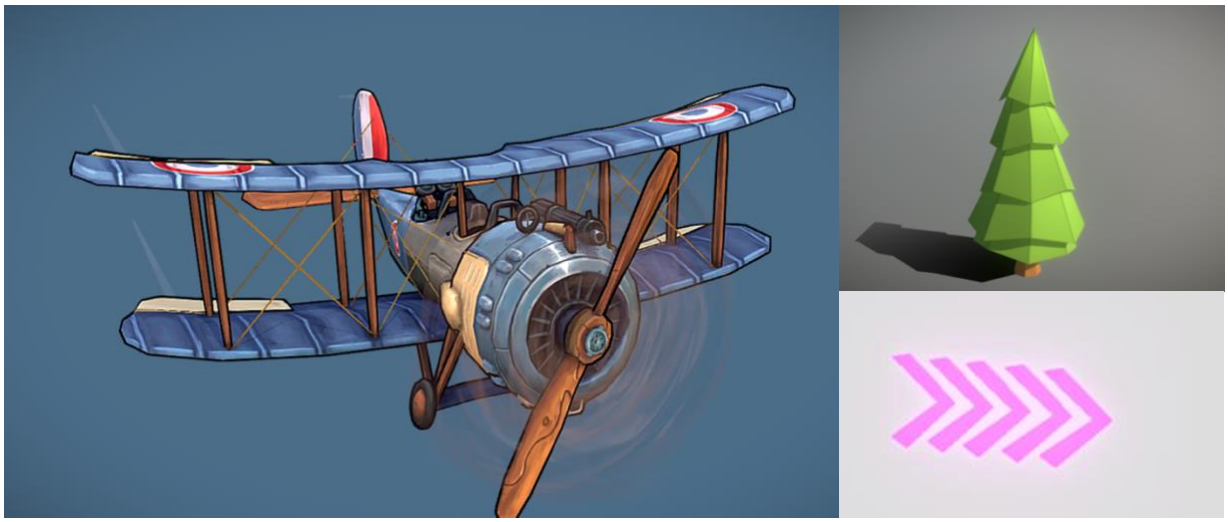

Figure 2. Low-poly models are sourced publicly from open-source model platforms such as SketchFab.

Our MVP goals were to have the player control an airplane through a single environment, avoid obstacles, and collect boosts.

Our stretch goals included changing the environment biomes to create a more interesting world, having the game difficulty increase over time, decorate the landscape, improve the aesthetics and lighting of the game, and improve the UI interface.

2. Methodology

User controls
The user controls are simple - players can move left, right, and down. Our implementation makes use of Javascript event handlers. On a keypress event, we determine if the key pressed is an arrow key, and if so, allow the airplane to move in the corresponding direction.

Our first implementation shifted the absolute position of the airplane in terms of global coordinates but constrained the airplane's position within the screen. However, we simulate the airplane's forward movement and descent by keeping the camera fixed and the airplane's z position fixed, and move the terrain up and towards the camera, so it felt inconsistent for the player to be able to move left and right in the xy plane but not in the z direction. Moreover, limiting the player to a fixed x and y range made the game much more difficult, as the randomly generated terrain is often unavoidable. Last, this implementation is not amicable to resizing the browser window. Therefore, we shifted our implementation such that the airplane always remains fixed at (0,0,0), moving only the terrain to simulate the airplane's movement. For example, moving the airplane to the left is equivalent to moving the terrain to the right. This approach is better as it allows the player to move freely through the landscape in the x and z directions.

For smoother controls, we implemented subtle shifts in camera position and angle to match the motion of the plane. For instance, turning left or right rotates the camera along with the plane and moving up or down shifts the camera down and up, respectively. The controls are also scaled with the speed of the terrain movement such that as the game gets progressively harder, the player also gains more maneuverability to avoid obstacles.

Infinite terrain generation
To procedurally generate new terrain as the player moves forward, we took inspiration from *Dream World*, but because we restrict the player to forward movement, we can make simplifications to reduce complexity and increase efficiency of the terrain generation. We split the terrain into "chunks," and for each chunk we use the simplex-noise package to randomize a height map for the terrain, using frequency and octave parameters according to the biome. The vertices of the terrain are initialized in a square grid, and then shifted up or down according to the height map. The fact that they remain in a grid with respect to the horizontal plane allows us to later perform easy calculations to determine where the player is with respect to the terrain and obstacles/rewards, at the cost of a slightly more rigid appearance. Whenever we load up the terrain, we also shift the positions of the obstacles to positions above random vertices of the terrain, but we do not allow the obstacles of the terrain to become visible immediately (except at the

beginning of the game, when the user spawns in a chunk); instead, we make obstacles visible as they get closer to the player. This is essential to improving the performance of the game.

We maintain a 2x2 grid of chunks, which are organized into "chunk lines" aligned with the forward axis. Whenever we enter a new chunk along the forward axis, we shift the previous chunk (which is now behind the player) to be ahead of the current chunk in each chunk line, and we reinitialize its height map, obstacles, and rewards and hide all of the obstacles/rewards from the scene. Then whenever the user gets within a chunk of the next invisible obstacle/reward, this obstacle/reward is made visible. Similarly, whenever a user passes an obstacle/reward, it is made invisible. For left/right movement, we shift chunklines left/right when the user gets sufficiently close to the edge of the current chunk grid.

Collision Handling: Obstacles

Handling collisions with obstacles is a challenge because the models we use are irregularly shaped. Our first implementation used simple axis-aligned bounding boxes, as Three.js has a built-in method for detecting collisions between boxes. Although this method is computationally efficient, there are two problems. The first is that the boxes are axis aligned, and as such do not rotate with the models when they rotate, resulting in inaccurate collision detection as models move over time. Moreover, many meshes are irregularly shaped, so bounding boxes are not precise, resulting in collisions occurring far away from the actual obstacle.

Figure 3. Visualization of the rays emitted from the airplane.

We considered importing a physics engine such as Ammo.js or Cannon-es to handle collisions, but we decided that since we aren't simulating other physics, we ruled this option out due to an unnecessary amount of overhead.

We ultimately decided to use Three.js's raycasting. For every position in the airplane mesh, we cast a ray pointing outwards in the direction of the normal and check if the ray intersects any obstacles in the scene. To optimize collision handling, we only check for collisions with obstacles within a certain xyz range, since the camera and airplane are fixed in the global coordinate system and so only obstacles near the camera and airplane should be considered as possible collision hazards. Moreover, we only check if the first returned collision is within a close distance of the airplane, since Three.js's

raycasting intersection method returns all intersections in increasing order of distance. We call the obstacle collision handling every frame in the main render loop.

Collision Handling: Boosts
We decided that ray tracing was wholly unnecessary for a simple shape like a boost rather than the more complicated structure of obstacles. We centered the hitbox of the boost at the middle of the arrow, and essentially made the hitbox an invisible sphere around the boost – if the plane comes within a certain radius of the center point of the hitbox, it is boosted. To further visualize this hitbox both for aesthetic and gameplay purposes, we also added a transparent green sphere around the boost arrow, visualizing exactly where you should hit in order to be boosted in the air and continue to play the game.

Collision Handling: Terrain
For terrain collisions, we used the predetermined heights for every vertex in the terrain that were computed in the terrain generation process. The heights at every vertex were stored in a 2D array for every chunk, so to determine the terrain height at the plane's current point, we calculated the relative offset of the terrain and the current chunk and found the nearest four vertices. We attempted several versions of vertex calculations including rounding down to the nearest vertex to our left and using the closest vertex to the current position. However, we found that the vertices were spaced too far apart (100px), causing the collision detection to make errors frequently if the plane was between two vertices of very different height. Instead, by using the four nearest vertices, we apply barycentric coordinates to find the weighted average height of the nearest vertices and compute the precise height of the terrain at the plane's current position. If this weighted average height is above the plane, we initiate the end game sequence.

Animations
Several of our Sketchfab models came with animations, including the plane and the boosts. To further supplement these animations, we made modifications to the magnitude and direction of some of our animations by manualling rigging them in Blender. For several of the movements of the plane, including its swaying, propeller speed, and wobble, we leveraged Three.js's animation mixer to integrate these animations from Blender into the game.

In addition to these "pre-made" animations, we added our own animations directly into our code by taking advantage of the fact that the objects are continuously updated. We added several animations that are based on an object's state (for example, whether the plane has just hit a boost.) So, our coded animations include the barrel roll of the plane upon hitting a boost, the spinning of the boosts themselves, the movements of the clouds, and the feeling of flying forward (which is just the terrain moving towards the player) – all of these movements are based on the continuous updates of the object.

Biome Changes
Biomes differ from each other in the different parameters used to control the terrain's appearance. These parameters include the bank color of the terrain, the middle color,

the peak color, the water color, the exaggeration of the terrain, and the decorations on the terrain. We create a dictionary of biomes mapping a biome name to the set of unique parameters that define that biome. Every 500 frames, we randomly select one of the biomes and update the terrain's parameters to create the biome.

Some biomes also feature decorative elements such as trees, cacti, sheep, and penguins. We used models from Sketchfab and randomly chose vertices on the terrain

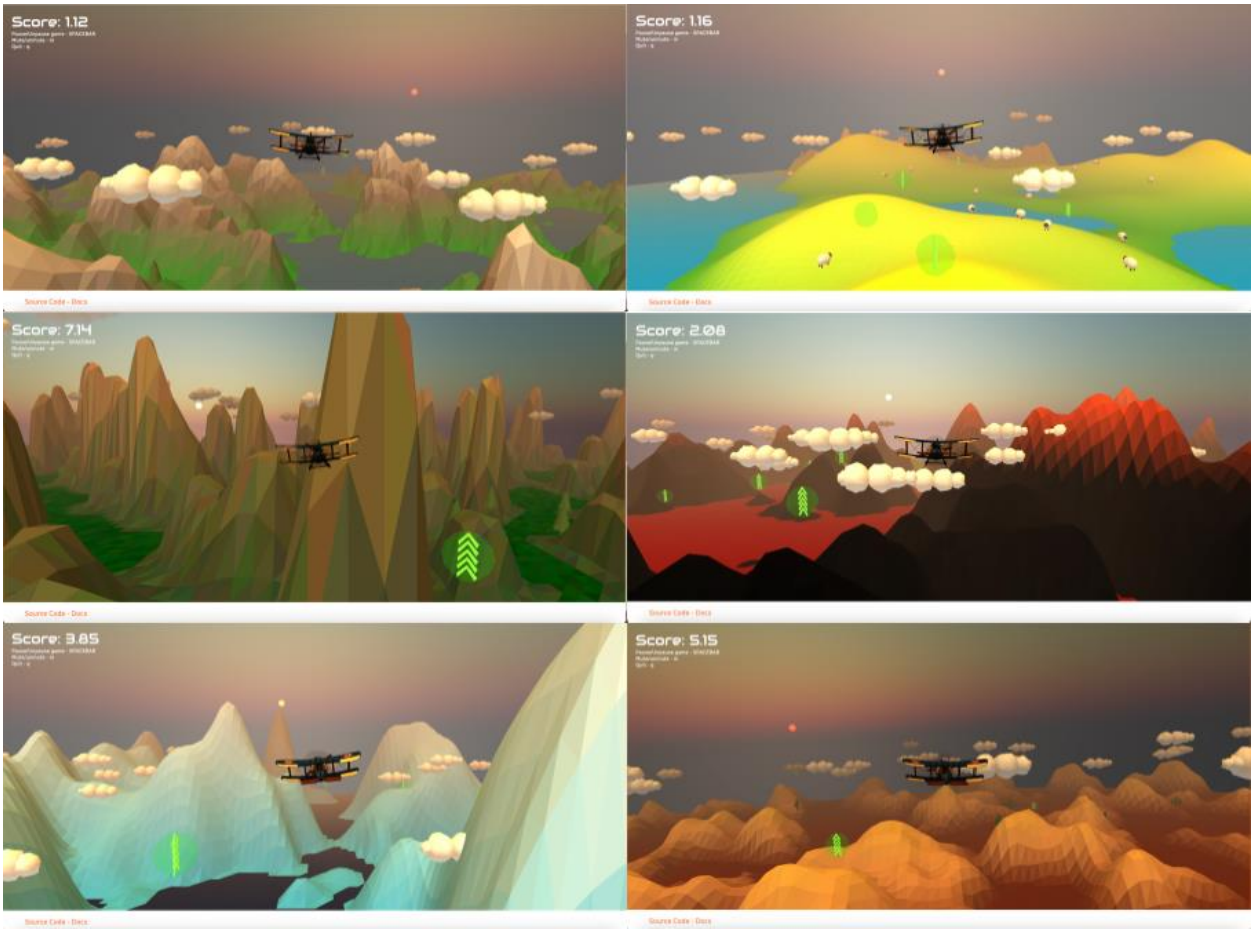

Figure 5. The six different biomes: Wetlands, Grasslands, Stone Forest, Volcano, Arctic, and Desert.

from the subset of vertices lying within a minimum and maximum height. This allowed us to place penguins in the water and the other decorations on land, and we also varied the number of elements in every terrain.

Game Progression

We increase the difficulty of the game over time by maintaining a frame counter. Every 450 frames, the speed of the airplane (or the rate at which the terrain moves up and towards the camera) increases by 10%. The game's speed maxes out at 3x the original speed.

Originally, we instantiated a Three.js Clock object, and increased the speed every fixed number of seconds. However, we noticed that the frame rate across devices vary widely, so using frames as a measure of time would make the game more consistent on different devices. Using a clock, for example, would mean a device with a lower frame rate would experience speed increases at earlier distances. Moreover, using a clock resulted in a bug where even when the game is paused, the difficulty would continue to increase, so upon unpausing, the game may be more difficult than it was when it was paused.

Day/Night Cycles

For our sky, we borrowed a sky shader implemented by Simon Wallner and Martin Upitis. This shader models real life physics with the optics of the sky including a consideration of Rayleigh scattering and turbidity: most importantly, it illuminates the sky based on the sun's position, which can be moved.

To simulate day/night cycles, we initiate the skies to have updatable variables, including the azimuth of the sun, a Rayleigh coefficient, exposure, and the elevation. Then, as the game progresses and the skies update, these variables slowly increment as well: the most notable change is that the sun simulates rising and falling with its "azimuth" and elevation properties: because of this, the sky illuminates and darkens like a day/night cycle as the player plays the game. The turbidity also slightly varies from day to day to provide subtle differences in the sky's lighting.

The sky actually provides no lighting, so we included a hemisphere and directional light to illuminate the plane and terrain. The intensity of these lights dynamically changes with respect to the sun's elevation such that it is darker at night and brighter during the day. Correspondingly, we added fog to smooth out the horizon and give a sense of depth, and this fog color darkens at night to match the lighting.

UI

For the UI, we implement a menu screen, a game screen, a pause screen, and a game over screen. We do so by manipulating the DOM using Javascript, similar to how *Portal* from COS426 Hall of Fame does. We maintain a dictionary of screen names mapped to booleans, with the corresponding screen set to true when it is displayed. Screens are toggled via keypresses - starting the game, for example, is triggered by spacebar; pausing the game is similarly triggered by spacebar; mute is triggered by the m key; quitting is triggered by the q key; and restarting is triggered by spacebar on the game over page. An event handler listens for such keypresses, sets the screen booleans, and removes/adds elements to the DOM accordingly.

For the menu page, the background image is a separate scene that is instantiated when the game first loads.

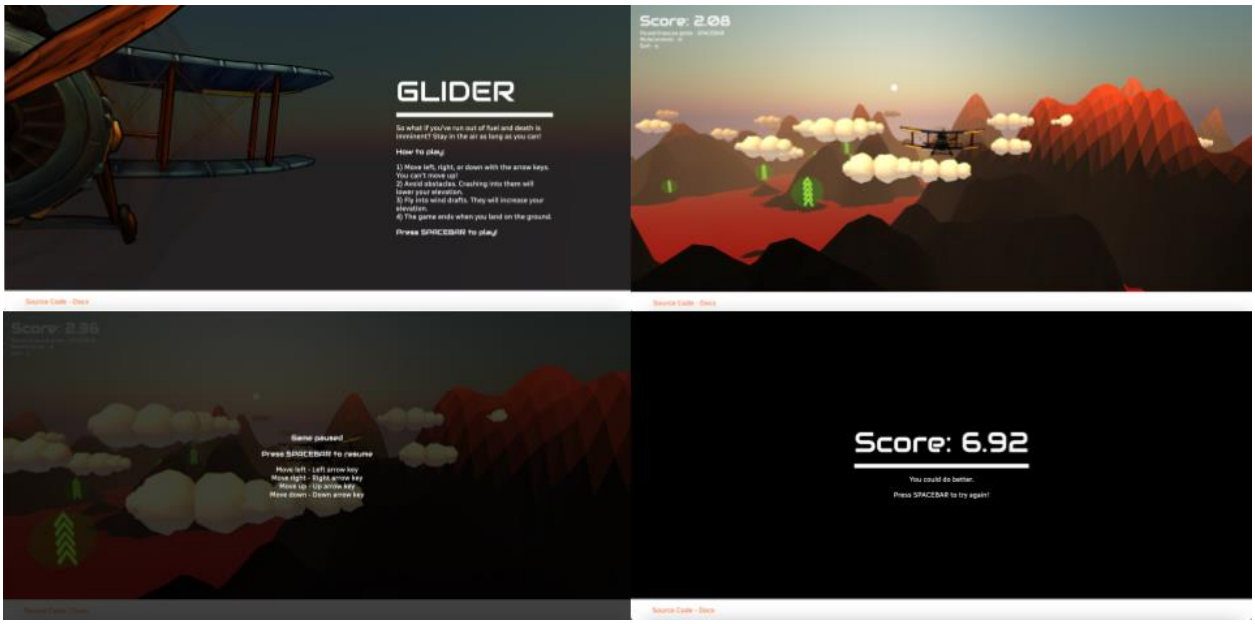

Figure 4. Title screen, game screen, pause screen, game over screen.

Scoring

We maintain a score counter. The score reflects how long the user has stayed alive. The score increments by 0.01 every frame. We increment the score based on frame rather than time to ensure consistency across devices which may have different frame rates.

The score is displayed on the top left corner throughout the game. It is updated continuously using Javascript. The score is also displayed on the game over screen. To add insult to injury, the game gives the player a passive aggressive insult depending on their score.

Audio

For the audio, we make use of both Three.js and the HTML Audio DOM element. Sound effects such as the crashing sound effect, the powerup sound effect, and the whirring propeller sound effect are implemented using Three.js's audio. However, the background music is implemented using an HTML Audio DOM element, because HTML Audio allows for altering the playback speed of the audio without altering its pitch. We have the background music become faster and faster the lower the player gets to the ground to induce a sense of panic, so using HTML Audio is more suitable for this purpose.

We also include a mute toggle which silences/plays all music and sound effects. This can be triggered by pressing the m key. All audio tracks are sourced from freesound.org, licensed under Creative Commons 0 (no copyright).

3. Results

We have produced a fully functioning game with polished, cohesive graphics and user interfaces.

Our metrics are mainly qualitative. The first metric we use to evaluate the game is seeing how playable the game is, and whether or not it has noticeable bugs. As developers, we assess our game by testing it thoroughly during production, such as trying to collide into obstacles, powerups, and terrain from different positions and angles and seeing if the game behaves as expected.

We have extensively debugged all forms of object detection and believe that the current game is largely free of collision bugs. We also test the UI extensively by testing edge cases, such as pressing a combination of various keys on different pages to ensure that the game switches between menu, game, and ending screens, as well as paused and mute states, seamlessly. We have not found errors in the current edition of our game.

The second metric of our game is seeing how players receive it. Being a game, Glider's success largely depends on whether or not people enjoy the game. We have had friends act as "beta testers" who play the game throughout various stages of its development to provide feedback. As of the final version, Glider has been positively received by other students, who claim it is fun, addicting, and aesthetically pleasing.

4. Discussion & Conclusion

Overall, we have attained our goals and have a fully playable game that is positively received by peers. We were able to produce our MVP game early on and achieve many of our stretch goals. Our infinite runner has simple controls, but at the same time feels dynamic, which we achieve using random infinite terrain generation, varying the environment biome, introducing day and night cycles, increasing the speed and difficulty of the game over time, and matching the audio speed to the player's elevation. The game is able to run smoothly due to our choice of using low-poly models, utilizing Three.js effectively, and optimizing our code to minimize computations.

Future work on this game includes creating a leaderboard, which was requested by our players, or creating more power ups, like fuel packs that can be deployed to give the player boosts or temporary invincibility. We could also further improve the aesthetic feel of our game, such as making the power ups look like wind drafts, which we had difficulty implementing, or adding effects like snow and rain.

5. Contributions

Andy Wang
I worked with Harvey to implement the infinite terrain generation and terrain collision system. I was also involved in implementing the obstacle/cloud/reward generation and

collision systems. In general, I also helped with debugging various issues, particularly involving collisions and chunk management.

Harvey Wang
One of my main focuses was the visual appeal of our game. I implemented the sky with its day/night cycles, as well as the lighting and fog. I also contributed to the decorative models seen in each biome, such as the penguin, sheep, and cacti.

I also worked on player and camera controls to improve the flying experience. This involved setting up rotations and camera adjustments to make turns and movements smoother.

I also worked with Andy to set up the terrain for infinite generation. This included structuring the chunk manager to handle “chunks” of terrain. I also contributed to the terrain collision detection system and associated calculations to improve the accuracy of the collisions.

Richard Cheng
My focus for this project was manipulating objects. I animated the plane’s movement, sway, and propeller speed using Three.js and Blender: this involved rigging the plane model and adjusting the frames of the animation to our desired magnitudes using Blender’s armature feature. I added the boosts and changed their emissive and metallic material properties for aesthetic purposes, as well as animating them to slowly spin in a circle.

Additionally, I worked on the plane’s interactions with objectives and obstacles: this involved animating a blinking phase when the plane hits a cloud and is temporarily invincible, wobbling the plane upon collision, increasing propeller speed upon collision of a cloud or boost, and animating a barrel roll when a plane hits a boost.

Sophie Li
I worked on several key parts of the project. One key component I contributed to was setting up the basic Javascript handling structure. I set up the event handlers that listen for keypresses that handle the controls and UI elements (muting, pausing, quitting, etc). I also created the UI, score counter, and the associated logic, such as setting up the menu, game, and game over screens, and the toggles between them. In addition, I found and implemented all audio/sound effect-related features.

My other main contribution was setting up collision detection with obstacles. I worked on researching the most suitable approach for our needs, including trying to install a physics engine, working with bounding boxes, and utilizing raycasting. I implemented the raycasting mechanism used in the final product.

I also implemented the environment-changing logic which allows the terrain to change biomes over time (such as changing from desert to grassland).

In addition, I added the feature where the game becomes progressively more difficult with time. I also contributed substantially to this report.

6. CC Attributes and Credits

Our code skeleton was provided by Reilly Bova ’20.

Inspiration:

- *Takeshi’s Challenge,* Nintendo 1986: https://en.wikipedia.org/wiki/Takeshi_no_Ch%C5%8Dsenj%C5%8D
- *Dream World*: https://dreamworld-426.github.io/dreamworld/

Models/Animations:

- Airplane: https://sketchfab.com/3d-models/stylized-ww1-plane-c4edeb0e410f46e8a4db320879f0a1db
- Boost arrow: https://sketchfab.com/3d-models/arrows-animated-025581191d0a4452aa5fe3b51c68bc40
- Tree: https://sketchfab.com/3d-models/low-poly-tree-concept-e815f8acd6d34528a82feef38d5af880
- Sheep: https://sketchfab.com/3d-models/low-poly-sheep-20ef523bee784e4a939b4c5f8734edfc
- Cactus: https://sketchfab.com/3d-models/cactus-add594bc9bfc465488bf76badfdd82cb
- Penguin: https://sketchfab.com/3d-models/penguin-swimming-855d54f1e67a4564aac7871b30ef687d
- Sky: https://github.com/loginov-rocks/three-sky

Audio:

- Main music: https://freesound.org/people/code_box/sounds/520192/
- Explosion: https://freesound.org/people/derplayer/sounds/587183/
- Propellor whir: https://freesound.org/people/rambler52/sounds/567742/
- Boost power up: https://freesound.org/people/GameAudio/sounds/220173/
- Obstacle crash: https://freesound.org/people/EvanSki/sounds/570241/

Terrain Generation Framework:

- *Dream World*: https://github.com/dreamworld-426/dreamworld

HTML Loader Reference:

- Portal*:* https://github.com/efyang/portal-0.5

Other References:

- Three.js documentation: https://threejs.org/
- User Controls: https://blog.logrocket.com/creating-game-three-js/
- Raycasting: https://stackoverflow.com/questions/39911602/physijs-simple-collision-between-meshes-without-gravity