Fury Rush: An Endless Flying Game

COS426 (Computer Graphics) Final Project Submission

Mila Bileska, Jason Oh, Chanketya Nop, Kritin Vongthongsri

Abstract Fury Rush" is an endless flying (runner) game built on HTML, CSS, TypeScript, and Three.js. The game showcases Toothless, a Night Fury dragon, soaring above the clouds at the break of dawn. However, having recently regained his ability to fly, Toothless is struggling to control his flying speed. The player must assist Toothless in avoiding obstacles by moving left, right, double left, double right, and has three lives to withstand collisions before Toothless descends to the ground. Additionally, Toothless can regain lives by collecting health power-ups. The user's goal is to help Toothless survive as long as possible.

Figure 1. Mid-Journey Generated Game Image

1. Introduction

1.1 Motivation. Endless runner games offer a perfect blend of challenge and addictiveness. Our project aims to replicate this thrilling environment, providing players with an opportunity to compete for the highest score. In addition, games like Temple Run and Subway Surfers, often feature low-definition objects and lack aesthetic appeal. Our game draws inspiration from the heartwarming moment when Hiccup introduces Toothless to Astrid. We aspire to deliver a visually pleasing and serene gaming experience, all while maintaining the exhilaration and rush of soaring through the skies and skillfully avoiding obstacles at high speeds.

Figure 2. Scenic Inspiration

1.2 Previous Work. Our primary goal was to make a game inspired by Subway Surfers and Temple Run. Subway Surfers succeeds in being well-rounded in terms of gameplay and user interface. Each player has a profile, where they can purchase different characters, link to social media accounts, and keep track of their (as well as other users') high scores. In terms of the gameplay, there is a diverse set of obstacles and terrains that the player moves through. Consequently, this makes the game more interesting and addictive. Moreover, the game runs smoothly across many devices and platforms, allowing for a wider audience.

A critique of Subway Surfers is the inability to move multiple lanes at once. The developers navigate this by adding a jump boost that enables players to skip a lane and land on the following one. However, this mechanic is not always available and its uses are limited. We add this feature to our game, where a click on the up-down arrow keys incites a new animation of Toothless and moves the playable character two lanes in the indicated direction.

1.3 Approach. Using the idea of infinite terrain generation, we were able to implement an obstacle dodging game. Unlike Subway Surfers, our game, Fury Rush, is set in the sky where the player is supposed to dodge floating islands, as well as air balloons. To make it more engaging, we added a power-up that increases a player's life by one heart, thus enabling the gameplay to last longer. From our Subway Surfers, we adopted the idea of discrete movement lanes (in contrast to that of Temple Run, where the player can continuously move along the x-axis). Additionally, we implemented an infinite object / lane generation with a static camera, for easier tracking and navigating of the playable character, Toothless. We also implemented a time-based scoring system, rewarding those who can navigate the game for an extended period without exhausting their three lives.

We utilized the course's alternative starter code, incorporating Typescript, Node, and Vite, to initiate our project. For the scenic backdrop, we leveraged Three.js's

open-source sky.js example and crafted clouds using Three.js planes along with a PNG image. To bring Toothless and various other obstacles to life, we sourced free 3D GLTF models from various online platforms and programmed the physics and behavior of each object from the ground up.

From the initial project discussions, our approach centered around having the terrain and obstacles move toward a static camera. The playable character is constrained to an axis (specifically the x-axis, as the camera faces +z) and can only move restrictively along it. This design creates the illusion that the character is advancing forward while allowing for lateral dodging of obstacles. This choice contributes to the game's simplicity, a goal we aimed to achieve. Our objective was to create an addictive and somewhat challenging experience while keeping the game mechanics straightforward and easy to comprehend.

After implementing the 3D models and the cloud scene, we built hitboxes that envelop each object. Upon collision, our program registers and determines whether it was a power-up (a heart-shaped object, resulting in the acquisition of an additional life) or a collision with an obstacle (leading to the deduction of one of the player's three lives)

2. Implementation

We modularized our code into three main components. First, we have “app.ts” which contains the main update loop and all the UI elements as well as the game state logic. At the second level, we have “SeedScene.ts” which is responsible for maintaining and loading all the objects into the scene. “SeedScene” provides a useful interface for “app.ts” to call and interact with all the game objects. At the bottom level, we have the game objects themselves that provide an interface to be called from “SeedScene.ts”.

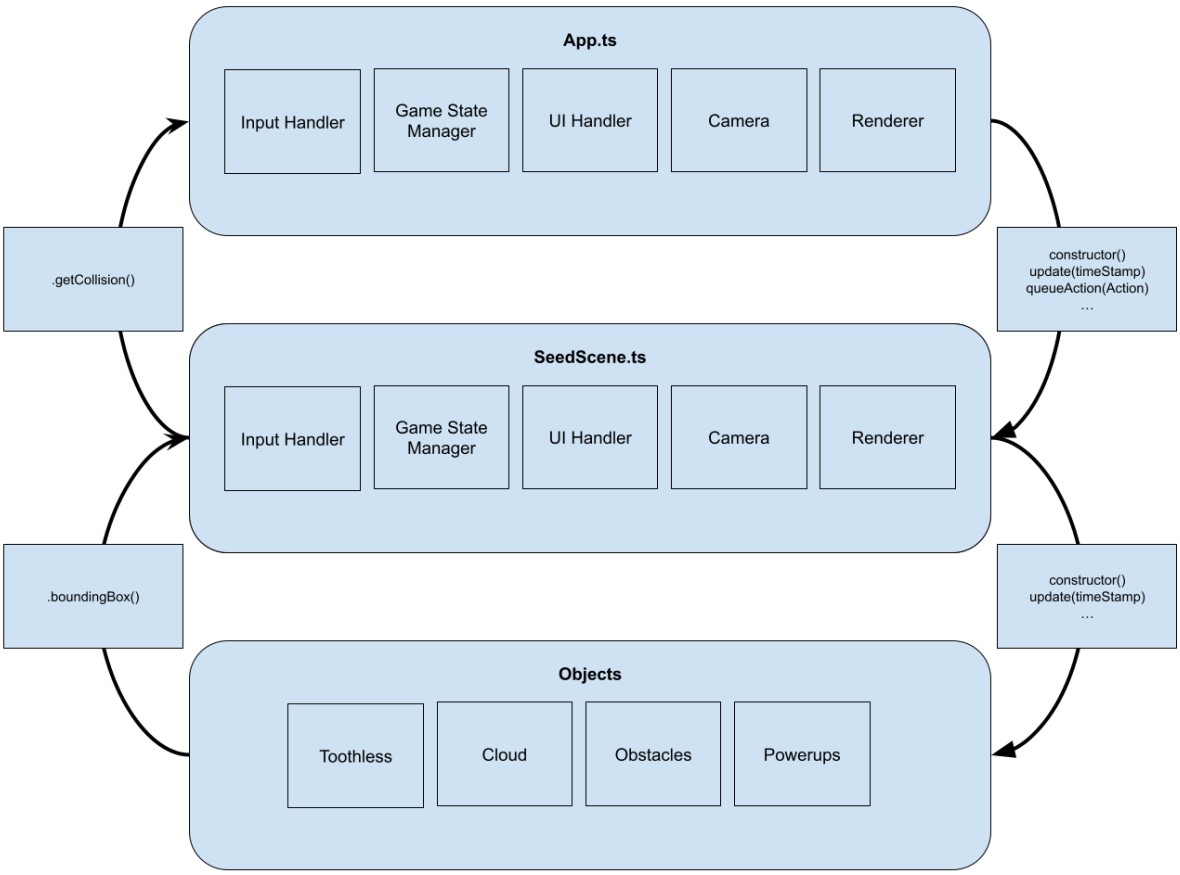

Figure 3: System Architecture

2.1 Character. The Toothless class represents the animated character Toothless in a Three.js environment.

Toothless (src/objects/Toothless/Toothless) The class handles Toothless' movement, animation, and bounding box for collision detection. It provides a movement interface for the input handler to call, as well as handles the actions queue for smooth animation and lane transition. The Toothless model is loaded from an FBX file which comes with a rigged animation attached. The animations are then loaded and managed using the Three.js AnimationMixer. The character can move left and right, perform double moves, and perform a spin move. The movement animation is achieved by changing the position and rotation of the objects and the blinking animation is achieved by adjusting the model's opacity. In addition, the class constrained Toothless to the lane boundaries to ensure Toothless stays within them during movement.

2.2 Scenery. Our scene consists of a visually engaging sky, complemented by clouds that enhance its aesthetic appeal.

Sky (src/scenes/Sky.ts) We utilized the Three.js examples library to implement a dynamic sky in a 3D environment. The Sky class is employed to generate the sky, and the position and appearance of the sun are controlled by a set of parameters defined in the effectController object. The code adjusts various sky properties such as turbidity, rayleigh, mieCoefficient, and biDirectional, updating the sky material's uniforms accordingly. By converting azimuth and elevation values to spherical coordinates, the sun's position is determined and reflected in the sky's appearance. The negative scale of the sky is set for typical Three.js sky objects. The implemented sky is then added to the main scene (SeedScene in src/scenes/SeedScene.ts).

Clouds (src/objects/Clouds/Clouds) The implementation involves defining shaders for rendering clouds, loading a texture for realistic cloud appearance (from a png image), setting up fog to enhance depth perception, and creating a material that incorporates these elements. A plane geometry is utilized to represent individual clouds, and a group named cloud is employed to organize and manage multiple cloud instances. Through a loop, numerous clouds with randomized positions, rotations, and scales are generated and added to the group. The updateCloud function ensures the animation of the clouds, creating a continuous and looping movement effect.

Our implementation of Clouds.ts drew inspiration from this YouTube video, accessible on this specific website. However, we encountered two significant issues with the provided code. Firstly, the version of Three.js used in the reference was outdated. Secondly, the initialization of all clouds (n = 8000) before rendering using Three.BufferGeometry restricted the potential for infinite terrain generation. To address these challenges, we modernized the code to align with the current version of Three.js and introduced an infinite generation feature by replacing Three.BufferGeometry logic with Three.Group. Initially, we attempted to create a new cloud every timestep, but this approach proved to be laggy. Consequently, we opted for generating a fixed number of clouds (n = 2000) and sending each cloud back to the furthest possible distance once they moved out of the frame, ensuring a smoother and more optimized performance.

2.3 Obstacles and Power-ups. All obstacles and power-ups are controlled by a manager class called Obstacles. The Obstacles class is a comprehensive obstacle and power-ups manager designed for a Three.js 3D scene. Its primary purpose is to control the spawning, movement, and removal of diverse objects within the scene, including obstacles and power-ups.

Obstacle and Power-ups Manager (src/scenes/Obstacles) The class maintains an array of unique obstacle and power-up classes, allowing for dynamic instantiation and management. The update method, crucial to the class, iterates through the objects, updating their positions based on elapsed time and performing collision detection with a

player character (Toothless). The collision information is then stored in the class's state, contributing to a responsive and interactive gaming experience. Additionally, the class offers helper functions for random object spawning, lane generation, and efficient management of the array of objects.

The modular design of the Obstacles class promotes extensibility and flexibility, making it adaptable for various gaming scenarios. The separation of concerns, such as collision detection, object removal, and object spawning, enhances code organization and readability. Overall, this obstacle manager encapsulates the core functionalities required for handling obstacles within a Three.js scene, providing a foundation for creating engaging and dynamic 3D gaming experiences.

Models for obstacles and power-ups were found on Sketchfab. They were loaded using GLTF Loader in their respective.ts file located in the Objects folder. The Island and Balloon are initialized using a random rotation, whereas the Health Heart uses the spin() function provided by the assignment to animate it with each time step. All objects are accessed through Obstacles.ts (a file in the folder scene).

Each object is initialized and categorized as either a power-up or an obstacle. Parameters of the all-encompassing class Obstacles are set, such as movementSpeed, spawnInterval etc. An update loop updates the position of each object, removing them from the scene if out of bounds (behind the playable character and the static camera's view). Additionally, within this function, we check whether there is a collision and set the global variable collided to true.

Obstacles	Powerups
objects/Balloon/Balloon objects/Floating Island/Island	objects/healthheart/Healthheart

2.4 Collisions. Collisions with the playable character are detected using a bounding box that is defined in the TypeScript files where the 3D .gltf objects are loaded into the scene. After computing a bounding box for each object upon initialization, the box's position is updated in each update loop to reflect the movement of the obstacles. In each Obstacles.ts update loop, we check Toothless's bounding box with active objects spawned by the obstacle manager. If there is an active collision, state variables are updated, and the main game loop (app.ts) will poll that and make corresponding updates to the game state (eg. health bar) as well as trigger Toothless collision animation.

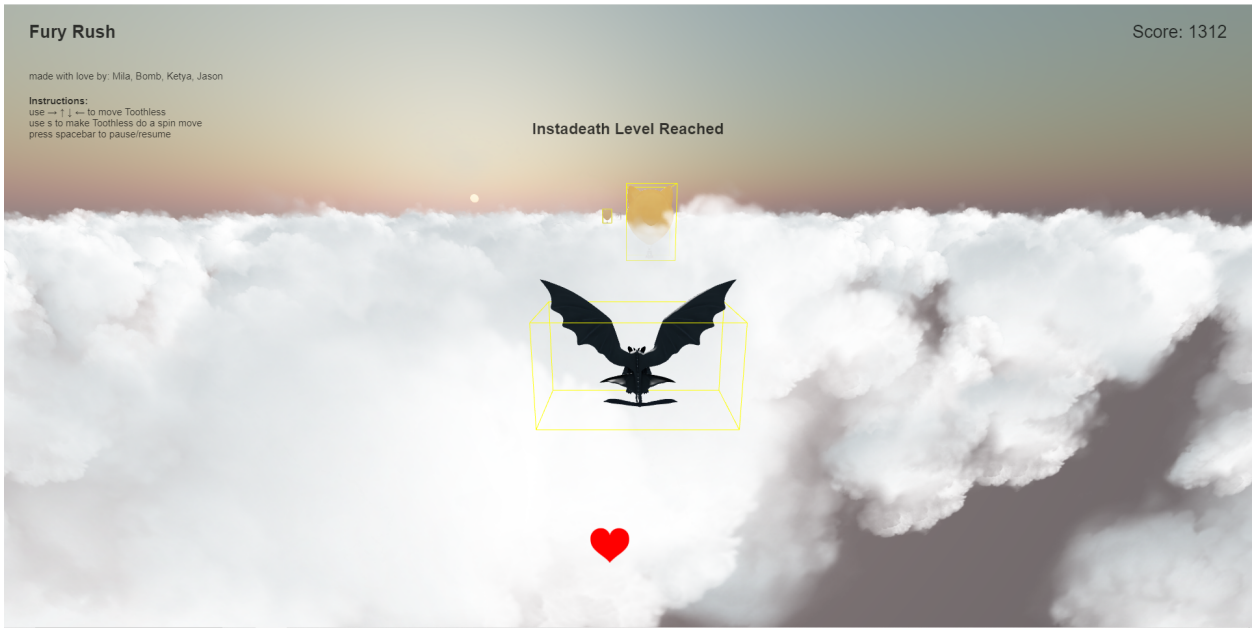

Figure 4: Collision Box Visualized

2.5 UI and Event Handling. App.ts handles all the event handling and game state logic. The event handler reacts to valid keyboard inputs. Directional keyboard inputs invoke methods from the scene file that move Toothless. Pressing the spacebar transitions the game between start - running - paused - game over. Depending on which state the game is in, the scene state is updated in the appropriate manner, and the game UI is updated in the appropriate manner as well. The UI is created by rendering the UI html & css (stored in istartscreen.html) over the Three.js canvas html. The javascript from app.ts dynamically changes the UI html based on the game state. One example of game state is the gametimer, which keeps track of how much running game time has elapsed since the game started. This would be used to calculate the score, which is then dynamically rendered on the screen. Another game state is the number of lives Toothless has. Depending on how many lives Toothless has, a lost heart could lead to game over or still allow for the game to continue. Additionally, app.ts imports the game's sounds and plays these sounds at the appropriate time (for instance, playing a "sparkle" sound when colliding with a heart). The sounds were free .wav files scavenged from the internet. The "Glider" project from Spring 2022 was helpful for figuring out the methodology of incorporating UI elements into our project. Additionally, it was helpful for figuring out how to incorporate Three.js audio functionality into our project.

3. Results

3.1 Technical Tests. We developed a working optimized endless-flying game with a user interface and 3D graphics. Assessing the performance of the game was mostly done

through qualitative means. Our initial test was to determine whether the game, even in the early stages of development, can run without lag on devices with various graphical capabilities. We did this by testing our code locally on multiple laptops, making sure that it is optimized and the computational power required is not exhaustive. From what we determined, even though lags occasionally happen, the game runs smoothly across multiple devices.

3.2 Gameplay. Our second test was to assess the playability of the game, and whether any noticeable bugs are present. This was done with extensive playing and testing limiting cases while doing so (edge movement, collisions, game speed, etc.) Thus far, the game is largely out of collision bugs (most collisions are accurately detected and the game responds accordingly).

Finally, we tested how the game is perceived by a wider audience. Asking our friends to beta test the game made us improve on things such as difficulty, movement stability, and responsiveness. Additionally, we asked questions regarding the graphics and design, as well as the interface. All of these responses helped us shape our game both visually and mechanically.

Video Demo: https://youtu.be/IlSDD8eFkho

4. Discussion and Conclusion

4.1 Difficulties. We had a lot of difficulty implementing our collision system. We initially used Three.Box3 and set the bounding box from model functions. But the gameplay result was not as smooth as we had hoped. As an example, the collision box generated was too small for the heart power-up and too big for the island obstacle. As a result, our beta testers felt like it was too difficult to collect power-ups and at times, they felt like they had avoided the obstacle yet the game still detected a collision and deducted a hitpoint. So instead, we opted to tweak the collision box manually until we reached a point where we felt our game was at a desired difficulty level.

Another difficulty we encountered was how laggy our game was. We first fixed this by preloading our assets into the cache and spawning objects from the cache instead of loading them from the asset files. Although this alleviated some issues, the game was still not running as smoothly as we had hoped. We proceeded to redesign our procedurally generated cloud in which instead of despawning the clouds when it flew past the player character and out of frame, we moved it to the front. This seems to largely solve our performance problems and the game now runs at an acceptable level of performance.

4.2 Further Development. Further, new mechanics can be added to the gameplay, such as a curved path, moving planes, falling meteors etc. A user interface that collects scores and allows players to customize their dragon can be a positive addition to the game. Moreover, leaderboards could encourage competition and enrich the gameplay. In terms of scenery, more different obstacles can be added, as well as power-ups that add a shield, speed up the score, allow the playable character to dash through a range of obstacles etc. Another point of future development is the addition of upwards movement which will make the game more challenging and interesting.

We implemented a spin feature (accessed by pressing s), which is currently just aesthetic. In the future that feature can be used to drill through objects and break them, or to activate a shield. Finally, a feature that can be made is a shooting-like mechanic where the player can target objects or meteors falling towards them in order to break it and not lose a life due to a collision.

5. Contributions.

Kritin Vongthongsri. Implemented scenic background and infinite terrain generation (sky, clouds). Created an Obstacle manager class responsible for managing spawning and physics of obstacles and power-ups. Reduced lag caching 3D models.

Chanketya Nop. Implemented player character movement and animation. Animation includes moving left and right, double move, spin move, and a blinking animation for when Toothless collides with an object. I also worked with Kritin to fix asset caching issues and implemented difficulty levels and the associated object spawning mechanism. Additionally, I worked with Mila to improve the collision system. This involves manually tweaking bounding boxes to fit better with the model and improving the collision logic to improve accuracy.

Jason Oh. Implemented game logic (e.g. keeping track of health, stopping the game when health reaches 0), handling player inputs, and designing plus dynamically rendering the UI elements of the game. Informed teammates regarding the scene level interface that needed to be implemented, for the top-level app code to work. My contribution is contained in app.ts and istartscreen.html.

Mila Bileska. Implemented the obstacle designs and animation of the power-up. Worked with Ketya to improve and optimize the collision system making. Created and optimized the obstacle files. Worked on the random objective spawning in Obstacles.ts, as well as SeedScene.ts. Created the fog and spawning effect of obstacles.

6. Works Cited

Inspirations

- Subway Surfers: https://subwaysurfers.com/
- Temple Run 2: https://imangistudios.com/thegames/temple-run-2/

Models and Animations

- Toothless: https://sketchfab.com/3d-models/toothless-test-animation-41304b55838b4f54b3b5db9576abcb51
- Health Heart: https://sketchfab.com/3d-models/healthheart-6a527144ffbc430681d8a886691adb00
- Island: https://sketchfab.com/3d-models/final-diorama-scene-f220b2b413c34b6784029ba25daf300f#download
- Balloon: https://sketchfab.com/3d-models/to-the-moon-doge-hot-air-balloon-ef9cbcdabdd5417c8eace6f84b4d2014
- Spaceship: https://sketchfab.com/3d-models/hull-spaceship-72ab5f7cbeb541c69f0ca286b2c0310b
- Lighting : https://discoverthreejs.com/book/first-steps/ambient-lighting/#:~:text=The%20AmbientLight%20is%20the%20cheapest,placed%20relative%20to%20the%20light.
- Sky Shaders: https://threejs.org/examples/webgl_shaders_sky.html

Audio

- Collision: https://pixabay.com/sound-effects/punch-140236/
- Flying: https://mixkit.co/free-sound-effects/wind/
- Background Music: https://www.youtube.com/watch?v=5t-oI7dPLto
- Wind: https://pixabay.com/sound-effects/howling-wind-73982/
- Power Up: https://pixabay.com/sound-effects/power-up-sparkle-1-177983/

Other References

- Clouds: https://www.youtube.com/watch?v=RYzBfAx6QTI&ab_channel=CoderNepal
- Midjourney: https://www.midjourney.com/home?callbackUrl=%2Fexplore