Falling Cat Adventures Written Report

Vivian Ha, Jiacheng Xu, Ajimelec Gonzalez
COS426 Fall 2024

Abstract: This project presents *Falling Cat Adventures*, an interactive 3D web-based game developed using Three.js, TypeScript, and React. This project is about a cat having a falling adventure descending to an island for many rounds, collecting halos as it falls through the sky, and also having a scenery of islands to view. Maneuvering carefully to collect as many halos as possible while avoiding the birds. The game features a unique falling mechanism where players control a cat character descending through a generated skyspace, collecting halos while avoiding hostile birds across multiple rounds of increasing difficulty. We implemented real-time smooth continuous movement, dynamic obstacle generation, and an innovative scoring system that rewards skilled maneuvering and strategic play.

Introduction: With simple controls and a clean design, the purpose of the game was to implement a fun yet progressively more challenging falling game. *Falling Cat Adventures* is inspired by *Legend of Zelda: Tears of the Kingdom*'s sky diving game and *Genshin Impact*'s gliding exam game. Our original idea was closer to *Genshin Impact*'s glider game, with the character using a glider to go through rings and the challenge being to get through all the rings before the time was up. After looking at *Legend of Zelda: Tears of the Kingdom*'s sky diving game, we decided to instead go for a top-down approach with the rings. As a result, our cat character falls through the sky and you must keep your character alive by going through the rings. In the reference games, you must hit all of the rings in order to pass the game. The diving/gliding challenges are specific challenges within each of the games. However, our game incorporates a health bar that continuously decreases. In our game, the purpose of the hoops is that for every hoop the cat character enters, the health will increase. The player does not need to go through all of the hoops, just enough hoops so that the health increases enough that the health bar does not fall to zero. Furthermore, we add obstacles (birds) that decrease your health and clouds to obscure the player's view to separate ourselves from the two reference games.

Approach: We decided to build the game using Typescript and ThreeJS. For the choice of typescript, we decided that it would be worth using as Typescript's more strict requirements, such as static typing, would incentivize us to write higher quality code that wouldn't fall prey to as many errors, and any errors that did come up would have to be fixed. It was also useful for debugging purposes.

ThreeJS was also appropriate for our needs, as it could handle a lot of the necessary functions related to rendering objects, handling lighting, etc. Additionally, because we all had familiarity with ThreeJS, it was a practical choice to stick with a framework we knew how to use, as opposed to learning an entirely new framework for a short-term project.

The most important piece of logic that we had to implement was collision detection. We took inspiration from a previous year's project (Glider 2022), which uses a raycaster to find collisions. We chose this for modularity purposes, and while it would be more computationally expensive, it was still feasible for the scope of our game.

Methodology:

Raytracer

For collision detection, it works by iterating through each vertex of a mesh, and from the vertex normal, sending out a ray. If that ray intersects something at a close enough distance, then we count it as an intersection. The original idea was that this could potentially make our game more modular for future features. For example, if we wanted to use characters besides the cat or obstacles besides the birds then this is mesh agnostic. If we were to try to implement intersections between say cylinders and spheres and try to do it very analytically, then it might be difficult to introduce different characters. In terms of the circumstances in which we thought this would work well, as long as there were only a few intersections to detect per update, this would be scalable. In practice, there might only be ~10 objects that could potentially be intersected with at a time. Because we knew our screen wasn't going to be crowded with excess objects, and we only cared about intersecting with the cat, then we believed this would be a feasible solution. Furthermore, trying to analytically solve for detections between various objects may not work as well practically speaking, as we may be more prone to mathematical errors. Additionally, we would have to account for things like rotations, and we may not have been able to assume axis alignment, as we have been in the past assignments.

Round Counting

The difficulty with implementing rounds was implementing how we would make the rounds more challenging as the game continued and making sure that the scene reset correctly. We originally were indecisive about whether or not to make an infinite game with no rounds and the user would play until they died. We decided on the rounds mainly because we wanted to make the game more challenging as the player progressed and this would be easier to do with the existence of rounds. Our RoundManager.ts handles the updating and resetting rounds, along with updating the round counter that exists on the screen. To end a round, we check the y position of the cat to see the distance from the ground, and if it is at a certain distance, the round ends. We also had to make sure the halos, health bar, birds, islands, and etc., all reset before a new round began.

Halos

The Halo system implementation was done using two classes one being Halo and the other HaloManager, providing clear separation between individual halo behavior and overall management. Each Halo combines a GLTF model for visuals with an invisible cylinder geometry for efficient collision detection, while the Halo Manager handles dynamic spawning and cleanup based on the cat's position. Rather than pre-generation all halos or using fixed patterns, we implemented a system that dynamically generates halos with controlled spacing (30 units) and randomized positions within a defined area, stopping generation near ground level (100 units buffer). This approach maintains consistent memory usage through efficient cleanup of passed halos while creating amazing gameplay through randomized positioning and sizes.

Health Bar

The decision to make the health bar continuously decrease was something that came early in the development process. Rather than just implementing a simple game where the halos had no effect (besides collecting a score), we believed that a health bar can add more interactiveness to the game. Now there is an incentive to go inside the halos and the user has to beat their previous score while fighting against the constantly decreasing health bar. Our implementation creates a health bar DOM element that allows the player to see the status of the health bar. The health bar continuously decreases at a certain rate and increases when the cat goes into the halo. The health decreases sharply when the cat hits a bird.

Birds

We took a long time to debate about the birds mainly because we also had thought about this idea when developing our collision method. The raycaster method we use for halo collisions enabled us to implement the collisions for birds. We had also considered doing other types of negative collisions beyond the birds but lacked the time for this. There are additionally two types of modes for the birds. One of them is the track halo mode, where the bird turns when it passes through the halo. Another mode is the track cat mode, where the bird starts following the cat or turns away from the cat. These different modes allow for different types of birds' actions to happen.

Plane (Ground) & Textures

Originally, we attempted to load the texture onto a plane that was at y = -3000 (our furthest ground distance). However, since the plane was so far away from the camera, we could not see the plane no matter how big we made the plane. We additionally attempted to put the texture in the background, but only one texture was allowed for the background. So we were not able to include the NORM, DISP, OCC, and ROUGH texture maps. As a result, with only the COLOR texture loaded, the background was just one solid color. With this, we had to innovate a method of keeping the plane closer to the cat and moving the plane downwards as the cat moved down. Because each round increases the depth of the game, we had to consider this when implementing this feature. So in this game, we have a real ground level (the cat must be 100 above this ground level to end the game). We also have a plane with a water texture that starts at a certain distance from the cat. The water texture plane will move downwards as the cat moves down so that the player will always be able to see the water texture plane. This simulates a fake ground, which solves the issue of the plane being too far away when we put it at the real ground level.

Ground & Background Islands

For the islands, we opted to split the implementation into separate classes rather than using a unified approach. This allowed for specialized handling of the landing zone and decorative islands, simplifying the independent scaling and positioning of the islands. The background islands implemented a dynamic generation strategy based on the cat's position, featuring alternating model types (low-poly and camping islands) with randomized positioning and scale variation. This is more memory efficient than pre-generating a fixed set of islands, as it allows for clean up of distant islands while maintaining the possibility of endless falling. In addition, for the ground island, we had to put the position on top of the water texture plane and move it downwards with the plane. This is so the player will always be able to see the island.

Menus

The menu system implementation leverages React with TypeScript, chosen over alternatives like pure HTML/CSS/JS. This decision provides significant advantages in component reusability and type safety while maintaining advantages in component reusability and type safety while maintaining a clean separation from the core game logic. We structured the menu as a single page with distinct sections for About, Controls, and Objective information, rather than implementing a multi-page system. This approach simplifies navigation and makes all information immediately accessible to players while maintaining a clear visual hierarchy that could potentially adapt well to mobile devices.

Game Controls

For game control, we implemented a key state tracking system using a boolean map, which proved more effective than alternatives like event-based or velocity-based movement. This system enables smooth continuous movement and efficiently handles multiple simultaneous inputs while remaining easily extensible for additional controls. The movement system uses normalized vectors with boundary clamping, ensuring consistent movement speed and smooth diagonal motion while maintaining position constraints. We included features like arrow key movement, strict boundary limits, and delta time-based updates. The implementation also includes proper clean-up through a disposal method and maintains a clear separation between movement logic and rendering.

What we didn't implement

We are happy with the current features of the game, as its current state is aesthetically pleasing and the mechanics make it playable and enjoyable. While it would've been nice to have some extra features, we think we did well in prioritizing the important key features, and these were simply ones we weren't able to get around to.

We had some stretch goals that we wanted to implement, but that we were unable to get around to. For example, we originally wanted to have some hoops move around, but this would be tricky to do because it might make the player's sense of depth perception worse, and it would present a technical problem; currently, our birds have 2 tracking modes: either tracking the cat or tracking a halo. And if the halo moved around during halo-tracking mode, we would've had to redo some of the halo-tracking code, which as of now assumes that halos don't move around.

We wanted to add other difficulty features, such as health decreasing faster per round and adjusting the probability of birds spawning. However, the health decreasing faster per round wasn't a strict priority, and we would've had to do much more rigorous testing in order to determine the maximum rate of health decrease to be "fair" to the player. We had a technical difficulty with adjusting the probability of birds spawning that we were unable to debug, where if the probability was set to below 1, no birds would spawn.

We also wanted to eventually add different character options, but this was also a lower priority, and we anticipate that if we were to continue development, this may not be too difficult.

Lastly, we wanted to have the character be able to accelerate/decelerate, but we were unable to find animations that would reflect this in a realistic way (we would've liked to be able to deploy a parachute), and we didn't have enough expertise to learn how to make animations ourselves.

How we measure success:

We set out a set of MVP (Minimal Viable Product) goals and a set of lower/higher priority stretch goals. The MVP goals included having a cat that could fall and being able to spawn the halos that would heal the cat. The higher priority stretch goals were more related to the aesthetics of the game (such as the islands that fill out the blank space and give a better sense of depth perception), as well as extra mechanics such as the birds that would act as obstacles, and clouds that would visually obstruct the player. An important stretch goal we were able to achieve is giving the birds interesting "attack" patterns; the birds come in 2 modes: cat-tracking mode and halo-tracking mode. As the name suggests, the former tries to match the cat's position (on the plane that it is generated on), while the latter oscillates around an associated halo to obstruct the cat. Also, they sometimes randomly flip in the opposite of their intended direction so as to not be too predictable. We got to some of the lower priority goals as well, such as adding fog to the game or adding a landing island, but we weren't able to get to 100% of our more complex stretch features; regardless, we believe that we've made a good game that is fun to play and we are happy with our results.

Primarily, we played this game ourselves as well as with the help of other friends. We wanted to make sure that the game played smoothly, and was intuitive for players without prior explanation or experience.

We conducted experiments to tune some important variables. For example, we tried various falling speeds, multiple hitbox shapes, and ring sizes. We would test one variable at a time, adjusting incrementally, until we were happy with how it felt for us as the player, and then we would lock that in. With this type of testing, we were able to get the sizes/speeds of our important objects correct, as well as things like generating an appropriate amount of clouds and islands so that it wasn't too visually cluttered. Additionally, we were able to make it so that birds get faster per round in a way that was fair to the player but still interesting, by testing a variety of different functions of speed based on round number. We settled on an exponential function to start slow, and gradually ramp up the speed of the enemy birds.

Discussion:

Overall, is the approach we took promising?

The overall approach we took was promising the use of React, Typescript, and Three.js was a good approach since it keeps it simple and easier to make changes to the game. Using just three javascript libraries instead of using other libraries would have made it difficult to handle debugging the code, being able to connect the website pages like the game menu and game over the menu and the game scene would have made it more difficult to connect them with just using simple HTML, CSS, and Javascript. Also for Three.js, it helped with the 3D rendering and physics of the game which is the core.

The raycaster-based collision detection system offered excellent modularity and flexibility for future feature additions. This approach proved particularly suitable given the relatively low number of simultaneous collision checks required (~10 objects).

The progressive difficult system through round-based gameplay successfully created an engaging experience that balanced the challenge.

The health mechanics with the halo collection create an interesting risk-reward dynamic that differentiates the game from its inspirations (Zelda: TOTK and Genshin Impact).

What different approach or variant of this approach is better?

1. **Physics engine Integration**: Implementing a dedicated physics engine like Ammo.js or Cannon.js could have provided more sophisticated collision detection and physic interactions, potentially reducing computational overhead compared to the raycaster approach.
2. **State Management**: Implementing a more robust state management system (e.g. Redux) could have simplified the handling of game state and made it easier to add features like save states or multiple games.

Ethical concerns

1. Ownership of Ideas

   Our project started with much inspiration from the sky diving game from *Legend of Zelda: Tears of the Kingdom* and *Genshin Impact*'s gliding challenge game. When reviewing the ethical concerns of a project, the question of what is the line between taking inspiration and stealing an idea arises. In this case, we reference section 1.5 of the ACM code of ethics: Respect the work required to produce new ideas, inventions, creative works, and computing artifacts. In this project, we mention specifically what areas of the games we took inspiration from and credited these works. We also do not directly copy the idea, as we add much of our own ideas and implementations. Our game takes inspiration, but is different from the gameplay in *Legend of Zelda: Tears of the Kingdom* and *Genshin Impact*.

2. Fairness & Discrimination

   One thing that always should be done is checking the accessibility of a project. In this case, we specifically reference section 1.4 of the ACM code of ethics: Be fair and take action not to discriminate. While we were developing our project, we used tools like color contrast checkers in order to determine whether or not our text colors would pass the WCAG compliance requirements. However, we do acknowledge that much more can be done to improve the accessibility of our game. These features can include: adjustable text size, custom controls, color blindness modes, high contrast modes, and etc. The text size of the menu could be adjusted along with the round and halo counter. Additionally, we could also allow for the adjustment of the health bar size. We believe accounting for these features in future development could mitigate issues, which addresses "Failure to design for inclusiveness and accessibility may constitute unfair discrimination" (ACM). If this game was to continue to develop, we would need to improve our accessibility in terms of controls and settings.

What follow-up work should be done next?

1. **Performance Optimization**:
    a. Reduce the computational overhead of the collision detection.
    b. Any other performance in our code as needed.
2. **Gameplay Enhancements**:
    a. Add varying enemy types with different behavior patterns.
    b. Add different character selections so that players can choose different characters.
    c. Implement different game modes (e.g. time trial, challenge modes, sandbox).
    d. Develop a power-up system
    e. Improve how the game becomes difficult (e.g. moving halos)
3. **Technical Improvements**:
    a. Refactor our code to improve readability and efficiency
    b. Maybe add support for mobile devices and touch controls
    c. Implement save states and persistent high scores

What did we learn by doing this project?

Through the development of *Falling Cat Adventure*s, our team gained extensive knowledge and experience across multiple domains. From a technical perspective, we deepened our understanding of 3D web development, particularly in implementing complex collision detection systems and their associated tradeoffs. Working with TypeScript enhanced our appreciation for type safety and its role in preventing runtime errors. The project also significantly strengthened our project management capabilities, as we learned to effectively balance our MVP requirements with stretch goals while maintaining a clear vision for the game. Our systematic approach to testing, which involved carefully calibrating game feel and difficulty scaling, proved invaluable in creating an engaging player experience.

The game design aspects of the project provided equally valuable lessons. We faced the challenge of creating a difficult progression that remained both challenging and fair to players, requiring careful tuning and iteration. Drawing inspiration from games like *Legend of Zelda: Tears of the Kingdom* and *Genshin Impact* while developing our own unique mechanics taught us how to effectively adapt and innovate upon existing game concepts. Perhaps most importantly, we gained practical insight into how technical constraints can shape game design decisions, leading to creative solutions that worked within our technological framework while still delivering an engaging player experience.

Working within the constraints of web technologies while striving to create a polished 3D game helped us understand the delicate balance between technical optimization and user experience. This experience has equipped us with the practical skills and insights that will prove valuable in future development projects.

Conclusion:

Based on our MVP goals and stretch goals, we believe that we have effectively made a good game that has covered all of our MVP goals and a good amount of important stretch goals that allow for interesting gameplay that gets harder over time, and fun aesthetics that makes the game visually appealing to the player.

In terms of the next steps, we would like to be able to make more interesting difficulty features so that the core gameplay loop is more varied, such as by adding different types of enemies besides the hostile (but cute) pigeons. We would also like to add different gameplay modes, instead of having the player just try to get higher and higher scores. Additionally, it would be nice to allow the player to have more customizations, such as switching out the character model. While the game in its current state is fun and playable, there are some optimizations we could probably do as well to make the game run a bit more smoothly as well, given that we use a raycaster.

References:

Inspiration from:

- *Genshin Impact*: Gliding License Challenge
- *Legend of Zelda: Tears of the Kingdom*: Sky Diving Challenge
- Glider (COS426 Spring '22 Final Project): Raytracing for collisions
  - Harvey Wang, Sophie Li, Richard Cheng, Andy Wang

Code used:

- Lightsaber Training Dojo (COS426 Spring '22 Final Project): Load material
  - Victor Chu, Arti Schmidt

Resources:

- Halo by Poly by Google [CC-BY] via Poly Pizza
- Cat by jeremy [CC-BY] via Poly Pizza
- Water 002 Texture from 3Dtextures.me
- Pigeon by Quaternius
- Secret Camping spot by Jake Blakeley [CC-BY] via Poly Pizza
- Island by Poly by Google [CC-BY] via Poly Pizza
- Cloud by Quaternius
- Cat Head icon by Icons8

Music:

- Background music: https://freesound.org/people/ZHR%C3%98/sounds/584430/
- Bird hit sound: https://pixabay.com/sound-effects/search/damage/?pagi=2
  - Punch 2
- Heal sound: https://pixabay.com/sound-effects/search/retro%20coin%20collect/
  - Coin collect retro 8 bit