COS426 Final Project: Cat Dash

Xi Chen
Department of Computer Science
Princeton University
xc11@princeton.edu

Abstract

Endless runner games have become increasingly popular in recent years. Inspired by the successful work of Temple Run [1] and Subway Surfers [2], I implemented a similar game called Cat Dash, where players control a cat to avoid obstacles and earn rewards. To enrich the game interface, scenery are randomly generated and spawned along sides of the cat's moving tracks. Optimizations have been done to guarantee the game's playability and compatibility, and enhance algorithm efficiency. Based on the comprehensive experiments and tests, the game runs buglessly and smoothly with different screen sizes and browsers.

1 Introduction

Recent years have witnessed the increasing popularity of endless runner games, and successful work such as Temple Run [1] and Subway Surfers [2] have achieved billions of downloads globally. Their success largely lie on their features of simple operations, fast-paced actions and quick feedback. However, most of them charge fees or have annoying advertisements, which takes away much fun from purely enjoying games. Therefore, I decided to create a similar game with all the above advantages but without any commercial factors to bring pure joy to players in their leisure time.

Since I am a cat lover, I use a cat as the player character. The game's setup is pretty simple: a cat sitting on a mop flying in the air, and players have to control the cat to switch among left, middle and right tracks to avoid randomly-generated obstacles and collect flowers (rewards). Some obstacles, for example spaceships and flying islands, are designed to move horizontally or vertically to increase difficulty. The cat's flying speed and density of obstacles increase as the game goes on. To make the game interface more interesting, scenery such as buildings, rainbows, cities and ships are randomly generated and spawned along the sides of the tracks. There are also a star and a sun rotating in the background to enrich the scene. Players' scores are calculated based on the elapsed time and the number of flowers collected, which is shown at the left top coner of the game interface.

To ensure the game works out smoothly, I implemented playability checks, non-overlapping controls and screen compatibility adjustments. Additionally, I optimized the random generation algorithm to improve the memory utilization. These optimizations are discussed in details in Section 2.4.

In the following report, I use the terms 'model' and '3D model' for different meanings: by 'model', I refer to the model component in Model-View-Controller architecture, while by '3D model' I refer to polygon meshes or glTF models of visible objects in the view.

2 Methodoloty

The game is constructed under Model-View-Controller (MVC) architecture (Figure 1). The following subsections introduce each of the three components in order.

2.1 Model

Model is the core component in the architecture, and it defines the game's state, data and updating rules. In this game, models are divided into two groups: a scene model and multiple object models.

The Scene model extends Three.js Scene class. It defines the game's world space and manages all object models added to it. To be more specific, it keeps track of current stage of the game (before start/on progress/end), objects that need to be updated, current flying track the cat is on (left/middle/right), and obstacle/reward/scenery generation probabilities etc.

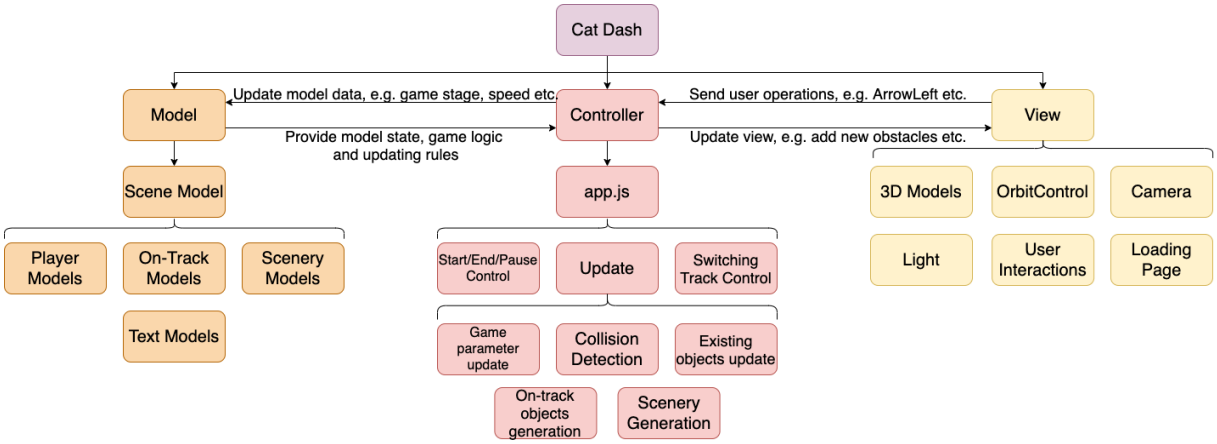

Figure 1: Cat Dash's high-level architecture, which consists of the major components of model, controller and view.

On the other hand, the object models extend Three.js Group class, and record states of single objects in the game. They are further divided into the following four categories:

1. Player models. They define the player character of the game, and consist of a cat model and a mop model. The models are initialized with $moveUp()$ and $moveDown()$ functions to control the player character moving up and down periodically to make the flying scene look more natural. Additionally, these models come with a $switchTrack()$ function, which defines the player character's left and right movement when receiving users' instructions. The $moveUp(), moveDown()$ and $switchTrack()$ functions are implemented using Tween.js.

2. On-track models. As the name suggests, this group of models define objects placed on the flying tracks, and include a reward model and several obstacle models. The obstacle models of spaceship and flying island are initialized with Tween.js horizontal and vertical movement functions respectively.

3. Scenery models. This category consists of a flying track model and several roadside scenery models, where the roadside scenery models include star, sun, rainbow, city, ship and building.

4. Text models. This category defines in game text, for example 'Click space to start' and 'Game Over'.

Except for Text models, all the remaining three categories of models come with an *update()* function, which define their updating actions in each time stamp. Additionally, player models and on-track models have bounding boxes, which are used in collision detection.

2.2 View

The camera, light and OrbigControl are setup using Three.js.

Most of the 3D models except for in game texts and flying tracks are fetched from Google Poly, and loaded using Three.js GLTFLoader. The in game texts are generated using Three.js TextGeometry, and the flying tracks are generated with Three.js PlaneBufferGeometry, and rendered with a shader from ShaderToy. The sources of the fetched glTF 3D models and the shader are cited in corresponding files.

To receive users' inputs, I bind 'keyDown' and 'click' eventListeners to the window object. The received instructions are passed by to the game's controller, which then respond accordingly.

Since loading 3D models takes time, I implemented a simple loading page to avoid the view being shown before it is complete.

2.3 Controller

The controller component handles incoming requests and processes the game's business logic. It has the following three main functions:

1. Start/End/Pause Control. This game has four stages, which are Before Start, On Progress, Pause, and End. The controller switches the

game's stages based on user inputs and game logic.

When the game is on Before Start stage, and receives a space key input from users, the controller switches the stage to On Progress, and adjusts related variables/objects accordingly. For example, it removes the starting texts ('Cat Dash' title and 'Press Space to Start' instruction) from the view, starts playing background music, and starts the clock to keep track of playing time.

During the game, if users press space keys, the controller switches the game stage between On Progress and Pause. In Pause stage, the audio and clock are stopped and update function is disabled.

In the End stage, if users click the screen, the game stage is changed back to Before Start again, and the views and related variables such as elapsed time, number of flowers collected are reset to initial values to properly restart the game.

2. Switch track control. When the game is on progress and receives left/right arrow key inputs from users, the controller invokes the $switchTrack()$ function, which checks player character's current track, determines whether left/right movement is valid under current state, and if it is then calls player models' $switchTrack()$ function to complete left/right movement.

3. Update. It is the core function of the controller, and is executed every time stamp when the game is on progress. Basically, in each update the controller does the following 5 things:

   (a) Update game parameters. The controller adjusts the flying speed and the obstacle/reward/scenery generation probabilities based on the elapsed time.

   (b) Collision Detection. If the player character collides with an obstacle, the controller ends the game and adds related text objects ('Game Over', 'Click to Restart') to the view. On the other hand, if the player character collides with a reward, the controller increases the score.

   (c) Update existing objects. All existing objects' $update()$ functions are invoked and update the objects' states correspondingly. If an object is 100 distance behind the player character, it is marked as 'expired', removed from the view and added to available item list for future use (see Section 2.4 for detailed explanation).

   (d) Generate new on-track objects. The controller randomly spawns new obstacles or rewards 250 distance in front of the player character. The randomness is in two distinct aspects: the spawned object is randomly chosen from all on-track object types, and the spawned track is randomly chosen from the three tracks. The generation frequency is controlled by the generation probability. There is a slight difference in spawning obstacles and rewards: if the controller is going to spawn an obstacle, it needs to do playbility check (Section 2.4) before adding the obstacle to the view.

   (e) Generate new scenery objects. This step is similar to the above step of generating new on-track objects.

2.4 Optimizations

As stated in the Introduction section, I made some optimization to enhance the game's performance, and they are explained below:

1. Optimization on memory utilization. Initially, I implemented the game with a naive random generation algorithm, that each time the controller decided to spawn a new obstacle/reward/scene, it created a new instance of the corresponding class. However, later I found that such implementation had very bad memory utilization performance, as memory occupied by 'expired' objects was not promptly released, while new objects were continuously created. Being aware of this problem, I modified the implementation to reuse 'expired' instances, and only create new instances when there are no more available existing instances. The method is implemented with dictionaries (key-value pairs), where the keys are classes' names, and the values are lists of available instances. Experiments suggest that the game runs much more smoothly with the new implementation.

2. Playbility check. In my initial implementation, obstacles were randomly spawned on

one of the three tracks, which leaded to the problem that sometimes all three tracks were blocked and the game could not continue. Realizing this problem, I adopt playability check before adding new obstacles, which compares the x coordinate (x-axis is along the tracks) of the new obstacle to the x coordinates of the two most recently-spawned obstacles in the other two tracks, and if the new obstacle is within 8 distance of both neighbors, the controller throws the new obstacle and does not spawn new object at this update step. An example is shown on Figure 2, where Obstacle 1 is the most recently-spawned obstacle on track 1, and Obstacle 2 is the most recently-spawned obstacle on track 3, the yellow rectangle is the area that is within 8 distance in front of Obstacle 1, and the purple rectangle is such area of Obstacle 2. New obstacles are not allowed on track 2 in the overlapped area of the yellow and purple rectangles, but they can be spawned after the purple rectangle ends.

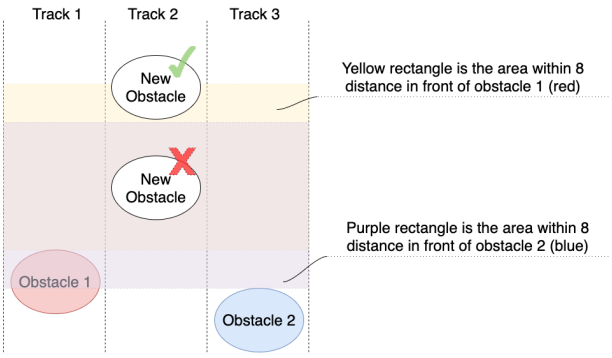

Figure 2: An example of playability control. Object 1 is the most recently-spawned obstacle on the first track, and Obstacle 2 is the most recently-spawned obstacle on the third track. New obstacles are not allowed on the second track in the overlapped area of the yellow and purple rectangles, but they can be spawned after the purple rectangle ends.

3. Non-overlapping controls. Given the scenery generation algorithm (Section 2.3), one problem is that sometimes multiple scenery objects collide with each other due to small intervals. To resolve this problem, I keep a list of the most recently-spawned scenery object for each side, and every time before spawning a new scenery object (except for rainbow, which does not have collision problem), the controller checks the positions of the new object and the previous object on the same track to see whether there is enough interval to avoid collision. Each object's required interval are measured manually.

   Before deciding on this method, I also tried to use bounding box to detect collisions, but it turned out there were loading problems which could not be resolved.

4. Screen compatibility adjustments. Since different players can have different screen sizes and I want the game to be rendered properly with any size, I adjust 3D models' scales by current window's width and height. The adjustment equation is:

$$scale = standard\ scale * \frac{current\ window\ size}{standard\ window\ size}$$

   where standard scale and standard window size are the scale and window size on my development screen.

2.5 Other Implementation Decisions

Except for the algorithm optimizations discussed in Section 2.4, I also want to briefly introduce several design decisions I made during implementation.

1. Randomly spawn scenery instead of fully covering the track sides. When I was designing the roadside scenery, I faced two options: one was to fully cover the track sides with scenery objects, which would enrich the game interface and make the view look pretty, and the other was to randomly spawn scenery objects, which would leave blank space on roadsides. I experimented with both implementations, and found that though the first option gave better visual effect, it harmed the game performance and caused UI lags on my computer. Based on the experiments, I finally decided on the randomly-spawning design to sacrifice the scene diversity for better game experience.

   Despite of my final decision, I think the first option could actually outperform the second one on high configuration computers.

2. Do not provide detailed guidelines. The game provides basic guidelines on the loading page, such as 'use left and right arrow keys to control the cat', and 'collect flowers', but it does not give very detailed instructions such as 'do not collide with trees, islands, spaceships',

'cloud is safe and collidable' etc. I actually received some complains from my friends about the vague instructions, but after consideration I still decided to let users to explore the different properties of different objects on their own, so that they can have a feeling of exploration and maybe some surprise when they did not 'die' after colliding with a cloud.

3 Results

Success are measured in three aspects: correctness, smoothness and level of fun.

To ensure the correctness of the program, I did unit tests of every function and module, and operated system tests by playing the game multiple times in different browsers (Safar, Chrom and FireFox).

The game's smoothness are measured by duration and frequency of UI lags. Based on my experiments, there are no significant lags or latency in the game.

To evaluate the game's attractiveness, I invited seven friends to play and rate the game. Overall, the game received an average rate of 8.7 out of 10, and the median rate was 9. The main reason for point deduction is that there are not much diversity on obstacles and roadside scenery.

Given the above evaluations, I consider the game successful especially when it was constructed under such limited time. However, I also admit that it can be further optimized in both the designs and algorithms.

4 Discussion

Overall, I am satisfied with the project's outcome and implementation. The outcome game runs bugslessly and smoothly with various screen sizes and browsers, and the implementation enables easy extensions to new game objects.

Through this project, I leaned to use the powerful library Three.js, and deepened my understanding of core computer graphics concepts, such as meshes, shaders and 3D transformations etc. Additionally, I got a basic sense of game development, and familized myself with the MVC architecture.

Below I propose several possible future directions, according to collected feedback and my implementation notes:

1. Optimize the scenery generation algorithm. As discussed in Section 2.5, I was forced to choose the random spawning method instead of the other nicer option due to the limited computational resources of my computer. However, I believe the algorithm could be further optimized to work more efficiently, and hence enabling the game to fully cover the roadside with scenery. Possible optimization methods include preloading and accurate updating control.

2. Enrich the obstacle and scenery types. Based on the feedback, I plan to add more obstacle and scenery models to the game to create more diverse and attractive views. This idea could be further extended to building several different scenes for players to choose, for example under water scene, volcano scene, snowfield scene etc.

3. Introduce new interaction modes. Currently, the users can only control the cat to move either left or right. In the future, I plan to introduce more interaction modes such as jump, power up, acceleration, deceleration, curved flying tracks etc. to make the game more interesting.

4. Support multi-players mode. It is natural that some players might want to play this game with their friends. Therefore, I feel it is necessary to extend this game to support multiple players to play together on different devices.

5 Conclusion

In this project, I constructed an endless runner game called Cat Dash, which allows players to control a cat to avoid obstacles and earn rewards. Several optimizations have been done to guarantee the game's playbility, compatibility and efficiency. Comprehensive tests have been conducted to ensure the correctness of the program. In the future, I plan to further optimize the algorithms to enhance efficiency, and introduce more scenery and interaction modes to increase the game' s attractiveness. Additionally, I plan to extend this game to support multiple players so that players can compete with their friends in the game.

6 Contributions

The project was all done by myself.

References

[1] Imangistudios.com. 2020. *Imangi Studios*. [online] Available at: ”https://www.imangistudios.com/” [Accessed 12 May 2020].

[2] Subway Surfers. 2020. *Official Homepage*. [online] Available at: ”https://subwaysurfers.com/” [Accessed 12 May 2020]