Flappy Nerds

COS 426 Spring 2016 Final Project Report

Naphat Sanguansin ‘16, Xiaohua Liang ‘16, Xuling (Richard) Hu ‘16, Eddie Chen ‘16

1. Introduction

Our team was inspired by the popular iPhone game Flappy Bird, and wanted to write our own Flappy Bird but add new features using features and techniques learned in this class. In the end, we created a web game in Javascript that has autopilot function, a chasing UFO with a hybrid first person and third person view, building on top of Three.js.

While the project is designed as a lightweight web game that would benefit both avid and casual gamers, it has great value to game and graphics designers on how to implement a fluid game in Three.js. For future students taking COS 426, the project could serve as a great starting point as well, with its use of animation, mesh manipulation, texture mapping, ray casting, collision detection and scene rendering and many more COS 426-relevant features. Additionally, it uses the latest version of Three.js, which is not a straightforward copy-and-paste from an assignment starter code.

Many previous attempts exist in replicating Flappy Bird, but none incorporates different perspectives - Github.com/jennyzw/flappy3d comes closest as the programmer used Three.js as well, but the game itself is very simplistic with a fixed view, without any extra flairs like lights, other flying objects and etc.

We started our project on top of an example code from Three.js (the link to the example is http://threejs.org/examples/#webgl_camera), taking the hummingbird mesh from the same source used by Assignment 4: Project ROME.

2. Methodology

2.1 First person and third person views with split screen

We use instances of Three.js’s PerspectiveCamera. The cameras are responsible for their positions relative to the bird. The cameras are then grouped into a parent 3D object known as a camera rig. The camera rig’s position is updated to match the bird’s. We are using the Three.js’s WebGLRenderer, with half the screen being rendered with the third-person camera and the other half with the first-person camera.

2.2 Autopilot function

The autopilot aims to keep the bird at some baseline y position (note that in our coordinate system, x points right, y points up, and z points out of the screen). Whenever the bird falls below the baseline, we trigger a jump. The baseline y position is taken to be the lowest position possible to pass the next obstacle, plus 30. 30 is a number found experimentally that handles cases where the bird is falling too quickly (we are only sampling up to 60 times per second). The UFO's autopilot does something similar, except it is not constrained to the jumping motion and so can update its y position directly. As a direct test of the autopilot feature, the bird did not die after 500 obstacles, which is likely much more than an average human could achieve.

2.3 User Interface (UI)

In order to pass the control of difficulty level to users and make the game experience flexible and adaptive, we have linked the speed of obstacle generation and the interval distance to the difficulty level set in *Config.ts*. Before the start of each game, a difficulty-selection window will show up on the screen, prompting the player to choose the appropriate difficulty level for the next game. Once the difficulty level is chosen, the player cannot change it in the current game. However, if the player decides to restart the game (either after the bird has died or in the middle of the game when he/she feels the current difficulty level is not fun enough), he/she will have the chance to re-select difficulty at the beginning of the next game.

During the game, the player has the option to pause the current game (by clicking the ***Pause*** button on the screen or pressing ***P*** on the keyboard). This will pause current progress, and the player can resume by pressing ***P*** again. If the player switches browser window or wants to check out the helper menu, the game will be paused automatically as well.

If the user is unclear about the keyboard commands for the game, he/she can check out the helper menu by clicking the ***?*** button on the screen or pressing ***?*** on the keyboard. This will bring up the helper menu, which contains useful keyboard commands. Pressing ***?*** again will bring the player back to the game (which is paused and the player needs to press ***P*** again to resume playing).

The user also has the option to restart the game at any time during the current game by clicking the ***Restart*** button on the screen or pressing ***R*** on the keyboard. This will reset the game (including all scores) and bring the player back to the starting point where he/she needs to select the difficulty level.

2.4 Textures on objects

The bird mesh is obtained from Project ROME, and it comes with a predefined texture (face colors and vertex colors). Therefore, we simply applied the given texture onto the mesh by computing the vertex normals and morph normals and using smooth shading (Phong model) in the material property of the bird mesh. The end-result is a green hummingbird with a red beak.

For the obstacles, we would like to use a wooden texture so that the obstacles look more like trees in the forest. As such, we found a picture of wooden texture from Google and applied it to the obstacles. We have also added the emissive property to the material of the obstacles, and the emissive color is set to be dark brown (hex value *0x331a00*) for better visualization when light is shone on the obstacles.

Interestingly, for the top and bottom planes, we have decided to use a water texture, and this makes the whole scene looks like it is deep in the water. We think this adds more fun to the play. Together with the UFO in the scene, such mixture of seemingly unrelated objects produces a comical effect that hopefully will make our players love the game more.

2.5 Lighting

To create a more interesting and realistic lighting effect, we would like the light source to come from the bird object, which means as the bird moves through the obstacles, the brightness of the background (top and bottom planes as well as the obstacles) will change depending on how close they are to the bird. This provides the player with some additional visual effects that make gaming more exciting.

In order to achieve this effect, we have placed 4 point light sources (THREE.PointLight) around the bird object (left, right, front and back). Note that no spotlight is placed inside the bird mesh because this would make the bird too bright for its texture to be seen. The light color is chosen to be a very pale yellow (almost white) and its hex value is *0xffffee*. The intensity of the light sources and their decay factors are adjusted to make the lighting effect easily visualizable but not overly bright. The point light sources move together with the bird object (i.e. their positions are updated together with the bird) so that the bird constantly remains the focus of the light sources. The obstacles in our game use MeshLambertMaterial while the top and bottom planes use MeshPhongMaterial. This means they can properly reflect the light shone on them.

In addition, to emphasize the importance of the UFO in this game, 2 point light sources are also placed around the UFO object. The light color is also *0xffffee* but they have slightly less intensity than the point lights surrounding the bird. This is to reduce the overall lighting intensity of the whole scene.

With these light sources, our scene displays changing light intensities on the obstacles and the top and bottom planes, as if the bird is flying through a dark forest (under the sea!) and lighting up its surroundings while flying.

2.6 Moving planes (top and bottom)

The planes on the top and bottom are actually moving at the same speed as the bird in the x direction. We move the texture in the negative x direction at the same speed to make it look like the plane is not moving but is actually infinite.

2.7 UFO

In order to make the game more fun and innovative, we added in a UFO that follows the bird and is controllable by either the same player or another player. The UFO can shoot as well, and the score goes down if the UFO wastes its ammunition but increases if it hits the bird. The UFO also carries the first-person view camera point that is displayed on the right half of the screen.

2.8 Bird animation

The bird mesh that we have used already contains built-in animation. However, we would like the bird to flap only when the spacebar is pressed. This closely resembles the actual “flappy bird” game and is also more realistic physically (since the bird only flaps wings when it is going up or down).

In order to achieve this effect, we have a separate counter in the bird object to control the flapping time and duration. When the spacebar is pressed, a flag will be set to true and the counter starts counting up. At the same time, the bird animation will be activated to display the flapping movement. Once the counter reaches its maximum (set to be 15 time intervals in our game), the flag will be set to false and the counter will be reset to 0. The bird animation will also be deactivated.

One thing to note is that if the spacebar is pressed while the bird is flapping, we will keep the flag to be true but reset the counter to 0. This essentially prolongs the time duration that the bird flaps and is consistent with player expectation when the spacebar is pressed multiple times.

2.9 Deterministic, pseudo-random number generation

A custom linear feedback shift register (lfsr) method is implemented to produce a deterministic (values determined by initial seeds), pseudo-random number generator that’s used in generating obstacles and levels. Because the lfsr package from npm only gives the user access to one random bit for every shift, so for a random float with reasonable precision we need to shift many times and it is too slow - our custom solution provides a reasonably random-looking number generator that is also fast enough to keep up with smooth rendering.

2.10 Particle based volumetric background

We decided to use particles to create our background, giving the illusion that we are in space surrounded by stars. To do this efficiently, we define a data structure known as a star block. Each star block is responsible for creating particles in a defined [minX, maxX] volume. At any given time, the scene has an array of star blocks. As the camera moves forward, a new star block is created accordingly. When a star block goes out of range, it is removed from the scene. The code for particles generation is taken from the Three.js example code (http://threejs.org/examples/#webgl_camera).

2.11 Real time level generation

Our level generator is built on top of the deterministic, pseudo-random number generator with a fixed seed. A new obstacle is generated every (interval + range * x) seconds, where x is a number between -1 and 1. Each obstacle will have the height (base + range * x) for the opening, where x is again in [-1, 1]. Once the height is known, the opening is given a random y position that is appropriate for its height. We chose to add randomness to the interval, height, and position of the obstacles to make the game a little harder. We also chose to have our generator be deterministic so that the players can train for the game.

Solvability is a requirement in our level generator. We guarantee solvability by making the intervals between obstacles sufficiently large. An alternative implementation that would allow us to make intervals smaller is to first come up with a solution (a sequence of taps from the user, generated by the pseudo-random number generator) and then put obstacles in. However, this is more complicated, and even at our hard difficulty, the game was already hard enough with a small enough interval that we did not have to worry about this.

2.12 Collision detection

There are two collision models in our game. First, we need to detect collision between the bird and obstacles. To do this, we use **ray casting**. More specifically, we loop over the vertices that make up the bird mesh and fire off a ray from the origin of the bird in that direction. If the ray hits anything that is closer than the vertex, then we have a collision. This is not a perfect collision model, as it will not work very well for obstacles that are much smaller than a face on the bird mesh. However, this is never the case in our game, so we chose to go with this simple model. We used Three.js's built-in Raycaster for this purpose. This model has a small bug, which is if the bird moves fast enough, we may not detect collision with a plane, which has 0 thickness. To handle this, we simply check if the bird's y position is in range.

The other kind of collision we need to handle is the collision between the bullets from the UFO and the bird. In this case, the bullets could be too small to be caught by our ray casting method, and we also needed to detect when the bullets hit other obstacles, too. Therefore, we tried using the same ray casting method, but with the bullets themselves as the source of the rays. This turns out to be too slow, and the slowdown comes from having the bird as one of the collidable meshes. Therefore, we simplified this by having a special, simplified algorithm for bullet-bird detection: simply take the distance between their midpoints, and compare that the radii of their bounding spheres. This is crude, but it works well enough (because they are both physically small objects), and the performance benefited from it. For bullets-obstacles detection, we cast rays from the bullets as before.

We could have gone with a more complete solution for our collision algorithm. In fact, we could have taken the physics library built on top of Three.js, Physijs, which supports all the physics we need. However, we decided to take the challenge of writing our own algorithms, and the ones we came up with are good enough for our purpose (just not good enough for a general-purpose physics engine).

3. Results

There are primarily two ways we measure success: the overall quality of graphic rendering of the game, and the smoothness of the game playing experience. Under rendering quality, we wanted to make sure the overall graphics look smooth and natural, with a consistent frame rate (around 60 fps).

Under game playing experience, we want to make sure controls are intuitive for users and the help panel is available. In general, the game is designed to be light, fun and addictive. We have achieved considerable success, as demonstrated by the fact that all four of us enjoy playing this game.

4. Discussion

Our primary work is building game physics on top of Three.js. For the purposes of education, this approach is fine. If we were building an actual game, we would have gone with a pre-built physics engine to avoid the unnecessary complexities. In terms of follow-up work, our code follows good programming practices in defining clear interfaces, allowing easy extensions to the game if we wanted (and if it were to benefit the game experience). A possible extension is to add new kinds of obstacles, even moving ones. Our autopilot should already be able to handle new obstacles. Another possible extension is to make this game work on mobile, which should not be too much work considering that Three.js already works decently well on mobile. We could also add background music to the game, with other acoustic effects when the bird is hit or clears an obstacle.

To make the game more fun for a pair, we plan to support controlling UFO's motions as well: so one player can control the bird while the other is controlling the UFO trying to shoot it.

Moreover, we are looking to add adaptive screen-size handling, so that the game adapts to the new size when one tries to resize the window during the gameplay.

5. Conclusion

Overall, we made a very playable, interactive and innovative Flappy Bird in Three.js with features learnt from COS 426. This game has blended what we learnt in image processing, modeling, rendering and animation, and is really a piece of work that we ourselves enjoy playing very much. From the design and implementation processes, we have reinforced our knowledge in computer graphics and have learnt many new elements in game design as well. Of course, there are still a number of features that could be further tuned, and we have made this project open-source so that any interested party could continue our work and make this an even better game.

The code for this project is currently hosted at https://github.com/naphatkrit/flappy-nerds, and the project itself is hosted at http://flappy-nerds.herokuapp.com and open to the public. We welcome feedback on our game, and encourage interested individuals/teams to provide more innovative ideas to be implemented.

Last but not least, we would like to thank the course staff, including Prof. Adam Finkelstein, Huiwen Chang, Nora Willett and Linguang Zhang, for their dedication to this class. It is indeed an enjoyable semester of playing with pixels and meshes. Wish you all the best!

6. Reference

Three.js - Javascript 3D library, http://threejs.org/

Project ROME, www.ro.me/tech/