Abstract

This paper details the methods and tools used to create our immersive space experience titled, “Princeton 2020: A Space Odyssey”. It was created with popular JavaScript frameworks three.js and cannon.js, alongside various assets that were both created and found in open source libraries. Our goal was to create a game with a focus on a stylized arcade environment and good performance that centered around a space theme. After taking inspiration from previous space flight games, we scoped our project out to focusing just on object collection and object collision with smooth first person controls. We created an MVP that consisted of both a start screen and a three.js scene with basic fuel charge support. After that initial testing, we went back and revamped our controls while adding in support for other objects, such as asteroids and invincibility upgrades. All throughout we continued to refine the visuals of the projects, such as adding in a space skybox and a particle system. This all went towards increasing immersion in our product. Our conclusion was that while fixing the controls helped add to the playability of our product, the lack of laser/enemy support weakened our immersion aspect.

Project Links

Source Code: https://github.com/teferiemmanuel/2020-A-Princeton-Space-Odyssey

Demo: https://teferiemmanuel.github.io/2020-A-Princeton-Space-Odyssey/

Introduction

“Princeton 2020: A Space Odyssey” is a space flight game where the player must navigate an ongoing asteroid field while also collecting enough fuel to survive. Players can hope to join an exciting, flight-centered space adventure. Pilot a spaceship flying through outer space as you attempt to warp home by collecting fuel charges. The primary objective will be achieved by the player collecting fuel charges as they spawn while avoiding flying asteroids or picking up powerups to become invulnerable to asteroid collisions.

Goal

The goal of our project “Princeton 2020: A Space Odyssey” was to create a highly stylized immersive experience with a focus on smooth movement and high performance in a 3D space using three.js. We wanted to take the skills acquired in the former half of COS426 and create a short enjoyable experience. While others would be able to enjoy our project as a mostly simple game, we hoped to create a learning experience that would encapsulate the concepts we were taught in class and give us the intrinsic value of a satisfyingly finished product in the end.

Previous Work

Over the past few decades plenty of arcade style space flight games have emerged.

- **Space Invaders (1978)** --- A space flight arcade game where shooting aliens is the primary objective. Space flight is de-emphasized and limited to the X dimension.
- **Galaga (1981)** --- Another space flight arcade game where the primary goal is shooting alien enemies. The player only moves their character left or right on a constantly forward-moving space terrain.
- **Star Wars video games (1983-2019)** --- These games allude to the Star Wars universe and have more well developed renderings and complex missions. Player movement is not limited to the X direction. Shooting enemy spaceships is the primary way of winning the game, but navigation has a notable role.
- **Star Fox (several made from 1993-2017)** --- Highly stylized on-rails space shooter, with a choice between first person and third person camera view. Focuses on space combat through set paths and occasional aerial dogfights.
- **A COS 426 Hall of Fame project called “Death Star assault” (2019)** --- Based on Star Wars. Its third-person perspective and shooting down enemy turrets with lasers is the primary objective.

In general, today there are many space shooting games in apps and gaming sites alike.

These related works are survival-based space flight games where the main emphasis is for the player to shoot down enemies to further advance and win the game. This paradigm is effective in that it has attracted users and brought about several well-known and loved games. These games also have clear levels and an end once certain objectives are achieved. Navigation improved in the later games and the first person perspective was a powerful way of engaging the players. The graphics and overall quality of image renderings improved in the later games, but still maintained an arcade look. We were curious to know if a slightly alternative paradigm could also be effective. Perhaps weaknesses with this paradigm could be that a game with a clear end to it gets old as soon as the player beats the game. Furthermore, it’s not clear how much value shooting adds to these space flight games. Maybe it reduces winning the game down to pressing a button or key repeatedly instead of focusing on the wonder of

space flight and the challenge of space navigation. This is what led us to explore creating a survival-based, space-flight game with entire emphasis on navigation--- away from asteroids in an endless shower and towards fuel charges. We hope this contribution could bring more perspective on possibilities in space arcade style games.

Approach

In order to create this arcade-style immersive experience, our group settled on the theme of space exploration; we chose this because we thought that this theme would give us the liberty of being creative with our environment while emulating cool player movement patterns. The approach we tried was ultimately similar to previous COS426 projects before us. We strove to build on top of the three.js framework to quickly build a 3D online experience with cannon.js serving as the framework in charge of dictating most physical interactions in this world.

With our frameworks chosen, we were able to create a high-level plan for how we would explore creating this 3D world. Firstly, for user experience purposes, there would be an initial start-up screen complete with instructions on how to play the game, acknowledgements, and a “Start” button. After reading and clicking the start “Start” button, the user would be brought into a Game Scene. This game scene would be populated with both objects that the user needed to interact with and objects that the user needed to avoid or otherwise face the consequences of an undesired loss. Because it was decided that the theme of this project would be space exploration, the general Game Scene was mostly space inspired; the user themselves would be in the cockpit of a spaceship, the object to interact with would be a fuel charge, and the object to avoid would be pesky asteroids. A holdover from the arcade inspiration would be additional invincibility upgrades that would aid the user in their ultimate objective: continue retrieving as many fuel charges before your fuel tank runs out!

It would be in that very Game Scene that a majority of our actual graphics related work would be. The approach to creating the start-up screen would be rooted more in HTML/CSS and JS work. However, the Game Scene required more use of graphics knowledge to accomplish. Here, the approach was to create a parent Scene object that would keep track of all happenings in the game. In addition to rendering, its job would be to spawn the locations of all its’ children objects, those being all the different asteroid, fuel, and power up objects at different vectorized positions and velocities. Our approach would also have this Game Scene object to be responsible for rendering a space skybox for immersion and randomly positioned “space dust” particles to better communicate the user’s movement to themselves. In terms of various physical interactions, defined as the player’s collision with any of the Game Scene’s children objects or asteroids against asteroids, cannon.js would be the chosen approach to calculating the physics between these interactors; players would gain fuel when colliding with a fuel charge, gain temporary invincibility when colliding against a powerup, and lose fuel when hitting an Asteroid. Custom controls would be programmed towards the purposes of improving the space exploration experience, and to reach the goal of performance, objects were made as buffer geometries and regularly disposed of if they left a defined game bounded area.

This approach's limit was mostly the computing power of the client running the game. We were cognizant that people's experiences would ultimately be dictated by the machine they were playing it on, so another part of this approach was to fine tune certain parameters (game bound size, max number of meshes spawned at a single time, the spawn rate) by as many machines as were available to us. We believed that our project would work well under those circumstances because it is the limitation one accepts when working with a JavaScript framework that computes all the necessary calculations on the client side. Despite this, we hoped that the stylized meshes and custom controls would still run smooth enough on a large population of users. We hoped to limit the effects of this by as much as possible.

Methodology

The project includes two primary user interfaces that the viewer navigates: The Menus and the Game Scene.

Menus

The primary user experience goes through a flow where the player reaches a starting splash screen (first screen of the Menus) containing important information about the Controls and Instructions along with a Start Button. Clicking the Start Button brings the player into the primary Game Scene which will be described in detail in a later section. After completing a round, the player is presented with a minimal Game Over menu which details the number of collected fuel charges and survival time (in seconds) during the course of the game.

There were two major options for implementing the Menus. They could be created as two-dimensional text objects in three.js or as HTML elements that could be hidden and displayed at appropriate times based on certain game actions signalled by Javascript variables.

Three.js objects would provide a seamless transition from the base starter code we were given as we would not have to configure webpack to serve up HTML and CSS (in addition to any frameworks we may have wanted to include). However, the three.js text objects could use up valuable resources on the Chrome V8 engine when running the application which we would want to save for rendering in-game objects.

Using HTML elements to render the Menus would be more comfortable for many team members due to our background and familiarity with basic web development. It would also strain the browser engine much less since it would not use Javascript at all and is little more than a text overlay on the screen. The main disadvantage of using HTML elements to serve up Menus is the initial engineering effort required to configure Webpack to serve an HTML page with the correct assets alongside the three.js canvas.

We eventually settled on implementing the menus using HTML and configuring Webpack to serve the HTML “index” page which included the three.js canvas in its body. We controlled it such that when starting the application, the user was presented with only a single div element while all other elements (including the three.js canvas) were hidden. Clicking the Start Button would hide the correct div and show the canvas and once the player’s game completed, the canvas would be hidden again and the Game Over menu would be displayed.

In the end, this implementation of the Menus allowed for a simplified user experience with a clear flow of movement between screens while being extremely easy to implement and iterate upon while saving on precious browser resources.

Game Scene

The Game Scene is entered after selecting the Start Button. We'll step through each of the components required to create the primary Game Scene.

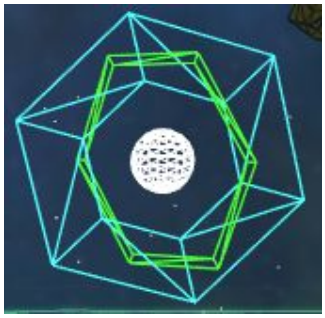
HUD

The HUD is once again a case of overlaying HTML elements on top of the three.js canvas. This allows for easy tweaking during the development process as well as easy display of Javascript variables. The HUD consists of a transparent cockpit image that the player views the scene through from a first-person perspective, two fuel and time indicators, and an alert message. Once again, this was all done in HTML because it was so lightweight and easy to work with. An alert message in the top-middle of the screen appears when the user collides with an asteroid, powerup, or fuel charge and informs them of the effect it has. On the left side of the screen, a time fuel collection indicator helps the player keep track of the number of fuel charges (which increase the amount of gas left in the player's tank) they've collected. On the right side of the screen is a fuel indicator that tells the user how many seconds of fuel they have left before the game is over. A handy fuel charge bar is also included to convey this information.

All HUD elements were created with the belief that the player should be able to view the environment without having their vision impeded by the user interface. The elements are minimal and provide enough information to be informative to any new player.

Environment

The primary backdrop for the in-game environment is provided by a “skybox” formed by a set of six images that texture a cube (three.js’ CubeTextureLoader) surrounding the player. Additional “star” particles were generated randomly to dot the sky and provide visual landmarks for the player since the skybox textures were too sparse outside of the sun spot on one of the panels. The main part of the environment were the three types of objects a player could collect/collide with to a variety of results.



First, there is the fuel charge. Steering into a fuel charge allows the player to fill up their gas tank by 3 seconds. This fuel was created by using three.js to create a group that represented the various aspects of this object. The overall mesh was a combination of 3 mesh’s put together: an outer cyan ring, an inner color changing ring, and an orb in

the middle. The rings were both RingBufferGeometries (as opposed to a regular RingGeometry for performance reasons) that were created at different sizes and rotated slowly at different angles. The middle orb was a SphereBufferGeometry rendered with 10 width and height segments—it wasn't necessary for the sphere geometry to have more segments than that because the sci-fi space theme didn't call for it. All of these meshes did not have any material or texture on them, but instead were left as wireframes because that best encapsulated the arcade style theme we were trying to emulate. Appearances finished, the last step was giving fuel charge the capability to interact with its environment. A cannon.js sphere shape was created with a tight radius that only just fit the fuel object inside of it. This shape was added to a cannon.js body which was then added to canon.js world, whose responsibility was keeping track of all physical interactions. This body was given no mass, since the interpretation of this object was mostly an ethereal concept.

The other positive object that the user can pick up is the invincibility powerup. Steering into a powerup allows the player to be invincible to potentially damaging asteroids for 5 seconds. The construction of this object was analogous to the fuel object. It consisted of an outer geometry with a geometry in the middle. The outer geometry was a TorusBufferGeometry that was generated with random colors while the inner object was a TetrahedronBufferGeometry. The entire material was wireframe again to go with the arcade theme. The cannon.js body that was created also similarly had no mass.

The final object is the asteroid. Colliding with a flying asteroid ruptures the player's gas tank and causes them to lose 5 seconds worth of fuel. Asteroids geometries were borrowed from user "capratchet" on Google Poly and physics were implemented with the cannon.js library. While the object also came with material, this was changed in favor of matching the aesthetic better; a black wireframe was overlaid

on the actual object to create an outline effect, and the asteroids were given solid colors. Asteroids were spawned with a random velocity unlike the other objects which are spawned with no velocity. In addition to this, they were spawned in a variety of different colors, scales and angular velocities to make the game's world even more immersive. The cannon.js body that was added to this mesh had a mass of 1 so that asteroids that collided into each other would respond appropriately.

Note that all objects mentioned above are initially spawned with a single instance in the world and are slowly spawned at random around the player's position as the player continues playing the game. We chose to not spawn all possible objects at once as this allowed players to "cheese" the system by quickly picking up an incredible amount of fuel charges from the beginning and make the rest of their session incredibly easy as their fuel tank would likely be maxed. Allowing for spawns over time made the experience more dynamic and unique with every new game session.

Including these three types of objects provided a strong base of replayable gameplay for the user that was both rewarding and challenging (with the use of flying asteroids).

Controls

The controls were inspired by three.js implementations of first-person controls (based on FirstPersonControls and PointerLockControls which are used for first-person 3D universes) and flying controls (based on FlyControls which is used for fly modes when transforming a camera around 3D space). The provided controls from FlyControls allowed for multiple modes of movement that were quite complex for the purposes of our game.

FlyControls provided the ability to move forward, back, left, and right with WASD, fly up and down with R and F, rotate the camera with Q and E, and pitch and yaw using the arrow keys. The sheer amount of controls provided by FlyControls is extremely overwhelming for a casual gamer and is difficult to memorize while in-game. Hence, we simplified the given controls to use a limited set of keys while using more intuitive input devices (namely, a mouse).

Indeed, the mouse movement was inspired by the PointerLockControls implementation, which allowed for a controlled and steady move of the camera. On the other hand, the mouse movement in the FlyControls implementation was incredibly sensitive and would constantly move based on the position of the mouse in relation to the screen dimensions. This was undesirable and caused nausea for some of our group members. We also experimented with combining the arrow keys with movement keys in the up, down, left, and right directions, as well as allowing for rotation with the arrow keys. This was ultimately unsuccessful and deemed too complex for players to understand. Indeed, the movement in the up, down, left, and right directions was rather unwieldy to work with and also unrealistic. The rotation of the arrow keys also interfered with the rotation vector and inner quaternion of the camera, which in some cases, would cause the mouse controls to be reversed due to the unsynchronized orientations. Hence, we decided to instead use just the mouse for observing the surrounding environment and removed the key bindings for movement and rotation keys.

As we mentioned above, we decided on letting the user change the position of the camera using the mouse so that it acts as a “free look” function in 360 degrees. We debated briefly on letting the user choose to use either the mouse for moving the camera or the arrow keys for rotation, but ultimately decided that the mouse scheme was more natural. We then added in key bindings to let the user’s “ship” automatically move forward at a constant rate that could be accelerated or braked with momentum using Spacebar and Shift, respectively. Holding down the spacebar would slowly build up momentum for the player while holding down shift would quickly slow the player down to a halt. Letting go of these keys would then move the player back to the original equilibrium speed. A novel barrel roll using Q or E triggers a “Star Fox”-style barrel roll for the player to experience.

In all, the controls help the player traverse the 3D space with ease while keeping things as simple as possible.

Results

We used both quantitative and qualitative metrics to measure the success of our product and various stages in development. Much of the feedback on the MVP was actually used to improve the end product. After creating our minimal viable product, we decided to let users try out the game and give feedback on improvement. We cared most about whether this game had the potential to be a popular game. The following questions were asked to participants:

1. How long did you play the game?
2. What additional features would get you to play the game again or play for longer?
3. How would you rate the user experience of this game?
4. How much did you enjoy playing the game?
5. How easy was it to play this game?
6. Would you recommend this game to a friend?
7. How would you rate the overall look of the game?
8. Comments or suggestions on how to improve?

Survey participants were anonymous COS 426 students on Piazza. The game was very difficult for many. Comments on MVP mentioned that instructions were needed, the controls system was too many buttons and confusing, and that the game was near impossible. Interestingly, other comments said/implied that the game was too easy. Also, some comments were more suggestive and brought up the idea of upgrades, a fuel gauge, closer fuel spawns, and a bounding box preventing the user from going too deep into space. The overall look and style was well received.

Another conclusion is that people did not yet see the value in changing the old paradigm of lasers and space battle. One participant mentioned that “It'd be nice if there was more of a space battle, either with enemies or just asteroids.”

Discussion

Our approach was to build an immersive arcade-style space game in a 3D space using three.js and cannon.js for basic physics. This proved to be the correct approach as it allowed us the freedom to include easily configurable objects in our world that could be stylized for the purposes of our style.

While making the game we learned that designing tight controls in a 3D space, even when given a base to start from, was very difficult to get working smoothly. This was even before including any sort of momentum or realistic movement patterns beyond an “auto-fly” movement. Having to fine-tune for any user that could be using a touchpad or a mouse made even things like mouse sensitivity difficult to configure correctly to make them feel tight.

Outside of the scope of the implementation details, we discovered that Javascript libraries are of varying quality. While heavily developed libraries like three.js are incredibly high quality and have great documentation, others like cannon.js contained problematic APIs that didn’t always work. For instance, a method to remove a physics body object from the universe was buggy and had been buggy for multiple years (since development stopped). Having to work around this shortcoming proved to be difficult.

One vital part of the development process we learned along the way is that Javascript’s *requestAnimationFrame*() method that is used to power the animations in three.js is tied to monitor refresh rate. Users with high refresh rate monitors literally experience the game at a speed higher than normal and as such find the game much more easy. This is an issue with the way we’re utilizing the Javascript engine and reworking this would be a project on its own.

We also learned surveying players that playtested our game proved to be very difficult as there’d be little agreement on things such as difficulty and smoothness. Some players found early iterations of our game incredibly easy while others found it difficult. In all, we took the commonly mentioned issues from reviews of our MVP and refined those pain points instead of focusing on varying opinions.

Conclusion

We successfully accomplished the goal we set out to meet by making an incredibly smooth space-themed experience with tight controls. Initially, we were very unsatisfied with the controls we’d experienced in any three.js games we’d found and wanted to make sure ours was at least better in some way. Our work on the controls eventually paid off and we made a smooth experience in terms of player experience and game performance. In addition, our ability to apply ideas from the course to the game showed from our applications of stylization through our experimentation with bloom and other design features.

One of the issues we’d have to address in the future are the parameters that determine the amount of objects spawned in the game and the rate at which these objects are spawned. Fine-tuning these parameters to be perfect would take much more

playtesting and iteration on finding the perfect balance. In general, it was difficult for us to balance having a well-populated game area (in terms of objects) and having a game that any user could run smoothly at 60 frames per second. We found that including enough asteroids to make the game area feel populated tanked the frames per second on even the most powerful machines.

The next steps we would take to improve this game would be to generally perform more feature work that we had originally set out in our stretch goals.

One of the largest areas of work we could venture into would be to design enemies that pose greater danger to the player by attempting to ram into the player or shoot at the player's ship. This would obviously require lots of complex AI programming for the enemy ships if we wanted it to be halfway decent rather than a basic "seek player position and shoot in general direction" scheme.

Another feature we wanted to implement was including different difficulty levels for players to experience if they felt the base difficulty was too overwhelming. This would be done with increasing the rate of fuel depletion, increased asteroid spawns, limited fuel spawns, etc. To provide a better user experience, perhaps an in-game leaderboard would also be appropriate so that players wouldn't have as much of a siloed experience.

Another large feature we'd want to include is possibly allowing the player to have some sort of laser gun for shooting asteroids and allowing them to be destroyed into many smaller pieces. This would require much more complex physics and modelling to determine how they would break apart and to simulate each new piece and would likely be a project on its own.

Finally, the last major change we'd want to make is to improve performance on all client machines. The game runs at a fairly low framerate for machines with integrated graphics cards due to the proliferation of objects interacting in the scenes. It also suffers from a drawback stemming from JavaScript's *requestAnimationFrame* method which ties frame updates to monitor refresh rate. As such, players with higher refresh rate monitors would experience the game at a faster pace than others. Fixing both these restrictions would greatly improve the user experience.

Contributions

Jessica Edouard: Contributed to original assets, designed and programmed scene objects, and implemented physics.

Emmanuel Teferi: Worked on controls and animated turns, spawning asteroids and upgrades, user feedback acquisition and analysis/evaluation of result, related work research, and parameter tuning.

Justin Tran: Acted as tech lead. Programmed menus and handled resetting the in-game environment upon replay as well as rethinking in-game controls. Touched many areas of the codebase except for object models and game bounds.

Henry Wang: Bootstrapped the initial control implementation and worked heavily on creating tight controls for in-game use. Worked on loading audio for background music and barrel roll sound effects.

Works Cited

- three.js reference API
- cannon.js reference API
- Skybox generated by Space-3D
- Sound effects
- Starwing Font
- Cockpit image asset
- Theme music
- Asteroid Mesh