The Drive

Parker Connelly, Ryan Gibbons, Kenny Poor, Christopher Speed

COS 426: Computer Graphics

Fall 2022

ABSTRACT

Our project, titled The Drive, is a driving simulator in which the player makes their way up a mountain while attempting to outrun (or rather, out-climb) a lurking particulate fog. We initially proposed an “urban hell” setting where the terrain would be randomly generated without end, but ultimately opted to instead create a smaller custom map given our time constraint. This choice sacrificed long-term replayability, with the tradeoff of more consistent and intentional gameplay with fewer bugs. As for the game’s atmosphere, we focused much of our efforts on developing features to further the gloomy feeling that we were aiming for - compelling our use of dark colors, the choice of having faraway objects fade away into the background, and the urgency caused by the rise and increasing density of the smoke. The result is a moody, off-road race toward the mountain’s summit - without any promise that the fog won’t follow you all the way to the top.

The starting menu screen of the game

INTRODUCTION

The goal of our final project was to utilize the existing web graphics and physics libraries of three.js and cannon-es.js as a baseline to create our own driving simulator game. The focus was not one of an infinite-runner, third-person arcade game, but rather a challenging first-person ascent up a mysterious mountain. This concept would ideally appeal to those who enjoy racing games as well as those who prefer single-player, closed-form narrative adventures, as it seeks to incorporate both into an atmospheric yet fun challenge to play.

We were familiar with examples of games that focused heavily on graphics and visual detail to connect a scene to some emotion or feeling, and our intention was to recreate this in our project. Games such as slowroads.io, A Short Hike, and Iron Lung provided us with a general feel for the game mechanics and a set of possible approaches to cultivating a consistent style for the in-game experience.

As for our approach, we veered away from attempting to develop our own well-functioning physics engine, gearing more toward world-creation and achieving graphical effects despite the genre's typical reliance on fun and consistent (if not realistic) physics. Instead, we used the aforementioned cannon-es.js package to generate our physics and collisions, achieving our goals for how the car would handle through the careful placement of spherical wheels on our box-car. Likewise, we relied heavily on the existing framework of three.js to create and organize our world and its aesthetic. While we did rely on these external libraries as a basis for the project, their use allowed us to focus on aspects of the game that were more exciting to us. This included custom meshes for the world and objects built in blender, a carefully-tuned fog that extended existing particle systems with self-interacting physics, and an engaging user experience complete with a starting menu, minimap, and sound effects among other features. We thought this approach would work well under the constraints that served as the base requirements for the project - a single-player, lightweight, and hopefully fun game that could run smoothly on the web. We think it should work well under these circumstances as it promises a new and unsettling experience for the user, in contrast to attempting to clone the tried-and-true arcade game formula.

METHODOLOGY

The fundamental components of our project are the player-controlled object (the car) and the environment with which that object can interact (the road and the mountain). We also needed a mechanism by which the player could lose the game. In order to provide a gameplay motivation other than simply not falling off, we implemented a rising fog-like particle plane that rises up over time, chasing the player as they scale the mountain. Multiple additional features were added to the game for sake of improved gameplay experiences (such as sound).

In total, out of the optional features, we implemented these 8:

- Texture Mapping
- Sound
- Particle System
- HUD
- Level Of Detail Rendering
- Frustum Culling
- Custom GLSL Shading
- Alternate View

THE CAR, INPUT, AND CAMERA

The car is made from four spherical wheels and a box body. It uses the rigid-vehicle object in which forces and steering can be applied to the wheel. Since tri-mesh

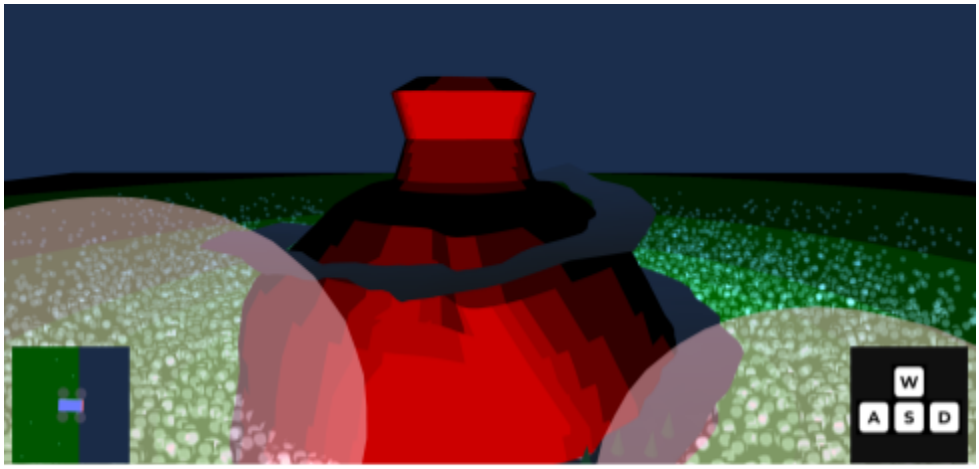
The input system listens for ‘keyup’ and ‘keydown’ and uses a key value map to keep track of which keys must be currently pressed down. This allows for detecting multiple keys being pressed down. By keeping track of the order opposite keys are pressed (up and down or left and right) we can properly handle conflicting key presses.

The camera panning does not use pre-built controls. Given the relative coordinates of the mouse on the screen, it calculates the targetLookAt vector in the local coordinates of the car. After translating to world coordinates every frame it linearly interpolates its current lookAt vector towards the targetLookAt vector in order to give the camera a smooth feel.

SHADING

In the initial stages of the project, we made use of the THREE.JS MeshToonMaterial to give an approximate feel for the general atmosphere and style. However, in the later stages we decided to implement custom shading as one of our additional features, and so a slightly more jagged cel-shading

style was chosen. From our implementation, we pass in several custom uniforms from our Javascript environment, making use of the position of the existing scene light and a default color chosen for the particular mesh being shaded. In order to achieve the banded cel-shading effect, we calculate the dot product of the normalized light vector and the normal of a given vertex, and then scale the default color based on this dot product, giving the harsher quantized cuts, rather than a smooth gradient (which was the desired effect).



An alternate view of the mountain's shading

THE ENVIRONMENT

As we became more well-versed in using the cannon-es physics library, it became clear that we would need to make use of the provided Trimesh physics body shape in order to have more complicated environments. In order to preserve modularity and ease of editing (in addition to potential performance improvements), the road, mountain, and other parts of the scene were modeled in Blender, with THREE.js materials being assigned once the models were loaded. In order to keep the physics world consistent with the rendered models, we created a custom "Road" object that managed the data for both physics and visual properties.

THE FOG

Our implementation of the fog element of our game was twofold, with the first part being a straightforward tuning of the FogExp2 class to create a vision falloff over distance. The latter and far more significant aspect to the fog was the particulate fog surrounding the map. To implement this, we created a particle system using the Points class which we populated with randomly generated particles all at the same y-height and with a random initial velocity. These points had a fog particle texture applied to them, giving them their appearance.

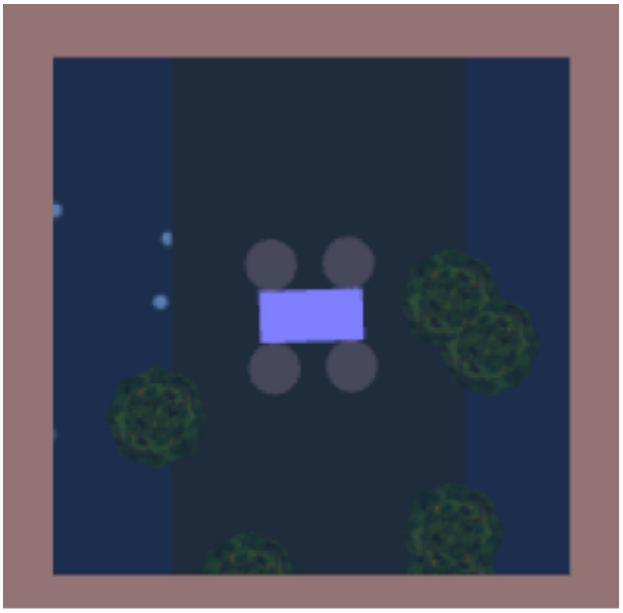
At every time step of the animation, an updateParticleSystem() function would be called to calculate the new locations for each particle. This function based the new positions on the current velocities of each particle, adjusting

the next velocity values based on gravitational interactions between randomly selected pairs of particles. Boundary conditions were handled by having particles bounce off the arbitrarily-decided bounding box of the world with a slight addition of randomness to their velocity, and by having particles that had been gravitated into the center of the mountain be randomly spawned at a new position. This was a similar approach to the fountain example discussed in class, without having to deal with the potentially expensive actions of deleting and creating new particles. Finally, the y-coordinate of each particle was slightly increased at each time step, creating the rising effect that we desired.

On the note of collisions, we handled this manually using a y-coordinate check on the vehicle rather than attempting to catch an interaction between the vehicle and each particle. This allowed us to perform the check quickly and add a bit of padding (similar to the EPS padding from our assignments) to make the game feel more fair, allowing the user to observe the fog overtaking the car in the event that they lost the game.

SCREEN ELEMENTS

The minimap, in the bottom left corner of the screen, is an additional view of the main scene. The camera used to render the minimap is an orthographic camera that is positioned above the car, looking straight down. The location and size of the minimap on the screen is controlled via the `renderer.setScissor()` and `renderer.setViewport()` methods. This lets us render the same scene with the orthographic camera and have it be displayed on a small portion of the game window, rather than the entire game window.



The minimap view. Here, there are trees nearby and fog particles (in white) underneath the road

The HUD, located in the bottom right corner of the screen, displays which directional inputs are currently pressed. Like the minimap, the HUD is a scene that is rendered to a small subsection of the game window. Unlike the minimap, the HUD uses a different scene than the rest of the game. The HUD scene consists of a plane with 4 smaller planes laid on top of it. Each of these 4 planes corresponds to one of the ‘WASD’ keys, and the planes are laid out in a similar manner to these keys on a typical

keyboard. The material of the planes is either the default material, which has a texture of a white png of that letter, or it is the “pressed” material, which has a texture of a black png of that letter. These materials are constantly updated based on booleans in the inputControl object that keep track of which directional keys are currently being pressed.

The HUD, currently indicating that ‘W’ and ‘D’ are being pressed

This HUD feature could have alternatively been implemented by having the small key planes in the main scene, positioned perpendicular to the camera and placed and updated such that they stay in the same place on the game window. This would have had the advantage of having a transparent background (our current implementation has the key HUD within a dark gray square that blocks a portion of the game window). However, the task of ensuring the scale and orientation would be correct every frame and ensuring the planes would not jitter or move around on the screen seemed too much given the very mild downside of a non-transparent background for the alternate scene implementation we chose to use.

ADDITIONAL FEATURES

The background music was composed using downloaded files from freesound.org. The basic Three.js method of adding sound to a scene was sufficient for our intended purposes, and so we used the Audio, AudioLoader, and AudioListener objects to load, play, and pause the background music. The inputControls object detects relevant keypresses and controls the logic for starting and stopping audio sounds.

Both texture mapping and Level-Of-Detail-Rendering were used to make the trees look realistic but be rendered efficiently. Each Tree object is at some world location, and at that location there are two models at the same spot: a low-poly tree and a high-poly tree. On any given frame, only one of

Nearby trees have a higher-poly model and have textures applied, but trees in the distance have a lower vertex count and have no special texture applied.

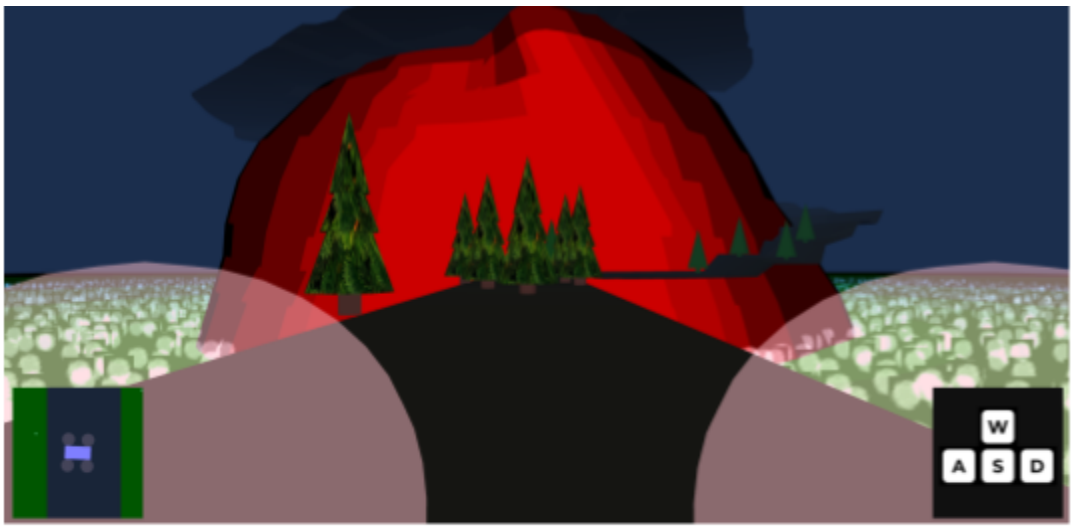
these models has the .visible parameter set to true, so only one of them will be included in the Three.js render pipeline. Calling the update() method of a Tree object and passing in a position will change which model is the visible one depending on the distance between the argument position and the Tree object's position. If the position is larger than some predetermined bound, the low-poly tree is visible; otherwise, the high-poly tree is visible.

The texture mapping was done with the built-in Three.js TextureLoader object. Local static files were loaded to the texture by using the .require() command.

The frustum culling class will traverse through all the objects in the scene every frame. It updates the objects bounding box and makes sure the three.js Frustum Culling is false. For each plane of the frustum it sees if the appropriate corner of the bounding box, the part that could clip, is in the frustum. If any corner is in the frustum we consider the object in view. The main bug was that if the object was a light, it wouldn't render the light offscreen even if it obviously would affect the environment.Therefore we had to filter out lights and non-meshes.

RESULTS

When all of the pieces came together, the resulting project was a fun first-person driving game with complex visual detail and many user-friendly features. Various screenshots of the game running are provided below. The player starts on a forested straight path with a mountain in the distance, and as the player approaches the mountain the road begins to wind around and up the mountain. The player must make it to the top of the mountain without falling off the road and without succumbing to the rising fog, which will kill the player if it reaches them.



An ordinary screenshot of gameplay, as the player approaches the mountain

DISCUSSION

The greatest setback we faced in producing the results we were envisioning was the amount of time we had. Despite this challenge, our finished product incorporates the intended features in a way that significantly contributes to the overall quality of the game, and so the project was a success.

The collaborative process taught us that dividing and conquering is by far the best way to maximize progress. Our team was at its most efficient when we were working on modular independent components (such as the fog and the HUD) that can easily be incorporated into the main game regardless of its current state. Most of these modular features were worked on solo, since in concept they were straightforward but required some labor to figure out the syntax and alignment.

Another useful lesson was using Google for assistance is better done sooner rather than later. For some of the features such as the minimap, attempting to implement it in some fashion and repeatedly searching for help with more specific errors was often fruitless, since the three.js world is so complicated and people use slightly different versions of these libraries for similar projects; finding direct help was either impossible or the provided solution was ineffective. Instead searching for the desired feature directly would lead to some example code already created by a three.js user that could be quickly adapted to our code.

Given more time, we would be the most eager to improve the driving physics and the overall feel of the game. We attempted to build the environment using dark colors and realistic geometry and textures, but there are some places where this falls short. Additionally, the car physics being very “bouncy” means that the first-person camera can rotate very quickly sometimes, which detracts from gameplay.

A major obstacle in the course of development was the slow pace at which editing positions, rotations, etc. for objects occurred. We lacked a real-time editing system or tools for building the world out, so putting the world together required frequent code reloading and guesswork. In retrospect, becoming familiar with the Dat.GUI and building out our tools earlier in the process would have made it much easier to scale and to put together more complex environments, since our efforts were definitely tempered by the long process of putting even one set piece in place.

CONCLUSION

We were able to effectively divide the workload of the project to maximize productivity and create a final product that is both aligned with our original vision and sufficiently complex from a computer graphics standpoint. Our biggest takeaways from the project were the very effective division of labor (particularly when using branches on GitHub), the complexity of trying to recreate certain effects in three.js (such as textures and an on-screen minimap), and the power of using a physics engine along with Blender models to easily model geometry that the player can interact with.

CONTRIBUTIONS

Chris was responsible for all of the Blender models and shader code. He and Kenny worked together on implementing the physics library into our game.

Kenny implemented frustum culling and the input control and camera framework. He worked with Chris on implementing the physics library into our game.

Parker made the various add-ons to the game, like the menu screen, background music, minimap, HUD, and LOD rendering/textures.

Ryan implemented the fog particle system, which involved carefully tuning a complex physics-based interaction system, and researched the GLSL shader implementation

WORKS CITED

Code snippets that were copied or adapted are cited in the code itself.

Sounds used to compose sounds and music:

background music:
https://freesound.org/people/T0XlC/sounds/553932/
https://freesound.org/people/morsine/sounds/607070/
https://freesound.org/people/Sotiris_Laskaris_Open_Studio/sounds/567387/
https://freesound.org/people/MATRIXXX_/sounds/657653/
https://freesound.org/people/bloompod/sounds/530896/
https://freesound.org/people/Garuda1982/sounds/661013/

Informal collection of links and sources:
https://stackoverflow.com/questions/42562056/how-to-use-rendering-result-of-scene-as-texture-in-threejs

https://en.threejs-university.com/2021/08/04/creating-text-with-three-js-json-fonts/
https://sbcode.net/threejs/loaders-gltf/
https://discourse.threejs.org/t/gltf-accessing-meshes-from-an-imported-scene/1845/2
https://www.tutorialsteacher.com/nodejs/serving-static-files-in-nodejs
https://threejs.org/docs/?q=audio#api/en/audio/Audio
https://gamedevelopment.tutsplus.com/tutorials/quick-tip-how-to-render-to-a-texture-in-threejs--cms-25686
https://discourse.threejs.org/t/the-webglrendertarget-texture-is-not-displayed-please-help-to-check/29175/2
https://stackoverflow.com/questions/20314486/how-to-perform-zoom-with-a-orthographic-projection
https://stackoverflow.com/questions/63872740/three-js-scaling-a-plane-to-full-screen
https://stackoverflow.com/questions/67334095/flip-camera-upside-down-three-js
https://threejs.org/docs/#api/en/materials/ShaderMaterial
https://medium.com/@sidiousvic/how-to-use-shaders-as-materials-in-three-js-660d4cc3f12a
https://threejs.org/docs/#api/en/core/Uniform
https://threejs.org/docs/#api/en/renderers/webgl/WebGLProgram
https://pmndrs.github.io/cannon-es/docs/classes/Body.html
https://pmndrs.github.io/cannon-es/docs/classes/Box.html
https://pmndrs.github.io/cannon-es/docs/classes/Vec3.html
https://pmndrs.github.io/cannon-es/docs/classes/Trimesh.html
https://pmndrs.github.io/cannon-es/docs/classes/Sphere.html
https://github.com/pmndrs/cannon-es/blob/master/examples/rigid_vehicle.html
https://www.khronos.org/opengl/wiki/Uniform_(GLSL)
https://www.youtube.com/watch?v=C8Cuwq1eqDw
https://thebookofshaders.com/glossary/?search=struct
https://threejs.org/manual/#en/load-gltf
https://www.pexels.com/photo/close-up-photo-of-green-pine-tree-leaves-4515743/
https://www.gamedevs.org/uploads/fast-extraction-viewing-frustum-planes-from-world-view-projection-matrix.pdf
https://jsfiddle.net/f2Lommf5/11653/
https://www.youtube.com/watch?v=otavCmIuEhY
https://solutiondesign.com/insights/webgl-and-three-js-particles/
http://stemkoski.github.io/Three.js/ParticleSystem-Attributes.html
https://www.gamedev.net/forums/topic/512123-fast-and-correct-frustum-aabb-intersection/

PROMPT

At the very least, your report should begin with a short abstract (a brief description of your project and what you accomplished) and contain descriptions of your project's goals, execution, and outcome. Make sure to document and justify (where appropriate) the approach you chose, the implementation hurdles you encountered, the features you implemented, and the results you generated/will generate. Additionally, you should briefly touch on related work when introducing your topic. We welcome you to also include simple diagrams of your project's overall architecture, as well as the occasional code block wherever they might provide any illuminating information. Finally, please touch on next steps for the project and any known issues as well. Please also provide a breakdown of each group member's specific contributions to the overall project, and include an informal works cited section (no specific/official format required).

The following is a brief outline you might follow; however, this is just a guideline to help you think about what to say, and these specific items may not match your topic.

- Abstract
- Introduction
    - Goal
        - What did we try to do?
        - Who would benefit?
    - Previous Work
        - What related work have other people done?
        - When do previous approaches fail/succeed?
    - Approach
        - What approach did we try?
        - Under what circumstances do we think it should work well?
        - Why do we think it should work well under those circumstances?
- Methodology
    - What pieces had to be implemented to execute my approach?
    - For each piece...
        - Were there several possible implementations?
        - If there were several possibilities, what were the advantages/disadvantages of each?
        - Which implementation(s) did we do? Why?
        - What did we implement?
        - What didn't we implement? Why not?

- Results
    - How did we measure success?
    - What experiments did we execute?
    - What do my results indicate?
- Discussion
    - Overall, is the approach we took promising?
    - What different approach or variant of this approach is better?
    - What follow-up work should be done next?
    - What did we learn by doing this project?
- Conclusion
    - How effectively did we attain our goal?
    - What would the next steps be?
    - What are issues we need to revisit?
- Contributions
- Works Cited