Dinky Dodger

COS 426 Final Project by Conner Kim, Mandy Lin, & Ivy Wang

Introduction

Dinky Dodger is an interactive 3D web game inspired by the game Subway Surfers. The goal of this game is to stay alive by avoiding an infinite stream of obstacles. Players can set personal records for how long they can stay alive. In our version of the game, the player controls an animated robot and attempts to dodge moving trains and brick walls. The robot and the obstacles move on three parallel railroad tracks that run from Blair Arch to a black hole. The player can control the robot with ArrowLeft, ArrowRight, and ArrowUp keys to move left, right, and jump, respectively. The trains can only be avoided by switching tracks while the brick walls can also be jumped over. The game ends when the robot collides with an obstacle. The obstacles are randomly generated so the scenario is different each round and the score can be tracked via the stopwatch in the upper left corner.

Related Work

Our game is inspired by Subway Surfers which involves moving trains and midheight level obstacles. The player in that game is able to move left, right, and jump along 3 lanes to avoid these obstacles. The game ends after a collision. Subway Surfers also includes other features such as collecting coins, ducking, flying, running on top of trains, and speeding up. These would be good features to implement if we decide to continue adding features to Dinky Dodger beyond this course.

We were also inspired by related work in 3D computer graphics. In particular, we referenced previous assignments in COS 426 to learn more about techniques and features such as texture, lighting, event listeners, camera views, ThreeJS, etc. We also looked at existing 3D models online for inspiration.

Methodology

Assets

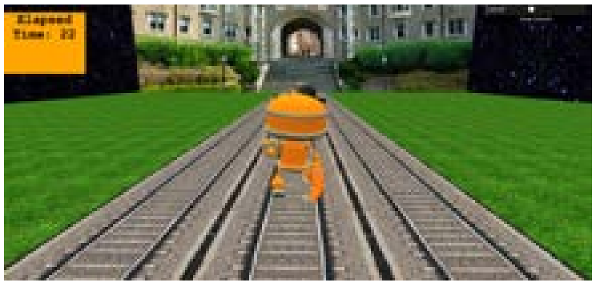
We externally sourced all assets used in the game, from textures to 3D models to sound.

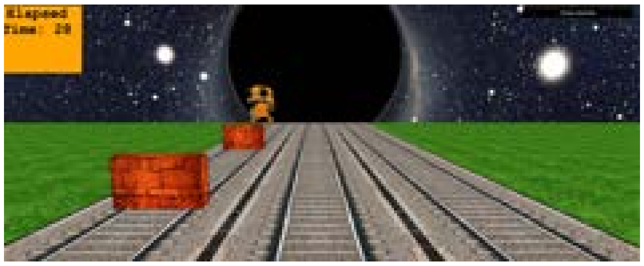
First, our webpack had to be configured properly to load binary files needed to support the .gltf files. Then, we tested each model to see if it fit our needs. The train asset was very large and took a long time to load into the game. To deal with this, our game has a loading screen about 3 seconds long before allowing the player to start the game. An easier way to check if the game is ready is to wait for the initial train to spawn on the track. In addition, instead of periodically generating and placing a new train model obstacle, we limited each track to be able to support one train each. They load in immediately when the player opens the game, but the trains on track

2 and 3 load out of view. Then, to mimic the behavior of obstacles spawning, we shift the train locations (once they are off-screen) back to the front of the game so it appears as if a new obstacle is coming toward the player.

The 3D model of the robot was taken from the official Github repository of ThreeJS examples, and came with a running animation that fit perfectly for our game. To use the animation clips that came with the model, we added a ThreeJS AnimationMixer to our app and called the mixer to update the animation and each timestep in the render loop. Our robot runs in place while objects move towards it, giving off the illusion that it is running forward.

The background, sky, and ground were implemented using textures and plane geometry. We found textured image files which we incorporated into the scenery. The ground in the game is made up of two textures: a railway track and grass. For the three lanes of railroad tracks, we used an image of one track duplicated side by side for 3 total track lanes. We then copied the tracks along the z-axis. The grass was incorporated similarly where we used one image of a patch of grass and duplicated it to complete the ground next to the railroad tracks. For the right, left, and sky backgrounds of the scene, we used an image of the galaxy. The image was expanded to the appropriate size and repositioned. For the background in the back, we used an image of a black hole to fit in with the galaxy theme and give the effect of obstacles disappearing into a black hole. Finally, the background in the front is Blair Arch, which fits in with the Princeton-themed game and is where the obstacles are coming from. In addition, we incorporated texture into the brick wall obstacle. This was created using box geometry and a brick wall texture file.

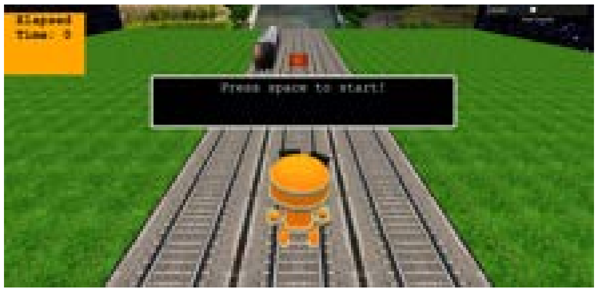




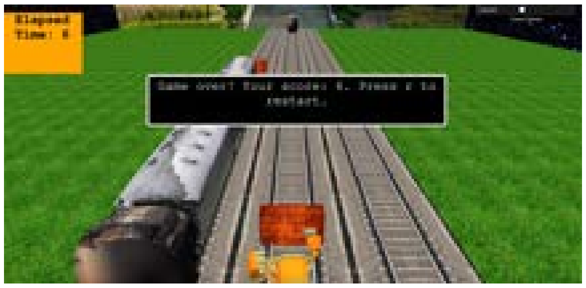
Finally, we found free-to-use game sounds to play background music when the game starts, and play a sad noise when the player runs into an obstacle and the game ends.

Game Controls

The game state is controlled by a few variables. We use a global window variable to communicate between the scene and the app when the player dies, to trigger the appearance of the end screen. A variable in the beginning controls whether the player can start the game. The 'r' key is used to restart the game, the spacebar is used to start the game, and the left, right, and up arrow keys are used to control the character.

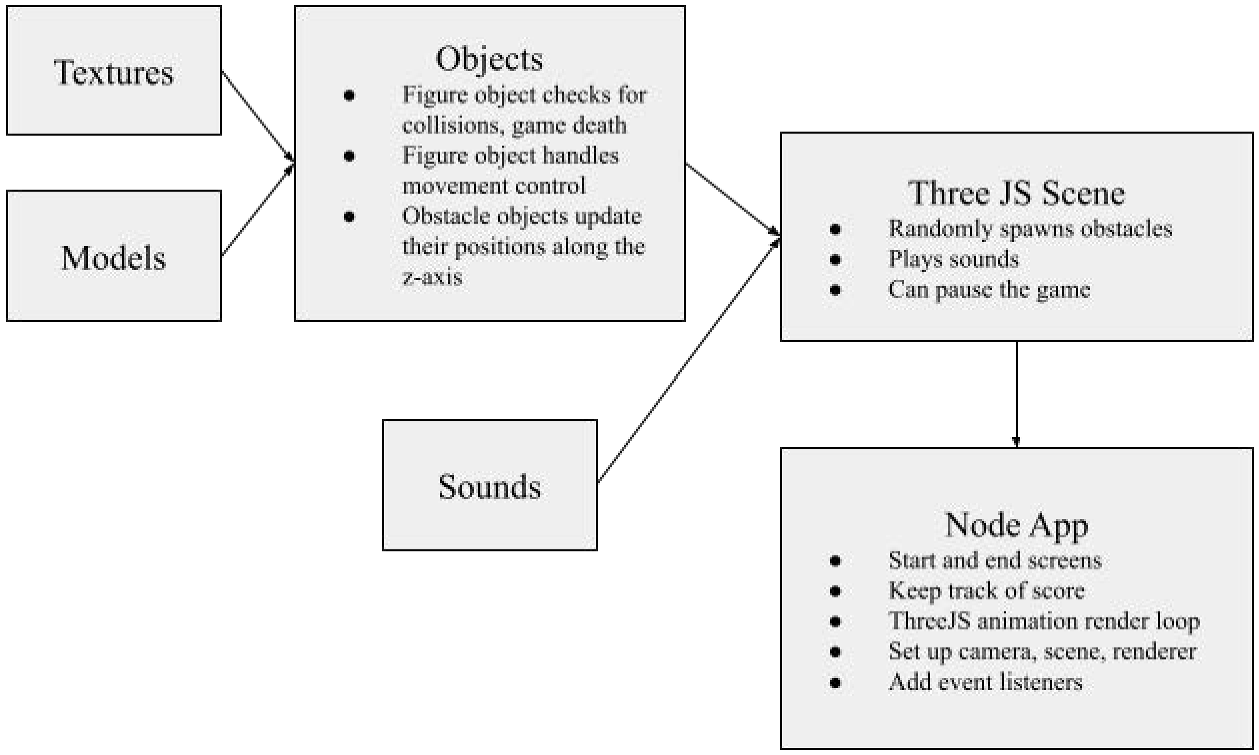


Press space to start!



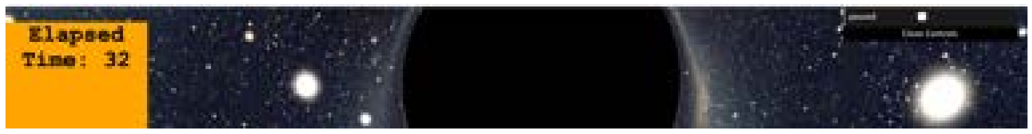
Game over! Your score: 8. Press r to restart.

Program Architecture


Textures
Models
Objects
Figure object checks for collisions, game death
Figure object handles movement control
Obstacle objects update their positions along the z-axis
Sounds
Three JS Scene
Randomly spawns obstacles
Plays sounds
Can pause the game
Node App
Start and end screens
Keep track of score
ThreeJS animation render loop
Set up camera, scene, renderer
Add event listeners

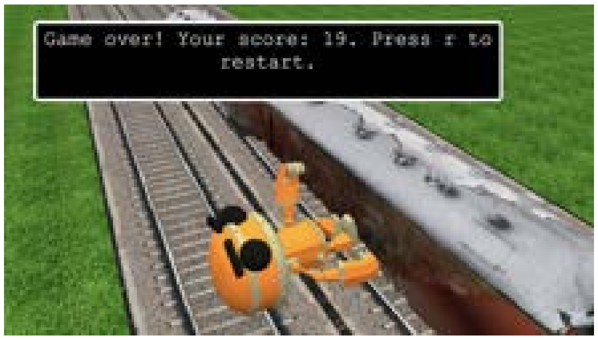
Advanced Features

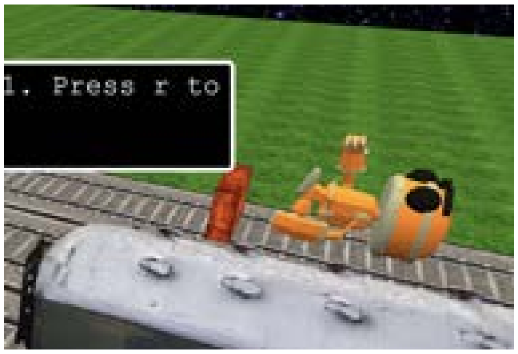
1. Texture mapping: This was explained in the previous section. The background, sky, and ground were implemented by mapping textures (galaxy, Blair Arch, black hole, train tracks, and grass) to planes. The brick wall obstacle was implemented by mapping an image of bricks to a rectangular box.
2. On-screen control panel: We used dat.GUI to implement a control panel that has a "pause" toggle which gives users the ability to pause the game. We also have a scoreboard that keeps track of the amount of time that the user has been alive. This was implemented with THREE.Clock().



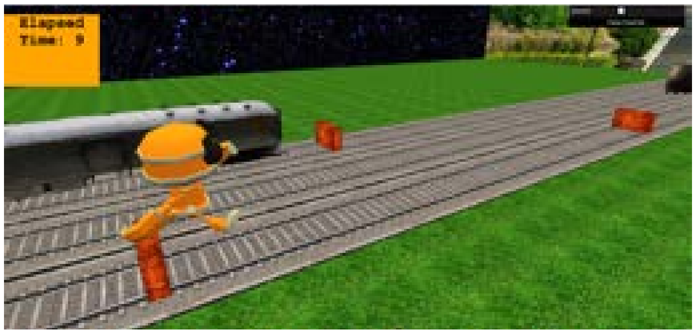
3. View frustum culling: Obstacles that move off-screen are not rendered. In particular, our game is a forward-facing game, where the user dodges the obstacles that come that them from the front. Obstacles that the user has successfully avoided would continue moving in the z-direction behind them, but these obstacles are now irrelevant to the game. To save on rendering time, we set the element's visibility to false in the rendering loop if they are outside the view of the camera.
4. Collision detection: The game ends when the robot collides with a train or brick wall obstacle. We implement collision detection by computing bounding boxes for the robot,

train, and brick wall. We took the position of the object and stored offsets for the width, height, and depth which we would use in our computation. When the bounding box of the robot intersects with a train or with a brick wall, the robot dies and the game ends. The intersection can be from running into the front, switching lanes and hitting the sides, or unsuccessfully trying to jump over the obstacle. After the robot collides, we rotate and reposition the robot so it is lying on the ground.


Game over! Your score: 19. Press r to restart.


1. Press r to

5. Simulated dynamics: We introduce a netForce property for the robot, and an applyGravity() method that was called in our update() method every time the render loop performed another iteration. applyGravity() calculated the net force using Newton's second law of motion, then updated the netForce property. On every update() call, we also call an integrate() method. This performs Verlet integration and uses the netForce property to update the position of the robot. We set a minimum y value of 0 so that the robot would never clip through the ground. To apply the jump mechanism, we simply applied a net upward force whenever the 'jump' key was pressed (utilizing an event listener), and set a maximum height for the jump in the form of a maximum y value.


Elapsed Time: 9

6. Sound: We incorporated background music and a game-over sound effect into our game. The background music starts when the user presses the spacebar key to start the game. When the robot collides and dies, the background music ends and the game-over sound effect is played. This was implemented in the scene using the audio loader and audio listener from ThreeJS.

Reflection

We learned a lot by making this game because it was like a reverse-engineering of an existing, beloved game. In particular, this was the first time any of us tried to animate in ThreeJS so we learned a lot about the animation system and how the render loop works to move a model through its animation states. Also, we learned about how webpack acts as a build tool for our app and why its loaders are needed to read different file types. It was challenging to figure out a lot of this on our own, but through online examples and asking on Ed, we managed to make things work out.

Next steps for this game would be to improve upon the gameplay experience. As mentioned during our presentation, if we move the floor texture at a constant rate the illusion of movement will be even more realistic. To make it a more challenging and engaging game, we could raise the base speed at which obstacles move towards the player. Then, as the game goes on for a longer time, we could raise the difficulty in stages so that the obstacles move and spawn at faster rates. Another stretch goal we had wanted to implement was the addition of coin-collecting. This would encourage players to switch tracks more frequently and each coin could be worth an additional few points in the score. Speaking of score, we could score the user's highest score as a cookie so that it appears as a personal record to beat. The possibilities are endless for gameplay; we could add in different power-up items to make the player's journey more fun. There could be a flying power-up that propels the player to higher heights and allows them to encounter new obstacles. Or, there could be a magnet that pushes all trains away from the player. Overall, we had a lot of fun watching the game come together the way it is, but when it comes to next steps, the limit is only in our imagination!

Contributions

Conner: Character movement (side to side, and jump), simulated dynamics and gravity, collision detection fixes.
Mandy: Obstacle movement and random spawning, Collision detection, Robot position after a collision, textured ground and background, brick wall obstacle, sound
Ivy: Find and load 3D models into the game. Add start and end screens, and the functionality that goes with them.

Works Cited

Models:

- Robot Model: https://skfb.ly/ouGrB
- Train Model: https://skfb.ly/ozH8v

Sounds:

- https://mixkit.co/free-sound-effects/game-over
- https://mixkit.co/free-stock-music/tag/video-game

Images:

- https://wallpaperaccess.com/4k-black-hole
- https://wallpapercave.com/black-space-wallpaper-hd
- https://www.dreamstime.com/royalty-free-stock-images-seamless-train-track-texture-rail-image35784519
- https://www.the3rdsequence.com/texturedb/category/grass/
- https://www.peakpx.com/en/search?q=brown+brick+wall
- https://www.cosmopolitan.com/politics/a8507607/princeton-swimming-and-diving-misogyny/

Texture mapping: https://threejs.org/manual/#en/textures
Sound: https://threejs.org/docs/#api/en/audio/Audio