Bashar Eid
Kris Selberg
Thomas Hughes

Advised by Adam Finkelstein

COS426: Computer Graphics
Final Project

Temple Run Xtreme

Abstract

Our project is a desktop variant of the popular mobile game Temple Run, and aims to recreate and expand upon the original game's mechanics, incorporating personalized elements such as unique obstacles, characters, sound effects, and textures. We paid homage to a beloved game and got some insight into game mechanics like terrain generation, collision detection, and dynamic difficulty adjustments.

Introduction and Previous Work

Temple Run was released in 2011 by Imangi Studios. The game was an instant success because of how it made use of the new mobile gaming medium. Gameplay was simple but controls were comprehensive: you're a hero running down a pathway, avoiding obstacles by tilting left and right, and making turns to stop from crashing into the walls of the path. You're being chased by demon monkeys who get closer the more you trip up. Our group spent way too many hours of our youth playing this game, such that we felt we could do it justice in desktop form.

With "Xtreme Temple Run," we sought to understand and reimagine this classic, enhancing it with our creative spin. Our primary objective was to replicate the core experience of Temple Run while introducing new elements. These included custom-designed obstacles and characters, immersive sound effects, and unique visual textures.

In setting out to create Xtreme Temple Run, we stood on the shoulder of giants. The top-rated game from last year's Graphics cohort, Glider, did a phenomenal job of making beautiful, seemingly infinite terrain which we wished to replicate. We drew inspiration from their dynamic world creation with their ChunkManager.js file and consistent player perspective.

Approach + Methodology

Spider-Man Character

In selecting a character to be the runner in our game, we were limited in that we knew they needed animation for sliding, jumping, and running. We selected Spiderman because the character had jumping and running animations, and we were confident we could create a sliding animation that we considered satisfactory.

Jump Animation

Our jump effect makes use of Spiderman's pre-existing jump animation. It first checks if the character is already in the process of jumping (this.isJumping), and if so, it does nothing–preventing mid-air jumps. The function then uses the TWEEN library to create a smooth jumping animation, which consists of two phases: an upward movement (jumpUp) and a downward movement (fallDown). These animations alter the character's Y-position, first increasing it to simulate a jump and then decreasing it back to the original level to simulate landing. Upon completion of the jump, the function resets the jumping flag (this.isJumping = false), ensuring the character can jump again. Additionally, sound effects and a transition back to the running animation (this.runAction) are handled to make it feel more natural. The sound effect for the jump is Thomas' voice.

Slide Animation

The slide() function manages the character's sliding action. Initially, we attempted to use relative coordinates with TWEEN to rotate and move the character. However, this approach led to difficulties in correctly orienting the character during the sliding action, especially when considering different directions like left or right. To fix this, we switched to using world coordinates. In this revised approach, we defined a world axis (X-axis) that changes based on the character's current direction (left, right, or straight). For example, when sliding to the left, the world axis is set to point leftwards.

The core of the sliding animation consists of two parts: rotating the character to lie on their back (rotateToBack) and then moving them along the defined axis (slide). These animations are created using TWEEN objects. The onUpdate method updates the character's rotation based on the current value of theta (the angle of rotation), ensuring the rotation is applied correctly according to the world axis. After the sliding motion, additional animations (getUp and rotateBack) are used to bring the character back to the standing position. These animations also use TWEEN and are chained to execute in sequence, providing a realistic sliding and recovery motion.

Chunks

The ChunkManager handles the infinite terrain creation, and evolved over the course of the project, as we added features and things got more complex. Initially, chunks were simple and moved straight back. The chunk that had just passed the camera (and could thus no longer be seen because it was behind the player) would get relocated to behind the furthest chunk, so it appeared as if the path went on forever. For development

and debugging, we used solid-colored chunks with alternating colors so we could easily distinguish between them. When creating a chunk, we'd add obstacles to every few. Later on, it became important to ensure there were no obstacles on the first chunk that the user encounters when they start the game, and that there were none on corner chunks.

Initially, cubes served as placeholders for obstacles, evolving into more complex shapes like tree trunks and a custom-designed floating bat as we became more confident that the game dynamics were sound. We used bounding boxes for collision detection, which we had to manually put together because we were having issues with the ones we attempted to generate programmatically. In future iterations of this project, it'd be desirable to handle more complex collision detection, which would also improve player experience.

Once we had chunks going to the back properly, we tackled turning. This turned out to be the most technically challenging part of the project.

Turning

The turning mechanics in the game are managed by the Person class, which interacts with the ChunkManager to enable smooth and responsive turning of the character. The game's controls are designed to ensure that turns are made within a specific turn window. If we think of the pivot chunk as Chunk A, and the chunk before it as Chunk B, the turn window is the front half of Chunk A and the back half of Chunk B. If you hit the correct direction arrow on the keyboard while the character is in this window, the character successfully makes the turn. Missing this window results in game over.

The initiateTurn method gives each chunk a depth offset and a turn offset. These values are used to place the current chunk in the distance so that it aligns with the turn. While we only sent chunks to the far back before this, turnSingleChunk used these values to put them in more specific locations that would complete the turns. The depth offset aligns the chunk with the farthest one, and the turn offset positions it correctly left to right in the relative direction.

An initial approach of using the z-coordinate of the farthest chunk for depthOffset for all chunks was ineffective due to the changing origin as the player moves forward. The solution was dynamically adjusting the depth offset for each chunk. As the player progresses, each chunk's position is updated based on its depthOffset and turnOffset. So a chunk which is just before the turn will have a much smaller depthOffset (but a large turnOffset) because it only has to be moved forward a little (but quite far to the left or right).

The implementation uses key event listeners that trigger turning actions when the player presses the left or right arrow keys. Upon detecting a turn command, the handleTurn method checks with the ChunkManager to ensure the turn is safe, checking for turns that would lead to game over scenarios.

Directional changes (DIRECTION.LEFT and DIRECTION.RIGHT) that came about as the result of turns required us to reason through each individual case action that the character participated in. If you're looking straight forward and running, it's just the z value of the chunks that needs to increase to come towards you. If you're looking left, however, we have to update the x value every frame to give it a similar effect.

When a turn is initiated, the turnLeft or turnRight method is invoked. These methods use the TWEEN library to animate the character's rotation smoothly. The target rotation is calculated as a 90-degree turn in the respective direction.

The turning mechanics are coupled with the game's camera movement, ensuring the player's perspective aligns with the character's new orientation at all times. The camera position is adjusted correspondingly to maintain a consistent view of the character and upcoming terrain.

Title Screen + Game Over Screen (index.html)

The index.html file for "Temple Run Xtreme" serves as the front-end structure of the game, displaying both the title screen and the game over screen. We designed the logo using the DALL-E model, and then got the color scheme for the title screen from that logo. The file employs CSS styles to create an engaging user interface, featuring a linear gradient background and a retro-style font ('Press Start 2P'), enhancing the game's thematic appeal. We did our best to make it feel like a real arcade game. TDynamic elements like the score display and game over overlay are initially hidden and later revealed, managed by app.js, allowing for transitions between different game states.

Speed Acceleration and Score

As the game progresses, the speed gradually increases. The scoring is tied to the elapsed time, with the player's score incrementing based on how long they've been playing. For accelerating speed over time, instead of right when the game starts, we did it when the first turn was initiated, because it was more modular than trying to send the score back to the chunk manager. However, a bug was identified where the speed and score could inadvertently escalate if the page was reset and left idle before starting the game again. We fixed this by starting the score on clicking the start game button.

Sound Effects

We added a few sound effects to give the game our own personal touch. Kris is the "game over" sound, Thomas is the "jump" sound, and Bashar is the "slide" sound. We use THREEJS's AudioLoader to load in each sound.

Potential Future Improvements

We found when testing that people tended to think they could go left or right around the obstacle, and then they'd be confused when they instantly lost. We could fix this by

making the obstacle cover the entire pathway, or by having pop ups for the first couple obstacles that tell the user what button to click to successfully get around it. With more time, we would also create animations for obstacles (i.e. bat “flying” animation with wings) and create a more cohesive theme for the game, because the home page strayed from the central gameplay design because of the limited models we could find which had jump animations and other crucial attributes.

Architecture:

The Root Directory (3dtemplerun) contains the main folders and files necessary for the project's configuration and execution. The Source Directory (src) hosts the core components of the game, including the game logic, rendering, and object management. Within src, there are two key subdirectories:

- components - further divided into:
    - lights: Includes files related to lighting in the game (BasicLights.js, index.js).
    - objects: Contains ChunkManager as well as various objects, such as Enemy, Flower, Grass, Land, Obstacle, Person, and a central index.js file for managing these objects.
    - scenes: scenes of the game - all of what we do is done in SeedScene.js.
- app.js: serves as the foundational script, initializing core Three.js components like the renderer, camera, and scene, starting the render loop, managing window resizing, and handling the game's audio and start events.

We also have a public directory that contains 3D models, textures, and images. We set up access to these assets in the web pack configuration file.

Contributions

Bashar Eid

I did most of the audio-visual design of the game, including textures, animations, title screen, game over screen, and sound effects. Additionally, I built out the speed mechanics and score handling so the game would increase in difficulty and the player would have the opportunity to compete against their past runs.

Kris Selberg

Thomas and I worked on the core chunk behavior together, starting with the simple process of creation and movement and then getting into the complicated turning mechanics. I also handled the keyboard control system and corresponding camera movements. Finally, I found the stump and implemented the stump obstacle, as well as designing the floating bat myself in Blender.

Thomas Hughes

Kris and I handled chunk management together, and I additionally did obstacle management (creation, making sure they were removed from corner pieces so the game

wouldn't be impossibly difficult, etc.) as well collision detection with obstacles and the boundaries of the course.

Works Cited

- Glider: https://github.com/harveyw24/Glider
- Original ThreeJS starter code from Adam Finkelstein
- Temple Run: https://play.google.com/store/apps/details?id=com.imangi.templerun&hl=en_US&gl=US&pli=1
- Spider-man Character:
    - * title: Low-Poly Spider-Man Advanced Suit 2.0
    - * source: https://sketchfab.com/3d-models/low-poly-spider-man-advanced-suit-20-58affbb9a9e44764a6311a51d4c78ed3
    - * author: jerrylxia (https://sketchfab.com/jerrylxia)
- Tree Stump Obstacle:
    - * title: Tree Stump
    - * source: https://sketchfab.com/3d-models/tree-stump-0928aff3c0ce4671a145fe334bbf1b6a
    - * author: PULSAR BYTES (https://sketchfab.com/pulsarbytes)