COS 426 Final Project
Yongwei Che, Kathy Li, Ben Liu, Baris Onat

Thief Simulator

Abstract:

Thief Simulator 2017 is a fun game where the player is put in a house as a thief with the task to steal a list of items from the house. The thief has a limited amount of time to find the items and escape the house, or else the owner of the house will come back and catch them! The thief must also avoid a security system of cameras and guards so that they won't get caught before the time runs out. Thief Simulator 2017 uses Three.js to give the user a fun experience exploring the house and avoiding the cameras.

Introduction:

The goal for our project was to create a thief simulator game. In the game, you are a thief trying to steal items from a house. The house has cameras and guards that you must avoid. We hope that this game is able to provide gamers an enjoyable yet challenging time trying to navigate the house, weaving around the cameras and guards and stealing the items.

Some related work other people have done include "The Very Organized Thief" by Gamejolt. This game succeeds in its ability to navigate around a house with a flashlight while searching for items but fails to include cameras and guards that could catch the thief and does not have a timer – all things that our game implements.

Our approach builds off of the starter code provided and leverages the Three.js library for graphics. At first our goal was to just have the ability to walk around in a dark room while also having a flashlight that would light up the scene in front of the player. After successfully implementing that, we moved on to adding guards, items to steal and a more sophisticated house. We discovered the Three.js editor, which helped us greatly in making the large house efficiently. This allowed us to see what was more of a complete scene than what we had before.

After the scene was more or less complete, we used Raycasters to implement cameras and guards detecting the player. This approach allowed us to split up the work by having different people work on different aspects of the scene, and it also prevents bugs from one section of the code from impacting the implementation of other aspects.

Methodology:

We built our first-person game from the starter code provided in the final project specification. We started by implementing basic first-person controls with a PointerLock. From there, we implemented:

Togglable Player Flashlight

The flashlight itself is a mesh that is rendered in the scene. Its position gets updated every frame to move with the player. The flashlight's position was tricky to determine. Reading the documentation for the camera, we found that we could get the vector that the camera is pointing towards as well as the vector pointing perpendicularly up to the forward vector. Using these two vectors we were able to find the rightward vector as well using the cross product. These three vectors allowed us to place the flashlight consistently in the bottom right corner of the screen by adding these vectors scaled by some constants to the position of the camera to find the position of the flashlight. The light from the flashlight is implemented using a spotlight. The spotlight is targeting an invisible cube that is always at the center of the screen, a distance away from the player. This spotlight is toggleable with the key F. We decided on this implementation because the spotlight would light up only part of the scene, keeping most of it dark to keep the sneaky feeling of the game.

Player-Scene Collisions

We used two schemes to check whether the player was intersecting with a mesh. The player moves left, right, forward, and backward in the xz-plane. Our first scheme involves using four raycasters to shoot rays in the positive and negative x and z directions. If the distance between the player and the closest intersection in the plane is less than a predetermined player width, the player is reset back in the opposite direction so that the width is maintained and the player does not get too close. The limitations of this approach are that if the refresh rate of the game is not high enough, the player could lag through a wall because their movement is unrestricted.

In order to fix this problem, a second collision scheme was added as a pre-emptive check on player movement. In the second scheme, whenever a player decides to move either forward, backward, left, or right, a ray is shot out in those directions (relative to the player's looking direction, not the world axes). If the expected location is too close to the scene, the player is prevented from moving in that direction.

The combination of these two collision schemes leads to a well-behaved collision detection system that avoids clipping and unusual behavior. Calculating collisions with bounding boxes instead of raycasters was also considered and experimented with. This implementation was eventually scrapped due to the complication of determining where to push the player back to when a bounding box collision occurred. This is because the direction in which the player is also factors into collisions, so a player often finds themselves colliding with something and unable to move unless they are looking in a certain direction. This leads to a very frustrating and unenjoyable player experience.

Patrolling Guards

When we were implementing the guards, we started out simply by getting the Xbot model provided by THREE.js to render on the screen. After this part was handled, we next imported the animations attached with this model. The next task was to get this guard to animate walking. Up until this point, there weren't any choices to make but we had many choices to arrange how the guards went around the house. It was possible to make the guard walk a set path or implement some sort of collision system to slightly randomize the guards' movements. Giving the guard a set path had its advantages such as making collisions predictable and seeing possible bugs before they happened. But it might have had some disadvantages as the rendering might differ for users; it was hard to create a set path for the guard.

Once we realized this was not easy or preferable, we went with a combination of the two approaches to minimize the amount of unpredicted outcomes. We gave guards an initial starting position and initial direction. As the guards walk, the guard shoots a ray and finds if it is colliding with any mesh in the scene. If so, the guard either turns right, left or back.

Sweeping Cameras

The sweeping cameras were implemented with four raycasters that rotate in a circular motion of a certain radius. If the player steps into the view of one of the raycasters, and it is the closest intersection to the raycaster, then the player is detected. A camera cannot detect a player if their flashlight is off. In addition to providing an additional challenge, the cameras also sweep in the x and z directions based on the room. The sweeping motion is determined by a minimum and maximum of x-coords and z-coords. The motion is periodic based on sinusoidal functions of time. The sweeping speed of the camera is also adjustable. The rays shot by the camera are visualized by red THREE.JS Line Segments. Since the line segments are purely visualizations, the player should not be able to intersect with them. Thus, each of these line segments has a property NoIntersect, and the player collision detection ignores any scene children with this property.

Stealable Items

For the stealable items, we added them using .mtl and .obj files. The files are loaded and added to the scene by using OBJLoader and MTLLoader. These items are labeled as clickable, allowing for the player to click on them and pick them up. The list of the items is an html element that changes what it says depending on what items you have collected. We chose to implement the items list this way because it is relatively simple while still accomplishing the same thing compared to other ways, such as having a list pop up from the bottom of the screen like some other similar games do. When you pick up an item on the list, that item disappears from the list, telling you that you have already picked it up. We show and hide this list by pressing the E key, listened for in the controller.

- **Timer**

For the countdown timer we used Math.floor(Date.now()/1000) to give the current time floored to a whole number of seconds. There were several ways to implement a countdown that could pause multiple times in the game. The implementation we chose to go with was the cleanest: three global variables for: if the game is paused, the exact time the game is paused, and the total amount of time the game has been paused already – these would be updated every time the game was paused/unpaused. Before the onAnimationFrameHandler loop we kept track of the starting time. In the loop, if the game was paused the amount of time left = time paused - start time - total time paused. If the game was not paused the amount of time left = the current time - start time - total time paused. By using this method we were able to keep an accurate track of time very simply.

- **Menus and Pausing**

The menus and pausing were implemented by adapting some of the code from the Portal 0.5 game and the Get Down game from last year's hall of fame. We used an HTML file containing what is shown on the pause and start menu. The Portal 0.5 game also uses a CSS file for styling but we couldn't get the CSS compatibility to work. Therefore, we just put the style for the HTML files in the HTML files themselves in a <style></style> tag. We show and remove these HTML elements by calling hideStartmenu, showPauseMenu, and hidePauseMenu from util.js when the user starts the game or pauses the game. The game starts by clicking into the game, locking the controls and hiding the start menu. The game is paused by unlocking the controls by pressing escape, showing the pause menu, and the game is unpaused by clicking back into the game, locking the controls once again and hiding the pause menu. To actually pause the game, we have a PAUSE boolean variable which is true if the game is paused and false when it is not. PAUSE starts as true, as the game should not start without the player clicking into the game. In onAnimationFrameHandler, it only calls window.requestAnimationFrame(onAnimationFrameHandler) when PAUSE is false. When the game is unpaused, the controls are locked again and it calls window.requestAnimationFrame(onAnimationFrameHandler) to restart the animation, very similar to how the Get Down game handles pausing.

- **House**

To build the house, we used the threejs editor, and used it to first make the five individual rooms based on a sample gltf room file and later connected the five together to create our final house.gltf file. We tried to base our house layout as closely and as accurately as we could replicate an actual house blueprint. The five rooms are as follows: living room/kitchen, master bedroom, children's bedroom, bathroom, and office. In each room, the threejs editor was used to modify the wall decorations, flooring, and added the necessary security cameras, doors, and lights that were later implemented in the code. As for adding the furniture, there were several possibilities (to add in our code or to include it in the imported house.gltf file). We chose to go with the second as the threejs editor made it very easy to change the color and mesh of each object and would simplify any code we needed to write.

Results:

We measured our success by comparing our progress to the MVP and Reach we included on our slide from our 426 Final Project elevator pitch. As stated on the slides:

MVP	Reach
- 1 level - Fixed "house" layout - Fixed cameras and moving patrollers - Mini map for player - Ability for player to move around and "steal" objects - Countdown timer and randomized List of Items for our thief to steal - Escape Checkpoint to Finish the Level	- Multiple levels - Randomized "house" layout - Randomly generated cameras and moving patrollers - Lighting of the room and your flashlight affects the ability of cameras or patrollers to locate you

We were able to implement all elements of our MVP. Addressing the reach components, it would be extremely easy to create multiple levels (it would just be more threejs house.json files). Randomly generating camera and item positions would also be very simple to add. It would be a little harder but still pretty easy to have the lighting of the room and the player's flashlight affect the ability of the thief to be caught. It would be much harder to have a randomized house layout because a complete random house would likely result in easy clipping and an unplayable game overall. Therefore, when thinking about randomness of a room, only so much can be random which would involve much more work by forcing us to establish restrictions on the randomness and implementing the restrictions.

In the last couple of days, we did a lot of testing/experiments to ensure collisions between the player and all the elements in the game worked with no bugs. All in all, our results indicate that it is very possible to create a first player thief based game with all the extensions and modifications this project has.

Discussion:

Overall, the approach we took was promising and produced great results. By first splitting up tasks and continually assigning them as the game and our progress progressed, we were able to have a smooth workflow where the game was built in a timely manner.

One modified approach that could have been better would be to code together more often rather than working individually and checking in every couple days. Because we each worked on many different components, it was easy for the code to easily get disorganized and refactoring the code was a very hard task that was never really fully completed.

Some opportunities for follow-up work include the Reach ideas we introduced in the results section. We would recommend starting with some of the easier ones that require less effort. Such things would be the randomized house, where the items and cameras would be randomly assigned a location chosen from a list of possible locations in the house / rooms and the room locations would also be placed randomly.

By doing this project we learned a great deal about Three.js, game design, and graphics. This project forced us to explore more into the documentation of Three.js than the assignments have. It also gave us more experience with different elements of graphics, especially ray casting. This project also taught us how to divide a project into multiple components, allowing us to work on separate parts of the project at the same time without interfering with each other's work.

In conclusion, we were able to attain our MVP and goal for this project: to create a timed simulation for the player to navigate around a house while avoiding cameras and guards. However, the game has a little bit of lag and The next steps would be to reduce the lag of the game and adding more items to the house, perhaps randomly generating items in each room. One issue that we find ourselves revisiting is the house.json file size exceeding the git file storage and git large file storage.

Contributions:

Yongwei: Controller, Collision, Cameras
Baris: Items, Flashlight, Guards, Scene (doors/light switches)
Ben: Menus, Items, Flashlight
Kathy: House, Timer

Works Cited:

This skeleton project was adapted from edwinwebb's ThreeJS seed project by Reilly Bova ’20.

Spring 2019 Hall of Fame Winner: Colo Ring (for timer)
https://github.com/beckybarber18/coloring

Spring 2021 Hall of Fame Winner: Get Down (for pausing and controller)
https://github.com/khatna/getdown

Spring 2021 Hall of Fame Winner: Portal 0.5 (for start menu and pause menu)
https://github.com/efyang/portal-0.5

Assets:
https://free3d.com
Used THREE.js editor to edit meshes and create the house

Guard Asset and Animation:
THREE.js