First-Person Shooting Game in THREE.js

Mingchen Li
Stanford University
Electrical Engineering Department
limc@stanford.edu

Mengxi Zhao
Stanford University
Mechanical Engineering Department
mxzhao@stanford.edu

1. Game Overview

We build a first-person shooting game using pure THREE.js. A player needs to aim at and shoot down all the objects in a virtual envrionment set on Mars within given time to win the game. The system uses IMU to track the orientation of the player's head and then control the orientation of the gun. Pose tracking can be achieved with a Lighthouse. A remote mouse is used to represent the gun. The player can shoot by clicking the left button on the mouse. A playing demo is shown in Figure 1. There are both dynamic and static targets in the game. Some of the targets are harder to shoot than the others. The easier targets can help the player to get used to the control of the gun and the motion of the bullets, so that he or she can be ready to shoot down the harder targets. Note that the score for shooting down each target is the same. The current number of targets is 7, and the time is 50 seconds.

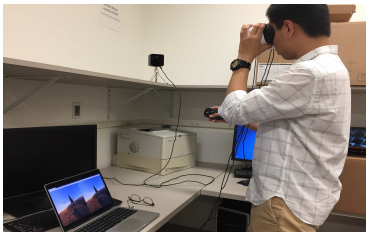

Figure 1. Playing demo

Upon starting the game, a background music (Escape.mp3) will start automatically. The music is chosen so that the player feels excited but not anxious. The player will be standing on the ground of Mars, with the background being yellow sand and universe, as shown in Figure 2. There are two guns available. The player can change gun using "1" and "2" on keyboard, where "1" is a sniper rifle and "2" is a handgun. The default gun is the sniper rifle. Each gun has a different sound when shooting a bullet: a sniper rifle sound for the sniper rifle and a 9-mm handgun sound for the handgun. The player can change gun at any time during the game.

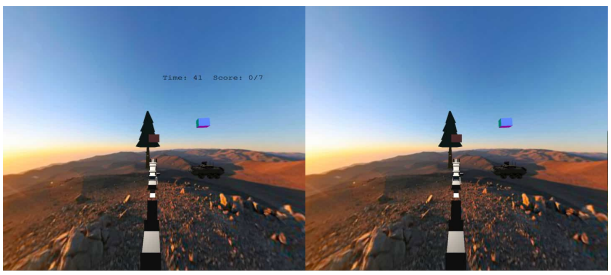

Figure 2. Starting scene

There are 7 targets in total. At the top of the scene, there is a counter that indicates how many targets have been shot and how much time is left. When a target is destroyed, there will be a glass-breaking sound. The targets can be classified into three groups (3:3:1) based on difficulty. The easiest targets are three cubes. One of them is sitting in the middle of the scene, so that the player can shoot it down right after entering the game, without any movement or rotation. The other two cubes fly horizontally along x-axis in the sky in opposite directions. They are closer to the player's starting position than other targets. Three middle level targets includes two pirate ships which fly in opposite directions but are further away from the player, and an infantry fighting vehicle (IFV) sitting in the scene. The reason this vehicle is harder than static cubes is that the player has to shoot the tire of the vehicle in order to destroy it. The most difficult target is the Starwar fighter spacecraft. It moves very fast, and the player has to shoot right at the center of it. Figure 3 is a zoomed in picture that shows the latter four targets.

If the player is able to shoot down all the targets within 50 seconds, there will be a "YOU WIN!!" shown on the screen, with a new background music (melodyloops-skater.mp3) which sounds happy. However, if the player is not able to destroy all the targets, there will be a "YOU LOSE..." shown on the screen, also with a new background music (Impossible-Decision.mp3) which sounds sad. Figure 4 shows the two situations.

Figure 3. Realistic targets

Figure 4. Win (top) and lose (bottom)

2. Features

In this section, we will describe some technical aspects of the game, including stereo effects, movement control of the player, object loading and background motion.

2.1. Stereo Rendering

We implement stereo rendering in THREE.js to provide good game experience for players. When a player wears the head mounted display (HMD), he or she is able to see the virtual environment in 3D. The method is that we render two identical scenes for left eye and right eye at the same time. There are two cameras, one for left eye, the other for right eye. The distance between them are properly set so that the player won't suffer from vergence-accommodation conflict (VAC). The two cameras are looking at the center of the scene we are rendering. THREE.lookAt() function is used to compute the view matrix for each camera. The monitor display is divided in half, with each half screen rendering the scene for one eye. Note that lens distortion correction is not implemented here since it is not significant when the player is playing the game.

2.2. IMU Orientation Tracking

In order to track the orientation of the player's head, we mount IMU on HMD. There are two different orientation sensors in IMU: gyroscopes and accelerometers. Gyros give us smooth results but with drift, while accelerometers give us noisy results but without drift. We implement a complimentary filter to get most from both sensors. We use quaternion to do orientation calculations instead of working with Euler angles to prevent any potential problems. All calculations are done on VRduino. WebSocket is used to get quaternion data from serial port to THREE.js. In THREE, both cameras' quaternions are set to the quaternion tracked by IMU. So when the player rotates his or her head, the cameras will rotate correspondingly.

2.3. Lighthouse Position Tracking

Lighthouse is used to control the position of the player. There are four photodiodes mounted on the VRduino board, a Lighthouse base station sweeps its invisible horizontal and vertical laser lines through the room, all four photodiodes would be triggered at a particular time stamp, measured in clock ticks of the microcontroller. On VRduino, we measure the horizontal and vertical time stamps on all 4 photodiodes and convert them into normalized coordinates. We then use the actual coordinates along with the normalized coordinates of the 4 photodiodes to compute the homography matrix. With the homography matrix we are able to map the position of HMD into the virtual environment.

2.4. Object Loading

We load three types of objects in the scene. The first one is objects provided by THREE, which includes the flying cubes. The second type is objects with .obj and .mtl extension from [3], such as the tree, the guns and the pirate ships. These objects are loaded by MTLLoader() and OBJLoader() in THREE. The third type is objects with .json extension from [1], including the Starwar fighter and the IFV. These objects are loaded by ObjectLoader() in THREE.

2.5. Background Motion

The background of the game is not static. It can move around in 360° with the orientation of the viewer. We use CubeTextureLoader() function in THREE to load the background. This function loads six images, which represent six faces of a cube, and construct a cube surrounding the scene. The cube is tied to the scene, so that it moves in the opposite direction to the cameras. The six images are converted from a panoramic image. We found a website that can help us do the conversion: [2].

3. Future Steps

Looking forward, we can add more levels to the game. Also, we can set different weights in scores for different targets and time. For example, if a target is destroyed in

the first 5 second, the player will get a higher score than if the target is destroyed in th later 5 seconds. What's more, we can use another VRduino to track the movement of the player's hand to control the gun while the one we already have will be used to control the movement of cameras.

References

[1] Exocortex. Online 3d modeling, 2013. clara.io/scenes.

[2] Gonchar. Panorama converter, 2014. gonchar.me/panorama/.

[3] Kenney. Kenney game assets, 2010. www.kenney.nl/.