**COMP410- COMPUTER GRAPHICS TERM PROJECT**

**Project Report**

**Alpay Sabuncuoglu – 40281**

# Introduction

In this project, I created a shooting game with WebGL. While the players continuously move in a tunnel, they are shooting some objects which the gun is tracked by tracking.js and bullets are using real physics engine ammo.js.

The project specifications in the proposal were:

- Continuous game-play in a tuple with Three.js.

- Character creation with Maya

- Object Tracking with tracking.js

- Real-physics simulation with ammo.js

# Program Design Process and Motivation

While I was planning this term project my motivation was learning maximum possible tools for everyday graphics application. WebGL is a web-content API based on OPENGL ES 2.0 which renders on HTML canvas object and it is supported by all modern browsers. It executes the program on native GPU. It is simple to create a WebGL program since it can be mixed with other HTML and JS elements and the program frame is ready because we are using it in a browser. A guide to build a WebGL application can be listed as:

## WebGL Program Structure

1. Create a canvas element.

2. Obtain a drawing context for the canvas.

3. Initialize the viewport.

4. Create one or more buffers containing the data to be rendered (typically vertices).

5. Create one or more matrices to define the transformation from vertex buffers to screen space.

6. Create one or more shaders to implement the drawing algorithm.

7. Initialize the shaders with parameters.

8. Draw

## But I am Working With Three.JS

Three.js is a JS library that allows users to extensive usage of WebGL. To be able to display anything with three.js, programmer needs three things: A scene, a camera, and a renderer so the scene can be rendered with the camera.

The program structure does not change with Three.js but the work I have done is dramatically change since I used this extensive library.

## Helper Libraries

I also used tracking.js to track movement of the gun (which is just a cyan plate – I am just tracking a cyan plate! -) and ammo.js to use physics engine in my program. Normally, the physics engine gives a better look on a game but it was not pleasant for my program, but I kept it to learn how to use it. Better and impressive examples can be found under the three.js/examples/physics (<https://threejs.org/examples/>).

# IMPLEMENTATION

First, traverse the program folder:

* assets
  + All library js files, common files for music and texture
* Css
  + Style file for html
* Monster
  + Monster object, which is exported form Maya

index.html

My implementation is coded at assets/index.js and the main program calls are implemented at index.html. Let’s look at the code at the index.html file:

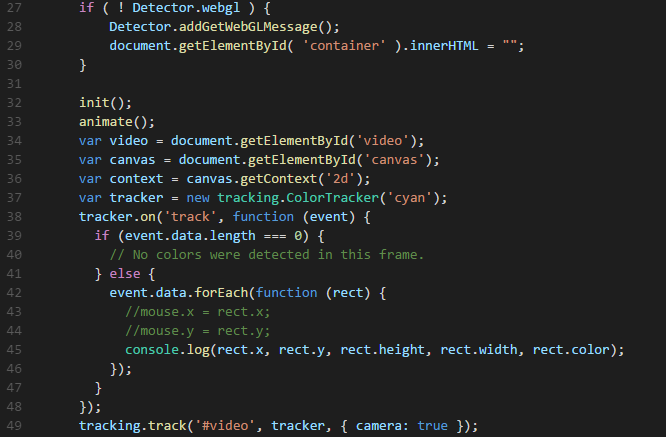


Figure . index.html script element

This is the whole code in the html file. It checks if the browser supports WebGL, initiate the graphics and physics functions, animate and render them. Also I placed the tracker code here, which identifies the blue object and track it. The detailed explanation and demo program can be found from: <https://trackingjs.com/examples/color_camera.html>, which I also implemented the same design pattern in my program.

## What does init() and animate() do?

init() does what Three.js wants, it defines a scene, camera and renderer (this is not all init() does, but it is main purpose is this.) It also instantiates the graphics and physics objects. The graphical objects are:

* Monster object which is loaded with the Object/MTL loader which is defined by Three.js libraries.
* Tunnel Object
* Ball (Maybe if I would say bullet it could be more convenient) Object
* Starfield (turns clockwise at the background)

init() also calls initPhysics() function to initiate physics engine of the program.

animate() is a more complicated function which has below functions in its structure:

requestAnimationFrame(animate);

render();

update();

requestAnimationFrame() is a beautiful recursive function which tells the browser that you wish to perform an animation and requests that the browser call a specified function to update an animation before the next repaint. The method takes as an argument a callback to be invoked before the repaint.

render() function does what its name tells, it updates some variables (which could alsobe implemented on the update part) and renders the scene with the call:

renderer.render(scene, camera);

update() function checks the variable positions and target positions and if they intersects gives points to the player.

# OUTPUT AND DISCUSSION

So, starting from simple when we have scene, camera and renderer there is nothing but after adding tunnelMesh it looks like I did something useful. The pictures at the Figure 2 are some of the screen outputs when I first create the tunnelMesh. I used texture wrapping around the cylinder (texture is assets/electricity.jpg), and set the camera inside from the cylinder and update the offset so it looks like turning around. When the player looks from the inside of a turning cylinder, it looks like moving forward. But nothing here goes anywhere!

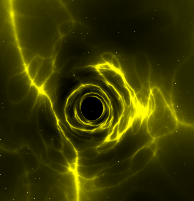
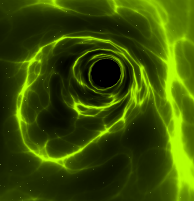
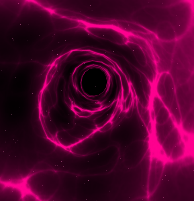


Figure . Screen outputs of the Tunnel Mesh

Then after adding monster to the system, I should have find a solution how to recognise hitting something. I used ray-casting as a solution:

mouse.x = (event.clientX / window.innerWidth) \* 2 - 1;

mouse.y = - (event.clientY / window.innerHeight) \* 2 + 1;

var vector = new THREE.Vector3(mouse.x, mouse.y, 1).unproject(camera);

var ray = new THREE.Raycaster(camera.position, vector.sub(camera.position).normalize());

var intersects = ray.intersectObjects(targetList, true);

code part of the solution is as simple as the code above. That is it, that is why Three.js is simple handled everything for the user. Intersect array stores objects hitted by the mouse (which I put the object list to the targetList so if another programmer wants to add another objects like monster they can easily add another targets).