CS324: Computer Graphics Coursework Report

For my graphics coursework, I decided to make two mini-games. This way it would meet the "Levels" criteria shown below because not only are the environments different but the gameplay in each level is different too. More information about each level will be detailed later in this report.

1. **Levels:** You must implement two uniquely designed environments. At the very least, environments are considered unique if they look distinctively different. Each level must have a distinct texture.
   (a) Environments generated using procedural algorithms will not be considered to be uniquely designed unless the environments are built using different units.
   (b) Altering parameters within the same environment will not be considered as a new level (such as lightning conditions, movement speed, etc.)

Both levels have an ambient source light as well two other distinct light sources, namely directional lights, and point light. Please see the code for this (especially the init function).

2. **Light Sources:** The game must have the following light sources implemented:
   (a) Ambient light must be implemented.
   (b) Must have at least two distinct light sources other than ambient light in each level.

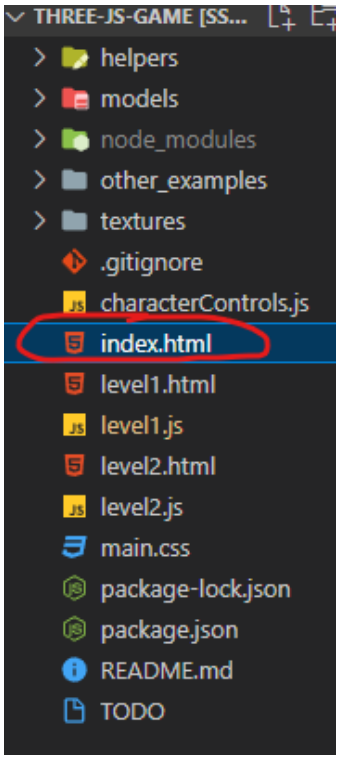
There is a main menu when first loading the index.html file. The instructions for each level are clearly displayed immediately before starting that level, when the player has read the instructions and is ready to play, then they can simply click on the screen which will begin the gameplay and they can pause the game at any point by pressing the "Escape" key (for both levels).

3. **Menu:** The game must have a main menu that can be operated using a mouse. The main menu must contain an instruction section to explain clearly how a player can control the game.

In level 1, because it is an FPS game this requirement is automatically passed. For level 2, you can "peek" forward whilst playing the game by pressing the "B" key on your keyboard and then switch back to the default view by pressing "B" again.

4. **Camera:** The game must contain at least 2 camera views. This can be achieved either by using 2 different cameras or a single camera with multiple views. Action games with a mouse-controllable viewpoint automatically pass this requirement.

How to run the code

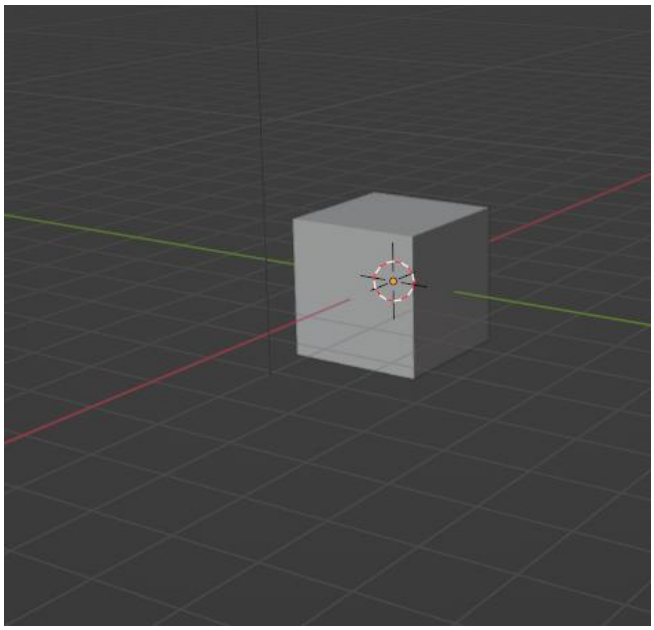


Navigate to the **index.html** file (which is at the outermost level of the directory) on your browser, then everything will be self-explanatory from that point. There are two buttons which will take you to each of the two levels.

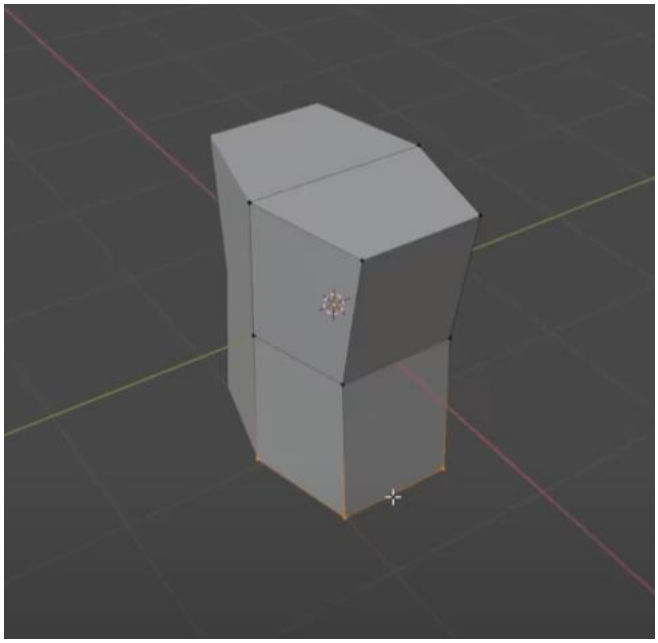
The characterControls.js was taken from a tutorial to make the third-person camera – this is also acknowledged in the code in the comments.

After completing each level, you will see a button that will navigate you back to the main menu.

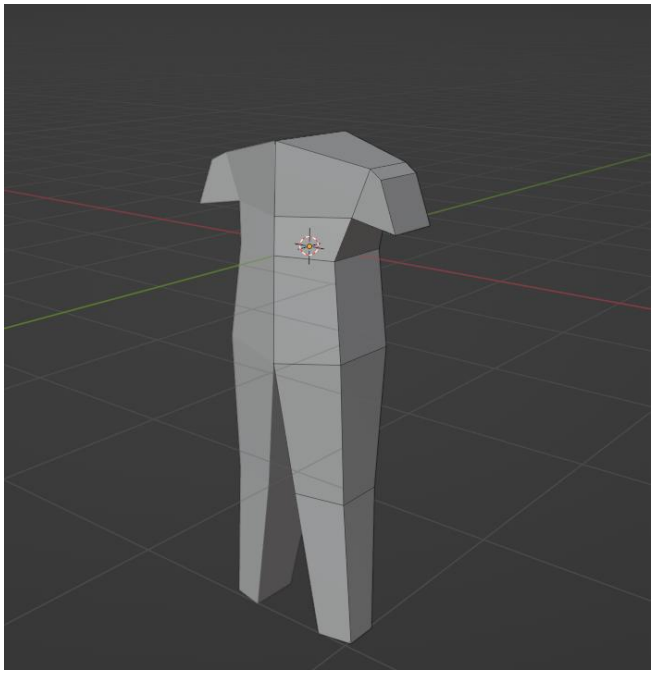
Blender Model



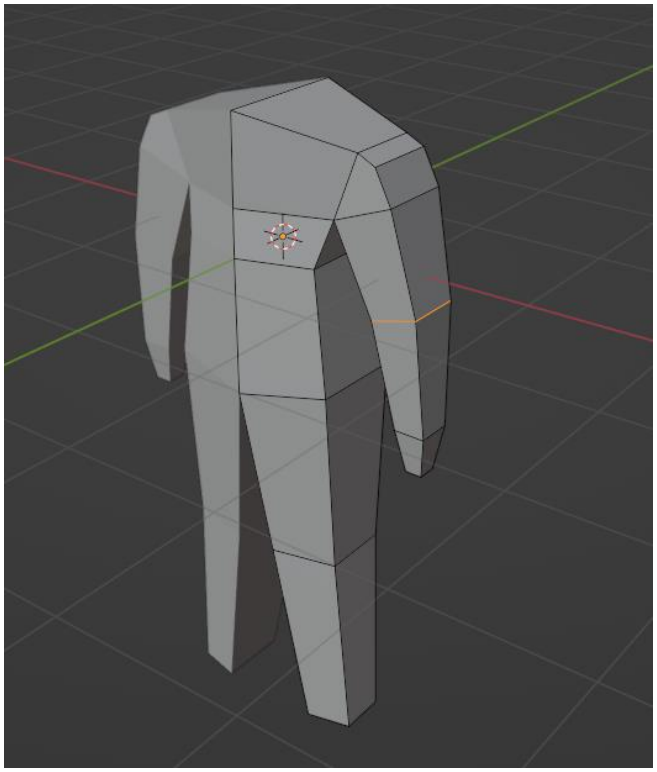
In the beginning, there is only a cube, the default shape present when you open Blender.



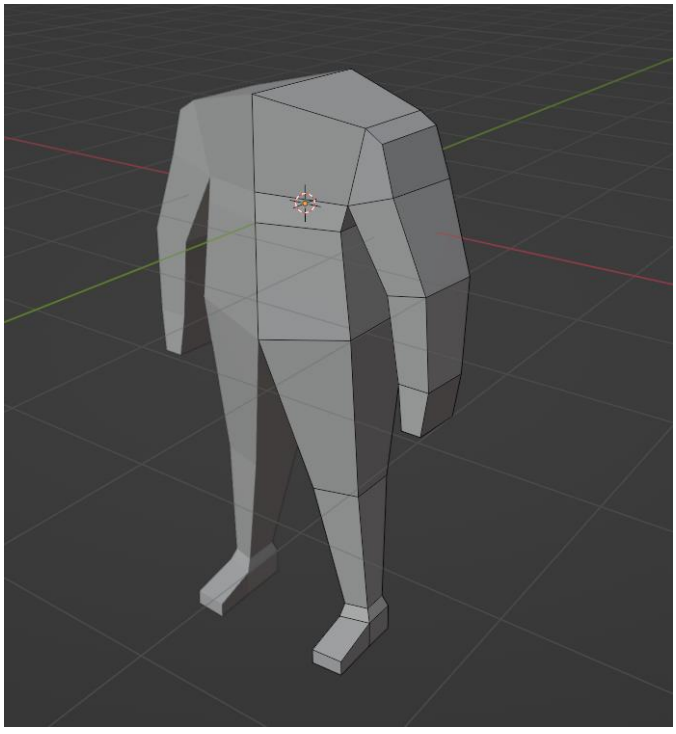
I used the mirror modifier in Blender to create the “torso” of the body from the cube.



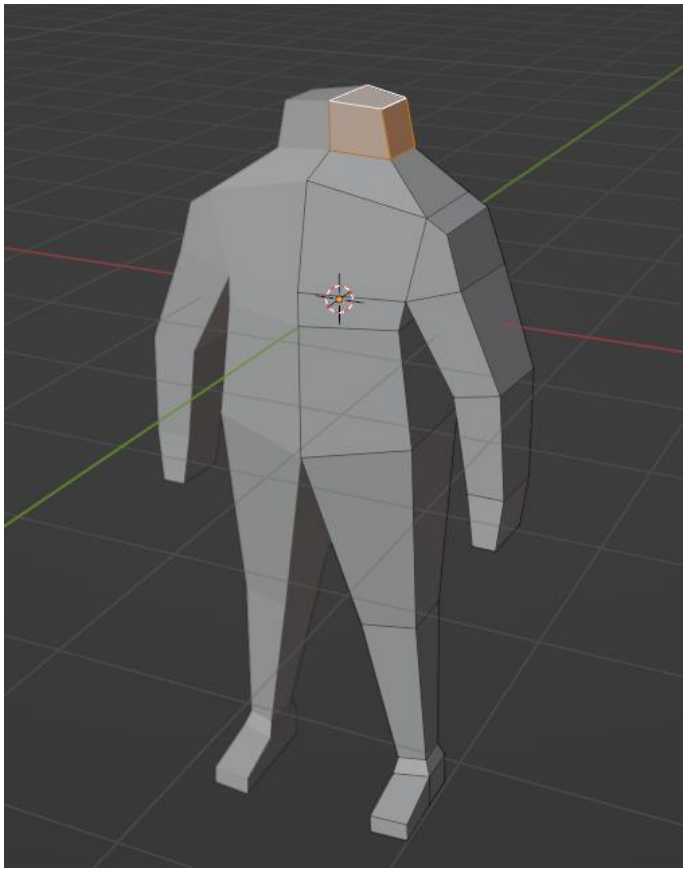
After creating the torse, I used the elongate tool in Blender to create this person’s legs and shoulders.



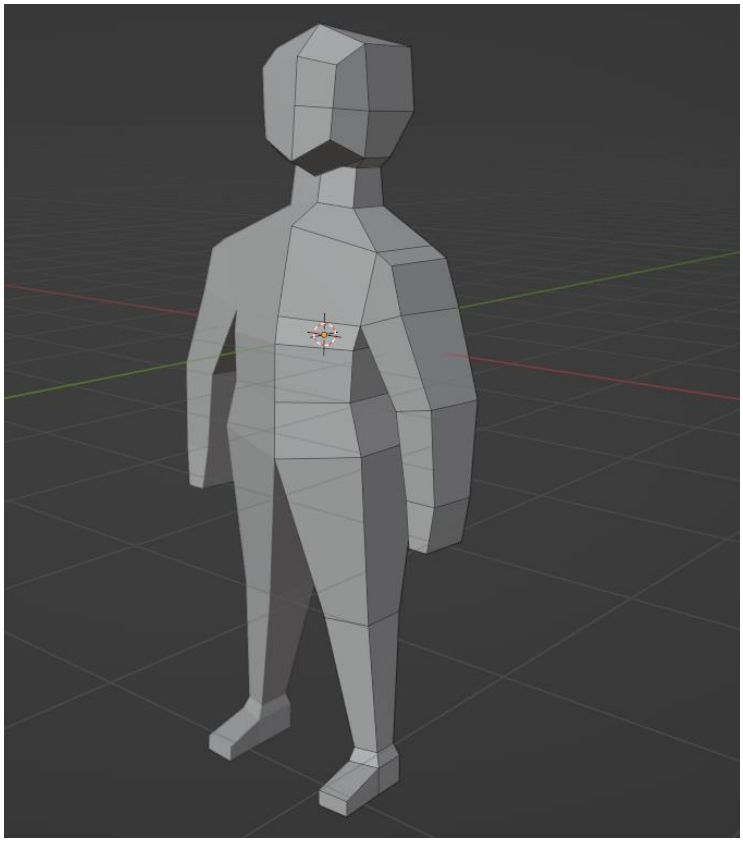
Extended the arms from the shoulders.



The lower body is done after adding the shoes.



Added a neck to the body.



Adding a basic head to the Blender model. I found modelling the fingers tricky, so I just left it at the hands.

Adding a bit of colour to the model, it now looks like the person is wearing clothes.

After a change of direction, I decided to make the person into a zombie.

This is the model that will be used in level 1 of my Three.JS game.

For level 2, I decided to use a pre-rigged character that came with the Three.JS examples instead of the person model.

Please see the submitted zip file for the original .blend file and the ".glb" file too. Do note that there is no .blend file for the soldier because I used one of the examples on the THREE.JS website.

Implementation: Level 1

For level 1, I wanted to make a relatively simple game. So, I made an FPS game where the objective of the game is to shoot as many zombies as you can. You start off with 3 lives and if you touch you lose a life, the game ends when you lose all 3 lives.

I built off the Pointer lock example on the Three.JS examples hence, the movement controls are the same. I've broken down this game into various aspects that when combined make up the game.

Shooting

The main aspect of this zombie shooting is arguably the ability to shoot the zombies! To make it easier to see where the player would be shooting, I inserted a simple crosshair right in the centre of the screen that would show the player the line of fire for their bullets. This was done in HTML with a bit of CSS so that the crosshair would always stay in the same position at the centre of the screen no matter where the player was looking, alternatively, quaternions could have been used to achieve the same effect.

To create the effect of "shooting", I spawned/created a bullet in the same location as the camera and made its velocity the direction that the camera was pointing at. This way, the bullet would fire in the same direction as the player's orientation. The bullet itself is a Three Mesh with a sphere as its geometry. To reduce the amount of computation the GPU must do, I add an asynchronous function that would delete the bullet in 1 second because the plane was only 200 units by 200 units.

Collision Detection

Collision makes the game complete, and it can be broken down into two constituent parts which are detailed below. For this level, I didn't use Raycasters but rather bounding boxes to detect collisions between objects.

Bullet With Zombie

In the animation function, the position of each zombie and each bullet is checked to see if their bounding boxes overlap. If so, the zombie is "respawned" which is implemented by placing it in a different location with a different velocity (speed and direction).

Zombie with Player

Because the player is just a camera, it isn't possible to use bounding boxes, so I used Euclidean distances to check if the player encounters a zombie. If this distance is less than a pre-determined distance, then this counts as a "collision".

Zombie Movement

The zombies are spawned at random locations and move at random velocities. They are also constantly rotating too to add another dimension to the gameplay. When they touch the walls (implemented by checking their coordinates and comparing them with the size of the floor plane), the appropriate part of their velocity vector is inverted which creates the illusion of them being deflecting off the walls.

Implementation: Level 2

Level 2 is a bit different to Level 1, the objective is to cross the bridge as quickly as possible. However, there are moving tiles on the bridge which should be avoided because if the player steps on them then it results in losing a life. And there are only 3 lives to start off with.

Something to note is that the blender model of the zombie is only used in level 1 and not in level 2 because I felt that level 2 is already sufficiently difficult and adding more things would make it unplayable and add clutter to the game.

Collision Detection

Every time the soldier touches the moving red tiles on the floor, a life is lost. This is implemented with a Ray caster (unlike with a bounding box like in level 1) with a vector pointing directly below the soldier to check if the soldier is stepping on a tile. This is very similar to how collision was done in the pointer lock example when **on top** of a cube. No collision detection is implemented to check when the player touches the walls simply because of a lack of time before the deadline, however, the idea was to have the walls "push back" on the player.

Third person camera

I wanted to make this level a bit different to the first level, so I made it a third-person camera which looks onto the player's avatar so it's clearer to see exactly where he is and what he steps on. If it were a first-person camera, it would be difficult to see what the legs are stepping on without constantly looking downwards.

The camera is stationary relative to the soldier, every time the player moves the camera gets updated and hence moves with the same displacement. This means it is always looking at the player. Furthermore, the camera is placed in an inconvenient position looking backwards at the character instead of forwards. This is intentional because when it was placed behind the player's avatar (the soldier) the game became way too easy. Placing it like so adds some amount of challenge because the player doesn't know what's coming next. However, I appreciate the need for players to look forward which is why I added a camera pointing in the opposite direction that you switch to by hitting "B" on the keyboard.

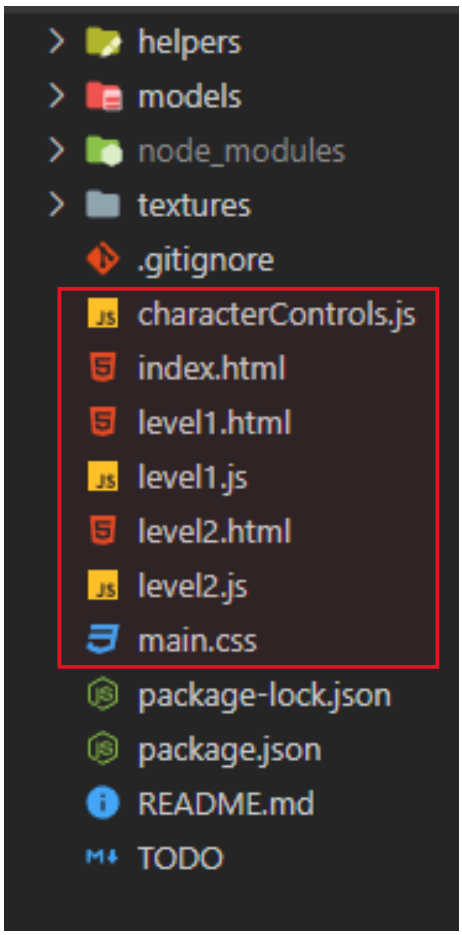
Tile Creation and Movement

There are a fixed number of tiles, but they are placed in random locations on the "bridge". After some trial and error, I felt that having 20 tiles on the plane of length 250 units would be ideal as it is not too easy not too difficult to play. Each tile moves independently with a random speed "bouncing" (implemented the same way as the zombies in level 1) off the two side walls continuously until the game is over.

Character Animation

You may notice when you play this level, the soldier is animated and therefore has realistic movement. I used an existing pre-rigged model from Three. JS. The character was loaded using a GLTF loader and the illusion of walking was created by cycling between each frame in the animate function (which repeats 60 times a second).

Code Structure



The bulk of the code is located inside the red box. The “models” directory contains all the 3D models used and the “textures” folder contains all the .jpg images used to decorate the walls and the 2D planes used in the game.