Computer Graphics

A Game With a WebGL-based Framework

Group 4

Le Viet Anh| BI9-035

Nguyen Tu Tung | BI10-187

Tran Hoang Minh | BI10-119

Tran Bao Huy | BI10-079

Pham Hoang Viet | BI10-192

I. Game/Scene idea

1. Description

- After WW3, the world has been occupied by infectious diseases. It's called Death Star. After the Russian used super nuclear bombs to defeat all the enemies and become the biggest empire in the world, the Russian government became dictatorial. The world is on the brink of collapse.
- Your mission is to use agility and combat skills to survive, find the cure for diseases and overthrow the domination of the Russian empire.
- Game categories: Survival, Open World, Zombies, PVC

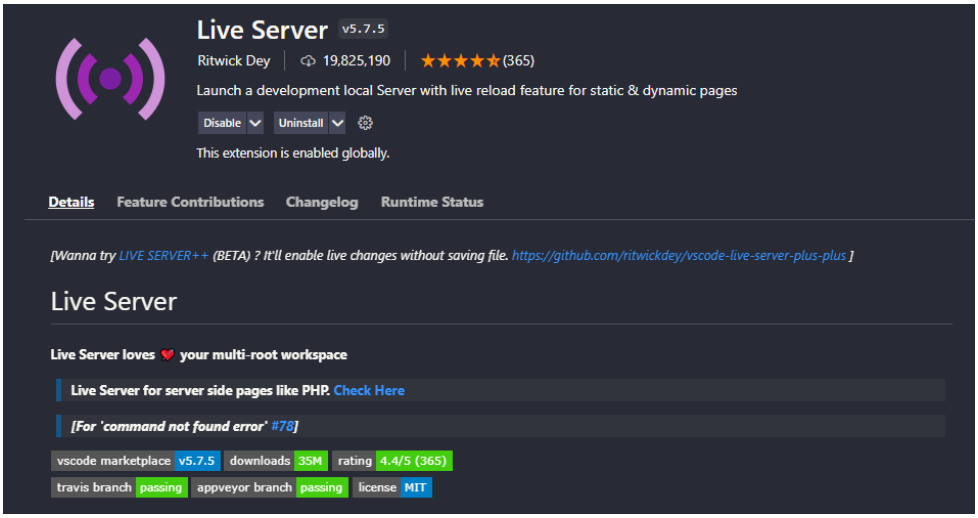
2. Interesting features

- zoom in and out by keyboard and mouse
- control the character by keyboard
- control the screen by mouse

3. How the users can use the game/program:

This is just a demo version so to play the game, you can:

- download from https://github.com/vietanh2000april/Flickering
- open in Visual Studio Code
- download Live Server extension



- run the index.html file by Live Server extension

II. Techniques used to implement features/ideas

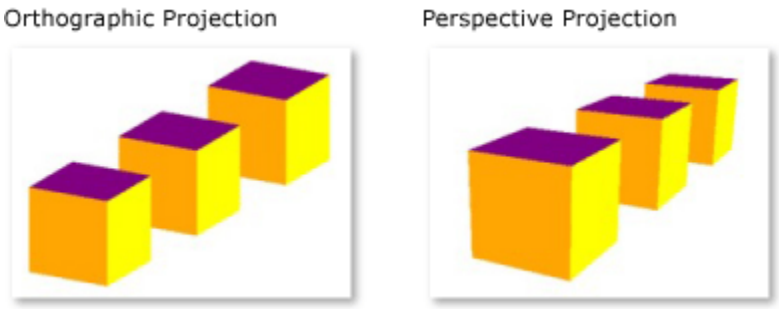
Camera

The camera is implemented first in this project because it is the way we see the world inside our game.

Our camera is affected by 6 factors:

1. Type

In this project we used Perspective Camera, instead of Orthographic Camera. Perspective Camera is how our eyes see the world.



Perspective Vs Orthographic Camera

Orthographic Camera will see all 3d objects in equal size regardless of zoom level.

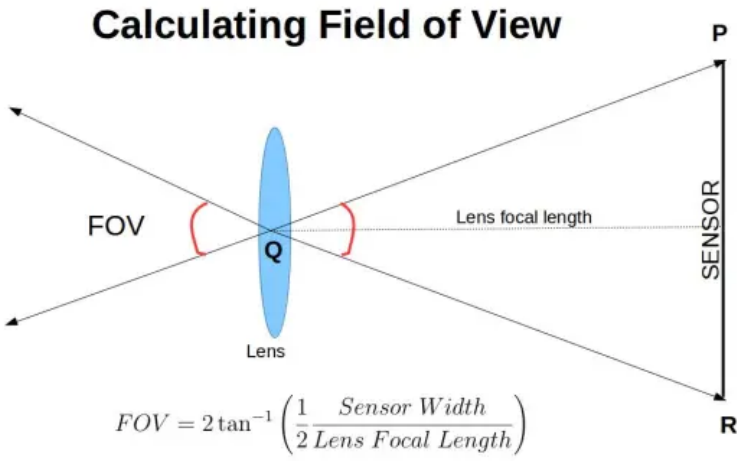
2. Position

Our camera position in our 3D world is determined by the coordinate (x, y, z).

3. Aspect Ratio

Reference: https://photographylife.com/aspect-ratio

4. Field Of View

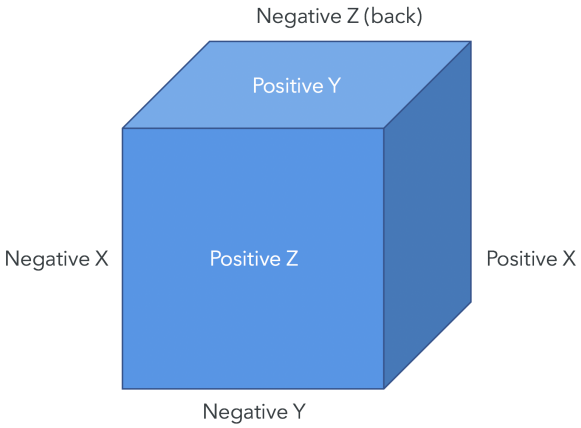


Visualization Of Field Of View

Field of view (FOV), in this project, is the open observable area a person can see through via a camera

Background With Sky Box

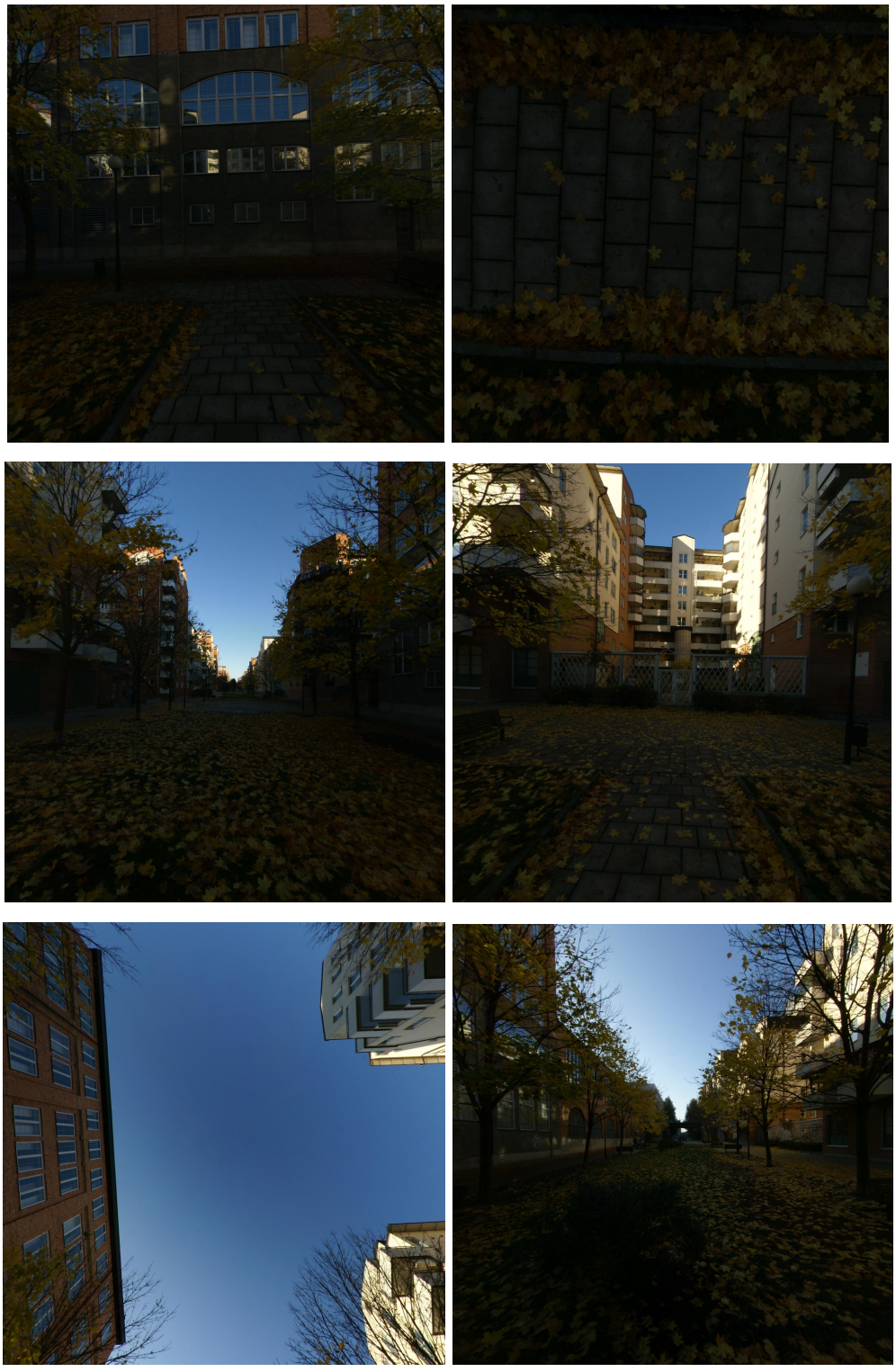
Using CubeTextureLoader, we were able to generate a static background for our application.



Sky Box With Cube Texture Implementation

In the image above, each face of the cube is an image. We require in total 6 images:

- Positive X: right face
- Negative X: left face
- Positive Y: upper face
- Negative Y: bottom
- Positive Z: forward face
- Negative Z: backward face



Images Used For Sky Box

```
const loader = new THREE.CubeTextureLoader();

const texture = loader.load([

        './assets/img/posx.jpg',

        './assets/img/negx.jpg',

        './assets/img/posy.jpg',

        './assets/img/negy.jpg',

        './assets/img/posz.jpg',

        './assets/img/negz.jpg',

    ]);

scene = new THREE.Scene();

scene.background = texture;
```

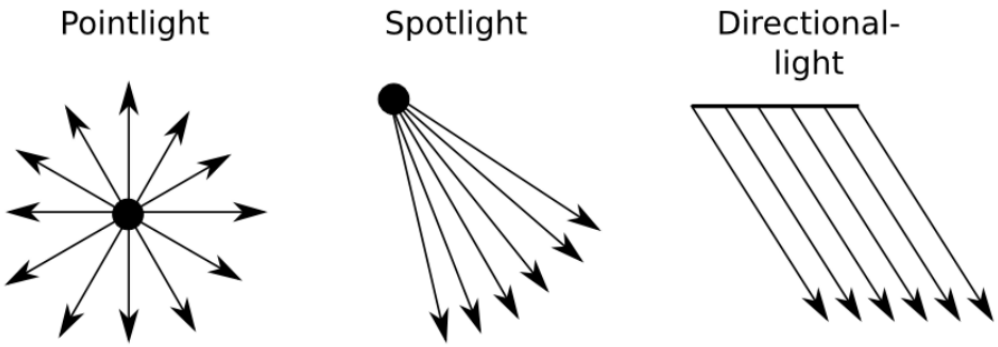
Javascript Implementation Of Sky Box

Shadow And Lighting

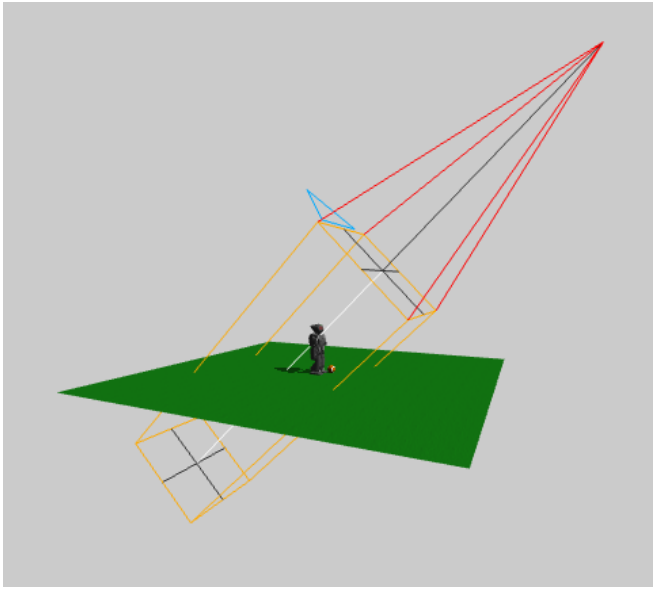
Shadow could be understood as the result of a lighting technique. In this project specifically, we applied the concept of Directional Light or light that comes from a source and shines in a straightway.

Other than Directional Light, we have Ambient Light and Hemisphere Light. Ambient Light globally illuminates all objects in the scene equally, while Hemisphere Light is positioned directly above the scene, with color fading from the sky color to the ground color.

Hence, to acquire shadow, we implemented Directional Light.



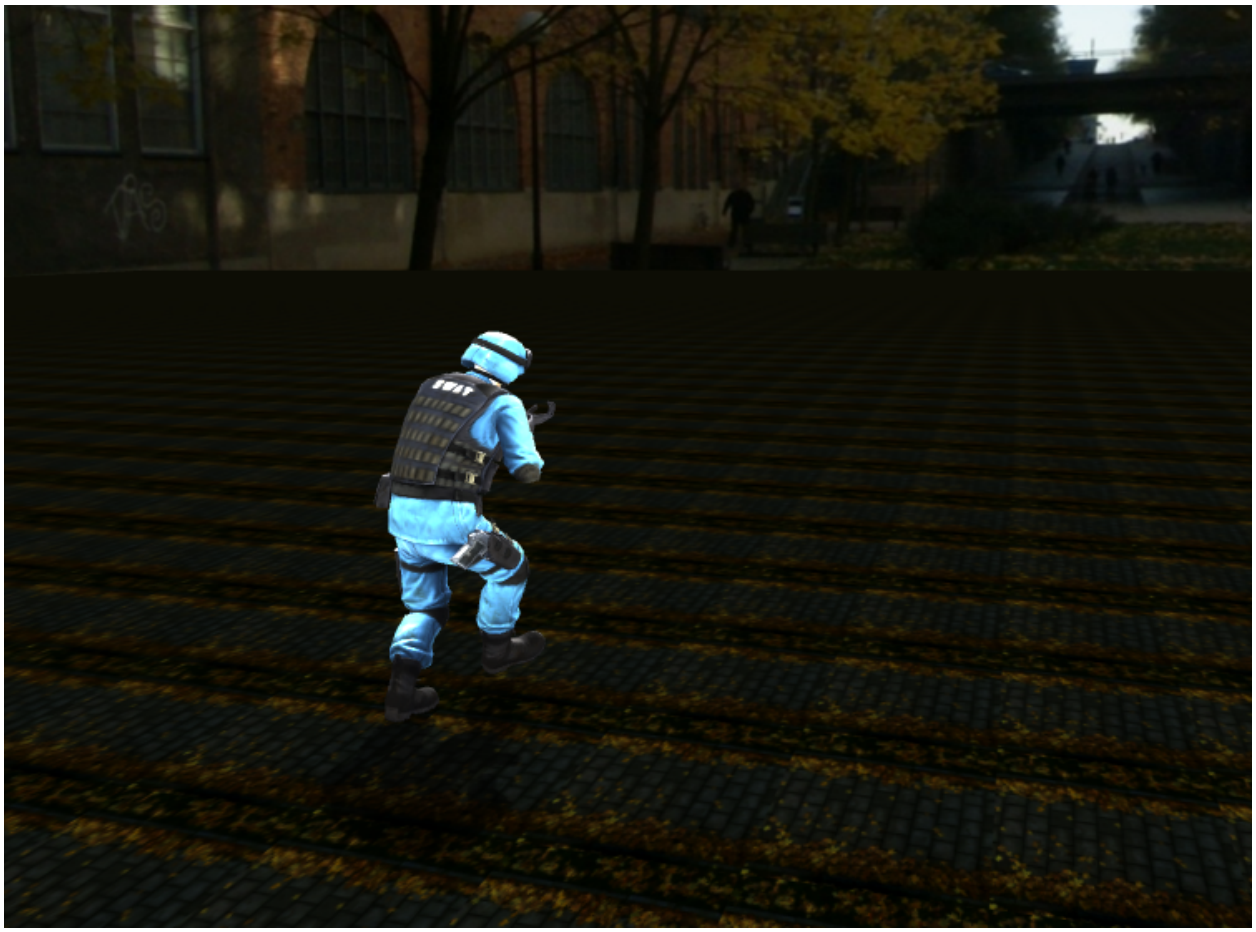
Lights In Three.js



Directional Light Visualization

```
    light = new THREE.DirectionalLight(0xFFFFFF, 10);
    light.position.set(20, 100, 100);
    light.target.position.set(0, 0, 0);
        // enable shadow map
    this.renderer = new THREE.WebGLRenderer({
        antialias: true,
        });
    renderer.shadowMap.enabled = true;
    renderer.shadowMap.type = THREE.PCFShadowMap; // shadow map type
        // enable shadow
    light.castShadow = true;
        // shadow map resolution
    light.shadow.mapSize.width = 10000;
    light.shadow.mapSize.height = 10000;
        // camera view for shadow map
    light.shadow.camera.near = 0.5;
    light.shadow.camera.far = 500;
    light.shadow.camera.left = 500;
    light.shadow.camera.right = -500;
    light.shadow.camera.top = 500;
    light.shadow.camera.bottom = -500;
    scene = new THREE.Scene();
    scene.add(light);
```

Shadow Implementation In Javascript



A Swat Soldier With A Trailing Shadow

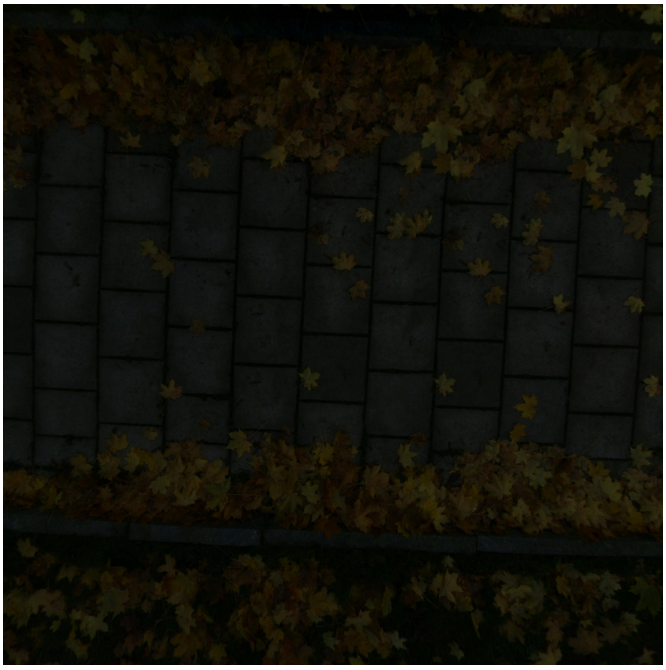
Control View Using Mouse - Orbit Control

We are able to use our mouse to control the view of our camera using OrbitControl. This feature allows us to change our line of sight freely.

```
const controls = new OrbitControls(camera, renderer.domElement);

camera.position.set( -20, 20, -100 );

controls.update();
```

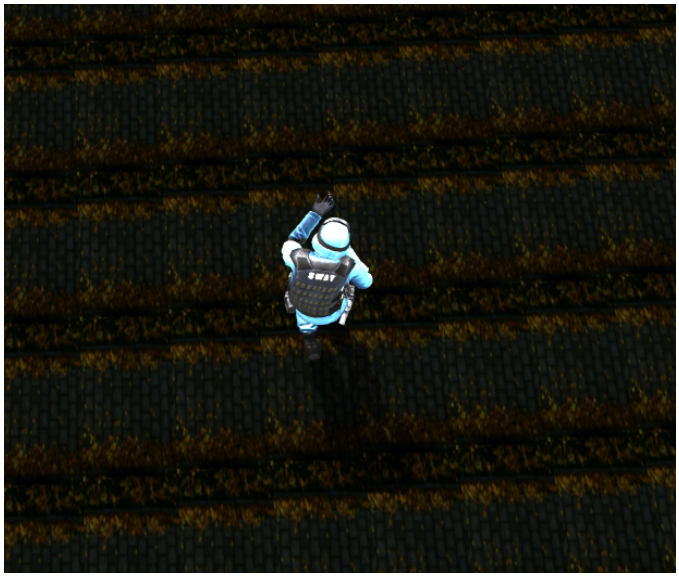
OrbitControl In Three.js

Ground Plane



Ground Texture

To lay the ground, we applied the Ground Texture above to our Mesh and introduced some Ambient Light to see the effect more clearly.



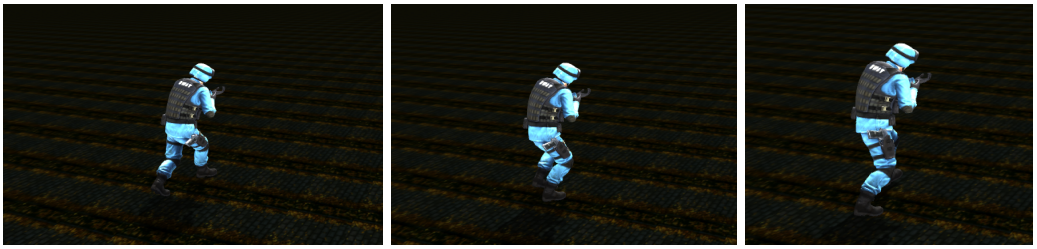
Ground Seen From Above

The texture is repeated in both directions: vertical and horizontal to generate the overall ground.

```
const ground_texture = new
THREE.TextureLoader().load("../assets/img/negy.jpg");

ground_texture.wrapS = THREE.RepeatWrapping;

ground_texture.wrapT = THREE.RepeatWrapping;

ground_texture.repeat.set( 100, 100 );

ground_texture.encoding = THREE.sRGBEncoding;

const ground_material = new THREE.MeshLambertMaterial( { map:
ground_texture } );

const mesh = new THREE.Mesh( new THREE.PlaneGeometry( 1000, 2000 ),
ground_material );

mesh.position.y = 0.0;

mesh.rotation.x = - Math.PI / 2;

mesh.receiveShadow = true;
```

Ground Implementation in Three.js

Load Animated Model



Animated Model

To load a model (our character) which is animated, in this project, firstly we acquired the model from www.mixamo.com in .fbx format, then added it to the scene using FbxLoader and AnimationMixer in Three.js.

FbxLoader loads fbx models in general, however, the model remains static. To render the animation, we implemented AnimationMixer, which is in short, an animation player.

```
let mixers = []; // mixer = animation player

const animation = new FBXLoader();

animation.setPath('./assets/fbx/');

animation.load('./characters/swat/Walk_Forward_inplace.fbx', (animation)
=> {

//theory:https://threejs.org/docs/?q=animatio#manual/en/introduction/Anima
tion-system

const mixer = new THREE.AnimationMixer(fbx);

mixers.push(mixer);

const action = mixer.clipAction(animation.animations[0]); // an animation
clip

action.play(); // play the model animation

});

// add model to scene

scene.add(fbx);
```

Loading Animated Model in Three.js

The animated model is divided into many clips, each will be played and updated continuously in a fixed interval time_elapsed_rescaled:

```
mixers.map(mixer => mixer.update(time_elapsed_rescaled));
```

where:

```
time_elapsed_rescaled = time_elapsed * 0.001
```

where:

```
time_elapsed = timestamp - previous_timestamp
```

where timestamp indicates the current time (based on the number of milliseconds since time origin (the beginning after the DOM finishes loading).

Character Control - Sync Keys With Movement

This is the most difficult feature to implement in our project by far. For our character to move in steps, we required physics concepts such as velocity, acceleration, deceleration and quaternion (used for rotation).

```
// on movement keys pressed

document.addEventListener('keydown', (e) => this.on_key_down(e), false);

document.addEventListener('keyup', (e) => this.on_key_up(e), false);
```

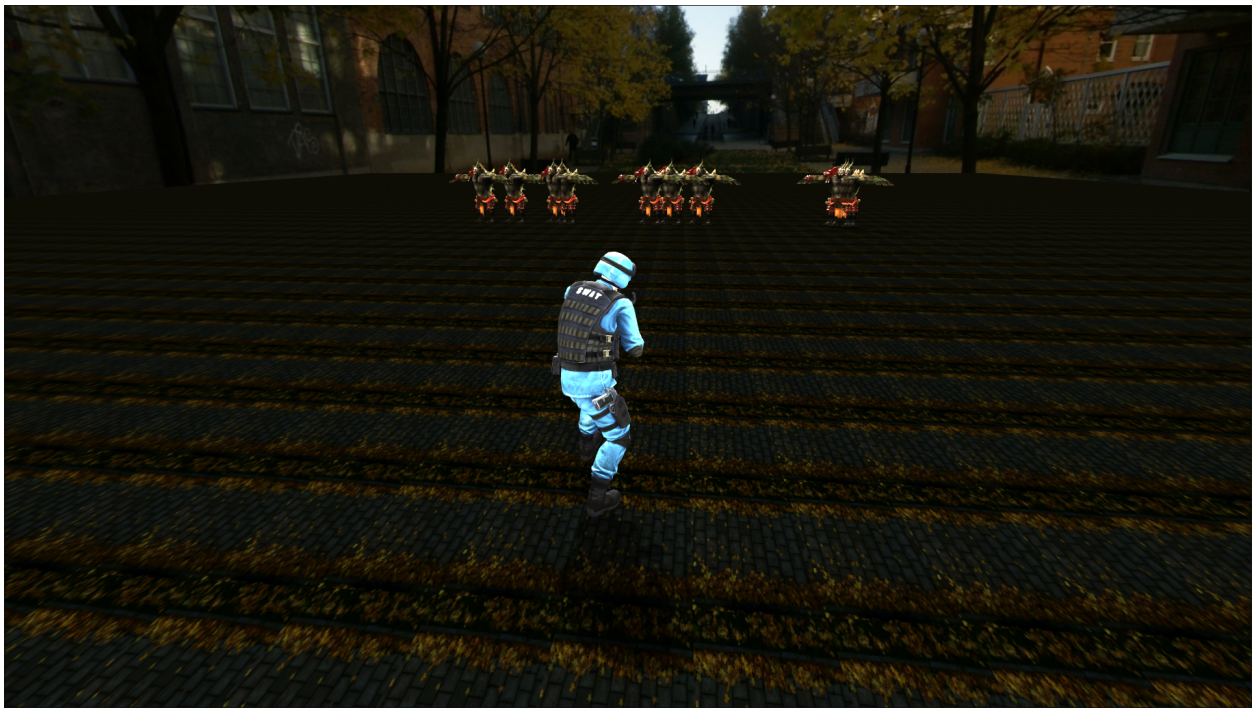
On Keys Pressed

The character movement is synchronized with the keyboard input, with W (up arrow) = forward, S (down arrow) = backward, A (left arrow) = to the left and D (right arrow) = to the right.

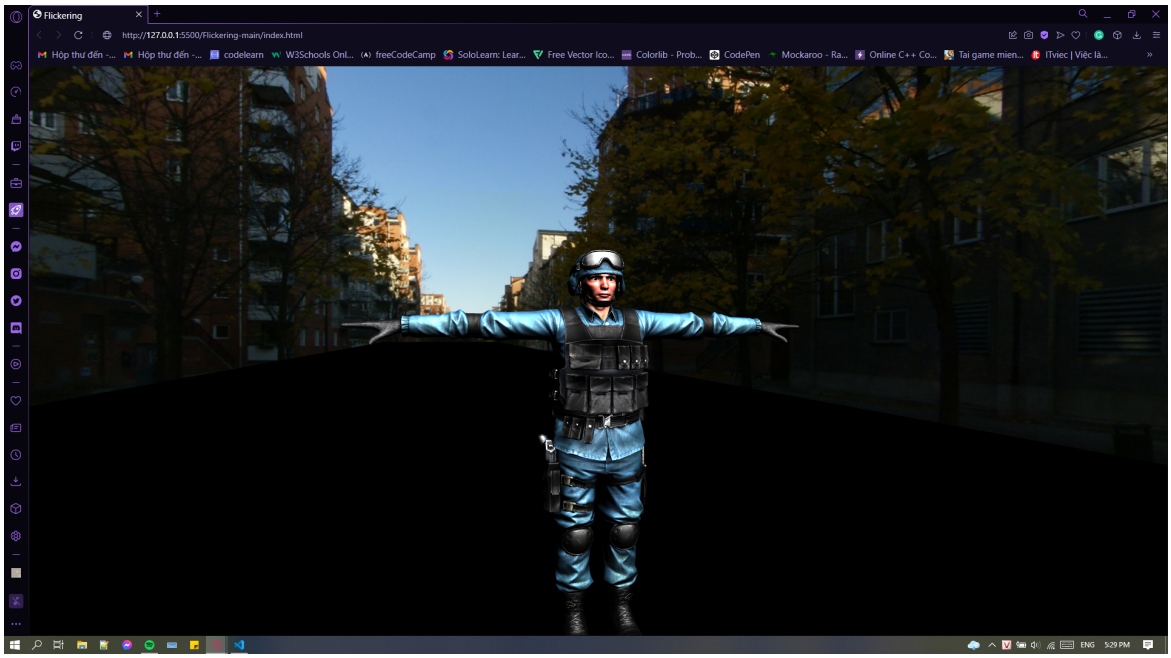
We declared a CharacterControl class with character_control instance being updated continuously as following:

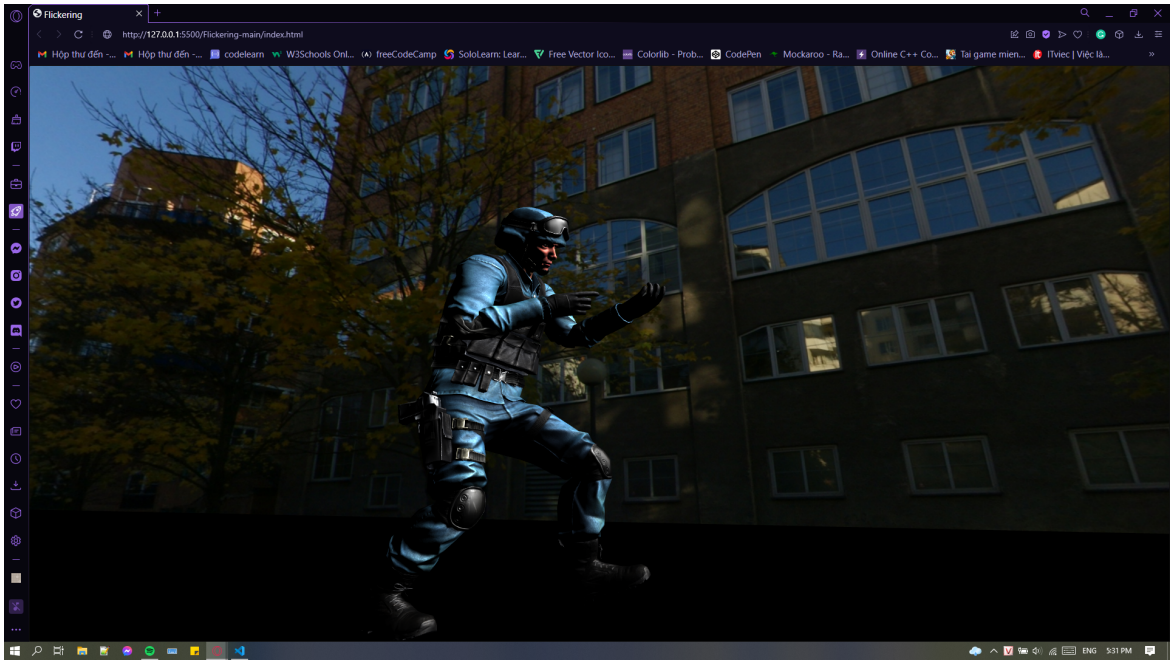
```
character_control.update(time_elapsed_rescaled);
```

III. Results



Loading Multiple Models



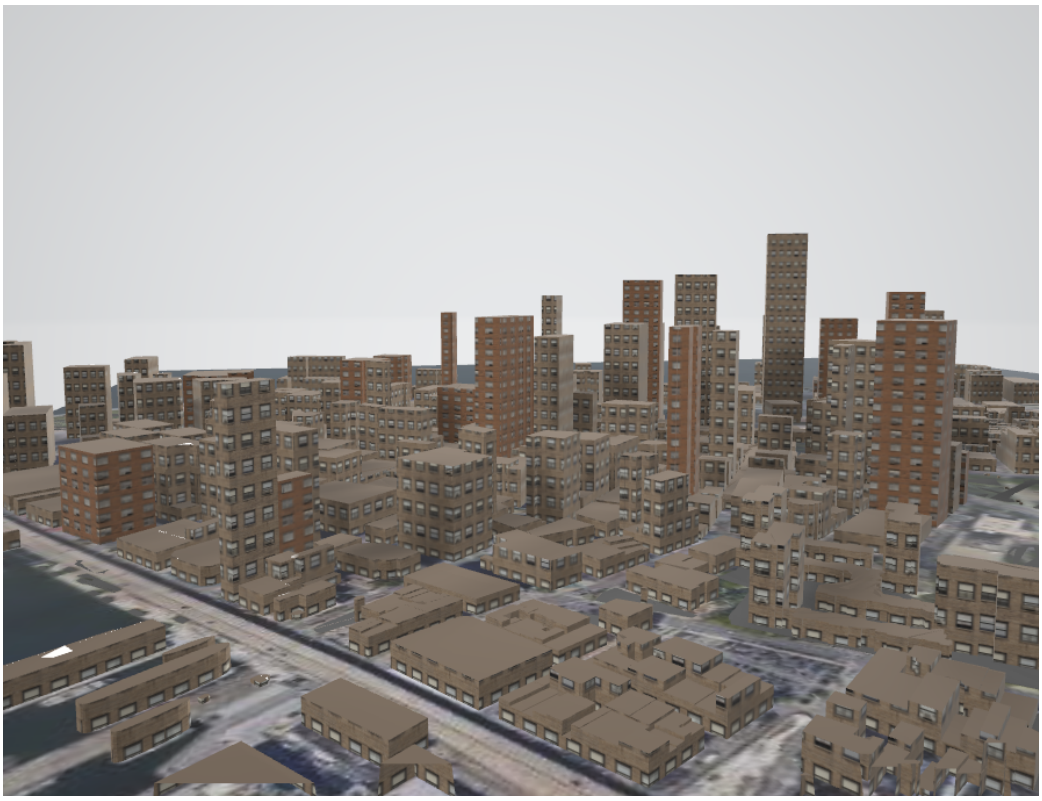


Because of time constraints, we haven't created a complete game. However, we have learnt to use three.js to create a basic game with control the character, control the view, add backgrounds and shadow.

IV. Future Improvements

- Third Person Camera that follows the character
- More interactivity with the environment
- 3D world generation
- Create a complete game

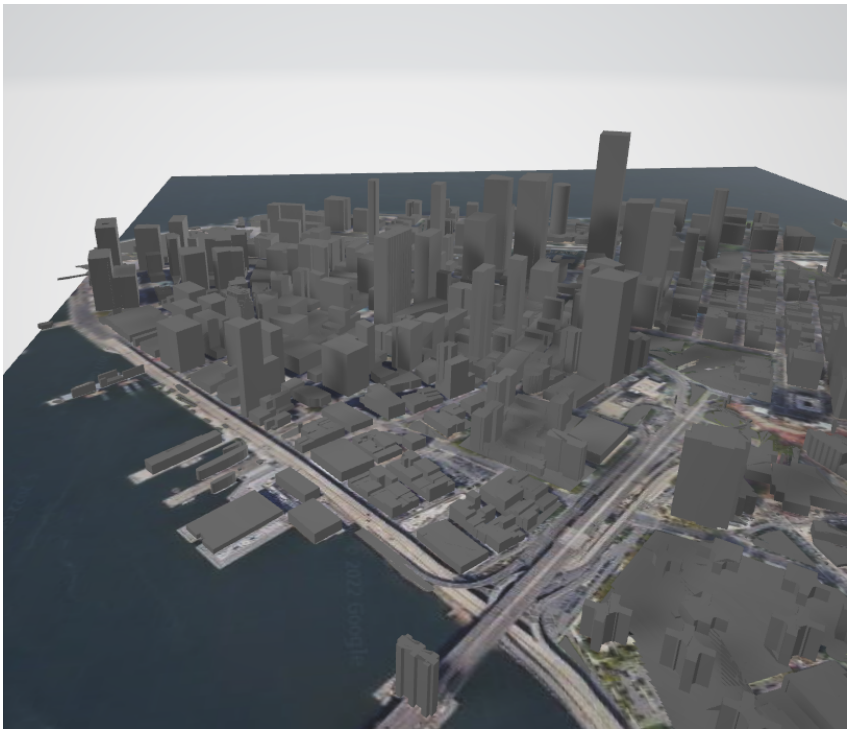
Originally, we were able to create a 3d city that looks like the following:



Soviet-styled City

However, when we tried to load the fbx model of the city above to our project, an unknown error related to the addon used to generate the city occurred.

Using Blender OSM, we obtained a 3D model of a city without any texturing.



New York City Model

We then applied the Soviet Styled textures of a building with Cube Projection down onto the 3D generated city.

Originally, our idea was to have our character move in such an environment then add some physics interactivity such as stop when in contact with the building.

Unfortunately, a lot of errors popped up to which we were unable to find a solution. Therefore, we went with Sky Box to somehow create a background.

V. References

Character Control

https://github.com/simondevyoutube/ThreeJS_Tutorial_CharacterController

VI. Source Code

https://github.com/vietanh2000april/Flickering

VII. Contribution

Lê Việt Anh: main coder

Nguyễn Tự Tùng: main report writer

Trần Bảo Huy: report and tester

Trần Hoàng Minh: presentation

Phạm Hoàng Việt: slide