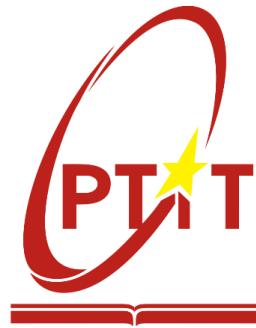


HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG  
**KHOA ĐA PHƯƠNG TIỆN**



**BÁO CÁO BÀI TẬP LỚN  
PHÁT TRIỂN ỨNG DỤNG THỰC TẠI ẢO**

Giảng viên hướng dẫn      **Nguyễn Thị Thanh Tâm**

Lớp                              **D19PTDPT**

Nhóm thực hiện              **03**

Sinh viên thực hiện:

Nguyễn Hoàng Anh          B19DCPT008

Nguyễn Thị Quỳnh Anh    B18DCPT012

Trần Trung Hiếu            B19DCPT088

Phạm Thị Hương            B19DCPT117

Nguyễn Thị Linh            B19DCPT140

**Hà Nội – 2023**

## MỤC LỤC

<b>1. Ý tưởng .....</b>	<b>3</b>
<b>2. Công nghệ sử dụng .....</b>	<b>3</b>
2.1. Webpack bundle.....	3
2.2. Ammojs, Bullet .....	3
2.2.1. Ammojs.....	3
2.2.2. Bullet.....	4
2.3. Threejs, Three pathfinding, setup môi trường thực thi .....	4
2.3.1. ThreeJs .....	4
2.3.2. Three pathfinding .....	4
2.3.3. Setup môi trường thực thi .....	5
2.4. OOP .....	9
2.5. MVVM.....	9
2.6. Finite State Machine.....	10
2.7. None player controller .....	11
2.8. Player controller .....	12
<b>3. Sản phẩm .....</b>	<b>12</b>
3.1. Game Play .....	12
3.2. User control.....	13
3.3. Model controller (player, mutant).....	15
3.3.1. Player .....	15
3.3.2. Mutant.....	23
3.3.3. AK47.....	37
3.4. Setup environment, audio effect .....	45
<b>4. Demo (video demo).....</b>	<b>50</b>
<b>5. Đánh giá nghiệm thu .....</b>	<b>50</b>
5.1. Ưu & nhược điểm.....	50
5.2. Khả năng thích ứng .....	50
5.3. Mức độ trải nghiệm .....	50
<b>6. Hướng phát triển .....</b>	<b>51</b>
<b>7. Tài liệu tham khảo.....</b>	<b>51</b>
<b>8. Phân chia công việc .....</b>	<b>52</b>

## 1. Ý tưởng

Xây dựng một website chơi game FPS online. Game cho phép người chơi nhập vai vào một tay súng. Nhiệm vụ của thay súng là tìm kiếm tiêu diệt quái vật và thu thập vật phẩm để gia tăng khả năng chiến đấu.

## 2. Công nghệ sử dụng

### 2.1. Webpack bundle

- Webpack là một công cụ đóng gói (bundler) mã nguồn JavaScript. Khi phát triển một ứng dụng web, chúng ta thường sử dụng nhiều tệp JavaScript và thư viện để xây dựng các tính năng. Tuy nhiên, khi chạy ứng dụng trong trình duyệt, trình duyệt phải tải nhiều tệp JavaScript khác nhau, điều này có thể làm cho ứng dụng chậm hơn và tốn nhiều băng thông hơn.
- Để giải quyết vấn đề này, ta sử dụng Webpack để đóng gói (bundle) các tệp JavaScript và thư viện vào một tệp duy nhất. Tệp bundle được tạo ra bằng cách trích xuất tất cả các phụ thuộc của ứng dụng (dependencies) và kết hợp chúng vào một tệp duy nhất.
- Webpack bundle cũng có thể được tối ưu hóa để giảm kích thước và tốc độ tải của tệp. Ví dụ, Webpack có thể loại bỏ mã không sử dụng trong tệp bundle hoặc nén tệp để giảm kích thước.
- Khi ứng dụng được đóng gói bằng Webpack, chỉ cần tải một tệp bundle duy nhất, điều này giúp cải thiện hiệu suất của ứng dụng và giảm tải cho máy chủ.

### 2.2. Ammojs, Bullet

#### 2.2.1. Ammojs

- Ammo.js là một thư viện vật lý JavaScript mã nguồn mở, cho phép bạn tích hợp tính năng vật lý vào ứng dụng web của mình. Nó là một phần của dự án Emscripten, cho phép biên dịch các thư viện C++ thành mã JavaScript, cho phép chúng ta chạy các ứng dụng C++ trong môi trường trình duyệt web.
- Ammo.js cung cấp các tính năng vật lý chuyên nghiệp, bao gồm va chạm giữa các đối tượng, động lực học, quán tính, ma sát, va chạm và bám dính, và nhiều hơn nữa. Với Ammo.js, bạn có thể tạo ra các ứng dụng web phức tạp như trò chơi, mô phỏng vật lý, và ứng dụng AR / VR.
- Ammo.js là một thư viện đáng tin cậy và được sử dụng rộng rãi trong cộng đồng phát triển web. Nó cũng được sử dụng trong các thư viện và framework khác như A-Frame, Three.js và Babylon.js để tạo ra các ứng dụng và trò chơi web với tính năng vật lý chuyên nghiệp.

### **2.2.2. Bullet**

- Bullet là một thư viện vật lý mã nguồn mở, cung cấp các tính năng mô phỏng vật lý như va chạm giữa các đối tượng, động lực học, quán tính, ma sát và các tính năng khác cho các ứng dụng và trò chơi 3D. Bullet được phát triển bởi Erwin Coumans, và được sử dụng trong nhiều ứng dụng thực tế như game, phần mềm thiết kế, và phần mềm mô phỏng.
- Bullet có thể tích hợp với các framework và thư viện khác như OpenGL, DirectX, OpenCL và CUDA, cho phép bạn tạo ra các ứng dụng vật lý mạnh mẽ và đa nền tảng. Nó cũng hỗ trợ nhiều hình thức va chạm, bao gồm hình hộp, hình cầu, hình trụ và các hình khối khác.
- Bullet là một thư viện đáng tin cậy và được sử dụng rộng rãi trong cộng đồng phát triển game và mô phỏng. Nó được sử dụng trong nhiều trò chơi nổi tiếng như Grand Theft Auto V, Battlefield 3 và 4, và cả những tựa game indie như Kerbal Space Program và Gang Beasts.

## **2.3. Threejs, Three pathfinding, setup môi trường thực thi**

### **2.3.1. ThreeJs**

- Three.js là một thư viện JavaScript mã nguồn mở, được sử dụng để tạo ra các ứng dụng và trò chơi 3D trên web. Nó cung cấp cho người dùng các công cụ và tính năng để tạo ra các đối tượng 3D, ánh sáng, động cơ vật lý và các hiệu ứng khác để tạo ra các ứng dụng và trò chơi tương tác trên trình duyệt web.
- Three.js được phát triển bởi Ricardo Cabello, và được phát hành dưới giấy phép MIT. Nó cung cấp một bộ các công cụ tạo hình học 3D, với hỗ trợ các định dạng file 3D như OBJ, FBX, Collada và GLTF. Nó cũng hỗ trợ các chức năng để xử lý ánh sáng, vật lý và các hiệu ứng khác.
- Three.js có thể tích hợp với các thư viện và framework khác như Ammo.js, Physijs và Cannon.js để tạo ra các ứng dụng và trò chơi 3D đầy đủ tính năng trên web. Nó được sử dụng rộng rãi trong cộng đồng phát triển web và game để tạo ra các ứng dụng và trò chơi 3D đẹp mắt và chất lượng cao.

### **2.3.2. Three pathfinding**

- Theo truyền thống, các trò chơi và ứng dụng 3D đã sử dụng các điểm tham chiếu để giúp các tác nhân AI của họ điều hướng. Điều này tệ và có nhiều vấn đề, nhưng nói chung là dễ thực hiện hơn so với lối đi theo hướng. Các lối đi theo hướng chính xác hơn, nhanh hơn và có tính đến kích thước của tác nhân AI (ví dụ: xe tăng cần không gian di chuyển để cơ động hơn so với binh lính).

- Three.js Pathfinding là một thư viện cho phép tìm đường trong môi trường 3D, được xây dựng trên nền tảng Three.js - một thư viện JavaScript cho phép tạo ra các đối tượng 3D trên web. Thư viện này cung cấp các thuật toán pathfinding như A\*, Jump Point Search và Recast, cho phép các đối tượng trong game di chuyển thông minh, tránh các vật thể và tìm lối đi tối ưu trong môi trường 3D.
- Three Pathfinding hỗ trợ các tính năng chính sau:
  - + Tìm đường thông minh: Thư viện cung cấp các thuật toán pathfinding như A\*, Jump Point Search và Recast, cho phép các đối tượng trong game di chuyển thông minh, tránh các vật thể và tìm lối đi tối ưu trong môi trường 3D.
  - + Tích hợp với Three.js: Thư viện được xây dựng trên nền tảng Three.js, cho phép tích hợp dễ dàng với các đối tượng 3D trong trò chơi.
  - + Tùy chỉnh linh hoạt: Thư viện cho phép tùy chỉnh các thuật toán pathfinding và các tham số khác để phù hợp với yêu cầu của trò chơi.
  - + Hỗ trợ đa nền tảng: Thư viện có thể chạy trên nhiều nền tảng khác nhau, bao gồm web, desktop và mobile.
  - + Tính tương thích cao: Three Pathfinding tương thích với nhiều trình duyệt web phổ biến và hỗ trợ các phiên bản Three.js từ 0.90 đến 0.130. documentation: <https://github.com/donmccurdy/three-pathfinding> demo: <https://three-pathfinding.donmccurdy.com/teleport.html>

### 2.3.3. Setup môi trường thực thi

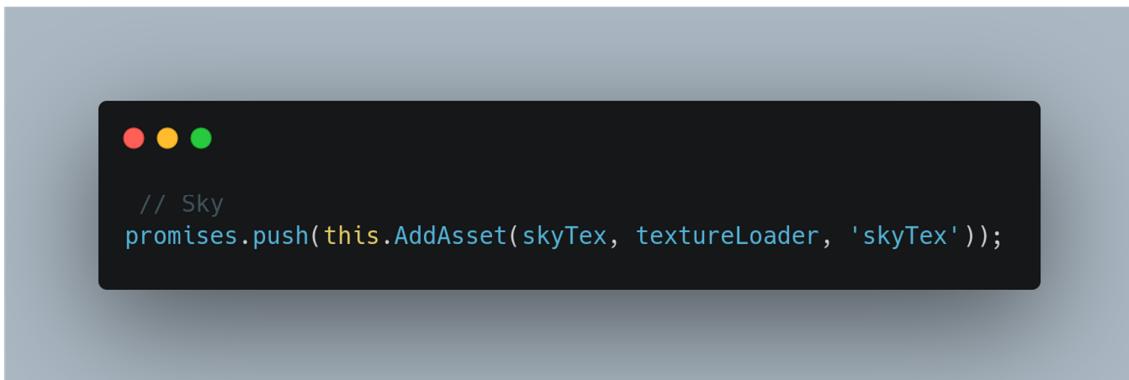
Việc setup môi trường thực thi trong game rất quan trọng bao gồm các thành phần như camera, audio config, và resize render để đảm bảo trải nghiệm chơi game được tốt nhất. Một số thủ thuật để setup môi trường thực thi trong game:

- + Camera: Camera là thành phần quan trọng trong game, cho phép người chơi nhìn thấy môi trường và di chuyển trong game. Để setup camera, bạn có thể sử dụng các thuộc tính như position, rotation và field of view để định vị camera và điều chỉnh góc nhìn của nó. Ngoài ra, bạn cũng nên cân nhắc sử dụng các thuật toán định tuyến để camera di chuyển một cách tự động và mượt mà hơn.
- + Audio config : Âm thanh là một phần quan trọng trong trải nghiệm chơi game. Để setup audio config, bạn có thể sử dụng các thư viện audio như Howler.js hoặc Web Audio API để tải và phát các file âm thanh. Ngoài ra, bạn cũng nên cân nhắc sử dụng các hiệu ứng âm thanh như reverb hoặc echo để làm tăng tính tương tác và chân thực hóa trải nghiệm chơi game.

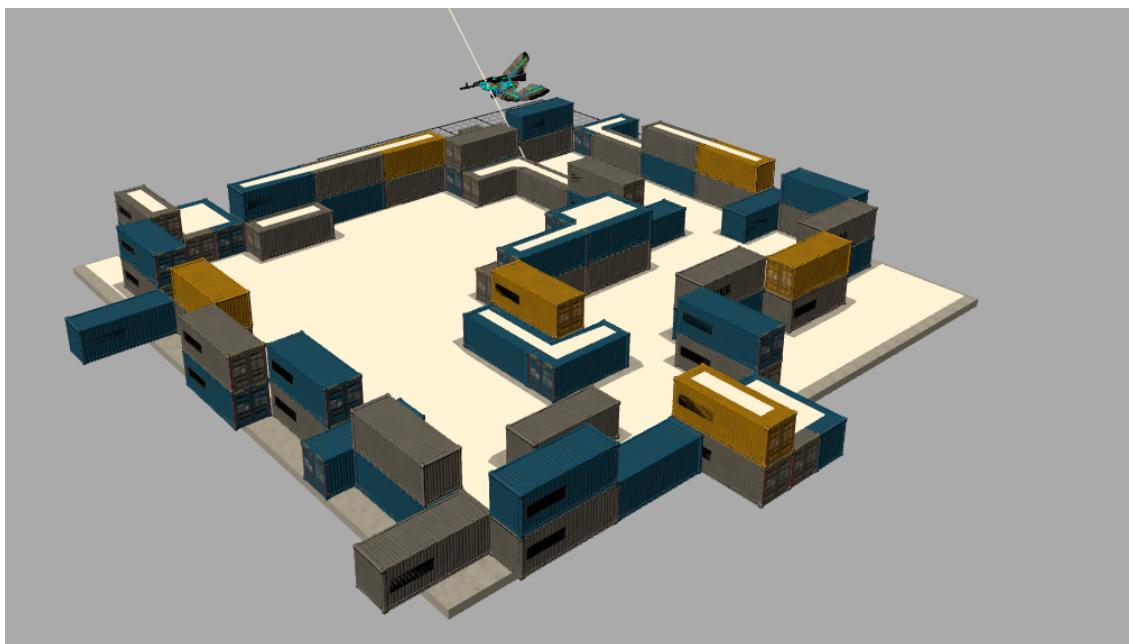
- + Resize render: Việc resize render là cần thiết để đảm bảo trò chơi có thể chạy được trên nhiều kích thước màn hình khác nhau. Để setup resize render, bạn có thể sử dụng các thuộc tính như width và height để định vị kích thước màn hình và điều chỉnh tỷ lệ khung hình để phù hợp với màn hình. Ngoài ra, bạn cũng nên cân nhắc sử dụng các thuật toán tự động điều chỉnh kích thước màn hình để đảm bảo trò chơi luôn chạy mượt mà và không bị giật lag.
- Cấu hình môi trường bằng chất liệu (texture) hoặc bằng model 3D. Điều này dễ dàng thực hiện triển khai trên nền tảng website khi có sự hỗ trợ của thư viện *Three.js*.



- + Từ một hình quang cảnh bầu trời như trên ta có thể dễ dàng trai chất liệu hình ảnh này vào môi trường 3D.



- + Ngoài ra ta cũng có thể sử dụng hàm *GLTFLoader* của *Threejs* để tải các file model có định dạng .gltf



- Về audio, bản thân các trình duyệt cũng đã hỗ trợ ta cách tải một file audio, đặc biệt là môi trường có sử dụng WebGL (một API JavaScript để hiển thị đồ họa 2D và 3D tương tác trong bất kỳ trình duyệt web nào)

```
● ○ ●

this.scene = new THREE.Scene();
this.renderer = new THREE.WebGLRenderer({ antialias: true });
this.renderer.shadowMap.enabled = true;
this.renderer.shadowMap.type = THREE.PCFSoftShadowMap;

this.renderer.toneMapping = THREE.ReinhardToneMapping;
this.renderer.toneMappingExposure = 1;
this.renderer.outputEncoding = THREE.sRGBEncoding;

this.camera = new THREE.PerspectiveCamera();
this.camera.near = 0.01;

// create an AudioListener and add it to the camera
this.listener = new THREE.AudioListener();
this.camera.add(this.listener);
```

Ngoài ra ta có thể sử dụng một vài tính năng thích ứng trên các thiết bị di động để tăng khả năng hiển thị của ứng dụng, đồng thời giúp cho hình ảnh hiển thị trở nên mượt mà hơn.

```
● ○ ●

// renderer
this.renderer.setPixelRatio(window.devicePixelRatio);

// handle window resize
this.WindowResizeHandler();
window.addEventListener('resize', this.WindowResizeHandler);

// resize
WindowResizeHandler = () => {
    const { innerHeight, innerWidth } = window;
    this.renderer.setSize(innerWidth, innerHeight);
    this.camera.aspect = innerWidth / innerHeight;
    this.camera.updateProjectionMatrix();
};
```

## 2.4. OOP

- OOP là viết tắt của Object-Oriented Programming (Lập trình hướng đối tượng), một phương pháp lập trình được sử dụng rộng rãi trong việc phát triển các trò chơi.
- Trong OOP, chương trình được xây dựng dựa trên các đối tượng (objects) và các lớp (classes) của chúng. Mỗi đối tượng có thuộc tính (attributes) và phương thức (methods) riêng, giúp cho việc quản lý mã nguồn trở nên dễ dàng hơn và cho phép phát triển các tính năng phức tạp một cách hiệu quả.
- Ví dụ, trong việc phát triển một trò chơi, ta có thể tạo ra các đối tượng như nhân vật, vật phẩm, môi trường và các quái vật. Các đối tượng này có thể được xây dựng bằng các lớp tương ứng, và mỗi lớp sẽ định nghĩa các thuộc tính và phương thức đặc trưng cho đối tượng đó.
- OOP giúp cho việc phát triển trò chơi trở nên dễ dàng hơn, cho phép tái sử dụng mã nguồn và giảm thiểu lỗi trong quá trình phát triển.
- Ngoài ra, OOP cũng cho phép tách biệt giữa dữ liệu và mã nguồn, giúp cho việc bảo trì và nâng cấp trở nên đơn giản hơn. Tính kế thừa (inheritance) và đa hình (polymorphism) là những tính năng quan trọng của OOP, giúp cho việc phát triển và mở rộng các tính năng của trò chơi trở nên dễ dàng hơn.
- Kế thừa cho phép tạo ra các lớp mới dựa trên các lớp đã có sẵn, thừa hưởng các thuộc tính và phương thức của lớp cha, giúp cho việc phát triển nhanh chóng và hiệu quả hơn. Đa hình cho phép các đối tượng cùng kiểu có thể có các hành vi khác nhau, giúp cho trò chơi trở nên đa dạng và phong phú hơn.

## 2.5. MVVM

- MVVM là viết tắt của Model-View-ViewModel, một kiến trúc phần mềm phổ biến trong phát triển ứng dụng và cũng có thể được áp dụng trong phát triển game.
- Trong kiến trúc MVVM mỗi thành phần của ứng dụng được phân chia thành ba phần chính:
  - + *Model*: đại diện cho các đối tượng và dữ liệu của ứng dụng.
  - + *View*: đại diện cho giao diện người dùng và hiển thị dữ liệu cho người dùng.
  - + *ViewModel*: đại diện cho logic xử lý dữ liệu và tương tác giữa Model và View.
- Trong game, các thành phần tương tự cũng có thể được phân chia vào ba phần tương ứng:
  - + *Model* : đại diện cho các đối tượng trong game, bao gồm nhân vật, vật phẩm, môi trường, quái vật, v.v.

- + View : đại diện cho giao diện người chơi và hiển thị các đối tượng trong game.
- + ViewModel : đại diện cho logic xử lý dữ liệu của các đối tượng trong game và tương tác giữa Model và View.
- ViewModel trong game có thể đảm nhận nhiều vai trò khác nhau, bao gồm:
  - + Quản lý trạng thái của các đối tượng trong game
  - + Xử lý sự kiện và tương tác của các đối tượng trong game
  - + Xử lý các nhiệm vụ và sự kiện trong game
  - + Quản lý các thông tin và cấu hình của game
- MVVM trong game giúp cho việc phát triển game trở nên dễ dàng hơn, cho phép tái sử dụng mã nguồn và giảm thiểu lỗi trong quá trình phát triển. Kiến trúc MVVM cũng giúp cho việc tách biệt giữa các thành phần của game, giúp cho việc bảo trì và nâng cấp trở nên đơn giản hơn.
- Để xây dựng MVVM trong game, chúng ta cần thực hiện các bước sau:
  - + Xác định các thành phần chính của game và phân chia chúng vào ba phần tương ứng: View, Model, ViewModel.
  - + Định nghĩa các lớp và đối tượng trong Model, bao gồm các lớp đại diện cho nhân vật, vật phẩm, môi trường quái vật,...
  - + Định nghĩa các giao diện người chơi và các thành phần hiển thị trong View, bao gồm các giao diện đại diện cho bản đồ, nhân vật, vật phẩm, các menu và công cụ khác.
  - + Định nghĩa các lớp và đối tượng trong ViewModel, bao gồm các lớp đại diện cho quản lý trạng thái của đối tượng trong game, và quản lý các thông tin và cấu hình game.
  - + Kết nối các thành phần của game với nhau bằng cách sử dụng các phương thức và thuộc tính của ViewModel để điều khiển các đối tượng trong Model và hiển thị chúng trong View.
  - + Kiểm tra và đánh giá hiệu năng của game bằng cách sử dụng các công cụ và kỹ thuật phù hợp.

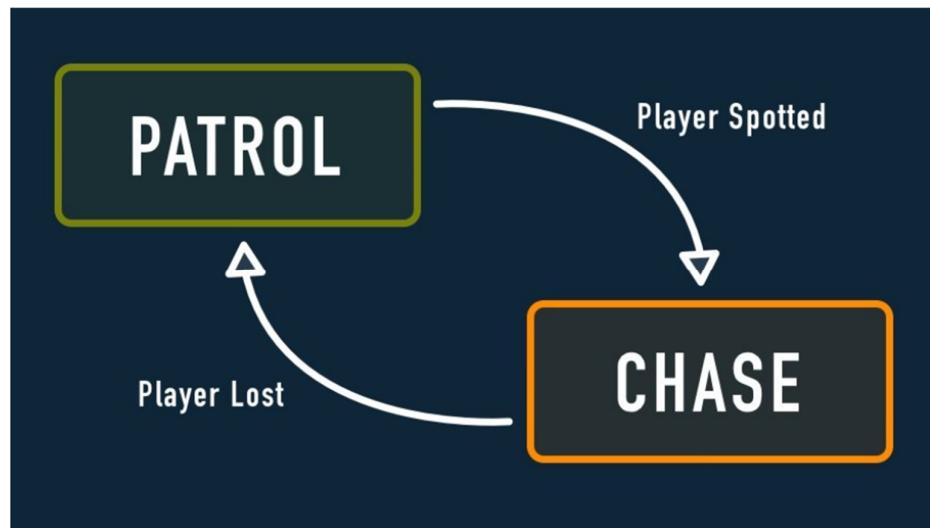
## 2.6. Finite State Machine

Mô hình State Machine là một trong những mô hình truyền thống trong khoa học máy tính - mô hình tính toán dựa trên một cỗ máy giả định được tạo thành từ một hoặc nhiều trạng thái. Chỉ một trạng thái duy nhất có thể hoạt động cùng một lúc, các trạng thái này có thể được sử dụng để quản lý các hành vi của đối tượng, cho phép nó chuyển đổi giữa các trạng thái khác nhau một cách dễ dàng.

Lợi ích của điều này là cho phép tập trung vào logic liên quan đến tình huống hiện tại trong bất kỳ thời điểm nào và chỉ trả về các điều kiện có thể làm thay đổi

trạng thái đó. Điều này làm cho việc làm việc với đối tượng dễ dàng hơn tùy thuộc vào hành vi của đối tượng.

Ví dụ, trong một trò chơi, có thể sử dụng FSM để quản lý hành vi của một nhân vật đối thủ. Nhân vật này có hai trạng thái khác nhau: Trạng thái tuần tra và Trạng thái đuổi theo. Khi nhân vật đang tuần tra, nó sẽ di chuyển xung quanh và tìm kiếm bất kỳ đối tượng nào để tấn công. Tuy nhiên, khi nhân vật phát hiện ra một đối tượng người chơi, nó sẽ chuyển sang trạng thái đuổi theo và tấn công người chơi.



FSM lập trình game có thể được thực hiện bằng nhiều cách khác nhau, bao gồm viết mã script hoặc sử dụng các công cụ hỗ trợ. Tuy nhiên, nó thường được sử dụng trong các trò chơi để quản lý hành vi của đối tượng và cải thiện trải nghiệm của người chơi.

## 2.7. None player controller

Non-player controller (NPC) là một thuật ngữ được sử dụng trong lĩnh vực lập trình game để chỉ các đối tượng không phải là người chơi (non-player characters) được điều khiển bởi máy tính. Đây là các đối tượng mà người chơi không thể điều khiển trực tiếp, mà chúng thường được sử dụng để cung cấp các yếu tố bổ sung cho trò chơi, chẳng hạn như các nhân vật phụ, địa hình, hoặc các quái vật.

NPCs có thể có nhiều hình dạng và kích cỡ khác nhau, từ những con thú hoang dã trong một trò chơi phiêu lưu đến các nhân vật phụ trong một trò chơi nhập vai (RPG). Chúng có thể được thiết kế để thực hiện các hành động cụ thể trong trò chơi, như tấn công hoặc di chuyển, hoặc đơn giản chỉ là các đối tượng tĩnh để tạo ra một cảnh quan sống động. Vì vậy, các non-player controller đóng một vai trò quan trọng trong việc xây dựng một thế giới ảo đa dạng và phong phú cho người chơi khám phá.

Để tạo ra NPCs trong một trò chơi, cần phải tạo một non-player controller (NPC controller) để điều khiển hành động của nhân vật. Các bước cơ bản để tạo NPCs bao gồm:

- Thiết kế nhân vật: Cần xác định hình dạng và tính năng của nhân vật, bao gồm cách nó di chuyển, tấn công và nói chuyện.
- Tạo mô hình: Cần tạo một mô hình 3D hoặc 2D của nhân vật, sử dụng các phần mềm như Blender, Maya hoặc Photoshop.
- Lập trình hành động: Sử dụng một ngôn ngữ lập trình như C++, C# hoặc Python để viết các hàm và đoạn mã để điều khiển hành động của nhân vật. Ví dụ tạo một hàm để đưa nhân vật di chuyển từ điểm này đến điểm khác trong trò chơi.
- Tạo non-player controller: tạo non-player controller (NPC controller) để kết nối giữa mô hình và lập trình hành động. NPC controller sẽ điều khiển hành động của nhân vật dựa trên các tương tác của người chơi và trạng thái hiện tại của trò chơi.

## 2.8. Player controller

Player controller là một thành phần trong lập trình game, nó giúp điều khiển và quản lý hành động của nhân vật người chơi (player character) trong trò chơi. Player controller đảm nhiệm việc xử lý các input như điều khiển di chuyển, tấn công, nhảy, và các hành động khác của nhân vật người chơi. Nó cũng có thể xử lý các sự kiện và tương tác với các đối tượng trong trò chơi, như việc nhặt vật phẩm hay mở cửa.

Player controller thường được thiết kế để phù hợp với phong cách gameplay của trò chơi và các tính năng của nhân vật người chơi. Nó có thể được xây dựng bằng cách sử dụng một số công cụ lập trình khác nhau, chẳng hạn như Unity hoặc Unreal Engine

## 3. Sản phẩm

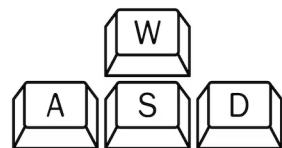
### 3.1. Game Play

- Người chơi sẽ nhập vai vào một tay bắn súng và đi tiêu diệt những con quái đã được lập trình sẵn trong game. Ngoài bắn súng để tiêu diệt quái, người chơi cần phải khéo léo tránh những đòn tấn công của những con quái để duy trì năng lượng. Nếu hết năng lượng, người chơi sẽ thua cuộc.
- Người chơi sẽ có tối đa 120 viên đạn, mỗi lần bắn sẽ bắn ra một viên đạn. Nếu người chơi diệt được hết quái mà vẫn còn đạn thì người chơi giành được chiến thắng.
- Người chơi sử dụng các phím trên bàn phím

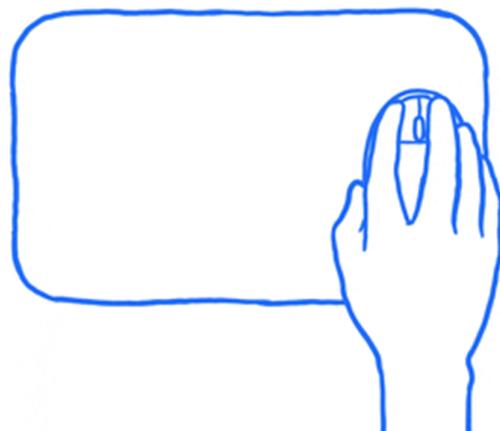
- + W: di chuyển về phía trước
- + A: di chuyển sang trái
- + D: di chuyển sang phải
- + S: di chuyển sang xuống dưới
- + Phím “ Space”: để nhảy lên
- + R: nạp đạn
- + Click chuột: bắn súng
- + Di chuột: thay đổi góc độ của camera

### 3.2. User control

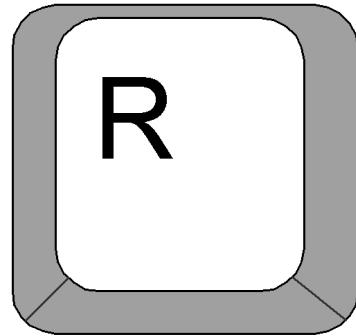
- Di chuyển nhân vật sử dụng các phím **W, A, S, D**



- Di chuyển góc nhìn (camera), hướng ngắm bắn sử dụng chuột



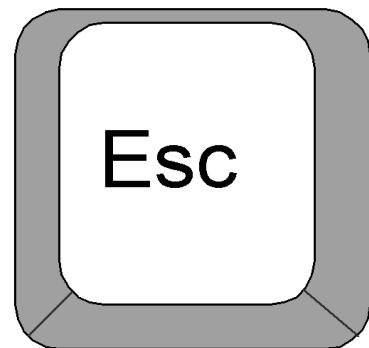
- Nạp đạn sử dụng phím **R**



- Điều khiển nhân vật nhảy lên sử dụng phím **Space**



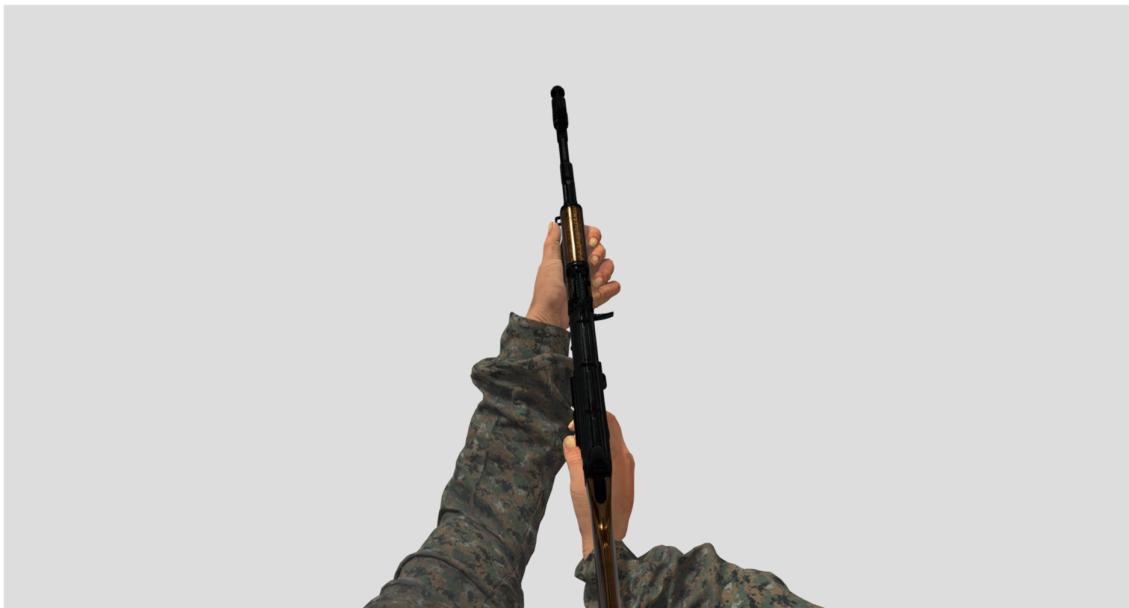
- Hiển thị con trỏ nhấn phím **Esc**



### **3.3. Model controller (player, mutant)**

#### **3.3.1. Player**

##### **a) Model**

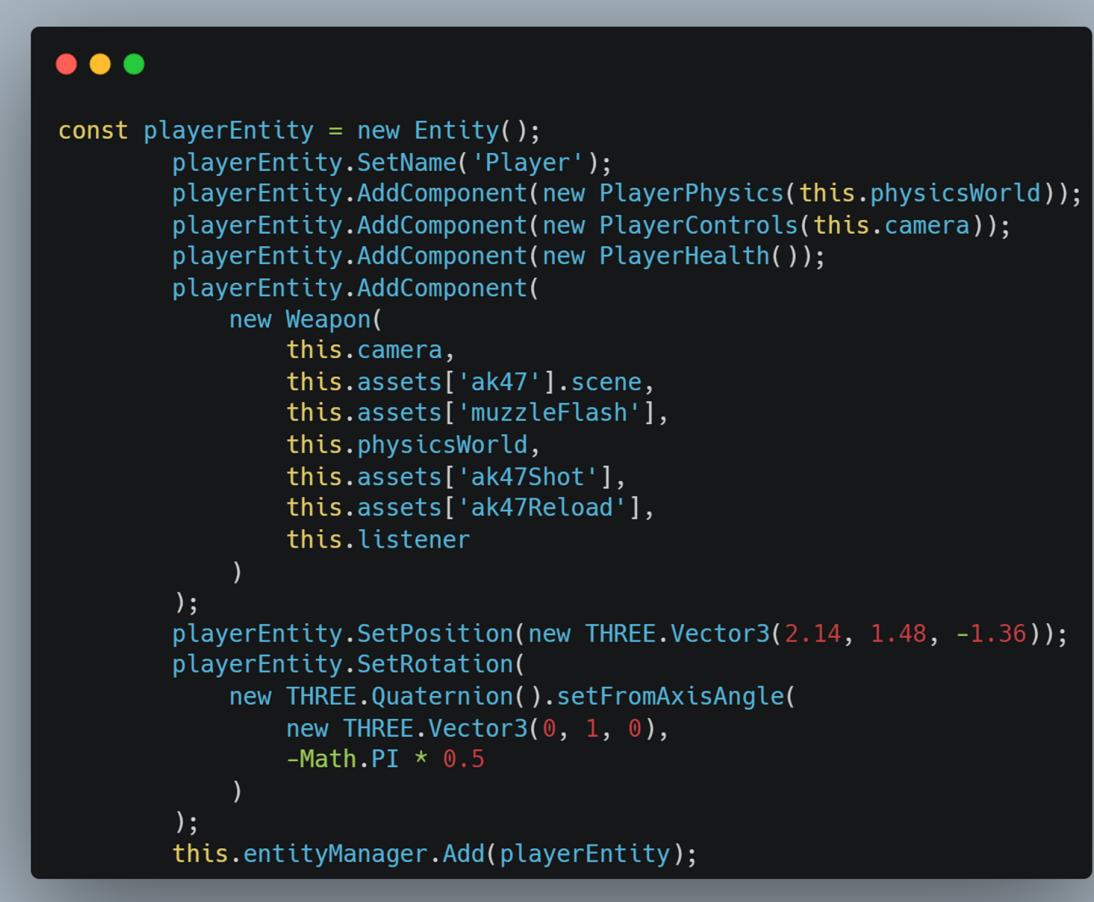




### b) Entity

- PlayerPhysics: là thành phần để xử lý vật lý của người chơi trong trò chơi, bao gồm việc sử dụng Ammo.js để tạo các hình hộp và đặt các thông số vật lý cho đối tượng người chơi.
- PlayerControls: là thành phần để xử lý điều khiển người chơi bằng bàn phím hoặc chuột trong trò chơi. Nó sử dụng các phương thức của Three.js để quay và di chuyển đối tượng người chơi.
- PlayerHealth: là thành phần để quản lý mức độ máu của người chơi trong trò chơi. Nó có thể được cập nhật bằng các sự kiện hit trong trò chơi.

- Weapon: là thành phần để quản lý vũ khí của người chơi trong trò chơi. Nó sử dụng các phương thức của Three.js để tạo hiệu ứng sáng và âm thanh khi người chơi bắn, và sử dụng Ammo.js để tính toán đường đạn.



```

const playerEntity = new Entity();
playerEntitySetName('Player');
playerEntity.AddComponent(new PlayerPhysics(this.physicsWorld));
playerEntity.AddComponent(new PlayerControls(this.camera));
playerEntity.AddComponent(new PlayerHealth());
playerEntity.AddComponent(
    new Weapon(
        this.camera,
        this.assets['ak47'].scene,
        this.assets['muzzleFlash'],
        this.physicsWorld,
        this.assets['ak47Shot'],
        this.assets['ak47Reload'],
        this.listener
    )
);
playerEntity.SetPosition(new THREE.Vector3(2.14, 1.48, -1.36));
playerEntity.SetRotation(
    new THREE.Quaternion().setFromAxisAngle(
        new THREE.Vector3(0, 1, 0),
        -Math.PI * 0.5
    )
);
this.entityManager.Add(playerEntity);

```

### c) Tăng - giảm tốc độ

Accelerate() được sử dụng để tăng tốc độ của đối tượng theo một hướng nhất định trong một khoảng thời gian t (t là thời gian giữa các khung hình), trong khi Decelerate() được sử dụng để giảm tốc độ của đối tượng theo một hệ số giảm tốc độ nhất định.

```
Accelerate = (direction, t) => {
    const accel = this.tempVec
        .copy(direction)
        .multiplyScalar(this.acceleration * t);
    this.speed.add(accel);
    this.speed.clampLength(0.0, this.maxSpeed);
};

Decelerate = (t) => {
    const frameDecelerate = this.tempVec
        .copy(this.speed)
        .multiplyScalar(this.deceleration * t);
    this.speed.add(frameDecelerate);
};
```

#### d) Di chuyển

- Xác định hướng di chuyển của bộ điều khiển nhân vật của người chơi. Hướng di chuyển là một vector 3D được chuẩn hóa phụ thuộc vào các phím mà người chơi đã nhấn: 'W' để đi về phía trước, 'S' để đi về phía sau, 'A' để di chuyển sang trái và 'D' để di chuyển sang phải.
- Kiểm tra xem người chơi đã nhấn phím cách và người chơi có thể nhảy hay không. Nếu có, nó đặt thành phần chiều dọc của vận tốc của người chơi thành một vận tốc nhảy được xác định trước và đặt một cờ chỉ ra rằng người chơi không thể nhảy lần nữa cho đến khi tiếp đất.
- Giảm tốc độ của người chơi theo hệ số giảm tốc, một giá trị được xác định trước để xác định tốc độ giảm khi người chơi ngừng di chuyển.
- Tăng tốc độ của người chơi theo hướng của vector di chuyển theo hệ số tăng tốc, cũng là một giá trị được xác định trước để xác định tốc độ tăng khi người chơi bắt đầu di chuyển.

```
Update(t) {
    const forwardFactor =
        Input.GetKeyDown('KeyS') - Input.GetKeyDown('KeyW');
    const rightFactor = Input.GetKeyDown('KeyD') - Input.GetKeyDown('KeyA');
    const direction = this.moveDir
        .set(rightFactor, 0.0, forwardFactor)
        .normalize();

    const velocity = this.physicsBody.getLinearVelocity();

    if (Input.GetKeyDown('Space') && this.physicsComponent.canJump) {
        velocity.setY(this.jumpVelocity);
        this.physicsComponent.canJump = false;
    }

    this.Decelerate(t);
    this.Accelerate(direction, t);

    const moveVector = this.tempVec.copy(this.speed);
    moveVector.applyQuaternion(this.yaw);

    velocity.setX(moveVector.x);
    velocity.setZ(moveVector.z);

    this.physicsBody.setLinearVelocity(velocity);
    this.physicsBody.setAngularVelocity(this.zeroVec);

    const ms = this.physicsBody.getMotionState();
    if (ms) {
        ms.getWorldTransform(this.transform);
        const p = this.transform.getOrigin();
        this.camera.position.set(p.x(), p.y() + this.yOffset, p.z());
        this.parent.SetPosition(this.camera.position);
    }
}
```

```
QueryJump() {
    const dispatcher = this.world.getDispatcher();
    const numManifolds = dispatcher.getNumManifolds();

    for (let i = 0; i < numManifolds; i++) {
        const contactManifold = dispatcher.getManifoldByIndexInternal(i);
        const rb0 = Ammo.castObject(
            contactManifold.getBody0(),
            Ammo.btRigidBody
        );
        const rb1 = Ammo.castObject(
            contactManifold.getBody1(),
            Ammo.btRigidBody
        );

        if (rb0 != this.body && rb1 != this.body) {
            continue;
        }

        const numContacts = contactManifold.getNumContacts();

        for (let j = 0; j < numContacts; j++) {
            const contactPoint = contactManifold.getContactPoint(j);

            const normal = contactPoint.get_m_normalWorldOnB();
            this.tempVec.setValue(normal.x(), normal.y(), normal.z());

            if (rb1 == this.body) {
                this.tempVec.setValue(
                    -this.tempVec.x(),
                    -this.tempVec.y(),
                    -this.tempVec.z()
                );
            }

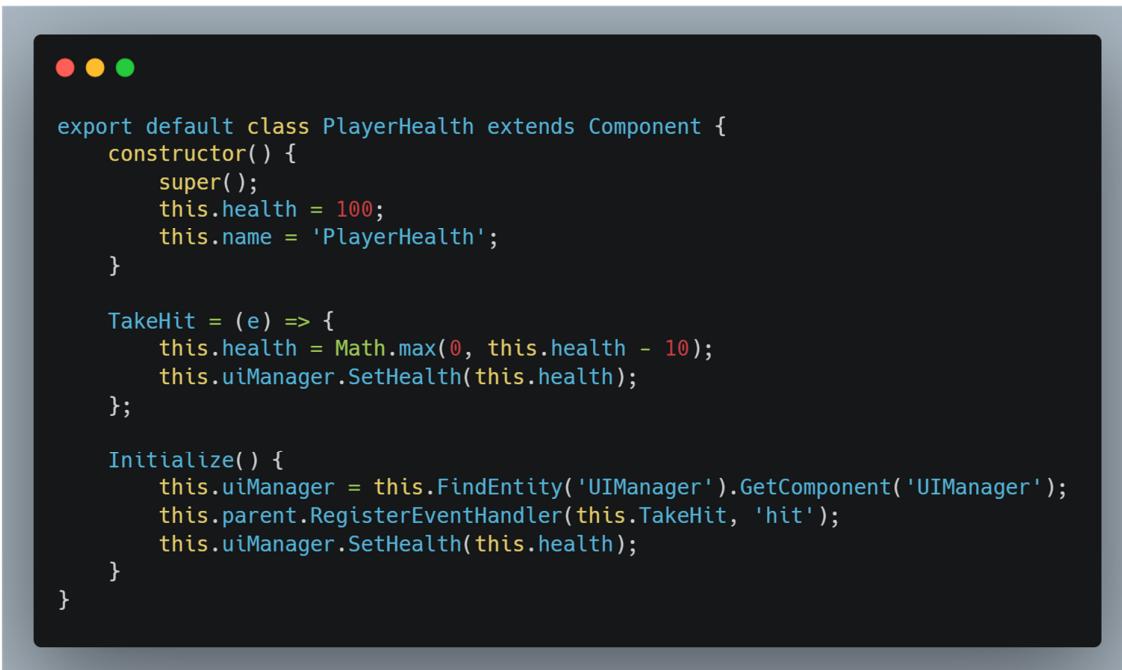
            const angle = this.tempVec.dot(this.up);
            this.canJump = angle > 0.5;

            if (this.canJump) {
                return;
            }
        }
    }
}
```

### e) Năng lượng

- Phương thức TakeHit() được sử dụng để xử lý khi nhân vật chịu sát thương. Khi gọi phương thức này, nó giảm giá trị của thuộc tính health đi 10 đơn vị và gọi phương thức SetHealth() của đối tượng UIManager để cập nhật thanh máu trên giao diện người dùng.

- Phương thức Initialize() được sử dụng để khởi tạo đối tượng PlayerHealth và liên kết với đối tượng UIManager. Phương thức này gọi phương thức FindEntity() để tìm kiếm đối tượng UIManager, sau đó gán đối tượng tìm thấy vào thuộc tính uiManager của PlayerHealth. Sau đó, phương thức này đăng ký sự kiện TakeHit() với đối tượng cha của PlayerHealth bằng cách gọi phương thức RegisterEventHandler(). Cuối cùng, phương thức SetHealth() được gọi để cập nhật thanh máu ban đầu trên giao diện người dùng.



```
● ● ●

export default class PlayerHealth extends Component {
    constructor() {
        super();
        this.health = 100;
        this.name = 'PlayerHealth';
    }

    TakeHit = (e) => {
        this.health = Math.max(0, this.health - 10);
        this.uiManager.SetHealth(this.health);
    };

    Initialize() {
        this.uiManager = this.FindEntity('UIManager').GetComponent('UIManager');
        this.parent.RegisterEventHandler(this.TakeHit, 'hit');
        this.uiManager.SetHealth(this.health);
    }
}
```

#### f) Các lớp vật lý

- Hàm Initialize() được sử dụng để khởi tạo các thông số cho rigidbody của player trong game.

```
Initialize() {
    const height = 1.3;
    const radius = 0.3;
    const mass = 5;

    const transform = new Ammo.btTransform();
    transform.setIdentity();

    const position = this.parent.Position;
    transform.setOrigin(
        new Ammo.btVector3(position.x, position.y, position.z)
    );

    const motionState = new Ammo.btDefaultMotionState(transform);

    const shape = new Ammo.btCapsuleShape(radius, height);
    const localInertia = new Ammo.btVector3(0, 0, 0);
    const bodyInfo = new Ammo.btRigidBodyConstructionInfo(
        mass,
        motionState,
        shape,
        localInertia
    );
    this.body = new Ammo.btRigidBody(bodyInfo);
    this.body.setFriction(0);

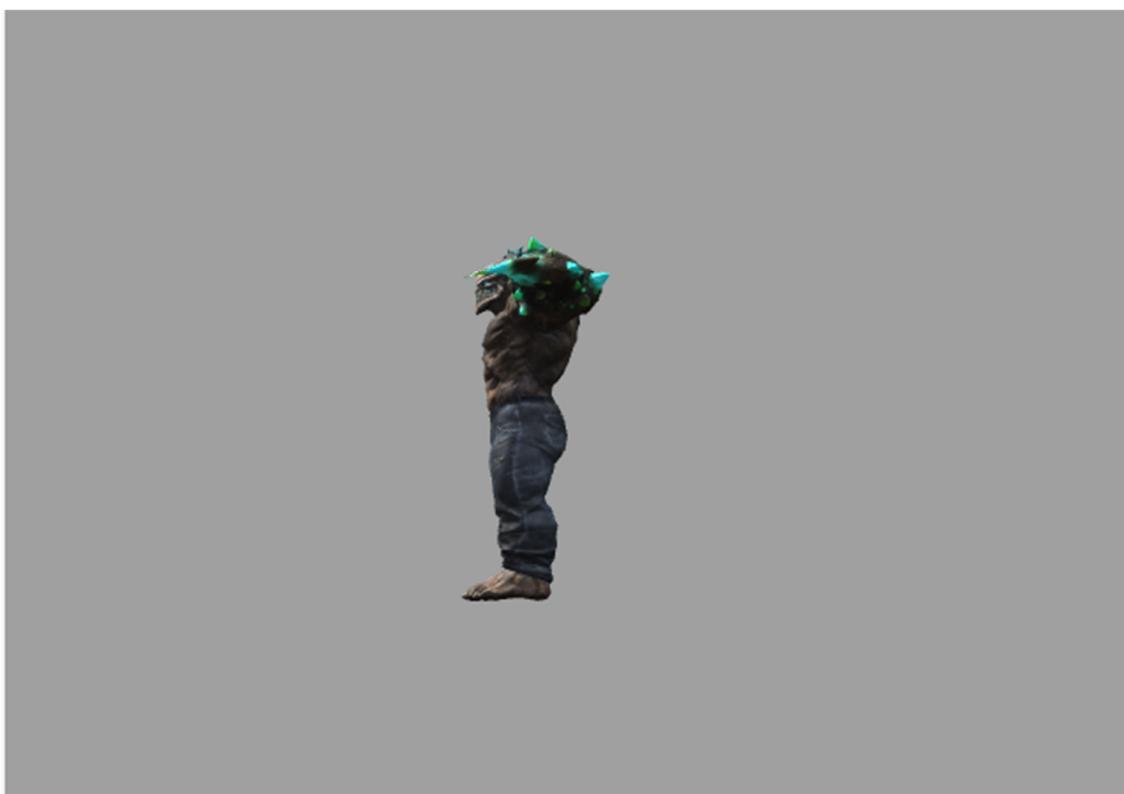
    this.body.setActivationState(DISABLE_DEACTIVATION);

    this.world.addRigidBody(this.body);
}
```

### **3.3.2. Mutant**

#### **a) Model**





### b) Load scene

- Thêm các tài nguyên vào game thông qua việc load chúng sử dụng các loại loader khác nhau, và đưa chúng vào một mảng promise để đợi cho đến khi tất cả các tài nguyên được load xong trước khi bắt đầu chạy game.

```
promises.push(this.AddAsset(mutant, fbxLoader, 'mutant'));
promises.push(this.AddAsset(idleAnim, fbxLoader, 'idleAnim'));
promises.push(this.AddAsset(walkAnim, fbxLoader, 'walkAnim'));
promises.push(this.AddAsset(runAnim, fbxLoader, 'runAnim'));
promises.push(this.AddAsset(attackAnim, fbxLoader, 'attackAnim'));
promises.push(this.AddAsset	dieAnim, fbxLoader, 'dieAnim'));

await this.PromiseProgress(promises, this.OnProgress);

this.assets['level'] = this.assets['level'].scene;
this.assets['ak47'].scene.animations = this.assets['ak47'].animations;
this.assets['muzzleFlash'] = this.assets['muzzleFlash'].scene;
```

### c) Entity

- Thành phần điều khiển nhân vật của NPC bao gồm một bản sao của model mutant được tải từ tài nguyên, các animation mutant, và đối tượng thế giới vật lý để điều khiển NPC trong không gian 3D.
- Thành phần AttackTrigger để xác định vùng kích hoạt tấn công của NPC.
- Thành phần CharacterCollision để xác định va chạm giữa NPC và các đối tượng khác trong không gian 3D.

```
const npcEntity = new Entity();
npcEntity.SetPosition(new THREE.Vector3(10, 0, 20));
npcEntitySetName(`Mutant`);
npcEntity.AddComponent(
    new NpcCharacterController(
        SkeletonUtils.clone(this.assets['mutant']),
        this.mutantAnimations,
        this.scene,
        this.physicsWorld
    )
);
npcEntity.AddComponent(new AttackTrigger(this.physicsWorld));
npcEntity.AddComponent(new CharacterCollision(this.physicsWorld));
this.entityManager.Add(npcEntity);
```

### d) Thêm animation

- Khởi tạo các animation của đối tượng mutant. Các animation được lưu trữ trong đối tượng this.mutantAnimations. Các animation này được thiết lập thông qua phương thức SetAnimation, trong đó tên animation và nội dung animation được truyền vào.

```
this.mutantAnimations = {};
this.SetAnimation('idle', this.assets['idleAnim']);
this.SetAnimation('walk', this.assets['walkAnim']);
this.SetAnimation('run', this.assets['runAnim']);
this.SetAnimation('attack', this.assets['attackAnim']);
this.SetAnimation('die', this.assets['dieAnim']);
```

- Sử dụng FSM ( Finite State Machine)

```
Init() {
    this.AddState('idle', new IdleState(this));
    this.AddState('patrol', new PatrolState(this));
    this.AddState('chase', new ChaseState(this));
    this.AddState('attack', new AttackState(this));
    this.AddState('dead', new DeadState(this));
}
```

- + Idle State: Đây là một lớp (class) định nghĩa trạng thái "Idle" (đứng im) của một đối tượng trong game. Lớp này kế thừa từ lớp State và định nghĩa các thuộc tính và phương thức của trạng thái này. Các phương thức của lớp bao gồm:
  - Name: trả về tên của trạng thái là "idle"
  - Animation: trả về đối tượng Animation của trạng thái đứng im
  - Enter: phương thức được gọi khi vào trạng thái đứng im. Phương thức này thiết lập các thuộc tính và chơi Animation.
  - Update: phương thức được gọi liên tục trong quá trình đứng im. Nếu thời gian chờ của trạng thái kết thúc, trạng thái chuyển sang trạng thái "patrol". Nếu đối tượng có thể nhìn thấy người chơi, trạng thái chuyển sang trạng thái "chase".

```
class IdleState extends State {
    constructor(parent) {
        super(parent);
        this.maxWaitTime = 5.0;
        this.minWaitTime = 1.0;
        this.waitTime = 0.0;
    }

    get Name() {
        return 'idle';
    }
    get Animation() {
        return this.parent.proxy.animations['idle'];
    }

    Enter(prevState) {
        this.parent.proxy.canMove = false;
        const action = this.Animation.action;

        if (prevState) {
            action.time = 0.0;
            action.enabled = true;
            action.crossFadeFrom(prevState.Animation.action, 0.5, true);
        }

        action.play();

        this.waitTime =
            Math.random() * (this.maxWaitTime - this.minWaitTime) +
            this.minWaitTime;
    }

    Update(t) {
        if (this.waitTime <= 0.0) {
            this.parent.SetState('patrol');
            return;
        }

        this.waitTime -= t;

        if (this.parent.proxy.CanSeeThePlayer()) {
            this.parent.SetState('chase');
        }
    }
}
```

- + PatrolState: Khi ở trạng thái này, nhân vật sẽ di chuyển đến các điểm tuần tra ngẫu nhiên trên bản đồ. Nếu nhìn thấy người chơi thì nhân vật sẽ chuyển sang trạng thái "chase" (tạm dịch là truy đuổi) để tấn công người chơi. Nếu đã hoàn thành tuần tra thì sẽ chuyển sang trạng thái "idle" (đứng yên). Trong class này, các hành động cụ thể như di chuyển, xác định điểm tuần tra, kiểm tra trạng

thái của nhân vật được xử lý bên trong đối tượng parent (tức là đối tượng chứa class PatrolState).



```
class PatrolState extends State {
    constructor(parent) {
        super(parent);
    }

    get Name() {
        return 'patrol';
    }
    get Animation() {
        return this.parent.proxy.animations['walk'];
    }

    PatrolEnd = () => {
        this.parent.SetState('idle');
    };

    Enter(prevState) {
        this.parent.proxy.canMove = true;
        const action = this.Animation.action;

        if (prevState) {
            action.time = 0.0;
            action.enabled = true;
            action.crossFadeFrom(prevState.Animation.action, 0.5, true);
        }

        action.play();

        this.parent.proxy.NavigateToRandomPoint();
    }

    Update(t) {
        if (this.parent.proxy.CanSeeThePlayer()) {
            this.parent.SetState('chase');
        } else if (
            this.parent.proxy.path &&
            this.parent.proxy.path.length == 0
        ) {
            this.parent.SetState('idle');
        }
    }
}
```

- + ChaseState: Đây là một trạng thái khi kẻ thù đang truy đuổi người chơi (player). Các thuộc tính và phương thức của lớp này bao gồm:
  - Thuộc tính updateFrequency: Khoảng thời gian giữa các lần cập nhật hướng di chuyển tới người chơi.

- Thuộc tính updateTimer: Thời gian còn lại cho đến khi tiếp tục cập nhật hướng di chuyển.
- Thuộc tính attackDistance: Khoảng cách để kẻ thù tấn công người chơi.
- Thuộc tính shouldRotate: Biến đánh dấu liệu kẻ thù nên xoay hướng để đổi diện với người chơi không.
- Thuộc tính switchDelay: Thời gian giữa các lần chuyển đổi trạng thái.
- Phương thức RunToPlayer(): Đặt trạng thái cho kẻ thù là đang chạy tới người chơi.
- Phương thức Enter(): Thực hiện các hành động khi vào trạng thái này, bao gồm cập nhật trạng thái chạy tới người chơi nếu có trạng thái trước đó.
- Phương thức Update(): Thực hiện các hành động cập nhật khi đang trong trạng thái này, bao gồm cập nhật hướng di chuyển, kiểm tra xem kẻ thù đã gần đủ để tấn công hay chưa, và quyết định khi nào chuyển sang trạng thái tấn công.

```
● ○ ● ●

class ChaseState extends State {
    constructor(parent) {
        super(parent);
        this.updateFrequency = 0.5;
        this.updateTimer = 0.0;
        this.attackDistance = 2.0;
        this.shouldRotate = false;
        this.switchDelay = 0.2;
    }

    get Name() {
        return 'chase';
    }
    get Animation() {
        return this.parent.proxy.animations['run'];
    }

    RunToPlayer(prevState) {
        this.parent.proxy.canMove = true;
        const action = this.Animation.action;
        this.updateTimer = 0.0;

        if (prevState) {
            action.time = 0.0;
            action.enabled = true;
            action.setEffectiveTimeScale(1.0);
            action.setEffectiveWeight(1.0);
            action.crossFadeFrom(prevState.Animation.action, 0.2, true);
        }

        action.timeScale = 1.5;
        action.play();
    }

    Enter(prevState) {
        this.RunToPlayer(prevState);
    }

    Update(t) {
        if (this.updateTimer <= 0.0) {
            this.parent.proxy.NavigateToPlayer();
            this.updateTimer = this.updateFrequency;
        }

        if (this.parent.proxy.IsCloseToPlayer) {
            if (this.switchDelay <= 0.0) {
                this.parent.SetState('attack');
            }

            this.parent.proxy.ClearPath();
            this.switchDelay -= t;
        } else {
            this.switchDelay = 0.1;
        }

        this.updateTimer -= t;
    }
}
```

- + AttackState: Lớp AttackState được sử dụng để xử lý trạng thái tấn công của kẻ thù. Các thuộc tính và phương thức chính của lớp này bao gồm:
  - Thuộc tính attackTime: đại diện cho thời gian còn lại để kẻ thù tấn công.
  - Thuộc tính canHit: đại diện cho khả năng tấn công của kẻ thù.
  - Phương thức Enter(prevState): được gọi khi kẻ thù chuyển sang trạng thái tấn công. Nó đặt cờ "canMove" của kẻ thù thành false, thiết lập thời gian tấn công và sự kiện tấn công của hành động và chơi hành động tấn công.
  - Phương thức Update(t): được gọi mỗi khung hình để cập nhật trạng thái của kẻ thù. Nó đặt hướng nhìn của kẻ thù đến người chơi, kiểm tra xem người chơi có ở trong phạm vi tấn công của kẻ thù và nếu có, thực hiện tấn công. Nếu thời gian tấn công đã hết và người chơi không ở gần kẻ thù, thì kẻ thù sẽ chuyển sang trạng thái đuối theo người chơi.

```

class AttackState extends State {
    constructor(parent) {
        super(parent);
        this.attackTime = 0.0;
        this.canHit = true;
    }

    get Name() {
        return 'attack';
    }
    get Animation() {
        return this.parent.proxy.animations['attack'];
    }

    Enter(prevState) {
        this.parent.proxy.canMove = false;
        const action = this.Animation.action;
        this.attackTime = this.Animation.clip.duration;
        this.attackEvent = this.attackTime * 0.85;

        if (prevState) {
            action.time = 0.0;
            action.enabled = true;
            action.crossFadeFrom(prevState.Animation.action, 0.1, true);
        }

        action.play();
    }

    Update(t) {
        this.parent.proxy.FacePlayer(t);

        if (!this.parent.proxy.IsCloseToPlayer && this.attackTime <= 0.0) {
            this.parent.SetState('chase');
            return;
        }

        if (
            this.canHit &&
            this.attackTime <= this.attackEvent &&
            this.parent.proxy.IsPlayerInHitbox
        ) {
            this.parent.proxy.HitPlayer();
            this.canHit = false;
        }

        if (this.attackTime <= 0.0) {
            this.attackTime = this.Animation.clip.duration;
            this.canHit = true;
        }

        this.attackTime -= t;
    }
}

```

- + DeadState: trạng thái của đối tượng khi nó đã chết.



```
class DeadState extends State {
    constructor(parent) {
        super(parent);
    }

    get Name() {
        return 'dead';
    }
    get Animation() {
        return this.parent.proxy.animations['die'];
    }

    Enter(prevState) {
        const action = this.Animation.action;
        action.setLoop(THREE.LoopOnce, 1);
        action.clampWhenFinished = true;

        if (prevState) {
            action.time = 0.0;
            action.enabled = true;
            action.crossFadeFrom(prevState.Animation.action, 0.1, true);
        }

        action.play();
    }

    Update(t) {}
}
```

#### e) Trạng thái phát hiện người chơi

- Phương thức này sẽ kiểm tra nếu nhân vật chính (player) có nằm trong góc nhìn của quái, và sau đó sử dụng Raycast để kiểm tra xem có bất kỳ đối tượng nào nằm trong đường đi từ quái đến người chơi hay không. Nếu như nhân vật chính là đối tượng đầu tiên mà Raycast gặp phải, phương thức sẽ trả về giá trị true, nghĩa là nhân vật chính đang được nhìn thấy. Ngược lại, phương thức sẽ trả về giá trị false.

```
CanSeeThePlayer() {
    const playerPos = this.player.Position.clone();
    const modelPos = this.model.position.clone();
    modelPos.y += 1.35;
    const charToPlayer = playerPos.sub(modelPos);

    if (playerPos.lengthSq() > this.maxViewDistance) {
        return;
    }

    charToPlayer.normalize();
    const angle = charToPlayer.dot(this.dir);

    if (angle < this.viewAngle) {
        return false;
    }

    const rayInfo = {};
    const collisionMask =
        CollisionFilterGroups.AllFilter &
        ~CollisionFilterGroups.SensorTrigger;

    if (
        AmmoHelper.CastRay(
            this.physicsWorld,
            modelPos,
            this.player.Position,
            rayInfo,
            collisionMask
        )
    ) {
        const body = Ammo.castObject(
            rayInfo.collisionObject,
            Ammo.btRigidBody
        );

        if (body == this.player.GetComponent('PlayerPhysics').body) {
            return true;
        }
    }
}

return false;
}
```

- Sau khi phát hiện người chơi, quái cắn xoay hướng về phía người chơi

```
FacePlayer(t, rate = 3.0) {  
    this.tempVec.copy(this.player.Position).sub(this.model.position);  
    this.tempVec.y = 0.0;  
    this.tempVec.normalize();  
  
    this.tempRot.setFromUnitVectors(this.forwardVec, this.tempVec);  
    this.model.quaternion.rotateTowards(this.tempRot, rate * t);  
}
```

- Sau đó, Phương thức IsCloseToPlayer có tác dụng kiểm tra xem nhân vật đang điều khiển có gần với nhân vật người chơi không. Đầu tiên, nó tính toán vector giữa vị trí của nhân vật đang điều khiển và vị trí của nhân vật người chơi. Sau đó, nó so sánh độ dài của vector này với bình phương khoảng cách tấn công. Nếu độ dài vector này nhỏ hơn hoặc bằng bình phương khoảng cách tấn công thì phương thức trả về giá trị true để biểu thị nhân vật đang gần với nhân vật người chơi, ngược lại thì trả về giá trị false.

```
get IsCloseToPlayer() {  
    this.tempVec.copy(this.player.Position).sub(this.model.position);  
  
    if (  
        this.tempVec.lengthSq() <=  
        this.attackDistance * this.attackDistance  
    ) {  
        return true;  
    }  
  
    return false;  
}
```

## f) Chuyển hướng

- Hai phương thức NavigateToRandomPoint và NavigateToPlayer được sử dụng để tính toán đường đi cho quái đến một điểm ngẫu nhiên hoặc đến vị trí của người chơi. NavigateToRandomPoint sẽ chọn một điểm ngẫu nhiên trên bề mặt Navmesh gần vị trí hiện tại của model và tìm đường đi từ vị trí hiện tại của quái đến điểm đó. NavigateToPlayer sẽ tính toán đường đi từ vị trí hiện tại của quái đến vị trí của người chơi.

```
● ○ ●  
NavigateToRandomPoint() {  
    const node = this.navmesh.GetRandomNode(this.model.position, 50);  
    this.path = this.navmesh.FindPath(this.model.position, node);  
}  
  
NavigateToPlayer() {  
    this.tempVec.copy(this.player.Position);  
    this.tempVec.y = 0.5;  
    this.path = this.navmesh.FindPath(this.model.position, this.tempVec);  
}
```

### g) Tấn công người chơi

```
● ○ ●  
HitPlayer() {  
    this.player.Broadcast({ topic: 'hit' });  
}
```

- Khi bị tấn công, Phương thức TakeHit được sử dụng để xử lý khi đối tượng nhận phải lực tấn công. Nó nhận đầu vào là một đối tượng msg, chứa thông tin về lực tấn công nhận được, bao gồm số lượng lực tấn công amount. Phương thức này trừ đi lượng lực tấn công từ máu của đối tượng, và nếu máu xuống còn 0, đối tượng chuyển sang trạng thái 'dead'. Nếu máu vẫn còn, đối tượng sẽ chuyển sang trạng thái 'chase' nếu đang ở trạng thái 'idle' hoặc 'patrol'.

```
TakeHit = (msg) => {
    this.health = Math.max(0, this.health - msg.amount);

    if (this.health == 0) {
        this.stateMachine.SetState('dead');
    } else {
        const stateName = this.stateMachine.currentState.Name;
        if (stateName == 'idle' || stateName == 'patrol') {
            this.stateMachine.SetState('chase');
        }
    }
};
```

### 3.3.3. AK47

#### a) Load sense

- Thêm các tài nguyên vào game thông qua việc load chúng sử dụng các loại loader khác nhau, và đưa chúng vào một mảng promise để đợi cho đến khi tất cả các tài nguyên được load xong trước khi bắt đầu chạy game.

```
promises.push(this.AddAsset(ak47, gltfLoader, 'ak47'));
promises.push(this.AddAsset(muzzleFlash, gltfLoader, 'muzzleFlash'));
promises.push(this.AddAsset(ak47ShotAudio, audioLoader, 'ak47Shot'));
promises.push(
    this.AddAsset(ak47ReloadAudio, audioLoader, 'ak47Reload')
);
```

#### b) Animation

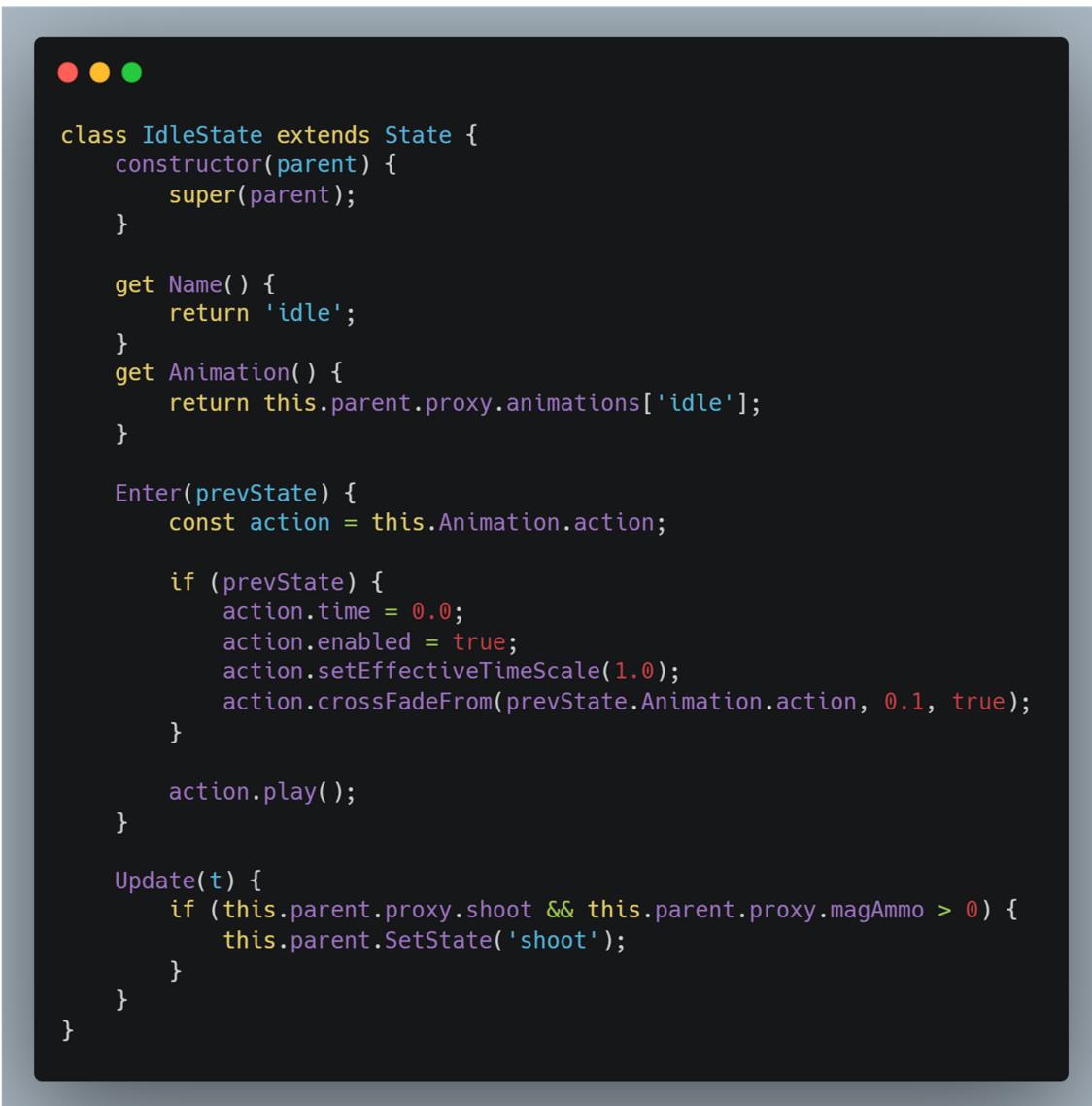
- Tạo các chuyển động cho súng: các trạng thái: idle, shoot, reload

```
Init() {
    this.AddState('idle', new IdleState(this));
    this.AddState('shoot', new ShootState(this));
    this.AddState('reload', new ReloadState(this));
}
```

- + Idle: Lớp IdleState trong đoạn mã này là một lớp con của lớp State và định nghĩa trạng thái "đứng yên" của nhân vật. Nó bao gồm phương thức Enter, Update và các thuộc tính Name và Animation.

Phương thức Enter được gọi khi trạng thái mới được đặt làm trạng thái hiện tại của nhân vật. Nó thiết lập một hành động animation (ví dụ: chuyển động đứng yên) và bắt đầu phát nó. Nếu trạng thái trước đó được truyền vào (nghĩa là trạng thái đã chuyển từ trạng thái trước đó sang trạng thái hiện tại), thì phương thức crossFadeFrom được gọi để chuyển đổi mượt mà giữa các hành động animation.

Phương thức Update được gọi mỗi khung hình và kiểm tra xem nhân vật có bắn và còn đạn hay không. Nếu đúng, nó sẽ chuyển sang trạng thái bắn (ShootState).



```

class IdleState extends State {
    constructor(parent) {
        super(parent);
    }

    get Name() {
        return 'idle';
    }
    get Animation() {
        return this.parent.proxy.animations['idle'];
    }

    Enter(prevState) {
        const action = this.Animation.action;

        if (prevState) {
            action.time = 0.0;
            action.enabled = true;
            action.setEffectiveTimeScale(1.0);
            action.crossFadeFrom(prevState.Animation.action, 0.1, true);
        }

        action.play();
    }

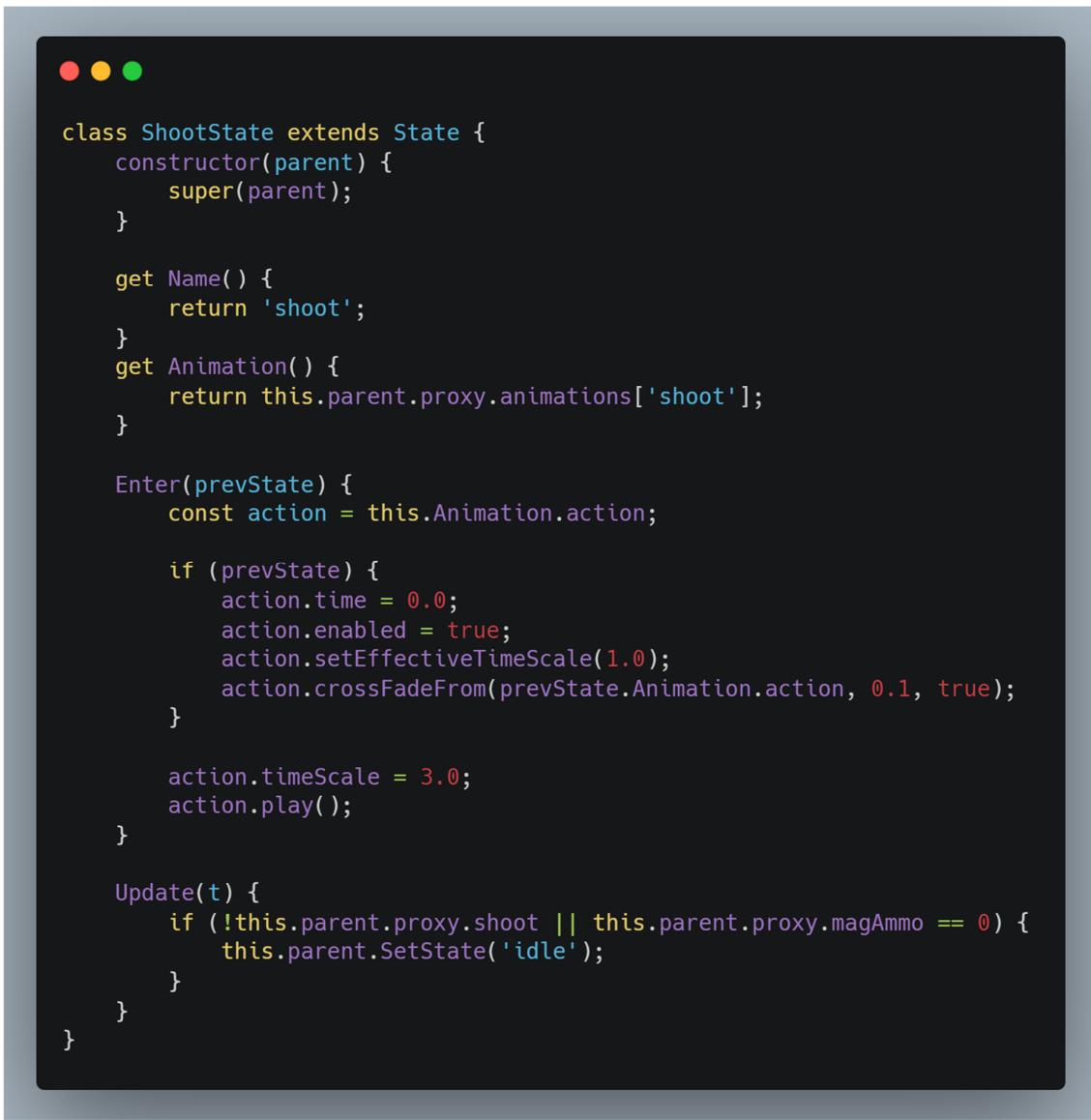
    Update(t) {
        if (this.parent.proxy.shoot && this.parent.proxy.magAmmo > 0) {
            this.parent.SetState('shoot');
        }
    }
}

```

- + Shoot: Lớp ShootState thực hiện các hành động trong trạng thái bắn súng của nhân vật. Nó kế thừa từ lớp State và triển khai các phương thức Enter() và Update(). Enter() được gọi khi trạng thái mới được thiết lập, và Update() được gọi mỗi khung hình.

Phương thức Enter() bắt đầu chơi animation bắn súng, đặt thời gian, tốc độ và lặp lại của animation. Nếu trạng thái trước đó tồn tại, nó sẽ chuyển từ trạng thái đó sang trạng thái bắn súng bằng cách sử dụng phương thức crossFadeFrom().

Phương thức Update() sẽ kiểm tra nếu nhân vật không bắn hoặc hết đạn, nó sẽ chuyển sang trạng thái rảnh rỗi (idle) bằng cách sử dụng phương thức SetState().



```

class ShootState extends State {
    constructor(parent) {
        super(parent);
    }

    get Name() {
        return 'shoot';
    }
    get Animation() {
        return this.parent.proxy.animations['shoot'];
    }

    Enter(prevState) {
        const action = this.Animation.action;

        if (prevState) {
            action.time = 0.0;
            action.enabled = true;
            action.setEffectiveTimeScale(1.0);
            action.crossFadeFrom(prevState.Animation.action, 0.1, true);
        }

        action.timeScale = 3.0;
        action.play();
    }

    Update(t) {
        if (!this.parent.proxy.shoot || this.parent.proxy.magAmmo == 0) {
            this.parent.SetState('idle');
        }
    }
}

```

- + Reload: Lớp ReloadState được kế thừa từ lớp State và chứa các phương thức và thuộc tính cần thiết cho trạng thái tái nạp (ReloadState) của nhân vật. Trong

constructor, nó gọi hàm khởi tạo của lớp State và thêm một sự kiện "finished" cho bộ trộn (mixer) của đối tượng nhân vật. Lớp cũng định nghĩa các thuộc tính "Name" và "Animation", lần lượt trả về tên của trạng thái và hoạt cảnh (animation) cho trạng thái tái nạp.

Phương thức "AnimationFinished" được gọi khi hoạt cảnh tái nạp kết thúc. Nó kiểm tra xem hoạt cảnh được gọi đã hoàn thành chưa, nếu chưa thì nó sẽ không làm gì cả. Nếu hoạt cảnh đã hoàn thành, nó sẽ gọi phương thức "ReloadDone" trên đối tượng nhân vật và chuyển sang trạng thái "idle".

Phương thức "Enter" được gọi khi trạng thái được thiết lập cho đối tượng nhân vật. Nó lấy hoạt cảnh tái nạp và thiết lập chế độ lặp lại (loop) một lần. Nếu trạng thái trước đó được cung cấp, nó sẽ chuyển sang trạng thái mới dần dần từ trạng thái cũ. Sau đó, nó bắt đầu chạy hoạt cảnh tái nạp.

```
class ReloadState extends State {
    constructor(parent) {
        super(parent);

        this.parent.proxy.mixer.addEventListener(
            'finished',
            this.AnimationFinished
        );
    }

    get Name() {
        return 'reload';
    }
    get Animation() {
        return this.parent.proxy.animations['reload'];
    }

    AnimationFinished = (e) => {
        if (e.action != this.Animation.action) {
            return;
        }

        this.parent.proxy.ReloadDone();
        this.parent.SetState('idle');
    };
}

Enter(prevState) {
    const action = this.Animation.action;
    action.loop = THREE.LoopOnce;

    if (prevState) {
        action.time = 0.0;
        action.enabled = true;
        action.setEffectiveTimeScale(1.0);
        action.crossFadeFrom(prevState.Animation.action, 0.1, true);
    }

    action.play();
}
}
```

### c) Hiệu ứng khi bắn súng

Hàm "SetMuzzleFlash" được sử dụng để tạo hiệu ứng sáng chớp trên nòng súng khi bắn. Các bước thực hiện của hàm này gồm:

- Đặt vị trí ban đầu cho hiệu ứng sáng chớp trên nòng súng.
- Xoay hiệu ứng sáng chớp để đúng hướng bắn.
- Thêm hiệu ứng sáng chớp vào mô hình vũ khí.
- Thiết lập thời gian của hiệu ứng sáng chớp ban đầu là 0.

- Thiết lập chế độ blending của vật liệu của hiệu ứng sáng chớp là THREE.AdditiveBlending để tạo hiệu ứng sáng chớp cộng hưởng.



```

SetMuzzleFlash() {
    this.flash.position.set(-0.3, -0.5, 8.3);
    this.flash.rotateY(Math.PI);
    this.model.add(this.flash);
    this.flash.life = 0.0;

    this.flash.children[0].material.blending = THREE.AdditiveBlending;
}

```

#### d) Animation cho hiệu ứng bắn súng

Phương thức AnimateMuzzle() được sử dụng để tạo hiệu ứng sáng tạo bên trong nòng súng. Trong phương thức này, chúng ta lấy tối phần tử của vật thể flash làm nòng súng và thay đổi độ trong suốt của vật liệu để tạo hiệu ứng lửa. Hiệu ứng sẽ mất dần theo thời gian và sẽ mất hoàn toàn khi life của flash bằng 0.0.



```

AnimateMuzzle(t) {
    const mat = this.flash.children[0].material;
    const ratio = this.flash.life / this.fireRate;
    mat.opacity = ratio;
    this.flash.life = Math.max(0.0, this.flash.life - t);
}

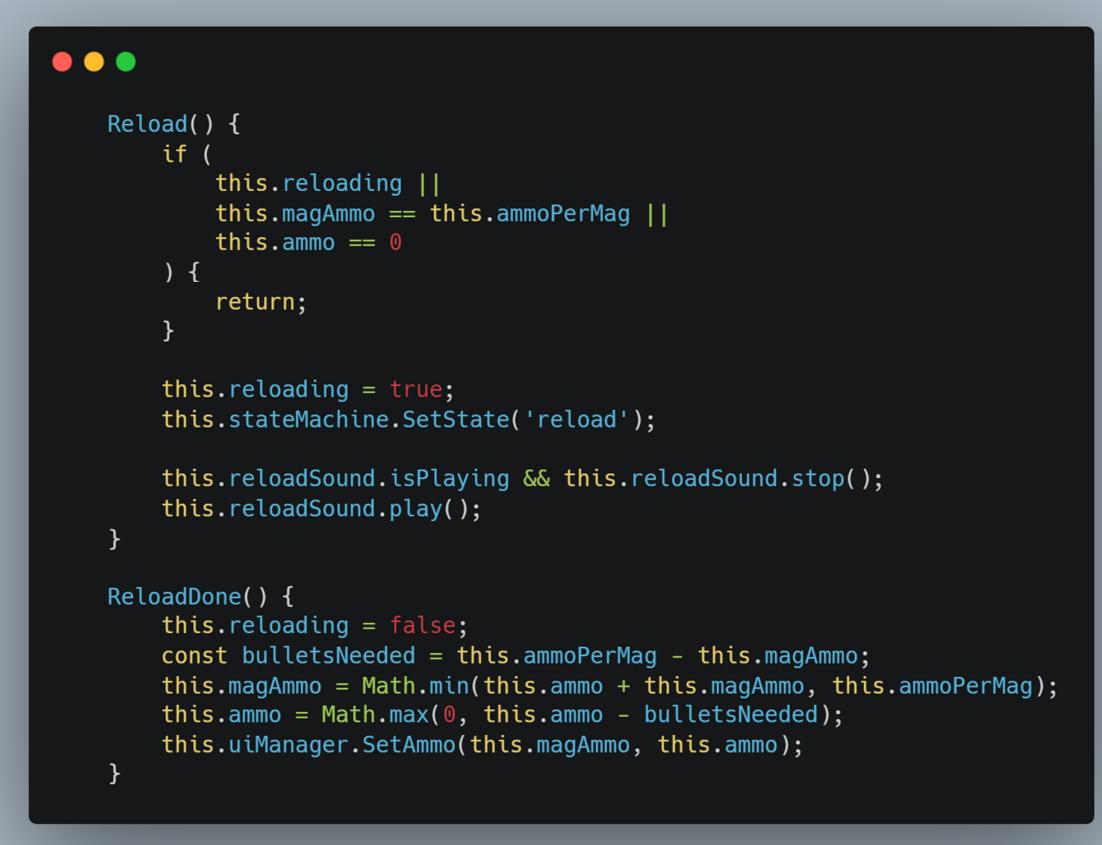
```

#### e) Nạp đạn

- Phương thức Reload() trong đoạn mã trên được sử dụng để thực hiện nạp đạn vào khẩu súng. Nếu trạng thái đang là "reloading" hoặc số viên đạn trong khẩu súng bằng số viên tối đa hoặc hết đạn, thì phương thức sẽ không được thực thi. Nếu không, trạng thái của khẩu súng sẽ được đặt thành "reload", âm thanh được phát và trạng thái của khẩu súng sẽ được chuyển sang "reloading".
- Phương thức Reload() sẽ xử lý việc nạp đạn cho súng, nhưng nó chỉ được thực thi khi súng không đang trong quá trình nạp đạn, tức là biến reloading là false và súng không đầy đạn và còn đạn trong kho. Nếu điều kiện thỏa mãn, reloading sẽ được gán là true, sau đó trạng thái của súng sẽ được chuyển sang

trạng thái reload thông qua việc gọi hàm SetState('reload'). Sau đó, âm thanh nạp đạn sẽ được phát.

- Phương thức ReloadDone() sẽ được gọi khi quá trình nạp đạn hoàn tất. reloading được gán là false, và số viên đạn cần nạp để đầy đạn trong súng được tính toán. Số viên đạn trong súng được cập nhật và số đạn trong kho còn lại sau khi bắn được cập nhật trên giao diện người dùng thông qua hàm SetAmmo() của đối tượng UIManager.



```
Reload() {
    if (
        this.reloading ||
        this.magAmmo == this.ammoPerMag ||
        this.ammo == 0
    ) {
        return;
    }

    this.reloading = true;
    this.stateMachine.SetState('reload');

    this.reloadSound.isPlaying && this.reloadSound.stop();
    this.reloadSound.play();
}

ReloadDone() {
    this.reloading = false;
    const bulletsNeeded = this.ammoPerMag - this.magAmmo;
    this.magAmmo = Math.min(this.ammo + this.magAmmo, this.ammoPerMag);
    this.ammo = Math.max(0, this.ammo - bulletsNeeded);
    this.uiManager.SetAmmo(this.magAmmo, this.ammo);
}
```

#### f) CastRayWeapon

Phương thức CastRayWeapon() dùng để bắn tia chạm đạn và xử lý sự kiện khi đạn chạm vào vật thể. Phương thức này sử dụng ba đối tượng THREE.Vector3 để tạo tia chạm đạn từ camera. Sau đó, phương thức sử dụng AmmoHelper.CastRay() để kiểm tra tia chạm đạn có chạm vào vật thể hay không. Nếu có, phương thức sẽ lấy thông tin vật thể bị chạm và gửi một sự kiện 'hit' cho entity của vật thể đó để xử lý việc giảm máu, hủy vật thể, v.v.

```
CastRayWeapon() {
    const start = new THREE.Vector3(0.0, 0.0, -1.0);
    start.unproject(this.camera);
    const end = new THREE.Vector3(0.0, 0.0, 1.0);
    end.unproject(this.camera);

    const collisionMask =
        CollisionFilterGroups.AllFilter &
        CollisionFilterGroups.SensorTrigger;

    if (
        AmmoHelper.CastRay(
            this.world,
            start,
            end,
            this.hitResult,
            collisionMask
        )
    ) {
        const ghostBody = Ammo.castObject(
            this.hitResult.collisionObject,
            Ammo.btPairCachingGhostObject
        );
        const rigidBody = Ammo.castObject(
            this.hitResult.collisionObject,
            Ammo.btRigidBody
        );
        const entity = ghostBody.parentEntity || rigidBody.parentEntity;

        entity &&
        entity.Broadcast({
            topic: 'hit',
            from: this.parent,
            amount: this.damage,
            hitResult: this.hitResult,
        });
    }
}
```

### g) Bắn súng

Hàm AnimateMuzzle được sử dụng để thực hiện animation của đầu khẩu súng sau khi bắn. Nó được gọi trong hàm Shoot của class Weapon, và nhận tham số t là thời gian giữa các khung hình.

```
Shoot(t) {
    if (!this.shoot) {
        return;
    }

    if (!this.magAmmo) {
        // Reload automatically
        this.Reload();
        return;
    }

    if (this.shootTimer <= 0.0) {
        // Shoot
        this.flash.life = this.fireRate;
        this.flash.rotateZ(Math.PI * Math.random());
        const scale = Math.random() * (1.5 - 0.8) + 0.8;
        this.flash.scale.set(scale, 1, 1);
        this.shootTimer = this.fireRate;
        this.magAmmo = Math.max(0, this.magAmmo - 1);
        this.uiManager.SetAmmo(this.magAmmo, this.ammo);

        this.CastRayWeapon();
        this.Broadcast({ topic: 'ak47_shot' });

        this.shotSound.isPlaying && this.shotSound.stop();
        this.shotSound.play();
    }

    this.shootTimer = Math.max(0.0, this.shootTimer - t);
}
```

### 3.4. Setup environment, audio effect

#### a) Setup environment

- Quá trình phát triển sản phẩm game FPS này, ta có thể sử dụng hai loại môi trường mô phỏng điển hình như mô phỏng vật lý và mô phỏng đồ họa. Mô phỏng vật lý cho phép ta hiển thị các tính tương tác, tác động dựa trên các định luật như va chạm, tốc độ, lực ma sát, cường độ tiếp xúc. Điều này dễ dàng thực hiện khi có được sự hỗ trợ của thư viện AmmoJS. Còn việc mô phỏng đồ họa bao gồm toàn bộ các chức năng hay các hình ảnh 3D trực quan thì ta có thể quản lý bằng thư viện ThreeJS.
- Cài đặt môi trường vật lý với AmmoJS

```
SetupPhysics() {
    // Physics configuration
    const collisionConfiguration =
        new Ammo.btDefaultCollisionConfiguration();
    const dispatcher = new Ammo.btCollisionDispatcher(
        collisionConfiguration
    );
    const broadPhase = new Ammo.btDbvtBroadphase();
    const solver = new Ammo.btSequentialImpulseConstraintSolver();
    this.physicsWorld = new Ammo.btDiscreteDynamicsWorld(
        dispatcher,
        broadPhase,
        solver,
        collisionConfiguration
    );
    this.physicsWorld.setGravity(new Ammo.btVector3(0.0, -9.81, 0.0));
    const fp = Ammo.addFunction(this.PhysicsUpdate);
    this.physicsWorld.setInternalTickCallback(fp);
    this.physicsWorld
        .getBroadphase()
        .getOverlappingPairCache()
        .setInternalGhostPairCallback(new Ammo.btGhostPairCallback());
}
```

- Cài đặt môi trường đồ họa 3D với Threejs



```
SetupGraphics() {
    this.scene = new THREE.Scene();
    this.renderer = new THREE.WebGLRenderer({ antialias: true });
    this.renderer.shadowMap.enabled = true;
    this.renderer.shadowMap.type = THREE.PCFSoftShadowMap;

    this.renderer.toneMapping = THREE.ReinhardToneMapping;
    this.renderer.toneMappingExposure = 1;
    this.renderer.outputEncoding = THREE.sRGBEncoding;

    this.camera = new THREE.PerspectiveCamera();
    this.camera.near = 0.01;
    // add element Three into dom
    document.body.appendChild(this.renderer.domElement);

    // Start monitor
    this.stats = new Stats();
    document.body.appendChild(this.stats.dom);
}
```

### b) Audio effect

Âm thanh được sử dụng trong sản phẩm bao gồm 2 loại là âm thanh nền và âm thanh hiệu ứng. Âm thanh nền sẽ được phát ngay khi hiển thị giao diện trò chơi và có thể điều chỉnh tắt bật tùy ý. Âm thanh hiệu ứng bao gồm âm thanh lúc bắn súng và nạp đạn.

Trong thư viện three.js, AudioListener là một lớp đại diện cho một bộ lắng nghe âm thanh. Nó được sử dụng để lắng nghe các sự kiện liên quan đến âm thanh, chẳng hạn như khi âm thanh được phát lại hoặc dừng lại.

Để sử dụng AudioListener, cần tạo một thẻ hiện của nó và gán cho camera bằng cách sử dụng phương thức setCamera(camera)

```
this.listener = new THREE.AudioListener();
this.camera.add(this.listener);
```

- Load âm thanh vào scene game

```
promises.push(
    this.AddAsset(backgroundAudio, audioLoader, 'backgroundMusic')
);
```

```
promises.push(this.AddAsset(ak47ShotAudio, audioLoader, 'ak47Shot'));
promises.push(
    this.AddAsset(ak47ReloadAudio, audioLoader, 'ak47Reload')
);
```

- Cài đặt âm thanh nền cho trò chơi với chế độ tự động phát và vòng lặp

```
SetSoundBackground() {
    this.backgroundSound = new THREE.Audio(this.listener);
    this.backgroundSound.setBuffer(this.assets['backgroundMusic']);
    this.backgroundSound.setLoop(true);
    this.backgroundSound.play();
}
```

- Tùy chỉnh bật/tắt âm thanh nền

```
SetToggleSoundBackground = () => {
    const volumeOn = document.getElementById('volume_on');
    const mute = document.getElementById('mute');

    const onPlaySound = () => {
        this.backgroundSound.play();
    };

    const onOffSound = () => {
        this.backgroundSound.stop();
    };

    console.log(this.backgroundSound?.isPlaying);
    if (this.backgroundSound.isPlaying) {
        volumeOn.style.visibility = 'hidden';
        mute.style.visibility = 'visible';
        this.backgroundSound.stop();
    } else {
        volumeOn.style.visibility = 'visible';
        mute.style.visibility = 'hidden';
        // volumeOn.setAttribute('visibility', 'visible');
        // mute.setAttribute('visibility', 'hidden');
        this.backgroundSound.play();
    }
};
```

- Cài đặt âm thanh hiệu ứng nổ súng và nạp đạn



```

SetSoundEffect() {
    this.shotSound = new THREE.Audio(this.audioListener);
    this.shotSound.setBuffer(this.shotSoundBuffer);
    this.shotSound.setLoop(false);
    // reload sound buffer
    this.reloadSound = new THREE.Audio(this.audioListener);
    this.reloadSound.setBuffer(this.reloadSoundBuffer);
    this.reloadSound.setLoop(false);
}

```

#### 4. Demo (video demo)

### 5. Đánh giá nghiệm thu

#### 5.1. Ưu & nhược điểm

- **Ưu điểm**
  - + Đồ họa đẹp mắt, chân thật
  - + Âm thanh nền, âm thanh hiệu ứng phù hợp
  - + Camera di chuyển linh hoạt theo đúng hướng mắt nhìn của người chơi
- **Nhược điểm**
  - + Thiết kế giao diện chưa rõ ràng, nhiều yếu tố về UI còn gây khó hiểu cho người dùng
  - + Nhân vật quái vật còn bị tình trạng kẹt khi chạm vào khung cảnh
  - + Chưa có chức năng tấn công quái vật, hiện tại đang giới hạn chỉ có di chuyển, bắn súng và để quái vật tấn công mình
  - + Hiện chỉ cho phép một người chơi, chưa có chế độ đa người chơi.

#### 5.2. Khả năng thích ứng

- Web có thể chạy được trên phần lớn các trình duyệt web thông dụng: Chrome, Firefox, Edge, Opera, ...
- Hiện tại web mới chỉ thiết kế cho màn hình thiết bị laptop, PC, chưa thực sự responsive với các thiết bị màn hình nhỏ hơn như điện thoại

#### 5.3. Mức độ trải nghiệm

Website này là một trò chơi FPS (First Person Shooter) hay game bắn súng góc nhìn thứ nhất. Nó có đủ tính chất của dòng game này, người chơi điều khiển

nhân vật thông qua góc nhìn từ chính nhân vật đó và tham gia vào một trận đấu súng. Đồ họa 3D sống động, chân thật từ những model quái vật, thùng container đẹp mắt đến những texture thực tế, sống động. Tất cả đã đem lại cho người chơi một trải nghiệm hết sức thú vị, chân thật.

## 6. Hướng phát triển

- Tối ưu nhược điểm đánh giá
  - + Về vấn đề UI, nhóm em sẽ bổ sung thêm những thiết kế phù hợp hơn với trải nghiệm người chơi. Ví dụ, thanh máu của nhân vật ở màn hình chính sẽ chuyển sang màu đỏ để mới nhìn lần đầu ai cũng nhận ra.
  - + Bổ sung thêm các tính năng để trải nghiệm chơi chân thật hơn: tính năng bắn súng tấn công và giết quái vật, di chuyển trên nhiều địa hình, sử dụng các loại vũ khí khác nhau
- Phương hướng phát triển
  - + Nghiên cứu đến khả năng tích hợp đa người chơi
  - + Tối ưu về mặt hiệu năng và performance
  - + Chuyển đổi kiến trúc từ web sang app
  - + Phát triển database để lưu điểm số, thành tích của người chơi.

## 7. Tài liệu tham khảo

- Webpack: <https://webpack.js.org/>
- Threejs: <https://threejs.org/>
- Three-pathfinding: <https://three-pathfinding.donmccurdy.com/>
- Ammos: <https://github.com/kripken/ammo.js/>
- Bullet: <https://pybullet.org/wordpress/>
- Texture: [https://polyhaven.com/a/veld\\_fire](https://polyhaven.com/a/veld_fire)
- PlayerAK47: <https://sketchfab.com/3d-models/ak47-6e51d6ffd33e412a930aff9f520066e1>
- Mutant:  
<https://www.mixamo.com/#/?page=1&query=mutan&type=Character>

## **8. Phân chia công việc**

Công việc	Người thực hiện
Player, AK47	Phạm Thị Hương
Mutant	Nguyễn Thị Linh
Âm thanh	Nguyễn Thị Quỳnh Anh
Physical environment + tìm hiểu các công nghệ sử dụng + lên ý tưởng	Nguyễn Hoàng Anh
Tìm model	Trần Trung Hiếu