Team Project — Requirements Specification

Team KnightLore

David Iozu, James Adey, Joseph Ellis, Kacper Kielak, Thomas Williams & William Miller

Contents

1 Approvals

Approval of the System Requirements indicates an understanding of the purpose and content described in this deliverable. By signing below, each individual agrees with the content contained within.

David Iozu	[illegible]
James Adey	[illegible]
Joseph Ellis	[illegible]
Kacper Kielak	[illegible]
Thomas Williams	Thomas James Williams
William Miller	[illegible]

2 Introduction

We will develop a three-dimensional first-person shooter (FPS) game using a raycaster graphics engine. The game will have an old-school feel and will bear stylistic resemblance to the 1990s hits of the genre such as Quake, DOOM, and Wolfenstein 3D. Despite their age, these games continue to impress players by means of their iconic graphics styles and immersive three-dimensional environments. We are of the firm belief that three-dimensional video games are more effective at submerging the players into the game and generally result in a more memorable playing experience. Therefore, we have made the decision that our game will feature 3D graphics.

3 Gameplay Requirements

After much deliberation about the style of game we wanted to create, we eventually decided that our game will be an arena-based shooter with a focus on fast-paced gameplay. The game should feature a wide variety of different game modes, including both competitive and cooperative play. We have established a total of four unique game modes that we will include in our game. These are:

1. Free-for-all competitive play, where each player in a lobby competes with each other player to score the most points. This game mode will be called 'deathmatch'.
2. Team-based competitive play, where players in a lobby are divided into two or more teams. The winning team is the team which scores the most points. This game mode will be called 'team deathmatch'.
3. Co-operative 'survival' play against AI, where each player in a lobby cooperates in order to survive against a horde of non player characters (NPCs). Scoring factors include the number of enemy NPCs killed and the survival time of the team as a whole. This game mode will be called 'cooperative survival'.
4. Single player 'survival' play against AI. This is the same as the cooperative 'survival' play, but the player must survive against the horde of NPCs alone. This game mode will be called 'single-player survival'.

3.1 Weapons and Powerups

Games within the FPS genre typically emphasize combat-style play between players. Our game will be no different. We require that the players have the ability to use a variety of different weapons, each obtained by finding 'chests' located at random positions within the level.

In order to make weapons distinctive, that each weapon requires following properties:

1. Damage dealt. The damage dealt by any weapon will be a function of the distance away from the affected player. Shotgun type weapons, for example, will deal immense damage at close range but their long-range effectiveness will be very poor.
2. Rate of fire. This determines how quickly the weapon can be used.
3. Ammunition. This determines how many times the weapon can be fired before needing to reload or find new ammunition.

We plan on adding at least four different weapons to the game – the properties of each are yet to be determined. Upon first entering the game, players will start with a pistol-type weapon. This weapon will deal low damage with a low rate of fire.

We also require that our game has powerups to keep the gameplay active. Powerups will randomly appear at certain locations on the map and disappear if they aren't collected within a fixed time period. They will provide players with temporary enhancements to give them a competitive edge. We are going to add the following powerups:

1. Double damage. The player's weapon deals twice as much damage for a fixed time period.
2. Shield. The player is granted a 'shield', which means that any damage dealt to them is instead dealt to the shield until it is destroyed.

We may add more powerups depending on whether we feel the gameplay is lacking during testing.

3.2 Maps

We agree that balance is prudent in tournament-style play; it would be completely unfair for a player (or team) to be put at a disadvantage due to an unfavourable map design. In an effort to keep the game balanced, the game will include numerous maps designed by the team. These levels will be hand-crafted to impose balanced with an emphasis on tactical and interesting gameplay. We plan on creating a total of 5 pre-defined levels for the game.

That said, it is inevitable that players will lose enthusiasm for the hand-crafted maps over time. Therefore, we will be focusing a lot of our efforts on the design and implementation of a procedural map-generation algorithm. Maps will be procedurally generated according to parameters supplied through the UI (ie. map size, weapon frequency, level style, etc). The inclusion of procedurally generated maps will increase variety between matches, allowing the game to stay interesting even after multiple plays.

An additional requirement for the procedural generation of maps is determinism. Maps must be generated deterministically according to a seed. This means that the player(s) can save the seed so that the same map can be played again in the future.

3.3 Scoring

In order to keep a competitive atmosphere between the players, a score convention will be provided. A leaderboard will be available for all players, showing scores and current positions. We hope that this will motivate players to continue playing in order to move up in the leaderboard.

4 System Capabilities

4.1 Major System Capabilities

The game must:

1. be able to be played with networked computers.
2. support a single-player mode.
3. be able to provide computer-controlled 'AI' players.
4. provide both hand-crafted map designs from the player and procedurally generated 'random' maps.
5. provide a level editor such that players can design and play their own hand-crafted maps.
6. support play on Windows, Mac and Linux operating systems.
7. provide real-time network play with a reasonable latency.

4.2 Additional Information on Game Rendering

Our game is required to be 3D. Therefore, to render the game world on the screen, we are going to implement a raycaster graphics engine. Original games in the FPS genre used a raycaster graphics engine to create a pseudo-3D effect – the most pronounced example of which is Wolfenstein 3D. This particular type of graphics engine works by dividing the screen into a series of fixed size 'strips'. For each of these strips, a ray of light is cast. We trace this ray of light until it intersects with a solid wall, then compute the distance between this wall and the player.

This information allows us to calculate the height that we should draw the wall on the screen. The further away the wall, the smaller we draw it. Similarly the closer we are to the wall the larger it should be rendered on screen.

Figure 1 illustrates the concept of raytracing graphically.

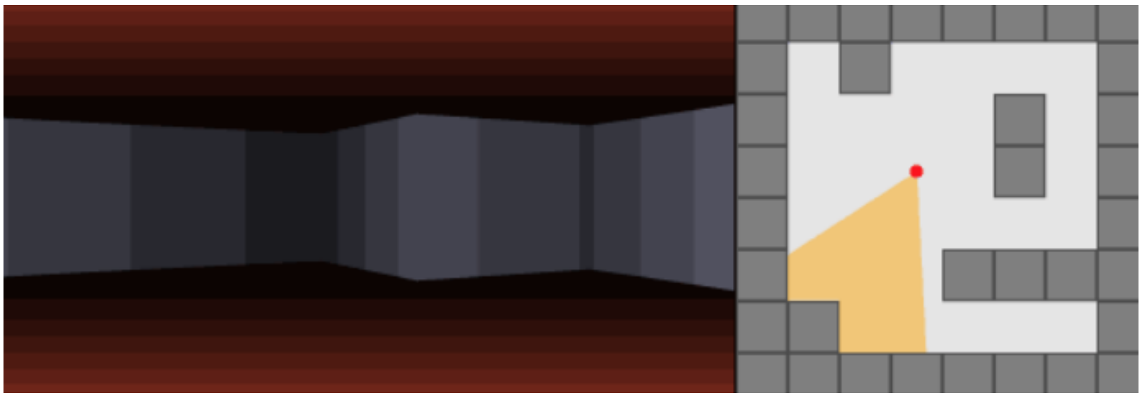


Figure 1: Example of raycaster rendering, image from Wikipedia. [1]

We require that our raycaster is implemented using Java's standard 2D graphics library. We will *not* be using any non-standard libraries to assist us with the 3D rendering.

Our game-world will consist of a grid of tiles, which are rendered as fixed-sized blocks from the perspective of the players. Refer to Figure 1 for an example.

Our graphics renderer should support:

1. The rendering of opaque *un*textured tiles.
2. The rendering of opaque textured tiles.
3. The rendering of translucent textured tiles. Blocks will have an 'opacity' property, where a smaller opacity indicates that the block is more transparent. This opacity property will be utilised when rendering the block on the screen. We feel that adding transparent/translucent blocks would not only look aesthetically pleasing, but offer interesting gameplay dynamics where players can see one another's position but cannot shoot each other. Additional requirements regarding transparency/translucency include:
   (a) Transparency should be an additive, compounding property. Many games implement tile transparency naïvely, whereby you cannot see any transparent objects whilst looking through something transparent. This is often put in place to simplify the rendering process. In an effort to make the graphics more appealing, we require that our rendering does not follow this naïve approach and instead will render compounding layers of transparent objects to better reflect how light behaves in the real world.
   (b) We require that the opacity of a block can change dynamically at runtime.
4. We require that the texture of a block can be animated, or otherwise change dynamically at runtime.
5. The rendering of sprites, whether these are other players, NPCs/AIPs, or weapons/powerups.
6. Pseudo-lighting. We plan on implementing lighting as a function of distance from the player, where blocks appear darker the further away that they are from the player.
7. Fog. In certain environments, we plan on adding a 'fog' element to the scene.

To facilitate the last two points of the above, we plan on implementing an environment interface to allow for the modular creation of unique environments within the game. Figure 2, taken from a rapid prototype of the raycaster, illustrate how environments can be used to modify the overall look and feel of a map.

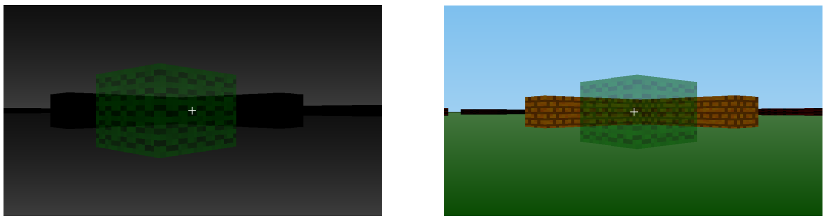


Figure 2: A dark and light outdoors environment.

4.2.1 Raycaster Drawbacks

Raycasters allow us to present a convincing three-dimensional environment to the player. Despite this, they are not perfect. Raycasters essentially render a three-dimensional perspective of what is internally represented as a top-down, two dimensional game. It is for this reason that they are called 'pseudo-3D renderers' – within the internal game representation, coordinates and position are represented with a two-dimensional vector instead of a three-dimensional vector.

Unfortunately, this means that the concept of height in our game will be difficult to achieve. At this point, we expect that all maps will feature a single height layer explorable by the player. Using a height map matrix it is possible to simulate height, but the results are variable and often undesirable for this particular style of play.

In order to compensate for the lack of true height in our game, we plan on compounding maps together and allowing the player to go from one map to the other through gateways or doors. For example: we could have a map for a basement and a map for the ground floor. We could link the two maps together using a gateway/door that teleports the player from one of the maps to the other.

To further immerse the player in the game, we will include different environments for the sub-maps. For instance, the previously mentioned basement level will have a dark environment, and the ground floor might be slightly brighter. We feel that this variety will allow for more entertaining and dynamic gameplay.

4.2.2 Art Assets

A raycaster is only capable of rendering very basic geometric shapes. Therefore, to represent enemies, items, and other players, our game requires the use of sprites. We require that sprites:

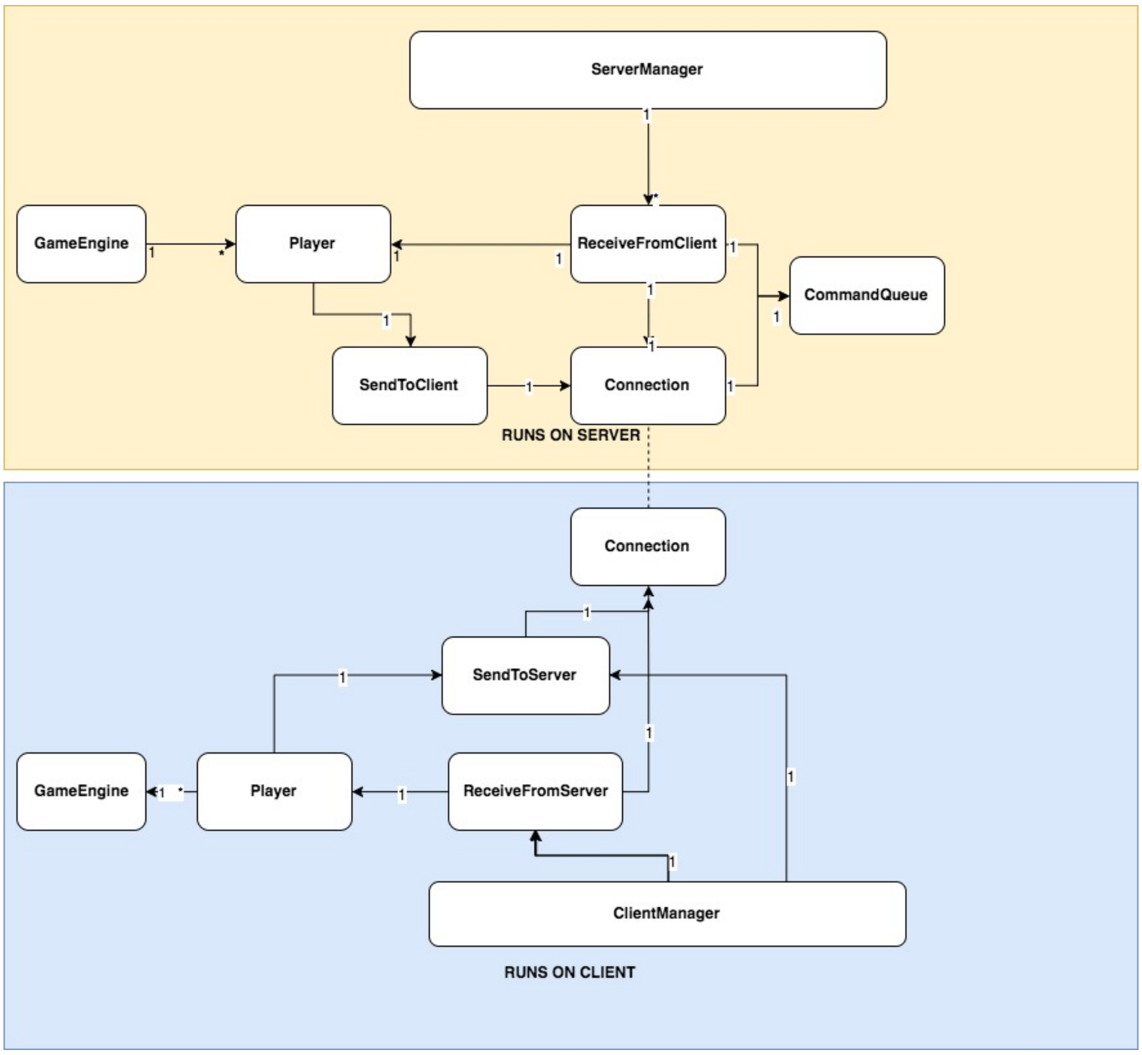
1. have directional alternatives. When the player is looking at a sprite in game, the graphic which is displayed should depend on the player's orientation relative to which direction the sprite is facing. To achieve a well-rounded directional effect, we will have 8 directional graphics for each sprite to represent each of the ways it could be facing.

2. are occluded properly. This means that if a sprite is partially blocked by a wall, we only render the part visible.

Sprites will be produced from 3D models as well as by hand painting. 3D models will be rendered from 8 directions in order to produce sprite sheets. We will produce a script to automatically generate these sprite sheets based on 3D models.

4.3 Network Architecture

The following is a draft class architecture for the networking component of the system. The dotted line connecting the server and client sections represents a socket connection over a network. This should implemented by an abstract 'Connection' class, which will initially be implemented using TCP, as a naïve proof of concept. Once the basic multi-player networking architecture is set up, we will evaluate the latency and optimise as necessary, for example by implementing virtual connections over UDP [2]. In order to reduce the appearance of lag, a prediction algorithm will be implemented on the client side keeping the server's full authority over the game state. A custom low-level protocol will be implemented to facilitate the packet transmission between server and client. This will ensure that the connection will be fast enough for a real-time game.



4.3.1 Server Commands

The server owner has ultimate rights over the games being played on the server. We require that the server owner has the following capabilities:

- Kick a player from the game. If a player is particularly disruptive in game or in chat (often called 'griefing' in gamer terms), the owner may want to remove them.
- Change the current map that is being played.
- Set the level generation parameters.
- Restart the current game round.
- Shutdown the server.

4.4 Procedural Generation

We require that the procedural generation process be broken down into two parts: room generation and path generation.

Rooms will be randomly placed on a grid. These will either be rectangular, with random width and height, or a more bizarre shape generated using techniques like simplex noise. Once generated, rooms will be populated with tiles determined by a room type (ie. a player spawn room, a room abundant in weapons, etc.) The room types present will be determined by parameters passed to random generation.

Paths will be created between rooms such that each room is accessible to each player. This will be guaranteed by producing the minimal spanning tree between rooms, and arcs can then be added if greater room connectivity is desired. Will be use A* to draw the optimal path between rooms. Non-optimal paths will be generated by using the simplex noise in our A* algorithm. Non-optimal paths may be requested as they arguably lead to maps with a more interesting shape.

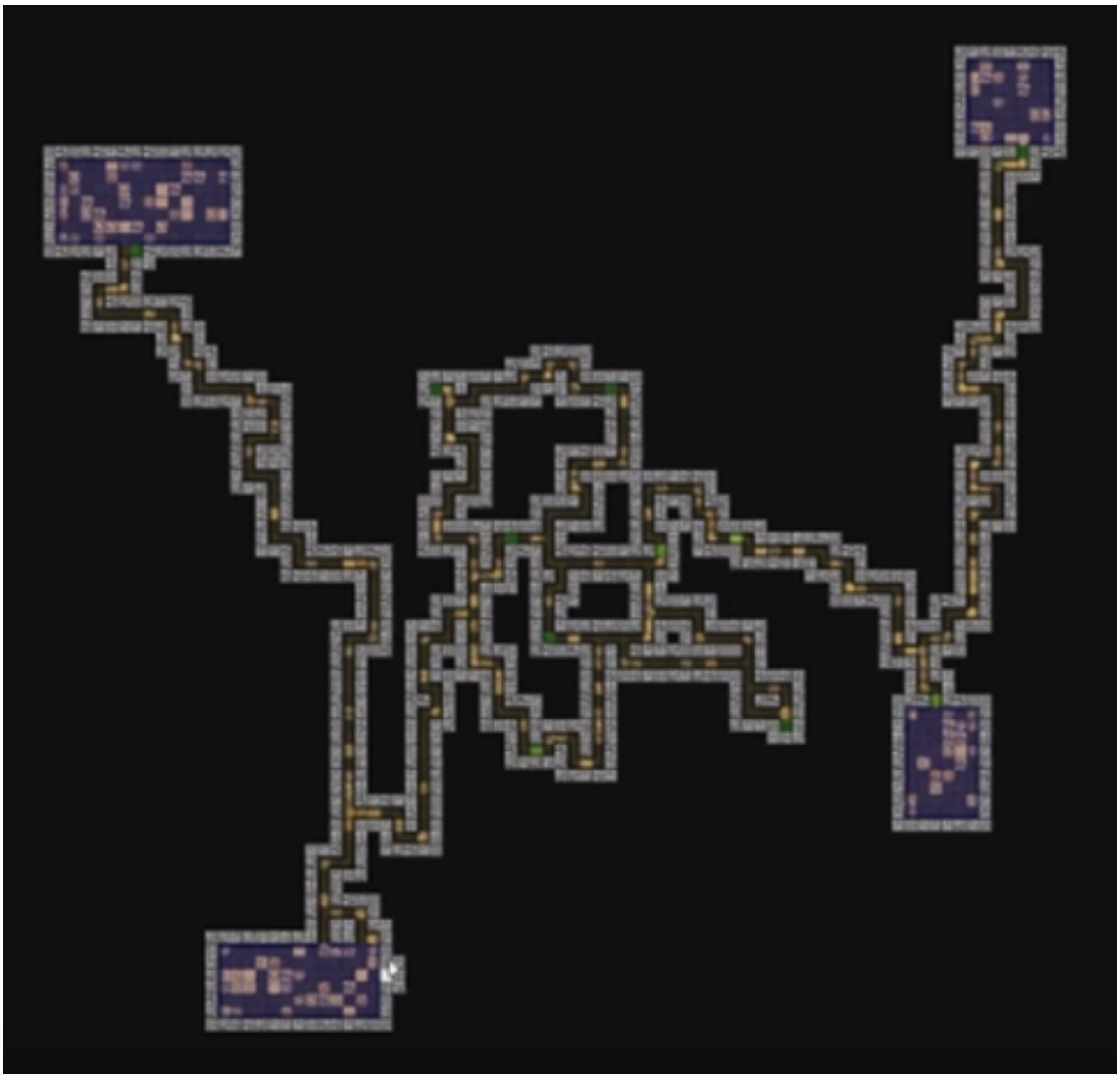


Figure 3: Screenshot from YouTube video on procedural generation [3]

A common feature in team-based games is symmetrical maps. To ensure fairness between teams, we will provide an option to the user to allow generated maps to be flipped along an axis to give symmetry. Team-specific rooms will differ to their reflected counterpart (i.e. a team A spawn point becomes a team B spawn point). A different number of reflections can be performed to accommodate a different number of teams: one reflection for two teams, two reflections for four teams). Similar to other features, symmetry or asymmetry can be specified in parameters handed to generation.

4.4.1 Non-Player Characters

Non player character (NPCs) are AIs within our game that do not resemble player characters. They will behave differently to the "typical player" and require less sophisticated AI (such as A* algorithm for path-finding). NPCs will feature in survival game modes, in which a horde of enemies work against the player. They will be aware of the player(s) position at all times – this is reminiscent of many 'zombie survival games' where it is impossible to hide from the enemy. Instead of hiding, the player(s) must fend off a swarm of enemy NPCs. The "difficulty" of NPCs in survival modes will be variable, with the game become progressively harder over time.

4.4.2 Artificially Intelligent Players

Artificially intelligent players (AIPs) will play a prominent role in certain game modes of our game. AIPs, unlike NPCs, serve the same function as player characters and will be considerably more sophisticated.

The specific requirements we have in mind for the AI aspects of our game are as follows:

1. The AIPs must use some pathfinding and/or reinforcement learning algorithms to travel through the map and interact with players and game objects.
2. When a player leaves the game (be it due to poor connectivity or other reason), they will be replaced with an AIP until they rejoin the game.
3. Some attributes of AIPs can be manually adjusted, such as health and accuracy.
4. Weak AIPs should be included in the tutorial level as target practice for players.

One notable detail about the implementation of AIPs is that they will be controlled exclusively by the server. Even in single-player survival mode, the player will be connected to a server (which may be run locally) that will handle all aspects of the AIPs playing.

4.5 Audio

4.5.1 Sound Effects

Audio feedback should be provided on major game events. This could include:

- shots being fired
- player movement
- effect/power-up activation
- spawning
- damage being taken

We plan on implementing directional sound for some of these effects. For example, the sound of a shot being fired can be panned appropriately to match the direction of the shot, and its volume can be determined by the distance from the shot.

4.5.2 Background Music

Appropriate background music should be played. This should be a looped track, to avoid gaps during the game. This should either be generated using a DAW and imported as a resource, or procedurally generated within the game. Different layers of the music should be able to be stripped/added depending on the current area of the map.

4.6 Major System Conditions

The game must:

1. allow networked play in lobbies of up to 8 players.
2. be programmed in the Java programming language.

As an additional challenge, we are going to program the game with minimal support from non-standard libraries (e.g. LWJGL, OpenGL). Not only will this allow us to build a game that we know extensively from the ground up, it will also allow us to modify any functionality with ease in the future.

4.7 System User Characteristics

1. Players. These will connect as "clients" in the network model.
2. Server Host. This is the person running the server application which listens for connections.

4.8 User Interface Design

4.8.1 Client Interface

The client program will require a GUI for the players to interact with. The exact layout and appearance for the GUI is left to implementation and will be heavily informed by user feedback. Here, we outline the capabilities that each GUI section must have:

Main Menu

- start the tutorial.
- start a survival game.
- join an existing multiplayer game.
- access the Options Menu.
- exit the game.

Options Menu

- change the screen resolution.
- change the 'blockiness' of the raycaster (more rays or less). This can be used to give the game more of an old school feel, as well as allowing players on a low-spec computer to achieve a reasonable framerate.
- change the draw distance (for lower spec machines).
- change the audio volume
- enable/disable special graphical effects
- close the Options Menu.

Heads Up Display (HUD)

- display player status.
- display victory status.
- display localized player area or "minimap".

In-Game Menu

- disconnect from the server and return to main menu.
- request to change teams.
- access the Options Menu.
- close the In-Game Menu.

Chat Display

- show a history of chat messages.
- allow the player to enter a chat message.
- allow the user to specify the target for their chat message.
- close the Chat Display.

Team Display

- show an overview of the status of all team members.
- show a detailed breakdown of the victory status.
- close the Team Display.

4.8.2 Server Interface

The server program will operate "headless", without a graphical component, this is to decrease the processing load on the server as it is responsible for running the AI as well as forwarding all the network traffic. The server owner can interact with the program via a simple command line interface to issue the server commands (see networking).

4.9 Level Creator

Our game will include a tool for creating and editing new levels/maps for the game. The level creator will require a graphical user interface. Here, we outline the core requirements of the Level Creator:

- allow the user to place tiles into the world.
- allow the user to place prefabricated Game Objects into the world.
- allow the user to specify the core data for the level (game modes, visual style, dimensions etc).
- allow the user to preview the level (view as if in-game).
- load an existing level for editing.
- save a created level.
- exit the Level Editor.

5 User Stories

- As a user, I want to be able to play against other human opponents.
- As a user, I want to have the option to play against computer-controlled players.
- As a user, I want to have the option to disable certain graphical effects to optimize performance in the case that I am running on a low-spec machine.
- As a user, I would like the option to disable purely aesthetic motion-related aspects of the game in order to prevent motion sickness.
- As a user, I would like the option to adjust the volume of in-game sound effects and music independently.
- As a single player, I want to be able to view my own status (score, health etc.), as well as that of the other players.
- As a member of a team, I want to be able to view the status of each team.
- As a member of a team, I want to be able to choose whether or not to enable friendly fire.
- As a player, I want to be able to communicate with the other players, and choose the scope of my conversation (global, team, etc.)
- As a server owner, I want to be able to customize the game mode running on my server.
- As a server owner, I want to be able to kick and ban disruptive players from using my server.

6 Policy and Regulation Requirements

The system is strictly required to store absolutely no data relating to real persons. Players will be required to enter an alias name, which is temporary and local to the server that they are connecting to. This alias will be used to identify players during the game. No permanent data will be stored on the local machine or server.

7 Security Requirements

1. The game should protect against man-in-the-middle packet manipulation. Man-in-the-middle attacks may sound innocuous for a simple game such as ours, but under the right circumstances packet manipulation can be used to give dishonest players a big advantage, particularly in the context of events such as tournaments and LAN parties. In order to mitigate MITM attacks, we plan on encrypting the communication between the clients and server. This prevents dishonest users from manipulating packets in transit.

2. The game should prevent against replay attacks. Consider a packet that informs the server that the client has shot *once*. In a replay attack, a malicious user would replay the packet to the server multiple times, clearing out the victim's ammo and leaving them vulnerable in game. We will mitigate this attack by including a unique, valid-once counter in each packet. The server will reject any packets whose counter is not the expected value.

3. The client-server architecture should ensure that the server stores the definitive state of the game, to mitigate attacks from rogue clients. At no point should the client be responsible for maintaining the state of the game.

After an initial discussion about encrypting network communications, we were concerned that the encryption and decryption of packets would create significant overhead and network latency. However, after some further research and tests, we are confident that employing network encryption will have a negligible impact on overall latency. In most modern processors, cryptographic operations are part of the standard instruction set.

8 Training Requirements

We aim to provide a tutorial or training level which will teach players the basic controls and elements of the video game in an interactive in-game environment. Failing this, a game manual should be provided, such that someone with no prior exposure to the game could pick it up and play it.

In addition to providing instructions for gameplay, we will also document the process of setting up a game server. The format of this may be a document or a video guide, depending on which is more appropriate given the server's capabilities.

9 Initial Capacity Requirements

The game must:

1. support at least 6 human players in networked play.

2. support maps of a size of at least 128x128 tiles.

3. support a minimum of 4 AIPs simultaneously.

4. support a minimum of 10 NPCs simultaneously.

10 Core Game Engine Architecture

The game engine will be implemented with a composition model. The core base class for all objects in the game will be a GameObject. Any entities/game components will derive from GameObject, so that they are processed within the engine.

The game engine will only provide object storing/updating and will serve as the main entry point of the game. It will not be linked to other core aspects of the game (networking, renderer, etc...). This is so that the game creation can be modularized, and to permit 'headless' dedicated servers that do not render the game on screen.

The game engine will be kept feature-light and delegate responsibility of game features to GameObjects. This is to give programmers maximum control over how the features are implemented, and to keep the core engine decoupled from gameplay.

10.1 Game Objects

The GameObject class will provide data that is commonly useful in many game components, such as position and rotation. Additionally, it will provide key functions from the engine, such as:

- onCreate: called once the object has been registered in the engine, and before it is first updated.
- onUpdate: called once per game-frame.
- onDestroy: called once the object has been removed from the engine, the object will no longer be updated by the engine.

11 Testing

11.1 Automated Testing

1. Unit tests should be written to ensure the core elements of each part of the program work as expected. We will write unit tests to cover:
   (a) The core game engine.
   (b) The renderer.
   (c) The server and client network interface.
   (d) The audio system.
2. Integration/functional/other higher-level tests could also be written.
3. Once we have a stable prototype on the master branch, we could set up a CI server (e.g. Jenkins) to continuously check for regressions.

11.2 Manual Testing

1. All team members will be expected to test their individual features as they commit them.
2. Later on in the process, users can be enlisted to beta test the game, and give feedback. We hope to offer the game to other students on the course, as well as from different departments, to play the game and provide feedback. We will set up a server for them to play on.

12 System Acceptance Criteria

1. All of the Major System Conditions (4.6) should be satisfied.
2. The results of user beta testing should be to our satisfaction.
3. All unit tests as specified in Automated Testing (11.1) should pass.

13 Current System Analysis

N/A: no current system exists.

Glossary

AIP

An **A**rtificially **I**ntelligent **P**layer. AIPs serve as substitutes for human players.

DAW

Digital Audio Workstation.

FPS

First-**P**erson **S**hooter: a video game genre centered around gun and other weapon-based combat in a first-person perspective.

FPS

Frames **P**er **S**econd: the number of display updates displayed on screen per second.

NPC

Non-**P**layer **C**haracter. Distinct from an AIP in that it does not mimic player characters.

Perlin Noise

Perlin noise uses a function to smoothly generate noise over a given space. Perlin noise takes a coordinate in space and returns a number between -1 and 1.

Raycasting

The technique of rendering a 3D world based on a 2D map.

Sprite

A bitmap image composited onto a larger scene.

Sprite Sheets

A Texture that contains multiple smaller images "sprites", that are arranged in a specific order.

References

[1] Screenshot of raycaster demonstration from Wikipedia.
https://en.wikipedia.org/wiki/Ray_casting

[2] Virtual Connection over UDP,
https://gafferongames.com/post/virtual_connection_over_udp/

[3] Screenshot of Genome2D Devcast - Procedural Dungeon Generation
https://www.youtube.com/watch?v=Z_n3cafMU-Y