Functional Specification for Dungeons and Crawlers

Team Members:
Ethan Doyle - 22497082
Caolan William Cochrane - 22490802

TABLE OF CONTENTS

1. INTRODUCTION

1.1 Overview

The system in development is a procedurally generated multiplayer block game, that combines dynamic storytelling and dialogue generation.

This is to solve the problem of replayability in video games. Dedicated story games can only be experienced for the first time once, then replayed. Procedurally generated games struggle to have a coherent story that isn't tied to in-game world structure or items, and it's hard to tell an immersive story this way. They become boring after a while. A lot of games try to simulate conversations by recording or writing a predefined set of dialogue beforehand. This is limited, and re-experiencing the dialogue breaks immersion, and makes it very hard to draw a personal connection to an NPC.

We will allow users to define the style and theme of the game world through the use of a prompt system. This prompt will have an effect on in-game dialogue and story narrative. This allows players to define their own unique experiences and modify and tailor them repeatedly.

The combination of the procedurally generated world with the dynamic NPC and quest generation allows for an incomprehensible amount of exploration. For players this means that they can seek out their own place in the virtual world, whether it might be a city, a forest, a mountain etc. The idea is that they never run out of ideas for things to do. Larger and more interesting quests might be tied to more interesting NPCs, allowing players to seek out the kind of quest they wish to embark on, and draw personal connections to the NPC's they meet along the way. The game will be seed driven allowing for replayability, and the same seed and prompt will produce the same world, so an experience can be re-experienced.

1.3 Glossary

1. **Replication:**
2. **Synthetic Database:**
   Synthetic data is artificial data that retains the characteristics of real-world data.
3. **Differential Updates:**
   An update that only has the differences between the previous build version and the new build version.
4. **Context Window:**
   The amount of text (in tokens) that a large language model can remember and use at any time.
5. **Inference Time:**

6. **LLama3.2-1B:**
   An open source LLM model released by Meta, containing 1 Billion parameters. It uses in the range of 4-8GB of VRAM at inference time.
7. **Listen-Server Architecture:**
   A listen server architecture is composed of a user running a game on their machine with this user acting as the server.
8. **Procedural Generation:**
   A method of creating data utilising an algorithm instead of creating data manually.
9. **Entity-Component System:**
   A software architecture pattern used as a representation of game world objects.
10. **NPC:**
    Non-Player Character
11. **LLM:**
    Large language model
12. **ThreeJS:**
    A JavaScript cross-browser library and API used to animate and display 3D computer graphics.
13. Websockets
14. flatbuffers

2. General Description

2.1 Product / System Functions

The intention of this system is to incorporate a procedurally generated, multiplayer web-based game driven by AI-generated storytelling and dialogue.

In our most ambitious implementation, we would like users to be able to define the themes and concepts of their game world with a prompt. The LLM would generate and integrate these features accordingly, having an influence on dialogue and world generation.

For the initial implementation, our focus is on building a strong foundation of AI-driven dialogue and procedural world generation that we can expand over time.

The major functions of our system include:

- **AI Driven Storytelling and Dialogue Generation:**
    - An LLM will generate dialogue and storytelling content, making the experience more varied and immersive for players when interacting with NPC's.
- **Procedural World Generation:**
    - The world will be generated using a unique seed, with the possibility of adding prompt-based elements over time. Terrain generation systems will be module and loosely coupled, allowing expandability and changes down the line.
- **Real-Time Multiplayer:**
    - We will be using WebSockets to enable real-time multiplayer gameplay. Player data and preferences will be stored through the use of cookies.
    - Real time replication is essential. Our frontend will implement client-side prediction and other methods to ensure a smooth player experience.

Additionally, this is a list of our main functional requirements, which will be elaborated on in section 3. We will add more if we believe there is anything worth adding.

- Replication
- User data saving
- Create a world
- Host a multiplayer world
- Join a multiplayer game
- Procedural generation
- Basic controls
- NPC interactions
- NPC generation
- Quest system
- Inventory system

2.2 User Characteristics and Objectives

We foresee that the main users of our systems will be game players - those that are familiar with and have an interest in playing video games, as well as those new to the video game landscape. Programmers and computer scientists may also be interested.

While our expected expertise for this system is a familiarity with videogames, we will also accommodate some things for non-game players and for those not interested in the story aspect, such as side events showcasing and an on-boarding tutorial.

We will accommodate the expectations of our expected users by providing:

- Good Graphics
- Fluid Controls
- Seamless Networking
- A polished product, free from excessive bugs
- An onboarding tutorial
- For those interested specifically in the NPC's themselves, we will integrate a small simulation/showcase option. This allows non game players interested specifically in the interactions between NPC's to experience it without becoming accustomed.

2.3 Operational Scenarios

- **Creating a New World:**
    - The user would open the game and select the option to create a new world. The user would be given the option to enter in a pre-existing seed to generate a replica of a pre-existing world.
    - If the user wishes to create an entirely new world, they would be given a prompt system to define the world's style. They would be able to enter in a theme and adjust some other parameters like the tone of the story.

- **Joining a Hosted World:**
    - The user will also have the option to join worlds hosted by other players through entering in an IP address. There will be an option in the main menu to facilitate this. Other users can secure their world by having a password and if so, the user will also have to enter in a password to join the world.
    - Upon joining, the user will automatically be put into a multiplayer world with other players that are already on that session.

- **Hosting a World:**
    - The users themselves will also have the option of hosting their own world. Upon creating a new world or when playing one of their pre-existing worlds, the user can go into the menu to host the server.
    - The user also has the ability to enter in a password to secure their server.

- **Game Event Based on World's History:**
    - Throughout the game, the user will be able to experience events that are based and informed off of the world's history.
    - An example of this event would be a newspaper system, where the user would be able to read newspaper articles based on their actions or quest completion in the world.

- **NPC Interactions:**
    - The user will be able to interact with NPCs in the game. The NPCs will have nuanced personalities that will be clear in their dialogue interactions with the player.
    - Players will also be able to visually interact with NPCs by watching how they talk to one another and go about their daily lives (they have a home they retreat to, a job they go to etc.)

2.4 Constraints

- **Hardware & memory constraints:**
    - Since we are using a listen server architecture, the ping and latency would be dependent on the machine hosting the game server. This means that an adequate internet connection and device can be necessary to play the game well. Additionally, there are other factors that should be considered such as real world distance and other outside factors such as weather. This means that to test our system, we will need adequate devices and internet connections ourselves.
    - Our LLM model requires the use of a GPU in order to be loaded. These requirements are different at training time, but a mid-range graphics card has the required VRAM for hosting the model at inference time.

- **Time constraints:**
    - A large problem with our system was a worry that we were being too ambitious in our design. There is a possibility that we will not be able to meet the deadline if we want to include all our features. We may have to crack down on some of our features, such as good graphics or some features related to procedural generation (not the whole thing, of course).

- **AI constraints:**
    - Given the AI aspect to our system, there is always a chance that something can go wrong in terms of character dialogue. We specifically aim to remedy this by fine-tuning our models as well as possible, as well as including an option to "reroll" or to request the AI to generate new dialogue.

3. Functional Requirements

This section lists the functional requirements in ranked order. Functional requirements describe the possible effects of a software system, in other words, *what* the system must accomplish. Other kinds of requirements (such as interface requirements, performance requirements, or reliability requirements) describe *how* the system accomplishes its functional requirements.

3.1 Replication

- **Description:**
    - Client side interactions and server side interactions with the server must be synchronized across all players in real-time, meaning everyone sees the same things happening at the same time. This has to feel seamless to ensure an enjoyable networked experience.

- **Criticality:**
    - Critical. Without synchronization, the game would feel out of sync, making multiplayer impossible to enjoy.

- **Technical Issues:**
    - To handle lag, we'll use client-side prediction, where the client "guesses" what will happen next and then adjusts when the server updates. The tricky part is ensuring everything stays consistent across all clients, especially in a big multiplayer world.

- **Dependencies with Other Requirements:**
    - This depends on the network architecture and chunk loading working well because the world has to load fast and reliably for everyone to see the same thing. (Also tied into NPC behavior, since NPCs need to be synchronized across players in the same way as the world and player actions.)

3.2 User Data Saving

- **Description:**
    - This function is done automatically by our system. The users will obviously want a way to be able to save their worlds and other data. This will be done through the use of browser cookies - small blocks of data which saves information from the user's session - which will save all of the user's data to the website.

- **Criticality:**
    - This function is essentially for allowing users to play in the long term - if every time they closed the website their worlds were deleted, then what would be

the point of playing? Additionally, this data needs to be saved to be served to the LLM as context.

- **Technical issues:**
    - There will be no technical issues involved in satisfying this requirement, as it is already done automatically by most browsers.

- **Dependencies with other requirements:**
    - None.

3.3 Create a World

- **Description:**
    - This function allows users to create a world. On the menu screen, players will be given the option to create a new world, and when navigating into this option, they will be presented with the option to enter in a seed and a prompt. The user can enter in a seed to generate a replica of a pre-existing world or they can enter in a prompt to generate a new world with the style and tone of the world being based off of this prompt.
    - A world is composed of a number of chunks, spreading virtually infinitely in all directions.

- **Criticality:**
    - This function is critical to our overall system as our whole system is based around the game and procedural generation in these worlds. The seed is very important for world generation

- **Technical issues:**
    - The above will be accomplished through a simple HTTP form which will be passed to our LLM. We will also give examples of prompts to users so they will be aware of what it entails.

- **Dependencies with other requirements:**
    - None.

3.4 Host a Multiplayer World

- **Description:**
    - This function allows users to host their world once they are in-game. Once in-game, a user can navigate to the menu, and they will have the option to host their world to other players. They can also secure the connection to this world with a password, making it so that when another player tries to join, they will also have to enter in a password to gain access.

- **Criticality:**

    - This function is essential as we are specifically making a multiplayer-based game. We want users to be able to connect to each other and play together and this is the main way to achieve this.

- **Technical issues:**
    - The server hosting will be done through a listen server architecture, which means that the server will be hosted on the host's machine, i.e. the user hosting their world becomes the server host.
    - NAT Type issues, Port Forwarding requirements; we don't want to make it a hassle for our users to start local games and share them.

- **Dependencies with other requirements:**
    - Requires a world to actually exist and be created.

3.5 Join a Multiplayer World

- **Description:**
    - This function allows users to join a hosted world. In the menu screen, there will be an option to enter another user's IP address. Once entered, the user will begin connecting to this other user. If the world is encrypted with a password, then the user has to enter in a password as well. Once connected, the user will be allowed to play multiplayer with the server host.

- **Criticality:**
    - Similarly to above, this function is essential as this is a multiplayer-based game system.

- **Technical issues:**
    - We will need clear error messages as to why a client can't connect to a host. This can happen for an array or reasons and it's important we give the client context to avoid frustration.

- **Dependencies with other requirements:**
    - Depends on world creation and world hosting.

3.6 Procedural Generation

- **Description:**
    - This function is one of the backbones of our system. When a world is created, it will be procedurally generated, which means that the world is generated based on an algorithm instead of somebody building it manually. This algorithm is based on the prompt defined in 3.2 during world creation and also on the seed.

- **Criticality:**

    - This function is critical. Our worlds being generated procedurally is a very important factor to our system.

- **Technical issues:**
    - World is virtually infinite; can't load it all at once. Have to carefully consider how world state is saved and loaded.

- **Dependencies with other requirements:**
    - Depends on the prompt defined in world creation.

3.7 Basic Controls

- **Description:**
    - This function is for basic controls for gameplay. Users will have basic movement controls, such as moving up, down, left, right, and the ability to jump, attack and interact with objects/NPCs in various ways (break, place, etc).

- **Criticality:**
    - This is fairly important to allow users to actually be able to interact with the game world and to show the system in its fullest.

- **Technical issues:**
    - Allowing users to define their controls and not coupling them to predefined keys. Has to feel seamless.

- **Dependencies with other requirements:**
    - Depends on context. Controls might be disabled when entering a UI.

3.8 NPC Interactions

- **Description:**
    - NPCs must be able to interact with the player through dialogue. This could be a small greeting, a request to leave them alone or an entire conversation.

- **Criticality:**
    - Critical, it is intrinsic to our selling point

- **Technical issues:**
    - Not all NPCs might be interested in talking to the player. Have to find a way to highlight certain NPCs that are interested in talking to the player.
    - Dialogue is LLM generated, with the NPC's json file as input. Have to generate this ahead of time. How much dialogue can we generate on a given system?

3.9 NPC Generation

- **Description:**
    - We generate an NPC in the form of a json file.

- **Criticality:**
    - Critical, it is intrinsic to our selling point

- **Technical issues:**
    - Context window of the LLM is huge, but processing tokens takes O(n^2) in the worst case, though there are some optimizations.
        - Need to establish benchmarks
        - Need to carefully fine-tune our prompting
    - NPCs need context on other NPCs to define complex relationships

- **Dependencies with other requirements:**
    - NPC Interactions

3.10 Quest System

- **Description:**
    - This function is for a quest system. The game system will contain quests, initially with only a very simple fetch quest.

- **Criticality:**
    - High, as quests are an incentive for players to interact with NPCs and to experience their dialogue

- **Technical issues:**
    - T

- **Dependencies with other requirements:**
    - NPC Generation
    - NPC Interaction

3.11 Inventory System

- **Description:**
    - This function details an inventory system for users to be able to manage items in their inventory. They can move items around, drop them or equip them to their character. There would be a button to open the inventory.

- **Criticality:**
    - Not essential, can be an optional feature.

- **Technical issues:**

    - Inventory button will be mapped to a key or will be accessed through a simple click on the UI.

- **Dependencies with other requirements:**
    - Basic controls

4. System Architecture

4.1 System Architecture Diagram

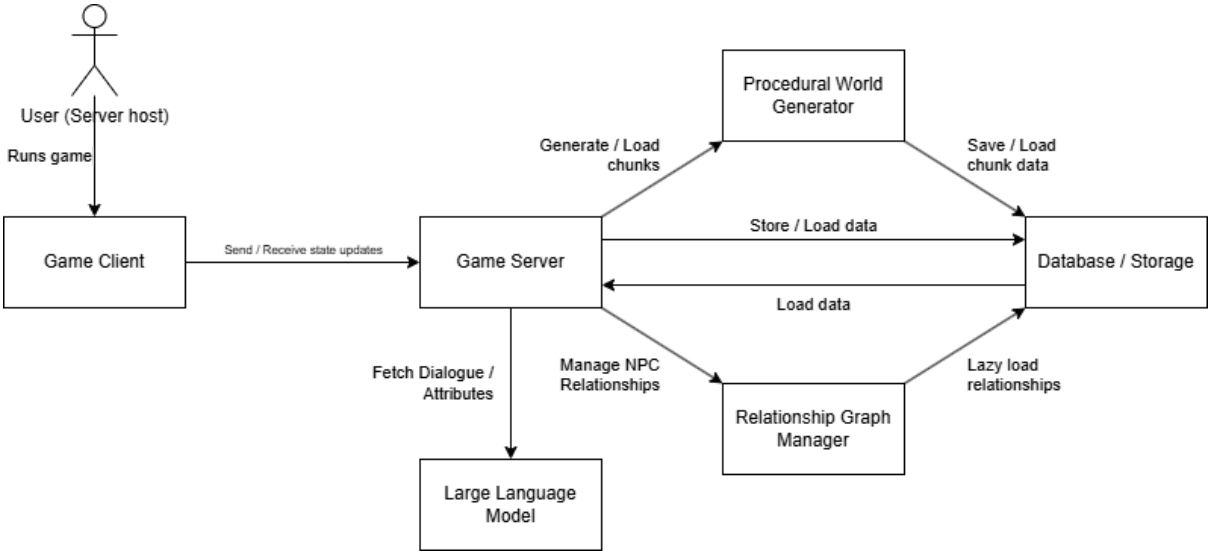


Fig 4.1: System Architecture Diagram

4.1.1 Reused Components

- **Three.js:**
  - Three.js is a Javascript library. It is used to render 3D graphics in the browser which enables smooth visualization of our worlds.
- **Llama3.2-1B:**
  - The Llama 3.2 collection of multilingual large language models (LLMs) is a collection of pretrained and instruction-tuned generative models in 1B and 3B sizes (text in/text out). [1] It is developed by Meta. We use it to generate dialogue and story elements.
- **PostgreSQL:**
  - PostgreSQL is a powerful, open source object-relational database system with over 35 years of active development that has earned it a strong reputation for reliability, feature robustness, and performance. [2]
  - PostgreSQL helps to store data about the world and may also be able to store player data.
- **Perlin Noise / Simplex Noise:**
  - Perlin noise is a popular procedural generation algorithm invented by Ken Perlin. It can be used to generate things like textures and terrain procedurally, meaning without them being manually made by an artist or designer. [3]
  - Essentially, perlin and simplex noise are functions that produce noise (random-like values) but the values change smoothly and consistently. Simplex is designed to be more efficient by using polygons and works perfectly well in the 3rd dimension (which is our max scope)

    - We use these procedural algorithms for terrain and world generation.

4.1.2 Architecture Overview

Figure 4.1 shows our main system components with the addition of a user (the server host in this instance). The server components are as follows:

- **Game Client:**
    - The game client handles rendering, user input and displays real world data and interactions. It utilises Three.js for graphics rendering and websockets for real-time multiplayer communication.
    - It connects to the game server for data synchronization and multiplayer interactions.

- **Game Server:**
    - Manages the game state, multiplayer sync, world replication and interactions with procedural generation. It is built with Node.js and websockets for server-client communication.
    - The game server stores and loads persistent data from the database and interfaces with the procedural world generator to create unique game environments. It also interfaces with the relationship graph manager to manage NPC relationships and statistics.

- **Large Language Model:**
    - The role of the large language model is to handle and generate dynamic NPC dialogue and story content based on player input and world events. It is powered by Meta's Llama3.2-1B LLM. We may have the ability to swap out models depending on how they perform.
    - The game server fetches dialogue / attributes from the LLM while the LLM communicates with the Game Server to receive world context.

- **Procedural World Generator:**
    - The procedural world generator generates unique game worlds based on seeds and optional prompts. It ensures that each world is procedurally generated based on perlin noise or simplex noise algorithms for terrain generation.
    - It works with the Game Server to generate and synchronise worlds among players, then sends data to the Database / Storage which is sent back to the Game Server.

- **Relationship Graph Manager:**
    - The relationship graph manager handles NPC relationships. It works with the Game Server to change these statistics for NPCs. then sends data to the Database / Storage which is once again sent back to the Game Server.

- **Database / Storage:**

- The Database's role is to store persistent player data, world states and NPC interactions. It interacts with and gets information from the Game Server, the Procedural World Generator and the Relationship Graph Manager. It then sends this information back to the Game Server.

5. High-Level Design

5.1 Communication Diagram

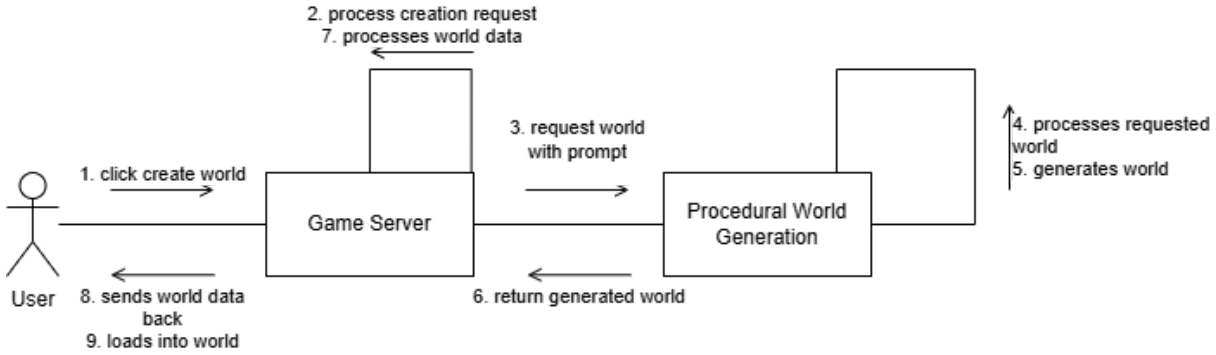


Fig 5.1: A communication diagram, showing the relationship between the user, game server and procedural world generation components.

Fig 5.1 shows us a clear example of the relationship between the user, game server and procedural world generation components.

1. The user clicks create world which takes them to the world creation screen.
2. The user enters in a prompt. The game server takes this request, then processes the request, performing initial checks for input.
3. The server then sends the world request with the user's selected prompt to the procedural world generator.
4. The procedural world generation will begin processing the requested world.
5. It will then begin to generate said world.
6. The generated world is sent back to the game server.
7. The game server processes the world data and prepares it for gameplay.
8. The world data is sent back to the user.
9. The user is allowed to load into the world and begin interacting with it.

Backend

As mentioned, our architecture

World Generation

A world is divided into 'chunks'.
A chunk is an 16 (resizeable, will have to await implementation) 3d array.
Each cell in the array contains a 32 bit unsigned integer, represented in binary form.

We think it makes the most sense to start with the simpler data structures that compose more complex ones. First up we have:

The Block

There are two ways to visualize the block:

- Graphical Representation
- World Generation

A block in world generation is represented as 32 bit unsigned integer, represented in binary form utilizing the following bitwise encoding:

Expansion	**Block State**	**Block ID**	**Block Namespace**
4 Bits	4 Bits	16 Bits	8 Bits
Spare bits in case we need them.	16 Possible States	65,535 Blocks Per Namespace	255 Namespaces

Namespacing is to allow for expandability and easy exposure of a modding API. We will use the standard namespace of 255 for base game blocks.

This bitwise encoding is both to optimize performance during runtime and to reduce the size of storing the block to disk. The world is procedurally generated and subject to change by the users or environment, so we need to store this information dynamically. Bitwise encoding is the most efficient means to do this.

Chunk

A chunk is a cubic matrix of a given size (presumably 16). Each cell contains a block (represented as an integer as mentioned above).

World

A world is composed of a number of chunks, spreading virtually infinitely in all directions (special considerations will have to be taken into account for indexes outside of the max and minimum **signed** integer range).

Chunk loading: There are two stages to chunkloading. Initializing a never before loaded chunk, and loading a previously loaded chunk.

Chunk initialising is the process wherein a chunk is generated for the first time. This is based on layered simplex (or perlin) noise.

Perlin and simplex noise are functions that produce noise (random-like values) but the values change smoothly and consistently.

Normal noise

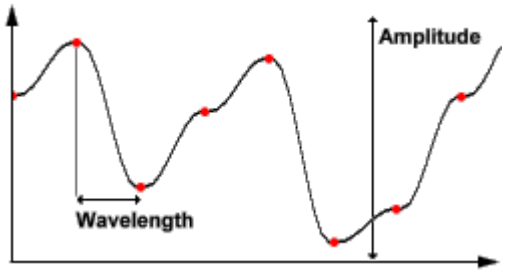
Perlin noise (Smooth continuous gradient values)

How is this useful

We can implement these noise functions ourselves, or we can use other packages and implementations. Either or works. (We have experience implementing a simple perlin noise algorithmProcedurally generation in games so far have been tied to their procedural nature. It's difficult to have a story build on itself outside of the player's control. This is what we are trying to solve, the 'need' of our system. We want to showcase how we can build a framework to use a large language model to solve this issue, and how it would be integrated into a block game. This will allow users to have a more in-depth experience and will enhance replayability for the system., simplex is designed to be more efficient by using polygons and works perfectly well in the 3rd dimension (which is our max scope)).

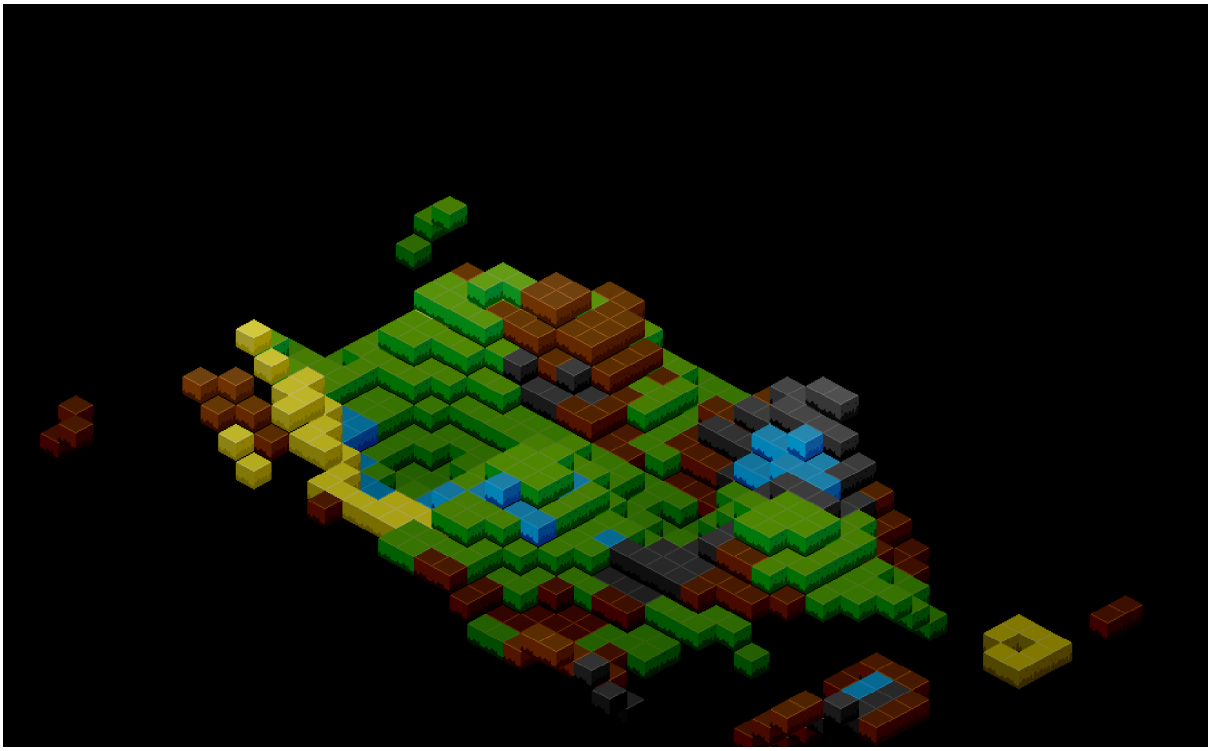
This diagram helps to describe frequency and amplitude of a simple perlin noise function in two dimensions. The y axis is a continuous range between zero and one. At a given x the output of the function is the y level (0-1). Random points are plotted along the x axis at a given wavelength. We then interpolate between these results, smoothing the random noise.

Wavelength: The distance between the red points on the graph.
Frequency: 1 / Wavelength
Amplitude: Highest point on the wave.
[2]



When we load a chunk, we are effectively iterating through its cells and populating it with a block. Say the most simple world has two types: a solid block and an air block. We want to populate everything up to a given y point with the solid block and everything above with air.

A perlin noise algorithm (as an example), returns a value in the range of 0 to 1. We pass the (x, y) coordinates of the block as input parameters to the function, and get a value back. Say we have a height limit of 10 blocks. We use the returned value to influence the height of a block at a certain point: floor(returned value * max_height). This gives us a y value for a given (x, y) that represents the height of the block (all blocks including and below this y value are to be solid blocks). This is the most simple way to utilise perlin/simplex noise to generate landscapes.



Example from a prototype for a similar game I worked on a while back.

We create more interesting landscapes by generating multiple layers of perlin noise at different amplitudes and frequencies and then using some operation on all of the layers to combine. The most common operation is to add all of the layers together. (Good example here: https://www.arendpeter.com/Perlin_Noise.html)

Chunk Unloading: Once players go out of a certain range of a chunk this chunk will need to be unloaded. Because changes may be made to a chunk (say we remove a block), we need to save this chunk to memory. We will use flatbuffers for this, as human readable formats like json come with huge overhead and incorporating a custom serializer is really unnecessary and leads to issues with backwards compatibility. The chunks are serialized and then saved to a postgres database. An optimization to consider is only writing changes, not the complete chunk.

Chunk Reloading: Once a player comes within range of a chunk it will need to be reloaded. The chunk data is fetched via its coordinates (as they are unique and allow for easy indexing). The serialised data is then deserialized back into their chunks.

5.2 High Level Design Diagram

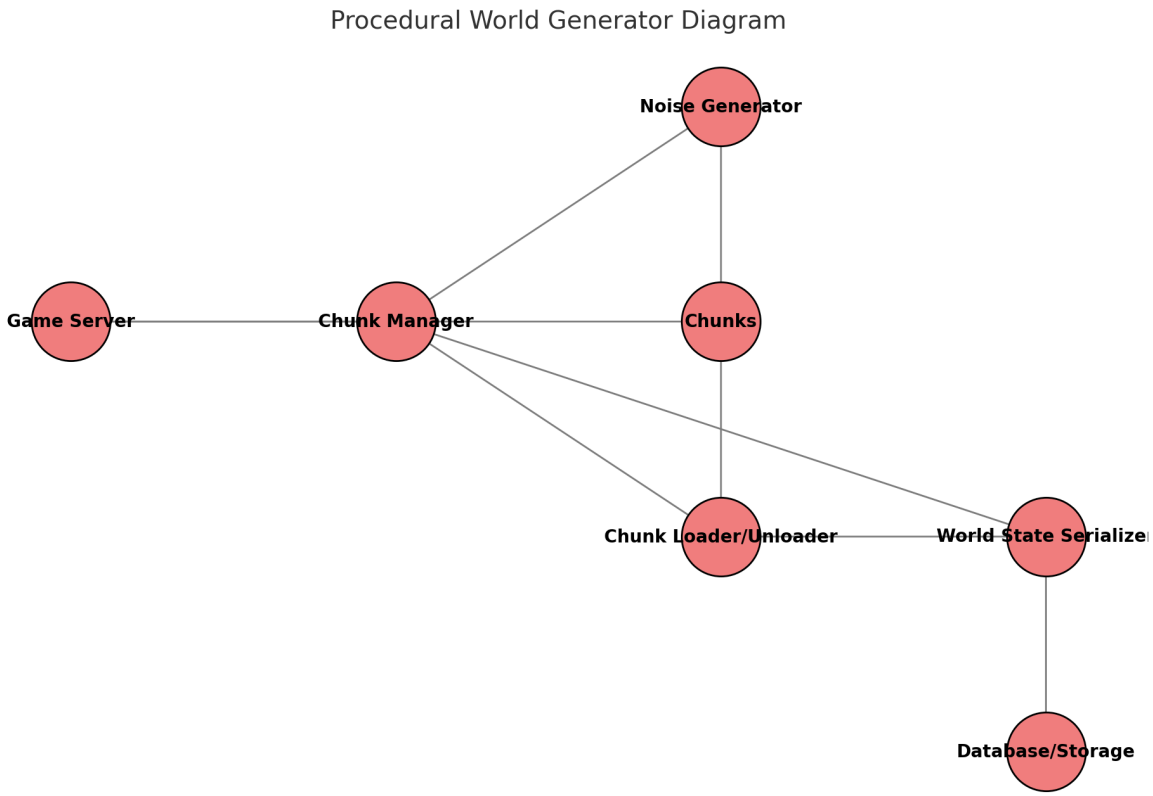


Fig 5.2
Fig 5.2 Shows a high level overview of the world generation system. It references an external component (the postgres db), but it's so intrinsic to the design of the system that it wouldn't make sense not to include it.

1. The Game Server initiates the process. (user's position is sent)
2. The Chunk Manager determines whether to load or generate a chunk
3. The chunk manager either calls the procedural world generator or the database, depending on the operation.
4. The chunk manager serialises the chunk.
5. The chunk is sent back to the game server, and eventually the client

For new regions, it triggers chunk initialization.
For previously visited areas, it requests chunk loading from the database/storage.
The Chunk Manager interacts with the Procedural World Generator.

When initializing a new chunk, it passes the required seed or parameters, such as user-defined prompts or biome specifications. The Procedural World Generator uses these inputs to produce a detailed chunk.
The Noise Generator generates terrain data for the chunk.

Algorithms like Perlin or Simplex noise create topography, defining features such as mountains, valleys, and water bodies.
The Chunk Manager populates the chunk with game entities.

Basic terrain data is enriched with NPCs, resources, and structures. NPCs and other objects are placed based on predefined rules or probabilistic models.
The Chunk Manager serializes the chunk.

It uses the World State Serializer to prepare the chunk for storage in the database or for real-time delivery to the game server. Serialization ensures efficiency and backward compatibility.
The Game Server receives the processed chunk.

It integrates the chunk into the active game world, ensuring that all players in proximity can interact with it. Synchronization is maintained across clients in multiplayer sessions.
The Database handles persistence.

For unloaded chunks or modified terrain, the World State Serializer saves them to the database. This ensures the world remains consistent across sessions.

How is an NPC generated and their relationships generated:

There are two ways in which to think of an NPC:

- An Entity in the Game World, represented as an **Entity** derived from components
    - An entity on the frontend derives similar components to the backend, but these are purely for visual purposes.
    - Components include:
        - Position
        - Abilities

- A fictional/generated Character with personal attributes and traits, represented as a JSON file (we could consider flatbuffers down the line, or any other format really, but for now keeping with JSON makes sense at the early stages of the project) . There are two main kinds of attributes defined in this JSON file.
    - Procedurally Generated Fields
    - LLM Derived Fields

Procedurally generated fields represent fields that are generated entirely programmatically, and include:

Numeric Attributes with a dynamic (adjustable) maximum range.
- HealthPoints
- StrengthPoints
- IntelligencePoints
- EnergyPoints
- HungerPoints

We will add to this list as the scope of the game expands.

String Attributes selected from a predefined **set** of options:
- Profession

    - E.g Carpenter, Smith, Doctor
- Personality
    - Set of traits, all NPC's have to be implemented by an NPC
- Skills
    - Functional skills that the NPC can utilise, i.e cooking food. (A player could also share this skill, that's what differentiates skills from interests). This could degrade without practice and increase with. There could be a random chance that an NPC decides to learn a new skill. That could be a highlight part of their day, and expressed in dialogue.
- Mood
    - Influenced by personality and memories (memories include today's memories as well as older ones)
- Gender
    - What the NPC claims that their gender is. **Note:** Gender doesn't inherently define a style or demeanour of an NPC, but it's gender in combination with its culture can.
- Interests
    - Abstract skills, i.e skills that a player character can't have. Poetry for example.
        - These could influence how an NPC might spend their free time. Eventually it might cause them to change professions for example.
- Desires
    - These can extensively be covered by virtual game concepts such as to increase wealth, craft and item, buy an item etc. These will be influenced by a number of attributes in order to prioritise desires (e.g hunger level dips below 20, it takes a very high priority)
- Memories
    - Split into segments
        - Today
        - Yesterday
        - This Week
        - This Month
        - This Year
        - Lifetime
    - Every day memories are evaluated, uninteresting ones are discarded, and ones that should keep their relevance move down a tier, with defining memories such as the death of a family member being retained for lifetime.
    - This is a complex one to evaluate and implement. Should memories be event based, and everything in the world that can happen triggers an event and the NPC is notified? Should an LLM handle the memory? Maybe a hybrid approach. We will need a stronger baseline implementation of the other attributes.
- Goals
    - Contains a current goal
    - Contains a list of goals that the NPC hopes to achieve, likely within a specified time frame. These are priorises and weighed.
- Social Status
    - These could be influenced by culture, but are picked from a predefined set of attributes.

- Race
    - Could this be an LLM derived component? That would be very interesting.
    - Races could have certain perceptions of other races? Will have to evaluate the ethics of this, as it may strike too close to home for some people.

LLM Derived Components: (We will not focus on these initially, rather the above.)

- Beliefs
    - These could include religion and moral frameworks. These might be impacted by profound memories.
    - Can have a wide range of influence on a given NPC.
- Culture
    - These could include religion and moral frameworks as well as history (more research needed on the definition of “culture”) .
    - Can have a wide range of influence on a given NPC.

This json is fed to the LLM, with the current goal of “talking to player”.
Two things that could happen:

- LLM simply returns dialogue (based on the context fed to it, which includes memories of past conversations, including the last thing that was said to it.)
- LLM actually returns an updated json file, updating the weights of certain attributes.
    - Might be prone to hallucinations.
- This will require lots of testing.

Personality Traits

Traits are on a scale, wherein contradictory traits are placed on a scale of 0-100:
All NPC’s will inherently have these traits, and a score of 0 is completely valid.

Bravery	0 == Cowardly
Generosity	0 == Selfish
Empathy	0 == Apathetic
Optimism	0 == Pessimistic
Confidence	0 == Insecure
Friendliness	0 == Reserved
Loyalty	0 == Disloyal
Humbleness	0 == Arrogant
Trusting	0 == Distrustful
Compassion	0 == Indifferent
Self-Sufficiency	0 == Dependant

Curiosity	0 == Apathetic

Note: Generative AI was used to assist in brainstorming of traits and finding a good balance of them with the aim of producing interesting results (i.e interesting and nuanced character personalities).

How are NPC relationships generated deterministically while remaining efficient?

One of the problems to address is that if we are allowing NPC's to have an abstractly large relationship pool, and these relationships should be able to link to currently ungenerated regions of the world, how do we supply information about ungenerated NPC's to NPC's that supposedly have knowledge of their existence?

We do this using *world regions*, and by defining some very specific criteria for NPC generation:

- NPC's must be completely responsible for their own generation.
- NPC's must provide a lazy_info (abstract name) method that allows us to get some basic info about them without completely generating them.
- The info provided must be of a procedural nature, and not reliant on anything an LLM might generate.
- NPC's must be identifiable by a unique ID.

World Regions

World regions exist as a medium to allow extensive and distant relationships between communities and individual NPC's to exist without fully generating these regions. It allows an NPC or community in a given region to be aware of a community or NPC in a neighbouring (ungenerated) region by establishing only the relationship between them. An NPC in a loaded region makes a request to the unloaded region for information regarding an ungenerated NPC, and is written to a 'queue' file representing that region, to be processed when that region is likely to be loaded.

Regions must have a limited number of NPC's/Communities. This number must be deterministic and a known property of the region before the region itself is ever loaded.

Say we populate a region with a pool of five communities (a community is defined as more than one NPC living in close proximity, dedicatedly) and the total population of the region is 200. Each individual NPC should be selectable from the pool and some basic deterministic information should be retrievable without loading the region.

Regions are loaded when a player character comes within proximity of or enters a single chunk in a previously unloaded region.

A region does not contain any data about the world, but divides the world into segments that contain community and NPC information.

Regions are yet undecided in size, but presumably very large.

Generating a Relationship Graph

Relationship generation, for our context, is best visualized (and likely implemented) as an abstractly large graph, where nodes are NPC's and edges represent relationships with other NPC's.

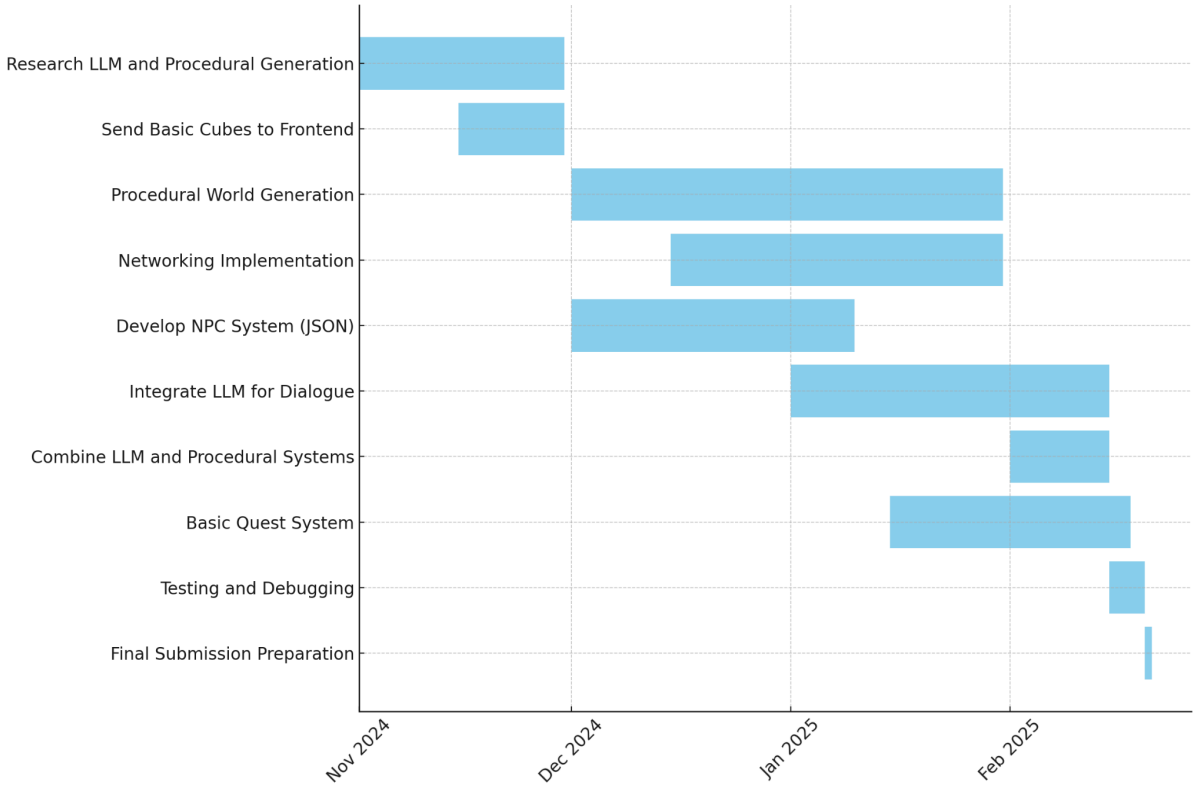
We can't load this entire graph at once, yet we need it to be of abstract size and deterministic. By this we have some constraints:

- An NPC can only ever reference another NPC in a different region within a given depth.
- For the same seed the same graph AND relationships must be generated.
- For relationships to be deterministic, so does NPC generation.
- In order to generate meaningful clusters on the graph e.g a family, separated distantly in the game world, we need to generate these clusters first, then overlay these clusters and add a chance for nodes to connect. This means that important relationships (like a mother or father perhaps) are guaranteed to be generated, but don't tie NPC's to only these relationships. Other relationships, such as friendships, rivalries or even just plain knowledge of another NPC's existence should be within the domain of general relationships.
- We need to load and save regions of the graph
- We need to be able to queue updates to unloaded regions of the graph (say an npc dies, might this reach their distant family, or will we present the player the option of informing that family?)

It's critical that we create a tool to help visualize the creation of these graphs for monitoring and debugging purposes. We've started work on one (web based) using sigma. We can expand this tool as needed, and quickly see how certain attributes affect the graphs and make some inferences through it.

6. Preliminary Schedule

6.1 GANTT Chart



6.2 Minimum Viable Product

A seamless multiplayer experience where users can explore a procedurally generated block world, with simple 3d graphics and a simple quest system (a fetch quest to start), allowing users to experience what 1-1 dialogue with a single NPC would look like.

Requirements:

LLM Integration

Physics Engine

Multiplayer Networking

Graphics

7. Appendices

7.1 Resources

https://rtouti.github.io/graphics/perlin-noise-algorithm
https://www.arendpeter.com/Perlin_Noise.html

7.2 References

[1] Hugging Face. *Llama-3.2-1B* [Online]. Available: https://huggingface.co/meta-llama/Llama-3.2-1B

[2] PostgreSQL. *PostgreSQL: The World's Most Advanced Open Source Relational Database* [Online]. Available: https://www.postgresql.org/

[3] Raouf's Blog. *Perlin Noise: A Procedural Generation Algorithm* [Online]. Available: https://rtouti.github.io/graphics/perlin-noise-algorithm