# Dungeon of the Mighty Titan - Vanquish

**Team 6: David Bond, Matthew Yengle, Timothy Canipe,** Vishal Bhatt

**Final Report**

# Table of Contents

# I. Project Definition

### Goal

To create a multiplayer video game that will allow 3 to 5 players to fight AI controlled monsters in a dungeon-like setting.

### Description

Over the years, there has been a surge in popularity of massively multiplayer online games. One prominent feature of these games is the dungeon raiding mechanics they employ. Players form groups to fight hordes of enemies and powerful bosses as they travel through dungeons in order to obtain rare items. However, players will usually have to play for many hours before they can experience these adventures, as they require high-leveled and well-equipped characters to survive. With our game, players will be able to form a group and raid dungeons right from the beginning. Furthermore, our game will capitalize on the popularity of several other genres, attracting many video game players.

# II. Project Requirements

The game will:

1. (Functional) Include multiplayer functionality:
    a. Using a client\Server Model.
    b. An in game chat system will not be provided.
    c. An area for people to find other people to play with will not be provided in the actual game.
    d. Internet connection (56Kbps or faster).
    e. One user designated as the server;  all others will be clients to this server

    f. The server (and only the server) will maintain a single copy of the GameMode actor.
    g. The server will provide each client with a GameState actor.
    h. Server will provide clients with current map when they connect.
    i. Successful client connections will receive a temporary PlayerController to replicate.
    j. Remote Procedure Call (RPC) will be used to update map and state changes.

2. (Functional) Allow the player to interact with aspects of the game, such as:
    a. Other player characters in the game.
    b. Enemy units and other non-player characters in the game world.
    c. The game world that the player character is in.
3. (Functional) Save the state of the game.
    a. Any one of the users playing the game should be able to save the state of the entire game.
4. (Functional) Allow a user to load the saved game.
5. (Functional) Have an inventory system with an item database.
    a. Each player character will have their own inventory system.
    b. Players will be allowed to trade items.
    c. Players can add items to their inventory from the game world.
    d. Players can drop items from their inventory in the game world.
    e. Players can equip equipable items.
6. (Functional) A combat system that gives the player options depending on their class.
    a. (Performance) As combat is a major component of dungeon-crawlers, the system should be refined and appealing.
7. (Functional) A turn-based movement and combat system.
    a. (Performance) While retaining fast-paced and exciting gameplay.
8. (Functional) Feature an attribute system.

    a. (Performance) The attribute system will give the user several options on how they want to play the game.

9. (Functional) Give playable character classes unique skills that distinguish them from other character classes.
    a. (Performance) Have enough skills that players will have a choice in how they build their character class.
10. (User Interface) Be represented using a visual method of design (GUI) that the player will be able to view and interact with.
    a. Include a Heads-Up Display (HUD) that the user will be able to see their player character’s stats and statuses.
    b. (Performance) Responsive and well-designed layout of the visual design.
11. (System) Be able to run on desktop computers running Windows 7 or newer.
    a. Requires no special hardware; uses standard keyboard and mouse setup.
12. (System) Use Unreal Engine’s C++ API and scripting language Blueprint.
13. (System) Development environment will be Unreal Editor and Microsoft Visual C++.
14. (Functional/Security) Use the Steam SDK package to allow Steam users to join game sessions.

## III. Project Specification

**Focus**

- Gameplay-driven over storyline.
- Balanced for multiplayer rather than single player.
- Turn-based combat system over real-time combat.

**Game Engine**

- The game will be made using the Unreal Engine 4 by Epic Games.
- The game will use the Advanced Turn Based Tile Toolkit by Knut Overbye made for use with the Unreal Engine.

**Platform**

- Desktop, Windows OS (7 and newer).

**Genre**

- RPG, Multiplayer, Turn-Based Strategy (TBS), Roguelike, Dungeon Crawler, Isometric.

## IV. Risks Assessment and Design Decisions

Risks:

1. No member on the team has created a full game before. Not only will it be challenging learning how to make a game, but we will also not be able to reasonably predict what we will be able to accomplish in the given time.
    a. Learning more about video game development from the beginning will help solve problems that may arise due to our lack of experience.

2. The Unreal Engine uses a combination of C++ and Blueprint, the latter being its own scripting language. Not all members are experienced with C++, and no members is experienced with Blueprint.
    a. One member is well-experienced with C++, and there are plenty of available resources for C++.
    b. Epic Games (creators of Unreal Engine) provide many tutorials and guides for Blueprint. C++ can be used for the majority of development in the unreal editor, allowing Blueprint to be learned over time.
3. Video games change quite frequently during development. Using the waterfall method of software engineering may be much less effective in game design.
    a. By leaving out a concrete story, we are given many options with the game's development. Furthermore, using a very popular setting (medieval/fantasy) gives us a great deal of available resources.
4. Video game development have many different roles (artists, storyboard writers, animators, etc) that are not fully present in our team full of programmers.

a. Independent videogame development is a well-supported community on the Internet, with plenty of information, resources, and free assets.

Design Decisions (showing alternative models that were considered):

i) Use a well-established game engine, or build backend from scratch.

Existing Game Engine (Unreal Engine 4)

+Overall less time spent coding the backend.

+Less risks using an established game engine that is known to work.

+Easier to synchronize different components of the project.

+If we wanted to release this software as open-source, using a well-known engine would allow others to easily modify the game.

-Learning how to use the game engine will take time.

Building Our Own

+More control over how the game will function.

-Problems may arise when attempting to synchronize the game engine with other components.

-Planning other parts of the game will need to wait until the engine is fully mapped out.

ii) Choosing to focus on the gameplay rather than the story.

Gameplay-Driven

+Narrative exposition can be implicitly told through the gameplay (e.g., environment, characters, enemies, etc.).

+In dungeon-crawlers and rogue-like, the mechanics of the game are more important than the story.

-Gameplay can become repetitive and stale; without a story, there will be nothing new after a few hours of gameplay.

Story-Driven

+People may be more interested in the game if it had a compelling story.

+Many resources are available for the medieval fantasy genre.

-Less time will be spent on the mechanics.

-A story can limit our options when designing the game.

iii) Turn-based system instead of free-roam system.

Turn-Based

+Easier to synchronize multiplayer.

+Gives players more time to think about their decisions, which is why turn-based mechanics are often used in strategic games.

+Places more emphasis on planning movements and actions with the entire group rather than individual skill.

+It is easier for new players to get into turn-based games, as they have time to think and react.

+Movement in the game can be synthesized with motion in the gameworld (e.g. flowing river, swarms of non-enemies bats, glowing lights).

-Players may get bored or lose excitement if a teammate is taking too long with their turn. It is for this reasons that the game is limited to a maximum of 5 players per group.

Free Roam

+The combat and movement system would be more exciting, if done properly.

-Planning, resource, and programming intensive for the movement system and other

-Without restricted movements, players can encounter bugs more frequently. With our limited time, we cannot reasonably perform large scale beta testing that is needed to find all possible bugs.

-Balancing will be harder to accomplish if the combat is not limited in some way, which may lead to less skills or attacks being in the game.

# V. System - Analysis Perspective

## Data Analysis

| **Field Name** | **Data Type** | **Description** |
|---|---|---|
| DungeonGameMode | AGameMode | Defines the rules for the game. [1] |
| DungeonGameState | AGameStateBase | Manages information to be known about for all connected players. Replicated extension of GameMode.[1] |
| DungeonPlayerControllers | PlayerController[] | An array to hold the controller of all human controlled players connected. The controller persists the entire time the player is connected. [2] |
| DungeonAIControllers | AIController[] | An array to hold the controller of all AI, non-human controlled players. [3] |
| DungeonPawns | DefaultPawn[] | An array to hold the pawns for all players (human and AI) in the game world. A one-to-one mapping is created between a Pawn and a PlayerController (or AIController). [4] |
| DungeonPlayerStates | APlayerState | An array of all the users connected to the server. The object will persist for the entire time the user is connected. [5] |
| DungeonGameSession | AGameSession | Created and owned by the DungeonGameMode class. Only exists on the server. Acts as an interface for player connections (finding and joining the server). [6] |

## Algorithm Analysis

### Turn Based Algorithms

PlayerQueue

The order of the queue will be decided at the initialization of the GameInstance. The size of the queue will be constant, and the exact length will be dependent on the number of human players (3-5). Therefore, the queue can be implemented as an array of PlayerControllers, with a fixed size. The order of the queue is decided based on each character's Speed stat, where the character with the highest Speed stat is first in the queue. If there is a tie, decide arbitrarily.

Time Complexity: To find the PlayerController for a character (denoted here as N), it will be at N position in the array. Thus, the time complexity of the queue is constant, or O(1).

Space Complexity: Each player character will need to be stored in the queue. For N players in the game, the queue will hold instances of those N players. Thus, the space complexity of the queue is the number of player characters (denoted here as N) in the server, or O(N).

AIQueue

The order of the queue will be decided at the loading of each level, when the AI controlled monsters for that level are created. Although the amount of monsters will vary depending on the level, the number of monsters on a single level will not change. Therefore, the queue can be implemented as an array, which stores the AIController for each AI-driven character. Since the stats of each type of monster will not be randomized, the queue order can be predetermined based on monster types in the queue. The order between same monster types can be chosen arbitrarily.

Time Complexity: To find the AIController for an AI monster (denoted here as N), it will be at N position in the array. Thus, the time complexity of the queue is constant, or O(1).

Space Complexity: Each AI character on a given level will need to be stored in the queue. For N AI characters in the level, the queue will hold instances of those N players. When the level is changed, the array can be destroyed, and a new array can be made. Thus, the space complexity of the queue is the number of AI characters (denoted here as N) on the level, or O(N).

**Item Database**

The item database will be a structure used in the unreal engine for databases.  It is indexed based and will be accessed by the game state to populate the world with items and the enemy drops.

Time Complexity: To find an item with index *n* it will be located at position *n*. Therefore, the time complexity is constant, or O(1).

Space Complexity: Each item in the game will be stored in the database.  So for *n* items to be stored, it will take up *n* spaces.  Therefore, the space complexity is O(n).

**AI Pathfinding**

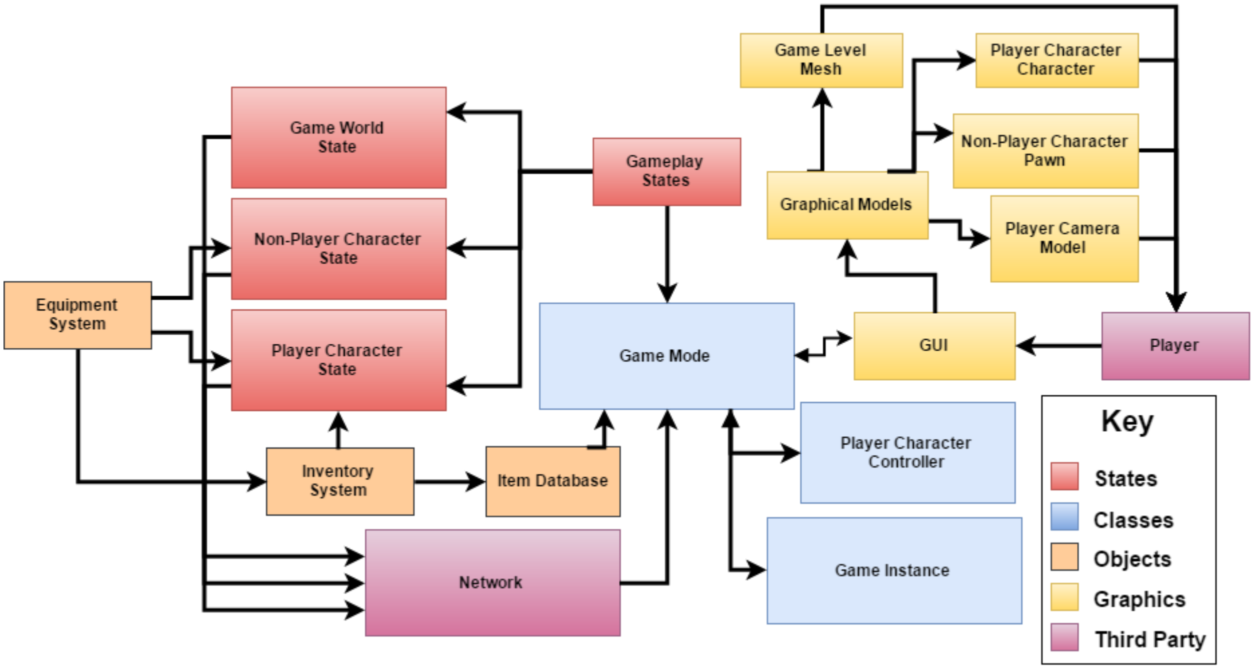
A* is the algorithm that enables AI to find the shortest path to the player. A* is a modified version of the Dijkstra's algorithm. Dijkstra's starts off by calculating the cost of each direction, this algorithm always favors the lower cost. The input is graph that has costs it will calculate the cost of each path and will always take the lowest one. A* uses a weighted graph to calculate the lowest cost, much like Dijkstra's, the major difference is that A* calculates the lowest cost of a certain point in the graph - not the entire graph.

The time complexity of this algorithm is O($b^d$) the variable b is the branch of the graph while variable d is the number of nodes creating the shortest path. This algorithm is exponential due to graph having nodes.

# VI. System - Design Perspective

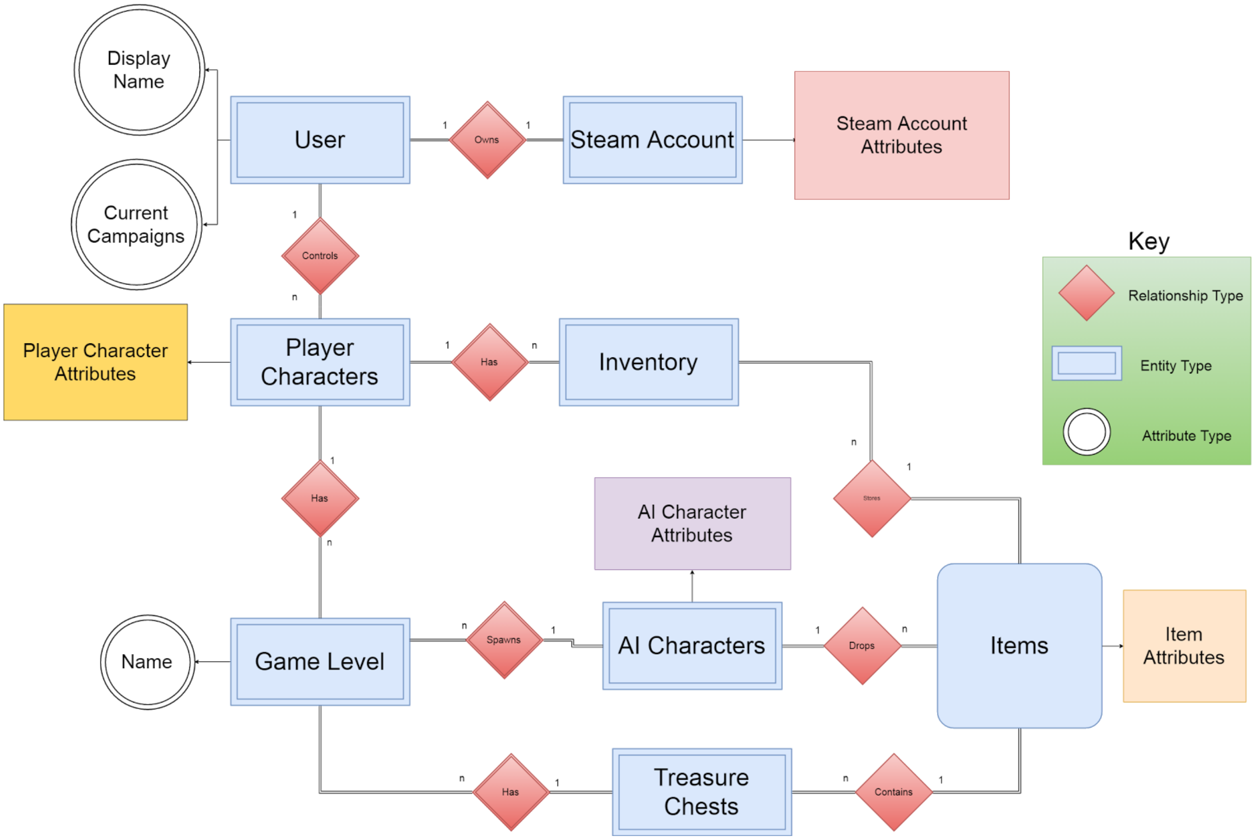
**System Model**

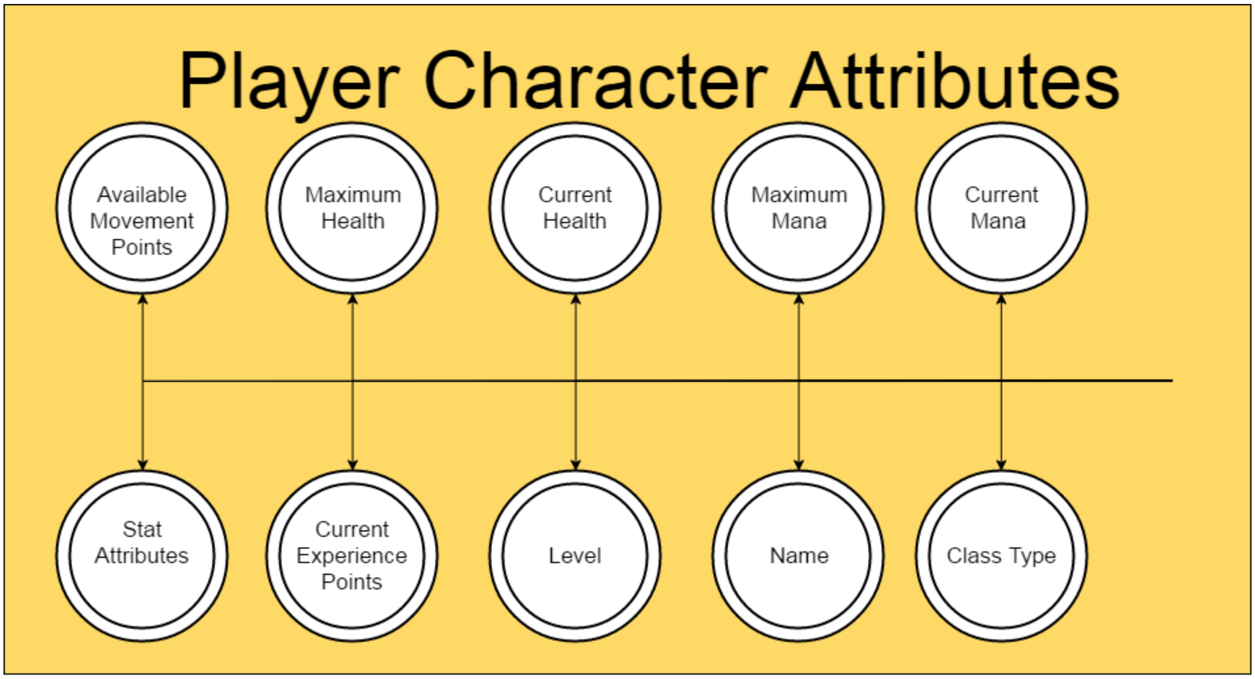
The system model below shows the relationship and general flow of data between the modules and subsystems. These subsystems and modules can be broadly grouped into four categories based on common functionality: states, classes, objects, and graphics. The classes can be thought of as the rules of the game, and control the flow of the data and gameplay. In this, the Game Mode defines the rules of the general game, such as how many players can play at once, where players and enemies spawn in the game level, etc. The Player Controller class defines the actions that each player can perform when it is their turn. The states store volatile data that is important to that game's gameplay. They need to be saved if the game is closed, and they can be loaded to recreate the previous game session. The objects have separate functionality and can be plugged into the states to offer the players those features. Finally, the graphics are made of the components that the player will visually see and interact at with on the screen and in the game world.

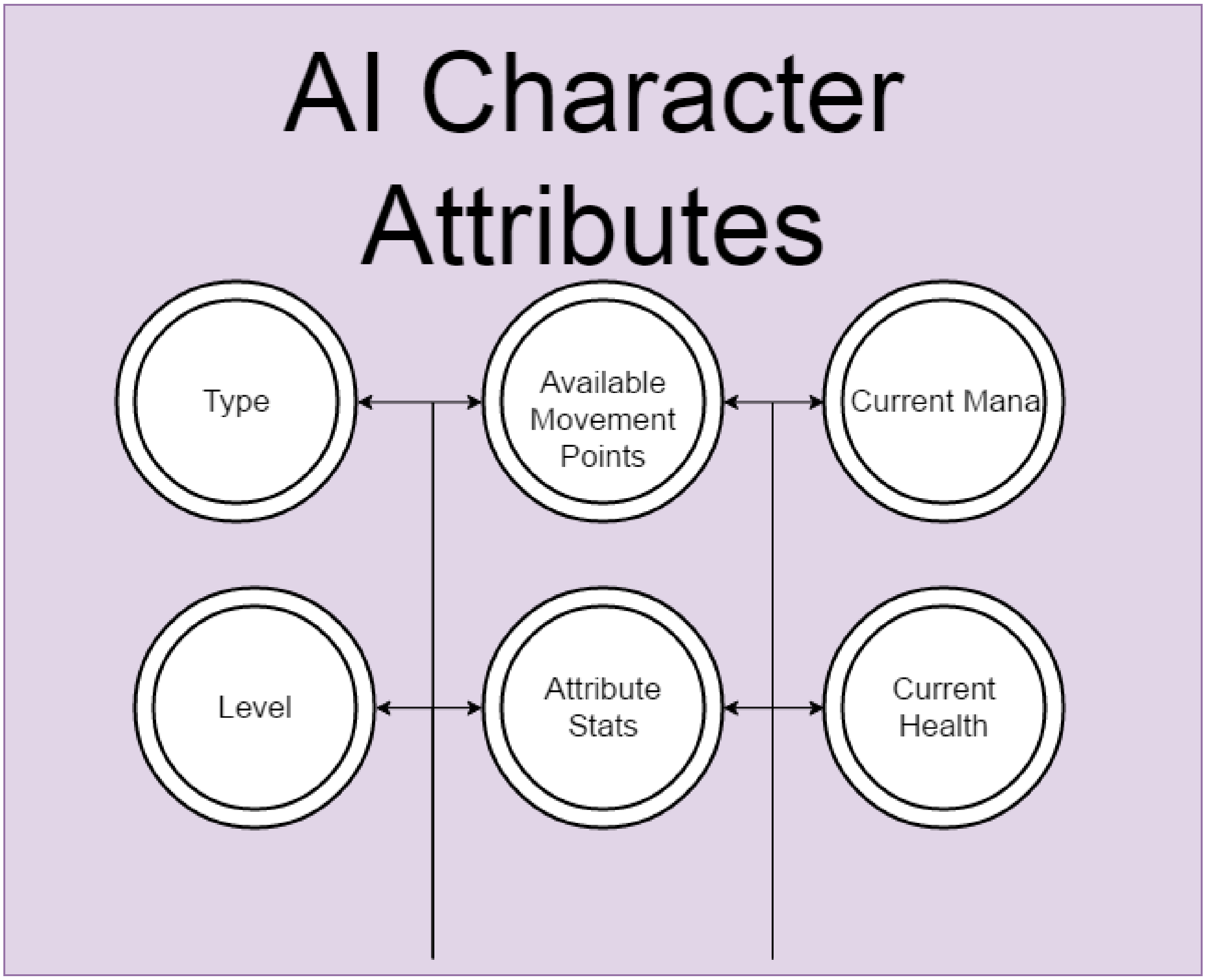


## Entity-Relationship

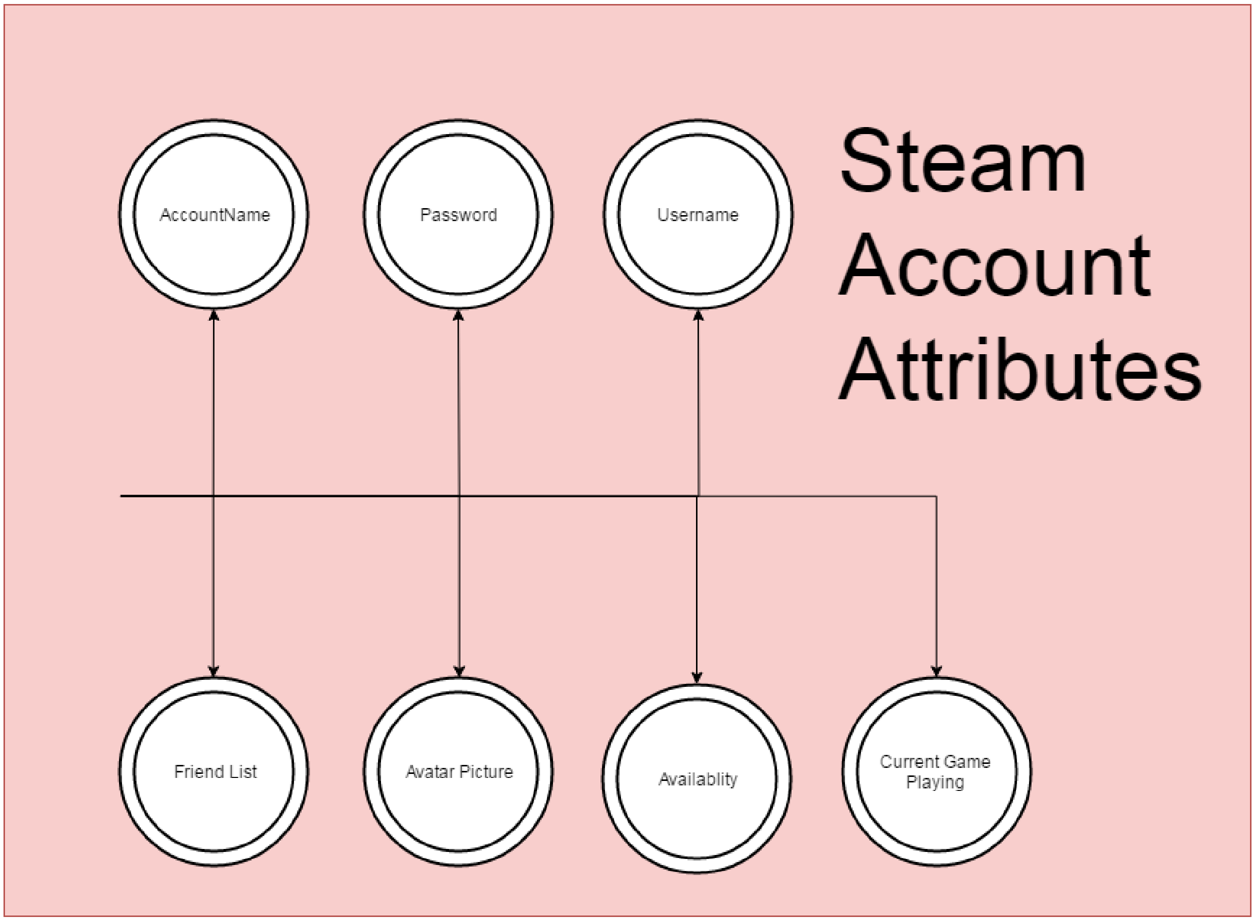
The entity-relationship model shows the relationship between different modules necessary for our game. In turn, each of these modules will have their own attribute types.

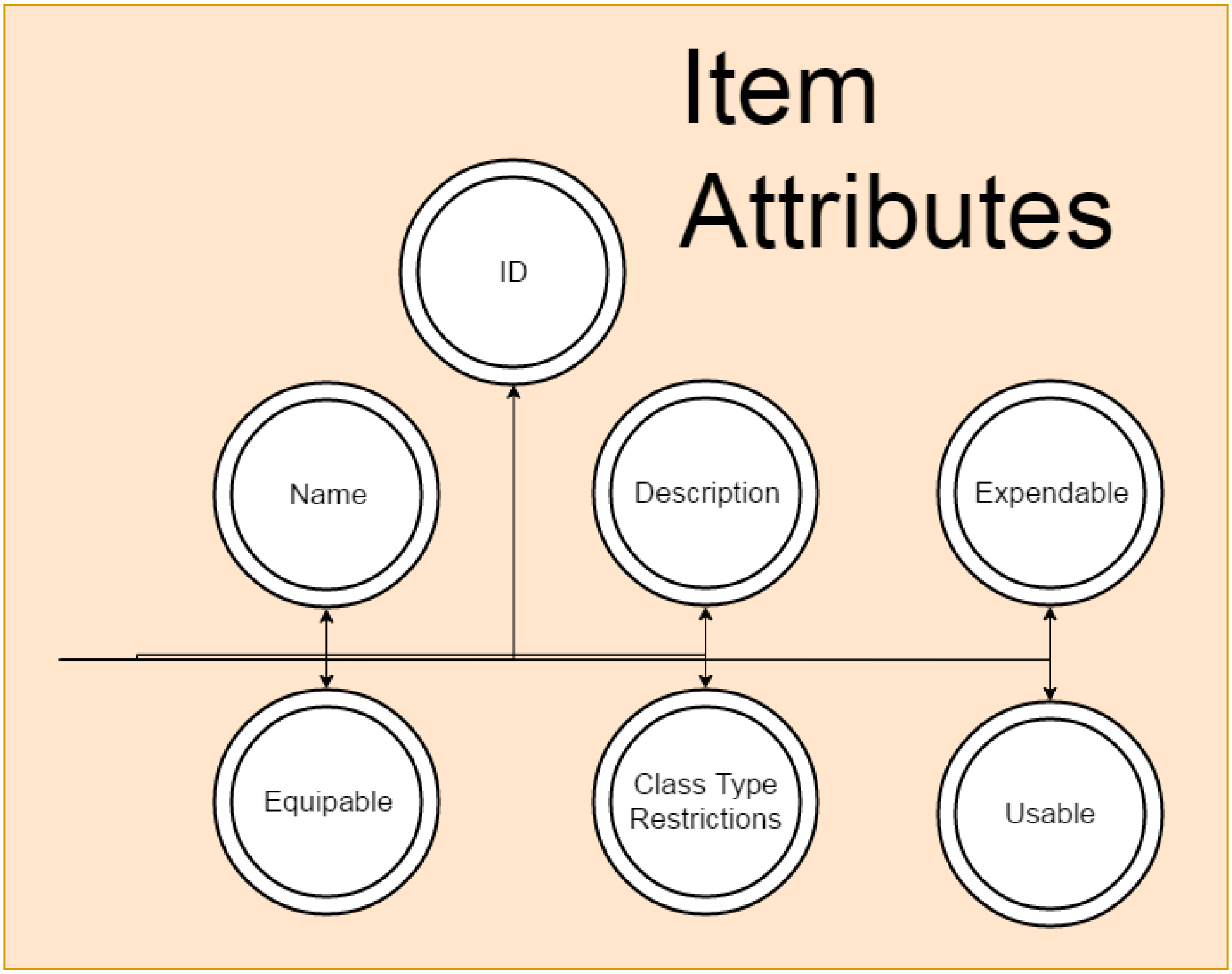

Display Name
Current Campaigns
User
1
Owns
1
Steam Account
Steam Account Attributes
1
Controls
n
Player Character Attributes
Player Characters
1
Has
n
Inventory
Key
Relationship Type
Entity Type
Attribute Type
1
Has
n
n
Stores
1
AI Character Attributes
Name
Game Level
n
Spawns
1
AI Characters
1
Drops
n
Items
Item Attributes
n
Has
1
Treasure Chests
n
Contains
1


Player Character Attributes
Available Movement Points
Maximum Health
Current Health
Maximum Mana
Current Mana
Stat Attributes
Current Experience Points
Level
Name
Class Type



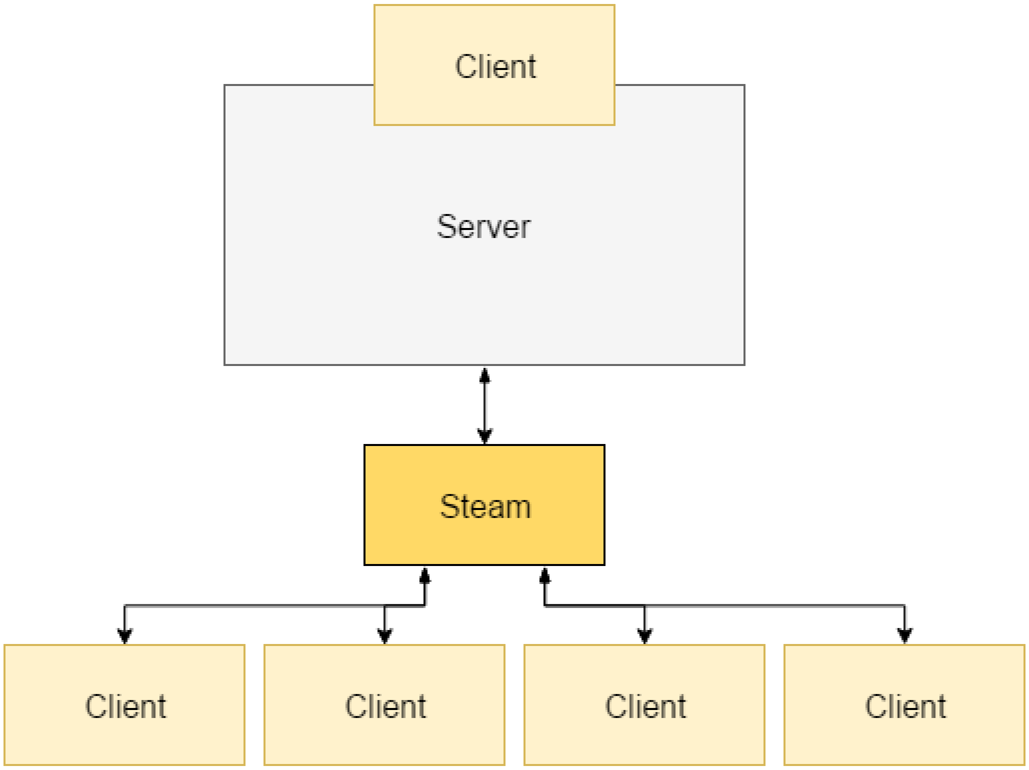
# AI Character Attributes


Steam Account Attributes
AccountName
Password
Username
Friend List
Avatar Picture
Availablity
Current Game Playing



**Network Model**

Up to four players will be able to connect to a session over the Internet using Steam. The player that hosts the server will reside on the server itself, and does not have to go through Steam. The players will be able to find sessions that are still in the Lobby by their name. Sessions that are in progress (playing inside of the game world) cannot be joined. Each client has their own PlayerController, and the server holds an additional copy of each player's PlayerController. The server will have the only instance of the GameMode module.



## VII. In Game Graphical User Interfaces (GUI) - Matthew Yengle

### Refined Model

The GUI subsystem is shown in the system model in section V. The GUI aspects are loading into the Player Controller, where the player can add or remove them from the screen at their discretion.

### Refined Design

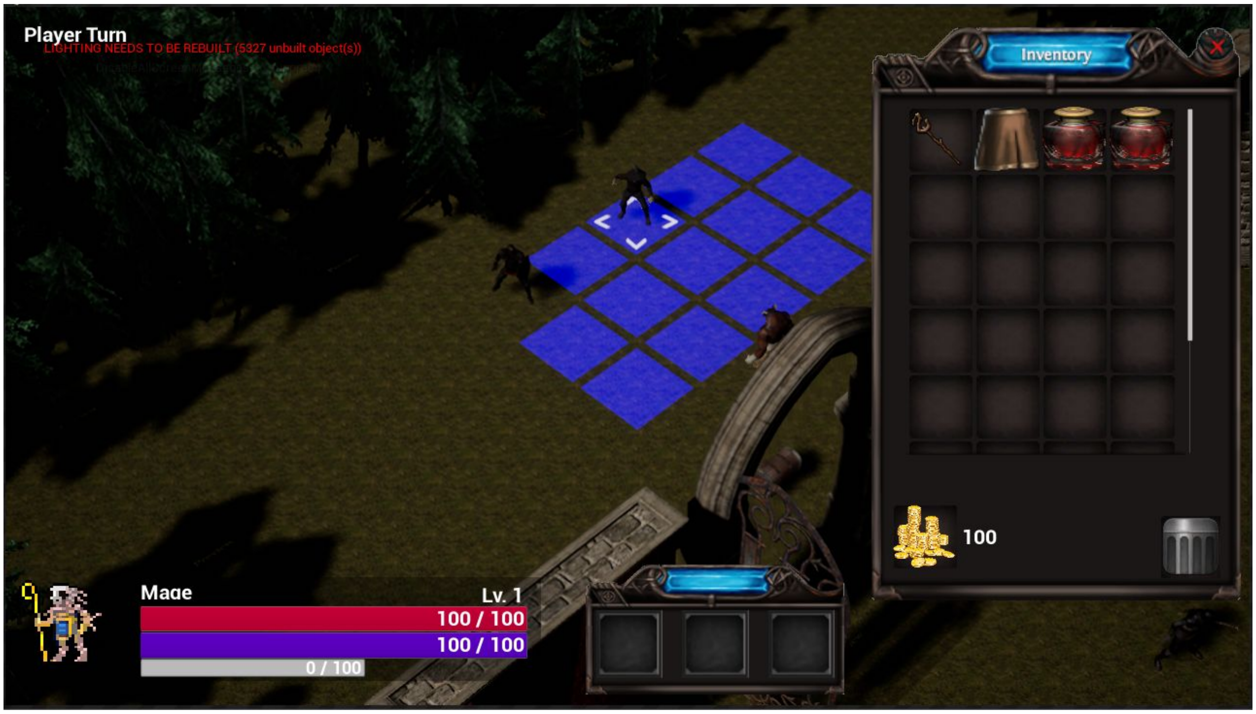
The inventory menu allows the player to access the items that their player character has stored. Most in game GUIs support drag and drop operations. The player can swap items in the inventory and discard items using the trash can. The character menu contains two sub-menus: the equipment menu and the attribute menu. In the equipment menu, the player can drag weapons and armor from their inventory into the appropriate equipment slot, and vice-versa. The attribute menu allows the player to increase the attributes of their player character.

### Coding and Testing

All code implemented for the GUIs was our own work, and not part of ATBTT or the Unreal Engine. However, the textures for the menus are from open-source artwork [7]. Testing was done through feature testing, where functionality was tested when the systems were added.

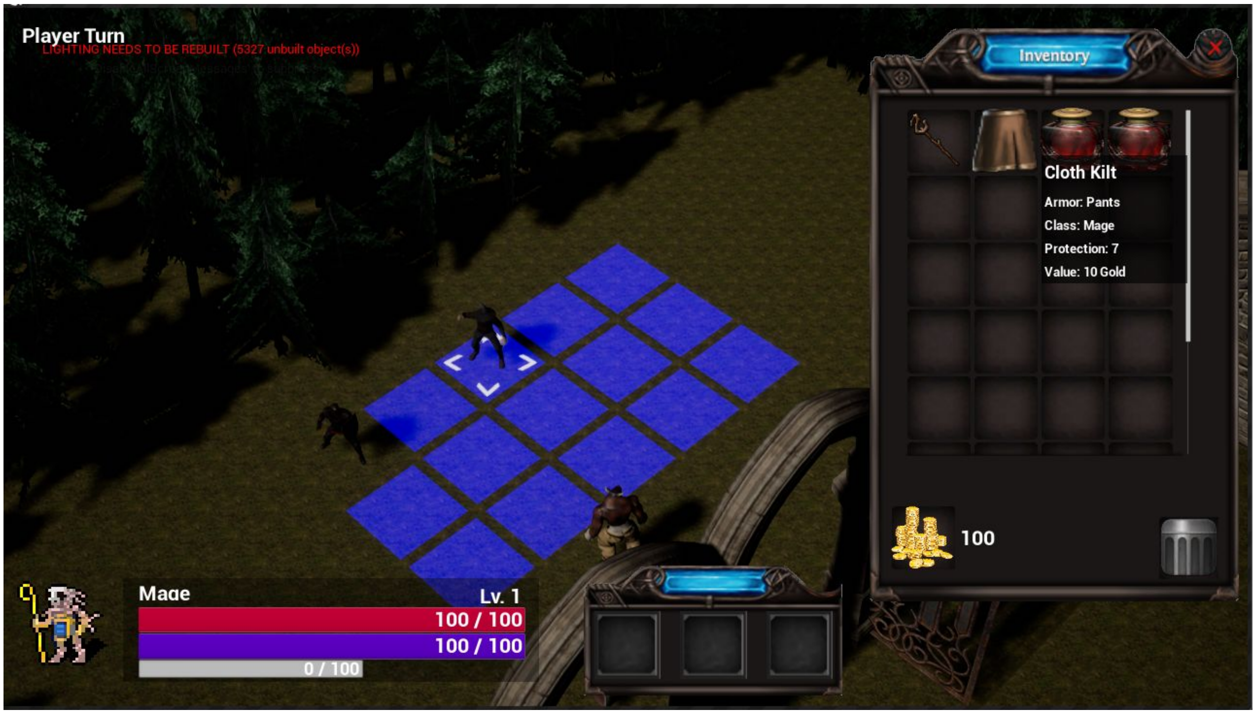
### Conversion and Training Plan

Items are stored in the player's inventory, which can be opened with the key **I**. Nearly all items in the game can be interacted with using **drag and drop** operations. The exception to this are potions, which are consumed by **double clicking** the potion in the inventory. An image of the inventory is shown below, with a few items.



LIGHTING NEEDS TO BE REBUILT (5327 unbuilt object(s))

**Hover the mouse** over most items in the game to display a description of that item. An image of the Steel Dagger's description is shown below.



Player Turn

LIGHTING NEEDS TO BE REBUILT (5327 unbuilt object(s))

Cloth Kilt

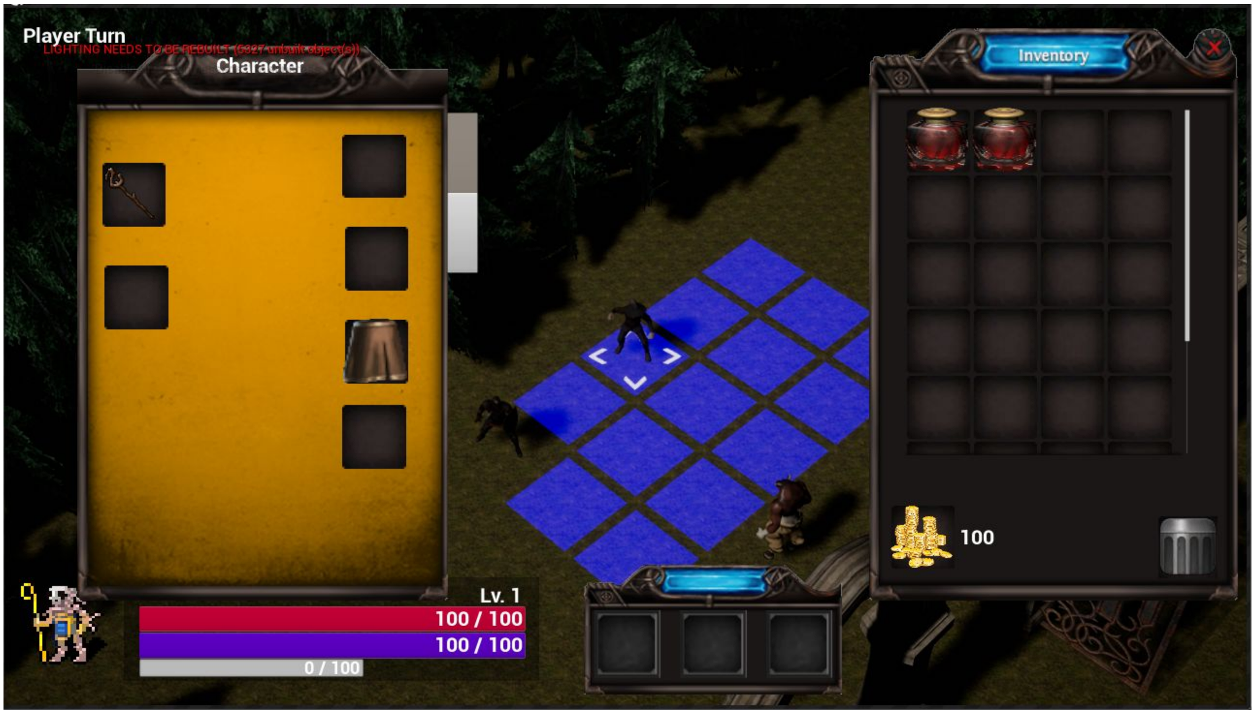
Armor: Pants

Class: Mage

Protection: 7

Value: 10 Gold

The player can open the Character menu with the key **C**. The Character menu has two pages of information: the Equipment page and the Attributes page. The Equipment page is shown by default, and is pictured below.

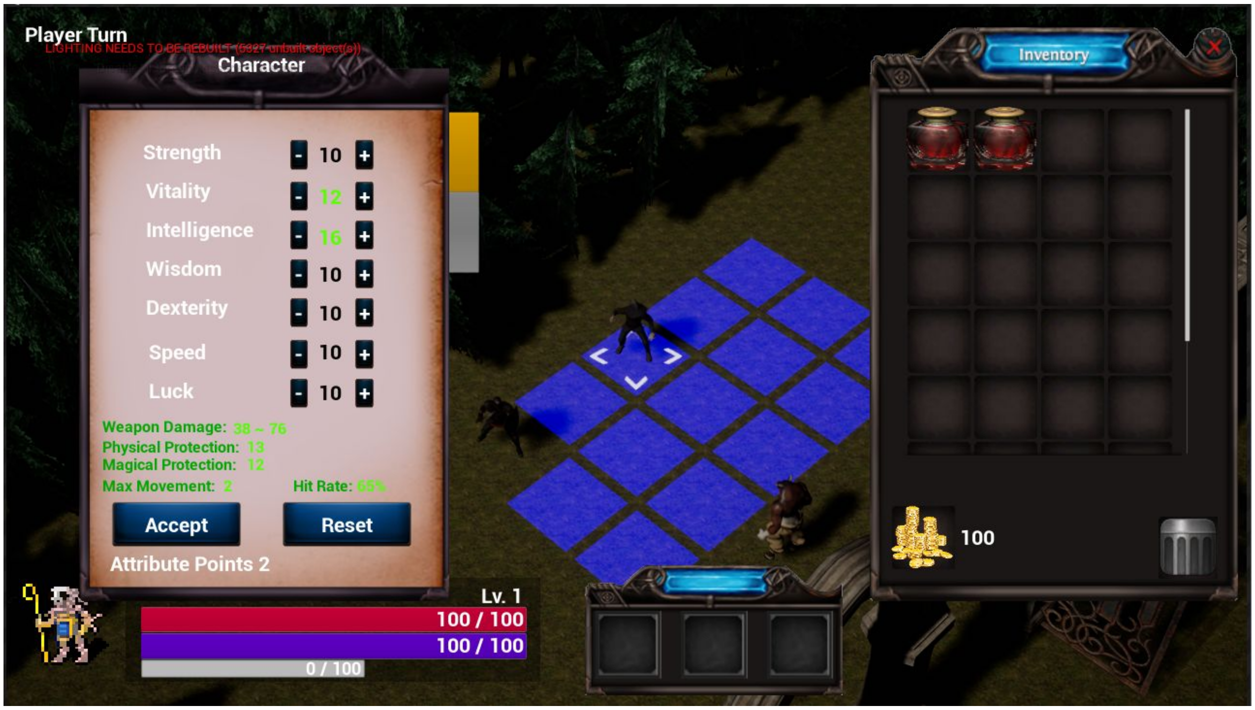


Player Turn

LIGHTING NEEDS TO BE REBUILT (5327 unbuilt object(s))

All players will begin a new adventurer with at least one piece of armor and a low-tier weapon. Items can be added or removed to the proper slots by a **drag and drop** operation. On the right, the slots are for: head armor, chest armor, pants armor, and feet armor. On the left are two weapon slots, for such things as daggers, swords, bows, and even shields. Two handed weapons such as bows, staves, and great swords must be equipped in the top left weapon slot. Items can only be equipped from the inventory item.

On the right side of the Character menu, there are two pages that switch between the Equipment page and the Attribute page, the latter is pictured below.



Strength - 10 +
Vitality - 12 +
Intelligence - 16 +
Wisdom - 10 +
Dexterity - 10 +
Speed - 10 +
Luck - 10 +

Weapon Damage: 38 ~ 76
Physical Protection: 13
Magical Protection: 12
Max Movement: 2
Hit Rate: 65%

Attribute Points 2

100 / 100
100 / 100
0 / 100
100

There are 7 attributes in total, which the player can allocate Attribute Points to upgrade. The player earns Attribute Points by leveling up. Increasing one of these attributes using the **+** button or decreasing one of these attributes using the **-** button is temporary. The player cancel the changes using the **Reset** button. To finalize the changes, the player must use the **Accept** button. Exiting out of the menu will cancel the temporary changes. The values shown below the attributes are reflected by any temporary changes.

## Final Subsystem

The subsystems implemented for the GUIs are robust, and was built for expansion in mind. Additions onto the Inventory, Equipment, or Character models should be easy to implement for future iterations.

## VIII. Item Database - Matthew Yengle

### Refined Model

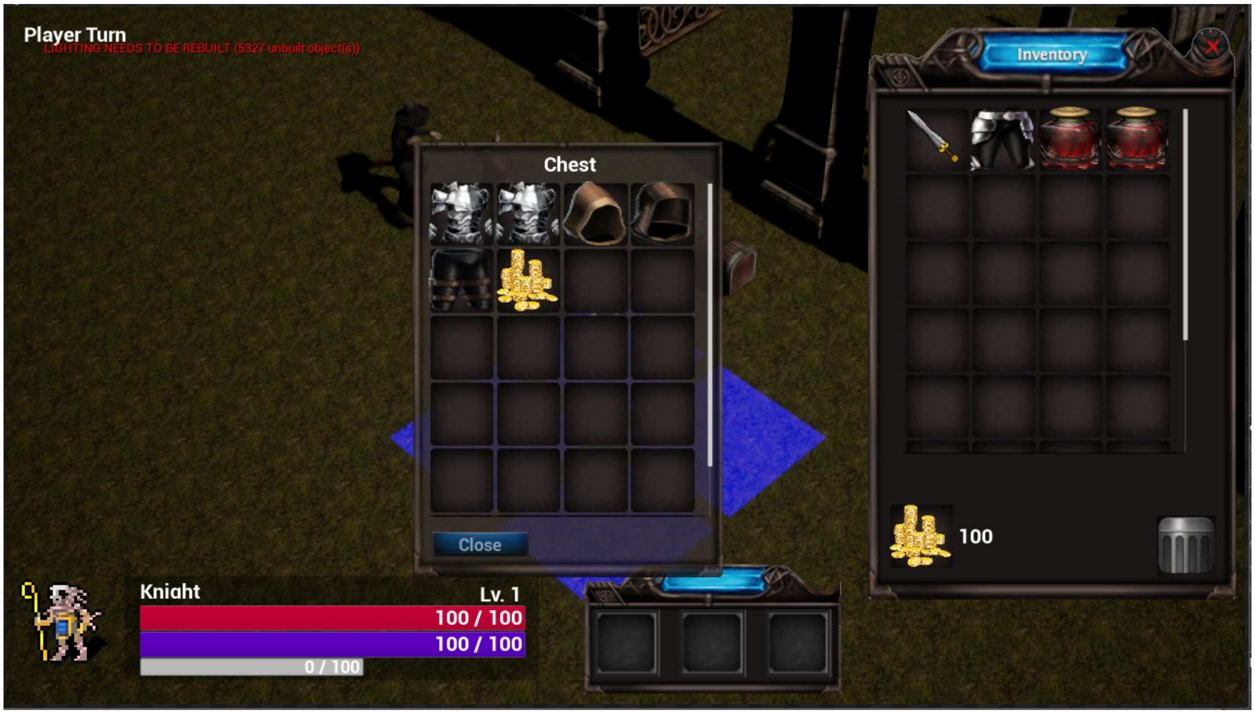
The Item Database subsystem is shown in the system model in section V. The item database is used in the Game Mode class to generate valid items to be used in the game.

### Refined Design

Each playable character has 5 or 6 weapons and 1 set of armor. Items are categorized into tiers based on a quality attribute, ranging from 0 (poor) to 5 (superb). There are three ways to obtain items: treasure chests, dead enemies (graves), and vendors. Equipment system allows for one-handed, dual one-handed, and two-handed weapons to be used. Ranged and melee class-restricted weapons give the characters different combat strategies.

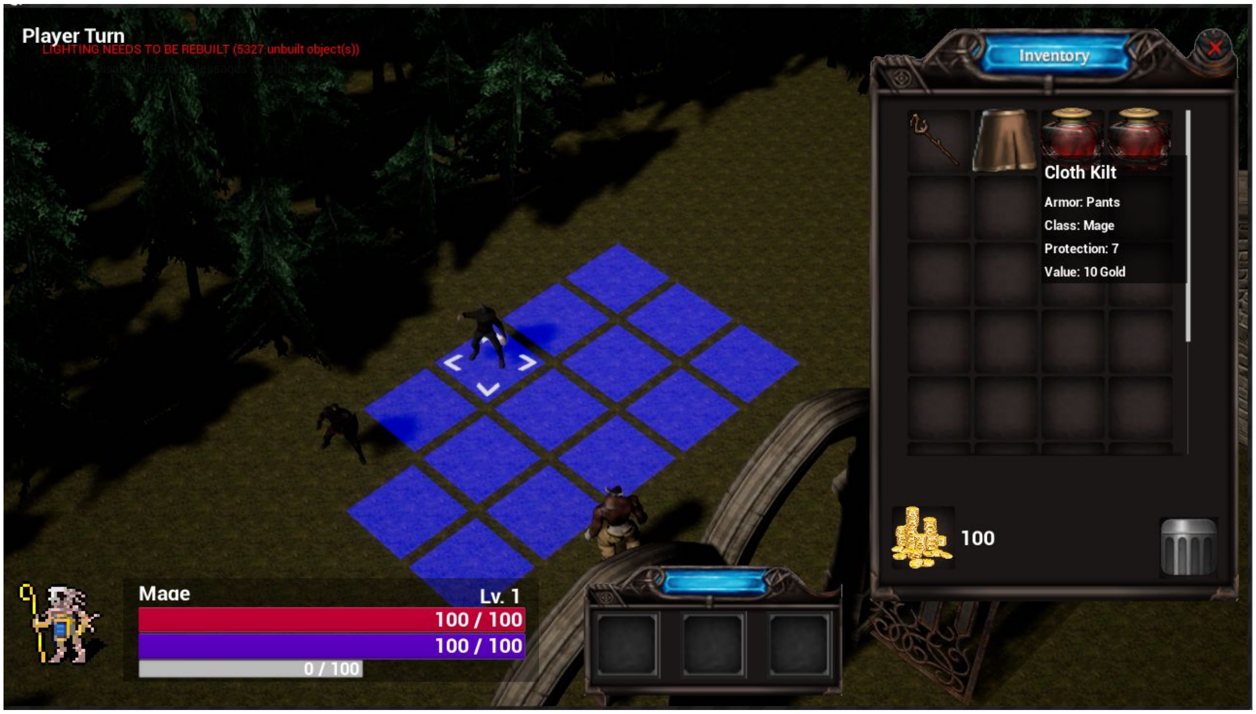
### Conversion and Training Plan

To open a chest, the player must be on an adjacent tile, and it must be their turn. Meeting these conditions, a player can open a chest using the **Right Mouse Button**. A player or enemy cannot walk on a tile that has a chest on it. An image of loot found in a chest is shown below.



LIGHTING NEEDS TO BE REBUILT (5327 unbuilt object(s))

**Hover the mouse** over most items in the game to display a description of that item. An image of the Cloth Kilt’s description is shown below.



LIGHTING NEEDS TO BE REBUILT (5327 unbuilt object(s))

Cloth Kilt

Armor: Pants
Class: Mage
Protection: 7
Value: 10 Gold

### Final Subsystem

As the item database is just a database, it can easily be added onto using other programs besides the Unreal Editor. The only real limitation to adding new items is getting textures to represent those items in a visual manner.

## IX. Player Controller - Matthew Yengle

### Refined Model

The Player Controller subsystem is shown in the system model in section V. The Player Controller class has the only reference to the owning Player Character Character and the corresponding Player Character State, which holds the Inventory system and Equipment system.

### Refined Design

The Player Controller subsystem is used by the player to perform actions in the game world through their character. In a multiplayer session, each player character would have two of their own Player Controller. One Player Controller resides on the server and the other on the remote client. The Player Controller class interacts with the Grid Manager given by the ATBTT to calculate a path that the player can choose. The Player Controller class also finds all enemy characters that can be attacked from the player's current position.

### Coding and Testing

The code that provides the player the ability to move and attack with their player character was provided by the ATBTT. Additional code was implemented to allow for interactions with items. Furthermore, all of the in game GUIs are controlled through the Player Controller class, where the player can open and close these GUIs using certain key bindings. The reason for this is that in a multiplayer game, each person will need their own individual GUIs.

### Conversion and Training Plan

Players are only allowed to move when it is their turn. This is indicated in the upper left hand corner of the screen. Once it is the player's turn, they can move by clicking with the **Left Mouse Button** on a tile within their allowed movement spaces, which is indicated by blue squares. The player can simply click on a blue tile, and their pawn will move to that tile. If the player does not want to move at all, they can click on the tile that the pawn is current on. The player can perform their movement process over multiple operations. The player is allowed one attack per turn, unless otherwise specified by a skill. To attack an enemy pawn, the enemy must be in range of the player’s pawn. A pawn’s range is determined by the type of weapon they are using. Melee weapons generally have a range of 1, where the enemy must be on an adjacent tile to hit. Ranged weapons have a range greater than 1. If the enemy is within range, the tile that the enemy is on be red. The player can then click on that tile using the **Left Mouse Button** to attack that enemy. If the enemy is out of range, but the tile is red, then that means the pawn will be close enough to be able to attack that enemy.

### Final Subsystem

The Player Controller subsystem will need to be continuously updated to support new features that the game may provide in the future.

## X. Game Mode - Matthew Yengle

### Refined Model

The Game Mode subsystem is shown in the system model in section V. The Game Mode is used as the central hub for most of the interactions between other subsystems.

### Refined Design

Formally, the Game Mode will define all rules for the game. It will setup the start of the game, define the winning conditions and the losing conditions, control the turn-based gameplay, and many other aspects of the actual game. To control these aspects, the Game Mode class will have instances of the 3 to 5 Player Character classes along with the number of Non-Player Character classes current on the level. These are stored in a queue to cycle through all characters per turn. For each character, the Game Mode class will tell the corresponding Player Controller that it is their turn.  The Game Mode exists only on the Server, and it needs to use other classes to interact with the clients.

### Coding and Testing

The code for the basic turn-based system in the Game Mode was provided by the Advanced Turn Based Tile Toolkit. However, code was added to the Game Mode to support additional features, where most interacted with the item system. Every addition to the Game Mode has been clearly marked with the author's name. Any comment that does not show the author's name should be assumed to be from the ATBTT. For these new additions, feature testing was used to assure the functions of the Game Mode class were working correctly.

### Conversion and Training Plan

The rules for the game are set at the beginning, and the player has little control over the Game Mode class. A character will automatically start their turn when the previous character ends their turn. A character can end their turn when there are not any options left for the character to perform, namely moving and attacking. If a character does not wish to move, they can click on the tile that they are currently on. If a character does not wish to attack, they can again click on the tile that they are currently on.

### Final Subsystem

As with the Player Controller, the Game Mode subsystem will need to be continuously updated to support new features that the game may provide in the future.

## XI. Player Character - Timothy Canipe

### Refined model

The Player Character subsystem is shown in the system model in section V.  The Player Character class contain the graphical representation of the character that the user plays as.

### Refined design

The Player Character is used to display the character in a way that the user is able to interact with it.  It also contains the different character attributes.  Each Player Controller will have three instances of this class if playing solo, and will have one instance if playing multiplayer.

### Coding and testing

The base Player Character was provided by the Advanced Turn Based Tile Toolkit. This base Player Character class has been added to in order to add new features.  The additions to this base class have been marked appropriately, so as to tell what was added and what was provided.  Along with this base class, there is an additional child class for each type of character created.  Each of these classes were tested using feature testing.

### Conversion and training plan

The Player Characters are spawned at the start of the game, and the Players do not interact with the Player Characters directly, instead they interact with the Player Characters through the player Controller.

### Final subsystem

The Player Character subsystem was created with change in mind.  It can be easily updated in the future.

## XII. Non-Player Character - Timothy Canipe

### Refined model

The Non-Player Character subsystem is shown in the system model in section V.  The Non-Player Character class contain the graphical representation of the enemy character that the user plays against.

### Refined design

The Non-Player Character is used to display the enemy so that the user is able to interact with it.  There will be one instance of this class for each enemy present on the map.

### Coding and testing

The Non-Player Character class has the same basic code as the Player Character class, but has a few differing variables and is assigned to an AI Controller instead of a Player Controller.  Each of these classes were tested using feature testing.

### Conversion and training plan

The Non-Player Characters are spawned at the start of the game, and the Players do not interact directly with the Non-Player Characters, instead the AI Controllers interact with them.

### Final subsystem

The Non-Player Character subsystem is easily updated and more Non-Player Characters can easily be added.

## XIII. Multiplayer - David Bond

### Refined model

The Network State is shown in the System Model and Client\Server Model in Section V. The Network State is responsible for coordinating multiplayer components, such as network connections and player state.

### Refined design

The Network state will use a client\server model, where one of the players will host the match. Each actor maintains a list of properties that can be marked for replication to other clients. Whenever the value of the variable changes it will be passed to the server to be replicated out to the clients. The server is the authoritative source of player state updates.

### Coding (includes planning) and testing

Unreal engine provides networking and multiplayer facilities through the use of sessions, which are coordinated instances of the game. The logic for coordinating network connections and authentication are builtin to the Sessions code. Games on a LAN will be coordinated directly through Unreal engine, while games over an internet or WAN connection will use the Steam network. The relevant code for the game sessions is contained in the Game Mode class, and has functions that will create, search for, join, and destroy a session. Each of those functions is tied to a menu item or event in the process of establishing a multiplayer game.

### Conversion and training plan

The player will not interact directly with the Network State, but is heavily dependent upon it for the multiplayer to work correctly. The most direct interaction the player will have will be on the Create a Lobby menu, where the player specifies attributes for a multiplayer match, such the number of players and the connection type.

### Final subsystem

The multiplayer subsystem, including network connections and game state coordination is functioning as intended, but may need some slight adjustment while correcting a camera issue.

## XIV. Game Level- Vishal Bhatt

### Refined model

The Game Level model is shown as Game Level in the Entity-Relationship model in section V. These aspects contain the implementation of many free assets that facilitated the build time.

### Refined design

This model is responsible for wrapping all other components, without this model there would be no game. The player has the ability to move their players via the turn based toolkit that was purchased. The map is 80 X 60 and was inspired by a town from a medieval times.

### Coding (includes planning) and testing

There was no coding involved with this submodel as most of the components are drag and drop. In order to test this there had to be multiple matches played in order to identify flaws in the game level.

### Conversion and training plan

To use the map you must have mouse, scrolling in and out will enable you to change the view. While A,S,D,W are to pan around the map which helps to identify chests and secrets that are stored in the level.

### Final subsystem

This map can be implemented in any type of game, all one must do is copy and paste the main files that hold the map and the textures so that the game level is intact. From here one can only add a whole new level.

# XV. Complete System

### Final software product

The final software product will be available through our website at https://vybhatt95.github.io/DMT-V/

Source code, a user manual, and proper documentation for assets used in the project are available on our github at https://github.com/Vybhatt95/DMT-V

### Resources

[1] Epic Games Inc. (n.d.). *Game Mode and Game State.* [Online]. Available: https://docs.unrealengine.com/latest/INT/Gameplay/Framework/GameMode/, Accessed Mar. 10, 2017.

[2] Epic Games Inc. (n.d.). *PlayerController.* [Online]. Available: https://docs.unrealengine.com/latest/INT/Gameplay/Framework/Controller/PlayerController/, Accessed Mar. 10, 2017.

[3] Epic Games Inc. (n.d.). *AIController.* [Online]. Available: https://docs.unrealengine.com/latest/INT/Gameplay/Framework/Controller/AIController/index.html, Accessed Mar. 10, 2017.

[4] Epic Games Inc. (n.d.). *Pawn.* [Online]. Available: https://docs.unrealengine.com/latest/INT/Gameplay/Framework/Pawn/, Accessed Mar. 10, 2017.

[5] Epic Games Inc. (n.d.). *APlayerState.* [Online]. Available: https://docs.unrealengine.com/latest/INT/API/Runtime/Engine/GameFramework/APlayerState/index.html, Accessed Mar. 10, 2017.

[6] Epic Games Inc. (n.d.). *Sessions and Matchmaking.* [Online]. Available: https://docs.unrealengine.com/latest/INT/Programming/Online/Interfaces/Session/, Accessed Mar. 10, 2017.

[7] J. Avila, "Moderna Graphical Interface", 2010. [Online]. Available: http://opengameart.org/content/moderna-graphical-interface