David Bond

Matthew Yengle

Timothy Canipe

Vishal Bhatt

Group 6

CSC 490

Report 2: Requirements Analysis & Specification

Project Name

DMT-V

Project Goal

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

Project 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.

Note: For video game genre definitions, please see the document on our github: DMT-V/Documentation/Analysis of Video Game Genres.

Subproblems

Beyond scope:

1. Port to iOS, android, linux, etc.
2. Single-player campaign (or balance for single-player play).
3. Aesthetically pleasing - as far as graphics, animations, models, and meshes.

Requirements

The game will:

1. (Functional) Include multiplayer functionality:
   1. Using a client\Server Model.
   2. An in game chat system will not be provided.
   3. An area for people to find other people to play with will not be provided in the actual game.
   4. Internet connection (56Kbps or faster).
   5. One user designated as the server; all others will be clients to this server
   6. The server (and only the server) will maintain a single copy of the GameMode actor.
   7. The server will provide each client with a GameState actor.
   8. Server will provide clients with current map when they connect.
   9. Successful client connections will receive a temporary PlayerController to replicate.
   10. 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:
   1. Other player characters in the game.
   2. Enemy units and other non-player characters in the game world.
   3. The game world that the player character is in.
3. (Functional) Save the state of the game.
   1. 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.
   1. Each player character will have their own inventory system.
   2. Players will be allowed to trade items.
   3. Players can add items to their inventory from the game world.
   4. Players can drop items from their inventory in the game world.
   5. Players can equip equipable items.
6. (Functional) A combat system that gives the player options depending on their class.
   1. (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.
   1. (Performance) While retaining fast-paced and exciting gameplay.
8. (Functional) Feature an attribute system.
   1. (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.
   1. (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.
    1. Include a Heads-Up Display (HUD) that the user will be able to see their player character’s stats and statuses.
    2. (Performance) Responsive and well-designed layout of the visual design.
11. (System) Be able to run on desktop computers running Windows 7 or newer.
    1. 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. (Security) Prevent the user from switching characters between instances of different games.
    1. Characters should be locked into one game, from which they cannot be moved.
15. (Security) Prevent the user from tampering with save files in order to change their game in unintended ways.

Risk Assessment

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.
   1. 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.
   1. One member is well-experienced with C++, and there are plenty of available resources for C++.
   2. 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.
   1. 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.
   1. Independent videogame development is a well-supported community on the Internet, with plenty of information, resources, and free assets.

System model:

(Note: Full-sized model will be included in the submitted folder as well as on our github page.)

System Model CSC490.png

Subsystems

**Game Mode**

Description: The Game Modesubsystem will be used as the central hub for the interaction between all other subsystems. 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 n amount of Non-Player Character classes currently in play. The Game World class will also be used to cycle through the different Game Levels, which are changed according to some requirement rules defined by the game mode. Every action performed by the player character should be checked by a rule in the Game Mode class for validation. This will, in turn, allow the Game Mode class to tell the Game State class the changes caused by the player’s action. As the Game Mode class does not actually store data on the current game, it will not need to be synchronized between the players of an online game.

**Game State**

Description: The Game State refers to the complete set of actors and their associated variables. In a multiplayer environment using the client-server model, each actor maintains a list of properties that can be marked for replication to clients. Whenever the value of the variable changes on the server side, the server sends the client the updated value. The variable may have changed on the client, but it will be overwritten by the new value from the server. Property updates only come from the server: the client will never send property updates to the server. Some properties replicate by default (e.g., Location). Replication is reliable, which means that the property of the client version of the Actor will eventually reflect the value on the server(3).

**User Interface**

Description: The User Interface subsystem will allow the player to interact with the game. In the game, the User Interface will be represented visually by a Heads-Up Display (HUD) showing the player character’s status, an inventory and equipment layout, and a menu system for control over the game settings and options. The User Interface will alert the Game Modewhen the user performs an event on the interface. In return, it will be expecting some type of response from the Game Mode. The UI will then be updated to relay the response to the user in an understandable form. Not all responses in the User Interfaceare a result of the user performing an action. In the Unreal Engine, the user interface will be made using the Unreal Motion Graphics UI Designer.

**Game World**

Description: The game world will consist of a grid based gameplay. The map(s) that are created will give each player the opportunity to move a certain amount of steps. Having the grid will enable the player to select many different movements.

**Game Level**

Description: Levels will have subsections where the environment will change after a certain point in time. Having this change will keep the player interested for longer. Essentially the player will start off in a dungeon making their way up to the boss that will be located at the top of the castle.

**Character**

Description: The Character subsystem will be used as the base class for both the Player Character and the Non-Player Character classes.

**Player Character**

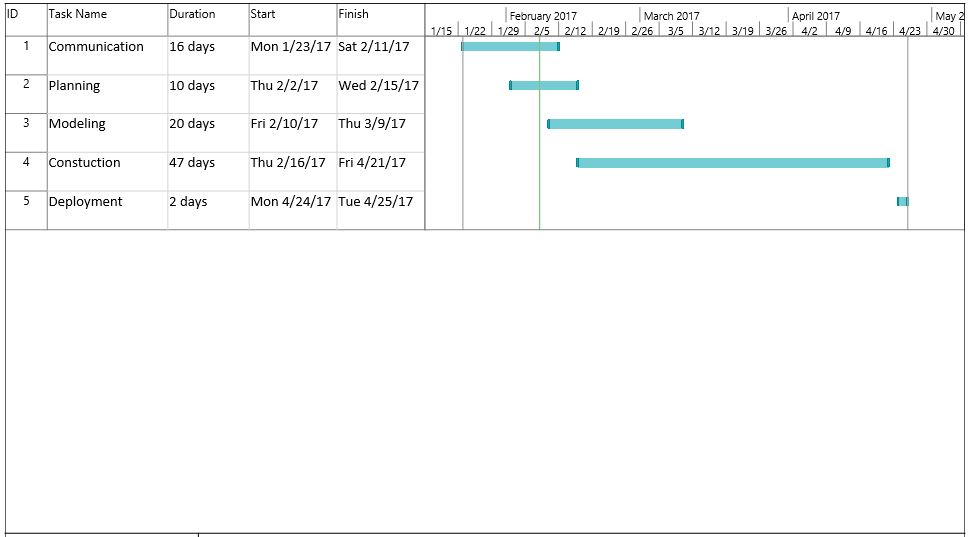
Description: The Player Character subsystem will contain the graphical representation of the character that the user plays as and also communicates with the inventory system, and the Character and Game Mode subsystems. In this game there will be an instance of this subsystem for each of the users who are in the Game World. The user does not interact with the Player Character directly; instead the user interacts with the UI, which interacts with the Game Mode, which then relays the information to the Player Character.

**Non-Player Character**

Description: The Non-Player Character subsystem will contain the graphical representation of all the enemies within the game. It will also interact with the AI system, and the Character and Game Mode subsystems. In the game there will be an instance of this subsystem for each enemy within the game.

Timeline

(Note: Full-sized Gantt chart will be included in the submitted folder as well as on our github page).



Group Roles

Decisions and arguments will be resolved reasonably and democratically, keeping in mind the best interest of the project.

David Bond:

* Coding of the Game State major subsystem.
* Research into the networking aspect of the Unreal Engine.
* Coding of the Save and Load systems.

Matthew Yengle:

* Coding of the Game Mode. Item Database, and User Interface major subsystems.
* Coding of the Equipment and Inventory minor subsystems.
* Research into open-source item models, including weaponry and armor.

Timothy Canipe:

* Coding of the Character, Player Character, and Non-Player Character subsystems.
* Coding of the Skills and Attributes minor subsystems.
* Research into open-source character models, animations, and assets.

Vishal Bhatt:

* Coding of the Game World and Game Level major subsystems.
* Coordination of audio and sound aspects of the game.
* Research into open-source world meshes, level designs, and audio assets.

Use cases:

**Name**: Attack an enemy non-playable character (NPC)

**Actors**: The player character (PC) and the NPC.

**Goal**: To attack the NPC.

**Precondition**: It must be the player character’s turn, they have not fought this turn yet, and the enemy unit is within range of the player character’s attack.

**Summary**: The user wants to attack an enemy unit. They click “Attack” on the GUI, choose their attack, then click on the enemy unit. If all actions were valid, then the user’s character will attack the NPC. The NPC will be properly damaged to reflect the attack.

**Steps**:

|  |  |
| --- | --- |
| User | System |
| The user clicks the “Attack” button on their GUI. | The system processes the input and changes the GUI to acknowledge the user input. It then waits for the next user input. |
| The user clicks on the NPC they wish to attack. | The system validates the target selected by the NPC. It will then check all other preconditions. |
| The user is notified as to whether the attack was valid or not through the GUI. | The game state changes to reflect the damage performed to the NPC. The user is unable to attack again this turn. |

**Post-Condition**: The health of the enemy is updated and the player character cannot attack again this turn.

**Name**: Move the player character

**Actors**: The player character (PC).

**Goals**: To move the player character to the designated tile.

**Preconditions**: It must be the player character’s turn, and the designated destination tile is within movement range.  
**Summary**: The user wants to move their character in the game world. The user clicks “Move” on the GUI, then selects where in the game world they want to move. If the location is valid and the player has moves left, the player will move to that location.

**Steps**:

|  |  |
| --- | --- |
| User | System |
| The user clicks the “Move” button on their GUI. | The system processes the input and changes the GUI to acknowledge the user input. It then waits for the next user input. |
| The user clicks on the space they wish to move to. | The system validates the preconditions. |
| The user sees the character move to the target location. | The game state will update the new location of the player in the game world. |

**Postconditions**: The player’s move count for this turn is decreased by the number of tiles they moved. If this reaches 0, then the player cannot move again until the next turn.

**Name**: Pickup an item

**Actors**: The player character (PC), the item.

**Goals**: To remove the item from the game world and add it to the player’s inventory.

**Preconditions**: The item must be within reach of the player character, and the player must have space in their item inventory.  
**Summary**: The user wants to pick an item up from the game world and add it to their player’s inventory. The player clicks on the item in the game world. If the player is within reach and there is room in their inventory, the user will be given the option to add it to their inventory.

**Steps**:

|  |  |
| --- | --- |
| User | System |
| The user clicks on the item. | The system processes the input and checks the preconditions. |
| The user agrees to add the item to their inventory. | The system gives the player the option to add the item to their inventory. |
| The user can now see the item in their player’s inventory. | The game state changes to show the item has been picked up. The player’s inventory is updated to include the item. |

**Postconditions**: The item is no longer on the ground. The player’s inventory space has less room in it.

**Name**: Use an item.

**Actors**: The player character (PC) and the item.

**Goals**: To use the item’s associated effect.

**Preconditions**: It must be the player’s turn, the player must have the item, and the item must be used correctly.

**Summary**: The user wants to use an item that they have in their player character’s inventory. The item’s effect should be given to the user in a short description shown when hovered over that item with the mouse actor. The user will double click the item in their inventory with the mouse, and expect the item effect to activate.

**Steps**:

|  |  |
| --- | --- |
| User | System |
| The user double clicks on the visually represented item in their inventory UI. | The system finds the item in the item DB, and obtains its associated use. |
| The user will be able to tell in some way that the item was or was not successfully used. | If the user used the item in a valid method, the associated use will occur. |

**Postconditions**: The item may or may not be destroyed.

**Name**: Acquire a skill.

**Actors**: The player character (PC).

**Goals**: The player uses skill points to acquire a skill.

**Preconditions**: It must be the player’s turn, the player must have enough skill points available, and the player must be a high enough level.

**Summary**: The player wants to unlock a skill. The player selects the skills button on the UI. If the player has skill points available and is a high enough level, then the user selects which skill to unlock.

**Steps**:

|  |  |
| --- | --- |
| User | System |
| The user clicks on the skills button. | The system processes the input and checks the preconditions. |
| The user selects which skill to acquire. | The system calculates which skills are available. |
| The user can now use the skill when applicable. | The game state changes to show the reduced amount of skill points available. The skill shows as unlocked. |

**Postconditions**: The player’s available skill points decreases by the number of points they used. If this reaches 0, then they cannot unlock any more skills until they obtain more skill points.

**Name**: Increase an attribute.

**Actors**: The player character (PC).

**Goals**: The player uses attribute points to increase their attribute levels.

**Preconditions**: It must be the player’s turn, the player must have points available, and the attribute must not be at max.

**Summary**: The player wants to increase one of it’s attributes. The player selects the stats button on the UI. if the player has points available then the user selects how many points to increase each attribute by, out of it’s available points.

**Steps**:

|  |  |
| --- | --- |
| User | System |
| The user clicks on the stats button. | The system processes the input and checks the preconditions. |
| The user selects how many points to use. | The system calculates the total available points. |
| The user can now see the changes in their stats. | The game state changes to show the reduced amount of points available. The player’s stats are updated. |

**Postconditions**: The player’s available stat points decreases by the number of points they used. If this reaches 0, then they cannot increase any attributes until they level up.

**Name**: Save the game.

**Actors**: The player character (PC).

**Goals**: The player wants to save the current game state.

**Preconditions**: The user must be in the game.

**Summary**: The player wants to save the current state of the game by using a menu system. In Unreal Engine, particular components of the state or the entire game state can be saved using a SaveGame Object. For simplicity, all components of the game state will be saved.

**Steps**:

|  |  |
| --- | --- |
| User | System |
| User opens menu system | A menu of options is presented to the player |
| User selects to Save | Unreal Engine creates a SaveGame Object, which is written to the hard drive. Game play resumes. |

**Post-conditions**: The exact state of the game will be saved to the user’s disk drive.

**Name**: Load the game.

**Actors**: The player character (PC).

**Goals**: The player wants to load a previously saved game

**Preconditions**: The user acting as the server must load the game. The original members of the save file must be connected to that server.

**Summary**: The player wants to load saved state of the game by using a menu system. In Unreal Engine, particular components of the state or the entire game state can be loaded using a SaveGame Object. For simplicity, all components of the game state will be loaded

**Steps:**

|  |  |
| --- | --- |
| User | System |
| User opens menu system | A menu of options is presented to the player |
| User selects to Load | Unreal Engine loads a SaveGame Object, which is read from the hard drive |
| User sees previous saved state | Unreal Engine resumes gameplay from the SaveObject state |

**Post-condition**: The users are loaded into game in the exact state as when the game was saved.

**Design Choices:**

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 various animations.

-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.