NTNU
Norwegian University of
Science and Technology

Project: Avacyn

Author(s)

Sebastian Kolbu
Kristian Gregersen
Petter Ballangrud

Bachelor in Game Programming
20 ECTS
Department of Computer Science
Norwegian University of Science and Technology,

16.05.2017

Supervisor(s)
Simon McCallum
Mariusz Nowostawski

Sammendrag av Bacheloroppgaven

Tittel:	**Prosjekt: Avacyn**
Dato:	16.05.2017
Deltakere:	Sebastian Kolbu Kristian Gregersen Petter Ballangrud
Veiledere:	Mariusz Nowostawski Simon McCallum
Oppdragsgiver:	Norwegian University of Science and Technology
Kontaktperson:	Sebastian Kolbu, sebastnk@stud.ntnu.no
Nøkkelord:	Spill, Programmering, Unreal Engine, C++, Blueprints, Dungeon Crawler, Action, RPG, Latex, IMT
Antall sider:	86
Antall vedlegg:	
Tilgjengelighet:	Åpen

Sammendrag: Prosjekt: Avacyn er et prosjekt om å lage et dungeon crawler spill i Unreal Engine 4. Vår implementasjon trekker inspirasjon og gjør forbedringer av ulike aspekter fra tidligere utgitte spill i samme sjanger. Utviklingen av spillet gir insikt til hvordan det er å utvikle et profesjonelt spill med Vulkan API integrasjon.

Summary of Graduate Project

Title:	**Project: Avacyn**
Date:	16.05.2017
Authors:	Sebastian Kolbu Kristian Gregersen Petter Ballangrud
Supervisor:	Mariusz Nowostawski Simon McCallum
Employer:	Norwegian University of Science and Technology
Contact Person:	Sebastian Kolbu, sebastnk@stud.ntnu.no
Keywords:	Game, Programming, Unreal Engine, C++, Blueprints, Dungeon Crawler, Action, RPG, Latex, IMT
Pages:	86
Attachments:	
Availability:	Open

Abstract: Project: Avacyn is a project to create a dungeon crawler game in Unreal Engine 4. Our implementation draws inspiration and improves on aspects from previously released titles of the same genre. The creation of the game also serves as an insight to how to professionally create a game with vulkan API integration.

Preface

We would like to thank the following:

- Erik Hjelmås for motivating the development of a LaTeX template for GUC's master's theses. This template has been a tremendous help by allowing us to focus on writing.
- Epic Games for an amazing game engine, good documentation, and excellent tutorials.
- Botanic from #unrealengine on freenode.net irc server for always being available to answer questions about blueprints.

Contents

List of Figures

1 Introduction

An introduction to our project, nicknamed Project: Avacyn. Terminology used for this thesis can be found in Appendix A.

1.1 Project Description

1.1.1 Background

Prior to this bachelor thesis, the three of us had collaborated on a couple of other projects during the Game Programming course, one of which being the project in the game programming subject where we decided to make a dungeon crawler game from scratch.

That project ended up as a tech demo of the engine we had created, as after coding the framework; networking, rendering etc. and the basics of character movement and camera control, we ran out of time. As that project focused on building a game engine, we did not have enough time to implement game mechanics. Therefore, we decided to implement the game we envisioned as our bachelor's project. Our third member wanted to work on GPU programming with Vulkan for his bachelors project. Vulkan provides more low level control over the graphics pipeline, opening up for greater performance optimization. We thought it could be beneficial for our game and decided to combine the projects as a unified thesis.

On top of enhanced performance, we decided to use Vulkan for dynamic destruction of objects and terrain within the game, which would result in some interesting gameplay experiences.

1.1.2 Motivation

We have been taught a lot about making games during our course here at the university. We have made the back end of a game engine. As mentioned earlier we had previously set out with the goal of making a game, but with so much work required to make the back end work, there was no time left to actually focus on making the game. Therefore we want to put our knowledge to use by creating a game.

To avoid the problem of not being able to focus on anything but the game engine, and hopefully manage to achieve a higher production quality of our work, we decided to use a preexisting game engine. This would also allow us to learn the ins and outs of one of the most popular game engines used by the industry, as well as teach us how to adapt to new development environments. We decided to use Unreal Engine 4, more details about this can be found at chapter 4.2.4.

1.1.3 Goals

Result Goals

- Produce a functional alpha version of a game, demonstrating robust, scalable and flexible implementations of different game systems.
- Excellent game design.
- Well written code with comments that both follow the decided upon standards.

- Proper documentation that accurately describes the game we are trying to create.
- Extensive planning to ensure a smooth implementation.

Effect Goals

- Deepen our understanding of C++ programming, especially in relation to larger systems that need to be both scalable and flexible.
- Familiarize ourselves with in the process of professional programming.
- Test and expand upon our knowledge of game design both from a creative and technical point of view.
- Obtain a deeper understanding of a professional game engine.
- Learn the visual scripting language Blueprints.

1.2 Project Organization

1.2.1 Academic Background

Everyone on the group has attends the Game Programming course, and therefore have a lot of experience with programming in C++. We have also gained game design experience from the subjects Game Design and Rapid Prototyping. Through the subject Rapid Prototyping, we took part in game jams, granting us the opportunity to test out both Unreal Engine and Unity before making a choice for which engine to use for this thesis.

In addition to this, our group member that was going to work with Vulkan had previous experience from research he had done in his spare time.

1.2.2 Responsibilities and Roles

For the project Sebastian was unanimously chosen as the group leader by the group. His role as the group leader would be; making sure progress is made on the project, setting up appointments with our supervisor if we felt the need for this, and directing the work flow.

Project responsibilities were divided as follows:

- Sebastian:
  - Game Design
  - Inventory System
  - GUI Implementation
  - Talent System
  - Stats System
- Kristian:
  - Game Design
  - Spell System
  - Artificial Intelligence
  - Stats System
- Petter:
  - GPU Programming
  - Modifying Unreal Engine to run on the Vulkan API
  - Dynamic Destruction

    ◦ Optimizations related to the GPU

1.2.3 Practises and Rules

Upon starting this project each group member signed a project agreement. These rules and practices were implemented with consequences for breaking the most severe rules, mainly pertaining to the rule about working on schedule:

- Time will be tracked using Toggl (Time-tracking tool).
- Work will be done from home, using our own desktop computers, as Unreal Engine is too demanding for some members laptops.
- Follow the working schedule starting Monday, Jan 9th.
- During scheduled work sessions you should always be available in the groups discord server.
- Follow the agreed upon coding and commenting standards.
- Remember to commit work and use the issue tracker.
- If a member does not work their share, the other members can discuss whether or not to kick them out of the group.
- If a member becomes ill and is unable to work, said member will let the other group members know before the next planned work session.

1.3 Development Plan

1.3.1 Software Development Methodology

We decided on using the agile development model Scrum for our project. This was because it is a development model we are all familiar with. The sprints also allow us to easily re-prioritize what we work on, and potentially cut features, to make up for time delays on other features on a regular basis. This is essential for a game development on a tight schedule, especially when we are fairly inexperienced with multiple parts of the project.

We decided to have the sprints last for two week. The sprints would start on Mondays, where we would start the day of with planning the next sprint. The planning consisted of selecting tasks to be worked on over the period of the sprint, as well as assigning each member to these tasks.

1.3.2 Schedule

At the beginning of the planning phase we created a Gantt chart (see figure 1) to as an estimate on how much time we would spend on each tasks as well as which tasks would belong to the different sprints over the project period. In the chart we had all the core features of the game mapped out, but assuming everything probably wouldn't be implemented without problems we were prepared to have to cut features for the end of the thesis version of the game.

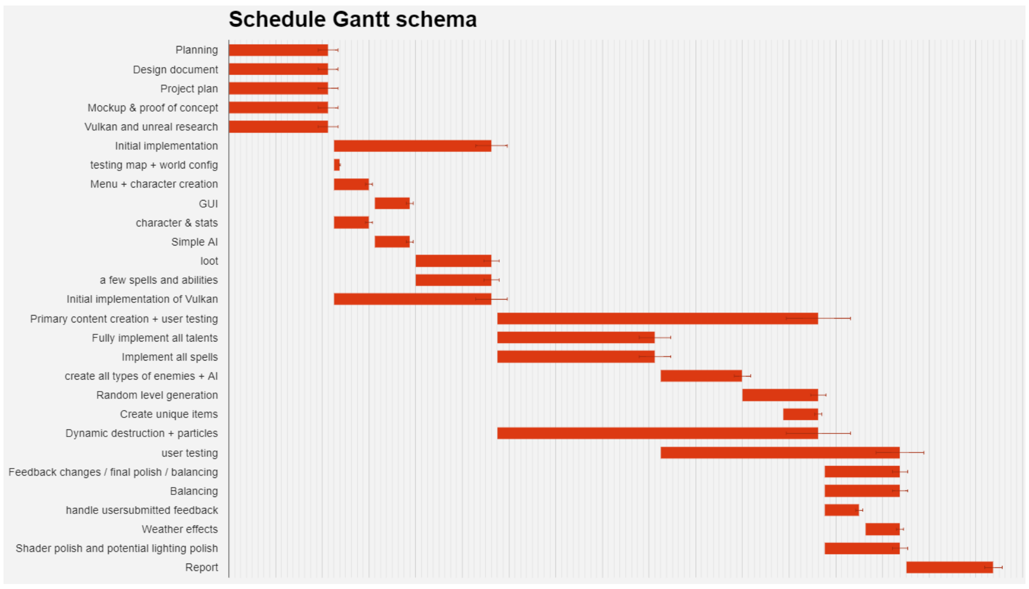


Figure 1: Initial Gantt Diagram

2 Requirements

2.1 Functionality

The main focus of our thesis was to implement these core features in a robust, scalable and flexible way:

- An AI that chase the player and attack them.
- A system that manages all the different stats and uses them for:
  - Calculating the damage of attacks
  - Calculating the damage received from attacks
  - Keeping track of the players resources (HP, Mana etc.)
- A GUI that allows the player to interact with the core features of the game.
- A system that allows the player and the AI to use all the spells available to their character.
- A main menu with a character creation tool that allows you to create a new character.
- Automatic saving and loading functionality to allow you to keep playing from where you last left off.
- A system that keeps track of your talent points, and allow you to spend them to further advance your character.
- An inventory system without a limit to how many items it can contain that has sorting filters to let you search through it quickly.
- A rich and diverse loot system allowing for synergies between loot and talents.
- Procedural level generation.
- Unreal Engine using Vulkan for rendering.
- Dynamic destruction of objects in the game world.

2.2 Usability

The player would initially start out in an area with very few mobs. With some on screen pop-ups displaying the very basic controls as guidance they would learn the basics of fighting monsters. After fighting monsters and progressing through this starting area they would eventually level up, again some pop-ups would appear, with instructions of how to spend your newly acquired talent point. There would be similar style pop-up tutorials throughout the game, popping up whenever the the user encounters something new in the game that we felt needed explanation. For experienced players, or players not interested in guidance, these tips could be disabled simply by checking a checkbox that appears with each new pop-up. In the settings menu there should also be a corresponding checkbox to alter this behavior under game settings.

We also tried to keep the GUI similar to that of other games in the same genre, where that would make sense for our game. Or in other cases, similar to other popular games in general. The key bindings for the action bars should also be changeable by the player to suit their play style.

2.3 Reliability

The game should be able to be played by the user for as long as they would like without the game crashing for any reason. Because of this expectation, it was all the more important to ensure no problems like memory leaks or incorrect destruction of objects happened.

2.4 Performance

The game is expected to run at 60 frames per second or higher at any given time on a modern gaming desktop or high end laptop. This is however not expected on lower end machines as the quality achieved by the unreal engine can become quite demanding.

To combat the demands of the engine on lower end machines, quality options would be implemented, to allow users to sacrifice the aesthetics of the game for a higher performance. This would ensure the game would be available to a larger portion of the market. Minimum system requirements from unreal can be found at [1]

As there could be a lot of enemies in play at the same time, a key point to keeping the performance requirements for the game down was to create the AI in a way that did not require a lot of processing resources. The behavior tree helped out a lot with this as it provided us with the functionality to only tick the tree every once in awhile instead of every game tick. Additionally certain nodes in the tree were made so they could return the state 'in progress', which meant that during the next tick of the behavior tree, it could just continue from where it last left off instead of traversing the whole tree again. This feature was used for example for the movement node, as walking around might take quite some time before it finishes.

Vulkan will allow is to gain performance compared to older graphics APIs, this does require more work in development to better optimize for modern, multi-threaded hardware.

3 Technical Design

3.1 Unreal Engine

We started developing the game using Unreal Engine 4.14.X but upgraded to version 4.15.X in the middle of the project. A feature of the engine is their visual scripting language called Blueprints. To use this in combination with C++ programming, we had to expose our code using macros defined by the epic team. See Figure 2.

```cpp
UFUNCTION(BlueprintCallable, BlueprintPure, Category = "Statistics")
float GetStat(ETypeOfStat stat);
```

Figure 2: C++ function exposed to blueprints

The UFUNCTION() macro tells the engine that I am about to declare a function. For the case above we chose to use the specifiers BlueprintCallable and BlueprintPure. BlueprintCallable allows any blueprint with access to this class to call and execute the function, while BlueprintPure means that the function does not affect the owning object in any ways. Specifying these lets the engine know where and how the function can and cannot be used. There are also other keywords that could be used;

BlueprintImplementableEvent: The function can be overridden in a blueprint.

BlueprintNativeEvent: The function is designed to be overridden in a blueprint, but also has a native implementation.

BlueprintAuthorityOnly: The function will not execute from blueprint code if running on something without network authority.

BlueprintCosmetic: The function is cosmetic and will not run on dedicated servers.

There are also macros for defining other types of objects.

UCLASS: Used for defining classes.

USTRUCT: Used for defining structs.

UENUM: Used for defining enums.

UPROPERTY: Used for defining variables.

UINTERFACE: Used for defining interfaces.

These share many specifiers among each other, most notable of the once we used are:

BlueprintType: Exposes this object as a type that can be used for variables in Blueprints.

BlueprintReadWrite: This property can be read or written to from a Blueprint.

BlueprintReadOnly: This property can only be read from by Blueprints.

MinimalAPI: Causes only the objects type information to be exported for use by other modules, this improves compile times for classes that do not need all their functions accessible in other modules.

BlueprintSpawnableComponent: Component class can be spawned by blueprints.

Support for Maps in Blueprints was added with version 4.15. Prior to this version update, we implemented functions in the statistics component to gain access to the map. The way we managed this was to extract the keys and values as two arrays with get-functions.

3.2 Components

When designing the stats system we knew we wanted to give the stats to both the player and the monsters, this could have been done through inheritance, as the player character and the base enemy classes both inherits from the engine's pawn class at some point. But it didn't seem like a very good solution because not only the player and enemies inherit from this class, and having a lot of stats on things that doesn't need them would unnecessarily increase the amount of memory (RAM) the game would need to run, and possibly also affect performance.

A better solution would be to use what Unreal Engine refers to as components. Components are modules of code that can be attached to any actor in the game. This allowed us to create a statistics component and simply attach it to the player character and base enemy classes.

We also used this feature when creating the spell casting component later on, as we wanted both the player and enemies to potentially have access to all the spells in the game.

3.3 Base Classes

We used base classes for both enemies and spells in our game. Having base classes means we don't have to code the same base functionality every child will need to have multiple times, examples of these kinds of functionalities are interfaces for receiving damage and healing or attaching components.

3.4 Importing spell data from csv file

With our complex set of talents and spells, it became clear to us that having a proper way to import the data into the game would be preferable to hard-coding. The desire to make game balance adjustments to spells and talents was also present, and we needed an implementation which supported quick adjustments. We use a google drive spreadsheet to store all information about talents, and export the document in the form of a csv (comma separated values) document. This document is then imported into an Unreal Engine DataTable.

"DataTables, as the name implies, is a table of miscellaneous but related data grouped in a meaningful and useful way, where the data fields can be any valid UObject property, in-

cluding asset references"[2].

In order to utilize the data table functionality in Unreal engine, we needed to create a struct inheriting from FTableRowBase. This struct has a variable with a descriptive name for each column in the csv file. The data table consist of 40 different variables, and each variable belongs to one of five categories. the categories are: General information (used for tooltips regarding the talents and spell), base spell information, DoT (damage over time) effect, Status effect, CC (crowd control) effect, and passive.

The data is then imported into an asset by unreal engine and can be accessed during run-time. We decided to store the information about talents and spells in the Game instance, due to it being persistent through the levels and allowing us to have access to the talent and spell information in both the main menu as well as in the world. The data is stored in a custom component in the game instance called 'SpellDetails'.

In the constructor for the game instance class, we extract the information from each of the rows in the data table into an array. This array is then passed along as a parameter to the function InitializeSpellDetails, where a for-loop is performed on the array, dividing the vaiables into the five categories previously mentioned. There is a custom struct for each category. These structs are stored in a map for each category(see figure 3). 'Tal-

```
TMap<ETalentName, FTalentInfo> TalentInfo;
TMap<ESpellName, FSpellInfo> SpellInfo;
TMap<ESpellName, FAugmentInfo> AugmentInfo;
TMap<ESpellName, FCCInfo> CCInfo;
TMap<ESpellName, FStatusEffectInfo> StatusEffectInfo;
TMap<ESpellName, FDOTInfo> DOTInfo;
TMap<ESpellName, FPassiveInfo> PassiveInfo;
TMap<ESpellName, FAugmentsPossibleInfo> AugmentsAvailable;
```

Figure 3: Maps containing the separated talent and spell data

entInfo' uses a different key because the base information about the talents are the same for all ranks of the same talent. However, the other maps use the ESpellName which is an enum containing all spells for each rank available. for instance 'Fireball_rank1', 'Fireball_rank2', and 'Fireball_rank3'.

3.5 Saving and Loading Game State

With most games, being able to save a game session and continue it later is a necessary feature. Especially a game that revolves around building up your character with talent points and equipment. The implementation of saving and loading state began during the first few weeks of initial implementation. As we built up the character creation menu, we saw it necessary to add saving and loading in order to confirm that our character creation system was fully functional.

3.5.1 Implementation

In order to save and load a character, we took use of the base functionality 'USaveGame' implemented in the Unreal Engine. USaveGame is a class made with the purpose of saving and loading information relevant to the player character. We implemented a class inheriting from USaveGame called UAvacynSaveGame'. USaveGame serves as sort of a container where you can create variables inside. The entire saveGame class is then serialized to a file located in the 'Saved/SaveGame' folder with a specified name and save slot.

We tasked the game mode with managing saving and loading of state.

3.5.2 Saving State

In the constructor for the game mode, a timer is enabled. This timer is tasked to call the function which handles saving every five seconds. With this, the player does not have to worry about losing progression.

```cpp
void AAvacynGameMode::InitializeAutoSaveGame()
{
  //Set up timer to save every 5 seconds by
  // calling the saveCharacter function
  GetWorldTimerManager().SetTimer(TimerHandle,
                                  this,
                                  &AAvacynGameMode::SaveCharacter,
                                  5.0f,
                                  true);
}
```

Inside the save character function a new instance of UAvacynSaveGame is created. Then all relevant data (see figure 4) is collected from player character, game instance, and the world, then stored within the UAvacynSaveGame. Last but not least, calling the

```cpp
    //name of savefile
  UPROPERTY(VisibleAnywhere, Category = "SaveData")
  FString SaveFileName;

    //save slot
  UPROPERTY(VisibleAnywhere, Category = "SaveData")
  int32 SaveFileSlot;

    //the characters inventory.
  UPROPERTY(VisibleAnywhere, Category = "SaveData")
  TArray<FItem> SaveInventoryItems;

    //player location in the world
  UPROPERTY(VisibleAnywhere, Category = "SaveData")
  FVector PlayerLocation;

    //character name
  UPROPERTY(VisibleAnywhere, Category = "SaveData")
  FString CharacterName;

  //Mesh data
  UPROPERTY(VisibleAnywhere, Category = "SaveData")
  USkeletalMesh* CharacterSkeletalMesh;

  //Talent information
  UPROPERTY(VisibleAnywhere, Category = "SaveData")
  TMap<ETypeOfTalentTree, FTalentTreeAllocationInfo> SaveTalentInfo;
```

Figure 4: Variables of UAvacynSaveGame

Gameplaystatics' function 'SaveGameToSlot'. This is where Unreal Engine takes over and saves the information as a binary file.

Originally we implemented this system with having a saveGame member variable in GameMode. This however did not work properly as it would eventually not be accessible and throw a ReadAccessViolation, resulting in the game as well as the entire Unreal Engine to crash. We believe the cause of this to be memory allocation. The program allocates the size of Game mode. When the size of SaveInventoryItems increased to a point where it would have to reallocate to a larger part of the memory, the reference would not be updated, resulting in the program reading from the incorrect location in memory. When clearing out the content of SaveInventoryItems then adding all inventory items again, it would attempt to clear out a part of memory it does not have access to.

3.5.3 Loading State

Loading is done during beginPlay in game mode. Data necessary in order to load a save game is extracted from Game instance. The character data stored in Game instance are the following:

```
UPROPERTY(VisibleAnywhere, BlueprintReadWrite,Category ="CharacterInformation")
bool bIsNewCharacter;

UPROPERTY(VisibleAnywhere, BlueprintReadWrite,Category ="CharacterInformation")
FString CharacterName;

UPROPERTY(VisibleAnywhere, BlueprintReadWrite,Category ="CharacterInformation")
FString CharacterFileName;

UPROPERTY(VisibleAnywhere, BlueprintReadWrite,Category ="CharacterInformation")
int32   CharacterFileSlot;

UPROPERTY(VisibleAnywhere, BlueprintReadWrite,Category ="CharacterInformation")
USkeletalMesh* CharacterSkeletalMesh;
```

The first variable, bIsNewCharacter lets the game mode know if it should create a new character or load an existing one from file. If the boolean is true, it creates a new saveGame and saves it immediately. However, if the value is false it will load the character.

Loading the character is done much like saving, just in reverse. The saveGame is loaded. If the result is a valid UAvacynSaveGame object, it extracts the information seen in figure 4 and set their associated values.

3.6 Map Generation

As our game is inherently fairly repetitive in nature we wanted to randomly generate maps to add some extra replayability where we could. We decided that randomly generating locations such as towns with friendly non player characters (NPCs) was rather unnecessary, and wouldn't add much in the way of gameplay, so those are pre made in the level editor. But our dungeons would be randomly generated using themes, layout styles, types of monsters and bosses etc. as variables for a generation algorithm that draws inspiration from both older titles like Angband or Tales of Maj'Eyal and newer titles like Diablo 3.

We looked at several methods for procedural generation of maps for our game, as we would only be generating dungeons and not the overworld we used that to figure out what we required form the generation algorithm. The method we chose to generate a baseline dungeon outline was cellular automata, a method that is very good at generating natural looking caves and caverns. We would take the result from this algorithm and run it through several layers of post-processing to achieve a good layout as an end result. The layout of the dungeon would have type specifiers for what type of "tile" would be used in different locations. The final step would be to place our world pieces according to the layout we generated.

4 Development Process

4.1 Working Hours

We decided early in the project that we would follow a set work schedule as we were going to work form home, and we needed to ensure that everyone was actually working. We decided that we would be working at least from 10-17 every weekday. On Thursdays we had GPU lectures, so we would spent the day working on that course if needed. Fridays were for code review and professional programming lectures.

4.2 Development Tools

4.2.1 Version Control

For version control we decided to use git as our source control system, and to use Bitbucket as our repository provider. We decided that we would commit our work often, even if it did not work. Then when it compiled we would push it to the appropriate branch in the repository. We would merge the branches when the functionality being worked on was finished.

4.2.2 Issue Tracker

We chose JIRA as our issue tracking system due to its integration with Bitbucket. JIRA was used to organize our development sprints, and track our progress. JIRA was also used to link commit messages with task progress.

4.2.3 Documentation

All documentation we wrote during the project was stored in a Google drive folder for the project. It contains all of our design documents, our meeting logs, individual logs, and our group rules. We used Google drive because it allows us to co-edit documents.

4.2.4 Engine

We decided to work with Unreal Engine for the project as it was the best option based on what we wanted to achieve. The other option we considered was to use Unity, but it did not match what we wanted to the same degree as Unreal Engine. We wanted to work with C++ as our programming language. Unreal Engine uses C++ as the programming language, this gave it an advantage over Unity which uses C#. Another big influence on the choice of engine was that we needed access to the source code to let us work with Vulkan. We wanted to work in a tool used by professionals to develop games, so that we could gain experience with tools that may become relevant in future jobs. We also had some experience with both Unity and Unreal Engine, and we liked Unreal Engine the best out of the two. When we considered all of this the choice was quite easy to make.

4.3 Communication Tools

As we were working from home we needed a way to communicate easily and quickly. We chose Discord as our main communication method because it has some nice features we found useful. It has voice chat with the option to go into separate channels so we could be available at all times, but not be distracted by someone talking. Discord also has text chat so it is possible to communicate with people even if they're not online. Discord also offer features such as posting screenshots without having to upload to a remote image-host site. We also used Skype's screen sharing feature when it was necessary to discuss visual aspects of the game, or help each other with coding challenges.

4.4 Workflow

When we started the project we sat down to plan our workflow. We decided to use scrum as our development methodology, and that we would have two week sprints that went from Monday to Friday the next week. We started each sprint with a meeting discussing our progress during the previous sprint, and what we would be doing during this sprint.

In addition to this we decided to have code reviews on Fridays, where we would look over each others code, the main goals behind doing this were:

- To find and fix flaws in the code as soon as possible.
- To better share understanding of the code and learn from each other.
- To make sure coding standards are followed.
- Understanding everyone's code will allow one person to step in for someone else should they not be available for any reason.
- To help maintain a level of consistency in our work.

4.5 Coding Style

As we were going to be using the Unreal Engine for creating our game, we decided early on that we would use the coding and commenting standard made by the Epic team. The coding convention can be found at [3].

5 Graphical User Interface

Graphical user interface is all in which the user can interact with. As with any computer program, a game requires elements the player can interact with. These elements were created and implemented using UMG (Unreal Motion Graphics). UMG is a tool for developing visual interfaces in Unreal Engine. UMG has a lot to offer, such as buttons, list, containment boxes (Horizontal box, vertical box, size box, etc.), and more which allows the developer to take full use of these features without having to implement them from scratch. These features can be modified and adjusted so that they fit the developers need. Figure 5 show an example of developing blueprints. The code developed in the figure is

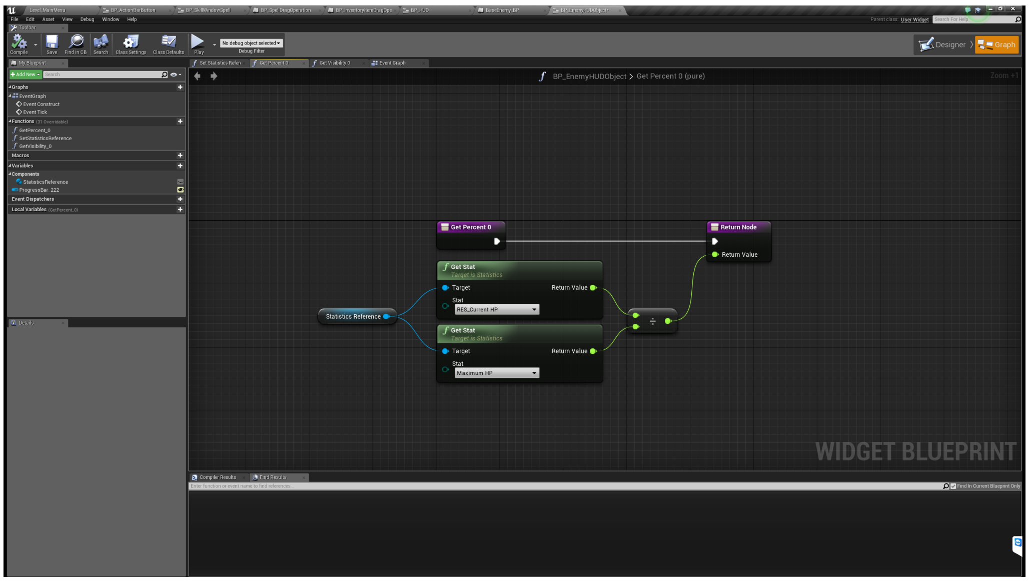


Figure 5: Blueprint visual scripting tool

an implementation of a binding. A binding is a way of connecting a widget element to a variable. When the variable is updated, the code shown is executed. In this case, when the health of an enemy unit changes, the health bar above the unit is updated.

In order to use UMG we needed to rely heavily upon the Unreal Engine Blueprints. Initially we were not very fond of this idea, as none of us had any experience using blueprints. After researching how GUI elements are implemented in C++ we came to the conclusion that it would take a lot more time to do it in C++ than it would take to become familiar with Blueprints.

5.1 Main Menu

The main menu is an essential part of a game. it is the entry point in which the player is greeted, and it was a logical place to begin developing the interface.

In the planning phase we created a mock-up of the main menu, containing which widgets and assets we would need. It's a simple menu, allowing the player to traverse between character selection, configuration, credits and quit the game. Not displayed in

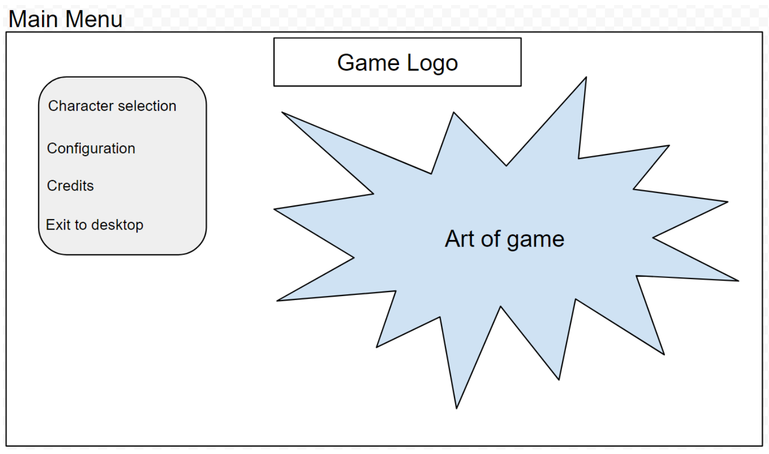


Figure 6: Main menu mock-up

the figure is a hidden 'back button'. the button would appear when you navigate away from the default screen. The back button will always take you back to the default screen, no matter where in the menu you currently are.

5.1.1 Initial Setup

The initial setup for main menu consisted of creating:

- Level asset - used for placing physical objects in.
- Main menu game mode - used for determining the rules for the main menu.
- menu player controller - takes input from the user with functionality specific to the main menu.
- Widget blueprints:
  BP_MainMenu - Main widget which creates and manages the underlying widgets.
  BP_CharacterListButton - Widget with functionality for when it's pressed.
  BP_Credits - styling for the credits page

level Asset

This level was to be the physical world where we would set the stage for character selection. In the first implementation of character selection, we set up five dummy models with a camera point at each of them. However, after realizing that the player might have more than five characters created, this system would not scale at all. Therefore, we removed all but one dummy model and camera, and switched to a method of changing the skeletal mesh of the model to the mesh associated with the character.

Main Menu Game Mode

This is a separate game mode created to be used with the main menu level. Information relevant to the main menu such as an array containing all names of characters the player has available, which character button is selected, and an array containing references to the the different skeletal meshes to be used with the dummy model.

Menu Player Controller

The Menu player controller is where we would have functionality for handling input from the user in the main menu. Early on when we had the implementation with multiple dummy models, we implemented features for clicking on the models to zoom in on the model to get a closer look. Additional functionality was implemented for clicking on another model to switch view to that model instead. The player could click the model again to zoom out. As model setup was refactored, the functionality for switching cameras became redundant. As of now, the Menu player controller has no relevant functionality.

BP_MainMenu

BP_MainMenu is the widget blueprint which contains all widgets. It also holds all functionality for traversing between the different menus. When a menu button is clicked, for instance the Configuration button, The container holding all elements in that menu is set to visible, as well as hiding the main menu elements. When the back button is clicked, the configuration menu and back button are set to hidden again.

BP_CharacterListButton

See section about Character Selection (5.1.2) for information about BP_CharacterListButton and its functionality.

BP_Credits

This is where the information about the developers is available. As we are only three developers, there was not much to do other than list our names and a short description of our motivation for creating the game.

5.1.2 Character Selection

From the main menu, you can navigate to the character selection screen 6. This is where you select a character from a list of characters you have created. these characters are gathered from the "saved/SaveGame/" folder located in the project folder. The list of characters is populated by looking through the SaveGame folder for files beginning with 'Character_', appended by a number starting with 1 and ending with 100. If a file is found, an instance of BP_CharacterListButton is created and added to the vertical box containing all characters. This button has a variable called 'AssociatedSaveGame' which is then immediately set. This variable stores the name of the save file and when the button is clicked, the information in game instance about which character to load is changed to the value the CharacterListButton has.

When launching the game for the first time, there would be no character saves available to load. This will result in the list containing available characters to be empty. The player would have to create a new character. This is done by clicking the new button in the character selection menu 7. When the new button has been clicked, the play button would be hidden and a text field, create button, and a list of buttons will appear(see figure 8). The text field is where the player inputs the desired name for the new character. the list of buttons each have a skeletal mesh associated with them. when clicked, they will change the skeletal mesh of the model shown to the player. the purpose of these buttons is to allow the player to customize which appearance they want for their

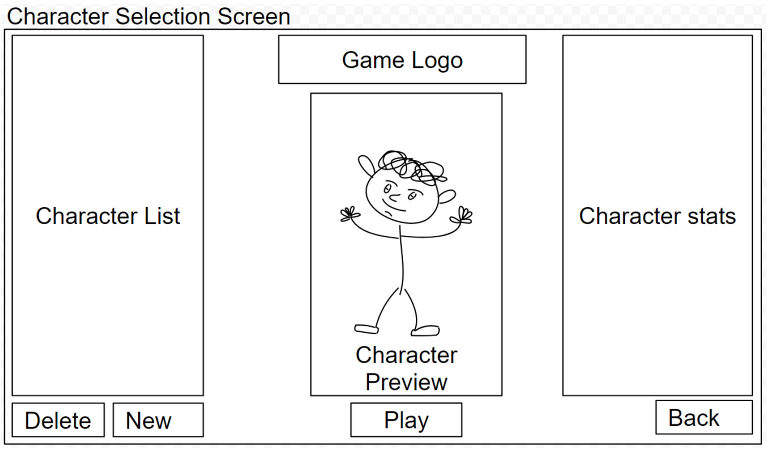


Figure 7: Character Selection mock-up

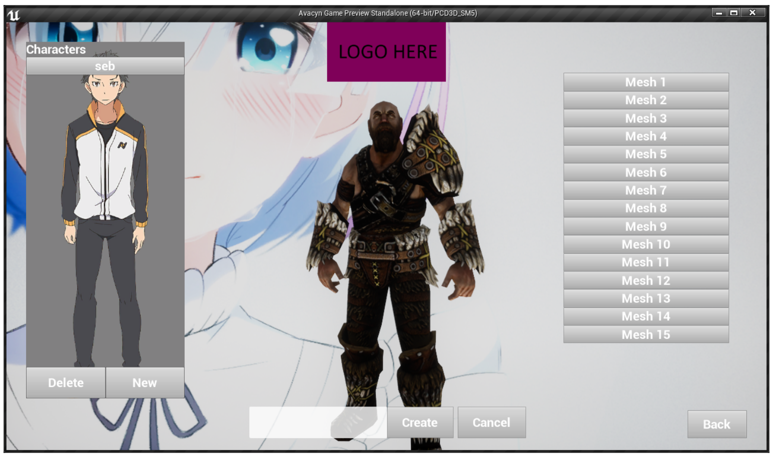


Figure 8: Character Creation with placeholder art

character.

When the player has either selected a character from the list or created a new one, the information necessary to load the character is stored in the Game Instance. Due to the Game Instance being persistent between levels[4], it is well suited to hold the information necessary to load the character on the level in which the player is going to.

5.1.3 Configuration Menu

Due to time constraints, the configuration menu was not implemented fully. The idea was to have a collection of variables the player could adjust to suit their personal requirement and desires for quality versus performance.

As figure 9 depicts, there are 4 categories; Game settings, key-bindings, graphics, and audio. Game settings would contain all options regarding the gameplay. Options such as enable or disabling tutorial, show or hide floating combat numbers, show or hide nameplates would be accessible from the game settings tab.

Key-bindings would allow the player to set their own key-bindings to the different buttons on the action bar. During planning we had the idea that it would be beneficial

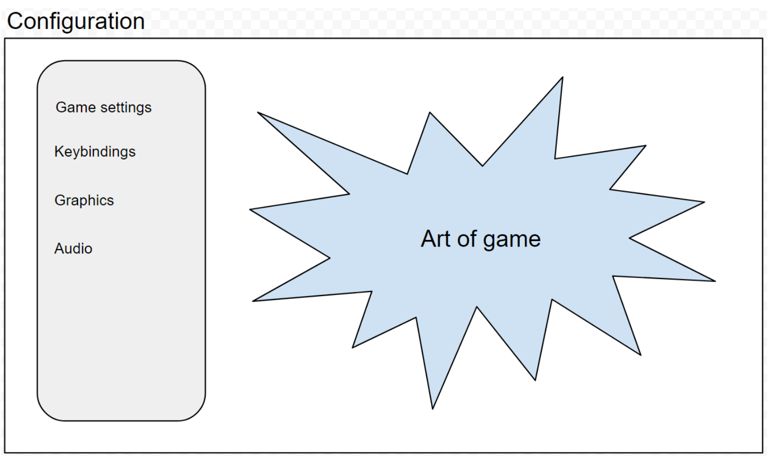


Figure 9: Configuration menu

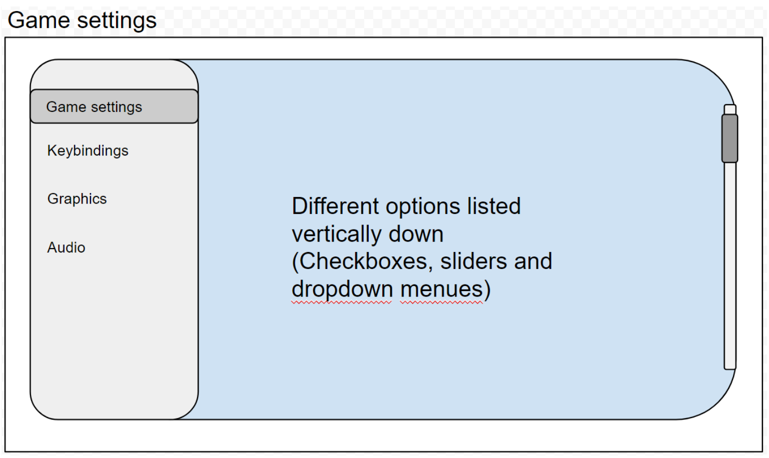

Game settings
Game settings
Keybindings
Graphics
Audio
Different options listed vertically down (Checkboxes, sliders and dropdown menues)

Figure 10: Game Settings

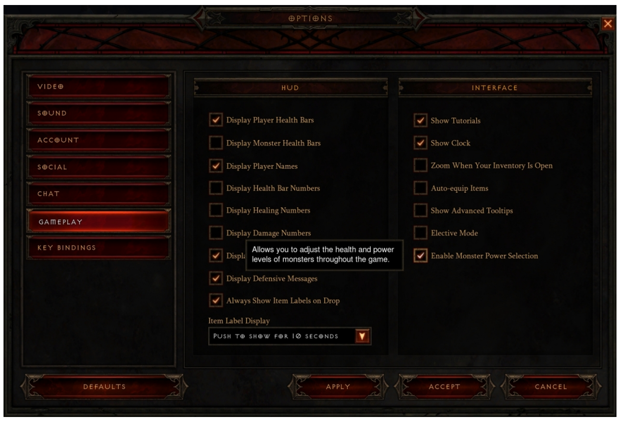
for the players if they were able to configure which keys on the keyboard they would use to cast spells.

Graphics would contain all options relevant to the rendering of the game. Resolution, texture quality, anti-aliasing, shadow quality and similar options would all be adjustable.

We wanted these options to be either drop-down menues or checkboxes. Drop-down menues for options that would require the user to specify an option and checkboxes for on/off options. this choice was inspired from Diablo 3's settings menu (see figure 11).

5.2 In-game User Interface

After having spent a lot of time creating the main menu, we started work on the in-game user interface. This time around we decided to split the UMG widget blueprints into smaller pieces in order to keep each blueprints simple and easier to work with. We began with creating a widget blueprint that would be used as our main blueprint. This blueprint is BP_HUD (head-up display). Anything that is to be displayed in the in-game menu would be constructed there and added to the view-port. By splitting up the widget blueprints we also gained the ability to mass-produce certain elements - making



- [✓] Display Player Health Bars
- [ ] Display Monster Health Bars
- [✓] Display Player Names
- [ ] Display Health Bar Numbers
- [ ] Display Healing Numbers
- [ ] Display Damage Numbers
- [✓] Displa
- [✓] Display Defensive Messages
- [✓] Always Show Item Labels on Drop

Item Label Display: PUSH TO SHOW FOR 10 SECONDS

Allows you to adjust the health and power levels of monsters throughout the game.

- [✓] Show Tutorials
- [✓] Show Clock
- [ ] Zoom When Your Inventory Is Open
- [ ] Auto-equip Items
- [ ] Show Advanced Tooltips
- [ ] Elective Mode
- [✓] Enable Monster Power Selection

Figure 11: Diablo 3: Options menu

the interface more scalable when new data is introduced. An example of this is when the player picks up an item from the ground, the inventory blueprint is notified that there is a new item in the players inventory and adds a new BP_InventoryItem with the data from the new item and adds it to the correct category.

In order to display these elements, we set up hotkeys which when pressed, toggles the visiblity of these menus. The hotkeys associated are:

- Inventory: I
- Skill window: K
- Talent tree: P
- Equipped items: C or U
- Quest log: J
- Development tool: Spacebar

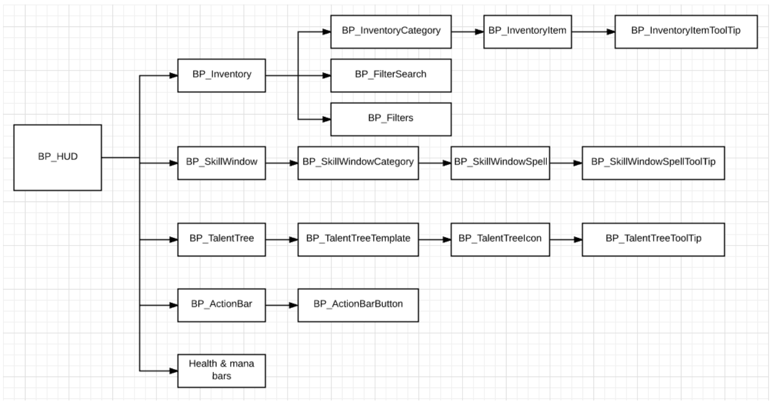


Figure 12: HUD components

5.2.1 Inventory

In the planning phase we discussed the inventory for hours, coming up with what we thought was missing from the game mechanic in other similar games. When playing similar games such as Diablo 3 or Path of Exile, we were always annoyed by having to play inventory-tetris to carry as much as possible and eventually go back to town to empty the inventory once it filled up. So with our game, we wanted to give unlimited inventory

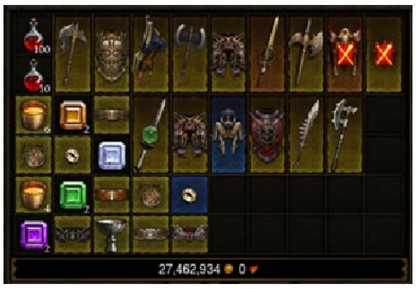


Figure 13: Diablo 3: Inventory

capacity a try. We would also couple unlimited inventory with an automated category system, sorting all items of the same type in a category (can be seen in figure 14). There are a few challenges to this system. There could be possible performance issues when players have collected 'too many' items and the inventory would begin to slow down the game due to the incredible amount of data. During implementation, there was only a few seconds of freeze when opening the inventory for the first time after loading the character with more than a thousand items in the inventory. After the first time opening the inventory, it would open smoothly without any freeze at all. This is due to the way Unreal Engine manages widgets. Widgets are not updated until they are shown. This is done to conserve processing power for processes that actually need them. However, in this example, when the inventory needs to be created, it is done immediately and will result in a short freeze. This freeze can be fixed by having a loading screen between the main menu and the level loaded.

The inventory consist of seven widget blueprints:

- **BP_Inventory**: The main widget holding all information and elements that are to be displayed in the inventory window. Any communications between other parts of the HUD are to go through BP_Inventory.
- **BP_InventoryCategory**: A special container holding all the items of the given type. The type is set when it is constructed by BP_Inventory.
- **BP_InventoryItem**: A one-to-one correlation to an item stored in the player's inventory component. Is unique to the GUID(Globally Unique Identifier)
- **BP_InventoryItemToolTip**: Tooltip used in Inventory item to display details about the item to the user.
- **BP_InventoryItemStatDisplay**: Contains two text boxes. Used in InventoryItem-ToolTip to display multiple stat values. the first text being the type of stat and the other the value.
- **BP_FilterSearch**: Part of the inventory filtering system where the player can search through their inventory for items with a name containing the input variable.
- **BP_Filters**: Part of the inventory filtering system where the player can specify to show only items with the given stat, rarity or type.

All the items the player current owns are stored in the inventory component. In this component there is an Array of TItems. TItems is a struct container with all information about a single item. Also in the component is a map of references to icons used to display items in the inventory.

Filters

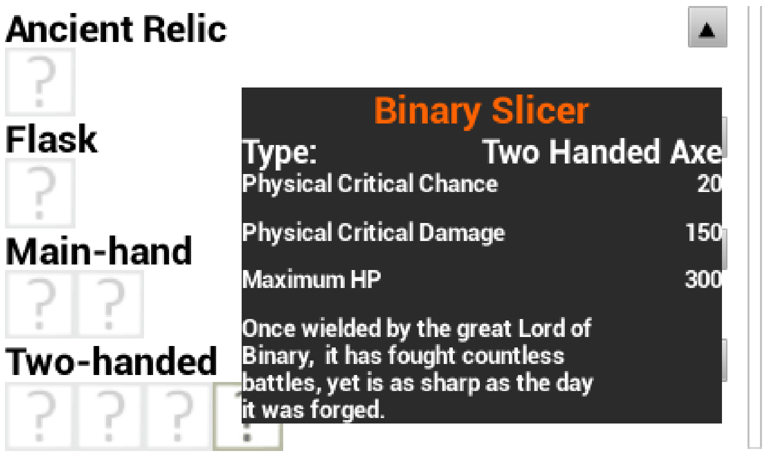
As mentioned earlier, due to the unlimited nature of our inventory, we wanted filters in the inventory system to allow players to find items they want with ease. There are two types of filter widgets implemented. The first, BP_Inventory is the main filter. This widget has an input field where the player can start typing in the name of the item. The filter is updated every time the text in this field changes - allowing the player to find what they are looking for fast. the second type of filter is BP_Filters. This filter has five different drop-down menus. The first one allows the player to adjust which type of filtering method to use. The methods available are: Item type, rarity, specific stat, and level. The player can add multiple of this filter to the inventory by pressing the + button. This will notify the inventory that the player desires additional filters. The inventory then checks how many filters it currently has. If there are less than 3 additional filters, it will spawn a new instance of BP_Filters. Subsequently, clicking the - button will remove the filter again. The player can also reset the filter by pressing the reset button. This will set the values of all the fields back to default. For more about the filtering system, see chapter 5.2.2.

Implementation Process: Inventory

We began creating the BP_Inventory widget blueprint. This blueprint contains a background, two vertical containers, and a few text boxes. These elements are necessary to create the shell for the inventory to be stored in. The first container holds all the filters and the other container holds all the inventory categories. In the constructor for the inventory, it first creates a filter of type BP_FilterSearch. Then, for every ETypeOfItem (an enum containing each type of item in the game), it creates a BP_InventoryCategory. Once all the categories have been created, it is time to fill the inventory. This is done by calling the 'Fill Inventory' function. In this function, the inventory get's a hold of the player character and requests the inventory component. Once it has access to the inventory component, it asks for all items in the inventory. Then, for each item in the inventory, it checks which type of item it is, then checks whether or not the category container already has added that item. This check is done using the GUID (Globally Unique Idenifier) and checking if the category already has an item with that GUID. If there is no item with that GUID, it creates a BP_InventoryItem and passes along the data. The inventory item then creates a BP_InventoryItemToolTip with the item data so that information is displayed to the user when hovering the mouse over the item (see figure 14).

The Inventory Category

The BP_InventoryCategory is a custom widget acting as a container with added functionality. If the player does not have any item of a certain type, the category container that holds all these items has its visibility set to collapsed. When the visibility is set to collapsed, it is hidden and does not reserve the space it would otherwise take up. It is also set to collapsed in the case where all children of this category are not visible either. This way it is also automatically hidden when all items it contains does not match any of the filter parameters (see section about the filtering system 5.2.2.) The container has a button, when clicked it shows/hides all InventoryItems it contains. (see figure 15).



Binary Slicer

Type: Two Handed Axe

Physical Critical Chance 20

Physical Critical Damage 150

Maximum HP 300

Once wielded by the great Lord of Binary, it has fought countless battles, yet is as sharp as the day it was forged.

Figure 14: Inventory item tooltip

Boots
Chest
Gloves

Figure 15: Inventory Category with chest category hiding its content

5.2.2 The Inventory Filter System

With our implementation of the inventory system, we thought it would be a great feature to be able to find specific pieces of items without having to look at every piece of item in the inventory before finding the item you are looking for. Naturally, a filtering system is what solves that challenge. In the constructor for BP_Inventory, a single instance of BP_FilterSearch is created and added to the filter-container. The player can add additional filters by clicking the + button. This is a different type of filter that allows the player to specify either Rarity, Type of stat, type of item, or items between X to Y required level. Once the player has three additional filters, the inventory will not add any more. We set this limit because being able to search by name, level, type, rarity, and stat with four different criteria should be sufficient in finding the desired item.

The 'filter inventory' function is called as soon as a variable in either of the filters change. Even if there is no valid selection, it will be called due to the function being bound to the combo-boxes(drop-down menus) on-change trigger. But in the case of an invalid search specification, that specific filter is ignored.

We implemented an array containing boolean values. these values are set to true or false depending on the values of the input fields from the multiple filters. If the BP_filtersearch has a string longer than 0 characters, the boolean in position 0 of the array is set to true. The remaining 4 positions (from 1 to 4) in the array are set if any of the BP_Filters has a valid input relevant to the given algorithm. An input is considered valid if a type of filtering method is set as well as a specific value from that enum is selected. Once the array contains atleast one true value, the inventory will commence the

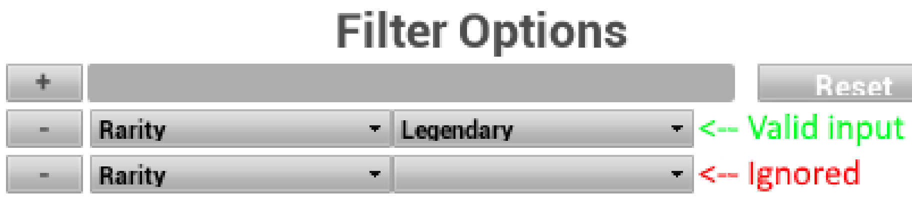


Figure 16: Valid and not valid filters

filtering. The first thing done is to go through each item in the inventory and setting their visibility to collapsed. As soon as all items are hidden, the process to find items matching the search begins. this is done with a nested for-loop. the first goes through each item in the inventory. then, the second for-loop goes through each value in the Array of booleans. If the value is true, it does one of five different actions depending on which position in the array.

- Position 0 = Checks if the name of the item contains the string input from the player.
- Position 1 = Checks if the item is the same type of item as specified in the filter.
- Position 2 = Checks if the rarity of the item matches the rarity specified.
- Position 3 = Checks if the item has each specific stat specified in the filter.
- Position 4 = Checks if the items level requirement is within the given range.

If an item matches all checks done, it is considered a match and will have its visibility set to 'Visible'. Once the for-loops are done, the values in the Array of booleans are reset to false. This marks the end of the filtering process.

5.2.3 Development Tools

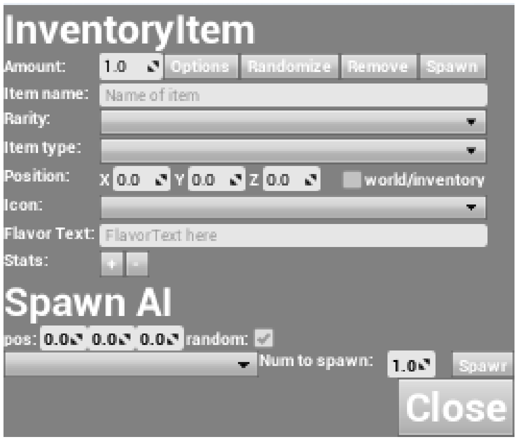
Early on we started to see the need for development tools in order to give the player items and spawn AI in real time. We took the time to make a proper tool where you can customize all the values for an item and spawn it in the inventory as well as spawning multiple AI pawns in the world. The development tool was intended to be extended with new functionality when it was necessary. There are two types of actions to perform; Spawn inventory item or spawn AI.

InventoryItem

In the section for spawning an inventory item there are input fields and drop-down menus for every variable. This allows us to create custom items during runtime instead of having to hard code them into our inventory system. The custom item is placed in the player characters inventory component. This tool was made before we had the functionality for spawning items in the world, but predicting this feature- we also added the option to spawn in the world at X/Y/Z coordinates.

AI

There are two options for spawning AI. the first is spawning an AI at a specific location in the world, and the other is to spawn multiple AI randomly in the world. The developer can also specify which type of enemy to spawn from the drop-down list.



InventoryItem

Amount: 1.0 Options Randomize Remove Spawn

Item name: Name of item

Rarity:

Item type:

Position: X 0.0 Y 0.0 Z 0.0 world/inventory

Icon:

Flavor Text: FlavorText here

Stats: + -

Spawn AI

pos: 0.0 0.0 0.0 random: ✓

Num to spawn: 1.0 Spawn

Close

Figure 17: Development Tools

5.2.4 Talent Tree System

During planning we had discussions about how to create an engaging and complex talent system without overwhelming the player. What we came up with was a series of separated talent trees that the player could choose from at any time. This meant that there would not be any character classes like most games in the genre. All talent trees would be available to any character. the idea was that all characters start out the same but after gaining experience, would differentiate themselves from other characters by the choices in talents they have made.

each talent tree would fit a certain theme, for instance, fire talents being thematically about casting fire spells and burning your foes. There are four different types of talents: Active, passive, augmentation, and sustained.

Active

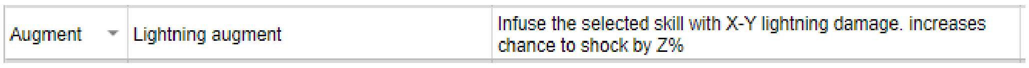
Active talents would grant the player a spell in their Spellbook. These spells could be offensive spells, defensive spells, or utility spells.

Passive

These talents are active at all times. Once points have been allocated in the talent, the effect of the talent would be applied immediately.

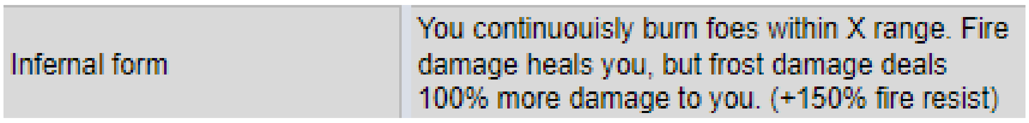
Augmentation

Augmentation is a type of talent that grants an effect when augmented on a spell the player has available. The idea was to allow players to augment spells with augments from different talent trees. Augments would drastically change the base ability, look and feel of the augmented spell. An example: Augmenting the spell Fireball with Lightning augmentation, figure 18. The augment would affect the fireball spell by changing the type of damage it deals to lightning damage as well as having a chance to apply the shocked status effect(increases lightning damage taken and attacks against the shocked pawn has a chance to stun for 0.5 seconds). The visuals of the spell would also change to reflect the new properties of the spell. This augmentation would allow players to change a spell to fit their build, even though the base spell is taken from a different talent tree. Augmentations can also have utility based effects, such as augmenting an ability to pull nearby enemies in, clustering the foes.



Augment	Lightning augment	Infuse the selected skill with X-Y lightning damage. increases chance to shock by Z%

Figure 18: Lightning Augmentation



Infernal form	You continuouisly burn foes within X range. Fire damage heals you, but frost damage deals 100% more damage to you. (+150% fire resist)

Figure 19: Example of a sustained spell

Sustained

Sustained spells are spells which reserve a specific statistic or resource the player has available. As seen in figure 19, the infernal form sustained ability alters the fire resistance of the player to such a high amount, that the player gains life instead of losing life from fire damage as well as burning any foes near the character. However, the price the player pays is to takes double damage from any source of frost damage. A sustained ability can also reserve a portion of the player characters health or mana, restricting the maximum availability of the resource.

Tiers

Every talent tree has five tiers, granting increasingly more powerful Talents than the tier preceding it. In order to claim talents from the higher tiers, the player must allocate talent points in the earlier tiers. The way this works is in order to be eligible for tier two talents, the player must first allocate a minimum of 5 talent points in tier one. For tier three, the player must allocate at least 10 talent points combined from tier one and two. It does not matter which tier these are allocated, as long as each tier requirement is met. for tier four and five the amount is increased to 15 and 20 points respectively. The player can also de-allocate talent points. The de-allocation also follow the rules of

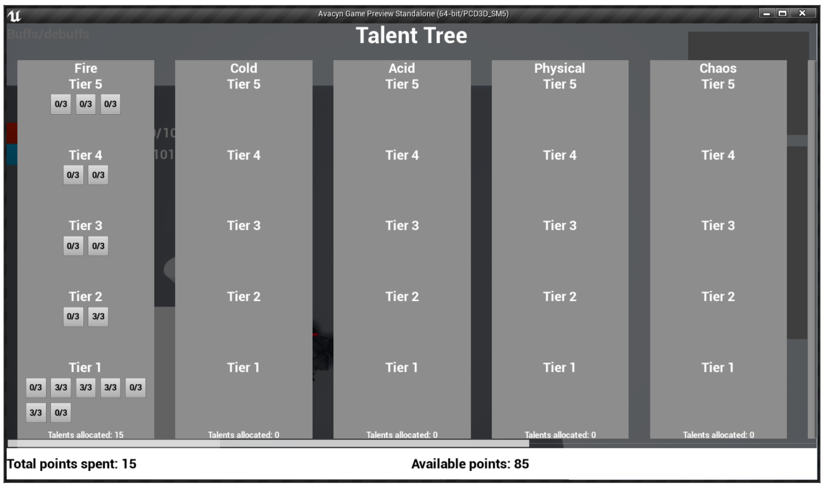


Figure 20: Talent Tree during implementation

the talent tree so that the player cannot cheat the rules. by cheating, we mean that the player for instance first allocates 5 points in tier one, then 5 points in tier two, then de-allocates all points in tier one in order to allocate them all back in tier two due to having unlocked it. This is not possible as the check to de-allocate goes through and checks if the de-allocation of the talent would break any of the constrictions. if it does, the action is discarded and the talent is not de-allocated. There are three ranks to each talent, increasing the potency of the talent for each rank allocated. The talent tree stores the points allocated in a component owned by the player character. This component is named 'TalentTree'. The talent tree component stores all information about all points allocated in the talent tree. This component is saved to the saveGame file so that the talents are restored upon re-entering the world with the same character. When the amount of points allocated in a talent changes, the talent clicked notifies the talent tree component by calling the function AllocatePointFromTalentTree, passing along three values: An enum value determining which talent tree, an enum determining which talent within that tree, and an uint8 with the amount of points allocated.

When a talent point is allocated, the talent tree notifies the skill window with how the new amount of allocated points. If the number of points allocated is greater than zero, the associated skill becomes visible.

Building the Talent Tree

The talent tree GUI is constructed using 4 different widget blueprints.

- BP_TalentTree
- BP_TalentTreeTemplate
- BP_TalentTreeIcon
- BP_TalentTreeIconToolTip

The talent tree is set up in a way that makes it highly scalable and easy to modify in the case of adding an additional talent tree. The way the talent tree construction works is that for each value in the enum ETypeOfTalentTree, an instance of BP_TalentTreeTemplate is created. Once it is created, the function 'ApplyTalentInfo' is called. This takes a parameter of type ETypeOfTalentTree. This variable is used to determine which talents are to be stored in this specific talent tree. This function goes through all talents and checks if they belong to this talent tree. If this is true, a widget blueprint of type BP_TalentTreeIcon is created. This widget is the talent itself. The data for the talent is passed along to the member function of BP_TalentTreeIcon called 'ApplyTalentData'. A pointer to the TalentInfo for that given talent is set as well as a reference to the parent. Then it creates a tooltip for each rank of the spell.

5.2.5 Skill Window

Looking at other published games in the genre - we saw potential to improve upon their implementations. Both Diablo 3 and Path of Exile have a clunky user interface (figure 21) and does not provide an overall view of which skills are available to the player. Wolcen however has a better implementation, more in line with the solution we decided to implement(figure 22). After our initial planning phase they have done modifications to their skill window, removing the category containers and instead have all spells of similar nature (Fire spells, bow skills, etc.) appear on the same row. The way we differentiate

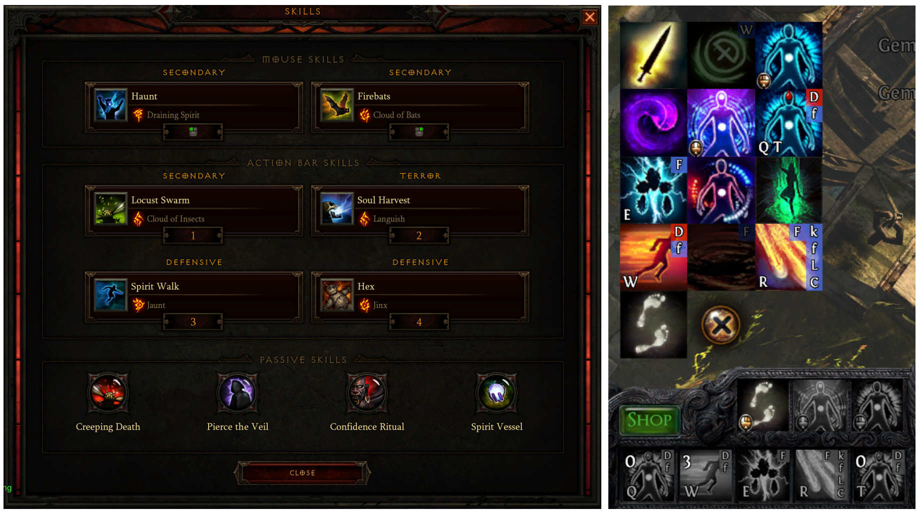


Figure 21: Skill overview: Diablo 3(Left), Path of Exile(Right)

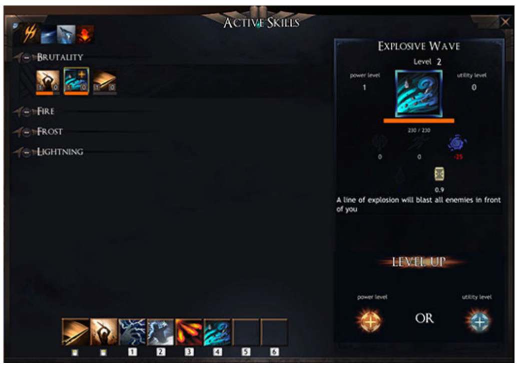


Figure 22: Wolcen:Lords of Mayhem Active skills interface

our implementation is by having the menu on the left side of the screen, and having the players drag the spells they want to the action bar.

The skill window consists of four widget blueprints, in a similar way to that of the talent tree. The four widget blueprints are:

- BP_SkillWindow
- BP_SkillWindowCategory
- BP_SkillWindowSpell
- BP_SkillWindowSpellToolTip

These blueprints are used as building blocks to construct a fully functional skill window allowing the player to get an overview of which skills and spells are available to use as well as dragging them to the desired slot on the action bar in order to be able to activate them. For more information about the action bar, see section 5.2.6.

Building the Skill Window

Much like the talent tree, the skill window begins by creating the categories by going through each value of ETypeOfTalentTree, creating an instance of BP_SkillWindowCategory.

In the constructor event for BP_SkillWindowCategory it goes through each talent and checking first if it belong to this talent tree, if so, it checks the variable TypeOfTalent, which is an enum consisting of all types of possible talent types. It checks if TypeOfTalent is of type Active or sustained. if so, it creates an instance of BP_SkillWindowSpell and passes along the spellInfo for all three ranks of the spell. Then the visibility of the newly created widget is set to collapsed as the skill window does not yet know whether or not there are any points allocated in order to grant the player access to the skill.

5.2.6 Actionbar

When planning how the action bar would function, we set a few requirements. Most games in the genre have a limit to how many spells and abilities the player can utilize at any one point. We decided against this because we wanted the decisions made in the talent tree to not be diminished as soon as the player gained a new ability and had to choose which to use. This would also allow us to create special spells which may only be good in certain situations. We took inspiration from World of Warcraft with how the actionbars look. We also had the idea that we would allow players to have multiple

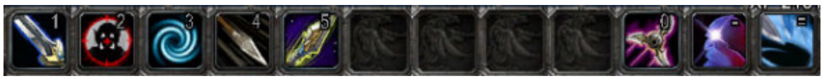


Figure 23: World of Warcraft action bar (default UI)

actionbars and could move and adjust them as they wished.

The actionbar consist of two widget blueprints: BP_Actionbar and BP_ActionbarButton. The actionbar being the entire bar and actionbarbutton being the ten specific buttons located on the actionbar. As seen in figure 24

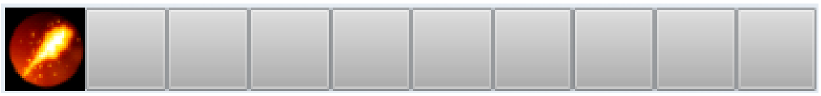


Figure 24: Early implementation of actionbar

5.2.7 Drag and Drop

Drag and drop was essential in order to place skills and consumables on the action bar in order to put them to use. During planning, this feature was not one we put much thought into. It was a feature we later on saw necessary. Our implementation of drag and drop functionality was inspired by a tutorial created by Epic games [5].

The current implementation does not include dragging inventory items to the actionbar. However, the system was planned with functionality for inventory items in mind, having some checks on whether or not the object dragged to the actionbar is that of a skill or an item from the inventory. Currently the checks only do something if the object comes from the skill window.

In order to take use of the already existing functionality for drag and drop in Unreal Engine, we needed to create our own 'drag and drop' operations, both inheriting base functionality from 'DragDropOperation'. Two blueprint classes were created: BP_SpellDragOperation and BP_InventoryItemDragOperation.

Then we overloadded default functions for the Button widget in both BP_SkillWindowSpell and BP_ActionbarButton. The functions overloaded are 'OnDrop', 'OnDragDetected', 'OnMouseButtonDown', and 'Event OnDragLeave'.

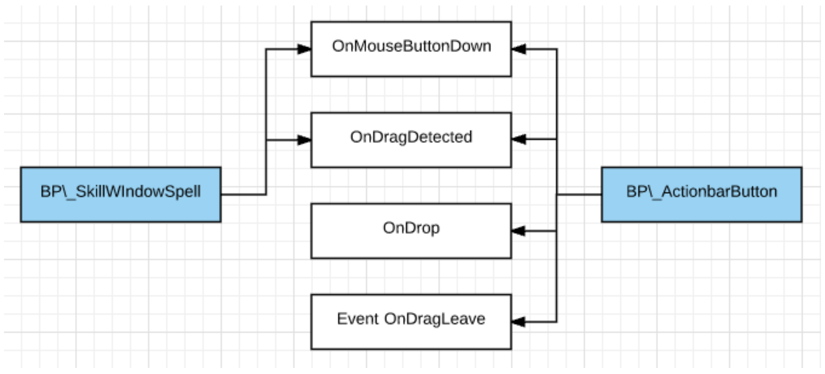


Figure 25: Showing which objects overload each function

OnMouseButtonDown

This function starts detecting drag if the correct key is clicked when dragging the mouse. SkillWindowSpell begins detecting drag when the key 'Left mouse button' is held down, and ActiobarButton on 'Right mouse button'. ActiobarButton uses the right mouse button instead of left because some players might want to click the spell with the mouse in order to cast.

OnDragDetected

As soon as drag is detected, an instance of the DragDropOperation is created. The DragDropOperation has to exposed variables that we set: Visuals and ObjectToDrag. Visuals is a reference to the object the drag and drop operation should look like. ObjectToDrag is a reference to the object that is being dragged, to be used when dropped on the actionbar.

OnDrop

When a DragDropOperation is dropped on top of the ActionbarButton, it checks which type of DragDropOperation it is by first trying to cast the object to BP_SkillWindowSpellOperation, if it succeeds, it continues by getting the reference to the object and casting the reference to BP_SkillWindowSpell. if this cast succeeds, some information is extracted and a few variables are set:

myStyle: the style used for the button in SkillWindowSpell and makes its own button look identical.

CurrentSpellToCast:an enum value determining which rank of the spell the player has access to and sets its own variable. This way when the key event assiciated to this specific actionbarButton is triggered, it knows which spell to cast.

bIsSpell: a boolean value used to determine which action to take when the associated key event is triggered. True being a spell, and false being inventory item.

The last thing done is to call the function 'AddActionbarReference' belonging to the SkillWindowSpell passing along a reference to self. The skill window spell stores that reference. This reference is used to update the actionbarButton when the points allocated in the associated talent changes.

Event OnDragLeave

This event takes care of removing actions from the actionbar after one has been placed. Once an actiobar button detects a drag operation leaving its area, it will check the drag operations tag. If the tag is equal to reset, it will call the function 'ResetSelf'. This function sets all relevant variables to default and sets the style back to the original look. The tag 'RESET' is only applied if the drag operation originates from an actionbarButton, leaving the skills and inventory untouched by this call.

5.3 Features not implemented

Due to limited time, we decided to cut some features of the GUI. We had a meeting midways through the implementation stage of development and discussed which features we needed to cut. The features were cut due to not being crucial aspects of gameplay. sections following describe how we envisioned the features not implemented.

5.3.1 Equipped Items and Stats

This feature is one in which we did not want to cut. If we managed to complete all tasks in the new estimation and had time to spare, this is the feature that was up next to be implemented. More about how we would implement the equipped items and stats in chapter 5.5.2.

5.3.2 Chat Window

Our plans for the chat window was to display communication between players as well as having a separate tab showing combat effects and values.

5.3.3 Menu Buttons

These simple buttons would allow the player to access the GUI elements with a click of a button in addition to the hotkeys associated.

5.3.4 Quest Log& Window

This window would display which quests and adventures were available to the player. The log would he a vertical list, meant as an overview. Clicking one of the items in said list would open up the quest window at the clicked quest displaying the quest in more details.

5.3.5 Mini-Map

The mini-map is located in the top right corner of the screen. The purpose of this mini-map was to display the world the player is located in as a 2D representation. This map would show icons for enemies, an outline of terrain, legendary and unique items.

5.3.6 Party Frames

As this game was designed with multiplayer in mind, a party frame is required. Party frames was cut as it did not make any sense implementing it before having network communication functional.

5.3.7 In-game Configuration Settings

Allowing the player to change configuration settings during gameplay is a quality of life implementation. This implementation would be to spawn an instance of the configuration menu used in main menu.

5.4 Enemy Health Bar

We decided to implement a system for displaying a health bar above the model of enemies. This serves as giving feedback to the player on how much of an effect their spell had. We implemented this system by creating and attaching a HUD component to the

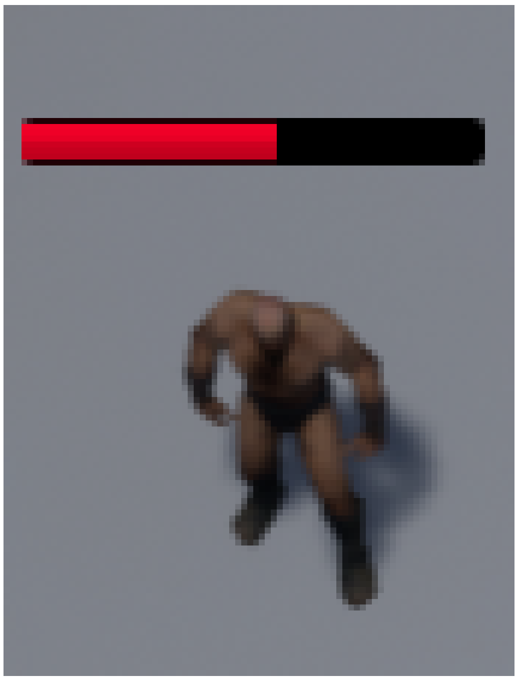


Figure 26: Enemy Health bar

enemy units. This component consists of a progress bar that is displayed in 2D just above the character model. The component has a reference to the statistics component of its parent. this reference is used to set up a binding for the progress bar. This binding extracts the current health and divides it by the maximum health. This will result in a value between 0 and 1 which is then used to update the progress bar.

5.5 Future Improvements

Although we feel we implemented the GUI elements in a good way, there could always be improvements. In this chapter, we explain which features and implementations we would improve upon, as well as how we would implement the changes.

5.5.1 Main Menu

As this was our first go at UMG widgets, there were a lot of mistakes and non-optimal implementation methods used. With the exception of the credits window, character in the character list, character model, and the world plane, every single element in the main menu is set up in a single blueprint. For future improvements, refactoring the main menu blueprints is number one on the list. This would be done similar to how the In-game menus were constructed. By extracting the elements into separate widget blueprints and create an instance of them in the constructor of the main menu. This way maintaining the code is easier as everything is not located in the same widget blueprint.

5.5.2 Equipped Items and Stats

This part of the inventory system was not implemented fully due to time constraints and other systems being prioritized. This system consists of a menu showing the items that are currently equipped. The way these items are represented correlates to which part of the body it makes sense to wear the item. Our implementation would look and feel much like other games in the genre (see figure 27). For stats we would list all stats contained

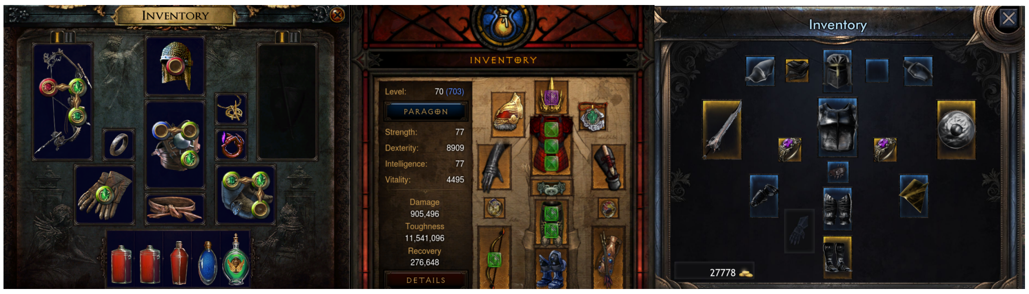


Figure 27: Equipped items implemented in simliar games (From left to right: Path of Exile, Diablo 3, and Volcen: Lords of Mayhem)

in the statistics component for the player character. This way the player can compare items and see what effect equipping the item has. Functionality for marking certain stats as favorites is also planned. This allows the player to quickly be able to see favorite stats by opening the menu, and not have to click the toggle stats button. We would also implement this part of the GUI in such a way that allows for it to be used in both the equipped items and stats window as well as in the character selection. Potencially also displaying the items currently equipped by the player.

5.5.3 Filtering

Even though the filtering system works well, we believe the method of filtering can be further optimized. For instance when filtering for a specific type of item, instead of going through each item, we could just hide the categories themselves. However, as the visibility of the categories is dependant on the items within them, some changes to the category blueprint would have to be done.

5.5.4 Drag and Drop

As we did not have time to implement drag and drop functionality for the inventory, this would be a good feature to add. This would allow the player to drag consumables to the action bar and use the hotkey in order to use them, instead of having to open the inventory, find the item, then click to consume. As combat can be hectic, this would be a great quality-of-life implementation. We would do this similarly to how drag and drop was implemented for skills.

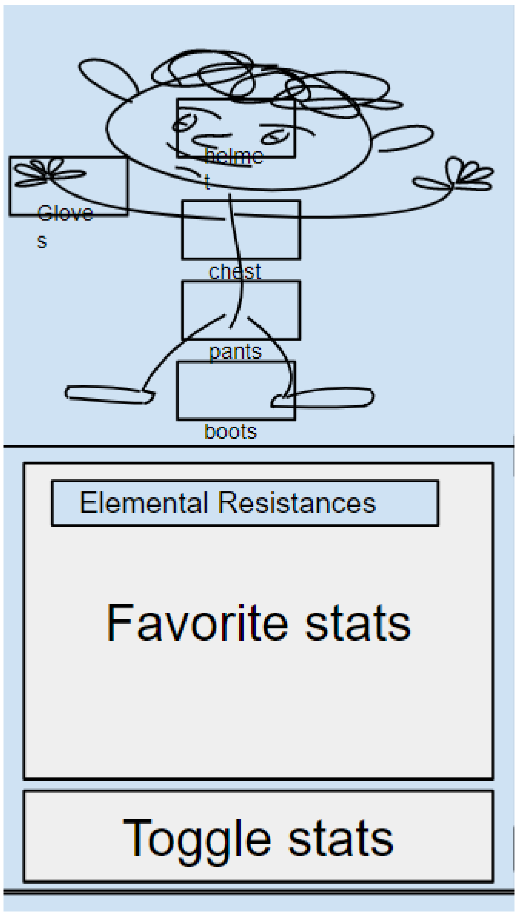


Figure 28: Mock-up for equipped items and stats

Another improvement to drag and drop is to take use of Tags better. By using tags, we can implement multiple actions to take for each different type of drag and drop operation. By using tags, we can implement general functionality for when a dragged object is dropped outside of any UI element. This would be used for dragging items out of the inventory and dropping them on the ground as well as dragging skills or inventory items from the actionbar outside the actionbar in order to remove them. Determined by the tag, one of the actions would be selected.

5.5.5 Damage Numbers

Another feature we had planned, but did not have time to implement is damage numbers. When casting a spell, and the projectile hits an enemy, the only visible feedback shown to the player is the health bar above the enemies dropping. We would like to implement a system for displaying damage numbers above the progress bar. This feature would help the player to see improvements to their damage and result in a sense of progression.

We would implement this system by having a GUI element with functionality for displaying numbers. Once an enemy is hit, it would calculate the damage taken, then notify this GUI element telling it where in the world it is located, and how much damage it received. The GUI element would then figure out where in the screen the character is located and display the damage taken. In the case where there are a lot of enemies and a lot of damage numbers to be displayed, the GUI element would try to not overlap the numbers, making them easier to read.

5.5.6 Inventory Item Tooltip

There is additional improvements we would wish to perform on the tooltips for items. Mainly having multiple tooltips for the various types of items. As weapons have base damage associated with them, it would be nice to display that information as well. Currenctly we only display a general tooltip that fits most other types of item.

6 Artificial Intelligence

As our game revolves heavily around the player killing monsters, a key point was to make this part of the gameplay as interesting as possible. Our Artificial Intelligence (AI) made this possible, by bringing the the monsters to life and having them try to kill the player. This creates a fun experience for the player where they have to kill their opponents while trying to stay alive using all the abilities they have at their disposal.

6.1 AI Requirements

What exactly do we need from our AI? The main point would be that at least all regular enemies use the same AI (we could make alterations to the AI for a special boss monster if needed). Other than that, it needs to:

- Choose between the abilities it has access to (Choice can be weighted based on variables in the environment, for example the distance to its target).
- Walk closer to its target when not in range to use the ability it selected to use.
- Be efficient, as there will be a lot of enemies on the playing field at the same time.

6.2 Choosing an Algorithm

Using these requirements we went about choosing what sort of AI algorithm to use for our project. The first, and most obvious option was to use the algorithm most well implemented in the Unreal Engine, namely a Behavior tree. This had several advantages from being well integrated in the engine, such as ease of debugging, tools to easier find performance issues for optimization, and clarity of what was going on in the AI during run time due to the engines neat GUI and the inherent structure of the behavior tree.

In addition to these advantages from already being implemented in the engine the behavior tree looked like a good candidate for our project due to how easy it would be to implement our simple AI with it, and how easy it would be to add new features later because of the modular nature of behavior trees.

Other algorithms that were considered are:

Genetic algorithms, neural networks, or other machine learning algorithms like NEAT (Neural Evolution of Augmenting Topologies, which is a combination of the two), but these kinds of algorithms didn't really fit too well with what we wanted to accomplish, as we didn't want the AI to discover the optimum strategies, but rather just use all the abilities we give it semi randomly. They would also require lots of extra time to implement and train/evolve.

Goal Oriented Action Planning (GOAP). The way this algorithm works is by giving the AI different actions, that have a cost, and change some state(s). Each action will have conditions/states that needs to be fulfilled in order to perform it, and performing it will change other states in turn. So by giving the AI an action to perform it will try to first achieve all the prerequisites needed to perform that action by performing other actions first. This could in theory work for our project, but as our goal would always be "kill the player character", and we don't necessarily want the AI to always cast one ability over the other to try and kill the player, this algorithm didn't seem like the best candidate either.

A Finite State Machine (FSM) was also considered, but we figured such a algorithm would scale poorly with our many spells and abilities, not be nearly as flexible as a behavior tree when it comes to altering the AI to work properly for special boss monsters and it would require at least as much work as using a behavior tree as the Unreal Engine isn't built with FSMs in mind.

Choice

With all this in mind we really didn't really see much reason to go for anything other than a behavior tree, and seeing as we wanted to learn more about behavior trees and how to program them as well this was a fairly easy choice in the end.

Looking back at it you could argue that very complex boss behavior would be easier to program using another algorithm, probably a variation of GOAP, but as we want fairly simple behavior for most enemies in the game, and even this complex boss could be programmed with the much more flexible behavior tree approach we felt we made the correct decision going with a behavior tree for our choice of AI algorithm.

6.3 Implementation

Our AI uses the behavior tree built into the engine by the Unreal team in combination with a blackboard. A blackboard is essentially a place the AI can store variables, either on a global basis for all behavior trees using that blackboard, or on an instance by instance basis.

The behavior tree is tree-like graph that is traversed by the AI. The tree contains service and task nodes. Service nodes are primarily there to do checks in the environment and set values in the blackboard based on these, as well as serve as a guide to how the AI should traverse the tree. Task nodes on the other hand are there to execute gameplay tasks for the AI controlled pawn, such as walking around or attacking. All of these nodes are programmed in C++ and put together in the behavior tree structure within the engine's GUI.

As the project stands at the time of writing this, the AI is fairly basic, and what it does can be boiled down to these simple steps:

- Check if there is a player character and store a reference to it in your blackboard (only for this instance of the AI)
- If you are not in range, or don't have line of sight to attack the player, move closer until you do.
- Attack the player.

Whether you are in range to your target or not is determined by checking against a variable in the blackboard that can be changed by other services in the behavior tree. The intention of this is to make sure the AI doesn't always move into melee range if it doesn't have to do so to use its selected ability, and rather just move within range to use the ability.

6.4 Improvements

With the AI in its current basic form the next step in improving it would be finishing up the planned features, namely; choosing from the set of abilities available to the character the AI is controlling, this being a weighted choice where the range of abilities and the range from you to the target would determine the weights. After the choice was made, set the new desired range to target in the blackboard, then use that to move within range and finally use the ability.

A diagram of the finished simple AI would look like Figure 29.

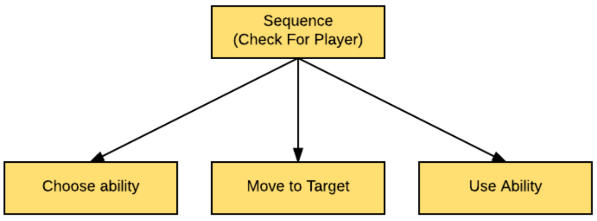


Figure 29: Simple Behavior Tree Diagram

Some improvements that could be done after this is to split the major tasks into multiple smaller tasks. For example by splitting the task that chooses what ability to use into smaller tasks like determining what abilities are available to the character, which you have enough resources to use, setting the weights of abilities based on the environment and so on. This would make it easier in the future to alter the behavior tree of certain enemies to make them more unique. For example making them act more randomly, more deterministic or perhaps making them more cowardice. Figure 30 shows an example of a behavior tree that is broken down into more modules where the AI also runs away from the player when it has low hp.

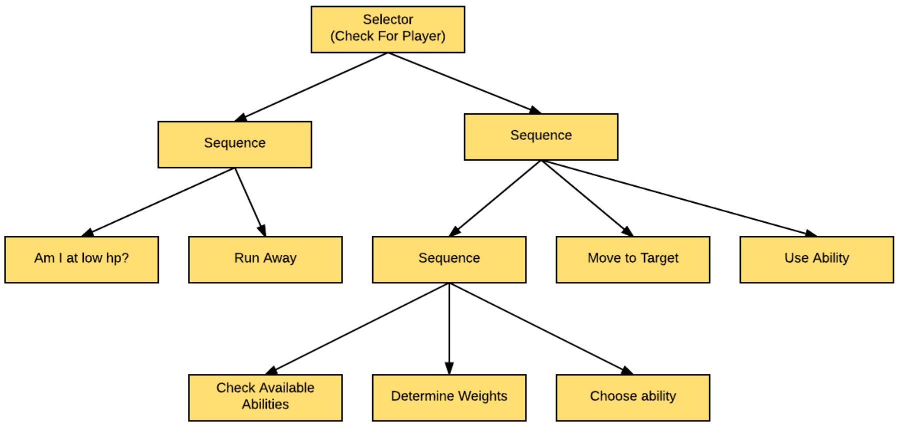


Figure 30: Expanded Behavior Tree Diagram

7 Spell System

Casting spells is a very integral part of our game, and to implement this in a nice way, where both the player and monsters can cast them, we created the spellcasting component.

7.1 Implementation

The system we have made for spells can be divided into three parts, these parts each perform their own separate tasks which in the end makes up a solid and flexible system for casting spells. The three parts are:

- The Spellcasting Component
- The spells
- The interfaces

7.1.1 The Spellcasting Component

The main job of the spellcasting component is to try to cast spells when its owner (the actor this component is attached to) asks it to cast a spell.

When an actor wants to cast a spell, be it a player or an AI, it would simply ask its spellcasting component to cast the spell with a function call that takes an enum and a location in world space as parameters. The enum is used to determine what spell to cast, and the location is the target location for the spell, depending on what spell is cast, the target location may be used differently. Examples of this could be; Fireball uses its owners location as the location to spawn the projectile, and the target location as the point to aim at. A flamestrike on the other hand would spawn the spell actor on the target location. There are also spells, most notably self buffs and heals, that simply ignore the target location.

Say an AI controlled monster wanted to cast the rank 1 version of the spell Fireball, then the function call would look something like this:

```
ABaseEnemy ControlledPawn = Cast<ABaseEnemy>(GetControlledPawn());
FVector TargetLocation = Enemy->GetActorLoaction();
ControlledPawn->CastSpell(ESpellName::Fireball_rank1, TargetLocation);
```

The other tasks the spellcasting component has are;

To keep track of the cooldown period (the time you must wait after casting a spell to cast the same spell again) of spells, and make sure you can't cast spells that are currently on cooldown.

To make sure the actor trying to cast a spell has enough resource to cast the spell, as well as making sure that resource gets spent when casting the spell.

And finally, to make sure the actor cannot cast spells outside the max range of a spell.

7.1.2 The Spells

All spells are derived from a base projectile that has functions to receive damage and behavior data on creation as well as functions for dealing damage to actors hit by it.

By having a base class that does these basic functions, we do not have to implement them in every single spell we make, and if we later find out we want some base function for several spells, we can simply code it once in the base spell.

The Base Projectile class is a very simple shell, allowing us to derive from it to create whatever spell we could think of that needs to exist in the world.

7.1.3 The Interfaces

The IDamageable and IHealable interfaces allows the spells to deal damage and heal any actor that has implemented these, the most basic version of these interfaces is implemented in the player character and the Base Enemy class.

Other than simply allowing spells to interact with characters in the game, they also make it possible to have enemies with special interactions, a good example of this we would use is having all skeletal enemies ignore bleed damage, as skeletons don't have any flesh to bleed from, as well as converting all holy healing they receive into damage, as they inherently are unholy creatures.

7.2 Player Input

In order to perform actions when the player press a key, we needed to do two things. the first was set up input bindings in the project settings. there we specified a key and a string. Once the specified key is pressed, Unreal Engine will broadcast the string. In order for our player controller to catch this string, we needed to create a binding for it as well. This binding specifies a function to be called when the given string is broadcast. Shown in figure 32, we listen for "Inventory" and want to do something as soon as the

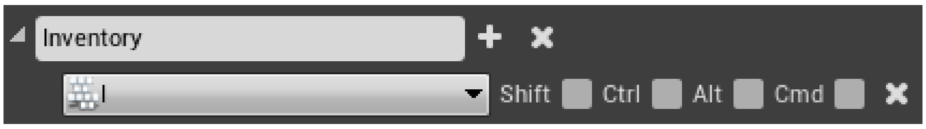

Inventory
I Shift Ctrl Alt Cmd

Figure 31: Project settings input binding

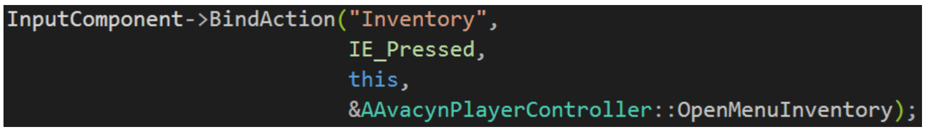

```
InputComponent->BindAction("Inventory",
                           IE_Pressed,
                           this,
                           &AAvacynPlayerController::OpenMenuInventory);
```

Figure 32: PlayerController binding

button is pressed. The action to be done is to call the function OpenMenuInventory. This function tells the HUD to show or hide the inventory window.

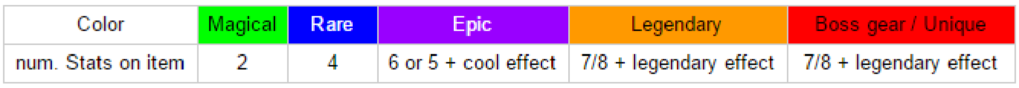
7.2.1 What could have been done better

When researching how to call blueprint functions from C++ code, we did not find any proper way of doing this. However, near the end of the project, we were informed of the functionality "CallFunctionByNameWithArguments"[6] to do this. This would make the interaction between the player controller and the various blueprints much better as the blueprints would no longer have to do a check every tick to see if there’s something that needs to be done, be it showing or hiding a menu, casting a spell from the actionbar, or adding a new item to the inventory.

8 Loot & Statistics

8.1 Loot

Loot that can drop in our game is divided into 5 categories as seen in Figure 33, where green quality gear is the weakest, orange quality gear is the strongest and red is a variation of orange quality gear with a predetermined set of stats and effect that is obtained from specific places or boss monsters. All items that drop in the game are magical in nature, either because they have been imbued by mages after their creation, or because they were made with magical items to begin with. Non-magical items are simply ignored by the character as they don't provide the same kind of benefits as magical items do.



Color	Magical	Rare	Epic	Legendary	Boss gear / Unique
num. Stats on item	2	4	6 or 5 + cool effect	7/8 + legendary effect	7/8 + legendary effect

Figure 33: Loot qualities

Types of Loot

There are twelve different types of items available in the game:

- Helmet
- Armor
- Legs
- Boots
- Gloves
- Ring
- Amulet
- Ancient Relic
- Flask
- Weapons
- Consumables
- Back
- Shoulders

8.2 Loot Drops

All monsters in Avacyn have a chance of dropping loot for the player when killed. To determine what drops we first look at the variables that tells us how many loot pieces a monster can drop followed by the chances of each loot piece dropping.

These values can change depending on what rarity the monster spawned in as, and what level it is. Each monster will have base values which can slightly increase with monster level as well as significantly increase if the monster spawned in as a rare monster.

After determining how many loot pieces a monster drops we look at at another separate weighted table to see what quality the gear will drop as and roll once for each drop.

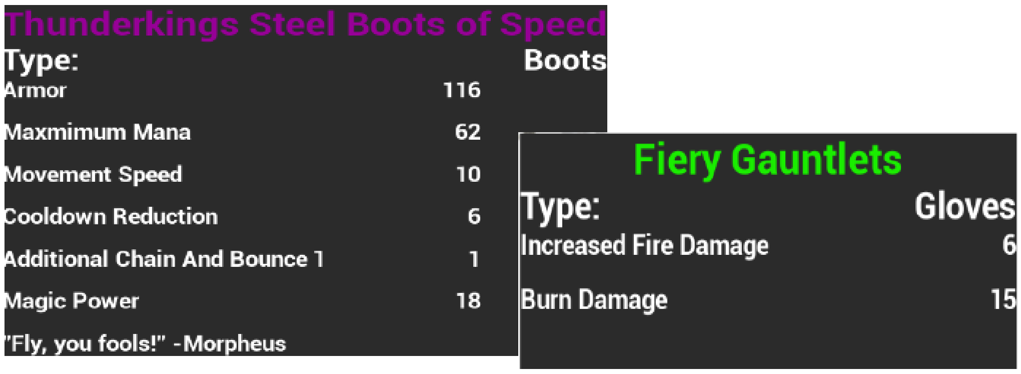
8.2.1 Loot Generation

To start the process we roll a random piece of loot from the game that is not currently restricted from dropping for the player because of factors like location, enemy type, level or other effects.

To determine the stats and name of an item during generation we have created a system where we have some categories of enums, each enum relates to some different stats and a string that will alter the name of the item generate. Depending on the category, the string that alters the items name will either be a prefix or a suffix, however, some categories might not alter the name at all.

Depending on the quality of the gear that is being generated a number of these categories are chosen, and an enum from each is randomly selected. We then put all the stats the enums relate to into a table, and randomly select as many of them as we need for the gear currently being generated. The name is also put together from the strings related to the enums previously chosen.

After what stats the item will have is decided, the amount of each stat is rolled, with the level of the item determining a range for how much of each stat there can be on the the item. Examples of generated items can be seen in Figure 34.



Thunderkings Steel Boots of Speed

Type:	Boots
Armor	116
Maxmimum Mana	62
Movement Speed	10
Cooldown Reduction	6
Additional Chain And Bounce 1	1
Magic Power	18

"Fly, you fools!" -Morpheus

Fiery Gauntlets

Type:	Gloves
Increased Fire Damage	6
Burn Damage	15

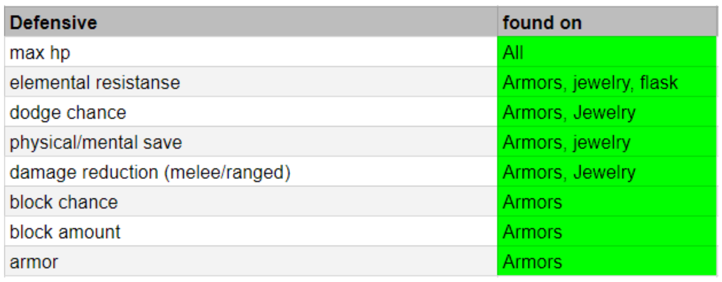
Figure 34: Examples of generated items

8.2.2 Stat Distribution on Items

We have implemented 108 different stats that can appear on items (chapter 8.3). We chose to restrict which stats can appear on items instead of tuning the values of a stat down enough to allow for it to be on all pieces without achieving incredibly high values, such as reaching 100% critical hit chance. Allowing a stat to appear on all items would in some cases dilute the value of a stat compared to other available stats. Instead, we restrict some stats to only appear on a certain category of items. As we do not want to restrict the player too much, these categories are broad and cover several items. The categories are:

- Armors = chest, boots, gloves, shoulder, legs, back, helmet
- All = all types except ancient relic and flask

- Jewelry = Rings, amulet
- flask
- ancient relic



Defensive	found on
max hp	All
elemental resistanse	Armors, jewelry, flask
dodge chance	Armors, Jewelry
physical/mental save	Armors, jewelry
damage reduction (melee/ranged)	Armors, Jewelry
block chance	Armors
block amount	Armors
armor	Armors

Figure 35: Defensive stat - gear distribution

When we discussed which pieces were to have certain stats, we were careful not to create specific item archetypes. By archetypes, we mean that all associated stats have a similar nature. An example would be the way Diablo 3 did their items: Chest armors can only have defensive properties, and no offensive. This would restrict the choices available to the player and hurt build diversity. We want the player to be able to have anything from full offensive to full defensive stats on all pieces of gear. This shifts the responsibility of handling the balance for power output and survivability to the player.

8.2.3 What we want our loot system to accomplish

There are two main ways for a character to get stronger in our game. The first is through leveling up and putting points in the talent tree to gain a passive increase in stats, the second is through obtaining better gear.

Through the early levels you will level up fairly fast, and your character will quickly be able to put more points in the talent tree and have these scale up with their level, also the new gear obtained through drops will quickly increase in power in relation to what dropped a few levels earlier, letting the character fairly easily find gear that is an upgrade due to the increased amount of each stat on the new gear piece.

As you progress further into the game it will start to take a little longer between each level, and for that reason you will have to kill more enemies before you can level up again. But we don't want the players character to stop getting stronger altogether, and that is where our loot system comes into play! As you get to the higher levels and stay at levels for extended periods of time, you will have a greater and greater chance of finding higher quality loot for that level before you move on to the next levels, so instead of gaining most of your stats from leveling up, where you get your stats from will shift more and more towards getting higher quality gear as you level up, and even further on getting high quality gear pieces with stats that are especially good for your talent choices become important to your characters strength.

One thing that is important to think about when designing a system like this is to not have obtaining the gear pieces you need at your current level to be too slow. With the amount of different stats items in our game can have, finding the right piece for your character in the later levels might become very difficult, which is why we decided to have higher level monsters overall drop more items, giving the user a higher chance of getting the loot they need. Having more loot drop is also one of the main drives behind the improvements we have decided to make to the inventory system over other similar titles. See Chapter 5.2.1.

8.3 Statistics

8.3.1 All Stats

As most games in the genre, we envision the core gameplay to be about creating interesting talent builds and finding items which compliment said builds. To make a loot system diverse, we needed to implement a variety of different stats. 108 unique values together form a complex system for enhancing the character. These stats are gained by equipping items as well as gained through passive talents from the talent tree. Some stats are defensive, offensive or utility based. By gaining multiple of these stats, synergies will emerge to empower the character. We planned our stat system to compliment the talent system by enhancing the effects of the talents the player gains. Talents that does something when the player lands a critical hit become increasingly more powerful as the players critical hit chance increases. Every stat has an associated entry in the Enum ETypeOfStat. These entries can be classified in one of the following categories:

- RES - Resources, maximum health, etc.
- DEF - Reduce damage taken
- LO - Focuses on gaining **L**ife **O**n hit, dodge, block etc.
- UTIL - Enhances utility characteristics, movement speed etc.
- POT - Affect potions.
- MINI - Empowers the minions owned by the actor.
- OFF - Increases damage output.
- OFFM - Affects magical damage.
- OFFP - Affects physical damage.
- DI - Increase the severity of a damage type.
- DP - Ignores a portion of the resistance for a specific type of damage.
- DIW - Damage increased while wielding a certain type of weapon.
- DR - Reduces the severity of a damage type taken.

Each entry has a prefix attached, identifying which of these categories they belong to. These categories helped us balance the amount of offensive versus defensive types of stats exist in the statistics system.

As there are more than 108 different stats, its too much discuss each individually. If desired, the stats can be found under Enums in the appendix D. The enum is named ETypeOfStat, and each value has a comment above it briefly stating what the value represents and how the value should be used.

8.3.2 Statistics Component

The statistics component is used to store all information mentioned in the previous chapter. These values are continuously modified by items equipped, talents, and status effects. Each Actor in the game world has its own statistics component.

The component consists of the following member variables:

```
// Map containing all the stats the character gets from gear
TMap<ETypeOfStat, float> StatMap;
```

```

// character
uint16 Level;

// character / player spesifics
uint32 CurrentExp;

// character / NPC spesifics
uint32 ExpRewardOnKill;

// if dual wielding
bool bIsDualWielding;

// is classified as elite
bool bIsElite;

// Map containing pointers to different functions
// to handle different types of damage
TMap<ETypeOfDamage,
     void (*)(ETypeOfDamage,
     FProjectileDamageData,
     UStatistics*)> HandleDamageMap;

// Helper map to map a type of damage to the resistance
// for that type of damage
TMap<ETypeOfDamage, ETypeOfStat> DamageToResistanceMap;

// Helper map to map a level to an armor value modifier
TMap<float, float> LevelToArmorMultiplierMap;
```

StatMap

This map is where all the possible stats gained from talents and items are stored. All these stats are stored as floating point values due to our decision to store all values in a single Map. Maps do not support different types of variables as value. Because of this, we decided on floating point values as some stats represent percent chance, ranging from 0 to 1.

Level

Level is a way to create character progression. As the player gains a a new level, it is rewarded with a new talent point as well as increased maximum health. The higher level a character is, the more talent points are available.

CurrentExp

CurrentExp stores the information about how far the character has progressed on the current level. Once the character reach a new level, the required experience for the new level is subtracted from the currentExp variable.

ExpRewardOnKill

This variable indicates how much experience the player should be rewarded when the enemy dies.

bIsDualWielding

As there are stats that depend on whether or not the player is dual wielding one handed weapons, we have implemented a boolean value that is set to true when the player equips two one handed weapons, and false when this is no longer the case.

bIsElite

This variable determines whether or not stats such as EliteDamage should be used in the calculation of damage done to this enemy.

HandleDamageMap

This map is used to map all our different types of damage to a function pointer. The function that is pointed to determines how calculation of the damage type in question should be handled.

DamageToResistanceMap

This map is used to map all our different types of damage to the corresponding resistance type. For example; ETypeOfDamage::Fire would map to ETypeOfStat::FireResistance

LevelToArmorMultiplierMap

This map is used to check how much physical damage reduction an actor should have against an attack depending on its armor and the level of the attacker.

9 Vulkan and Dynamic Destruction

Vulkan[7] is a state of the art, cross-platform graphics and compute API that was released early in 2016. It is intended to become an industry standard, and an eventual replacement to OpenGL. Vulkan is designed to take advantage of modern hardware that older APIs could not utilize very well; it also has support for computing pipelines removing the need for a dedicated compute API. Vulkan has much lower CPU overhead compared to earlier graphics APIs allowing there to be a lot more calculations performed on the CPU. Vulkan is also made with support for multi-threaded CPUs, letting developers take advantage of modern hardware to a much greater degree than OpenGL would allow. Vulkan is a low level API, meaning developers will have a much greater degree of control over their applications compared to some other APIs. This means there will be a heavier load on developers when starting Vulkan development, but it also allows for greater control when optimizing for the individual needs of the project. The nature of Vulkan is to give developers a greater control over how their software utilize resources. Being a fully cross-platform API there are no restrictions as to what OS applications using it will be able to run on; it is supported on everything from windows to android; there are even unofficial ways to use it on iOS and macOS. This allows developers to make their applications available on multiple platforms while doing less customization for each platform.

9.1 Shader

Shaders are how the graphics API send commands to the graphics pipeline. They are small programs written in a way the graphics card can interpret to customize the way it processes data sent to it by the application. Vulkan uses an intermediate language called SPIR-V to take the shaders written in OpenGL shader language(GLSL) and compile them into a binary shader that the GPU can use. The way it is done with other APIs requires the graphics driver to include a compiler for the shader language, which adds both to the size and complexity of the drivers, and the need for shaders to be compiled at runtime each time. The feature level of the shaders supported by Vulkan is currently GLSL version 4.50. This means that Vulkan supports these shader types:

- Vertex shader (Required): Handles processing of individual vertices passed to the GPU.
- Tessellation control shader (Optional): Manages the amount of tessellation to be used by the tessellation evaluation shader.
- Tessellation evaluation shader (Optional): Takes input from the tessellation control shader, and combines it with vertex data from the vertex shader to interpolate additional geometry.
- Geometry shader (Optional): Takes one input primitive, and can output an arbitrary amount of primitives. The main reason to use it is to render the same thing to multiple targets. Not commonly used in modern applications.

- Fragment shader (Technically optional): Calculates color and depth used to finalize the pixels that gets shown on screen.
- Compute shader (Optional): Can be used for general computation. Not part of the graphics pipeline, but can share variables with it when used in Vulkan.

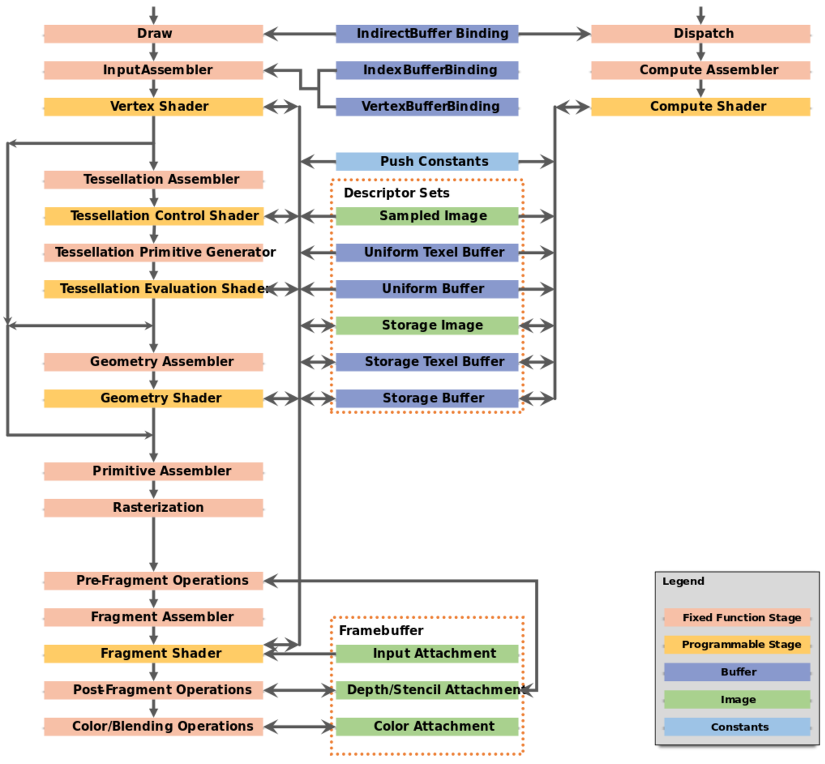


Figure 36: Vulkan Pipeline showing shared variables between Graphics and compute

9.2 Why Vulkan

Vulkan is a state of the art graphics API on the way to become common in the games industry. It is a low level, low overhead API designed to get the developers closer to the hardware. It also gives the possibility to increase performance as Vulkan is designed to support multi-threading and native compute pipelines. Vulkan provides much greater control over the graphics pipeline than older APIs allowing developers to turn of parts they don't need, and customize the ones they use to a much greater degree. Another advantage of the new pipeline model is that the graphics pipeline and the compute pipelien can access shared variables, so you can use the results from the compute pipeline for rendering wihout passing it via the CPU. Working with Vulkan for this project is a great opportunity for learning to use new technology in a real world application. Learning Vulkan will be very useful as professional game developers including some of the biggest publishers have already started adapting it.

9.3 Unreal Engine and Vulkan - a story of love, loss, and despair

Since we chose to use Vulkan for our graphics, and because we had chosen Unreal Engine as our game engine this would not be a massive issue. The source code for Unreal Engine is freely available to its users to do as they want. The engine also had some basic support for Vulkan already implemented with plans to add full support in the future. The first while of the project was used getting familiar with the source code of Unreal Engine, this was a big endeavor as the source code is extensive. After getting comfortable with it work began to implement the Vulkan functionality required by us. I got Vulkan to a functional state but there were some issues with the interfaces between the different parts of the engine preventing us from getting all the functionality we needed. When Unreal Engine came with an update where they claimed to fix their Vulkan support we updated our development environment to the new version, the update fixed some issues, but introduced a bug that broke the engine when using Vulkan. We would have gone back to the old version if it was not for other very useful features that made other parts of the development a lot simpler. We decided to develop the dynamic destruction system separately, and implement it into our game when Vulkan is fully supported by Unreal Engine. During the process of fully implementing Vulkan with Unreal Engine I learned a lot about how the engine it self was built. I spent a lot of time going through the source code to add the parts that was missing for Vulkan to work, so I became familiar with large parts of the rendering part of the engine. When we updated to the version of Unreal Engine that was supposed to have improved Vulkan support, it turned out there was an issue with memory management related to multi-threading in the interface between different parts of the engine. I was unable to locate the problem as it came from a part of the source code I had not gotten familiar with. This led to large amount of time having been wasted.

9.4 Dynamic Destruction

One of the advantages of developing with Vulkan is the inclusion of a proper compute pipeline, allowing us to use the GPU rather than the CPU to perform calculations like physics and collision.

We wanted to utilize the compute capabilities of Vulkan to implement a system for dynamically breaking down complex 3D models in to smaller pieces in real-time. The way this is usually done is by dividing the models beforehand, and replace the larger model with the smaller ones when it is destroyed. Computationally this is a cheap way to do destruction as it does not need to calculate anything extra, and the cost will only be to add a few more objects for physic calculations. Doing dynamic destruction calculations in real-time on the CPU is just not feasible. The GPU on the other hand is excellent at performing large amounts of calculations in parallel. Being able to push the work to the GPU allows us to perform enough calculations to make true dynamic destruction a possibility.

9.4.1 Methods

There are multiple methods ways to do dynamic destruction that each have its own advantages and disadvantages.

Prefabrication

The most common way to make objects destructible is to divide the 3D-models into smaller chunks before they are put into the game, and then replacing the whole model with its pre-divided chunks when the destruction occurs.

Advantages:

- Computationally cheap.

Disadvantages:

- Does not reflect impact location.
- Requires artists to divide each model manually.

Voxel based modeling

A method that has become very popular in recent years is to build the entire game out of cubes. Using cubes to build all you models means that when it gets destroyed it is as simple as disconnecting already separate pieces. This gives it a very predictable and minimal performance impact.

Advantages:

- Computationally cheap.
- Simple for artists to create models.
- Supports actual dynamic destruction.

Disadvantages:

- Gives a very particular art style that may not suit all games.
- Unnatural and cubic destruction patterns.

Constructive solid geometry - CSG

This method is not used that much in professional game development, but it is used in some built in editors in games and game engines due to its ease of use. It has been used in some games to allow for gameplay involving cutting models based on planar intersections, but it is unsuited for fracturing.

Advantages:

- Can dynamically divide models.
- Planar cutting is simple.

Disadvantages:

- Can't be used on complex polygonal models.
- Using CSG for anything more than cutting is not feasible.

Volumetric Approximate Convex Decomposition

This method is not used in games today as it was first proposed in this form in this article[8]. It is a method that combines approximate convex decomposition and approximate convex hulls with fracture patterns to make real-time dynamic destruction on a large scale feasible.

Advantages:

- Accurate representation of impact location.
- Truly dynamic.
- Negligible computation cost when performed on a low number of objects simultaneously.

Disadvantages:

- Higher computation cost than other methods.
- Requires a mesh separate from the visual mesh for complex models.

We decided to work with the volumetric approximate convex decomposition for our project as that would give the best visual effect out of the methods presented above. This method is viable due to the power of modern hardware, and advances in graphics technology that allows for better multi-threading support.

9.4.2 Volumetric Approximate Convex Decomposition

A very new method for doing dynamic destruction. Gives a very accurate representation of the actual impact that caused the destruction High fidelity compared to other methods used today

Volumetric approximate convex decomposition uses Voronoi decomposition to create a compound mesh consisting of convexes closely matching the visual mesh of the object in question. Voronoi decomposition divides an area based on the distance between points placed on the model. It groups everything that is closer to one point than to another point into a convex, and the boundry where the distance from two points is equal becomes the dividing line between convexes. To get the convexes as close to the model as possible a convex hull is created, but for complex models this would contain far too many vertices and faces so an approximate convex hull is used to make a representation that is good enough for this purpose. When we have the compound mesh it can be used for the destruction. The basic idea of the fracturing of this method is the use of a fracture pattern where the point of impact is located at the origin, and the pattern is aligned so the origin overlap with the actual impact. We can now check the intersection between every compound and every cell in the fracture pattern, and then replace the existing compounds with new compounds that match the fracture pattern. Some optimizations are applied to reduce impact on frame rate e.g. combining convexes where multiple convexes fill an entire cell, finding if convexes were created from pieces that are not actually connected, and separating them. We can now go through the visual mesh and perform clipping checks between the visual mesh and the convexes we have created. If a convex intersects the visual mesh, new vertices are created and geometry is generated for the new faces. After the new mesh is generated, we recalculate the convexes so that they will fit the new visual mesh closely.

10 Deployment

We will be deploying our project as a self-contained package generated by Unreal Engine. For now we are only deploying for testing purposes, and will be optimizing the process if we decide to launch for market. Our deployment methods would have to change for launch, as the game would then support multiplayer feature necessitating online infrastructure. The end goal would be to deploy our game through a platform like steam to reach as many potential customers as possible.

10.1 Repository

Due to going over the size limit for the first repository, we decided to continue work from a new repository in order to keep commit history. Source files can be found at the following repositories:

link to the first repository: https://bitbucket.org/gtl-hig/avacyn

link to the second repository: https://bitbucket.org/gtl-hig/avacyn_part2

link to build: https://www.dropbox.com/sh/q4yf96whx00w1wr/AABgYGynIcCKHKtxS_gLXMpba?dl=0

There is no repository containing the modified source code for Unreal Engine, as it is far to large to use any repository solutions available to us.

11 Discussion

11.1 Visual Assist

Visual assist is a replacement tool for Microsoft IntelliSense better suited for handling large projects. It is a massive improvement over IntelliSense, as it is both a lot faster, and comes with a lot more features. When working with Unreal Engine source code Visual assist was invaluable, as IntelliSense was useless when working in such a massive project.

11.2 Unreal Engine as a Development Environment

For the most part, developing with Unreal Engine was great. Except for when we had syntax errors in the C++ code, resulting in the engine crashing. This wasted a lot of time as it would occur somewhat frequent, and re-launching the engine could take some time. One of the issues we had with Unreal Engine was working with Vulkan, as Vulkan is very new technology there are a lot of changes being made to support it. When we had managed to get it somewhat working there came an update to Unreal Engine where they had fixed a lot of issues with their implementation of it, but they also introduced an issue that made it impossible to actually run the engine using Vulkan.

11.3 Information and Knowledge Gathering

We knew going into this project that we would have to read/watch tutorials online in order to learn how to work with Unreal Engine. We were skeptical to tutorials, as we knew that community created tutorials can range from poor to excellent coding quality. We would assess these tutorials, determining whether or not they met our expectation of code quality. If they did, we would put them to use. However, most of the information we gathered from tutorials was merely to learn how things are done with Unreal Engine.

A high portion of the knowledge we gathered were extracted from reading the documentation of the various systems in Unreal, in order to know how to use them.

11.4 Bitbucket Issues

Early on in the implementation stage of the project, we accidentally pushed all of our assets to the repository. As a result, the size of the repository ended up at 2.5GB. This surpassed the limitations bitbucket has of 2GB. Due to this, we had to undo these commits. It turned out that we had pushed the starter content assets that come with unreal very early in the development process. As we wanted to preserve our commit history, we decided not to reset the repository, but instead create a new one and continue the work from there. Both repositories are linked to JIRA.

11.5 GUI

We initially thought developing the GUI would not be as much work as it turned out to be. in our initial implementation sprint (week 2 - 4), we estimated the GUI to be completed.

This sprint was extended and edited due to implementations not being complete. We had never made any proper GUI elements before - and it shows in our sprints.

11.6 The Result of our Development

Even though we were overly ambitious with the amount of work we planned, we can not help but to feel a little disappointed, not implementing every feature we had planned. That being said, we are still very satisfied with the features we implemented as they resulted in being both interesting game mechanics and implementation challenges.

As to whether or not we will continue working on the project, is uncertain. We feel what we have created so far can serve as a great base to actually finishing the game as a business project, but this would require quite some time and the help of graphical artists.

12 Conclusion

We had a clear idea of what we wanted the game to be during the design stage. A lot of the design decisions made, can be used for further development. We spent more time on game design than was needed for this thesis, but if we decide to take this project further, having designed the game to the degree we did will be very useful.

Our estimation of how long tasks would take to implement were not as accurate as we would have preferred. This was due to lack of experience in many of the areas we worked on.

A lot of time was spent modifying the Unreal Engine to fully support Vulkan. Even though this could not be used in the result of our project, it was an excellent way to learn how a modern game engine is built.

Further development would consist of making improvements on systems we implemented for optimization purposes. We would also implement networking and features requiring internet access in order to support multiplayer gameplay.

We consider the project to be a success, as we achieved what we set out to do. The systems we implemented are robust and scalable, so they require no extra work to support additional features and content.

Bibliography

[1] *Hardware & Software Specifications*. (Visited May. 2017). URL: https://docs.unrealengine.com/latest/INT/GettingStarted/RecommendedSpecifications/.

[2] Games, E. *Documentation page: UDataTable*. URL: https://docs.unrealengine.com/latest/INT/Gameplay/DataDriven/.

[3] *Coding Standard*. (Visited May. 2017). URL: https://docs.unrealengine.com/latest/INT/Programming/Development/CodingStandard/.

[4] *Game Instance and persistent data*. (Visited May. 2017). URL: https://wiki.unrealengine.com/Game_Instance,_Custom_Game_Instance_For_Inter-Level_Persistent_Data_Storage.

[5] Games, E. Drag & drop with umg series. URL: https://www.youtube.com/watch?v=wyC5vl64V9k.

[6] Games, E. *CallFunctionByNameWithArguments*. URL: https://docs.unrealengine.com/latest/INT/API/Runtime/CoreUObject/UObject/UObject/CallFunctionByNameWithArguments/index.html.

[7] Group, K. *Documentation page: Vulkan API Reference*. URL: https://www.khronos.org/registry/vulkan/specs/1.0/apispec.html.

[8] Kim, M. M. N. C. T.-Y. 2013. Real time dynamic fracture with volumetric approximate convex decompositions. URL: http://matthias-mueller-fischer.ch/publications/fractureSG2013.pdf.

[9] *Scrum*. (Visited May. 2017). URL: https://en.wikipedia.org/w/index.php?title=Scrum_(software_development)&oldid=780412457.

A Terminology

Scrum - An agile development method, see bibliography link [9]

CPU - Central Processing Unit.

GPU - Graphics Processing Unit.

RAM - Random Access Memory.

FPS - Frames Per Second.

Player - Refers to the person playing the game.

Player Character - Refers to the actor controlled by the player during play.

AI - Artificial Intelligence.

The Epic team - The creators of the Unreal Engine.

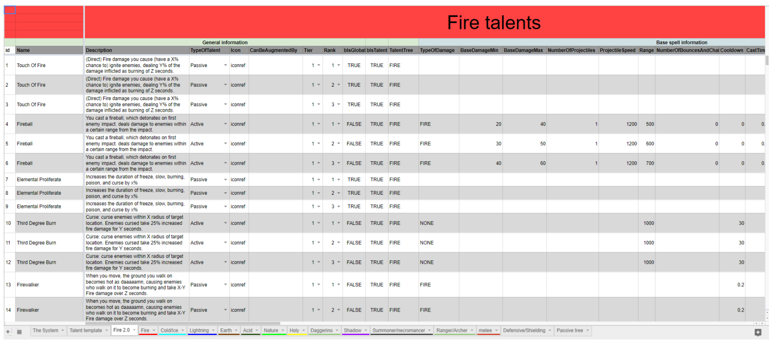
GUI - Graphical User Interface.

UMG - Unreal Motion Graphics.

B Planning Documents

The following are documents we created during planning and design.

B.1 Talent Trees with Spell Data



Fire talents

		General information										Base spell information							
id	Name	Description	TypeOfTalent	Icon	CanBeAugmentedBy	Tier	Rank	bIsGlobal	bIsTalent	TalentTree	TypeOfDamage	BaseDamageMin	BaseDamageMax	NumberOfProjectiles	ProjectileSpeed	Range	NumberOfBouncesAndChai	Cooldown	CastTim
1	Touch Of Fire	(Direct) Fire damage you cause (have a X% chance to) ignite enemies, dealing Y% of the damage inflicted as burning of Z seconds.	Passive	iconref		1	1	TRUE	TRUE	FIRE									
2	Touch Of Fire	(Direct) Fire damage you cause (have a X% chance to) ignite enemies, dealing Y% of the damage inflicted as burning of Z seconds.	Passive	iconref		1	2	TRUE	TRUE	FIRE									
3	Touch Of Fire	(Direct) Fire damage you cause (have a X% chance to) ignite enemies, dealing Y% of the damage inflicted as burning of Z seconds.	Passive	iconref		1	3	TRUE	TRUE	FIRE									
4	Fireball	You cast a fireball, which detonates on first enemy impact. deals damage to enemies within a certain range from the impact.	Active	iconref		1	1	FALSE	TRUE	FIRE	FIRE	20	40	1	1200	500	0	0	0.
5	Fireball	You cast a fireball, which detonates on first enemy impact. deals damage to enemies within a certain range from the impact.	Active	iconref		1	2	FALSE	TRUE	FIRE	FIRE	30	50	1	1200	600	0	0	0.
6	Fireball	You cast a fireball, which detonates on first enemy impact. deals damage to enemies within a certain range from the impact.	Active	iconref		1	3	FALSE	TRUE	FIRE	FIRE	40	60	1	1200	700	0	0	0.
7	Elemental Proliferate	Increases the duration of freeze, slow, burning, poison, and curse by x%	Passive	iconref		1	1	TRUE	TRUE	FIRE									
8	Elemental Proliferate	Increases the duration of freeze, slow, burning, poison, and curse by x%	Passive	iconref		1	2	TRUE	TRUE	FIRE									
9	Elemental Proliferate	Increases the duration of freeze, slow, burning, poison, and curse by x%	Passive	iconref		1	3	TRUE	TRUE	FIRE									
10	Third Degree Burn	Curse: curse enemies within X radius of target location. Enemies cursed take 25% increased fire damage for Y seconds.	Active	iconref		1	1	FALSE	TRUE	FIRE	NONE					1000		30	
11	Third Degree Burn	Curse: curse enemies within X radius of target location. Enemies cursed take 25% increased fire damage for Y seconds.	Active	iconref		1	2	FALSE	TRUE	FIRE	NONE					1000		30	
12	Third Degree Burn	Curse: curse enemies within X radius of target location. Enemies cursed take 25% increased fire damage for Y seconds.	Active	iconref		1	3	FALSE	TRUE	FIRE	NONE					1000		30	
13	Firewalker	When you move, the ground you walk on becomes hot as daaaaamn, causing enemies who walk on it to become burning and take X-Y Fire damage over Z seconds.	Passive	iconref		1	1	FALSE	TRUE	FIRE	FIRE							0.2	
14	Firewalker	When you move, the ground you walk on becomes hot as daaaaamn, causing enemies who walk on it to become burning and take X-Y Fire damage over Z seconds.	Passive	iconref		1	2	FALSE	TRUE	FIRE	FIRE							0.2	

The System · Talent template · Fire 2.0 · Fire · Cold/Ice · Lightning · Earth · Acid · Nature · Holy · Daggerino · Shadow · Summoner/necromancer · Ranger/Archer · melee · Defensive/Shielding · Passive tree

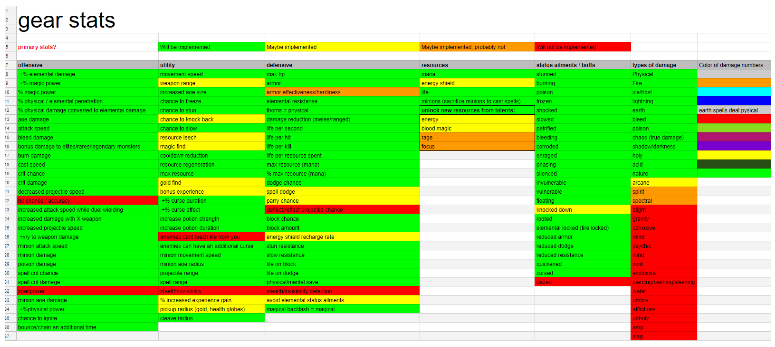
Figure 37: Talent trees with spell data

https://docs.google.com/spreadsheets/d/1f7EeMg76hzn-XiwW8spAfZxvTix8fkUhyYdQWWkHqt8/edit?usp=sharing

B.2 Project Plan

https://docs.google.com/document/d/1CqdQSdjZ55Ujr47SlFlrS5Cvp1NPgciDFscWgvcrjKO/edit?usp=sharing

B.3 Gear & Statistics



gear stats

5	primary stats?	Will be implemented	Maybe implemented	Maybe implemented, probably not	Will not be implemented		
7	offensive	utility	defensive	resources	status ailments / buffs	types of damage	Color of damage numbers
8	+% elemental damage	movement speed	max hp	mana	stunned	Physical	⬜
9	+% magic power	weapon range	armor	energy shield	burning	Fire	🟧
10	% magic power	increased aoe size	armor effectiveness/hardiness	life	poison	ice/frost	🟦
11	% physical / elemental penetration	chance to freeze	elemental resistanse	minions (sacrifice minions to cast spells)	frozen	lightning	🟦
12	% physical damage converted to elemental damage	chance to stun	thorns = physical	**unlock new resources from talents:**	shocked	earth	earth spells deal pysical
13	aoe damage	chance to knock back	damage reduction (melee/ranged)	energy	slowed	bleed	🟥
14	attack speed	chance to slow	life per second	blood magic	petrified	poison	🟩
15	bleed damage	resource leech	life per hit	rage	bleeding	chaos (true damage)	🟪
16	bonus damage to elites/rares/legendary monsters	magic find	life per kill	focus	corroded	shadow/darkness	🟪
17	burn damage	cooldown reduction	life per resource spent		enraged	holy	🟨
18	cast speed	resource regeneration	max resource (mana)		phasing	acid	🟩
19	crit chance	max resource	% max resource (mana)		silenced	nature	🟩
20	crit damage	gold find	dodge chance		invulnerable	arcane	
21	decreased projectile speed	bonus experience	spell dodge		vulnerable	spirit	
22	hit chance / accuracy	+% curse duration	parry chance		floating	spectral	
23	increased attack speed while duel wielding	+% curse effect	deflect/reflect projectile chance		knocked down	blight	
24	increased damage with X weapon	increase potion strength	block chance		rooted	gravity	
25	increased projectile speed	increase potion duration	block amount		elemental locked (fire locked)	corrosive	
26	+x/y to weapon damage	enemies cant leech life from you	energy shield recharge rate		reduced armor	mind	
27	minion attack speed	enemies can have an additional curse	stun resistance		reduced dodge	psychic	
28	minion damage	minion movement speed	slow resistance		reduced resistance	wind	
29	poison damage	minion aoe radius	life on block		quickened	void	
30	spell crit chance	projectile range	life on dodge		cursed	explosive	
31	spell crit damage	spell range	physical/mental save		dazed	piercing/bashing/slashing	
32	spellpower	stealth/invisibility	stealth/invisibility detection			water	
33	minion aoe damage	% increased experience gain	avoid elemental status ailments			umbra	
34	+%physical power	pickup radius (gold, health globes)	magical backlash = magical			afflictions	
35	chance to ignite	cleave radius				unholy	
36	bounce/chain an additional time					amp	
37						slag	

Figure 38: Gear and statistics

https://docs.google.com/spreadsheets/d/1cyh9e3ItqBxmDLcH58yJQS2uLOWmN3yihGqDWwoGe2I/edit?usp=sharing

B.4 Stats: how they work

https://docs.google.com/document/d/1HvEt0sjKxr1CdNHiGieELwx_NEdk7n2HhzY7zhafcD4/edit?usp=sharing

B.5 Bachelor Decision Notes

https://docs.google.com/document/d/1f1-j0gJsinu9-4E0KpkZsn7LslyjyIn44LH2TU-qom0/edit?usp=sharing

B.6 Risk Analysis

https://docs.google.com/spreadsheets/d/1eu4ZvMS97GmrGyZpk-1aNXwYRyhLgyS-wx3QKZXZJ04/edit?usp=sharing

B.7 GUI Mockups

B.7.1 Main Menu

https://docs.google.com/drawings/d/1XZ4ja2_eENPB6u35cVZkFIcJ0sdWqA2Phrd029bbU94/edit?usp=sharing

B.7.2 Ingame Menu

https://docs.google.com/drawings/d/1JpWKXR4Qz-ZRIDjEST-4e7Bd0fHQXU1YC4Wsqy-HTlk/edit?usp=sharing

B.7.3 Talent Tree

https://docs.google.com/drawings/d/1k7dhuA6sxu7BMyZfztsHrDEAHNdH0jZirYgUWPjNSsU/edit?usp=sharing

C Design Document

What follows is a downloaded copy of our design document located in a shared google drive folder. Hyperlinks located inside the design document appended below does not work. If desired, you can access the original document here. https://docs.google.com/document/d/16A9gJoNx2cxYq8mkVOpDGJb_xe35nrBI9LLNNuW4bAY/edit?usp=sharing and access the hyperlinks from there.

<Codename: Avacyn>

<Your Company Logo Here>

Revision: 1.0.0

Overview

Theme / Setting / Genre

- Dungeon Crawler / Hack and Slash / Action / RPG
- Set in a high-magic world. Think Medieval with orcs, goblins, trolls, wizards, dryads, ents, dragons, warlocks, witches, cultists worshiping different religious/powerful beings, and many more.

Core Gameplay Mechanics Brief

- Increasingly difficult dungeons with increasingly more powerful loot.
- Procedurally generated maps and dungeons.
- Diverse ability system through talent tree.
- Softcore and hardcore modes.

Targeted platforms

- Windows PC
- Linux PC

Monetization model (Brief/Document)

Paid final product. No ads, micro-transactions or subscriptions. There is a possibility to monetize through micro-transactions such as unique skins for character model/gear.
Link to Monetization document.

Project Scope

Time scale for the project
Expenses

Assets

To spice up the game, making it aesthetically pleasing. The amount of assets and which assets we need is still unknown. We will mostly stick to the Infinity Edge asset packs. These packs have been released to the public for free.However, we intend to purchase assets such as spell effects, as most of the effects from Infinity edge feel a bit underwhelming.

Time Scale

18 weeks total. Refer to the Gantt diagram for more details.

2 weeks of planning

Planning all features of the game as well as agreeing on a design for the game.

4 weeks of initial implementation.

Initial implementation consists of setting up a testing map, configuring game settings such as camera angle and position, creating menus and GUI for the game. Implement character statistics, a simple AI, a few spells and abilities for testing.

8 weeks of Primary content creation + user testing.

This is where most of the programming will be done. In this part we create all the spells that are planned to be in the game, implement all the different models, create a random map generator, create custom magical items for the loot system as well as a user test phase in the later parts of this phase.

2 weeks of feedback changes + final polish + balancing.

After having gotten some feedback from testing, we go through all the feedback and fix bugs and improve upon the system depending on the user feedback received. We also do final polishing of features. If time allows for it, we will implement weather effects as well.

2 weeks scheduled Report writing.

Although we plan to write the report throughout the project, we set aside two weeks to write the unfinished parts of the report. This time will also be used to go through the report and make sure nothing is missing.

Team Size
Kristian Gregersen
Game design, talent system design, ability/spell design, AI design

Sebastian Kolbu
Game design, level design, character design, talent system design, inventory system design.

Petter Ballangrud
Graphics programming, Unreal Engine programming, Optimization.

Licenses / Hardware / Other Costs

Visual Assist
makes coding easier as Intellisense for visual studio does not quite work with large projects such as unreal engine. [2]

Total Costs with breakdown
Visual assist (Academic License) - $49 per license. (3 Licenses)
FX packs ~ 10-25 Eur/ea
Enemy characters ~ 17.5 Eur/ea

Influences (Brief)

Diablo series (Game)
Very similar style of game to what we are making.
Path of Exile (Game)
A complex talent and gear system, much like the game we envision.
Torchlight series (Game)
Very similar style of game to what we are making.
Borderlands (Game)
Interesting stat and loot system. Great story. Funny game.

Divinity: Original Sin series (Game)
Interesting gear and combat abilities. Strategy mechanics.
Tales of Maj'Eyal (Game)
A turn based approach to a dungeon crawler / RPG which has some interesting spells/abilities and stats.
Wolcen: Lords of Mayhem (Game)
A dungeon crawler currently in development. Interesting twists on genre mechanics. Interesting talent system. Innovative resource system.
World of Warcraft (Game)
We take inspiration from different spell mechanics and the GUI customization.

The elevator Pitch

<Codename: Avacyn the angel> is a modern take on a hack & slash dungeon crawler with a completely open talent system that allows the player to chose their unique progression path. You will be able to explore a high magic medieval world, filled with magical creatures to fight and legendary items for your character to use on its journey.

Game Description (Brief):

Our game is a dungeon crawler / Action / RPG game where the main focus is gear acquisition and character customization through talent specializations. The player run around, defeating monsters and gain gear by doing so. Progression is gear and experience based.

The perspective is a top-down isometric view. The camera follows the player at a fixed position.

Game Description

What sets this project apart?

- Unique talent system
- Foes have unique abilities, never before seen in games.
- Spell augmentation system

- Combining equipment customization and talent system to create powerful synergies.

Menus (put it here for now)
We will have a menu that is our entry point to the game. When the game launches, you will appear in the menu. The menu will be available from within the game as well, though in a smaller scale(?).
Link to mockup of the Menus.

NPC dialogue (Put it here for now)
For now, we have no NPC dialogue. Will get back to it once we know that we have enough time to implement such a feature.

Graphical User Interface

Inventory system
The inventory system can be accessed by pressing ‘i’. Once pressed, a panel will slide from the side and display the current character's inventory.
Mockup of inventory system.

Switch inventory button
Once clicked, this button will display a list of all characters on the player’s account. Selecting any other character than the one currently playing will open up their inventory for you to look at, but you can not do anything with these items. It is merely to conveniently allow the players to look at their other characters if they are for instance looking for a specific item.

Item filtering and sorting systems
There are drop down menus to select options for how you want to sort and filter your items. Sorting can be done by; item rarity, amount of a specific stat on the item and level requirement. You can sort by up to three things at the same time, to make it as easy as possible to find the items you are looking for in your inventory. On top of this, you will have the option to filter by similar things to gray out all items that doesn’t match the filter.

Category display
There is a list of all types of items the character has collected and is currently stored in the inventory system. When clicked, it will expand and show a grid, filled with all items of that type.
A category will only be displayed if the character has collected and is currently in possession of an item of that kind.
Equipped items
The equipped items are displayed when opening the inventory system. They are displayed on top of an image of the player's character, with ‘boxes’ position at specific locations representing how the character wears them. For instance the boot box, located at the feet of the image. This makes it easier for the player to intuitively find where an item is located. This is illustrated on image 5 of the mockup: Game UI.

Quest system

Story quest line
This is an ongoing quest line which follows the events of the story written further down in this document. The plan is to have a simple quest tracking system initially.

Interface
The interface for showing quests can be found in-game by pressing ‘j’. Once pressed, a window will slide inn from the right. This window contains all the quests you are currently working on. There are a few different tabs you can select. One for quests available, allowing the player to get an overview of all quests he/she can take. Another tab for quests completed, in case the player is unsure if they have completed a certain quest or not.

Skillbook
The skillbook is where the player can find all abilities gained through the talent system.To access this menu, the player can press ‘k’. The skills are categorized by which talent tree they are gained from. To reduce clutter, a category is only shown if the player has actually gotten an active skill from that category. This way it is easy to get a good overview of all available active skills.

Character statistics

We have made a list of all the different stats we want in the game. This list is quite extensive, so writing them down here and defining them takes up too much space (for now). Look in the Gear & statistics document for a complete list.

Nearly all stats in the game will come from items and equipment the character uses, with the exception of main stats and some resistance gained from the talent trees. All stats belong to one of four groups of statistics. These groups are the following:

Offensive

The offensive stats are mainly focusing on increasing the character's damage power.

Defensive

Defensive stats increase the toughness of the character, allowing the character to take more damage before dying.

Utility

Utility is a collection of interesting stats that focus on tuning the character to the player's liking. An example of this is getting a pair of boots which increase movement speed, allowing the player to traverse the map faster, increasing the speed at which the character clears the level.

Resources

Increases the amount of resources the player has access too, such as increased maximum mana pool.

Core Gameplay Mechanics (Detailed)

Character creation

To begin playing, the player must first create a character to play with. To do this, they go to select character from menu, and click the create new character button. In that menu the player can customize the physique of the character, although this has no effect on the stats or capabilities of the character itself. Due to our unique talent system, there are predefined character classes to choose from.

Therefore, all players start out the same, and then customize and tweak their own character to fit their playstyle.

In order for character creation to complete, the player must choose a name for the character. The name has to be unique.

Character progression
A character can gain experience by killing foes. When a character reach a certain amount of experience, they increase their level. Starting at level 1, a character can progress all the way up to level 100. 1 talent point is rewarded each time the character gains a new level.

Increasingly difficult dungeons with increasingly more powerful loot
The player will start of doing fairly easy dungeons with mobs that drop fairly weak gear. As the player gets better gear, he will be able to beat stronger monster, which drops better loot, which in turn allows him to kill even stronger mobs.

The dungeons will scale based on player level and the players chosen difficulty level. Higher difficulty levels reward more experience, currency and a higher chance of better loot, but will require you to defeat stronger mobs.

Loot
In the game, monsters and the environment will drop loot. The rarity of the loot is graded on this scale:

Color	Magical	Rare	Epic	Legendary	Boss gear / Unique
num. Stats on item	2	4	6 or 5 + cool effect	7/8 + legendary effect	7/8 + legendary effect

The different stats that gear can have is listed in the gear & statistics document. This is a living document, so changes may occur. There are too many stats to list in this document.

Procedurally generated maps and dungeons
There will be randomly generated maps and dungeons with mobs that are scaled to the player’s level /*eh, placeholder details*/

The randomly generated maps and dungeons will based on themes for the dungeon, layout styles for the maps, sets of monsters and boss mobs etc. Some maps will have a set theme and type of monsters/boss(es) to allow for story development, while others will be mostly random.

Diverse ability system through talent tree

The player will be able to choose talets from multiple talent trees with different categories. To gain these talents, the character must gain additional levels which grant talent points. These points are allocated in order to gain the desired talents. Each talent will either give an active skill for the player to use, or a passive skill which either augments a skill they already have or amplifies the characters power in some other way. Each talent has three ranks, increasing the power of the talent for each point spent. The entry tier is 1. To reach the next tier of abilities in a tree, you need to cumulate 5 points per tier.

Tier Rank Example 1:

In order to allocate points in tier 2, the character needs to put 5 points total into any talents selectable in tier 1. To reach tier 3, the character must allocate 10 points combined in tier 1 and 2 talents located in the current talent tree.

For each point spent in a specific talent tree, that talent tree will grant you a pre-defined main stat according to how many points you have spent in that tree, multiplied by level. Talent trees can in addition to main stat also grant resistances relative to the type of tree, for instance fire granting fire resistance.

Example 1:

you spend 20 points in the fire magic part of the talent tree, at level 20 you will then inherently gain 20 * 1int * 20 levels + 20 * 0.05% * 20 levels = 400 int & 20% fire resist.

Example 2:

earth magic. 10 points spent in tree. Lvl 15 character.

10 * 0.5 con * 15 + 10 * 0.5 int * 15 + 10 * 0.05% * 15 = 75 con & 75 int & 7.5% earth resist.

As well as having put points in acid part of the talent tree, granting
5 * 1 int * 15 + 5 * 0.05% * 15 = 75 int & 3.75% acid resist.
With a grand total of 75 con, 150 int 7.5% earth resist and 3.75% acid resist.

Softcore and hardcore modes
Softcore is the more casual friendly game mode, that allows the player to die without much consequence, giving the players the opportunity to try and fail as many times as they desire. Hardcore on the other hand, once a character dies, it is transferred to Softcore, allowing the players to continue with their character. Hardcore mode is more of a prestige mode, where players can brag about their accomplishments without dying a single time.

When you first create a character, you can select between softcore or hardcore mode, which determines the rules for the game while playing said character. If that character dies in hardcore, all equipped gear and items in inventory will be transferred along with the character to Softcore. Items stored in the shared storage as well as currency remains only accessible to hardcore characters.

Gameplay (Brief)
<The Summary version of below>

Gameplay (Detailed)

Look and feel
We wanted to look and feel of the game to be customizable in the sense that the player may configure their own GUI by moving GUI-objects around on the screen themselves, tailoring their own experience.

The overall tone of the game is not particularly dark or grim, however there may be times, especially going through the storyline that areas and events will be darker. With our Realmwarp™ game-mode, we plan to have all sorts of different realms.

Camera

The camera is positioned above the character at a slight angle, also called an Isometric view. When the character moves, the camera will follow the character at a matched pace.

Key inputs

The default key inputs will be:
1 through 0 will map to Actionbar 1.
q,w,e will map to flasks 1 through 3.
‘J’ map to quest log panel.
‘K’ map to skill panel.
‘I’ map to inventory panel.
‘P’ map to talent window.
‘c/u’ map to character statistics panel.

However, the player can go into settings/options and choose which keys map to which actions themselves.

Maps

The maps/zones Will be semi procedurally generated. Only parts of a map will be generated, this is so that the maps can have a certain identity. All maps will have some parts of them that make them unique, as well as each map has a predefined size.

Environment

Weather effects

We are currently undecided on whether or not to implement weather effects. These weather effects would also affect the player in different ways. If we decide to implement them, the following effects are the once we’re considering:

- Snow
  Snow will make surfaces slippery, affecting the controls, making movement feel slightly sluggish.
- Rain
  Rain will drench the character, slowing down the movement speed as a

result of becoming heavier. Reduces lightning resistance depending on the intensity of the rain.

- Thunderstorm
  Lightning strikes down randomly, if it hits the character, it becomes shocked as well as take a little bit of damage.
- Windy
  When the environment is windy, the player gains movement speed when moving along the wind direction, and reduces movement speed when going against the wind.
- Extreme heat
  Extreme heat reduces the characters fire resistance depending on the intensity of the heat.

Mesh destructible objects

Some objects in the world can be destroyed by the player.

Story and Gameplay

Story (Brief)

The guardian angel Avacyn has been corrupted by an unknown source. As a result, she is rampaging throughout the lands, destroying everything in her way. Avacyn is not completely corrupted yet, and is talking to you begging for help.

Story (Detailed)

Main character lives isolated, deep in the forest on a farm. There's a village a day's walk away. One day, when visiting town, the main character hears rumors about Avacyn. The rumors talk about that she's not acting normal lately. There are also rumors about villages being destroyed. The main character just shrugs it off, thinking they are just exaggerating.

One night, while sleeping, the main character hear whispers in his/her dream. The whispers become louder and more clear, when suddenly, the main character wakes up. It is silent for a few seconds, then the whispers continue.

<insert dialog between main character and avacyn, where avacyn asks for help to free her from this madness>
Hesitantly, the main character agrees on venturing out to help Avacyn. The main character packs his/her bag, and sets off for an adventure to the land far away named <insert name>.
Avacyn guides the main character through the lands. During the travels, Avacyn tells the main character about a mystical device which has the capability to free Avacyn. But, the device has been lost to time, so the main character has to find clues about where it is.

After looking for it, the main character discover that the device has been shattered in 3 pieces. The main character has to find all the pieces and put them together again.
Once the device is whole again, the main character heads to to lair of the evildoers, located far north in <Insert name>.

Once there, the main character realizes that there's no way he/she will get inside through the main entrance, which is covered with armed cultists. So he/she has to find another way to get inside.

After a while of looking, the main character discovers the secret side entrance. Inside is a labyrinth of dangerous rooms filled with traps. Throughout the labyrinth the main character finds rare, magical items. The main character confront the evildoers, but the evildoers escape. The main character chases after them. Cornered, the main character stops the evildoers and save the angel Avacyn. Avacyn is grateful, then asks for one more favor. Avacyn tell the main character about all the other realms which exist, and that evil has been stirring everywhere, more now than ever. She can't handle it all alone, and asks you to help her. Avacyn teleports the protagonist to different realms to deal with the evil.

Assets Needed

2D

Textures

- Environment Textures
- Character texture

- Equipment textures
- Weapon textures

3D

Characters List

- Character #1
- Character #2
- Character #3

Environmental Art Lists

- Example #1
- Example #2
- Example #3

Sound

Sound List (Ambient)

Outside

- Level 1
- Level 2
- Level 3

Inside

- Level 1
- Level 2
- Level 3

Sound List (Player)

Character Movement Sound List

- Example 1
- Example 2
-

Character Hit / Collision Sound list

- Example 1
- Example 2
-

Character on Injured / Death sound list

- Example 1

- Example 2
-

Code

Character Scripts (Player Pawn/Player Controller)

- Example
-

Ambient Scripts (Runs in the background)

- Example
-

NPC Scripts

- Example
-

Animation

Environment Animations

- Example
-

Character Animations

Player

- Example
-

NPC

- Example
-

FX

To display the abilities in our game the best way possible, we need assets made from effect designers.

Schedule

Planning

Time Scale: 2 weeks
Design document
Project plan
Mockup and proof of concept

Initial implementation

Time Scale: 4 weeks
Character
Implement all stats
Testing Map
Loot & stat system
Simple AI
A few spells and abilities

Primary Content creation + user testing

Time Scale: 8 weeks
Implement all spells
Create unique items
Fully implement all talents
Create all types of enemies
User testing
Random level generation

Feedback changes / final polish / Balancing

Time scale: 2 weeks
Weather effects

Report

Time scale: 2 weeks
Write the damn report.

References/credits

[1]

GDD Template Written by: Benjamin “HeadClot” Stanley and Alec Markarian
https://docs.google.com/document/d/1-I08qX76DgSFyN1ByIGtPuqXh7bVKraHcNIA25tpAzE/edit

[2]

Visual Assist - http://www.wholetomato.com/

[3]

Kakky - Unreal Marketplace assets. (holy shit they are good)

[4]

Rgy0409 - Unreal Marketplace assets (cool effects)

[5]

Quantum - Unreal Marketplace assets (cool weapon enchantments)
https://www.youtube.com/watch?v=-DjJjnzTKYc

[6]

Explosions Builder - W3 Studios Unreal Marketplace assets
https://www.youtube.com/watch?v=cl52HTbm8-Y&feature=youtu.be

[7]

mr.Necturus - Unreal Marketplace assets (enemy characters)

[8]

Procedural Dungeon Generation Algorithm - A Adonaac
http://www.gamasutra.com/blogs/AAdonaac/20150903/252889/Procedural_Dungeon_Generation_Algorithm.php

[9]

Path of Exile - Map generation presentation
https://www.youtube.com/watch?v=GcM9Ynfzll0&sns=em

[10]

[11]

[12]

[13]

D Enums

The following are enum values we created to keep track of different types of elements.

```
UENUM(BlueprintType)
enum class ETypeOfStat : uint8
{
  /* resources */
  // Int - current maximum possible health for the pawn
  RES_MaxHP                    UMETA(DisplayName = "Maximum HP"),
  // Int - current health for the pawn
  RES_CurrentHP                  UMETA(DisplayName = "Current HP"),
  //Float - how much life the pawn gains per second
  RES_HPRegenerationPerSecond          UMETA(DisplayName = "Regeneration Per Second"),
  // Int - current maximum possible mana for the owned pawn
  RES_MaxMana                    UMETA(DisplayName = "Maxmimum Mana"),
  // Int - Current mana for the pawn
  RES_CurrentMana                  UMETA(DisplayName = "Current Mana"),
  //Float - how much mana the pawn gains per second
  RES_ManaRegenerationPerSecond        UMETA(DisplayName = "Mana Regeneration Per Second"),
  // Int - defensive stat, reduces physical damage taken
  DEF_Armor                    UMETA(DisplayName = "Armor"),
  // Int - deals damage back to the instigator when struct by physical damage
  DEF_Thorns                     UMETA(DisplayName = "Thorns"),
  // Int - deals damage back tothe instigator when struct by magical damage
  DEF_MagicalBacklash              UMETA(DisplayName = "Magical Backlash"),
  // Float - % value
  DEF_PhysicalSave                 UMETA(DisplayName = "Physical Save"),
  // Float - % value
  DEF_MentalSave                   UMETA(DisplayName = "Mental Save"),
  // Float - % value, determine whether or not the owner is hit by ability or not.
  DEF_DodgeChance                  UMETA(DisplayName = "Dodge Chance"),
  // Float - % value, chance to block an attack
  DEF_BlockChance                  UMETA(DisplayName = "Block Chance"),
  // Int - reduces the amount of damage taken by this value on successful block
  DEF_BlockAmount                  UMETA(DisplayName = "Block Amount"),

  /* defensive / life regen       */
  // int - determines how much life the pawn gains per attack
  LO_LifeOnHit                   UMETA(DisplayName = "Life On Hit"),
  // int - determines how much life the pawn gains when blocking an attack.
  LO_LifeOnBlock                   UMETA(DisplayName = "Life On Block"),
  // int - Determines how much life the pawn gains when dodging an attack.
  LO_LifeOnDodge                   UMETA(DisplayName = "Life On Dodge"),
  // int - Determines how much life the pawn gains when killing an enemy.
  LO_LifeOnKill                  UMETA(DisplayName = "Life On Kill"),
  // int - Determines how much life the pawn regenerates per second.
  LO_LifePerSecond                 UMETA(DisplayName = "Life Per Second"),
  // int - Determines how much life per resource(mana) the pawn spends when using abilities.
  LO_LifePerResourceSpent            UMETA(DisplayName = "Life Per Resource Spent"),

  /* utility              */
  // float - % value - determines how much faster than base speed the pawn moves.
  UTIL_MovementSpeed               UMETA(DisplayName = "Movement Speed"),
  // float - % value - Reduces the cooldown on abilities.
  UTIL_CooldownReduction             UMETA(DisplayName = "Cooldown Reduction"),

  /* utility / offensive        */
  // float - % value - increases radius of spells and attacks
  UTIL_AoeSize                   UMETA(DisplayName = "AoE Size Modifier"),
  // int - increases the travel distance of projectile attacks
```

```
UTIL_ProjectileRange            UMETA(DisplayName = "Projectile Range"),
// int - increases the maximum distance from pawn a spell can be cast
UTIL_SpellRange                 UMETA(DisplayName = "Spell Range"),
// float - % - increases radius of cleave attacks
UTIL_CleaveRadius               UMETA(DisplayName = "Cleave Radius"),
// float - % - chance to freeze enemies on hit
UTIL_ChanceToFreezeOnHit          UMETA(DisplayName = "Chance To Freeze On Hit"),
// float - % - chance to stun enemies on hit
UTIL_ChanceToStunOnHit            UMETA(DisplayName = "Chance To Stun On Hit"),
// float - % - chance to slow enemies on hit
UTIL_ChanceToSlowOnHit            UMETA(DisplayName = "Chance To Slow On Hit"),
// float - % - increases duration of curses cast
UTIL_CurseDuration              UMETA(DisplayName = "Curse Duration"),
// float - % - increases effect of curses cast
UTIL_CurseEffect                UMETA(DisplayName = "Curse Effect"),
// int - increases maximum number of curses the pawn can apply to the same foe
UTIL_MaxNumberOfCurses            UMETA(DisplayName = "Max Number Of Curses"),

/* utility / defensive      */
// float - % - increases the potency of potions used
POT_PotionStrength              UMETA(DisplayName = "Potion Strength"),
// float - % - increases duration of potion effects
POT_PotionDuration              UMETA(DisplayName = "Potion Duration"),

/* minion              */
// float - % - increase movement speed of owned minions
MINI_MinionMovementSpeed          UMETA(DisplayName = "Minion Movement Speed"),
// float - % - increase aoe radius of attacks done by minions owned by pawn
MINI_MinionAoeSize              UMETA(DisplayName = "Minion Aoe Size"),
// float - % - increases aoe damage done by minions owned by pawn
MINI_MinionAoeDamage            UMETA(DisplayName = "Minion Aoe Damage"),
// float - % - increases damage done by minions owned by pawn
MINI_MinionDamage               UMETA(DisplayName = "Minion Damage"),

/* offensive           */
// float - % - increases damage done to elite enemies
OFF_EliteDamage                 UMETA(DisplayName = "Elite Damage"),
// float - % - increases damage done for aoe attacks
OFF_AoeDamage                 UMETA(DisplayName = "AoE Damage"),
// int - increases travel speed for projectiles
OFF_ProjectileSpeed             UMETA(DisplayName = "Projectile Speed"),
// int - increase amount of times a spell chains or bounces
OFF_AdditionalChainAndBounceTargets      UMETA(DisplayName = "Additional Chain And Bounce Targets"),
// float - % - chance to apply ignite to enemies hit by attacks
OFF_ChanceToIgniteOnHit           UMETA(DisplayName = "Chance To Ignite On Hit"),
// float - % - chance to apply bleed to enemies hit by attacks
OFF_ChanceToBleedOnHit            UMETA(DisplayName = "Chance To Bleed On Hit"),
// float - % - chance to apply poison to enemies hit by attacks
OFF_ChanceToPoisonOnHit           UMETA(DisplayName = "Chance To Poison On Hit"),
// ??
OFF_BurnBaseDamage              UMETA(DisplayName = "Burn Damage"),
// ??
OFF_BleedBaseDamage             UMETA(DisplayName = "Chance To Bleed On Hit"),
// ??
OFF_PoisonBaseDamage            UMETA(DisplayName = "Poison Damage"),

/* offensive / magic      */
// float - % - base power of magical attacks
OFFM_BaseMagicPower             UMETA(DisplayName = "Base Magic Power"),
// float - % - increases magical power of spells
OFFM_MagicPower                 UMETA(DisplayName = "Magic Power"),
// float - % - chance for spell to critically hit
OFFM_SpellCritChance            UMETA(DisplayName = "Spell Critical Chance"),
// float - % - damage done by spell when successfully critical hit
OFFM_SpellCritDamage            UMETA(DisplayName = "Spell Critical Damage"),
// float - % - speed at which spells are cast
OFFM_CastingSpeed               UMETA(DisplayName = "Casting Speed"),

/* offensive / physical */
// float – % – chance for physical attack to critically hit
OFFP_PhysicalCritChance          UMETA(DisplayName = "Physical Critical Chance"),
// float – % – damage done by attacks when successfully critical hit
OFFP_PhysicalCritDamage          UMETA(DisplayName = "Physical Critical Damage"),
// int – minimum damage done by physical attack
OFFP_PhysicalBaseDamageMin        UMETA(DisplayName = "Minimum Physical Damage"),
// int – maximum damage done by physical attack
OFFP_PhysicalBaseDamageMax        UMETA(DisplayName = "Maximum Physical Damage"),
// float – % – determines how fast the pawn attacks
OFFP_AttackSpeedModifier         UMETA(DisplayName = "Attack Speed"),
// float – time between attacks
OFFP_CooldownBetweenAttacks       UMETA(DisplayName = "Cooldown Between Attacks"),
// float – % – determines how fast the pawn attacks while dual wielding one handed weapons
OFFP_AttackSpeedIncreaseWhileDualWielding UMETA(DisplayName = "Attack Speed Increased While Dual Wie

/* damage modifiers / offensive */
// float – % – increase damage done by attacks with the given damage type
DI_PhysicalDamage            UMETA(DisplayName = "Increased Physical Damage"),
DI_FireDamage              UMETA(DisplayName = "Increased Fire Damage"),
DI_FrostDamage              UMETA(DisplayName = "Increased Frost Damage"),
DI_LightningDamage             UMETA(DisplayName = "Increased Lightning Damage"),
DI_HolyDamage              UMETA(DisplayName = "Increased Holy Damage"),
DI_ShadowDamage             UMETA(DisplayName = "Increased Shadow Damage"),
DI_NatureDamage             UMETA(DisplayName = "Increased Nature Damage"),
DI_PoisonDamage             UMETA(DisplayName = "Increased Poison Damage"),
DI_AcidDamage              UMETA(DisplayName = "Increased Acid Damage"),
DI_BleedDamage              UMETA(DisplayName = "Increased Bleed Damage"),
DI_ChaosDamage              UMETA(DisplayName = "Increased Chaos Damage"),

// float – % – ignores resistance for attacks done with the given type.
// resistance ignored is the same as damage type of attack
DP_DamagePenetrationPhysical       UMETA(DisplayName = "DamagePenetrationPhysical"),
DP_DamagePenetrationFire         UMETA(DisplayName = "DamagePenetrationFire"),
DP_DamagePenetrationFrost         UMETA(DisplayName = "DamagePenetrationFrost"),
DP_DamagePenetrationLightning       UMETA(DisplayName = "DamagePenetrationLightning"),
DP_DamagePenetrationHoly         UMETA(DisplayName = "DamagePenetrationHoly"),
DP_DamagePenetrationShadow        UMETA(DisplayName = "DamagePenetrationShadow"),
DP_DamagePenetrationNature        UMETA(DisplayName = "DamagePenetrationNature"),
DP_DamagePenetrationPoison        UMETA(DisplayName = "DamagePenetrationPoison"),
DP_DamagePenetrationAcid         UMETA(DisplayName = "DamagePenetrationAcid"),

// float – % – damage increases while wielding the given type of weapon
DIW_DamageIncreaseWithSword_1H       UMETA(DisplayName = "Damage Increased with 1h Sword"),
DIW_DamageIncreaseWithDagger       UMETA(DisplayName = "Damage Increased with Dagger"),
DIW_DamageIncreaseWithMace_1H       UMETA(DisplayName = "Damage Increased with 1h Mace"),
DIW_DamageIncreaseWithFist        UMETA(DisplayName = "Damage Increased with Fist"),
DIW_DamageIncreaseWithAxe_1H       UMETA(DisplayName = "Damage Increased with 1h Axe"),
DIW_DamageIncreaseWithShield_MH      UMETA(DisplayName = "Damage Increased with MainHand Shield"),
DIW_DamageIncreaseWithSword_2H       UMETA(DisplayName = "Damage Increased with 2h Sword"),
DIW_DamageIncreaseWithMace_2H       UMETA(DisplayName = "Damage Increased with 2h Mace"),
DIW_DamageIncreaseWithStaff        UMETA(DisplayName = "Damage Increased with Staff"),
DIW_DamageIncreaseWithBow        UMETA(DisplayName = "Damage Increased with Bow"),
DIW_DamageIncreaseWithAxe_2H       UMETA(DisplayName = "Damage Increased with 2h Axe"),
DIW_DamageIncreaseWithScythe       UMETA(DisplayName = "Damage Increased with Scythe"),
DIW_DamageIncreaseWithShield_OH      UMETA(DisplayName = "Damage Increased with Offhand Shield"),
DIW_DamageIncreaseWithQuiver       UMETA(DisplayName = "Damage Increased with Quiver"),
DIW_DamageIncreaseWithElementalOrb     UMETA(DisplayName = "Damage Increased with Elemental Orb"),
DIW_DamageIncreaseWithSpellbook      UMETA(DisplayName = "Damage Increased with Spellbook"),

// float – % – damage taken reduced of type
DR_DamageResistancePhysical        UMETA(DisplayName = "Physical Damage Resistance"),
DR_DamageResistanceFire         UMETA(DisplayName = "Fire Damage Resistance"),
DR_DamageResistanceFrost         UMETA(DisplayName = "Frost Damage Resistance"),
DR_DamageResistanceLightning        UMETA(DisplayName = "Lightning Damage Resistance"),
DR_DamageResistanceHoly         UMETA(DisplayName = "Holy Damage Resistance"),
  DR_DamageResistanceShadow          UMETA(DisplayName = "Shadow Damage Resistance"),
  DR_DamageResistanceNature          UMETA(DisplayName = "Nature Damage Resistance"),
  DR_DamageResistancePoison          UMETA(DisplayName = "Poison Damage Resistance"),
  DR_DamageResistanceAcid            UMETA(DisplayName = "Acid Damage Resistance"),
  DR_DamageResistanceBleed           UMETA(DisplayName = "Bleed Damage Resistance"),
  DR_DamageResistanceChaos           UMETA(DisplayName = "Chaos Damage Resistance"),

  //float - % - damage taken reduced of all types of damage
  DR_DamageReduction               UMETA(DisplayName = "All Damage Reduction")
};
```