NTNU
Norwegian University of Science and Technology

Creating a framework for an ARPG-style game in the Unity game engine

Author(s)

Ole-Gustav Røed
Eivind Vold Aunebakk
Einar Budsted

Bachelor in Programming [Games|Applications]
20 ECTS
Department of Computer Science
Norwegian University of Science and Technology,

20.05.2019

Supervisor Christopher

Sammendrag av Bacheloroppgaven

Tittel:	**Rammeverk for et ARPG-spill i spillmotoren Unity**
Dato:	20.05.2019
Deltakere:	Ole-Gustav Røed Eivind Vold Aunebakk Einar Budsted
Veiledere:	Christopher
Oppdragsgiver:	Norwegian University of Science and Technology
Kontaktperson:	Erik Helmås, erik.helmas@ntnu.no, 61135000
Nøkkelord:	Norway, Norsk
Antall sider:	63
Antall vedlegg:	4
Tilgjengelighet:	Åpen

Sammendrag: Spillet som ble laget for denne oppgaven er innenfor sjangerne Action Role-Playing Game (ARPG) og Twin Stick Shooter ved bruk av spillmotoren Unity Game Engine. Spillet inneholder tilfeldig utstyr som spilleren kan finne, og fiender spilleren må drepe på sin vei gjennom spillet. Denne oppgaven vil ta for seg forklaringen av utviklingsprosessen og vise til endringer og valg som ble gjort underveis.

Summary of Graduate Project

Title:	**Creating a framework for an ARPG-style game in the Unity game engine**
Date:	20.05.2019
Authors:	Ole-Gustav Røed Eivind Vold Aunebakk Einar Budsted
Supervisor:	Christopher
Employer:	Norwegian University of Science and Technology
Contact Person:	Erik Helmås, erik.helmas@ntnu.no, 61135000
Keywords:	Thesis, Latex, Template, IMT
Pages:	63
Attachments:	4
Availability:	Open

Abstract: The game created for this thesis is focused around the Action Role-Playing Game(ARPG)/Twin Stick Shooter genre making use of the Unity Game Engine to create and deploy the game. The game contains randomized items and enemies where the player is tasked to kill their way through the level. This thesis will outline the process of development and the different choices and changes that were made along the way.

Preface

We would like to thank IDI Ntnu for the opportunity to create a game and Christopher K. Frantz for helping us during the development and writing of this thesis as our supervisor.

Contents

List of Figures

List of Tables

Listings

1 Introduction

1.1 Introduction of group

The group consists of three people from the game programming course. We have no prior experience working together so this thesis will be our first project together. The game we wanted to make was within the "Action Role-Playing Game"/"Twin Stick Shooter" genre with inspiration in games like Diablo, Path of Exile and Enter the Gungeon. Our primary goal for this thesis is to gain better insight into the creation and planning of larger game projects with a focus on networking, game mechanics and AI. Two of the group members have more experience with the Unity Game Engine having used it for rapid prototyping of games, as well as other smaller projects. The last member has less prior knowledge of Unity, but a big upside to using Unity is that the learning curve is rather low so a prior knowledge from game programming in the past helps a lot.

1.2 Project Goals

The primary goal of this project for our group is to gain more knowledge on how to organize and develop bigger game projects. More specifically we want to explore interoperability and re-usability between components and better understand the different ways to handle this on a collaborative team. We currently have no plans to continue working on this after the project deadline, and we are treating this project as a learning experience.

The main focus of this project will be to create the systems and frameworks needed to design and make an Action-Oriented RPG (ARPG). This means that we will also spend time on creating editor tools and other out-of-game utility to better support reuse and variations of the different entities we create. By exploring how to use the editor we hope to establish a more solid understanding of how to allow designers to quickly change and add functionality without doing the lower level work. Unity is a complete ecosystem with support for custom editor tools so we will primarily focus on the different GUI frontends we can create in the Unity Editor. The end product will be a game prototype built using the systems and tools we create in the Unity Editor, and our focus will be on those tools before we start adding variety or visual-only changes.

The two main games we have used as inspiration are Diablo and Path of Exile, which is two of the most popular ARPGs on the market. Both of these games make use of the click to move style and the player is not in direct control of the character. This is the biggest difference between these games and the one we are creating as we are using a Twin-Stick style movement. Aside from that many of the elements we have chosen to implement are directly or inspired by those two games and we have access to the Path of Exile wiki [2] which contains a detailed description of how they handle different events in the game. This will serve as the foundation for our own game design and act as guidance in order to avoid many of the common problems developing an ARPG. Problems such as which order damage is applied, how attributes are calculated, etc. are all smaller issues that can be hard to get right without a lot of testing or using a third-party source as a base.

As a further limitation, we also wanted to make the game networked with co-operation functionality. To accomplish this we will have a look at the current unity HLAPI and a look at a third party networking library called Lindgren. The main reason for the use of a third party library is that the current Unity HLAPI is being phased out and its replacement has not yet been released. We will touch more on this later in the report.

We want to get a deeper look at what constitutes a good character system and how to create a stable environment for the character to interact with. Part of this environment includes things like event triggers and a physics foundation to support environmental interactions and a reliable system for interactions between the character and the world. Another part of the character system is the ability system that allows for customized variations and is further extensible on top of the base ability. We want to explore the different possibilities for abilities in the game and work on making these abilities as accessible as possible for other uses, such as in AI.

We want to explore the different ways to create a tailored level experience. Usually, most ARPGs contain pseudo-randomized levels, but we wanted to instead focus on a more hand-made approach in order to gain some better insight into what makes a good level, and what steps to take in order to conform to certain visual aesthetics. This will be a minor goal for our project and the levels we create are mostly to support the prototype game we create. Hand-made levels involve a lot of artist work and design and as such a big focus on this would not produce much useful information for this thesis. Creating a pseudo-random level generator was on the list of features we were looking into but considering the complexity and work needed for a stable output we chose to not do that.

A big part of the ARPG genre, with games like Diablo and Path of Exile, is the extensive loot or drop system. Items are randomly generated from a pre-defined set of domains depending on their rarity, called modifiers. We want to explore ways to create and make use of a similar system in order to better understand how an item system should be designed in order to create meaningful rewards for the player.

AI and agents will be a big focus of our goal as that is an important part of a PvE (Player vs. Environment) game. We want to explore different ways to approach AI in games and get more knowledge on creating dynamic and responsive AI. Dynamic and responsive is here in relation to the agent's ability to switch between modes of operation/engagement as a response to its environment and the player's actions. The main focus of the AI is to implement a way to design behavior comparing a few commonly used architectures to find the method that works best for our use case.

Our previous experience on bigger projects is of varying degree so a major part of our focus will be on project management and source control. This includes the different tools currently used in the industry. We want to gain greater insight into source code management and external meta information about the source tree. This meta functionality includes cooperation tools like Discord/Stack and the use of bug-tracking tools like issue boards and git. Another part of project management is time estimation for feature implementation. Part of our goal is to get a better look at the time used to implement a feature and to get a better understanding of how we should go forward in order to reach that estimation.

1.3 Target Audience

The thesis is targeting anyone with moderate to advanced knowledge in unity and game programming. The reader will gain insight in how we decided to design some common systems found in a wide range of games as well as some other alternative approaches. The reader having at least some experience in unity and C# programming is encouraged.

2 Game Design

2.1 Initial Idea

This section of the initial idea is our aspect of the full game and will contain some features we did not have time to implement. These changes are further explained in section 2.2.

The initial idea was to create a game with inspiration in ARPGs and bullet-hell type games similar to Diablo and Enter the Gungeon. The three main areas we wanted to focus on were:

1. Networking: Create a co-op experience where players can play together. Create situations and events that are unique for single/multiple people and allow players to work together to overcome situations in the game.
2. AI: Create a base for AI and agents that allows for variation and extensibility. Enemies of different archetypes (humanoid, creature, etc.) exist as a base to create the AI on top of.
3. Abilities: Create a system for abilities allowing us to design abilities of different types. We wanted to implement the required components to allow for abilities ranging from projectiles, to buffs to melee attacks. With a solid foundation, this system will be used for both the player and the AI and in some cases both can make use of the same abilities.
4. Gameplay: Create a world with different interactions and events the player has to overcome. Also creating a character system that gives room for fast-paced action and variety in player style.

2.1.1 Gameplay

We set a few goals that we wanted to reach with the game design. Fast action and giving the player access to randomized loot and classes/talents that provides changes to their abilities. To start with we spread wide with the different concepts we wanted in the game, as we were unsure of how much content we would have time to implement. We used the first few weeks of development to create a priority for the different systems and concepts we initially set. The main priorities were getting the general gameplay working and then create the systems to add more content to the game.

2.1.2 Goal

Our initial idea consisted of having a randomized loot system and several classes and abilities the player could choose from. Typically for ARPGs, the goal is to get as much good loot as possible as fast as possible, but often in these games the player themselves create a goal. Beat x boss with y class or before a certain level are examples of goals that the player can set for themselves, without there being an implicit system with special rewards in place. We wanted to mimic this effect with our design by giving the player a lot of options to approach a level.

To give the player a greater goal than to get loot we wanted to create a quest-system and a story-line/plot the player will follow. A common factor of what we were looking for in this quest system was not to create the generic repetitive quest types. These quests include tasks like killing x amount of an enemy or collecting x amount of a resource that drops from an enemy. Instead, we are going to focus on the general story-line and use the quests to guide the player along with the story and point them in the right direction. Instead of repetitive quests, we will be using achievements and a stat-tracker to keep count on how many times a player has killed a certain enemy or completed some other tasks in the game. This will still allow the player to have a more persistent goal than just the quests while not bothering players that don't care about meta-information.

2.1.3 World Design

Creating randomized levels is a good way to give variety to the player's experience, but is time-consuming to implement for more advanced cases. Mostly as a limit to the scope, we chose to create handmade levels, with certain areas created as reusable components. Spawn locations, chest locations and other content of the level will be randomly placed at different places in the level for each playthrough of that level. This will still give a feeling of difference between each playthrough even though the shape and size of the level stay the same.

We wanted to create a short segment of the story to show off some aspects of the game. We started by limiting ourselves to an outside area and a dungeon consisting of fights and puzzles with a boss room at the end. The levels will be laid out in a branching world giving the player some choice as to which areas they want to explore first. A quest is given to the player that takes them on a trip through these branching levels. Our goal with the level/world design is to have side-quests in each of the branching areas that are required in order to continue the main branch. At the end of the game, there will be a boss which is ultimately the final bad buy of the game.

2.1.4 Player

Classic ARPGs like Diablo are commonly controlled via point-to-click movement, but we felt like changing this up and implement direct control of the character. The controls of the player can be described as typical Twin-Stick Shooter controls, where one stick controls the movement and the other for aiming with the character. This allows the player to move in one direction and look in another, allowing for greater freedom of movement while trying to attack an opponent. With this increased freedom of movement we can from a design standpoint create a more hectic experience for the player with more enemies and effects active at a time.

2.1.5 Character

The player and other agents based on the player character system will make use of two resource pools, health, and power. Power is most similar to mana in other RPGs and acts as a source of energy to use abilities. This exists as a way to limit the overuse of powerful abilities and creates another way to balance abilities without removing the punch from it. Both these systems are regenerative resources and will slowly restore when the character has not received damage for some time.

We went through a few different ARPGs to find different ways to create a character system. The character system is here the different RPG elements of a character. We chose to make use of a minor stat only attribute system which means that we will not make use of the common trifecta of main-stats, strength, dexterity, and intelligence. These main stats are commonly linked directly to two or three minor stats and usually have no other character value in of itself, which is the main reason we skipped them. Each character has a set of base values for each of the attributes, and items, spells, and other game elements will modify those values to improve the strength of the character.

Minor Stats List:

- Armor - Value reduces incoming physical damage
- MinDamage - Minimum physical damage of certain attacks
- MaxDamage - Maximum physical damage of certain attacks
- Range - Melee attack range
- AttackSpeed - Determines the rate of attacks for certain abilities
- SpellDamage - A flat value added to magical abilities damage
- CastSpeed - A percentage increase/decrease in the cast speed of spells
- CritChance - Chance to deal additional damage with an attack
- CritMultiplier - The amount of extra damage a critical strike does
- Health - Determines the number of health points the character has
- MagicResistance - Value reduces incoming magic damage
- HealthRegen - Delta regeneration of health regeneration
- HealthRegenDelay - Delay after receiving damage before health regen starts
- AttackPower - A flat value added to physical abilities damage
- MovementSpeed - Percentage increase in base movement speed of the character
- Power - Amount of power resource available to the character
- PowerRegen - How much power is regenerated each frame
- PowerRegenDelay - Delay before regeneration starts after receiving damage
- Level - Used to modify base values with given level's multiplier

Each attribute can roll as either a flat, multiplicative or additive modifier to give an additional axis of randomness to the values an item can have. A flat modifier will give a direct increase in the value but multiplicative and additive are two ways to sum up percentage values to get a multiplier. Items in the game can have one of six different rarities where five of them are randomized and one category is for named items. Named items are commonly referred to as uniques and the set of modifiers on it is predefined but the values that can roll are not. The other five categories are Common, Uncommon, Rare, Epic and Legendary and each category allows for a different max amount of modifiers it can receive.

2.1.6 Enemies

Initially, we planned to implement a lot of different types of agents, but access to enough assets and as a time limit we ended up on implementing humanoid and quadrupedal type agents. Four initial concepts for enemy types were made during the beginning of development, which were a wolf, a mage, a rogue and a warrior. These four would give a solid foundation to further create more enemy variety down the line. We did have more general ideas but a lot of those features were reintroduced in the four given enemy types and the boss encounters.

Humanoid and Quadrupedals

Humanoid agents in the game will move in the same way as the player, they are able to move in one direction and look in another direction. This characteristic is used to allow the agents to back off away from a player while still being able to prepare an attack. When the agent wants to break it will use the velocity direction in order to do so.

Contrary to humanoid avatars, the wolf is a quadrupedal animal that moves on four legs.

As such, they won't possess the same ability as a bipedal when it comes to directional movement. They can only add velocity to their facing (forward) direction and they have to rotate their entire body in order to change direction. This is a choice made as it fits best with the natural counterpart to what we are trying to create.

The Warrior

The warrior will make use of a basic attack that it can use all the time. It will stay close to the player and try to stay within the aggro zone(front) of the player. At certain intervals, the agent will pick an attack depending on the distance to the player. To close the gap it can make use of a charge attack to intercept the player. When it is in close range it will choose from two different abilities and pick the first one off cooldown. The first ability is a buff giving some positive effect to the agent and the second is an offensive attack that does a greater amount of damage.

The Mage

The mage is an ability user only and will not have a basic attack. It will try to keep its distance from the player and mainly use projectile attacks. The agent will be able to lead its shot and predict where the player will end up in order to do so. Some abilities can be given a greater accuracy than others to allow for different interactions with the player to different abilities.

The Rogue

The rogue tries to avoid the player but will keep a close distance in order to try and backstab the player. It will try to move behind the direction the player is looking and will do extra damage from behind. To avoid the enemy it can dodge away and make use of two abilities. The first ability is commonly referred to as "Shadowstep" and will teleport the rogue behind its current target to set up for an attack. It can also go invisible for a few seconds in order to reposition, but it will leave a smoke trail to still allow the player to keep an eye on it.

The Wolf

The wolf will roam around the player until its attack is ready again. When the wolf prepares for an attack it will run straight towards the player and leap towards them from a small distance. At under half health, the wolf will use a special attack that also slows the player for a few seconds. The main idea of the feel behind wolves is that one alone is not dangerous, and the feel of threat is mainly there when fighting four or five of them at the same time.

Interactions

Enemies can be equipped with different sensors to give them access to information about the world around them. Two basic sensor types that are used by most of the AI in the "PerceptionSensor" and the "ObstacleSensor". The perception sensor is responsible for information about the agents and players in the vicinity of the agent. Targets and objects of interest can be scanned for and used by the brain to create a world view. The obstacle sensor provides feedback about obstacles around the agent to allow them to dynamically respond to the environment around them. One use of the obstacle sensor is to slow down and steer away from a wall giving the interaction a more natural feeling.

A range of different actuators will allow the agent to get additional contextual information about the world and a way to react to them. To give the agents more fluidity in combat they will get access to information about incoming projectiles, which state the target is in, how much health the enemy has and a range of other statistics of varying scope. It will then be able to use this information in order to dodge away from attacks or assess if it should try to avoid the player until it can heal.

Boss

We wanted a boss archetype that allows the design of boss fights with different phases and abilities. The bosses will be put in special boss rooms that give them access to different room effects. We are then combining these different parts to create a boss encounter that pushes the player to their limit and provides a different combat experience than regular mobs.

2.2 Changes made during development

One of the goals for the design set by the product owner and the supervisor was to implement networking. We chose to do this by creating a co-op mode where up to four players can play together. The co-op mode works exactly like the single-player and allows players to team up and progress through the story together. To keep within the scope of time for the project we chose to not delve too much into special events for co-op only and chose to rather spend that time creating a solid experience accessible from both modes.

We were planning on spending more time on the level design and create a branching world which the player can choose their path. Because of the time we spent on the other parts of the game we quickly found out that this would take too much time to set up. As such we scrapped the plan for making a branching level design and instead focused our time on creating a smaller outside segment and a dungeon. The dungeon design we came up with was to create around seven rooms with a couple of puzzle rooms put in between fight rooms. The fight rooms will present the player with different enemies and ramp up towards the last room which is the boss room.

3 Technical Design

3.1 Game Application

3.1.1 Unity

Unity is the game engine used to develop the game for this report. It contains all the commonly used functionality of an engine and provides a framework for both the gameplay and the tools aspect used in the game development process. Following this is a short writeup of the integral parts of the Unity Game Engine to give a better foundation of understanding for our own work later on.

Unity is an entity component-based game engine. Entities in the game are known as GameObjects which is a container for components. To enable this the engine uses a monolithic hierarchical approach with less focus on inheritance and more focus on serving components as a separate service with limited scope. This has a few advantages over the use of OOP and strong use of inheritance as the functionality of each component is as isolated as possible from others, which allows for easier testing and better reusability of code. On the other hand, it can create more trouble down the line when extending the functionality of those components. The use of the monolithic component structure is not forced and we can and have made use of OOP in some of our approaches to allow more flexibility and create a stronger foundation for some of the systems.

Compiler

Up until a year or two ago Unity only supported the older version of .NET, version 3.5. They have now added support for the open source Roslyn compiler which in addition to incremental compilation gives us access to the latest features in .NET. The currently supported C# version is .NET 7.3 and that is the version we made use of for this project. On top of support for newer functionality in .NET the incremental compilation will also drastically reduce the compilation time whenever we make a change in our code. Instead of having to encapsulate our code in DLLs or wait for recompilation of the whole source code the Roslyn compiler will only recompile the source code that has actually changed.

On top of this Unity has created a compiler utility, called Burst, to create optimized native code for each of the target platforms. This alone does not give that much of a performance improvement as the utility is still in the experimental stage and is mostly targeted towards the new Entity-Component-System(ECS) framework that is supposed to replace the current one. It still gives a small performance boost, around 20-30%, compared to unoptimized code which means that we don't have to care about platform specific optimization.

We initially planned to use the new ECS framework but we quickly found out that it is not at a stage where we can make use of it. Mainly because it lacks the support of many of the base features like Mechanim (animation), networking and other smaller functionality we might need during the development process. Because sub-systems in the engine were not the focus of our project goal we scrapped this plan in favor of the current system. The new ECS framework has much greater performance, 200-600+% depending on the situation, when a lot of entities are present but this will be a rare occurrence in our game.

Experimental Entity-Component System

There is currently a new Entity-Component System (ECS) being implemented in the Unity experimental branch and we initially planned to make use of this for our project. The new system is more data-oriented and would allow us to more easily optimize for native data support and multi-threading. After the first sprint, we quickly decided that the new system was not in a development-ready state and would require us to spend more time on the internal parts of a game engine. Our goal for this project was more focused on the actual development of a game and not the technical groundwork done by most game engines and as such, we changed our plan to use the current entity component system in Unity.

Component and GameObject

Components all have to inherit from the internal MonoBehaviour class to be able to attach it to a GameObject. Unity will handle the lifetime of the entities internally and control the instantiation and destruction of entities. GameObjects can be saved as asset files to be reused as Prefabs, which is Unity's terminology for a saved instance of the GameObject. In addition to this Unity also contains the ScriptableObject class that can be inherited from. This class behaves in many ways as a MonoBehaviour but it does not attach to a GameObject as a component and as such allows for a lot of extensibility to MonoBehaviours. We decided to make good use of ScriptableObjects because it adds that extra bit of flexibility in how we approach each problem. It also provides faster access to the data as you don't have to scan the hierarchy of the prefab to update it from the editor. ScriptableObjects are fully serializable and Unity itself contains the functionality to serialize and save an instance of a ScriptableObject for later use. This enables us to decouple a lot of the variables from the component so each variation of the variable values won't need its own Prefab instance.

We will make use of the internal command structure in place on MonoBehaviours/GameObjects. It allows us to use the SendMessage function of the GameObject class to issue a command to all MonoBehaviours attached to that GameObject instance. This provides a better way to connect events that need to be received on multiple unknown MonoBehaviours. For the most part, we only issue SendMessage to the target object and avoid sending messages up the hierarchy in order to avoid performance issues related to climbing through large and unknown tree hierarchies. One downside to using SendMessage is that the link is not visible to the static code analysis tools in the coding environment. This means that it will be harder to track unused code sections and checking if a receiver exists requires some additional code.

MonoBehaviours also have support for Coroutines and unity has their own wrapper for the .net Coroutine library. Coroutines allow us to implement functions that can be run over a period of time or until a specified condition is met. This can be used to easily implement events where time has to pass before the rest of the function is executed. It can also be used to implement over time effects where it can trigger a piece of code every frame or at a given time interval. Coroutines are not the same thing as threading and for the most part, we won't have to worry about synchronization between the coroutines and the Update function. We will still have to solve situations where two coroutines set a flag and then unsets it after some time, where the first one will unset the flag before the last one has finished running.

Scene and World

To put everything together into a playable level Unity makes use of a scene system that stores each scene as a hierarchy in a YAML asset file. Much the same way as Prefabs each Scene file store a copy of everything in its scene hierarchy. Because of this we can store variations of prefabs in the scene file and avoid several prefab instances for smaller changes in the variables.

Unity internally updates the world on two different ticks. There is one game tick that is unlocked and a fixed tick that is locked to 60 FPS. The fixed tick is mainly used for the dynamic physics of Unity. Unity uses PhysX as their physics solver for 3D and contains a dynamic part but also allows for static calls to the PhysX instance of the world. We chose to make use of the dynamic part to handle triggers and create a character controller that has more control over the different interactions with the world. PhysX has support for layers and in the long run, allows for a lot of optimization. We use these layers to differentiate between different entities in the game so we can target those specific layers in calls to the physics world.

Editor

Unity is not only the game engine but also a framework to create editor tools. This framework allows us to extend the components and scripts we create with visual help and representation for the developer in the editor. A custom GUI solution can be made for individual fields (PropertyDrawers) and components (CustomEditor) but it also allows us to create Editor-Window's with the freedom to create almost any tool we might need for development. The editor contains functionality to handle objects in a serialized way using the SerializedObject class to encapsulate an instance of a class inheriting from UnityEngine.Object. This allows for a cleaner interface to the data in the different editor tools we create with support for undoing changes.

The GUI system used in the editor is currently split between two systems and does not always make use of a layout system to allow for easier setup of UI elements. PropertyDrawers cannot make use of layouts and as such is it much more cumbersome to setup UI in this class. We have to manually set up Rects to place the different elements in the UI. To help with this problem we will make use of a GUI wrapper (it's in the uFAction library) that allows us to make use of layouts and a more stable framework to avoid the common problems with the native Unity wrapper.

3.1.2 UI

Unity's UI backend makes use of an event-based system and is very similar to other UI systems that make use of events to update the interface.

3.1.3 Networking

The networking follows a client-host pattern where the "host" is the client that created the game session and is also acting as a server and a "client" is any player connected to the server. As this is a PvE (player versus environment) game with no competitive elements we allow each client to be in full control of the movement of their own character and the host will trust any information sent from the other clients. The server is responsible for ruining everything else including the AI characters, item transfers, doors and environmental effects. With the amount of systems planed for the framework the networking module was built with focus on ease of use and extendibility in mind. While this comes with the cost of performance is was deemed necessary to allow rapid development and iteration, especially for the ability system where every ability has different logic and requirements.

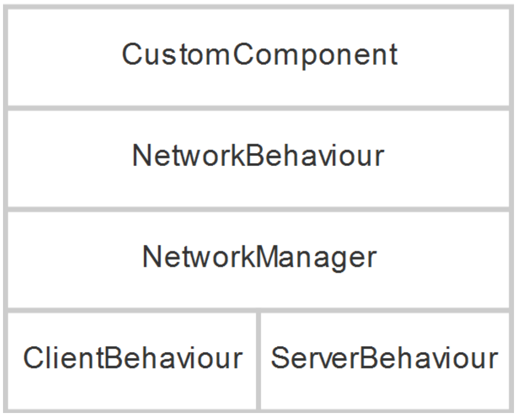


Figure 1: A simplified diagram of the network stack

At the heart of the framework is the NetworkManager. It manages the lifetime of the ClientBehaviour and ServerBehaviour, scene management, instantiating new entities on the network and routing incoming data to the correct NetworkBehaviour. When connecting to an existing game the NetworkManager will create an instance of the ClientBehaviour component. The ClientBehavior manages the connection the the server and sending and receiving data through the connection. When creating a game the an instance of the ClientBehavior component is instantiated instead. The ServerBehaviour manages connections to all the connected clients, identifying the sender and receiver for each packet and relaying and boadcasting information to all clients when applicable. Any data not directly handled by the ClientBehavior or ClientBehavior is passed on the the NetworkManager.

The main components for the networking framework the the NetworkIdentity and NetworkBehaviour components. The NetworkIdentity component represents the identity of a gameobject on the network and each GameObject with a NetworkIdentity attached will have the identity synchronised across all the clients to allow data to be sent to the correct destination; a NetworkBehaviour requires a NetworkIdentity attached the same GameObject to function. In addition the NetworkIdentity manages whether a player of the server has authority over the NetworkBehaviours on the object. NetworkBehaviour is the base class for all components that require network functionality and provides an easy way to send and receive data as well as identifying if the component is running on the server or a client.

3.1.4 Application

We create a manager as a singleton instance whenever we need a central place for information. This allows us to hook up information in the manager and then access that information from any point in the code. Pathfinding, object pools, and UI are examples of systems that will be accessed through these singleton instances in order to bind the different parts of the game together.

Blackboard

To give the agents access to a central location of information a basic implementation of a blackboard data structure is created. The blackboard data structure allows for a globally accessible table of information that can be set to only allow writing from the GameObject the blackboard is attached to. The blackboard is accessed on a GameObject through the BlackboardUser component, found in BlackboardUser.cs, and will automatically add certain enums as keys depending on the components that are attached to the GameObject. This will, in turn, allow for easier setup to access the values in the blackboard at a later time.

The blackboard is filled with BlackboardItem's, which is a container class for the different data types that can appear in the blackboard. It supports the use of specified enum's to be used as a key, in order to give easier access and in turn a name that can later be used when designing the AI or referencing blackboard items in the code.

Delegates

MethodReference is a reflection helper that allows us to serialize delegates in the editor and then deserialize them during runtime. In addition, a MethodReferenceDrawer is created to allow the user to bind methods in the editor to be called at runtime. Two attributes are made, MethodReference and Parameter. The first one is a class attribute and is used to sort out relevant classes to take methods from, and the second is a method attribute that takes strings as a parameter. The Parameter attributes parameters are displayed as a dropdown list in the editor to allow for different pre-defined choices that the method uses.

3.2 Item System

3.2.1 Items

Each item has a name, description and icon. Equipment is a special kind of item. It contains information about which equipment slot is fits in, the 3D models for the male and female characters, the UI icon, the base stats and generated properties such as the rarity and affixes.

3.2.2 Containers

The container system provides a consistent way to transfer any object between containers and ensures that any two containers are compatible with each other; the Inventory and CharacterEquipment are two examples of containers in the game. A special type of container is the NetworkContainer; it provides the same functionality as a normal container, but in addition any transfers between two NetworkContainers is synchronized for all clients in the game.

The Inventory is the simplest. It has a fixed size and accepts any Item, if all the slot are occupied an item can always be replaced with another.

The CharacterEquipment is more advanced as it only accepts Equipment and it must be placed in the correct slot, it'll check the equipment's level requirements against the character, make sure it's not using two two-handed weapons, not using a main-hand weapon in the off-hand and so on. If any of the requirements are not meet the object will be rejected.

The container system will handle any instance where a object can't be transferred as well cases where multiple objects are transferred at once such as when replacing two one-handed weapons with a single two-handed weapon, which is not possible is the characters inventory is full, without requiring any custom logic in the Inventory or CharacterEquipment.

3.2.3 Rarities

When generated equipment is assigned a random rarity which defines the minimum and maximum number of Affixes the equipment gets and heavily impacts the stats and the perceived value of the item.

3.2.4 Affixes

An affix is a randomly assigned stats modifier which adds either a prefix or a suffix to the name of the item it's assigned to, this allows us to generate a larger variety of items with different properties and names.

3.2.5 Loot tables

Treasure chests and enemies have a loot table which defines which item then can drop. A loot table is a list of items where an item can be picked at random based on the items weight in the table. Tables can also be nested in other tables to more easily create interesting lists of items. There is also a set of generated loot tables for all the different equipment slots and weapon types for different level intervals which makes it easy to create new tables with relevant content such as making a boss drop a few specific items in addition to a random level 40-45 helmet.

3.3 Character System

3.3.1 Character

All entities that are either an agent or a player makes use of a combination of UnityEngines premade components as well as a custom-made PhysicsComponent. The two internal unity components used for the character controller is the RigidBody and the CapsuleCollider. They provide us with an interface to the physics world in the unity engine and allows entities to be registered with the physics engine.

Character Controller

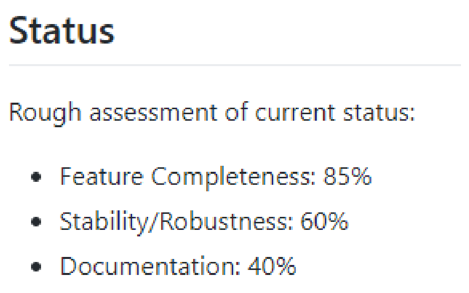
Hectic situations in combination with a dynamic physics system can be too random for the player. In order to tighten in the controls, we therefore opted for a static character controller implementation. Unity has a standard character controller component however to have more control over each interaction we implemented a custom solution. Our custom solution is structured much the same as the Unity internal one but simplifies some stages and adds more choices to layer interactions during different events. Another framework we used when designing this system was SuperCharacterController [3] but that project is no longer maintained and we only used it as a solution to certain problems.

The PhysisComponent maintains an internal velocity that is used to move the GameObject every frame. Velocity is a protected variable and is interacted with through meta functions like AddVelocity. Those functions make use of the acceleration of the entity to add to the velocity vector and as such this vector is mostly used for general movement. To allow for pushing and pulling of the entity a ForceEvent class is created that stores the variables used for that event. Each event is put into a queue and is not re-queued after its duration has expired. Different queues exist for different layer interactions in order to give access to functionality for passing through walls or other entities. All events in the queues are processed each frame and summed up into a common direction and force, reducing the number of raycasts needed in order to do collision checking.

3.3.2 Player

Input

As specified in the initial report, we decided to try out unity's new experimental input system. This system is not yet complete. As of 05.09.2019 unity's rough assessment of current status can be seen in figure 2. Those numbers were significantly lower during time of development. The fact that documentation was very lacking (roughly 20% at the time) made using this system rather cumbersome.


Status

Rough assessment of current status:

- Feature Completeness: 85%
- Stability/Robustness: 60%
- Documentation: 40%

Figure 2: Unity's new input system progress status

There are three main ways of using this system. One can Query the input states of devices (very much like how unity's old input system) is mostly used, implement event handlers that processes events directly or by using action maps. A action map contains a list of actions that the player can perform, such as moving around, using ability slot 1 or open the inventory. Any number of functions can be registered to the action and when the action triggers, those functions get called. Each action can have any number of bindings that will trigger it. Bindings can operate for one or more controller schemes which simplifies making support for multiple input devices. Entire action maps can be swapped runtime. This is great if for example there is a mini-game in which we want the controller scheme to change. In our project we used action maps for everything but obtaining mouse position. For cursor position we simply queried the input device directly wherever we needed it. This would be problematic later on if multiple input devices were introduced. It also takes a toll on performance as multiple queries and corresponding ray casts can happen on the same frame. A proper solution we have planned is to use action maps for mouse position as well. A "cursor moved" action would be created. For a mouse/keyboard scheme a binding of "mouse position changed" would be added. For a game-pad, binding the right joystick would be logical. Whenever the action is executed, the positional data of the input device would be stored in a module that other parts of the game can access. In code, the cursor would now be controller scheme independent and number of raycast to translate the cursor position to world position would be limited to one per frame.

Classes

We did not invest much time on designing classes. Creating a class with an unique set of abilities and talents is very time consuming. Because our focus was more on creating systems for a game, rather than the game itself, we decided that this was not to be prioritized. We created two classes; a mage and a warrior. A class is represented by a scriptable object containing the data of the class. this data includes a list of abilities, a prefab representing the talent tree system of the class and a few other minor things like the class name.

3.3.3 Ability System

There are many ways to design an ability system. Different approaches comes with different advantages and disadvantages. There is generally not a "best" solution and what works well for one game might not be great for another. The traditional unity approach would be using monobehaviour and prefabs. However in recent years scriptable object has surfaced as a serious contender and should definitely be considered when deciding on a system. There is also the question about having a more composition based system, where abilities can be built up by individual components, each responsible for a small piece of logic such as targeting or spawning a projectile, or going with inheritance. Because of our emphasis on systems and reusability rather than creating a complete game, we wanted a flexible solution that could handle both simple and advanced abilities and would be viable in a wide array of scenarios. We also wanted the system to be usable by both players and AI, thus coupling the abilities to players was to be avoided. It was also important for us that it would be easy to create multiple variations of an ability.

The system we implemented use a base class for abilities and one level of inheritance (excluding unity's scriptable object) to form individual ability types, such as projectile, field or dash. These again can be used to form many different abilities of that type by modifying data. For instance, Projectile ability can be used to create a frostbolt, a fireball, a chaos bolt or even a projectile that will heal your allies. Abilities also contains a component based system for adding additional behaviours without the need of further deriving those classes. Our system can be broken down to three parts:

- **Ability** is the base class that every ability will either use as is or derive from. It contains data that every ability requires such as the name, cooldown and power cost, as well as multiple virtual functions that can be overwritten by derived classes. It also contains a list of ability behaviours. Ability inherits from ScriptableObject. This enables us to rapidly create multiple variations of an ability type and save them as individual assets with a few button clicks. For instance, say we have a projectile ability. To create a fireball and a frostbolt ability all that would be needed is to create two separate assets of the projectile ability and set the data for the graphics to fireball and frostbolt respectively, as well as setting other data like damage and projectile speed to whatever is desire
- **AbilityBehaviour** is a base class for creating additional behaviours that can be added to an ability. Is contains numerous virtual functions that are called when various events regarding it's associated ability happens. Ability behaviours are meant to be used for modifications, but can theoretically form an ability from the ground up if attached to an instance of the ability base class. A simple example of a behaviour that we use for multiple abilities is one which applies a given status effect to any character the ability hits or even the caster itself.
- **AbilityUser** is a component that acts as the controller and monobehaviour proxy class for abilities. This component is attached to every gameobject that makes use of abilities. It stores an instance of each ability the actor has in its arsenal and provides exposure to functionality regarding abilities such as using them, manipulating them or listening to ability events. It also contains the logic associated with casting an ability. With casting we mean the "charge up time" between initiating the use of an ability until is launches, what is usually displayed with a castbar for the player in your typical mmorpg and other genres. It makes sense to have casting logic centralized in one place as we do generally

not want to have multiple casts happening simultaneously. In particular, casting multiple spells with animations in parallel would appear weird. Having the casting logic in one place makes it more prone from developers errors. It also makes listening to casts a trivial matter while at the same time keeping the ability base class cleaner. AbilityUser is also responsible for checking if the requirements such as power and cooldown are met to allow using the ability. These requirements can be toggled on and off.

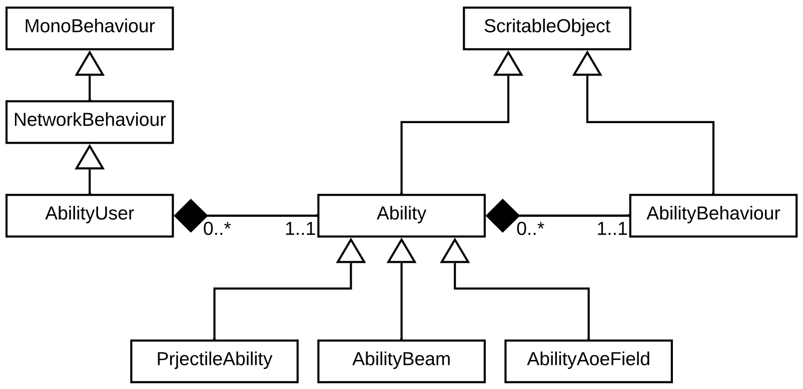


Figure 3: Simplified uml of the ability system's architecture. Note that this is representative for an individual actor runtime. The serialized ability and ability behaviour assets can be used by an unlimited amount of ability users and abilities respectively, but individual instances of these will be assigned runtime.

Global Cooldown

A common way to prevent a player from spamming their abilities is to have a global cooldown, preventing any use of abilities for a specified amount of time, usually a second or less. We initially went with the most common approach of handling global cooldown which is always have it be of equal duration. However, this didn't feel too great when play testing as it made combat flow somewhat awkwardly, especially since we wanted high paced combat. To allow for absolute flexibility and control, we decided to make each individual ability responsible for setting how long of a global cooldown it will trigger. This is one of the exposed fields in the Ability base class and can easily be set and tweaked in the inspector. Some abilities might want to ignore the global cooldown. In general, instant use abilities that does not make use of animation are good candidates here. Being able to use that panic heal or blink ability instantly or even when casting a different ability can make for an experience which feels more fair to the player. To achieve this, each ability has the option to ignore global cooldown and this can simply be toggled on or off in the inspector.

3.4 AI and Agents

The agents in the game are built up of several components. The main parts of the agents are sensors, actuators, brain and behaviours/actions which each serve their own purpose. Sensors are responsible for gathering outside information and feeding it to the brain. The brain digests information and makes a decision based on what it has gathered. Actuator is a framework for different methods for the output of the brain and is limited to smaller tasks like looking at a target or adding velocity to the character. Lastly, the behaviours connect the actuators together with animations and other events that are outside the scope of the actuators and allows the design of the AI to enable reuse of code.

The ability system described in section 3.3.3 is used for many behaviours and actions that the agent can make use of. One reason for this is that the ability system contains many of the needed functions from the behaviours and allows variations by using ability behaviours. Repeating tasks based on cooldowns and lockouts can easily be managed and code that can be localized won't clutter up other places. How much the ability system is used is up to the developer, but encapsulating functionality in the ability system will help in organizing the project and better allow reusability of components between agents.

Pathfinding

The game is played in 3D space and as such, we needed a pathfinding architecture to support this environment. Unity has built-in support for Navigation Meshes which has a huge advantage of being able to parse almost any type of mesh. It is split into two parts, one is responsible for rasterizing the world by dividing it into voxels and looking up hits to the triangles in the scene. All meshes that are supposed to be rasterized needs to be marked in the editor and then the navigation mesh needs to be baked. The second part is the runtime component being made up of static methods to do operations on the navigation mesh and the MonoBehaviour component, NavMeshAgent, that is a character controller with support for crowd handling, movement, etc. We opted to not use the NavMeshAgent component as it adds a lot of overhead to our prefabs and we instead create a PathfindingComponent that handles the path that is given by the static Unity class NavigationMesh. The PathfindingComponent will query the PathfindingManager class, which is a singleton component we created, for a path when it needs it. As such the PathfindingComponent will only keep control of the current path target and is responsible to point to the next target when it is reached.

Having a singleton or manager that handles the path lookups enables us to optimize and reduce the number of times a lookup is made. The start and end point of a path can be on any point of the mesh, but the sequence of nodes between them is always bound to the corners of the navigation mesh. This allows us to cache path requests in a look-up table that stores the triangles of the start and end point and the list of corners between. Later on, this system can be used to create a congestion map to reroute agents around heavily trafficked areas. This optimization is mostly needed when there are a large number of enemies, in order of hundreds, which will be a rare occurrence in this type of game.

Enemy interactions in ARPGs are often based on if they are within the cameras FOV (field of view) and the need for pathfinding is mostly to give agents more natural handling of the environment. If the agent loses line of sight to its target behind a corner a path can be generated to the target that the agent can follow. Instead of running straight into the wall or becoming unresponsive the agent can now chase down the player. The agents are given a target lifetime in the instance they go out of sight and will attempt to track the player using pathfinding until the timer runs out. This short term memory can then cut the chase so they don't follow the player indefinitely.

3.4.1 AI Design

To facilitate reusability and keep the design of the AI separate from the logic and behaviours we decided to create a visual node editor. The node editor will support two common types of AI structures, decision trees, and behaviour trees. The latter has in recent years become one of

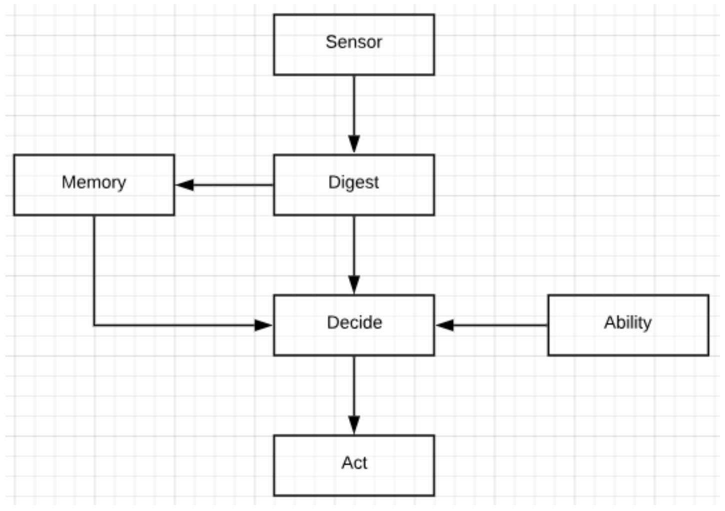


Figure 4: General overview of the structure and design of AI

the main choices to approach AI in games. It's a concurrent graph solver which means that it will not evaluate the whole graph and gives control to each individual node. This is different from a decision tree where the solver runs until it reaches a leaf every time the state is evaluated. Decision trees are commonly used as state machines. Both of these have advantages and disadvantages and can, in the end, be used together to combine their functionality.

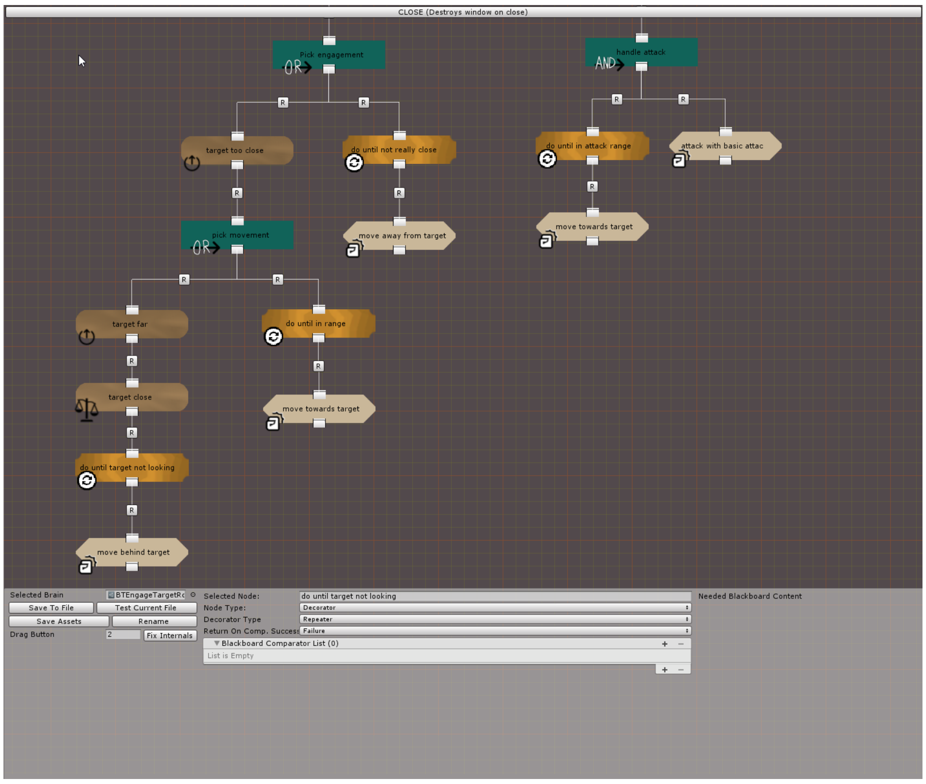
The performance difference between a behaviour tree and a decision tree will vary greatly depending on a lot of different factors. The size and depth of the tree can greatly increase the time a decision tree uses to reach an answer. This is not the case for the behaviour tree as it will run concurrently and control can be switched between different parts of the tree without evaluating the rest of it.

3.4.2 Node Tree Editor

To store the node trees we made use of ScriptableObjects, with one copy for the editor and one for the runtime component. Using different storage containers for the two versions of the node tree allows for some more freedom in the editor while also making it easier to optimize the structure of the runtime variant. A very simple linked list is implemented in the Node class which takes care of managing the ID and connections of the node. Runtime nodes are stored in the RuntimeNode class which will encapsulate all the different node types and the data they need in order to run. In the runtime storage, only nodes connected to the root node are stored and the nodes reference each other directly in the linked list to omit constant lookups of the graph. One limitation of using ScriptableObject and Unity serialization is that the maximum recursive depth is set to 7 as default. This will cause problems with direct references to other nodes so the editor variant will store a reference to the node IDs instead as to avoid the serialization issue.

3.4.3 Nodes

The nodes in the tree can return one of three statuses, running, failure or success. Depending on the node type the node can have zero or many children, but every node type is limited to one parent node. The main reason for this choice is that it is easier to implement and



Selected Brain BTEngageTargetRc
Drag Button 2
Selected Node: do until target not looking
Node Type: Decorator
Decorator Type Repeater
Return On Comp. Succes: Failure
Blackboard Comparator List (0)
List is Empty
Needed Blackboard Content

Figure 5: Example of how the node tree editor displays the tree and how the designer can modify the properties of a node. The current graph showed is part of the one that controls the rogue enemy type.

gives less opportunity for the designer to create loops or other undesirable situations. A tree structure also makes it easier to implement functionality to jump around the code. A node can be accessed on the way down (downward/below) and on the way up (upward/above) which is important to distinguish between different ways the solver is moving around the tree.

Node Types

Below is a general explanation of the use and design of each node.

1. Composite Node: A composite node can have one or more children. They will process these children in the specified order (In order or reverse) using the physical order (left to right) used in the node graph editor. They will process the status of the children in the current sequence position and either move to the next or cancel out. Composite nodes is a downward facing action, meaning that it will pass control upwards as long as it has not been accessed on the way down. The main reason for this choice is being touched on in the Operator Node explanation below.
   The composite node can be assigned entry and exit delegates that will be called whenever the solver enters and leaves the nodes. They are not required but are there to provide additional functionality and a fallback to make sure different values are set and signals are called.

Composite node sub-categories:

1. Sequence - The sequence node will run the next node in the sequence if the current

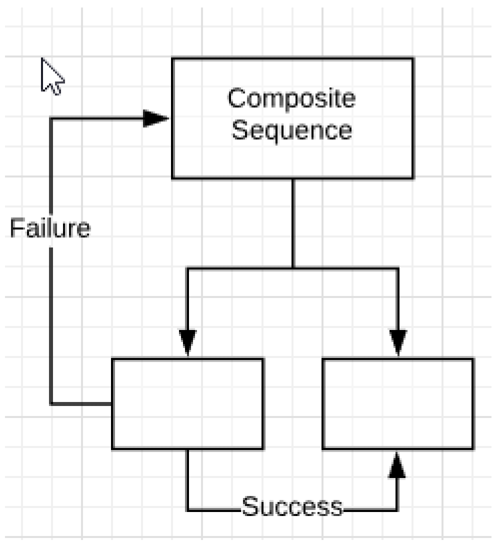


Figure 6: Composite Sequence Node Diagram showing the position of the current node as it passes through the sequence node

one returns a status of success. As soon as one of the children returns a failure status it will cancel out and give control to its parent. These can be chained together to create a sequence of actions that need to be performed only if the previous ones were successful.

2. Selector - The selector node is sort of the opposite of what the sequence is. It will

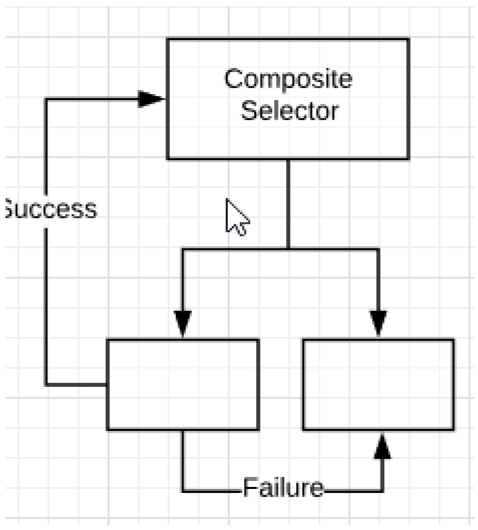


Figure 7: Composite Selector Node Diagram showing the position of the current node as it passes through the selector node

run the next node in the sequence only if the current one returns a status of failure. When one of the children in the sequence returns a true status it will cancel out to its parent. Selectors can be used to pick the first action, out of a group of actions, that is successful and run down that part of the tree. If any of its children are returning failure it will also continue on to the next sequence position.

3. None - Generally, there isn't a use for a none sequence, but it will allow the designer to close off a part of the tree from downward access. If a node tries to give control to a none composite, it will immediately return a failure and give control back to its parent.

2. Leaf Node: Leaf nodes are responsible for interacting with the agent's behaviour methods. In addition, they can be used to evaluate a game state or otherwise fulfill the role of both assessing the game state and make the agent perform a task. They can't have any children in the node tree editor and are thus the leaves of the tree graph, same as their name suggests.

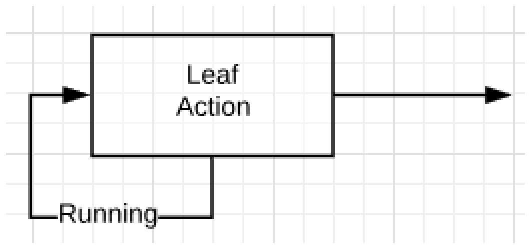


Figure 8: Leaf Action Node diagram showing how the node keeps control as long as it returns the running status.

1. Action - Connects the node to a function call from the wanted agent type. It contains function calls for initialization, execution, and exit of the node that can be bound in the node editor. An action node remains in control as long as it returns the running status and can return success/failure depending on the outcome of the action. If the initialize function returns failure it will also cancel the leaf node and return to its parent. All these delegates return a RunStatus and takes a string value as a parameter.
2. State - This node is only available for the Decision Tree and gives access to the different states that have been created for the state machine AI.
3. Sub-Tree - The sub-tree node contains no functionality during runtime other than to act as a marker for where the sub-tree starts. Sub-trees are stored separately in the editor but are combined into a single tree for the runtime component. The sub-tree node is still left in the runtime storage to enable runtime swapping of sub-trees in later iterations of the project.

3. Operator Node: Operator nodes fulfill more specific roles that can be complicated to implement using leaf nodes. They can have a single child or no children.

1. Comparator - Comparator nodes give access to the blackboard of the agent. It can keep a list of BlackboardItemComparators which is a metaclass to perform comparisons on BlackboardItems. It will run the comparators and return the result as a success/failure RunStatus. There are two different uses of the comparator node depending on if it is given a child or not. If it is given a child it will only enter it when the comparison return success. If it's used as a leaf of the tree it will return

    the result of its comparison to its parent. The functionality of comparators can be expressed through the use of action leaves but it will lead to a more complicated system to allow designers easy access to the blackboard.
    2. Interrupter - This node is only used in Behaviour Trees. Interrupters allows the upper parts of the tree to take control over a lower part. They will interrupt based on a comparator or action returning success. Another way to understand this node is that it is a local interrupter with authority over its children only. More on this below in section 3.4.4
    3. Global Interrupter - This node is only used in Behaviour Trees. Global Interrupters can't gain control from above and will only run its children if its list of comparators or actions is returning success. It will not interrupt its children if they are running and that is the only instance that a global interrupter will not take over control. More details on the behaviour of this node below in section 3.4.4

4. Decorator Node:
    1. Succeeder - This node will simply return success no matter what the children return as a status. The use case for something like this is limited, but can prove to be useful to enable the use of some behaviours in multiple situations.
    2. Inverter - Takes the result of its children and inverts it only if the status is Success or Failure. Success is flipped to Failure and Failure is flipped to Success.
    3. Repeater - This node will continue to run its children until the given list of comparators or actions is returning success. It can also be set to run until a given status is reached to allow the solver to stay on the current node until the ability is ready or some other event has triggered.
    4. Timer - A timer using the TimerCounter class can be used to run a section of the tree for a set time period. Similar functionality can be reached by creating a structure for it with Action nodes, but that tends to get quite tedious if you want to reuse it. It can also be accomplished by using repeater nodes and signal flags in the blackboard but requires the implementation of behaviours to make use of it. The timer node is a tree solver implementation for that "run for time" functionality and allows the designer to create duration based behaviours.

3.4.4 Tree Solver

A base class serves as the foundation to build the behaviour tree and decision tree solvers on top of which is the GraphSolver class. They contain some logic to initialize the solver, but specific implementations have to be made for both of the planned solvers. In addition, they have abstract methods for all the different node types described in this section. The base GraphSolver class will contain all the characteristics of a graph/tree solver to handle the different operations that are common between them. These operations include resetting the tree, clearing memory, finding a common parent and other utility to make the processing of the graph easier. 3.4.3.

Decision Tree

The general idea behind Decision Trees (DT) is to allow for the evaluation of a branching set of decisions or comparisons. For our use-case, we wanted the ability to evaluate different agent flags in order to reach a decision about the state of the game. It is more difficult to create variety and interactions with different situations as there is no elegant way to switch control between different parts of the tree. The main reason for implementing decision trees is that this was something that could be created alongside the node tree editor. The decision tree will be used together with a state machine that can be used to create a pre-cursor to finite-state machines. This can be used for very simple AI but decision trees are more useful as a component of a bigger system.

One big advantage of this implementation of the decision tree is that it only needs to run when new information is received from the sensors. There is no need for synchronization of wait functionality in order to keep the agent running while the brain is in the inactive state. A major disadvantage of decision trees is that its structure needs more planning and is harder to modify at a later time. Bigger decision trees will be hard to maintain and can prove difficult to debug as its size can become large pretty fast. This is one of the main reasons for moving on to Behaviour Trees (sec. 3.4.4) as the functionality in that architecture is a lot more flexible.

The Decision Tree mainly makes use of the node types Sequence, Selector, Comparator and State/Action and there is not much use of the rest of the nodes in this solver. The reason for that is that this is not a concurrent evaluation and as such will not have any need for concurrent behaviours.

Behaviour Tree

Behaviour Trees (BT) in the industry has a pretty loose standard and there isn't a right or wrong way to implement it. The main idea is that you implement a set of different types of nodes that can be put together in an adjacency or linked list in order to create more advanced behaviours. The two node types that appear in every implementation of behaviour trees are the composite and the leaf node. These two nodes alone can accomplish most tasks needed by the behaviour tree but to allow for better separation of the visualization and functionality some additional node types can be used. These are decorators, which alters the signal from its children, and operators which gives some additional functionality. The solver is located in the BehaviourTreeSolver class that inherits from the previously mentioned GraphSolver. The graph solver is initialized through the Init method which is required to prepare the RuntimeNodes for use. After this, the tree can be ticked forward using the RunGraphSolver method which is also responsible for running interrupters before the current node runs.

The architecture used by Chris Simpson in his article [4] on behaviour trees in games he uses three categories of node types, composite, decorator and leaf nodes. These three categories reappear in almost all behaviour tree implementation and form the core foundation of the functionality of the architecture. These three categories also reappear in the scientific paper [5] that utilizes the same core functionality but has a more advanced look at the different meta functionality needed to create a dynamic tree. Most importantly it touches on the aspects of reusability and makes use of Modular Behaviour Trees in its approach. Modular Behaviour Trees, or MBTs, is the idea that a tree can link to a sub-tree and even make use of different approaches such as FSM and Decision Trees in those parts. This can greatly increase the flexibility and allow the designer to seek the best architecture to solve an AI problem.

To get better insight into an actual implementation the documentation for NodeCanvas [6] was used. NodeCanvas looks a lot like an implementation of the paper [5] and contains many of the components described in that paper. It is a good starting point to get a direction to implement variables and signals to better allow for two way communication between the agent and the behaviour tree. Additionally, it helps form an idea of how the editor should look and what components are needed from it.

Behaviour Trees generally run on a predefined tick-rate and any node is in control as long as it returns the running status. To allow different parts of the tree to take over control from any other part of the tree a local interrupt and a global interrupt node is implemented. An important thing to note about the interrupt ticks is that these will take control right away. Let us say the behaviour tree has triggered an ability with a casting duration. We either have to stop the main tick or we can implement an override lock. The override lock can be set to true while an animation or some other event is happening, which will prevent any interrupters from taking over control. This flag is needed to avoid confusing structures to appear in the node tree editor trying to overcome that problem.

The behaviour tree makes use of a simple memory to store the running state of each node. A dictionary is preloaded with the ids of the nodes as keys and the RunStatus of the node. This enables the solver to be aware of which part of the tree is in control and to know if the current node got control from above or below in the tree.

Comparison

Table showing the positives and negatives of the different AI approaches:

Architecture	Good	Bad
State Machine	Simple and efficient Robust with less points of failure	Harder to extend later on Less organized than the others Harder to create variation and choice
Decision Tree	Support for branching choices Easy to create situational behaviours Better suited for smaller more confined behaviours	Hard to extend functionality later on Bigger trees are slow to evaluate
Behaviour Tree	Easy to modify and extend Allows for dynamically acting agents that can respond to a multitude of situations More efficient than Decision Trees	Designing BTs requires greater knowledge of the architecture Less efficient than State Machines

Table 1: Table showing the advantages and disadvantages of the different AI approaches explained above

3.5 Choice of technology

3.5.1 Unity Game Engine

Our main reason for picking Unity as a game engine/editor is that it has the most complete set of features. Not only does the game engine itself contain a lot of the base functionality we need but it also contains an extensive editor. It has the biggest community of all the game engines out there and most searches related to game development will result in a Unity related answer, even if the keyword Unity is not included. This almost ensures that we will never meet a problem that no one else has had and thus we can focus on the code we implement.

3.5.2 Store Bought Assets

The unity environment includes an asset store with a great range of different assets ranging from free to having a price. We chose to pick out a few of these packs and buy them for use in the game we made for the thesis. The main reason for this is to avoid the conceptual look and get more experience with using completed assets in the development process. By doing this we also get more time to spend on the different goals (sec. 1.2) we set for the project.

Figure 9: Example scene from the unity asset store for one of the graphical asset packs aCquired for use in this project [1]

To keep the atmosphere and aesthetics as coherent as possible we chose to go for a polygonal art style. The polygonal art style makes use of low-poly models and usually only an albedo map for textures. This results in a simple style that can still possess a unique aesthetic. Low-poly art is closely related to cartoony art so we have mixed these two categories and tried our best to keep the general feel of the assets the same.

Particle effects can be tedious to set up and requires a lot of time to get correct while maintaining a pleasing look. We chose to acquire a couple of asset packs that included a range of different spell effects ready for use. These pack only contains the visual assets and we create the systems that make use of these effects and combine them to create the spell effects we have in the game. The asset packs we use are making use of the CPU particles in unity and as such we are stuck with the slower CPU instance of particles. Unity has GPU particles in an experimental branch but we chose to not focus on converting the asset packs to the newer system as it would take a considerable amount of time. This will generally not be a problem unless there is an unreasonable amount of particle systems active at a time in the scene.

We chose to buy the animation assets separate from the models as that gave us a lot more options in terms of character variety. The humanoid animation packs contain all the different animations we need to create characters that can use everything from an axe to a bow. The animation packs we acquired for the projects

Figure 10: View of all the different weapon assets included in one of the asset packs [1]

3.5.3 Third Party Libraries

We used NaughtyAttributes [7] to give us access to a library of different PropertyDrawers and attributes we can make use of. Mainly it gives easy access to create buttons and special field handling without having to write a custom inspector utility for every single minor editor utility we want. Having access to the grouping of fields allows us to visualize the different variables in a more clean way. We can also add hiding of fields in the editor based on a bool, allowing us to hide functionality based on the meta information of the class instance.

uFAction [8] is another free utility library that adds a range of GUI utilities and serialization of Unity.Action and Unity.Func. This allows us to serialize these two classes that are wrappers for delegate functions, giving us access to link up events in the editor. It also includes a GUI wrapper for the editor that allows us to use layouts in the OnGUI function of PropertyDrawers, which makes the whole process of extending the editor with tools a lot easier.

The node editor, to create BehaviourTrees and other graph solvers, is based on code from Gram [9]. It only includes the basic framework for the grid, nodes, and connections between nodes. Acting as a rough starting point it will decrease the development time of the node tree editor and less time is spent on making the base functionality feel good. There is a lot of work added on top of this to add on functionality for different node types, saving the trees properly, duplicating nodes and a range of other smaller quality-of-life improvements. More on the editor is touched upon in section 3.4.2 and section 5.4.3

4 Development Process

4.1 Choice of software development model

The development time of the prototype is short and we wanted to utilize the time as best as possible. We found that an agile development model would fit our group the best and chose scrum as our framework for software development. With short feature periods and rapid implementation, we aimed to be flexible in what areas of the game are being worked on. Our plan was to have a weekly scrum meeting that combined both the review of the previous week and the planning of the next week. We skipped the daily meetings in favour of continuous communication. Whenever a problem or question related to the work came up we communicated via Discord to resolve it continuously. This seemed to fit our group the best as we didn't have to micromanage several smaller meetings to fit everyone's schedule.

In retrospect, we did not strictly follow the weekly scrum plans we set and sometimes the scope of the features were too big to complete in one week. As such many of the Scrum planning meetings will continue a previous week's features and gave fewer options for us to discern the work that was done from scrum to scrum. As a solution to this, we could have split each of these longer-lasting goals into smaller goals that might have allowed us to plan and synchronize some features ahead of time.

4.2 Tools used for project management

4.2.1 Git and Gitlab

Git

We made use of the different services available at Gitlab's website to manage our project. Gitlab has support for most of the management tools we need to coordinate our efforts. These tools include the git repository, Large-File Storage support, development boards, and issue tracker. Whenever we are working on a new feature a branch is created for it and work is done there to create a patch that is later merged with the main branch. All merges are done through merge requests on Gitlab so that every merge is properly tracked with a small description to explain it.

One of our goals for the project was experience with larger and production ready assets. These files generally take up a lot of space and Git is not the best at tracking changes in these files and tend to be pretty slow at handling them. To avoid merge issues in texture images, models and other visual assets we used Git LFS (Large-File Storage) to store these files separately from our codebase. Asset files are then stored in FTP-like storage separate on the server and the LFS plugin will automatically pull the needed files when a pull request is made. This feature needs to be enabled on both the client and server to work and as such, we needed a host with support for it.

Unity by default stores its meta and asset files as bytecode and as such has little compatibility with Git, and to better support version tracking we need to enable Unity to store those files as YAML. A big downside to this is that changes to these files are tracked by Git and Unity changes the order and content of these files with very small changes to the actual content of them. On top of that, the YAML files contain mostly internally generated IDs and can it can be hard to solve merge conflicts gracefully. Using YAML means that all the assets in the project at this time sums up to about 17 million lines which means that it will create a lot of unreadable changes in our commit history. It is still worth it in our case to use YAML to store asset files as it makes the process of merging a lot easier, and we can make use of other tools to hide YAML file changes when looking at the commit history.

Because of this, we found out during the development process that we should put as much of the data as possible in ScriptableObjects to avoid those annoying merge conflicts that are hard to solve. It doesn't fix all the conflict issues, but changing values like position or another property on the prefab will not cause those issues to appear. ScriptableObjects aren't serialized the same way as scenes and prefabs and are instead represented as a class in the saved structure. As such saving and loading of these objects will not have special handling and will not store the different fields and values as internal ids.

Project Management

Boards are used to keep track of the features we are planning and working on and are sorted based on a set of labels we have created. These labels are either different areas of the development, a priority or a combination of both so we can organize the different issues we put on the board. Another use of the issue tracker is to keep a list of bugs that are found but aren't resolved. This allows a central place to keep track of information related to the bugs and can make it easier for us to cooperate on solving bugs.

To keep track of every meeting log generated from the weekly scrum meetings we created a template in Latex that is stored on Overleaf (www.overleaf.com). Overleaf is an online Latex editor that allows for cooperative work on the same document similar to how Google Docs work. The use of a template in Latex fits our use case as we are also using it to write this thesis. The reports can be found in Appendix C.

4.2.2 Codebase Health

Keeping control of the content of the codebase is important and we chose to make use of Doxygen to generate the documentation. This requires us to create XML-style comments on all classes and class members describing its use and parameters. All public members should be commented even if their functionality is explained implicitly so that other members of the team can more easily understand how to make use of them. This is especially important in the different frameworks created so that it is clear how one should use it. Graphviz is used together with Doxygen to automatically create diagrams showing inheritance and other static information present in the codebase. Another huge advantage to using XML-style comments on class members is that the code editor can show tooltips with this information which is super helpful when developing code.

All members of the group use a version of Visual Studio (Code or IDE) to do the code development as they provide a range of different useful tools and extensions. Both of them provide a step-through debugger that can attach to the unity editor to allow for more advanced debugging with conditional breakpoints and stack-trace tracking. The Visual Studio IDE also contains more advanced functionality such as a better profiler than VS Code that can be useful for debugging certain problems. Static code analysis is done with OmniSharp which is a .net package that is included in VS IDE, but VS Code requires an extension to make use of this tool.

4.2.3 Game Engine

The game engine serves as the base for our scope of the thesis, and Unity felt like a natural choice for this project. It has most of the groundwork laid out which allows us to focus on the gameplay and unload a lot of the asset management and pipeline process over to Unity. Having access to higher level abstractions of lower level systems gives us more room for prototyping and rapid implementation even though it can lead to a less optimized end result.

Unity is constantly under development and new feature patches are at times released monthly. To avoid having to update deprecated functionality or redoing parts of the game whenever a new patch occurs we chose to stay on version 2018.3.4f1 and not update the editor during the duration of the project. To come to this conclusion we had a look at the Unity Roadmap (https://unity3d.com/unity/roadmap) to establish which parts of the engine will change that is relevant to our project. Most of the features were not related to our project and the ones we were most interested in were served through the Package Manager and didn't need a patch to make use of newer versions.

4.2.4 Cooperation

Not all problems that need to be communicated fall within the scrum meetings. As such a tool to communicate between team members is needed so we have a common place where general information, discussions, and questions can be posted. We chose to use Discord for this project as that allows us to create our own server, for free, with support for different text and voice rooms. All three group members are also Discord users outside of the project so it came as a natural choice over other options such as Slack or a forum solution. We had an internal discussion if we wanted to keep those logs public or not, and ended up keeping them private as they provide very little addition to this thesis.

4.3 Graphs and statistics

show activity on git throughout dev. process. pinpoint areas around milestones

5 Implementation

5.1 Networking

5.1.1 NetworkIdentity

The NetworkIdentity is based on a GUID (also known as an UUID [10]); a 128-bit number which when generated is for all practical purposes considered unique. Due to the standard implementation System.Guid not beeing serializeble, we implemented a wrapper Networking.Guid which is serializeble and as a result can be stored in unity's scene files. This is required to make a object placed in scene have the same id on all clients loading that scene. When the GUID is set, the NetworkIdentity sets the GUID on all NetworkBehaviours on the Gameobject except the last byte in the id is set as the NetworkBehaviour's index on the GameObject. The NetworkIdentity and the NetworkBehaviours are then registered in the NetworkManager allowing them to receive data sent to their id.

5.1.2 NetworkManager

ClientBehaviour

The ClientBehaviour component manages a Lidgren NetClient, a NetPeer limited to one connection which has methods for sending data to the one connection. The ClientBehaviour recieves all inncomming data and handles debug messages, errors and connection status changes. It also adds a number of callbacks for UI interaction.

ServerBehaviour

The ServerBehaviour component manages a Lidgren NetServer, a NetPeer with unlimited connections and is able to send data to a specific connection or simultaneously send to all connected clients. The ServerBehaviour handles handing out client ids used to send data to specific clients and relaying broadcasts from a client to all the other connected clients.

Scene management

When the server wants to switch scenes it's broadcast to all clients. When a client is finished loading the scene it'll send a message to the server which keeps a list of all the ready clients. When all clients are finished loading the server will broadcast a message letting the client know they can now run the scene.

Instantiating

Instantiating a prefab not stored in the scene requires a way to identify which prefab to instantiate as well as synchronizing the GUID. The NetworkManager stores a list of all prefabs that can be network instantiated and any client can instantiate a new object by broadcasting the prefab's index in the list and the GUID that should be assigned to the NetworkIdentity on the newly instantiated object.

Lidgren NetOutgoingMessage & NetIncomingMessage

Data sent and recieved by Lidgren.Network is stored in NetOutgoingMessage and NetIncomingMessage respectively. Both inherit from NetBuffer which essentially is a byte array with methods for reading and writing data where the data must be read in the same order it was written.

5.1.3 NetworkBehaviour

NetworkTransform

The NetworkTransform synchronizes the position, rotation and scale of a GameObject. It'll periodically check each value and if the difference from the last broadcast reaches a set threshold it'll broadcast all the values over the threshold. On the receiving end it'll interpolate the values resulting in smoother movement, however it does not provide prediction or lag compensation.

NetworkAnimator

The NetworkAnimator synchronizes the parameters of a Animator component as well as setting triggers. With parameters of unknown types and to avoid sending unnecessary data it has a bit field and periodically checks all the parameters and marks the ones with a difference over the threshold. It will then broadcast a message containing the bit field as well as the dirty parameters in the same order they have in the Animator. This allow the reliever to iterate over the bit field and check the parameter type in the Animator when reading the data.

5.2 Application

5.2.1 Blackboard

The blackboard content itself are stored as BlackboardItems in the BlackboardStructure class to allow usage of the blackboard outside of the GameObjects. It stores the content in a dictionary that is preloaded with wanted information when the structure is initialized. It is very similar to any other list implementation except that there are two ways to retrieve values from it. The user can either request the BlackboardItem directly through the GetItem method, but the actual values are accessed the same way as an array. Square brackets are overridden to make it easier to access the value of the items but this is a read only action. The values of the BlackboardItem are set through the use of the Set method, mostly because the setting of a value also requires the Guid of the setter if the blackboard is set to local write only. Keeping square bracket override will therefore avoid situations where this rule can be broken.

For the entities in the game to make use of the blackboard they can use the BlackboardUser component class, attaching it to the GameObject. This class contains a method, SetupEnumInBlackboard, which will take an enum and create BlackboardItems that are filled in the blackboard. Using enums for some of the known and shared data makes it easier to use that information other places without preexisting knowledge of the content.

5.2.2 MethodReference/FieldReference

There are several ways to bind methods to variables with varying degrees of support for serialization, usability and how the stack is called. C# have extensive support for delegates, both through a wrapper class and using the delegate keyword. On top of this Unity have their own implementation of events that doesn't function exactly like C# events. UnityEvent is a wrapper class that enables the designer to bind methods during the design of the game while C# events are populated at runtime and can only be invoked locally by the containing class. The biggest downside to UnityEvent is that it can't have a return value and as such it is mostly used to issue messages to its registered listeners.

To allow us to bind method with a return type during the design a wrapper class was created. The wrapper class for methods is in MethodReference which is a generic implementation that can return a value while taking an argument. It has a generic way to invoke the method so each wrapper class that inherits from MethodReference only need to contain the data specific to the return/parameter type. To allow for serialization it will only store two strings in order to work, the class name and the method name. Storing references to other objects is generally a bad idea and will more often than not lead to issues down the line. These issues are related to lost references and it will be less generic.

At runtime it will make use of the GameObject it is attached to and Reflection in order to bind the delegate to an active object. Doing constant reflection lookups can be performance heavy and after the first time the method is invoked the MethodReference class will cache the delegate so it can be called directly later on. This is pretty much necessary as the time it takes to invoke the delegate will be much higher if it's not done.

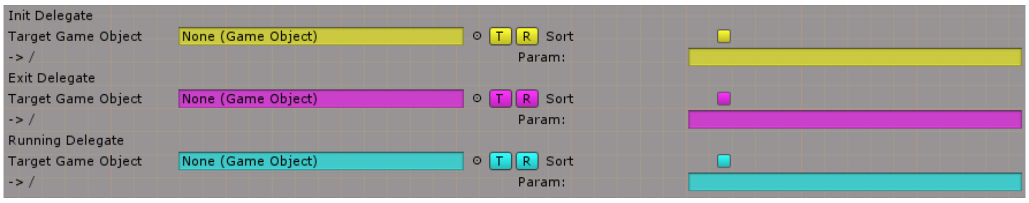

Init Delegate
Target Game Object None (Game Object) T R Sort
-> / Param:
Exit Delegate
Target Game Object None (Game Object) T R Sort
-> / Param:
Running Delegate
Target Game Object None (Game Object) T R Sort
-> / Param:

Figure 11: Example of how the MethodReference PropertyDrawer looks in the action leaf node. The "R" button will open a dropdown that lists all methods (with RunStatus as return type and string as parameter) in classes with the MethodReference attribute

A downside to the generic implementation of MethodReference means that it requires more work in order to create a property drawer for the class. A property drawer is a needed functionality for MethodReferences as we want to be able to bind methods in the editor and as such a drawer for each class that inherits from MethodReference is needed. An implementation of a property drawer for MethodReference is found in MethodReferenceDrawer.cs. It currently contains some debugging information to in order to test if the method invocation works but the "TargetGameObject" field does not need to be filled in order for it to work at runtime. Binding the field to a GameObject will only list methods from components listed on that GameObject, so it can be used if the designer only wants methods from that object.

Reflection is used in order to search the assembly to find all the appropriate classes and method names. It makes use of Linq in order to do a parallel lookup to help with performance on larger assemblies. The helper methods to do this lookup is located in ReflectionHelpers.cs and an example of an assembly lookup is showed in figure 5.2.2.

FieldReference was the initial idea to bind variables in the blackboard and node tree editor but it does not support recursive lookup and as such wont have access to variables in a stored class instance. This makes the functionality of the FieldReference less useful but it can still be used to access global variables from other objects.

Listing 5.1: Code snippet showing use of reflection to find methods in a generic way from the current assembly. It makes use of SQL style Linq syntax that was added in .NET 7. and selects public non-static methods.

```
var methodswithreturntype =
    from a in AppDomain.CurrentDomain.GetAssemblies ( ).AsParallel ( )
    from t in a.GetTypes ( )
    let attributes = t.GetCustomAttributes (typeof (T), true)
    let methods = t.GetMethods (BindingFlags.Public | BindingFlags.Instance)
    where attributes != null && attributes.Length > 0
    select new { Type = t.Name,
        Methods = methods.Where
            (mi => mi.ReturnType == typeof(U)).Select (mi => mi.Name) };
```

	Static	Dynamic
Good	More control over individual interactions. Easier to optimize for performance and crowds	Real physics interactions imitating real life. Most of the framework for general interactions are there.
Bad	Requires more work to enable dynamic collision handling.	Requires more work to find the correct parameters for intended functionality. Harder to tweak for performance later on without implementing custom handling.

Table 2: Table showing a comparison between static and dynamic physics.

5.3 Physics

The character controller and physics are all contained within the same class, PhysicsComponent. It contains all the necessary handling of collisions in order to keep the character within our world constraints. These constraints are position and rotation which the rest of the component is responsible of handling. Both of these are locked within the component and can only be accessed through methods in the PhysicsComponent class.

To check for collisions we have made use of the static class to the physics engine, which means that we aren't using any dynamic physics. Using static class ensures that we have more control over how the character interacts with the world and other physics entities. To allow for use of dynamic triggers the GameObject containing the PhysicsComponent will also have a kinematic rigidbody and a capsule collider. See figure 2 for a more direct comparison of dynamic vs static physics.

When doing static collision checking there are a few areas one have to consider in order to make things appear smooth. Since we are doing everything in 3D we can't just rely on raycasts in order to check for collisions and keep entities from entering each other. All characters in the game makes use of capsule colliders but they also have to interact with other types of colliders. Raycasting in the direction of travel will lead to situations where the sides of the collider crashing through the sides of other colliders. To avoid this a combination of both raycasting and capsule overlap have to be used to ensure these situations don't incur. Unity does have a static method to check overlaps of capsule colliders so we don't have to spend time designing a custom solution for this problem.

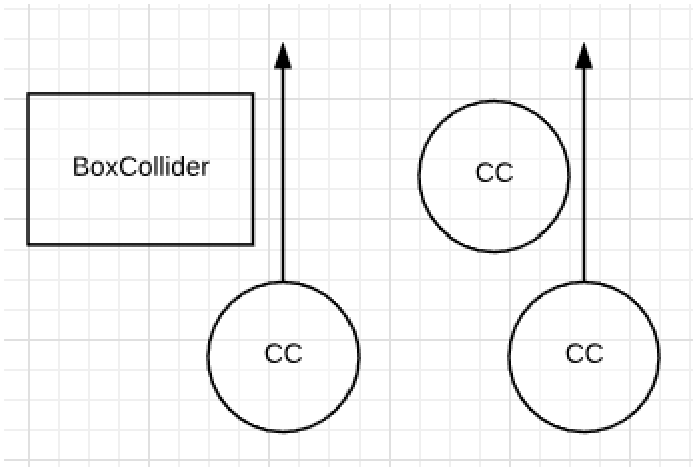
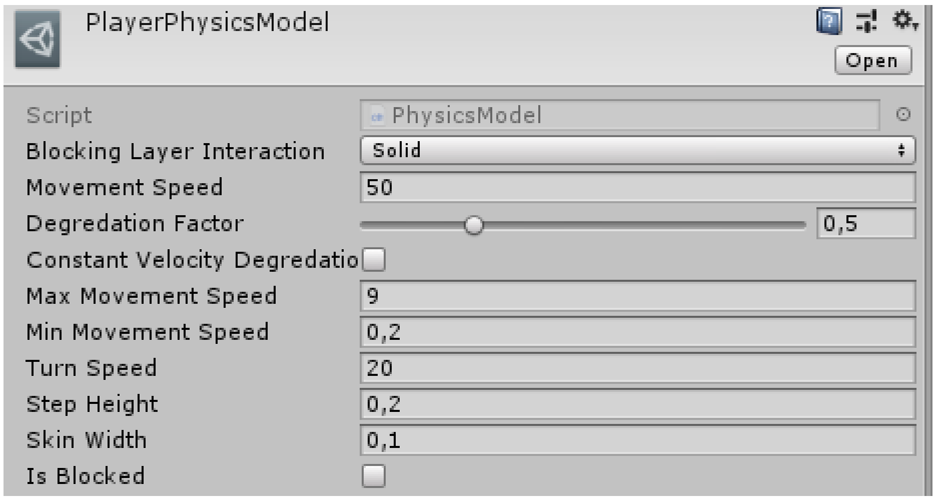


Figure 12: Diagram showing two instances of raycast collisions where the side of the colliders would still collide even if the main raycast didn't.

Unity have support for checking overlaps over a distance for capsule colliders but this is more performance heavy than using one capsule overlap and one raycast. Usually this will be more than enough to design a character controller while maintaining collisions that resolve in a smooth manner.


PlayerPhysicsModel

Script	PhysicsModel
Blocking Layer Interaction	Solid
Movement Speed	50
Degredation Factor	0,5
Constant Velocity Degredatio	☐
Max Movement Speed	9
Min Movement Speed	0,2
Turn Speed	20
Step Height	0,2
Skin Width	0,1
Is Blocked	☐

Figure 13: PhysicsModel which defines the different physical properties of an actor in the game.

To store the different values for physics used by each actor ScriptableObjects were used. These values are stored in the PhysicsModel class which allows us to have different physical properties for each actor. It contains many of the common properties that are used in most character controller implementations in order to create an actor that approximates a real world actor. The physics does not make use of weight and there is no explicit bounciness. There is still a hidden force being applied when two actors collide where the velocity of the entity is rotated away from the colliding target. This gives an implicit bounciness in order to keep characters from getting stuck on each other and makes it easier for the actors to manouver around in crowds.

Velocity and Force Events

Moving an entity using the PhysicsComponent is either done through the Velocity variable or by using the ForceEvent system. ForceEvents can be compared to RigidBodys impulse force where the mass of the entity is not used in the calculation. They therefore only makes use of time, force and direction in order to calculate the total amount of movement for that event. All force events are added to a queue or stack and one of these queues exists for each of the layer interactions we have implemented. This is stored in a dictionary with the layer mask as key and a queue as the value. As of right now this can be optimized a bit and we can save some space and calculation time if we multiply force and direction together when the force event is first added.

All force events are resolved outside of the velocity vector and will not add or remove from it. This is because force events are designed as an outside force and we still want the character to keep its own velocity. Mainly to keep the velocity clean for use in animation and other events. Each queue is summed up individually and any force event where the time is up is not added back to the queue. Before the summed up force event is applied to the transform a capsule overlap check is done and a raycast is made to check if the event collides with another entity.

Functions only used only by the player or agents are encapsulated within their own extension class inside a relevantly named namespace. The namespace encapsulation of static extension class allows us to only show those methods whenever the namespace is included in a file. This removes a lot of clutter from the code completion and generally makes the base class focused on the base functionality it is supposed to provide. A major downside to this is that it makes it harder to organize and it is not always supported from static analysis tools and might not be represented correctly in document generation. Because of this static extensions should be kept to a minimum and contain as little code as possible.

5.4 AI and Agents

All agents makes use of two main components in order to work, AgentComponent and BrainComponent. These two classes only contains the base functionality needed from the agents and further specialization for specific agent types is done through inheritance. The two bases contains no animation or effect hooks which should be done in a child class in order to avoid unnecessary clutter. This enables us to reuse functionality such as pathfinding and physics across the different agents without having to interface with those systems every time we create a new agent type.

The main idea behind the AI framework is to bind actions together using delegates making use of an event based system.

The data for agents are stored in ScriptableObjects located in the classes AgentModel and HumanAgentModel. AgentModel is common between all agents in the game and contains the fields that every agent type will make use of. AttackTimer can be used as a master override or lockout for abilities and attacks and is used differently for the wolf than the humanoids. The enemy front angle variable decides the radii that is considered a frontal attack for the AI. HadTargetLifetime controls how long an agent will remember its target which can be used if we want an agent to search for a recently seen player. EngageRange defines the agents radii which it will engage the target within, as long as the target is in line of sight. AttackRange is the range a target needs to be within for the agent to be able to attack, but with the use of

Behaviour Trees this can be omitted to have a bigger range of attack distance depending on the ability used.

The humanoids in the game is currently the only ones making use of the ability system. This is because the ability system in its current state was not ready until after the quadrupedal/wolf was created and because of time restraints we didn't get time to make use of this for those agents. In the HumanAgentModel abilities can be set up from three different categories: Offensive, Defensive and Support. This is mostly used to discern between different abilities from a design aspect, but can also be used if we want the agent to use a random ability from one pool specifically. When the health pot section was implemented the ability system didn't support multiple uses and is mostly an artifact left behind from then.

5.4.1 AI Design

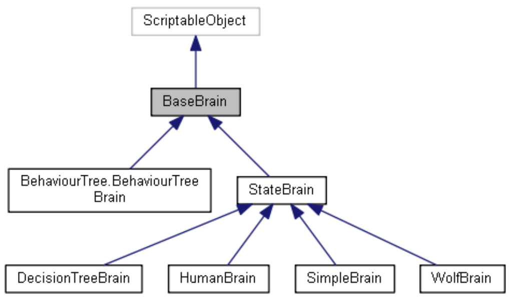


Figure 14: Inheritance diagram showing the different brains that the agents can make use of.

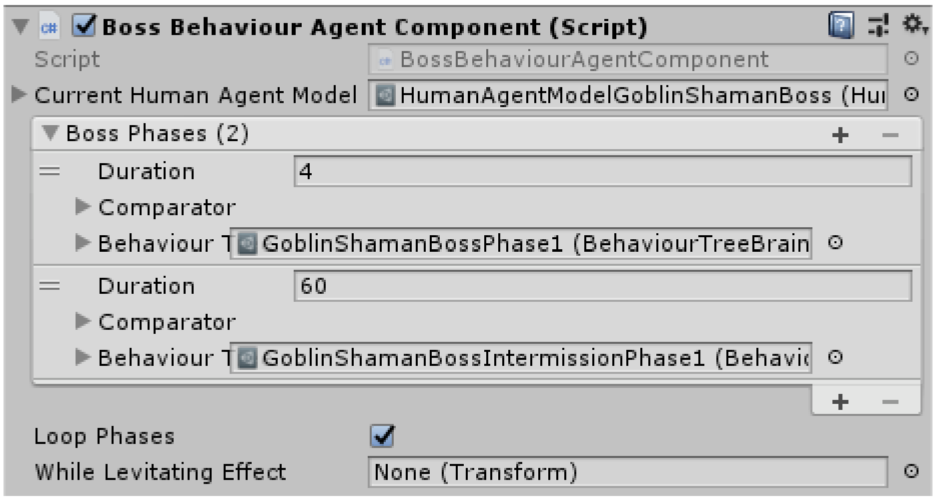
To serve as a base for all brains in the game a BaseBrain class is created. This will only contain the name of the brain and nothing selse, used as a way to identify all brains. Behaviour Trees is the only brain type that is directly inheriting from BaseBrain. StateBrain servers as the foundation for brains that makes use of a state machine in order to function which is where the Decision Tree is inheriting from. This will make it easier to expand the framework later on when other agent types are added that requires more functionality on top of the currently implemented brain types.

State Machine

State machine agents have to inherit from StateAgentComponent as that contains the necessary functionality to handle a state based brain. States can be registered by adding them to the BrainState enum and then adding them to the dictionary in the StateAgentComponent that binds BrainStates to Actions. State Actions are composed of three methods, entry, exit and execute where execute is invoked continuously and the other two are invoked once on entry and exit of the state. Having those two methods than runs once becomes very useful in order to set specific variables and other information based on the state that is running.

5.4.2 Boss

The boss can be configured to make use of different phases that can be set to loop or stop at the last phase in the list. Phases can be switched based on time or if a comparator is successful. All of this framework can be implemented directly in a behaviour tree but doing it this way makes it easier to manage. It will not switch the phase automatically and this have to be done through the linked behaviour trees. When the boss component wants to switch


Boss Behaviour Agent Component (Script)

Script: BossBehaviourAgentComponent

Current Human Agent Model: HumanAgentModelGoblinShamanBoss (Hu

Boss Phases (2)

Duration: 4

Comparator

Behaviour T: GoblinShamanBossPhase1 (BehaviourTreeBrain

Duration: 60

Comparator

Behaviour T: GoblinShamanBossIntermissionPhase1 (Behavi

Loop Phases: ✓

While Levitating Effect: None (Transform)

Figure 15: Image shows how the boss can be configured to make use of different phases that switch based on time or a comparator.

phase it will set a signal in the blackboard and that can be picked up by the tree. The tree then have to signal the boss component through the behaviour method ReadyForNextPhase when it is ready for the next phase. After the switch to a new phase a setup signal is set in the blackboard which allows the behaviour tree to do some work at the start of the phase. Currently this is used for one of the prototype bosses in order for it to run to the center of the boss room and levitate in the air during its intermission phase. After the setup is done it has to signal the component through the SetupDoneSignal behaviour method that is defined in BossBehaviourAgentComponent. There is a lot of improvement to be made to this but it's simple and fulfills its task without a lot of complicated logic. The "WhileLevitatingEffect" property is an artifact that haven't been removed yet as the levitation part is now handled as an ability in order to make it reusable and to avoid a lot of clutter in the class.

5.4.3 Node Tree Editor

The grid, nodes and connections are all handled in the NodeEditorBase class which the NodeGraphEditorWindow class inherits from. Edges and vertices are stored in two seperate lists and the connections are linked to vertices through its id. RuntimeNodes are stored as a separate list and only the editor nodes store which RuntimeNode it is connected to. All three of these lists are stored in the NodeEditorBaseStorage class together with the asset id of the runtime storage to allow saving and loading of different trees. RuntimeNodes in the editor storage are not connected together, which only happens after it is saved to runtime storage.

Node Tree Editor Feature List

Before the tree can be used it has to be prepared and saved to the runtime storage. Preparation involves connecting together the nodes as a linked list and ordering the children of composite nodes in the correct order. The same saving method is used for both decision- and behaviour-trees but they store their content to different storage containers. All runtime storage containers inherit from the BaseBrain class which only contains the name of the tree and variations for different runtime solvers are created from it. Behaviour Trees store to the BehaviourTreeBrain and Decision Trees to the DecisionTreeBrain. One important task of the

Table 3: Table showing the current features implemented for the node tree editor

Feature	Description
Save/Load	Saving and loading of node trees in editor and to the runtime version of the tree
Rename	Renaming trees, there are multiple files that needs to be renamed which requires a custom renaming function
Copy/Paste	Duplication of nodes, either a single or multiple. Both nodes and connections are copied.
Selection Box	The selection box is a draggable area that can be used to select multiple nodes.

storage is to bind the BlackboardItemComparators and MethodReferences to active gameobjects during runtime which is necessary for them to work.

The node tree editor makes use of the Blackboard in order to get access to information about the agent or other entities. The initial plan was to make it so you have to add items to the node trees blackboard first and those values can be used in the designing of the tree. This is an alternate solution to using enums and is less hardcoded than the current solution, but it would require more work on setting that information from the agents themselves.

5.4.4 Nodes

1. Composite Node: Internally each composite node has an integer that keeps track of the current child node that is selected. This is cleared out whenever a node above it in the tree takes over control or if the last child node is completed. It is important to keep this stored as long as a child node is in control to know which node is next or if it has completed its run.

    1. Sequence -
    2. Selector -
    3. None -

2. Operator Node: The comparator node is the only one that is handled the same way as other node types and the interrupters will not be evaluated during regular tick-through.

    1. Comparator -
    2. Interruptor - Interrupters runs before the main tick and are stored as a compressed "banana" stack. It runs from the bottom up starting at the first parent interrupter of the current node until one of them returns success or it has reached the root. If it reaches the root that means no interrupters were currently triggered. The way this is implemented means that every interrupt node above the current node have to be ran every main tick which can lead to performance problems on big and complex behaviour trees. Because of this the use of interrupters should be kept to a minimum as there usually are ways to design around this.
    3. Global Interruptor - Global interrupts runs before both the main- and interrupt-tick and are stored in a priority list that runs in descending order. They are subject to the same performance issues as the local interrupter but the problem can appear much faster because they run before every main tick even if the current node is in a completely different part of the tree. Because of this functionality the global interrupter node is not a concurrent solution and as such it is best to use this type

of node only when absolutely needed. Other concurrent alternatives using other node types should be evaluated first as that provides the smoothest response from the AI and avoids sharp, quick changes that can lead to jittery response from the AI.

3. Leaf Node:

    1. Action - The action node has a straight forward implementation as it will never have any children. It is mainly responsible for executing the init and exit delegates whenever the running delegate is not returning the running status, before and after respectively.
    2. State - The state node will call on the SetState function of the StateBrain class and the StateBrain is responsible of swapping control to the new state. Doing this it will call the exit delegate of the previous state and then swapping control to the new state while calling the init delegate for that state. It's a simple, yet effective way of controlling the state of the agent.
    3. Sub-Tree - The functionality of the sub-tree node is done in the editor when the node tree is saved to runtime. At runtime the sub-tree node will only pass on the control to its child or parent depending on if it exits or enters the node. Currently there is no functionality to replace sub-trees at runtime, but this is possible to accomplish in a relatively easy way since the sub-tree node is still present. The nodes below the sub-tree node would have to be disconnected and the new sub-tree connected which would only require swapping which child node it points to.

4. Decorator Node:

    1. Succeeder - The succeeder node will simply take the RunStatus of the child when given control and then set the RunStatus it passes to its parent to Success no matter what the child gave.
    2. Inverter - Inverter only affects the Success and Failure status and will flip the result to the other option and pass it to its parent whenever it is given control from the child.
    3. Repeater - The repeater node will pass control back down the tree until the given RunStatus is passed from its child or until the given action or comparator returns the Success status.
    4. Timer - The timer node works much the same as the repeater and will continue to run the child node no matter the status returned (except Error) until the timer is finished.

5.4.5 Graph Solver

Instead of hard-coding the node graph for each individual solver an abstract class named GraphSolver is created. This class contains all the base functionality to traverse the tree and initialize the graph for use in runtime. It contains an abstract method for every node that is described in the section above and in section 3.4.3 to make it easier for the developer to make use of the different node types.

GraphSolver is inheriting from the IGraph interface, located in IGraph.cs, which is there to provide the different methods needed to set and retrieve the files. In the same file there is a static class containing different debugging methods that are useful when developing the tree. It's an extension class which means that it will extend anything that uses the IGraph interface with those methods. Mainly it has a method to print the tree breadth first showing the depth and children of each node, except for the leaf nodes. This makes it easier for the developer to get an overview of what the tree looks like and a quick way to check if the tree is in the correct order. Debugging is touched upon further below in this section.

Decision Tree

Decision trees have a straight forward implementation and only makes use of operator, composite and leaf nodes. It only has access to State Leafs, Sequence/Selector Composite and Comparator Operator nodes and will not makes use of any of the other sub-types of the different node categories. The main reasoning for this is that the rest of the node types will not serve a purpose in a non-concurrent algorithm. The decision tree solver will run until it reaches a leaf node and will not evaluate the rest of the tree after that, if that leaf node returns success status.

Since the solver stops at the first leaf node it reaches it is important to order the decision tree such that the nodes or comparators that run more often are reached first. This is mostly to make sure that the tree is as optimal as possible and as few unecessary nodes as possible are ran. Currently this have to be done manually but we can make use of other optimization techniques in order to optimize this automatically. We could record data from which nodes run the most often and create a set of weight that represents this order to build a more efficient lookup table. This is currently not implemented as it was outside the scope of the projects lifetime and we would not see much performance benefit in our use case as it is mostly used for simpler AI and decisions.

Behaviour Tree

The current implementation of Behaviour Trees can be extended to be ran as a recursive algorithm, but the recursive step is removed to allow for stepping through at a tick-rate instead. This tick-rate can be set to something else than Δtime but in the current implementation it will tick at the same rate as Δtime. This can be used for performance tweaking and also to give the AI longer response time so they act more like the human brain. Humans themselves usually have about 50-400ms response time depending on their experience with a situation and as such this could be used to give a more natural and fair AI behaviour. Interrupters can also make use of a different tick-rate than the main tick but this depends a lot on the behaviour tree layout and would require more tweaking to make it feel right.

Currently both the interrupters and the main tick is running on the same tick-rate and interrupters always run before the current node. The call graph diagram in figure 16 shows the general flow from the entry point where the tree is ticked from, the RunGraphSolver method. One thing to note is that the ResetMemoryLower method is mainly called from the individual node methods and is not always called from the SetCurrent method. Main reason for this is that setting a node doesn't always need to reset the memory of the lower parts of the tree as that would remove the information needed by composite/operator nodes to run. They need access to the child nodes RunStatus so it is important to pay attention to this in order to not break the functionality of the algortihm.

The call graph for RunGraphSolver does not show the order of execution only the path the code takes once it has reached that point. To give a better idea of the order it starts with the RunGlobalInterruptors method and then runs FindFirstParentInterruptor if the global interrupters did not activate. Global interrupters are stored as a priority list and will run in descending order until one of them interrupts or the end of the list is reached.

Local interrupters run up the tree from the first parent interrupter of the current node until one of them interrupts or the root of the tree is reached. As mentioned in section 5.4.4 local interrupters are stores as a compressed banana-stack that is extracted from the list of runtime nodes when the GraphSolver is initialized. It stores the references to those nodes and as such there won't be a negative performance or storage impact from doing it this way. After the first parent interrupter node is found it will send that node reference to the RunInterruptorsRecursive method that will run the interrupter. Local interrupters are stored in a tuple data type where Item1 is the node and Item2 is its parent interrupter. A thing to note about tuples is that they store their content as values and not by reference. As such the current implementation will require more space in its current state and this might be something that can be improved in the future. An upside to this is that we don't want to change the content of the interrupters and using tuples protects the values of the actual nodes.

Testing and Debugging

There are several ways to implement debugging of a node tree. For this implementation we found that the most efficient way was to create unit tests. A small set of unit tests were written for the node tree and nodes in order to make the debugging process easier. These tests makes use of the Test Runner package that is included with the Unity Editor and it makes use of the NUnit framework and allows us to create both runtime and static tests. A major downside to the testing framework in unity is that all data that you want to access with it needs to be available in runtime. This will add a lot of unnecessary data to an eventual deployment build and we have to make sure to remove or omit these files before such a build.

The Test Runner framework contains three modes of operation we can make use of. The three modes are MonoBehaviour, static and step-through tests that allows us to test a wide variety of situations. For the Nodes/Behaviour Trees the step-through method is used as we want to be able to check that we are in the correct position every time we tick through the tree.

The main functionality of the tests is to make sure each individual node and the tree as a whole is working as intended. For each test a small confined node tree is created that contains the nodes needed in order to complete it. The test makes use of Coroutines in order to step-through and simulate how the solver would behave during runtime and allows us to better test the complete running order of the node(s). To create these tests we first have to analyze how the created tree will function according to the specification we created in section 3.4.3.

5.5 Abilities and Ability System

5.5.1 Casting and animations

The animation state to use for a cast is stored in the ability base class. When a cast begins we transition to the animator state directly from code using a "Animator.Crossfade" call. This make the animator transition nicely to the animation. A float representing the normalized time of the cast is passed to the animator from the update loop while casting. Normalized time in this case is a number between 0 and 1 representing what point in time the animation should display. The normalized time of the animation will always become equal 1 as the

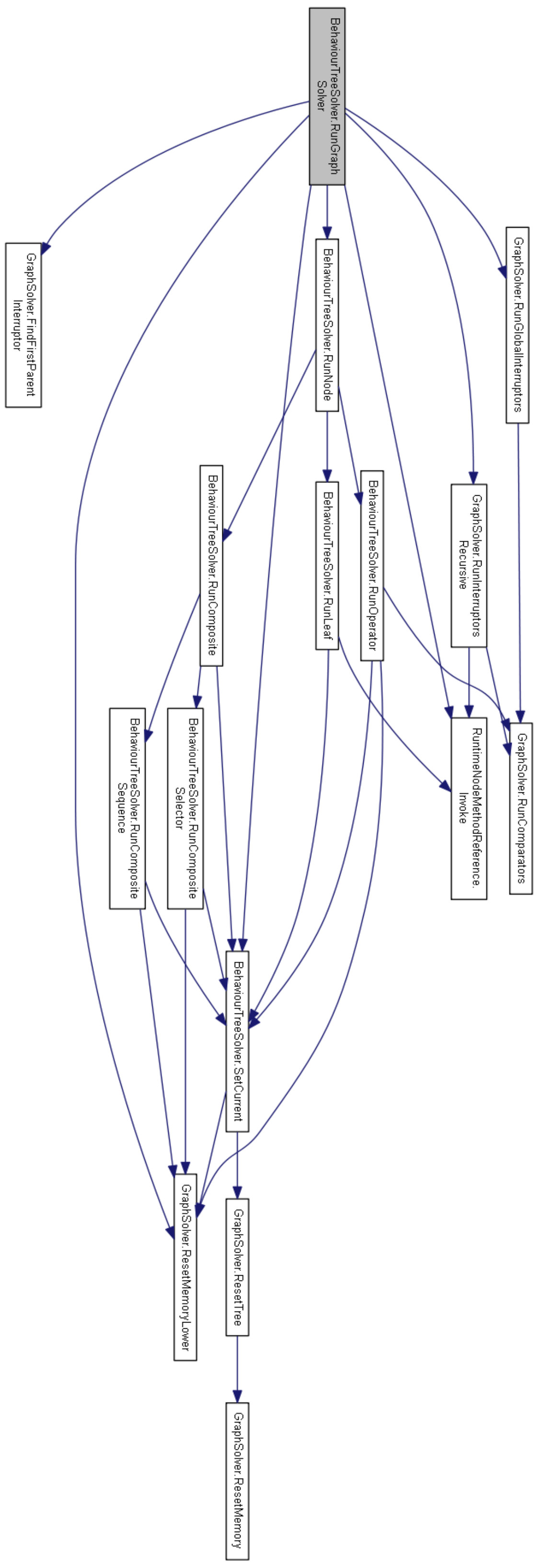

BehaviourTreeSolver.RunGraph Solver
GraphSolver.FindFirstParent Interruptor
BehaviourTreeSolver.RunNode
GraphSolver.RunGlobalInterruptors
BehaviourTreeSolver.RunComposite
BehaviourTreeSolver.RunLeaf
BehaviourTreeSolver.RunOperator
GraphSolver.RunInterruptors Recursive
BehaviourTreeSolver.RunComposite Sequence
BehaviourTreeSolver.RunComposite Selector
RuntimeNodeMethodReference. Invoke
GraphSolver.RunComparators
BehaviourTreeSolver.SetCurrent
GraphSolver.ResetMemoryLower
GraphSolver.ResetTree
GraphSolver.ResetMemory

Figure 16: Call Graph for the Behaviour Tree Solvers RunGraphSolver implementation.

cast completes. This means that the animation must always end on the frame the cast will happen. This results in having to split a "full casting" animation into two parts. one which ends with the cast and one which plays the motions after. Changing the frame in which the cast completed therefore requires editing the animations. This is one of the downsides to our approach which would not be present if using animator events, where simply moving the event along the timeline would do the job.

5.5.2 Using Abilities

As mentioned in section 3.3.3, we needed to avoid coupling abilities with players. As such, player input and keybindings are not part of an ability. The ability base class has fire and release virtual functions which in the players case translates to key up and key down respectively. However responsible for translating input into using abilities is the ability slot the ability is currently attached to. Using an ability is rather straight forward. The following line of code will try and access the AbilityUser component on the game object we are calling from. If the component exists it will proceed to use the ability passed in by name.

```
GetComonent<AbilityUser>()?.Activate("Ability Name");
```

We were a bit on the fence about using ability names as identifiers, but decided that it is was an acceptable approach in our case. The main downside of this is that no AbilityUser can have more than one ability of a specific name at the same time. This would be particularly annoying if we were to have multiple ranks of each ability. The advantage is that it is easy for human eyes to comprehend what is going on when using specific abilities from code, something AI will do a lot. using enums would not work in our case because of the multiple different abilities we create out of each type. We do however have a more robust solution in mind. a unique string or number would be generated using either using GUID or a custom system. This identifier would be combined with the ability name to form a final identifier that would both be readable and unique.

5.5.3 Locking of abilities

It is possible to prevent an actor from using abilities for a certain amount time or until a condition is met. The way we implement this is by having a simple int representing how many locks are currently applied to the ability user. If this number is anything but zero, abilities will not be usable. A public method is provided for adding locks and takes one of unity's yield instructions as a parameter. It will increase the lock counter and start a coroutine, when the routine yields the lock is removed.

5.5.4 Damage and Healing

Derived ability types that deals damage or heals contains a base damage/healing value. We use this number together with the character's stats to form a final value. Currently this is calculated in the ability itself as it allows each type of ability to create its own formula. The formula is hard-coded. This works relatively well but has one drawback; Two abilities formed from the same ability type will always use the same formula. Instead of hard-coding the formula, having some sort of system that can build a formula based on data the game designer can input, would have been better and would resolve the above mentioned problem.

5.5.5 Melee weapon animations

For basic attacks we make use of animator events. Attack animations fires an event when the attack actually happens. The timing is different from each animation, but for a horizontal sword slash this would generally be when the sword is pointing forward. When we receive this event we spawn a slash animation using the position and rotation of the character's hand. Initially when we had rather low attack speeds, this worked perfectly. However, when attack speed was raised, a rather big issue surfaced. The animation would play so fast that the event would be fired when the sword was way further in the animation than it was supposed to. The issue here is that animator events fire on the first update frame after the point in the animation the event is set to has passed. With a high attack speed the animation would simply move too far each frame for that to be precise. This resulted in the slash effect spawning incorrectly as the hand would not have the correct orientation and position when we accessed its data. This sounds like something that can easily be solved as The animation event happens at a specific frame in the animation and all the positional data of the hand for that frame exists in the animation. Unfortunately, this data cannot be accessed at runtime. There is two solutions to this problem. One is to play all animations with attack events at a low speed while loading and then store the positional data of the hand for each event to use later. This is likely the best solution but would require a good amount of work to set up and would increase loading times. The other solution and the one we used is to manually type the data and have the animation events pass it as a parameter. This did not require much work in our case but would take a lot of time if were a lot of different animations

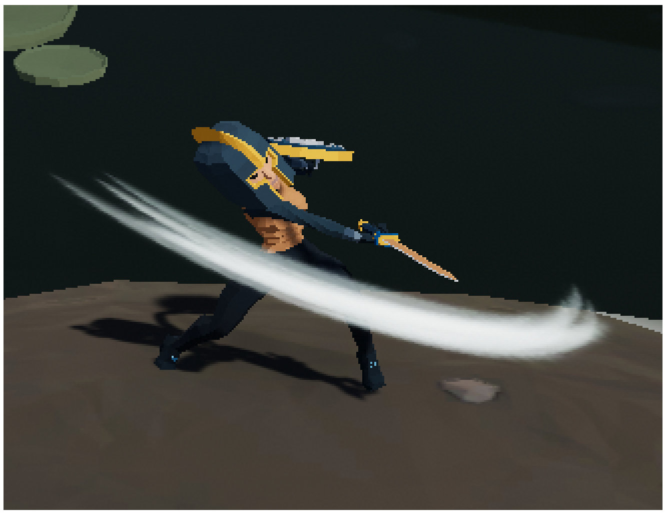


Figure 17: Example of a sword slash

5.5.6 Status Effects

Status effects are what players usually refer to as buffs and debuffs. These are either positive or negative effect that the character will temporarily receive. Our implementation makes use of a base class; StatusEffect. As many of our base classes, it is derived from ScriptableObject. A status effect can be either a buff or a debuff, this is simply decided by an enum member. It also contains an enum deciding if any crowd control should be applied. This currently includes stuns and slows. A field representing the amount of crowd control is also provided, but this will only be visible in the inspector if slow is selected and in that case will represent the slow percentage. Status effects make use of System.GUID to generate unique id's. A convenient thing with scriptable objects is that any field declaration with a definition will be evaluated upon creating an asset. This is an advantage over using monobehaviour and prefabs where this is evaluated runtime which would require some system to keep id's consistent and permanent. The Status effect base class can be used as is for effects that only applies crowd control. For more intricate effects, classes deriving from it must be created.

To manage status effect for a given character, we implemented the StatusEffectComponent. This component is attached to every game object that should be affected by status effects and contains every effect currently active on that entity. A character can have multiple of the same status effect active at the same time. However, only one instance of a given id can be applied by the same character. This means that if two players applies the same debuff to an enemy, the enemy will have two instances of that debuff. On the other hand, if the same player tried to apply it twice, it would only receive one instance, alternatively it would add an additional stack to the specific instance if the status effect allows multiple stacks. When a character reapplies a status effect the duration is refreshed.

5.6 Character System

5.6.1 Talents

The Talent Class

Talent is an abstract class and serves as the base class for every talent. It derives from ScriptableObject and contains data such as how many points can be spent on the talent, how many are currently spent, and the icon to display. It also provides derived classes with abstract and virtual methods to override.

TalentNode

TalentNode derives from Monobehaviour and has two main jobs; To act as monobehaviour proxies for talents, as well as connecting talents together. Talent nodes is what the user will actually see when looking at the talent tree. TalentNode implements Unity's IPointerClickHandler interface to handle the player input. To link talents together into trees or other structures, some way of linking nodes together is needed. We achieve this by having each node contain a list of parents and children. For the player to be able to select a talent, one of it's parents must be fully decked out. The exception to this is if the node has no parents, in which case selecting the talent is allowed. For the player to be able to remove a talent point, the talent must not have any active children. If however an active child has another fully decked out parent, it will be allowed. With this implementation and the fact that talent nodes exist in the GameObject hierarchy, no manager class for the nodes are necessary. Talent trees can be constructed in the editor by placing talent nodes into a prefabbed game object representing the talent tree and assigning talents to them. Adding parents and children to the node can also be done. Arrows displaying parent and child relationships are automatically created runtime to save the game designer some work.

Talent Types

We decided to implement two different types of talents. Both of those are classes derived from Talent and they cover a wide array of use cases.

- Modifier talents are used to create talents affecting stats. Things like increased attack speed, bonus critical strike chance and movement speed goes into this category.
- Ability behaviour talents can modify abilities by adding ability behaviours described in section 3.3.3. Examples of this could be a talent that makes your melee attack apply slow, your dash stun enemies it passes through, or giving an ability an extra charge.

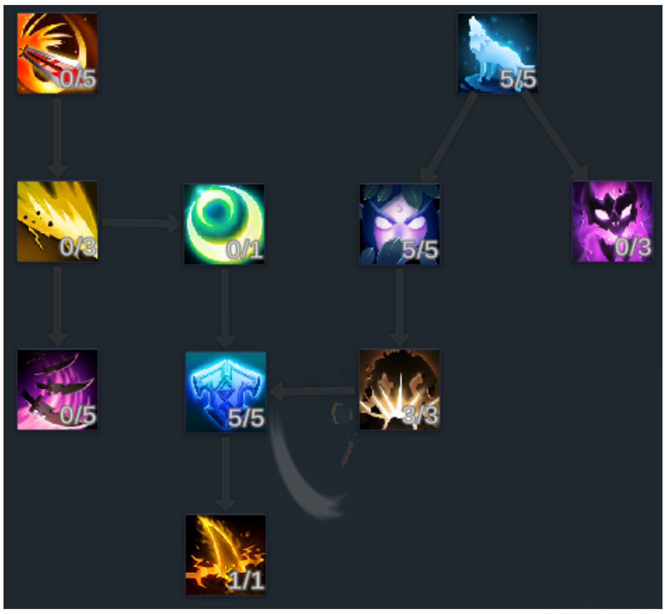


Figure 18: Example talent tree we created

5.7 UI

5.7.1 Drag and drop

The drag and drop system is built directly on top of the container system allowing it to interact with any class implementing IContainer, as a result the inventory and character equipment has no dedicated UI code and are displayed by a DragAndDropContainer instead. In the future it could be used form treasure chests and even a store with little added code. All DragAndDropContainers are registered in the DragAndDropManager. When the user starts draging an object from a DragAndDropPanel the object and the container it's in is registered, and when the object is dropped on another panel the DragAndDropManager calls the ContainerManager to handle the transaction.

5.7.2 Nameplates

We decided to implement nameplates to display important information regarding enemies. A simple way of handling nameplates in unity is to have an individual canvas containing a nameplate attached to each character. This approach comes with some drawbacks, particularly in the performance department. Our implementation makes use of a single canvas that every nameplate share. The downside of this is that unity won't automatically handle the positioning of them. To correctly position the nameplates we translate the position of the character from world to the screen space in the nameplate's update loop. We do this by using unity's Camera.WorldToScreenPoint function. To manage nameplates we created a component called NamePlates. Each client has it's own instance of an game object with this component as well as a canvas attached. It contains functionality such as adding, removing and hiding nameplates. Currently we use nameplates to display healthbars and status effects, but the idea is to also display things such as the name and level of the character. Status effects can be toggled on and off. Buffs are represented with a green border and debuffs with red. By default, buffs and debuffs have separate anchors but we also made it possible to anchor them together with debuffs appearing after buffs.

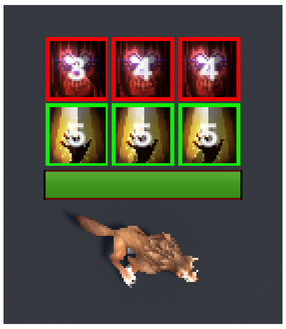


Figure 19: Example of a nameplate

5.7.3 Combat Text

Combat text was implemented to give the player visual feedback of damage and healing done. We did this by creating a CombatText component which is attached to players. When a character takes damage or receives healing, an event is sent through unity's SendMessage method to the character who performed the attack. Data of the damage/healing event is passed along with it. The combat text component catches this event and translates it to combat text if the player receiving it and the client matches. A single canvas which ignores raycasts is used to draw all combat text.

6 Deployment

6.1 Unity Deployment

When compiling a deployment build in unity we only have access to the content in the UnityEngine namespace and the .NET framework. If we try to build the project with code that references the UnityEditor namespace it will fail. This isn't really a big problem as most of the functionality contained in the UnityEditor namespace is not relevant to the deployment build. We still have to make sure to encapsulate all this code in a compiler preprocessor whenever it is used for debugging or other utility that is used during runtime. This is functionality already present in Unity and as such we only need to include the preprocessor symbol. An example of how this is used is shown in code listing 6.1.

Listing 6.1: Code snippet showing use of if preprocessor directive in C#

```
#if UNITY_EDITOR
    // Code to be omitted from runtime compilation
#endif
```

The rest of the process is mostly handled by the UnityEditor internally and we only have to register the scenes we want to be added to the deployment build. As long as we don't use any third party library that is not supported across platforms, building to any of the major platforms should work without any additional input from us. Since we make use of reflection we cannot deploy using Mono as that build target does not support Ahead-Of-Time (AOT) [11] compilation but Unity also supports ILL2CPP for those situations. ILL2CPP is the most optimized build target so this would be our choice, even if we didn't make use of reflection.

Another thing to take into consideration is shader support on the different platforms. We make use of mostly "desktop" shaders and as such we do not support targeting of mobile platforms or older consoles that lack support for never GPU frameworks. In the project plan, we only planned for targeting Windows, OSX, and Linux which all have the required shader support as long as the user has the appropriate hardware. As such we never worried about support for other platforms or alternative controller inputs as that would have complicated the development process a lot.

6.2 Binaries

Since the core aspects of our goal were to create a framework for creating an ARPG we didn't get time to create a stable and representative binary of the prototype. As the last weeks of the projects lifetime were used on writing the thesis we didn't get time to fix some essential problems related to the prototype before the deadline. Because of this we chose to not include a binary for our project as that would not be a good way to show our project. As such, we instead created a series of smaller GIFs and videos to show off the different areas of our framework. On top of this we have included a series of photos in appendix D to show areas that didn't need a video segment or to better show of the individual parts from the videos.

Copies of the videos and images are appended to the zipped file that came with this thesis report in case the links die.

6.2.1 Video Examples

Remember to set quality of video playback to 1080p60 or else the quality will be too low to actually see what is going on. Video clips were downscaled from 4k because of upload bandwidth which is the reason for the lack of sharpness in text elements as well as the tiny UI.

Node Tree Editor Example

This video example shows how the node tree editor works and presents some of the features that are present in it: https://youtu.be/uAJ5gG868DY

This video shows the current implementation of recursive node dragging. Dragging a node will also move its children nodes. Currently nodes selected with the selection box will not be dragged together. https://streamable.com/d28s8

Affixes and Equipment Example

This video clip shows the editor tools to set up custom equipment and how to add affixes that can spawn on a randomly generated item: https://youtu.be/AjYjdcWoKmM

Character Controller and Abilities Example

This video example is from the user test prototype we made showing how the character controller moves and interacts with the environment and how the abilities on the warrior class works. When applying the weapon buff you can see how the cast time of the ability affects the duration of the casting animation: https://youtu.be/yKeqrGGk3oU

Ability Configuration Example

This video clip shows how we can configure abilities and abilitybehaviours in the editor. This is done directly on the ScriptableObjects: https://youtu.be/HzVP4DY_Mk4

UI Example

This video clip shows the different UI elements in the game and how the inventory and character equipment menus work: https://youtu.be/AUFwXUzGpMc

Loot, Chests and Equipment Stats

This video clip shows how loot spawns from chests that can be picked up and equipped. It also shows how gear with different types of stats will affect the characters stats in the character menu. https://youtu.be/NPnz6adhCaA

Enemies and Interactions

This video segment shows most of the different enemy types currently implemented and how they interact with the player. https://youtu.be/Kd9xFj7YaBQ

6.3 Source Code

The source for our project only includes the code we have written ourselves. The main reason for this is that a lot of the assets we have used are under copyright and as such, we could not include those in the published source. The accompanying zip attached to this thesis report will include the folders containing the source code we have implemented ourselves. The source code itself, found in the source_code folder, is cleaned of all unity and git related meta files through the use of the "remove_meta.ps1"(lst. 6.3) powershell script found in the same folder. This is done to remove any unneeded files as they aren't really relevant and leads to a better experience when looking at the source code.

Listing 6.2: Powershell snippet found in remove_meta.ps1 to clean up any meta files from the current folder recursively

```
Remove-Item * -Include *.meta, .keep, .git -Recurse
```

7 Testing and User Feedback

7.1 Local and Internal Testing (QA)

During the different early stages of development, most of our testing revolved around local integration tests. This means that we would test that the functionality we have implemented worked without network synchronization first and then, later on, we would implement the synchronization and then test that it works. The main reason for this testing mentality is that the networking was implemented alongside the rest of the systems which made it hard to pre-plan what parts we wanted to synchronize.

After the weekly scrum meetings, we usually merged the different features that were ready for merging with the master branch. After we were done with the merging we would go through the different functionality we had implemented and checked that it still worked. This was done in order to catch bugs and errors that weren't previously present as soon as possible as it made the debugging a whole lot easier. Instead of not knowing exactly where the issues originated we would have a clearer understanding of what parts could contain the bug which made many of the smaller issues easier to fix.

The last part of our general testing was to compile the deployment version and see if that was successful. Most issues related to deployment compilation was using code and frameworks that were only available in the editor. These parts would then have to be encapsulated in a compiler preprocessor so that the Unity build pipeline could ignore these parts of the code. Not waiting until the end of the development period to do this would ensure that the project actually built successfully and we wouldn't end up with a large number of places we would have to fix after the fact.

To enable the testing we created a user test scene to only focus on the specific parts we wanted to test. This means that the dungeon and level design we have implemented didn't go through a testing phase and as such we don't have any outside feedback on that part. Since the testing world was kind of bland and uninteresting it can take away from the tester's experience and make the whole experience feel more like a development world. This will vary greatly on the type of tester and their interest in this type of testing which can reflect on the answers given in the feedback.

7.2 Player Test/Functional Testing

We went for an unsupervised functional player test where we sent out the playtest binary to different testers. The testers were picked at random with different knowledge of the game's genre and game development in general. This means that the feedback we got was spread over different levels of interest which provides less specific feedback. The upside to this is that the feedback we do get will touch upon different areas we might not have thought of at first. From our perspective we have a lot of experience in these types of games and the varied feedback can provide some more insight into the parts that didn't fall directly in our circle of attention.

7.2.1 Did you find the UI intuitive to use?

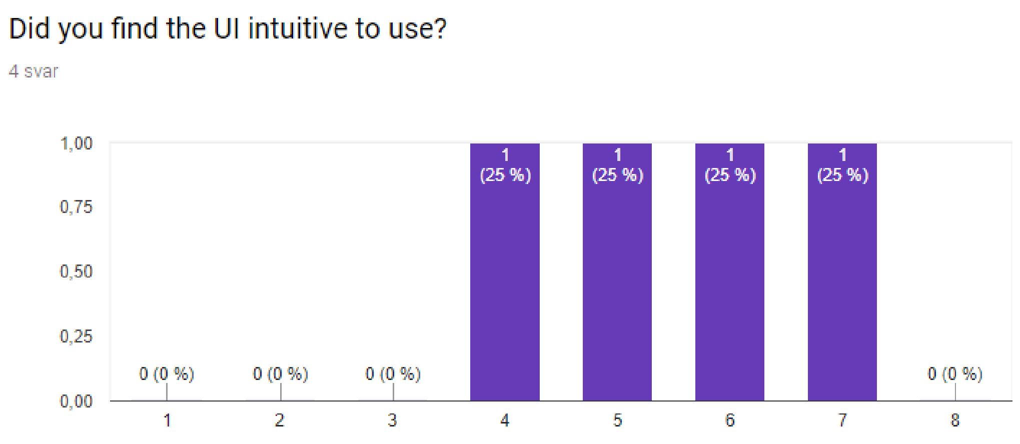


Figure 20: Overview graph of feedback to the "Did you find the UI intuitive to use?" question

The main feedback on the UI was related to a lack of a shortcut bar for the different menu windows in the game. A few of the testers weren't aware of the inventory and character menu which means that they would not immediately be aware of the loot system. Loot spawns on the ground but since they can't inspect it after picking it up they kind of missed out on the loot the first time around. This issue came forward in other parts of the UI too where it was hard for the tester to be aware of the talent menu and as such, it had a compounding effect. This is something we should have been aware of before the testing client went out and made the test less effective than we wanted.

Since we made use of a pre-made asset pack that included all the UI elements our UI felt complete and had a finished look even though it lacked a few critical areas for a new tester.

7.2.2 How does the character movement feel like?

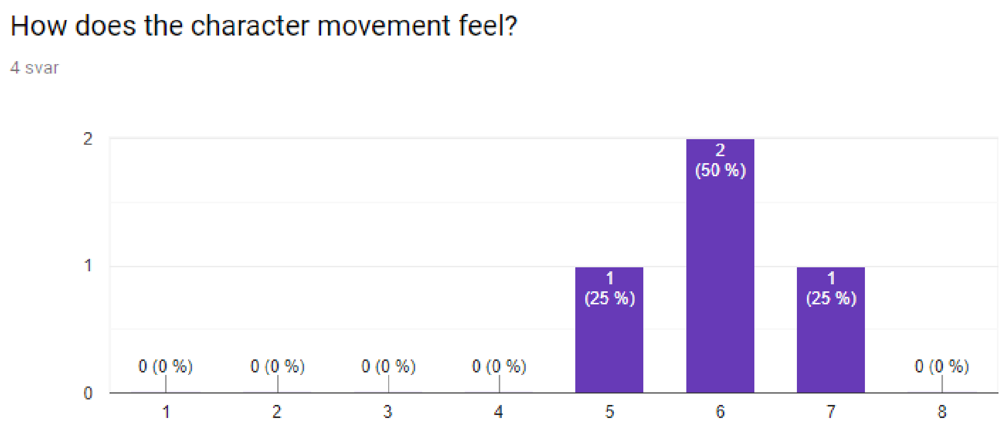


Figure 21: Overview graph of feedback to the "How does the character movement feel like?" question

The controls of the character weren't immediately known and as such the tester would have to learn just by playing. If we would have added a menu with an overview of how the controls work before the test started it would be easier for the tester to get into the game. More time was spent on actually getting the knowledge of the twin-stick controls which could take away from the value of the testing.

Other than this the testers reported that the controls felt smooth and it didn't hinder them in their gameplay. The biggest issue was that aiming at a location without solid ground would turn the character in the same direction which would confuse them and it would take away from the experience where they would focus more on if that was a bug and how to avoid it.

7.2.3 Did you find the abilities fun to use?

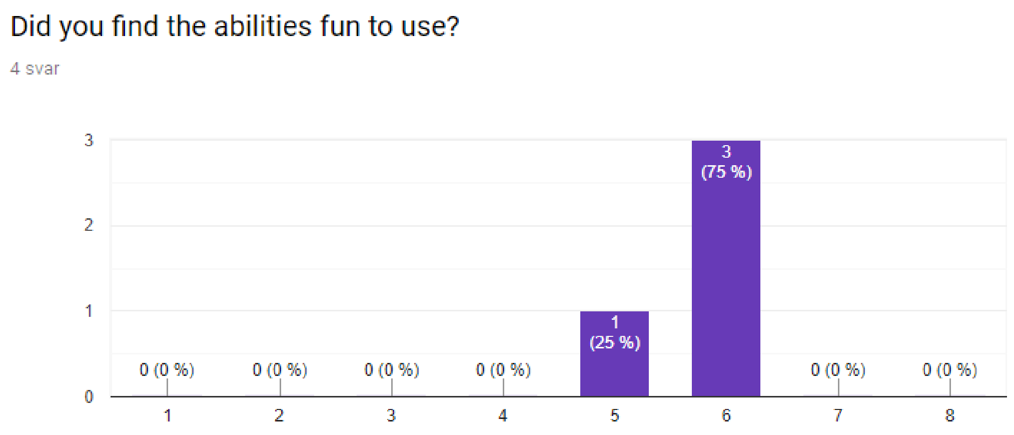


Figure 22: Overview graph of feedback to the "Did you find the abilities fun to use?" question

The testers found most of the abilities cool but a lack of proper balancing means that some abilities were used more than others. As such it didn't really give the tester a proper feeling of the abilities and they often ended up using one or two abilities instead of the whole arsenal. If we had spent more time in our internal testing on the balancing it would be easier for the tester to provide better feedback on the different abilities and talents.

7.2.4 Did the talents add any interesting effects?

The talents did add a lot to the abilities but the enemies were balanced around a character without any talent points placed. As such the player ended up extremely powerful and often just one-shotting the enemies as many of the talents were meant for a character in higher level against higher level enemies.

7.2.5 Was the enemy interactions with the player challenging?

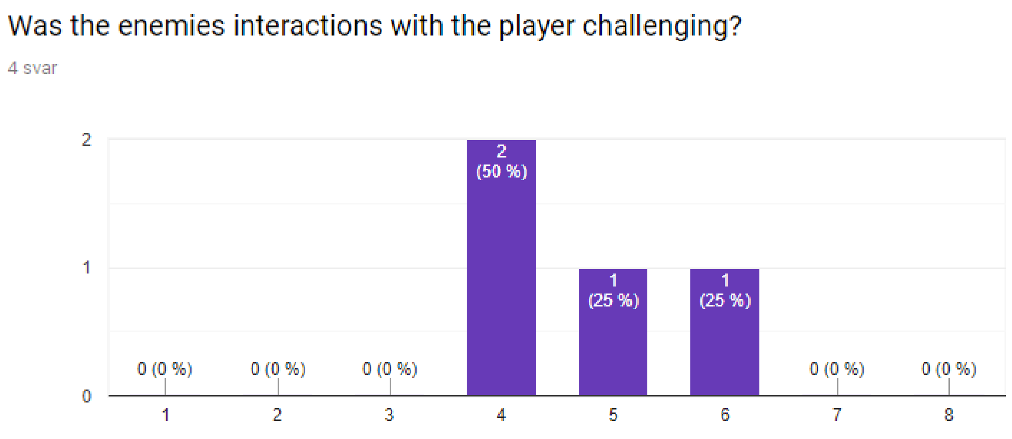


Figure 23: Overview graph of feedback to the "Was the enemy interactions with the player challenging?" question

Since the enemies were a little under-scaled the testers didn't feel that the enemies provided enough challenge. It's mostly related to learning how the warrior class works and how to overcome the enemies. We could have spent more time on the balancing for the test or added more enemies so that the testers would have to be more aware of the things around them.

7.2.6 Is the variety of enemies interesting or does it feel the same?

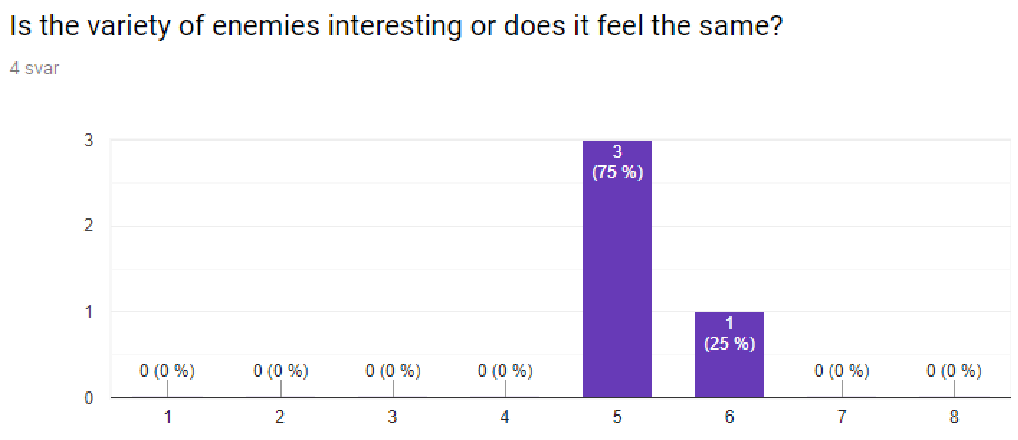


Figure 24: Overview graph of feedback to the "Is the variety of enemies interesting or does it feel the same?" question

From the feedback we did get it seems they found the variety in enemies in the middle of the road. We could have added more variety in terms of abilities making use of the same behaviours by creating smaller variations. It still gives us some insight into what kind of changes should be made in the future. By making use of ability behaviours we could have more easily added those smaller variations and is something that would be required in a more finished game. As of now it still points to that we were moving in the right direction and given more time we would have been able to move further in that direction.

7.2.7 Did the loot dropped by enemies give a feel of reward?

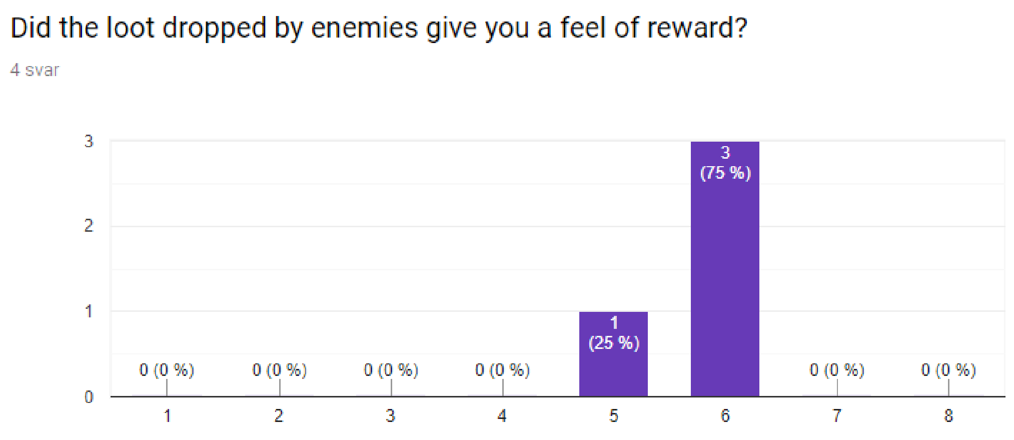


Figure 25: Overview graph of feedback to the "Did the loot dropped by enemies give a feel of reward?" question

Most testers found the loot interesting but because the test was so short it was hard to get a feel of the reward from them. Getting a good piece of gear at the end of the test didn't give the player time to experience that reward and as such, they didn't get enough time to give proper feedback on it. Testers that were familiar with the ARPG genre could still see the opportunity of the reward but they could not give feedback on how that affected their gameplay.

7.2.8 Do you have any other feedback you want to add?

We didn't get many answers on this question which might be related to how the test was performed. Most of the areas we wanted to cover already had relevant questions and the test didn't really cover much else. This might be an indication that we should change up how the questions are asked or how we present the content of the test in the future.

7.2.9 Test Conclusion

In the end, we should have run a couple of test phases in order to implement the minor features we didn't immediately think of. Since our group didn't have much experience from user testing this would have helped us greatly in order to provide better testing feedback. As such the effectiveness of this testing phase wasn't as great as we would have hoped and in the future, we should make sure that all the minor issues had at least a temporary solution in place.

The main issue was with the shortcuts for the different menus and this would have easily been solved had we been more focused on the testing phase. This is something we as a group have learned a lot from and should have done differently to get more appropriate feedback.

Adding better scaling for the abilities and talents in relation to the enemies would allow the tester to better experience their effect. Being super powerful is fun but it doesn't really provide the player with a challenge and makes it hard for them to establish the different utility they provide.

8 Discussion

8.1 Development Decisions

8.1.1 Issue boards and project management

Most of our use of issue boards on Gitlab involved planning future features. In hindsight, we should have spent more time on updating this board and making use of it actively in the development. A lot of the issues we did put on the board were broad in their description and splitting those up into smaller tasks as we went along would give us a better overview of the status of the project. This is something we should have spotted earlier as it gives much better insight for ourselves and in the case that a new member joins the team. Right now the issues on the board are most relevant to the current group and any future developers that want to make use of it would need some time on the source code first. Without deeper knowledge of what is actually implemented and the status of those implementations, it will be hard to get the proper information on what to do next.

In addition to this, we should have used the board more for bugs during development. Right now we don't really have a central repository for the smaller bugs in the game and an outside user would have to look through the source code for comments that point to the bugs. This is something we would have changed for future work and it is hard right now to know if an issue is related to a bug, a lack of implementation or if it's actually intended behaviour. We as a group can recognize most of these problems as we were the ones implementing the features but from a project management perspective, this is a bad way to handle this.

8.1.2 Using Latex and Overleaf for writing the thesis

Latex is very powerful in that when we are writing we don't need to worry about meta-information and we can focus on the actual text we write. Even though it still requires a lot of work at the end of the writing in order to position figures and other content where we want them it's still easier to use than word-processors like Microsoft Word. Having a template as a foundation meant that we didn't have to spend time putting together the different packages and the general layout of the thesis. This was a great help as it can be hard to create a finished product that looks and feels like an academic paper.

Using Overleaf was a great choice, in that it allows all members of the group to work together and we can see the changes another person makes in real time. This is great for sharing general knowledge about Latex and how to use the different components and packages in the correct way. The biggest downside to using Overleaf is that we can't make use of grammar tools like Grammarly and LanguageTool as the word processor on the site isn't visible to those plugins. This means that in order to make use of those tools we would have to download the native application and make use of copy/paste, which from a revision control perspective is bad practice. As none of us are native English speakers there are moments where the semantic meaning and sentence structure gets lost in translation. Having access to tools that can help us directly in the editor would probably improve the overall quality of our thesis, which will show in certain areas of the report.

8.1.3 Moving away from Unitys built-in networking

The question of which networking solution to use arose when we started the project and was looking through Unity's (poor) networking documentation. UNet (Unity's built in networking framework) is deprecated.

At first we decided to try UNet and the HLAPI (High Level API) as it made the most sense being the built in solution and it would be supported unity 2021, long after the lifetime of our project. Though as we started implementing it we discovered the many small annoyances we've later read about on numerous forum posts by people having the same problems. It's built in components are difficult if not impossible to modify and all network calls are done with XML RPC. This was a huge problem for us as we couldn't implement the solution we wanted.

As the HLAPIs successor the com.unity.transport package promised to fix many of the above issues, and it was working very well for a while. It's well structured and has low level UDP sockets we can control however we want. The only issue is the maturity, it's still in alpha and is lacking some critical features before it can be considered production ready. The culprit for us the is lack of a reliability layer on top of the UDP sockets, as some of the communication in the game, such as item transfers needs to be reliable.

Lidgren.Network is a mature open source .NET UDP networking library updated for use with Unity. It was surprisingly easy to setup and contains features such as reliability with multiple channels, LAN discoverability and UPnP while still giving you complete control over the data that's being sent.

8.2 Source Code Documentation

We made use of Doxygen and Graphviz in order to create the documentation for our source code. Most of the public methods and variables are documented using XML style comments in order to work with Doxygen. This can be found in the Doxygen folder in the root of the delivery folder and is compiled into HTML. We chose not to include the source code documentation as an appendix as this isn't really useful and using the HTML variant allows the user to search through an indexed database. Graphviz generates different diagrams for inheritance, collaboration and code calls which is found in the different parts of the Doxygen documentation. By going to the "Classes -> Class Hierarchy" in the top bar menu an overview of the different parts of the programs and how they are related can be found. This gives a brief overview of the entire source code without the user having to sift through a long list of class names. Opening the "index.html" file found in the Doxygen folder will open the entry point for the Doxygen documentation and just acts as a shortcut so the user won't have to look or search for the entry point in the HTML folder.

9 Conclusion

9.1 Conclusion

Even though our project goals contained a lot of different systems that we wanted to implement we still managed to establish at least the basics of each of them. Making use of an established game engine with the largest community like Unity allowed us to put most of our focus into the goals. We feel strongly that Unity was the best decision for this project and it allowed us to implement gameplay but also focus on making designer tools available in the editor. There are still some ways to go before this framework is in a development-ready status as we lack in game balance tools, but considering our approach we feel confident that this can be improved upon. Having more experience in how to create the different tools, frameworks and utilities will be a huge help in the future and gives us greater insight into how a higher level framework should be developed.

In total, we didn't spend as much time as we wanted on each individual system which is something that is reflected in the prototype game we created for the framework. The end product still reflects the core objective that we presented in the project goals, in section 1.2, which was to explore how to create systems that allow re-usability, and interoperability between components. We ended up with a greater understanding on how to work together on a project where the different systems we created were able to function together and be used in situations that weren't directly related to the intended behaviour.

9.2 Future Work

9.2.1 Cleanup

We are going to use the time between the deadline of this report and the presentation of the thesis to fix and clean up some parts of the code. Our main priority in this period is to sort out any missing comments and to finish the dungeon that will be used in the final prototype that we will use for the presentation. This will involve a lot of minor bug fixes as well as putting the pieces we have finished together into a more coherent experience. Right now we don't have a prototype that makes use of all the different parts we have created in the projects lifetime and as such we would like to do this before the final evaluation. Hopefully, this will allow us to better present the content of our thesis in a better way and make it easier to show our intended goal.

9.2.2 UI and tutorial

A big part of the future work for this project is to establish a good framework and solution to handle tutorials and UI feedback. Currently, most of the UI is there but it lacks in meta-information about how to use the UI. If this game framework were to be used in a real-world scenario the developer needs to be aware of the lack of certain UI features that a new player needs. A shortcut bar that shows the key-binds and a button to open the different menus in the game would greatly improve the player experience and make it less annoying to find the different features of the game.

The UI as of now is lacking in feedback on the abilities and talents and they only contain the bare minimum to give the player some information about their functionality. This is something that would need to be worked out if the framework is to be used to develop an actual game.

9.2.3 Game Balance

One feature we are lacking in the design of the game framework is proper balancing tools. Right now it is cumbersome to tweak values as they are spread out between a few different files. Creating a proper factory to set up enemies, abilities, talents, and items is something that should be implemented by a future user of this framework. Right now this is the biggest issue with our framework and we wanted to spend more time on this but it didn't fit within the projects lifetime.

9.2.4 AI Node Tree Editor

The current implementation of the node tree editor to create AI behaviours in the game have some room for improvement. It currently contains most of the functionality for creating behaviour trees and decision trees and has a range of quality of life (QOL) improvements. In the future, more work should be put into making the core functionality of the node editor be separate from designing just AI. This includes creating a better system to support a wider range of node types so the editor can be used in more situations. One aspect of game design it would be nice to use the editor for is general events and gameplay scripting. It can be expanded to handle quests and NPC dialog which could come in handy in a scenario where the game design involved a more story-based experience with branching dialogue trees. If an NPC should give a dialogue option after a certain event is completed this can easily be implemented using a node based editor. It makes it easier for a designer to manage those options without the need for a programmer. Branching decisions is really the strong side of node editors which could be explored more in the future of this project. As of now, this is the biggest downside to how the node editor was implemented as it is pretty much hard-coded for an AI behaviour scenario.

One handy feature that was on the list of planned features was to allow a sub-tree of a Behaviour Tree to be a Decision Tree or a Finite State Machine. This is currently not implemented and could lead to some pretty interesting possibilities when designing the AI. Allowing lower parts of the tree to make use of either FSMs or DTs can allow the designer to make use of the power of those algorithms while still being able to create dynamically controlled behaviour through Behaviour Trees. Most of the groundwork has been done for this to be implemented but it still requires a solution in order to make the design process accessible for designers in a non-complicated way. In addition to this, a utility to transform Behaviour Trees and Decision Trees into Finite State Machine can make it easier to do debugging and verifying that the tree follows the intended behaviour.

Future QOL improvements to the node editor include the ability to create templates of parts of a tree. This would make it easier to manage a larger amount of trees where they have a lot of common functionality between them. The ability to create sub-trees is handy but this should be avoided for very small features. Being able to save those features into a template that can later be inserted would be a great improvement and make it easier for the designer to work with the editor. Another missing feature is a simple way to duplicate whole trees which is possible to do manually, but it would be easier for designers to create smaller variations without having to go through the rather complicated duplication process.

Bibliography

[1] Polygon dungeon assets on unity asset store. https://assetstore.unity.com/packages/3d/environments/dungeons/polygon-dungeons-pack-102677, 2019.

[2] Path of exile wiki. https://pathofexile.gamepedia.com/g00/Path_of_Exile_Wiki, 2019.

[3] Super character controller github repository. https://github.com/IronWarrior/SuperCharacterController, 2019.

[4] Chris Simpson. Behaviour trees article on gamasutra. https://www.gamasutra.com/blogs/ChrisSimpson/20140717/221339/Behavior_trees_for_AI_How_they_work.php, 2019.

[5] Tomas Plch, Matej Marko, Petr Ondareck, Martin Cerny, Jakub Gemrot, and Cyril Brom. An ai system for large open virtual world, 2014.

[6] Nodecanvas - unity store node editor. https://nodecanvas.paradoxnotion.com/documentation/?section=getting-started, 2019.

[7] Naughty attributes github repository. https://github.com/dbrizov/NaughtyAttributes, 2019.

[8] ufaction editor plugin for unity. https://assetstore.unity.com/packages/tools/ufaction-17817, 2019.

[9] Gram Game Studio. Gram node editor for unity editor. http://gram.gs/gramlog/creating-node-based-editor-unity/, 2019.

[10] Uuid on wikipedia. https://en.wikipedia.org/wiki/Universally_unique_identifier, 2019.

[11] Unity documentation - scripting restrictions. https://docs.unity3d.com/Manual/ScriptingRestrictions.html, 2019.

A Project Agreement

NTNU
Norwegian University of
Science and Technology

PROJECT AGREEMENT

between NTNU Faculty of Information Technology and Electrical Engineering (IE) at Gjøvik (education institution), and

RICHARD BARLOW

______ (employer), and

EINAR KASPERSEN BUDSTED

OLE-GUSTAV RØED

EIVIND VOLD AUNEBAKK (student(s))

The agreement specifies obligations of the contracting parties concerning the completion of the project and the rights to use the results that the project produces:

1. The student(s) shall complete the project in the period from 30.01.2019 to 20.05.2019.

   The students shall in this period follow a set schedule where NTNU gives academic supervision. The employer contributes with project assistance as agreed upon at set times. The employer puts knowledge and materials at disposal necessary to complete the project. It is assumed that given problems in the project are adapted to a suitable level for the students' academic knowledge. It is the employer's duty to evaluate the project for free on enquiry from NTNU.

2. The costs of completion of the project are covered as follows:
   - Employer covers completion of the project such as materials, phone/fax, travelling and necessary accommodation on places far from NTNU. Students cover the expenses for printing and completion of the written assignment of the project.
   - The right of ownership to potential prototypes falls to those who have paid the components and materials and so on used to make the prototype. If it is necessary with larger or specific investments to complete the project, it has to be made an own agreement between parties about potential cost allocation and right of ownership.

3. NTNU is no guarantor that what employer has ordered works after intentions, nor that the project will be completed. The project must be considered as an exam related assignment that will be evaluated by lecturer/supervisor and examiner. Nevertheless it is an obligation for the performer of the project to complete it according to specifications, function level and times as agreed upon.

4. The total assignment with drawings, models and apparatus as well as program listing, source codes and so on included as a part of or as an appendix to the assignment, is handed over as a copy to NTNU who free of charge can use it in lessons and in research purpose. The assignment or appendix cannot be used by NTNU for other purposes, and will not be handed over to an outsider without an agreement with the rest of the parties in this agreement. This applies as well to companies where employees at NTNU and/or students have interests.

   Assignments with grade C or better are registered and placed at the university's library. An electronic project assignment without attachments will be placed on the library part of NTNU's website. This depends on that the students sign a separate agreement where they give the library rights to make their main project available both on print and on Internet (ck. The Copyright Act). Employer and supervisor accept this kind of disclosure when they sign this project agreement, and they must possibly give a written message to students and dean if they during the project period change view on this kind of disclosure.

5. The assignment's specifications and results can be used by the employer's own work. If the student(s) in its assignment or while working with it, makes a patentable invention, relations between employer and student(s) applies as described in Act respecting the right to employees' inventions of 17th of April 1970, §§ 4-10.

6. Beyond the publishing mentioned in item 4, the student(s) have no right to publish his/hers/theirs assignment, fully or partly or as a part of another work, without consensus from the employer. Equivalent consent must be made between student(s) and lecturer/supervisor regarding the material placed at disposal by the lecturer/supervisor.

7. The students shall hand in the assignment with attachments electronic (PDF) in NTNU's digital exam system. In addition the students shall hand in a copy to the employer.

8. This agreement is drawn up with one copy to each party. On behalf of NTNU it is the head of the Department/Group that approves the agreement.

9. In each case it is possible to enter separate agreement between employer, student(s) and NTNU who closer regulate conditions regarding issues such as ownership, further use, confidentiality, cost coverage, and economic utilization of the results.

If employer and student(s) wish an additional or new agreement, this will occur without NTNU as a partner.

10. When NTNU also act as employer, NTNU accede to the agreement both as education institution and as employer.

11. Possible disagreements concerning understanding of this agreement are solved by negotiations between the parties. If consensus is not achieved, the parties agree that the disagreement is solved by arbitration, according to provision in Civil Procedure Act of 13th of August 1915, no 6, chapter 32.

12. Participants by project implementation:

NTNUs supervisor (name): CHRISTOPHER FRANTZ

Employers contact person (name): RICHARD BARLOW

Student(s) (signature): [illegible] date 30/01/2019

[illegible] date 30/01/2019

[illegible] date 30/01/2019

_________________ date _________

Employer (signature): DocuSigned by: [illegible] 5BD2C2D109DB4F7 date 2/1/2019

The Project Agreement is to be handed in in digital version in Blackboard. Digital approval by head of the Department/Group.

If a paper version of the Agreement is needed, is must be handed in at the Department in additional.

Head of Department/Group (signature): _________________ date _________

B Initial Project Plan

ARPG - Bachelor Thesis Project Plan

Ole-Gustav Røed (131326), Eivind Vold Aunebakk, Einar Kaspersen Budsted [*]

January 2019

[*]NTNU IDI

Contents

1 Project Plan

1.1 Background

The groups members are all studying game programming, where most of the members have some former knowledge of the Unity Game Engine.

1.2 Technology

Unity Game Engine on the latest version(2018.3.3f1) making use of Unity.Experimental.Input for the new event based input system in Unity, Unity.Experimental.Entity(and the related systems) to make use of the new ECS system currently being incorporated in to Unity. This will allow optimization on big group of entities. Pure ECS is currently missing a lot of key features so a hybrid implementation using the component system currently in Unity will make the whole process a lot cleaner. ECS offers a different way to think of entities in a program and offers less complexity with what each controller is responsible for at a cost of more complexity in the structure of the programs source code.

Git, issue tracker and collaboration software will be used to more easily coordinate the different group members. Gitlab/Bitbucket for Git, Jira/Trello for issue tracking and Discord for team collaboration. Gitlab supports issue tracking, boards, milestones, tags and a few other features, so we will be using Gitlab as the primary source code management tool. Git LFS (Large File Storage) will be used to store a hash of large files (3D Models, textures, etc) that doesn't comply with the git revisioning system. Unity also has a tool named UnityYAMLMerge which will make git merging of unity .scene and .asset files a lot easier.

VS Code is the primary IDE as it is lightweight and contains all the necessary features for debugging, linting and exterior project management.

Mono/C# .net 4.X(latest Unity using Roslyn has .net 7.2 compatability) is the primary compiler toolset used making use of the Roslyn compiler to allow incremental compilation. Incremental compilation allows unity to only compile the files that have changes and not every file in the project making the compilation process a lot faster on bigger projects.

Latex/Overleaf to cooperate on the thesis writing, making use of Zotero to keep a list of references. Overleaf supports git so it should be mirrored and included in the Gitlab project for backup.

1.3 Project Goals

Create a set of tools and utilities to support the development of an ARPG game. This includes tools for game design, level design, asset creation pipeline and other components commonly found in ARPGs. Making use of data oriented design and the new ECS(Entity-Component-System) that is currently being incorporated into Unity for the agents(i.e. enemies) and other entities that can make use of the performance benefit.

Implementing the base components that makes up an ARPG. This includes: abilities, talent points, character system, modifiers for items, modifiers for enemies, itemization, creatures, agents/ai(NPCs, enemies, etc.), level generation/design, story arc, inventory system, attributes, classes, resources (hp, mana, gold, etc.), status effects(buffs, debuffs, auras, etc.), quests/events, audio, ui, crafting and so on. These systems will have different priority of implementation and many of them are not possible to implement within the time limit.

The last part of our goals is to put this all together into a working game with support for multiplayer. We have three focus areas of features in the game, which boils down to AI, Networking and UX. All of these are pretty loosely defined, but they all revolve around the core aspects of the game making process. After the core gameplay is implemented the group will split their focus between these feature groups in order to meet the final project goal.

1.4 Scope

1.4.1 Areas of Expertise

Focus on reuse of code and ease of modification and create a development environment in which every component that logically fit together will fit together. This means that an attribute modifier should work on all entities that have that attribute, and so on. This is to ensure that the game design is not hindered by limitations of the code base.

1.4.2 Scope of Limitation

Limit scope of the assignment to backend systems and frontend logic. This means that only tools for desired game design limitation is focused on and not the game design aspect in of itself. This does not limit mathematical implementations of game design, but limits the time used on making things "feel balanced", i.e. minor tweaking to game balance. AI, networking and UX are the three main areas we have set the scope of the project to, after the core game functionality is implemented. These areas are further explained in the Project Goals[1.3] section above.

1.5 Project Structure

No specific leader is chosen, but each member will be responsible for a certain area of the project. There are three members on the group so any decision will always be resolved. The responsibilities will rotate between the group members every two weeks. The areas that need a specific leader are as follow:

- One person is responsible for contact with the product owner and supervisor of the Bachelor thesis. One point of contact means that all members of the group will have access to any important information. Another solution is that we always forward the mails to the other group members.

- One person is responsible for project management. This involves making sure any changes to the main branch follows the rules and is responsible for the final decision when implementing new features to the game. Everyone has a responsibility to ensure they don't pollute the main branch. This ensures that one person has a view of the bigger picture so that merge requests are fullfilled properly.

- One person is responsible for documentation and scrum meetings. This person will take notes from all meetings and make sure the development documentations are kept up to date.

1.6 Planning, supervision and documentation

1.6.1 System Engineering Paradigm

Scrum/XP with focus on reuse of code and frequent implementation of new features. Making use of an iterative process with focus on code reusability fits well with the game development process. Prototyping and later reimplementation of points of congestion can allow for rapid progress, but code reviews and refactoring should be interweaved to ensure that the code base stays healthy and to minimize mirror code.

Two short daily meetings and cooperative planning should be held during periods where the team members work on the same parts of the code ensuring that the team is properly coordinated and to minimize the amount of mirror code we produce. Outside of this there are weekly code reviews and scrum meetings. The sprint meetings should produce a short report with meeting notes, and the code review should contain the latest information on performance (UnityEditor Profiler data, production build compilation errors and other relevant information) and a short report on the areas of the code base that was under review. Any bigger issues found during code reviews should be documented in the issue tracker.

1.7 Quality Assurance

1.7.1 Documentation, Coding Convention and Collaboration

Code convention should follow the standards set for the programming/scripting language used. Documentation should follow Doxygen standards where every Class, Interface, public Member and Enum is documented. ~~Lua functions are global so no two functions can have the same name. The general style to follow should be the Allman style.~~

Issues and bugs should be tracked accordingly, making sure that they are followed through.

Allman Coding Convention:
`https://en.wikipedia.org/wiki/Indentation_style#Allman_style`
C# Official Coding Convention:
`https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/inside-a-program/coding-conventions`
C# Coding Convention Examples:
`https://github.com/ktaranov/naming-convention/blob/master/C%23%20Coding%20Standards%20and%20Naming%20Conventions.md`

The main branch on the git tree should always be kept clean, making use of branches. Branches should have a clear and concise task to fulfill(issue, new functionality, etc.) and be merged with the main branch through a pull request. Every pull request should be linked to an issue or a functionality and the pull request should contain relevant information found during work on it. Issues and pull request should refer to debug information and output from the Unity profiler in the UnityEditor. This ensures that it will be easy to create documentation for the projects progress and allow better understanding of problems during the development. The downside to this increased complexity in the structure of the source tree and the members of the group needs to ensure they stay on the correct branch to avoid massive merging traps.

1.7.2 Testing

Testing will be an important part of the development process. The project can make use of exploratory or scripted testing where the first is an after-the-fact method and the last is a preemptive method of testing. There are benefits and drawbacks to both methods and the choice of testing paradigm will depend of the part of the game being worked on.

Networking and game logic are the main areas we will be doing testing. An after-the-fact testing method fits the best with the system engineering paradigm we have chosen and is most likely what we will be using. Whenever a feature is pushed on the main branch the related tests should be written to ensure any future changes conforms to the current specifications.

Unit Tests should be written for all logic present in the game. Functional/Integration Tests for all the interactions in the program and game. So called runtime tests are written to ensure that what the user does is working. Depending on how structured the gameplay is this can be hard to make use of and might be outside the scope of limitations.

1.8 Development Plan

17 weeks to the delivery date(20th May) from week 5(28th Jan).

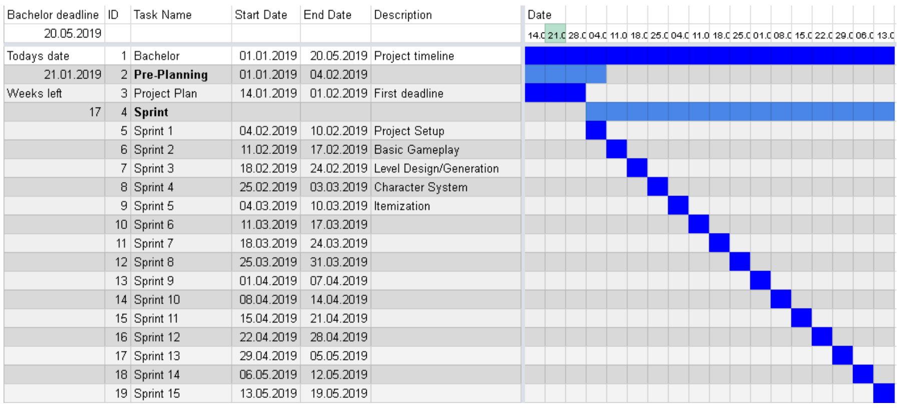
Ten weeks are planned to be used on implementing the features of the game making use of rapid implementation. Then 5-7 weeks are ear marked for writing the report, fixing bugs, cleaning up the code base and complete other tasks needed before the delivery date.

Some of the milestones we have planned are(in order of importance):
1. Setting up the base project, ready for split teamwork, 1 week
2. Main components of the Character System and its interactions, 2 weeks
3. AI/Agents and their interactions, 2 weeks
4. Level Generation/Design and environmental objects, 2 weeks
5. Base game finished with a working binary, 2 weeks
6. Creating the components making up the UX (UI, Camera Director, etc.), 2 weeks
7. Writing the report, finishing touch, 3+ weeks

All of the above steps can be broken down in to smaller tasks where some of the milestones have overlapping tasks.

A slice of the tasks we have planned for the project:
- Setting up all tools required for project management/work and drafting the project structure.
- Implementing base components for game, including character movement, game state, input system, etc.
- Implementing attribute and modifier systems
- Implementing an inventory system, creating the base for items in the game
- Implementing itemization and a simple item generator
- Implementing simple agents to act as enemies/NPCs
- Implementing a boss AI making use of phases that changes the boss behaviour.
- Implementing a level generator to support prototype gameplay
- Creating the gameplay prototype, connecting all parts created for the game
- Seperating prototyped content in to reusable components
- Implementing the audio system and dispatching of audio cues. (AFX)
- Implementing a Camera Director and a trigger system for AudioVisual cues. (SFX)
- Setting up physics interaction between entities.
- Implementing an ability system to handle attacks and spells.
- Setting up model commands to retreive entity status/properties.

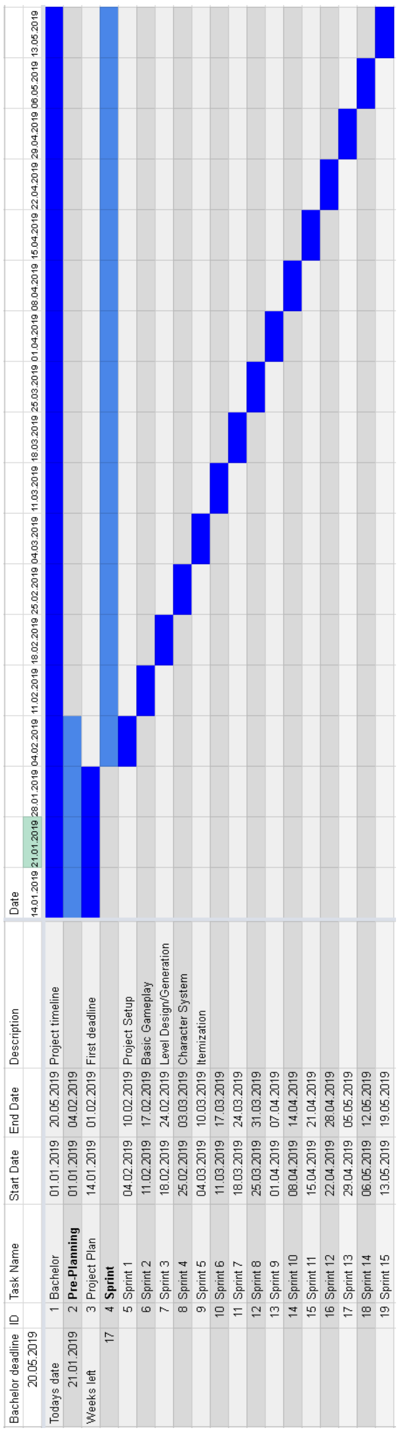
1.8.1 Gantt Chart



Bachelor deadline	ID	Task Name	Start Date	End Date	Description	Date																	
20.05.2019						14.0	21.0	28.0	04.0	11.0	18.0	25.0	04.0	11.0	18.0	25.0	01.0	08.0	15.0	22.0	29.0	06.0	13.0
Todays date	1	Bachelor	01.01.2019	20.05.2019	Project timeline	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■
21.01.2019	2	**Pre-Planning**	01.01.2019	04.02.2019		🟦	🟦	🟦	🟦														
Weeks left	3	Project Plan	14.01.2019	01.02.2019	First deadline	■	■	■															
17	4	**Sprint**							🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦
	5	Sprint 1	04.02.2019	10.02.2019	Project Setup				■														
	6	Sprint 2	11.02.2019	17.02.2019	Basic Gameplay					■													
	7	Sprint 3	18.02.2019	24.02.2019	Level Design/Generation						■												
	8	Sprint 4	25.02.2019	03.03.2019	Character System							■											
	9	Sprint 5	04.03.2019	10.03.2019	Itemization								■										
	10	Sprint 6	11.03.2019	17.03.2019										■									
	11	Sprint 7	18.03.2019	24.03.2019											■								
	12	Sprint 8	25.03.2019	31.03.2019												■							
	13	Sprint 9	01.04.2019	07.04.2019													■						
	14	Sprint 10	08.04.2019	14.04.2019														■					
	15	Sprint 11	15.04.2019	21.04.2019															■				
	16	Sprint 12	22.04.2019	28.04.2019																■			
	17	Sprint 13	29.04.2019	05.05.2019																	■		
	18	Sprint 14	06.05.2019	12.05.2019																		■	
	19	Sprint 15	13.05.2019	19.05.2019																			■

Figure 1: Gantt Chart Compressed

Full version [2]



Bachelor deadline	ID	Task Name	Start Date	End Date	Description
20.05.2019					
Todays date	1	Bachelor	01.01.2019	20.05.2019	Project timeline
21.01.2019	2	**Pre-Planning**	01.01.2019	04.02.2019	
Weeks left	3	Project Plan	14.01.2019	01.02.2019	First deadline
17	4	**Sprint**			
	5	Sprint 1	04.02.2019	10.02.2019	Project Setup
	6	Sprint 2	11.02.2019	17.02.2019	Basic Gameplay
	7	Sprint 3	18.02.2019	24.02.2019	Level Design/Generation
	8	Sprint 4	25.02.2019	03.03.2019	Character System
	9	Sprint 5	04.03.2019	10.03.2019	Itemization
	10	Sprint 6	11.03.2019	17.03.2019	
	11	Sprint 7	18.03.2019	24.03.2019	
	12	Sprint 8	25.03.2019	31.03.2019	
	13	Sprint 9	01.04.2019	07.04.2019	
	14	Sprint 10	08.04.2019	14.04.2019	
	15	Sprint 11	15.04.2019	21.04.2019	
	16	Sprint 12	22.04.2019	28.04.2019	
	17	Sprint 13	29.04.2019	05.05.2019	
	18	Sprint 14	06.05.2019	12.05.2019	
	19	Sprint 15	13.05.2019	19.05.2019	

Figure 2: Gantt Chart Full

A Group Contract

Group Members:

Ole-Gustav Røed
Eivind Vold Aunebakk
Einar Kaspersen Budsted

Each member owns an equal part (33%) of the project.

1. Project Property is any asset produced, purchased or licensed for use in the whole group projects period. This includes source code, documentation, game design, art, libraries, etc.
2. Any asset bought is owned by the buying party, but is a part of the project property until the end of the projects lifetime.
3. If a member is dismissed or leaves the group, any rights that person has to the project property is lost, any bought assets follows rules set in point 2.
4. If a member is not attending meetings or not contributing to the project without prior notice for more than 2 weeks they will be subject to termination. The other members of the team will have to agree on the outcome, and the supervisor will be involved to provide an outside perspective.
5. The project will be released as open source, unless the group decides that they want to keep it closed before the project ends.

B Game Design

- **Genre:** Action RPG
- **Multiplayer:** Online co-op, possibly local split screen
- **Controls:** Mouse/Keyboard, Controller
- **Platform(s):** PC/Linux, Mac
- **Camera:** Birds eye, with director control
- **Thematic Setting:** Norse Mythology, Gods, Vikings, Magic
- **Game Summary:** Action experience centered around killing large groups of enemies with some boss fights and puzzles.

  The game starts with the player encountering a god. The player which is a quite normal viking discovers he/she is actually a demigod and that a powerful evil is threatening Asgard. The player is sent off on a great adventure to save the world.

  The game will use a map structure similar to what is commonly found in dark souls and zelda games. There will be one main zone which branches out into smaller areas and dungeons with set objectives the player must complete to gain access to new branches.
- **Core Player Experience:** Challenging, Thrilling
- **Central Theme:** Unexpected hero, Call to adventure
- **Comparative Products:** Diablo, Titan Quest, Torchlight
- **Art Style:** Low-Poly art style. Ex: [Fig. 3], [Fig. 4]

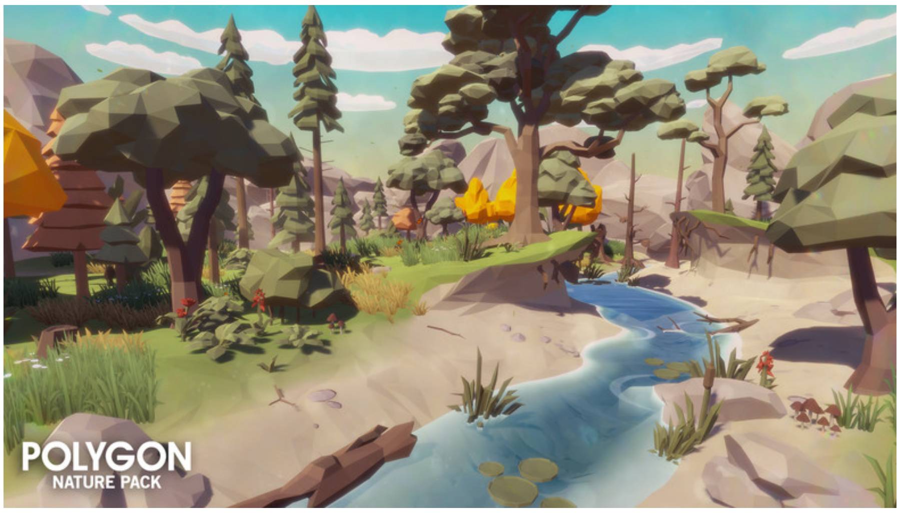


Figure 3: Low poly example 1

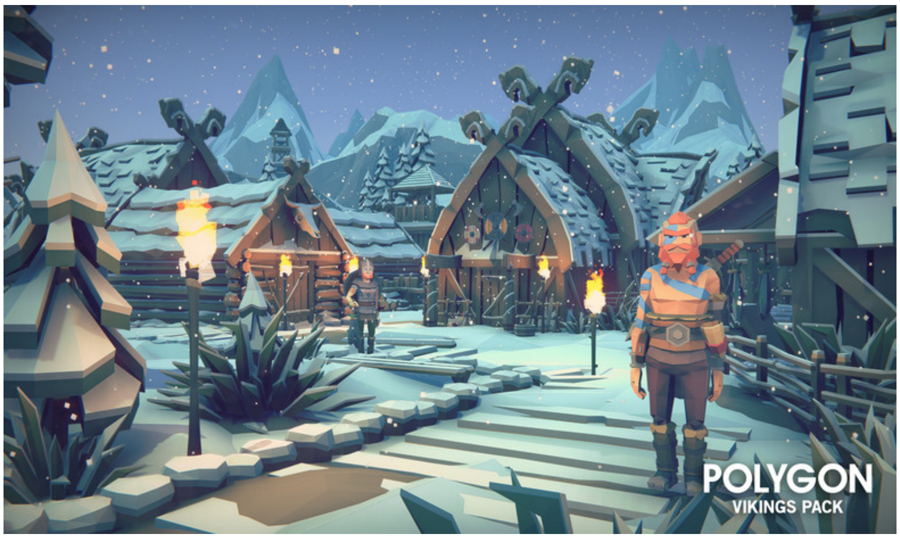


Figure 4: Low poly example 2

C Meeting Logs

Sprint Period:

Planing Meeting:

- Setting up unity (2018.3.4f1 instead of the planned 2018.3.3f1)
- Implementing camera, input and character controls
- Implementing simple networking (two characters and a lobby)
- Setting up the basics for Agents

Daily Notes:

- : Setup GitLab boards
  Decided against initially using ECS for the player and agents as there's is no trivial way of incorporating animations and network synchronization

Retrospective:

-

Review Meeting:

-

Sprint Period:

Planing Meeting:

- More ai (state machine, state handlers, sensors/telemetry)
- More character controller (ability system)
- utility (raycasting controller,)
- Expanding multiplayer code for local server

Daily Notes:

- :

Retrospective:

- We managed to meet most of the criteria we set for the sprint. The basic parts were implemented, but some more time could've been spent on each part.

Review Meeting:

- All initial goals were met for this sprint. Basic character control and camera is setup, multiplayer framework is set up and the initial AI/Agent behaviour is implemented. All work was done on different branches and all the branches were merged in to main branch at the end of this sprint.

Sprint Period:

Planing Meeting:

- Expanding the ability system
- Implementing a few different enemy types
- Start working on itemization and character system

Daily Notes:

- :

Retrospective:

-

Review Meeting:

- AI framework for sensors and decision making is implemented. The work on AI is now ready to implement specific enemy types. Networking has reached a point where we need more work done on the core aspects of the game before the networking can be expanded. The base for an ability system and corresponding GUI got implemented.

Sprint Period:

Planing Meeting:

- Implementing more GUI and HUD
- Refactoring and improving the functionality of the ability system
- Refactor and improve the AI framework for more than one enemy type

Daily Notes:

- :

Retrospective:

-

Review Meeting:

- Mostly implemented character equipment and stats

Sprint Period:

Planing Meeting:

- Creating a small world/area with events/objectives
- Finish up the ability and character system
- Synchronize two player over network
- Finish up sensor/brain framework
- Implement a humanoid AI

Daily Notes:

- :

Retrospective:

- Could have done a merge earlier for some parts of the sprint

Review Meeting:

- Completed the layout for most UI elements. Implemented inventory, character equipment and ability bar

Sprint Period:

Planing Meeting:

- Finish human agents with ranged attacks
- Change physics component to use capsule collider
- Add/remove objects from NetworkContainers
- Drop/pickup items from the ground
- Chest that drops items when opened
- Implement a class system and ability book

Daily Notes:

- :

Retrospective:

-

Review Meeting:

- Changed networking from UNet to Lidgren Network Library
- Restructured the container system; inventory and character equipment implements a common interface, synchronizing transfers between any NetworkContainer
- Synchronized player name, movement and equipment/inventory
- Remade movement/physics to a single component used by all actors
- Set up the prefab and initial workings of the humanoid AI
- Cleaned up inner workings of AI to uncouple some features from the decision logic
- Incorporated character stats into the abilities

Sprint Period:

Planing Meeting:

- Continue on class system
- Continue on human enemy and node editor
- Create an environment

Daily Notes:

- :

Retrospective:

-

Review Meeting:

- Implemented an affix system for randomly generated equipment
- Made a treasure chest that drops items on the ground when opened
- Players can drop items
- PhysicsComponent uses Capsule as collider, helper functions for static Physics calls implemented.
- Base for agent pool and an area spawner created
- Initial Node Editor for brain and runtime interpreter implemented

Sprint Period:

Planing Meeting:

- Finish up combat text, and ui bindings
- Create the initial layout for the dungeon
- Finish up refactoring of agent and brain to allow for more choices/variance
- Finish up Blackboard structure and variable binding functionality

Daily Notes:

- :

Retrospective:

-

Review Meeting:

- Started on a map, needs encounters
- Added more situational handling in PhysicsComponent (void, on solid, etc)
- Setup a DecisionTree brain to interpret decision trees
- Blackboard structure for global access/local authority of shared data

Sprint Period:

Planing Meeting:

- Create a dungeon
- Create behaviours for human agents

Daily Notes:

- :

Retrospective:

-

Review Meeting:

- Injecting silhouette shader in wolf to show a silhouette through walls
- Behaviour tree implementation underway, with sub-tree nodes, action nodes and composites.
- BehaviourAgent with behaviours that can be called/setup in the node editor
- Serializable MethodInfo/FieldInfo to connect behaviours and data to node tree
- Refactoring of node editor to better support serilization
- Added required items for intractables
- Made it possible to place specific items in a chest
- Nested loot tables
- Added sounds to all the current abilities
- AudioManager for managing scene music and music zones
- Added buffs and debuffs
- Added power
- Additional abilitises and talents

Sprint Period:

Planing Meeting:

- Continue on dungeon
- Create more enemy variety

Daily Notes:

- :

Retrospective:

- sprint period was interrupted by a game jam during the weekend. This gave some insight into what was needed, but also put us back a bit in terms of development time on the main branch.

Review Meeting:

-

Sprint Period:

Planing Meeting:

- More work on creating the later parts of the dungeon
- more work on enemies

Daily Notes:

- :

Retrospective:

-

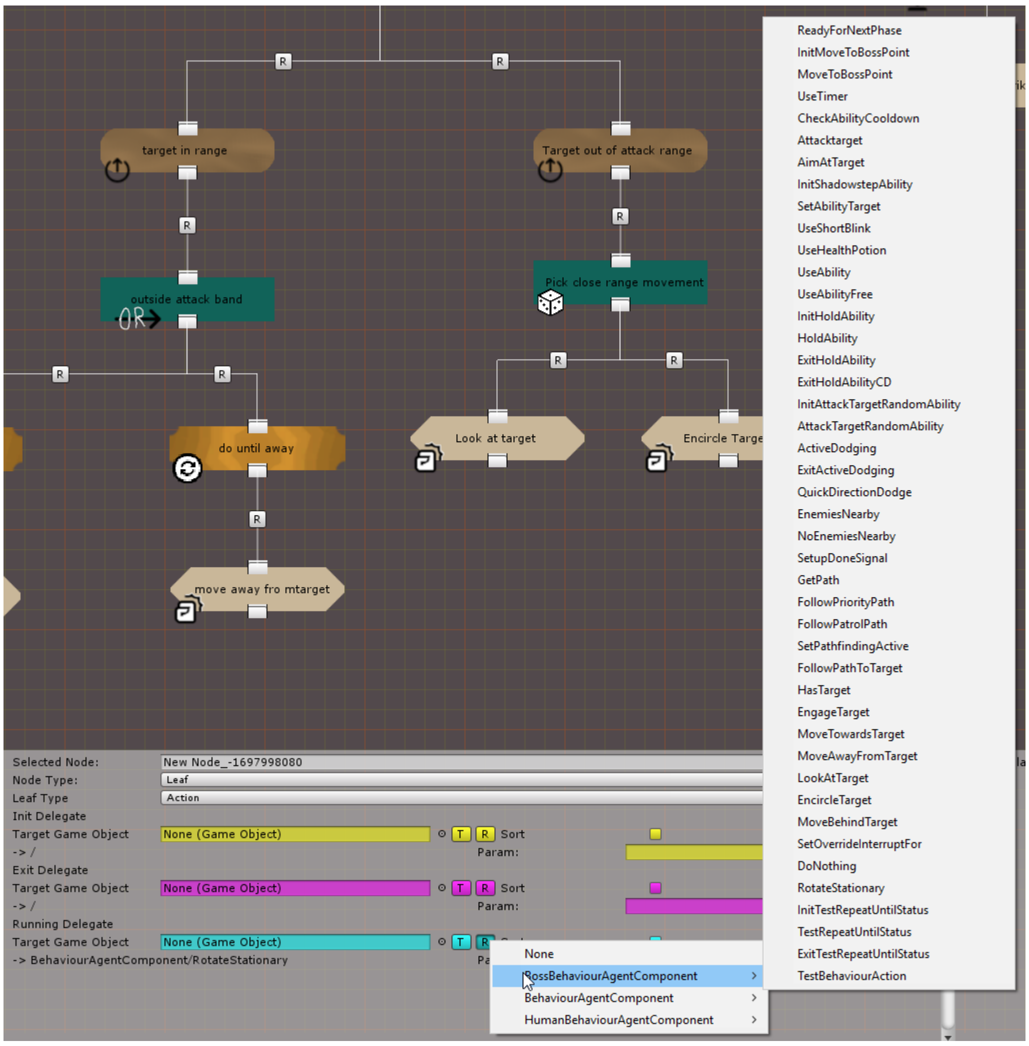
Review Meeting:

-

D Example Screenshots

This appendix will contain a range of pictures from the different parts of the framework and prototype we created.

D.1 Node Tree Editor



Selected Node: New Node_-1697998080
Node Type: Leaf
Leaf Type Action
Init Delegate
Target Game Object None (Game Object) T R Sort
-> / Param:
Exit Delegate
Target Game Object None (Game Object) T R Sort
-> / Param:
Running Delegate
Target Game Object None (Game Object) T R
-> BehaviourAgentComponent/RotateStationary

Figure 26: Image is showing how a method is bound with the MethodReferenceDrawer utility. The root names are the names of classes with the MethodReference attribute and the methods have RunStatus as return value and string as parameter.

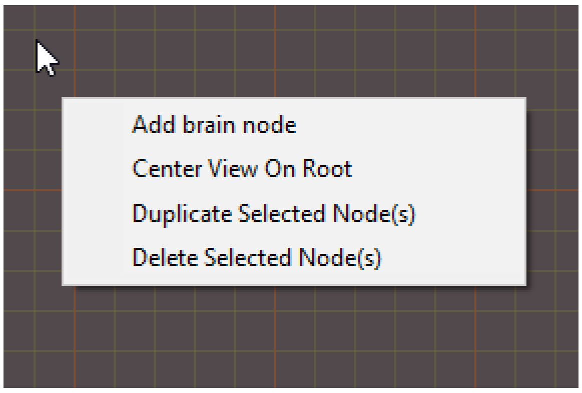


Figure 27: Image is showing the contex menu in the Node Tree Editor to work with the editor and selected nodes.

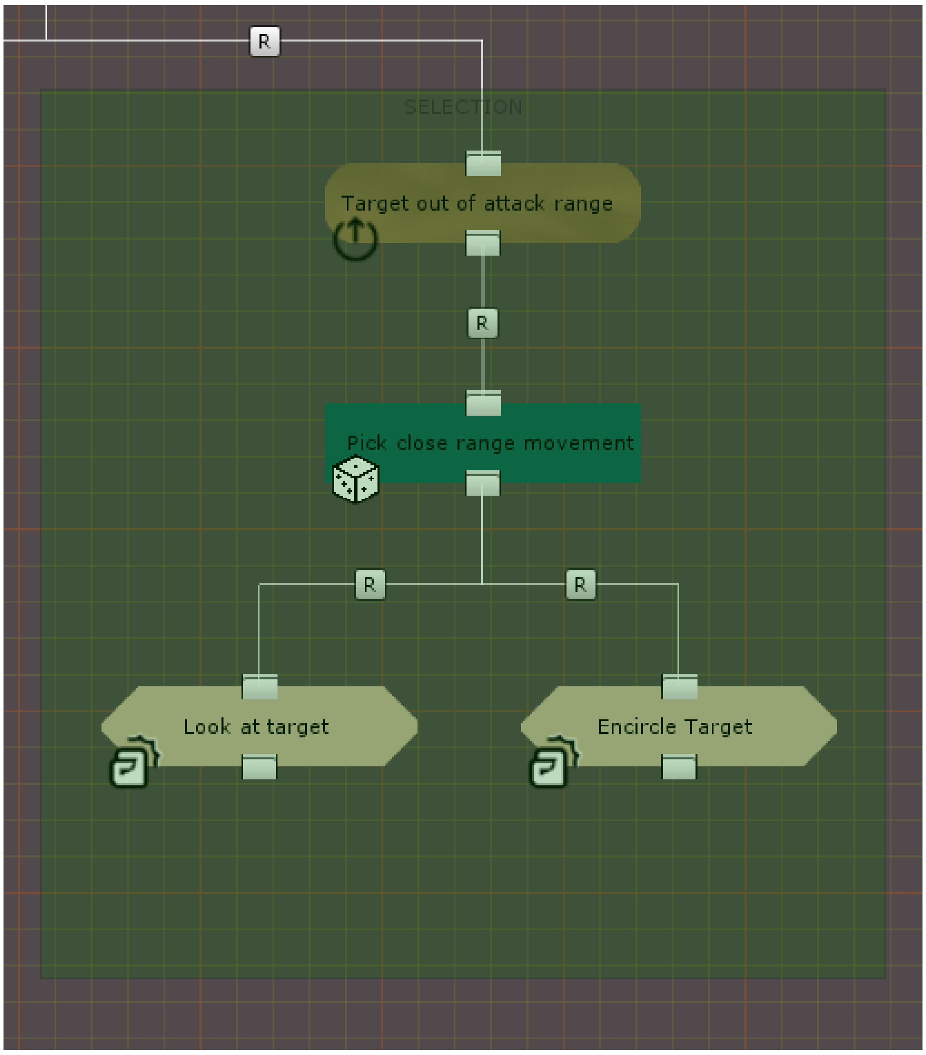
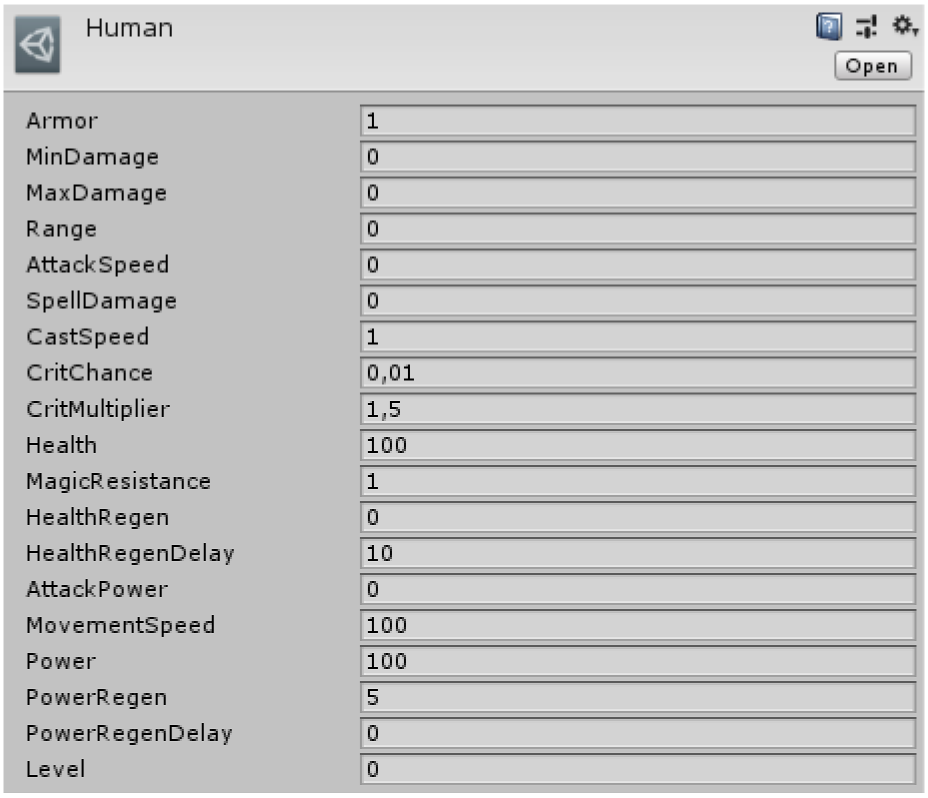


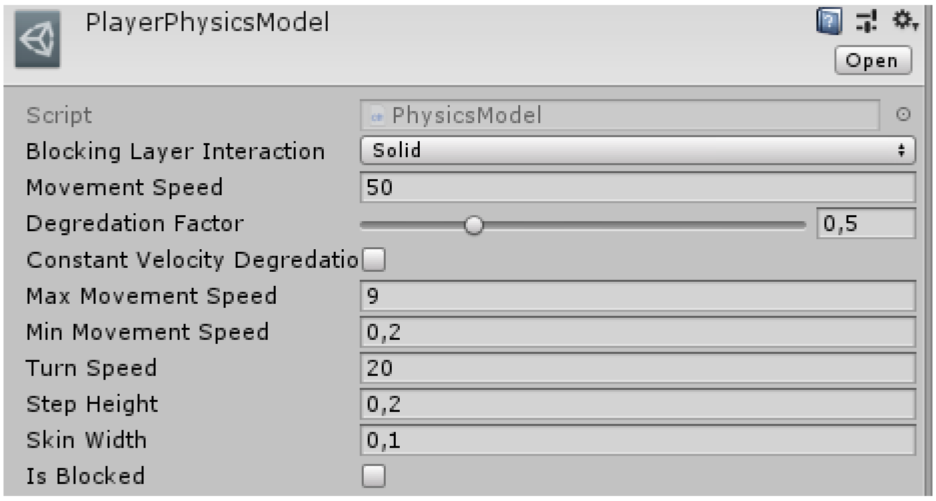
Figure 28: Image is showing the selection box for the node tree editor.

D.2 Character


Human

Armor	1
MinDamage	0
MaxDamage	0
Range	0
AttackSpeed	0
SpellDamage	0
CastSpeed	1
CritChance	0,01
CritMultiplier	1,5
Health	100
MagicResistance	1
HealthRegen	0
HealthRegenDelay	10
AttackPower	0
MovementSpeed	100
Power	100
PowerRegen	5
PowerRegenDelay	0
Level	0

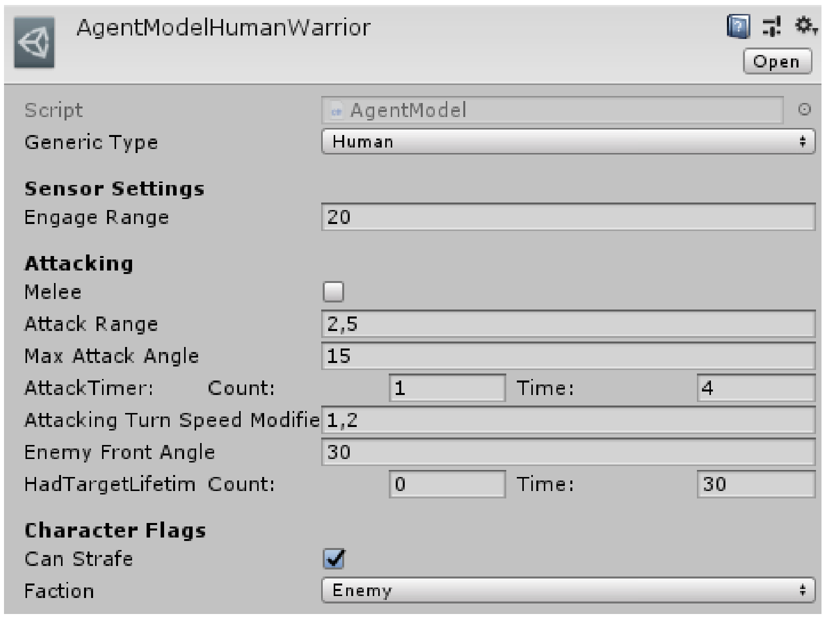
Figure 29: Image is showing the setup of the base stats for actors in the game


PlayerPhysicsModel

Script	PhysicsModel
Blocking Layer Interaction	Solid
Movement Speed	50
Degredation Factor	0,5
Constant Velocity Degredatio	☐
Max Movement Speed	9
Min Movement Speed	0,2
Turn Speed	20
Step Height	0,2
Skin Width	0,1
Is Blocked	☐

Figure 30: Image is showing the setup of the physics variables for an actor in the game

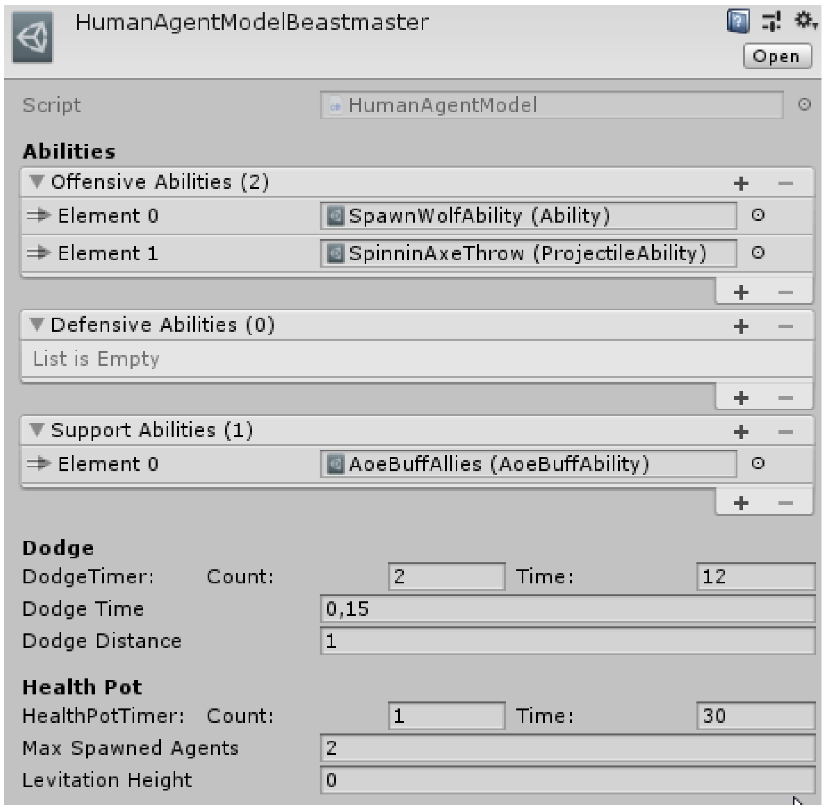
D.3 AI and Agents


AgentModelHumanWarrior

Open

Script	AgentModel
Generic Type	Human
Sensor Settings	
Engage Range	20
Attacking	
Melee	☐
Attack Range	2,5
Max Attack Angle	15
AttackTimer: Count:	1 Time: 4
Attacking Turn Speed Modifie	1,2
Enemy Front Angle	30
HadTargetLifetim Count:	0 Time: 30
Character Flags	
Can Strafe	✓
Faction	Enemy

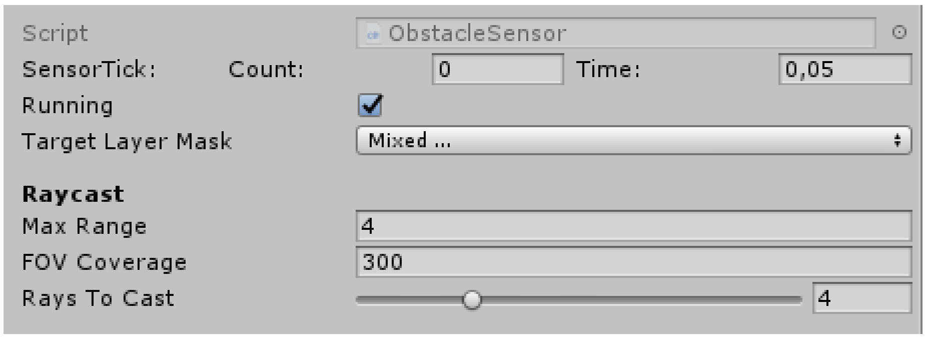
Figure 31: Image is showing how the AgentModel is setup for agents


HumanAgentModelBeastmaster

Open

Script	HumanAgentModel
Abilities	
Offensive Abilities (2)	
Element 0	SpawnWolfAbility (Ability)
Element 1	SpinninAxeThrow (ProjectileAbility)
Defensive Abilities (0)	
List is Empty	
Support Abilities (1)	
Element 0	AoeBuffAllies (AoeBuffAbility)
Dodge	
DodgeTimer: Count:	2 Time: 12
Dodge Time	0,15
Dodge Distance	1
Health Pot	
HealthPotTimer: Count:	1 Time: 30
Max Spawned Agents	2
Levitation Height	0

Figure 32: Image is showing how the HumanAgentModel is configured for use on actors

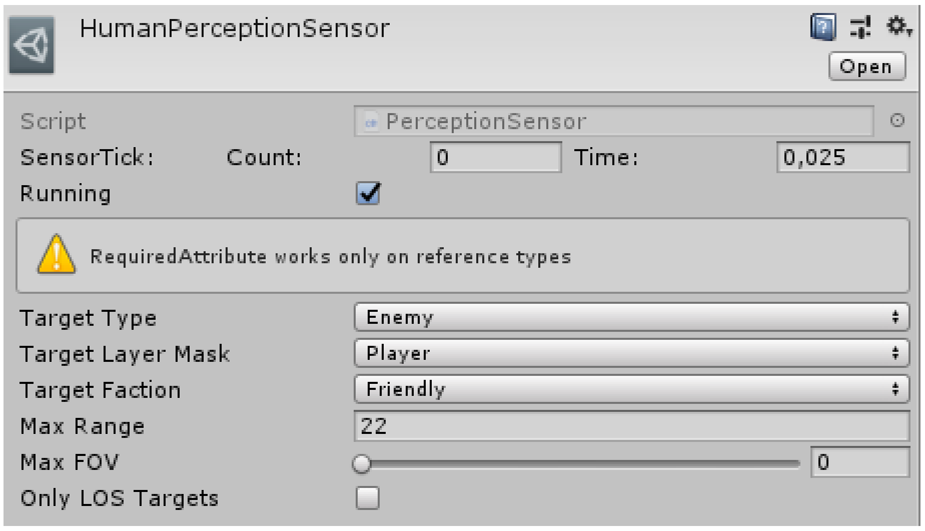


Script	ObstacleSensor		
SensorTick:	Count: 0	Time:	0,05
Running	✓		
Target Layer Mask	Mixed ...		

Raycast

Max Range	4
FOV Coverage	300
Rays To Cast	4

Figure 33: Image is showing how one of the sensors is configured in the framework, the Obstacle Sensor, which is responsible for finding a direction that is free of obstacles


HumanPerceptionSensor

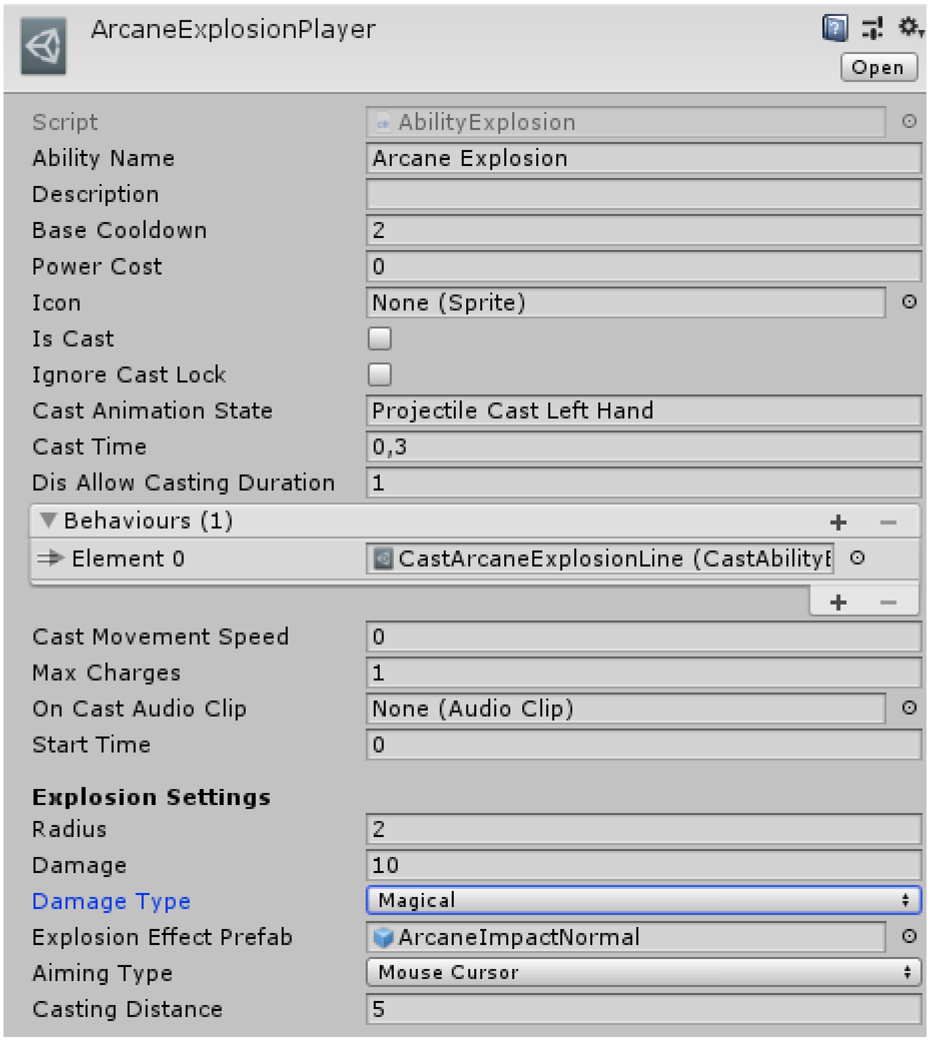
Script	PerceptionSensor		
SensorTick:	Count: 0	Time:	0,025
Running	✓		

RequiredAttribute works only on reference types

Target Type	Enemy
Target Layer Mask	Player
Target Faction	Friendly
Max Range	22
Max FOV	0
Only LOS Targets	☐

Figure 34: Image is showing how one of the sensors is configured for in the framework, the perception sensor, which is responsible for registering nearby actors or other entities of interest.

D.4 Abilities


ArcaneExplosionPlayer

Open

Script	AbilityExplosion
Ability Name	Arcane Explosion
Description	
Base Cooldown	2
Power Cost	0
Icon	None (Sprite)
Is Cast	☐
Ignore Cast Lock	☐
Cast Animation State	Projectile Cast Left Hand
Cast Time	0,3
Dis Allow Casting Duration	1
Behaviours (1)	
Element 0	CastArcaneExplosionLine (CastAbilityE
Cast Movement Speed	0
Max Charges	1
On Cast Audio Clip	None (Audio Clip)
Start Time	0
Explosion Settings	
Radius	2
Damage	10
Damage Type	Magical
Explosion Effect Prefab	ArcaneImpactNormal
Aiming Type	Mouse Cursor
Casting Distance	5

Figure 35: Image is showing how the ability AbilityExplosion is configured.

D.5 UI

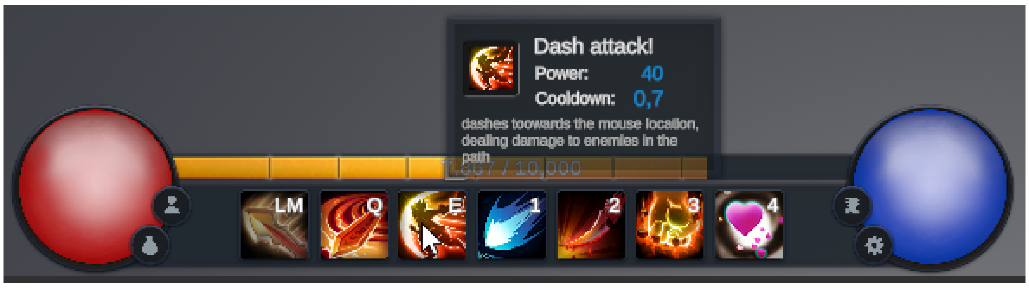
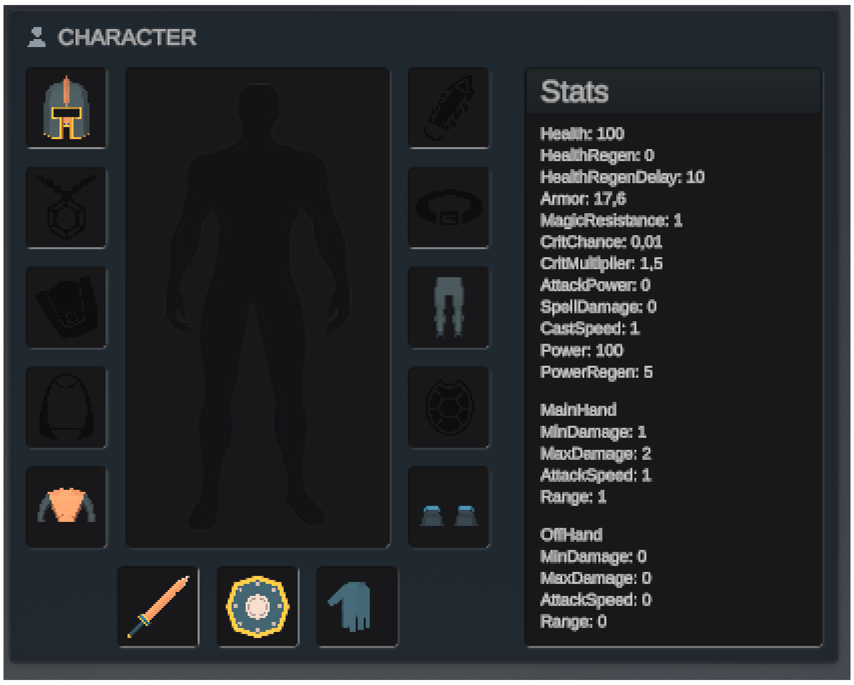

Dash attack!
Power: 40
Cooldown: 0,7
dashes toowards the mouse location, dealing damage to enemies in the path

Figure 36: Image is showing the ability bar, health and mana status and the initial implementation for ability tooltips looks like. The XP bar is currently non-functional.



Health: 100
HealthRegen: 0
HealthRegenDelay: 10
Armor: 17,6
MagicResistance: 1
CritChance: 0,01
CritMultiplier: 1,5
AttackPower: 0
SpellDamage: 0
CastSpeed: 1
Power: 100
PowerRegen: 5

MainHand
MinDamage: 1
MaxDamage: 2
AttackSpeed: 1
Range: 1

OffHand
MinDamage: 0
MaxDamage: 0
AttackSpeed: 0
Range: 0

Figure 37: Image is showing the character menu, which shows the stats and current equipment for the player's character

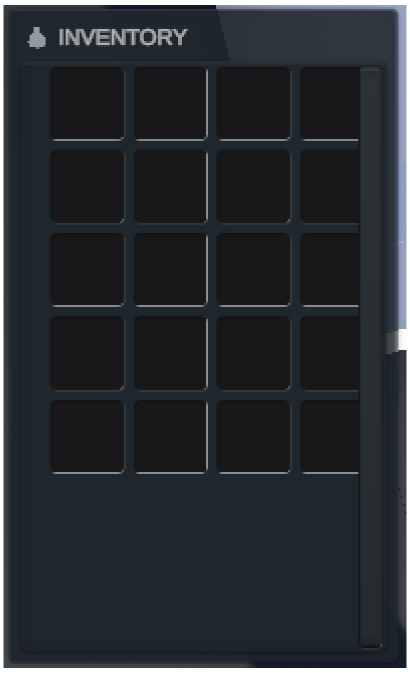


Figure 38: Image is showing how the inventory menu looks when it's empty.

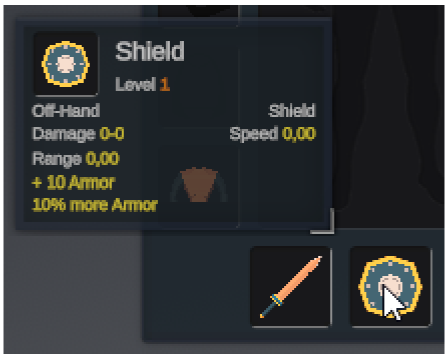


Figure 39: Image is showing how item tooltips looks when an item or equipment is hovered over.

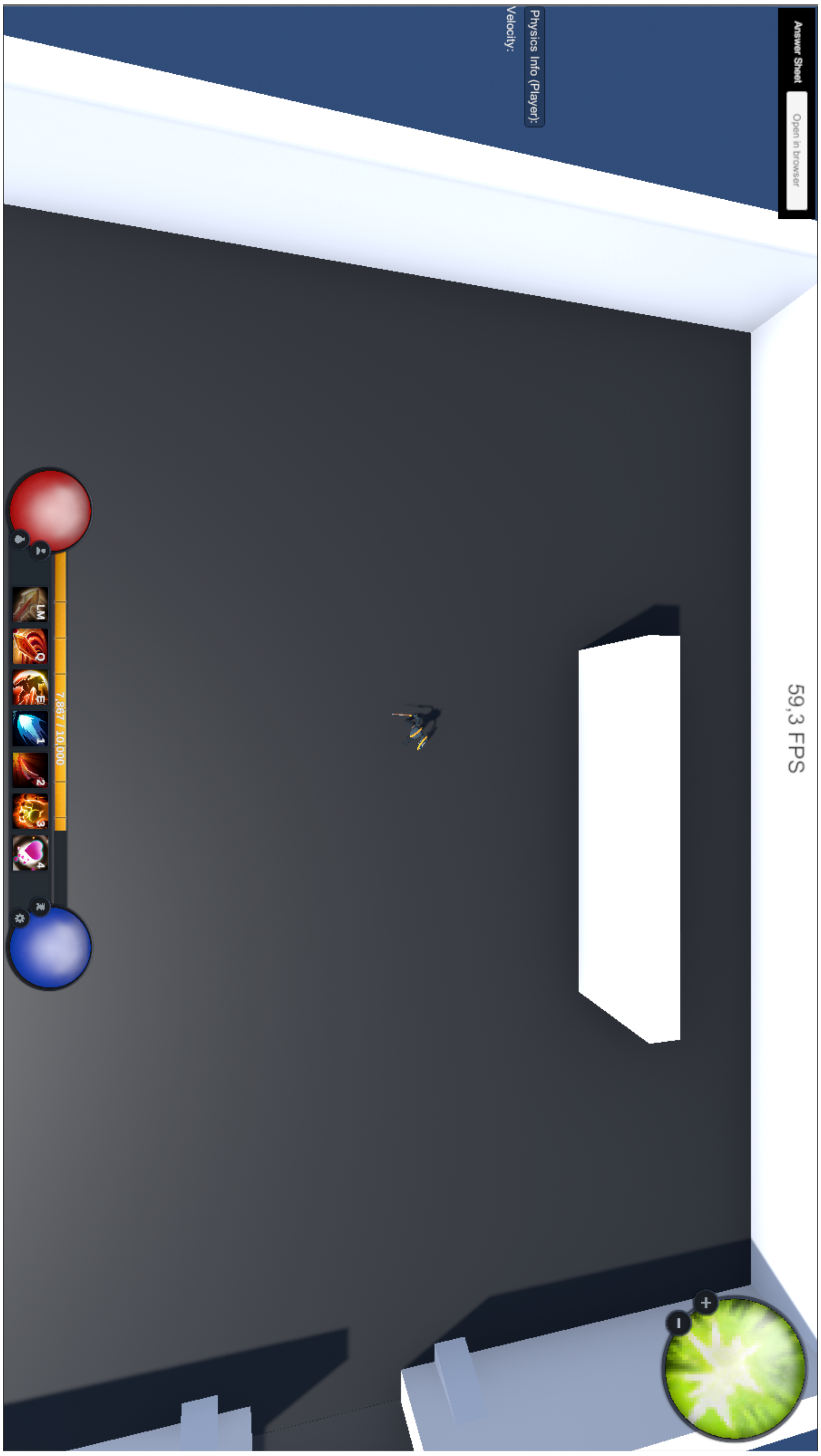


Figure 40: Image is showing the full UI when no elements are active/is being hovered over