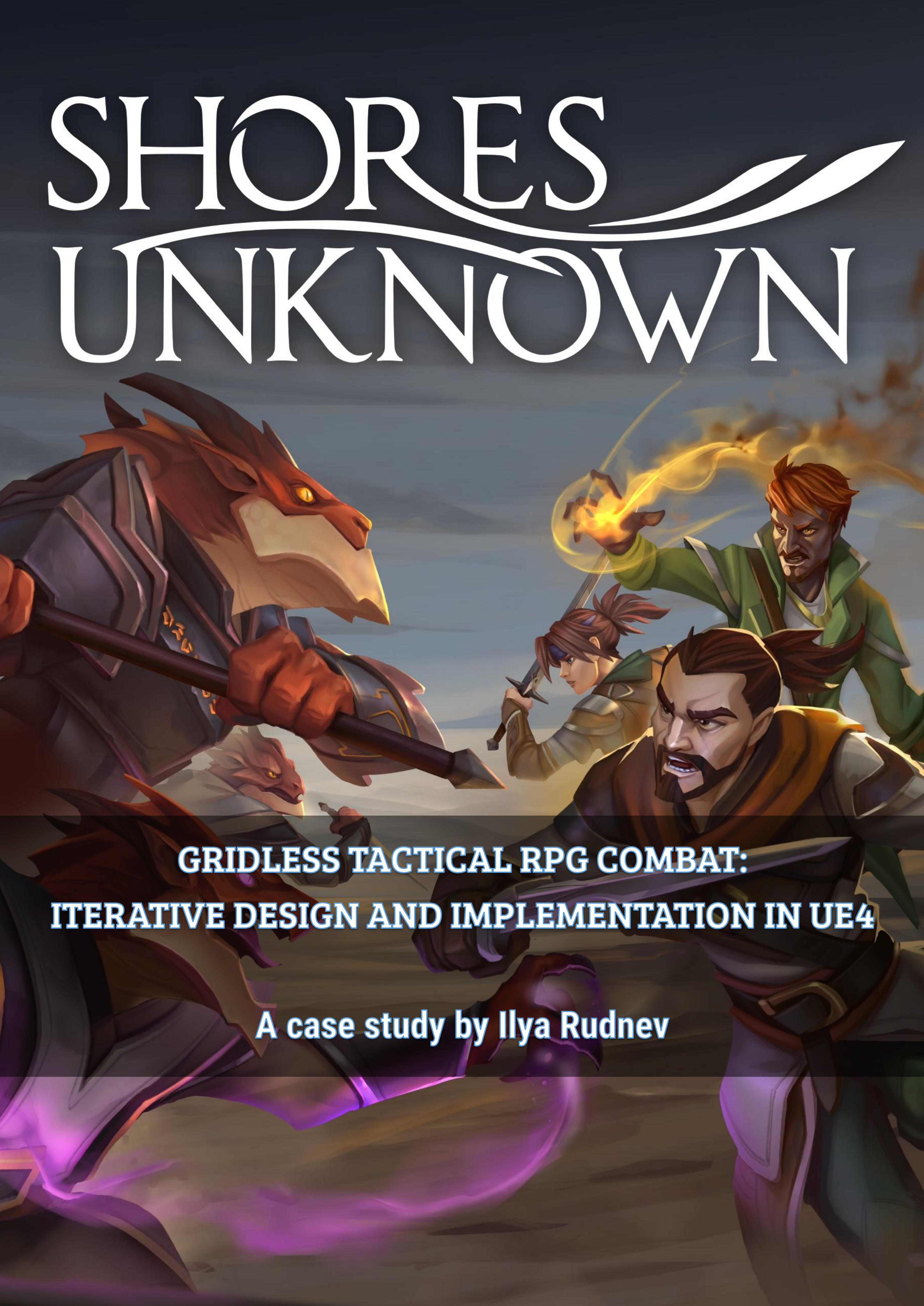


SHORES UNKNOWN

GRIDLESS TACTICAL RPG COMBAT: ITERATIVE DESIGN AND IMPLEMENTATION IN UE4

A case study by Ilya Rudnev

INTRODUCTION

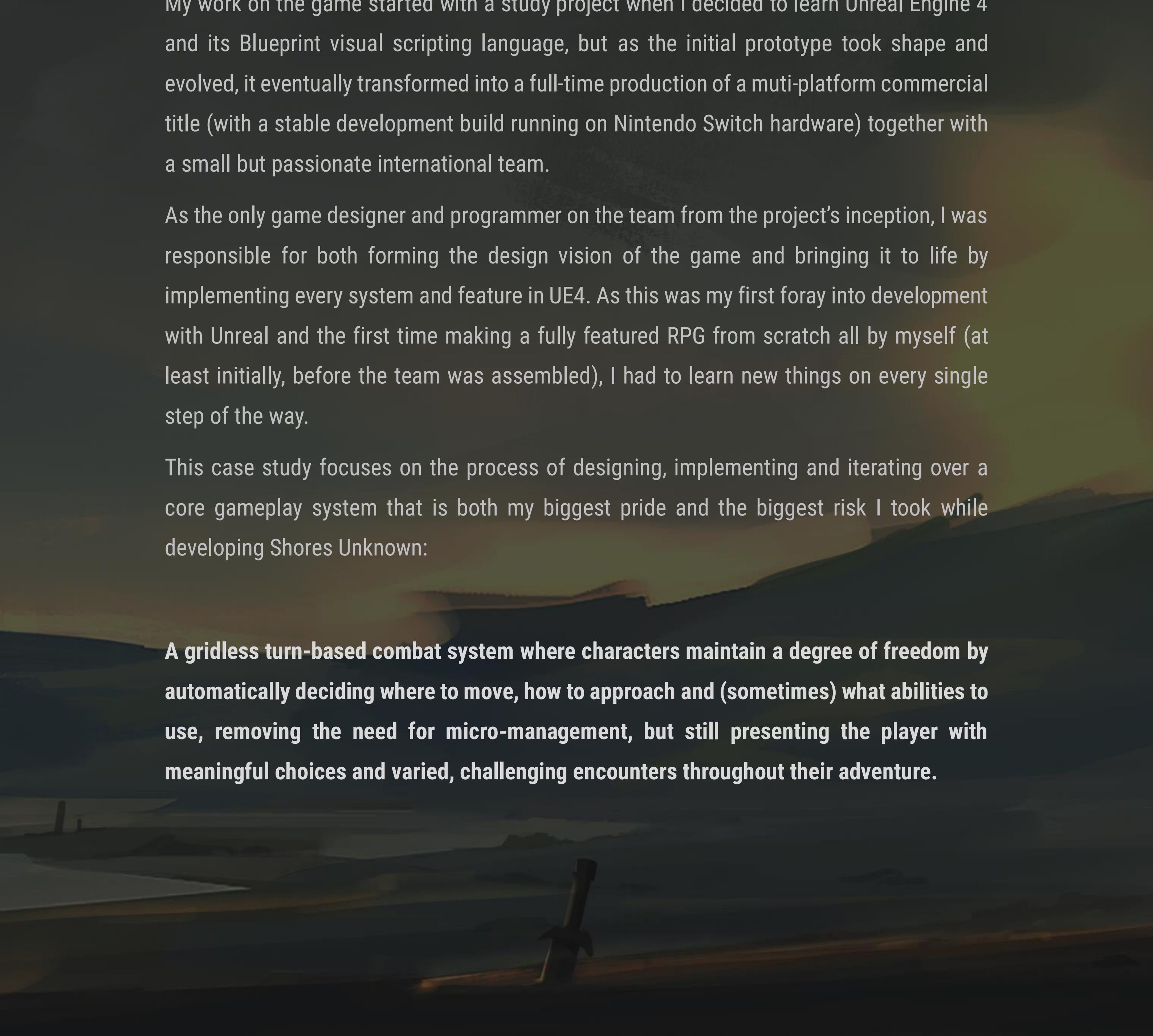
Shores Unknown is a single-player tactical RPG in a stylized fantasy setting, developed in Unreal Engine 4 over the span of 5 years and currently released in Early Access on Steam and GOG with breadth of content spanning ~20 hours of gameplay in a single playthrough.

My work on the game started with a study project when I decided to learn Unreal Engine 4 and its Blueprint visual scripting language, but as the initial prototype took shape and evolved, it eventually transformed into a full-time production of a muti-platform commercial title (with a stable development build running on Nintendo Switch hardware) together with a small but passionate international team.

As the only game designer and programmer on the team from the project's inception, I was responsible for both forming the design vision of the game and bringing it to life by implementing every system and feature in UE4. As this was my first foray into development with Unreal and the first time making a fully featured RPG from scratch all by myself (at least initially, before the team was assembled), I had to learn new things on every single step of the way.

This case study focuses on the process of designing, implementing and iterating over a core gameplay system that is both my biggest pride and the biggest risk I took while developing Shores Unknown:

A gridless turn-based combat system where characters maintain a degree of freedom by automatically deciding where to move, how to approach and (sometimes) what abilities to use, removing the need for micro-management, but still presenting the player with meaningful choices and varied, challenging encounters throughout their adventure.



INSPIRATIONS AND FIRST STEPS

While working on the vision of the combat system of Shores Unknown, I was influenced by a number of classic Western and Japanese RPG combat systems.

In particular, Infinity Engine RPGs (Baldur's Gate 1&2) and World of Warcraft were a big source of inspiration, making me embrace the classic tank – healer – DPS class role triangle and certain other mechanics (such as approach to character buffs and debuffs).

But the biggest inspiration - and the game that made me want to try designing a combat system of my own in the first place - was The Last Remnant, a highly experimental JRPG published by Square Enix in 2009, which featured a unique multi-party combat mechanic where, instead of giving direct commands and choosing the positions of each character on the battlefield, the player would instead give general orders to their troops, and each of them would automatically reposition and perform a suitable action during their turn based on the order given and the individual character's class and stats.

My first goal as I began to study UE4 was to deconstruct The Last Remnant's combat system and implement a similar system of my own in Blueprint that would operate under the following rules:

- The player can control up to 5 parties ("unions") with up to 5 characters in each
- The player gives orders on the battlefield to each union (not individual characters) by selecting a target, the type of action (attack, support and so on) and the amount of so-called "preparation points" (PPs) that each character in the union can use in their turn
- The action is paused while orders are being made, so the player can take as much time as they need to assess the situation and come up with a strategy
- When the action starts, the characters are automatically separated into one or multiple queues which govern the order of action: characters in a single queue resolve their actions one-by-one in order of their speed, while multiple queues are resolved simultaneously
- Once all the queues are resolved, the game switches to the next order phase, or the combat ends if all the characters on a single side (friend or enemy) are defeated

Furthermore, the following mechanics affect the characters' actions in the combat:

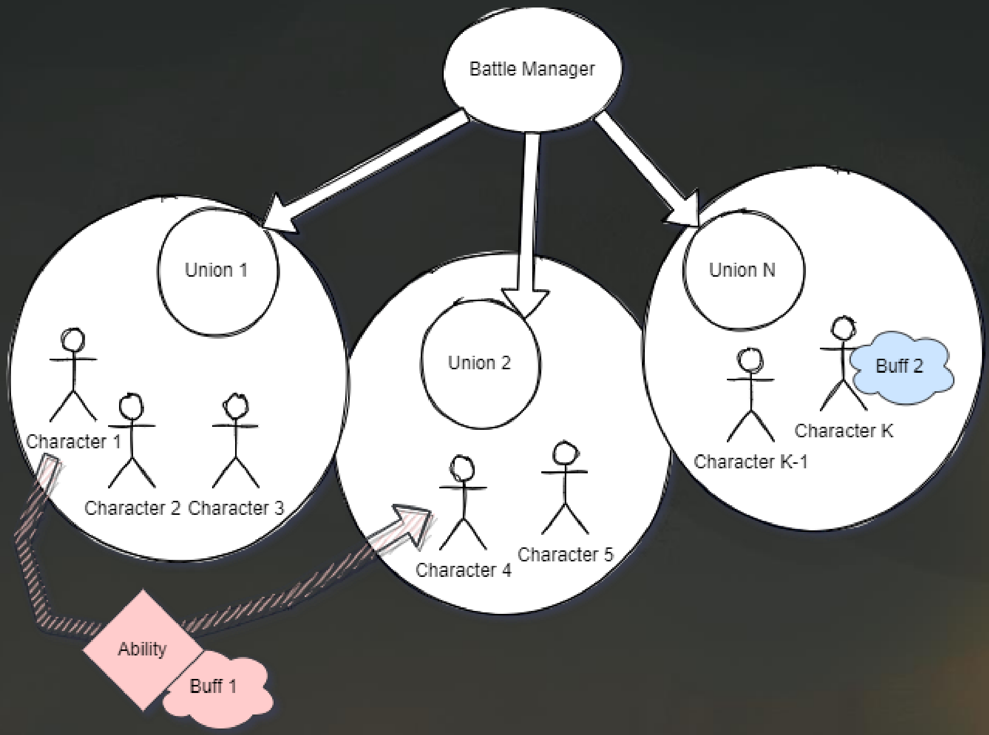
- Each character can be equipped with weapons and items freely outside of combat, and the core moveset of each character is defined by their equipment (e.g., a character with a greatsword will use abilities from Two-Handed tree, while a shieldmaiden will use abilities tied to Shield Stance tree)
- Some ability trees (usually magic) are not tied to specific equipment and can be used regardless of what the character is wearing
- As the characters use their available abilities, they gradually accumulate experience, increasing their core stats (Strength, Versatility and Intelligence), and even automatically unlock new abilities in the tree they're specializing in

As I quickly found out, out of the box UE4 didn't provide much in terms of existing functionality for turn-based games, RPGs or games with bird's eyes view and large number of acting characters on-screen. While working on the first iteration of combat system, I had to research, design, and implement the following core classes and logic:

- Battle manager actor class, that controls order and action phase change, party spawns and action order calculations
- Union (party) actor class, which encapsulates all the logic and variables pertaining to a single party in battle: queued action type, total party health, speed and other relevant aggregated stats, and general positioning on battlefield. Additionally, NPC unions are possessed by an AI Controller which governs strategic decision-making of each party
- Character actor class, which manages combat stats of each character, assesses and spawns abilities during actions, tracks buff/debuff effects. Each character is possessed by a Behavior Tree-driven AI controller that guides their movement during combat in real-time.[1]

[1] In the early iterations of combat system, although battles were turn-based, the characters would still actively maneuver around the field in real-time even outside of their action phases to simulate a feeling of "skirmish", a behavior that was later greatly toned down to improve the readability of combat during order phases.

The interaction between different battle actors can be summarized by the following diagram:



As the primary way one character can affect another, abilities are a major part of Shores' combat system. Instead of using one of the existing solutions on Marketplace at the time or attempting to integrate Epic's own Gameplay Ability System, I opted to design my own ability framework, both as a learning experience and to ensure maximum flexibility of a system made to adhere to my game's unique combat mechanics.

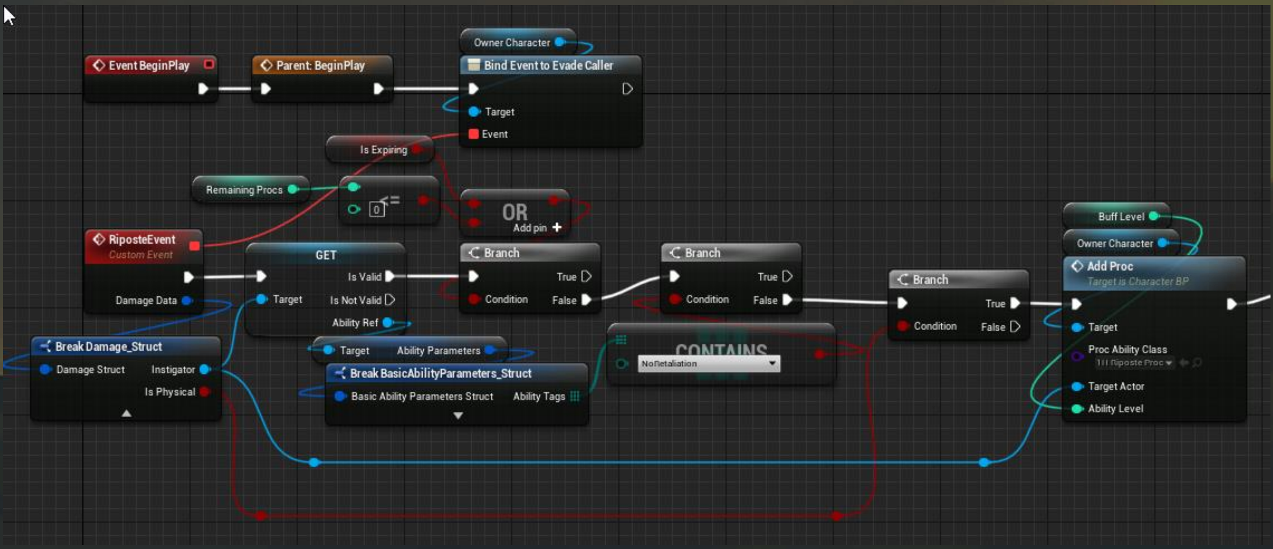
I wanted my system to allow myself or any other member of the dev team to easily create new abilities in a modular fashion, fully in Blueprint, and allowing for implementation of a wide range of mechanics: from basic damage actions to battlefield wide AoEs, from one-shot healing effects to persistent party buffs and character summons.

I also wanted to make sure to utilize the parent-child hierarchy of Blueprint classes, going from generic parent types ("damage", "healing" and so on) to highly specialized unique abilities (e.g., "dragon fire breath" ability class inheriting from "party-wide AoE", which in turn inherits from "damage"). Of course, "damage" and "healing" also inherit from "parent ability" actor class.

Buff actors also used a similar parent-child hierarchy approach, with "Buff Parent" class implementing the shared logic for all buffs, which could then be expanded in child classes.

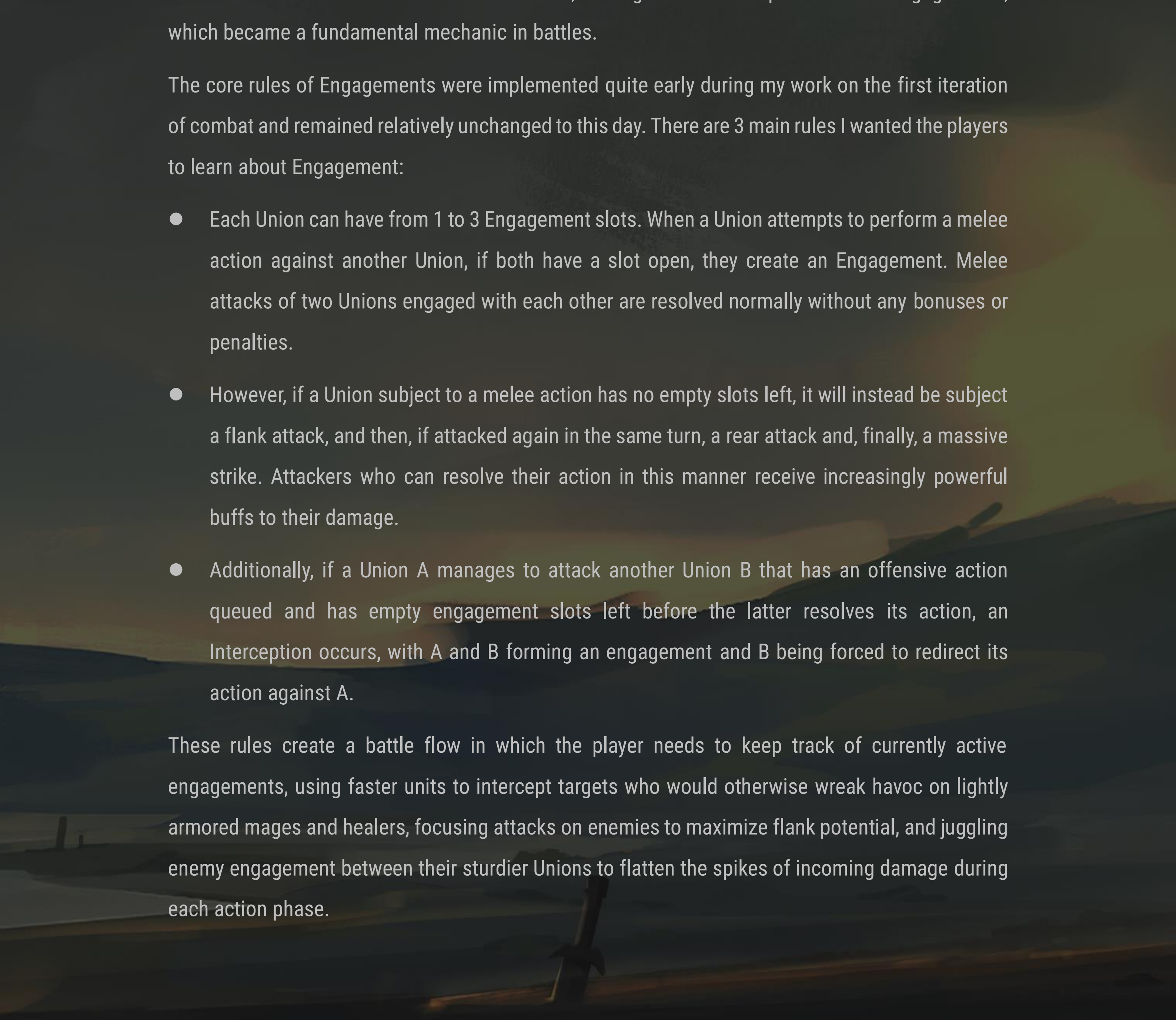
To keep things controlled, I established the following ground rules for character, ability, and buff interactions:

- Only a character actor can spawn ability actors, and only in the order governed by battle manager
- While an ability is spawned and active, it takes temporary control of the caster character’s animation and movement
- Only one character within a single queue can be acting (= “under control of a spawned ability”)
- A buff can be spawned by an ability or, in some specific cases, by another buff
- A buff can also request an ability to be spawned by a character as a “proc” under certain conditions, with timing of such spawns controlled by Battle Manager, tied to character action states via event dispatchers (for example, “if this character dodges an attack, they will perform a riposte attack at the end of current action”)[2]



Owner triggering an Evade event causes this buff to queue a proc if the conditions are met

[2] At the early stages, the game only had a simple proc type that would happen immediately after the action that triggered it was resolved. This system was later expanded to include a lot more conditions, e.g., On Character Death and On Damage Target, and extra timeframes in which a proc could be resolved, such as at the beginning or end of an action phase.



ENGAGEMENT SYSTEM

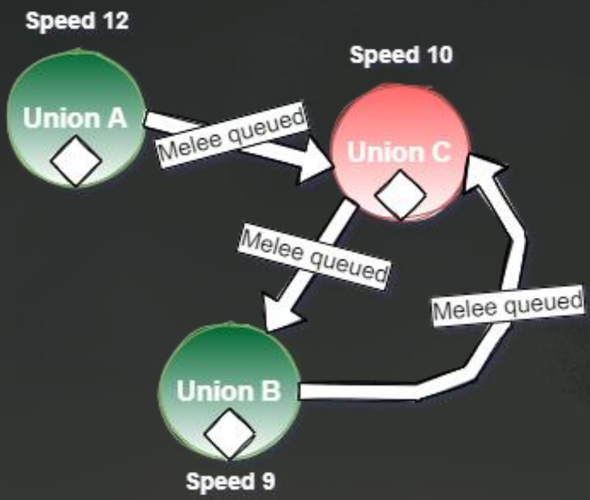
Although the positioning aspect of classic turn-based combat systems is not present in Shores Unknown, I still wanted to have the characters to be able to flank, intercept and backstab their targets. These mechanics also needed to be tied to the player's choices in combat – albeit in a less direct manner. Based on these considerations, I designed the concept known as "engagement", which became a fundamental mechanic in battles.

The core rules of Engagements were implemented quite early during my work on the first iteration of combat and remained relatively unchanged to this day. There are 3 main rules I wanted the players to learn about Engagement:

- Each Union can have from 1 to 3 Engagement slots. When a Union attempts to perform a melee action against another Union, if both have a slot open, they create an Engagement. Melee attacks of two Unions engaged with each other are resolved normally without any bonuses or penalties.
- However, if a Union subject to a melee action has no empty slots left, it will instead be subject a flank attack, and then, if attacked again in the same turn, a rear attack and, finally, a massive strike. Attackers who can resolve their action in this manner receive increasingly powerful buffs to their damage.
- Additionally, if a Union A manages to attack another Union B that has an offensive action queued and has empty engagement slots left before the latter resolves its action, an Interception occurs, with A and B forming an engagement and B being forced to redirect its action against A.

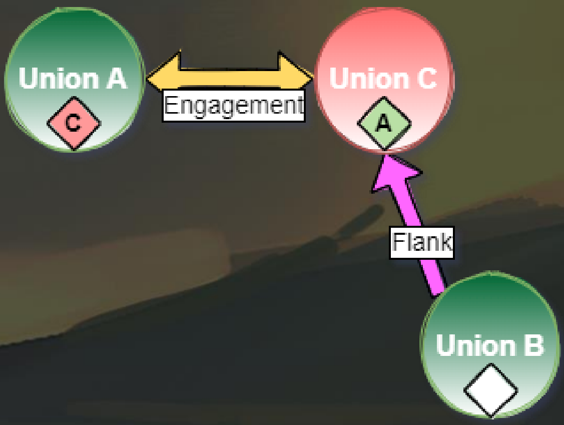
These rules create a battle flow in which the player needs to keep track of currently active engagements, using faster units to intercept targets who would otherwise wreak havoc on lightly armored mages and healers, focusing attacks on enemies to maximize flank potential, and juggling enemy engagement between their sturdier Unions to flatten the spikes of incoming damage during each action phase.

Here's a basic example of how Engagement comes into play when resolving actions of two friendly unions A and B against a hostile union C.



During the **Order phase**, all three Unions have a melee action queued. Union **A** (speed 12) wants to attack the hostile Union **C**, Union **C** (speed 10) wants to attack Union **B**, and Union **B** (speed 9) also targets Union **C**. None of the Unions have any prior engagements.

Here's how the situation is resolved during the **Action phase**:

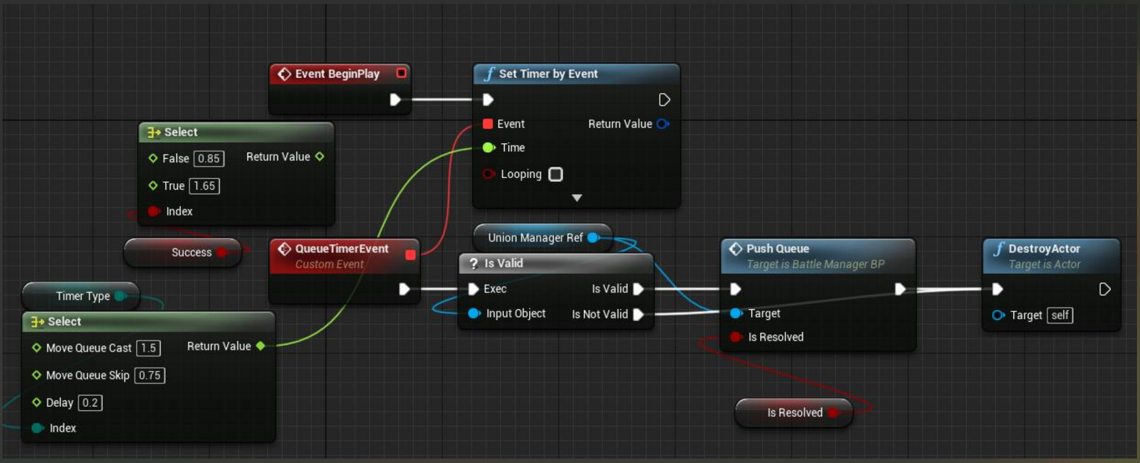


1. As it is the fastest, Union **A** moves first. Since Union **C** is targeting a different Union with its queued action, an interception occurs.
2. Unions **A** and **C** are now engaged. **C** is attacked by **A**, as was planned.
3. Union **C** moves next. As it has been intercepted by **A**, it must use the melee action to attack **A** instead of **B**.
4. Union **B** is the last one to act. Because **C** no longer has any engagement slots left after being intercepted by **A**, **B**'s action against it is resolved as flank attack, dealing additional damage.

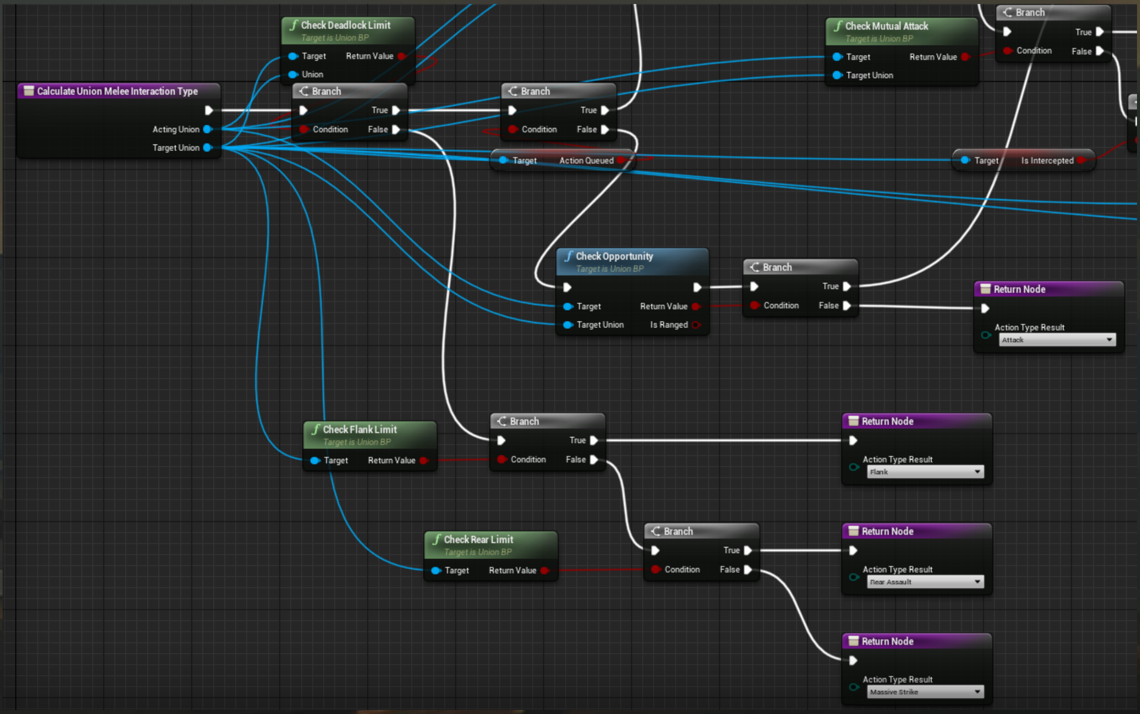
It's important to note that once the Action phase begins, individual Union orders are locked in and can no longer be changed by the player. Instead of reactive play, strategic thinking and the ability to plan multiple steps ahead become paramount.

Under the hood, the Battle Manager actor takes care of everything related to Action phase resolution. This includes generating an array of acting Unions sorted by Speed stat and then resolving each Union's action until the Acting Union array is empty.

Timing of action resolution was important, so rather than relying on Actor Tick, I used custom Timer logic to ensure full control of when one action ends, and another begins. As the first iteration of combat could have multiple queues per action phase (and therefore multiple actions resolving simultaneously), the action timer logic was implemented in a separate QueueHelperTimer actor:

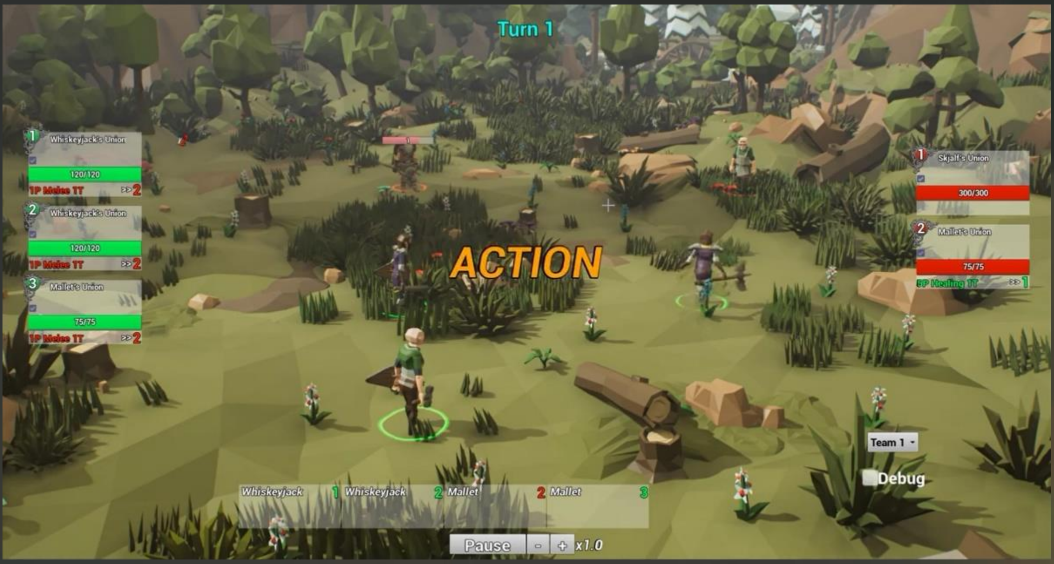


Engagement logic is also implemented as a function in BattleManager class:

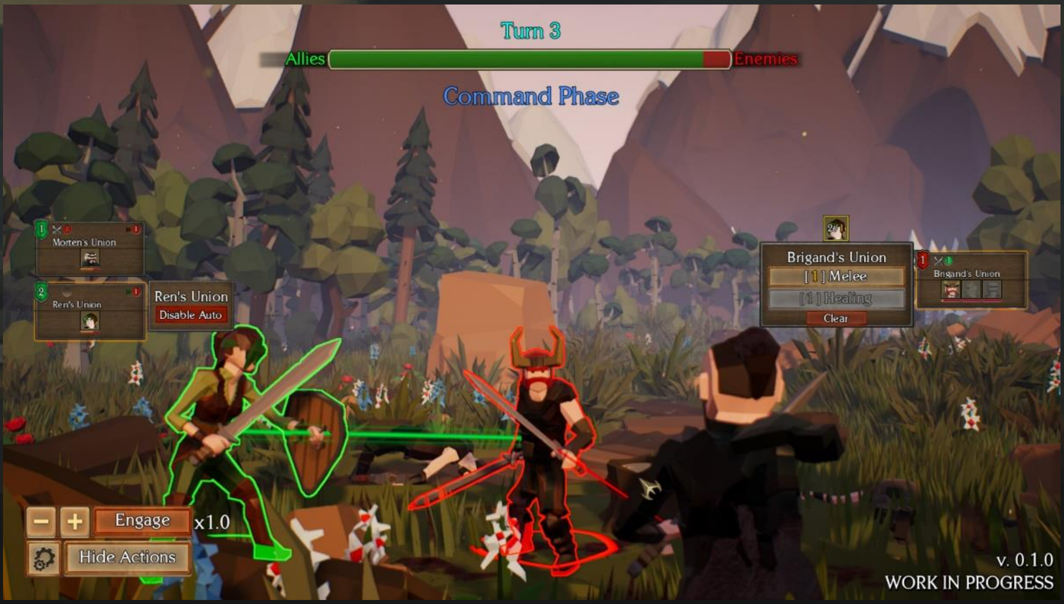


PRE-ALPHA AND PIVOT

After the first couple months of working on the game (which was then known as “Project Shore”), I had a working combat system prototype using placeholder assets. At the same time, the team size had increased to 3 people, adding a level designer and a scenario writer.



The combat felt fun in our internal tests, so we decided to push for a pre-alpha build which would feature about an hour of gameplay, including the early story beats and a few battles of different scale and difficulty. In addition to the scripting work needed for exploration gameplay, as well as designing and tuning the combat encounters, I also had to work on the UI implementation using Unreal’s UMG.



By the end of 2017, after passing several internal tests, our pre-alpha build was released on [Indiedb](#) and [Gamejolt](#), where I also maintained a devlog series.

Project Shore received a warm reception from the players and was noticed by AlphaBetaGamer (who kindly recorded [a full playthrough video of the build](#) and wrote [an article with their impressions](#)), as well as getting noticed by several publishers.

It was ultimately the combination of player and publisher feedback, as well as watching our players play the game on streams, that made me reconsider some of the choices made at the early stages of development. The main takeaways were these:

- Multiple queues during the action phases made large-scale battles feel chaotic and hard to follow, even when using the Pause functionality that we provided the players with to stop the action at any time
- The Last Remnant-styled generic commands to unions resulted in outcomes that were hard to predict, with no guarantee that a character would use an ability the player expected them to
- Between a morale bar (which was usually assumed to be some sort of "overall health bar"), union formations, buffs, and various bonuses for "technical" actions such as flank attacks, there were simply too many variables for most players to keep track of

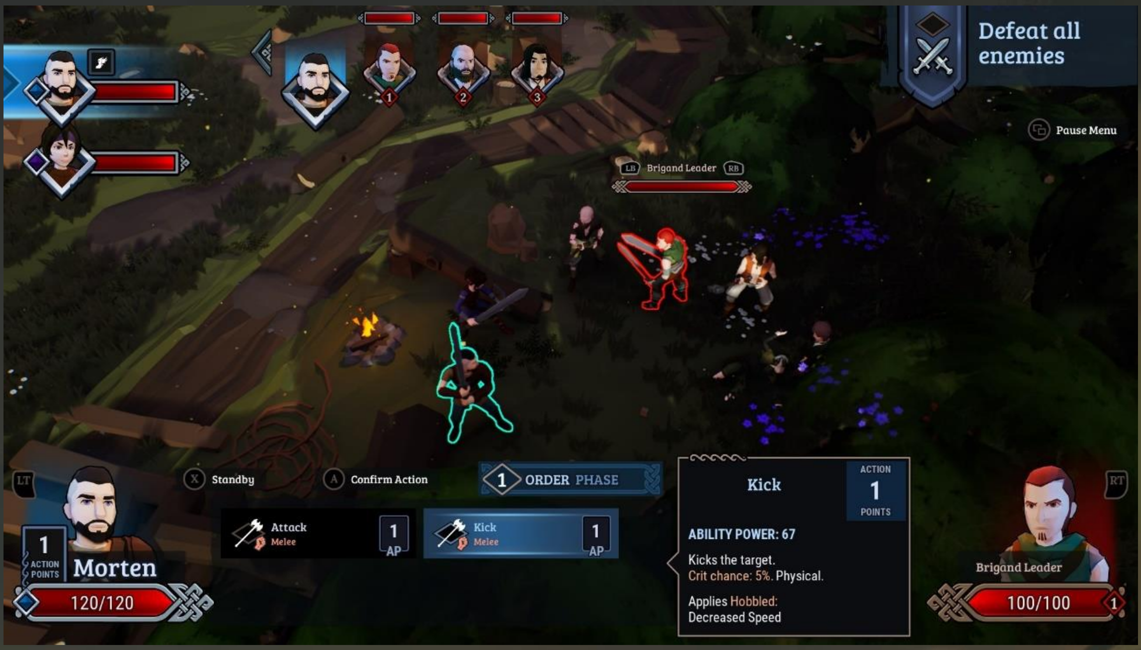
As I was already looking at Shores as a commercial project by then, I made it my next goal to create the next iteration of combat which took care of these three problems while keeping intact the fun factor of a micromanagement-less strategic battle planning and the fantasy of running your own mercenary party with minds of their own.

In the second iteration of the system, after extensive playtesting and discussion with our publisher, I decided to remove the multi-party entirely, reducing the number of units under player control to a maximum of five. This was a drastic change, bringing the game's presentation closer to classic party-based RPGs, but it also allowed me to give the player much more control as we made it possible to queue specific abilities instead of generic commands.

This change also resulted in a combat flow that was much easier to follow for players. All the actions were now resolved in a single queue, and the characters would always prioritize the ability chosen by the player during order phase when assessing their actions.

At the same time, the characters kept a part of their free will, as I retained the automatic positioning layer of the combat and the Engagement mechanic intact – only instead of Unions, it was now resolving the interactions between individual characters.

Under the hood, the Union actor layer remained to perform its old duties of spawning characters, strategic decision-making AI calculations and general positioning of units on the battlefield, though limiting it to a single character simplified the logic significantly.



The final shape of combat UI: timeline widget showing the action order was implemented in UMG

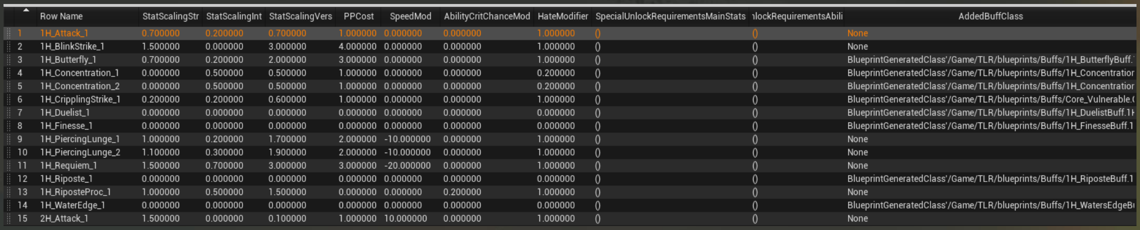
The change also untied my hands when it came to designing new abilities: as both the player and AI now had direct control over what action to use, I was now able to design and implement effects with a much bigger impact on the battlefield. Without 20+ units on the screen, having a powerful enemy unchecked on the field no longer resulted in unfair losses for the player. Such enemies now needed to be identified and taken care of as priority targets, but the players also received a few new tools to turn the situation on the battlefield to their favor – from slow effects to taunts, on-demand absorb shields and modal passive abilities which could be toggled on or off depending on the situation.

Morale bar and party formations were also removed during the pivot – after examining player behavior and feedback, I felt that neither mechanic added anything meaningful to the combat that was not already being achieved by clever use of buff abilities and paying attention to action timeline.

EXCEL-POWERED AND DATA-DRIVEN

Due to my previous experience working as a game designer on commercial projects, I wanted to utilize Excel datasheets in my game as much as possible in everything related to numbers and balancing. Luckily, UE4 provided a way for me to connect my Excel workflows and in-engine work using DataTable assets. Generally, I would use a VBA macro in Excel to export my data into .csv format which could then be imported into UE4 and used in-game.

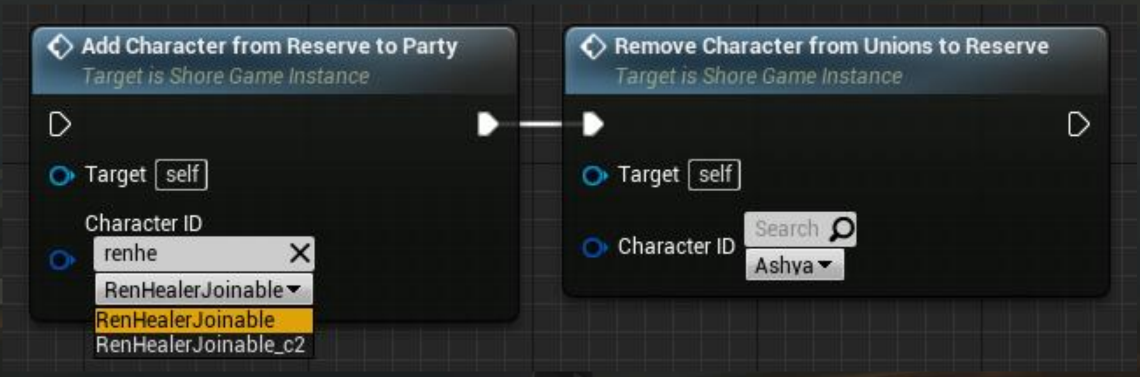
This proved particularly effective when working on balancing combat encounters and fine-tuning character abilities, as all the interactions between characters in combat are number-driven, and given the scale of battles in our game (which, at that stage, could involve 20-50 units at once), a way to quickly tweak multiple values at once was needed.



	Row Name	StatScalingStr	StatScalingInt	StatScalingVers	PPCost	SpeedMod	AbilityCritChanceMod	HateModifier	SpecialUnlockRequirementsMainStats	ılockRequirementsAbili	AddedBuffClass
1	1H_Attack_1	0.700000	0.200000	0.700000	1.000000	0.000000	0.000000	1.000000	()	()	None
2	1H_BlinkStrike_1	1.500000	0.000000	3.000000	4.000000	0.000000	0.000000	1.000000	()	()	None
3	1H_Butterfly_1	0.700000	0.200000	2.000000	3.000000	0.000000	0.000000	1.000000	()	()	BlueprintGeneratedClass'/Game/TLR/blueprints/Buffs/1H_ButterflyBuff.
4	1H_Concentration_1	0.000000	0.500000	0.500000	1.000000	0.000000	0.000000	0.200000	()	()	BlueprintGeneratedClass'/Game/TLR/blueprints/Buffs/1H_Concentratior
5	1H_Concentration_2	0.000000	0.500000	0.500000	1.000000	0.000000	0.000000	0.200000	()	()	BlueprintGeneratedClass'/Game/TLR/blueprints/Buffs/1H_Concentratior
6	1H_CripplingStrike_1	0.200000	0.200000	0.600000	1.000000	0.000000	0.000000	1.000000	()	()	BlueprintGeneratedClass'/Game/TLR/blueprints/Buffs/Core_Vulnerable.(
7	1H_Duelist_1	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	()	()	BlueprintGeneratedClass'/Game/TLR/blueprints/Buffs/1H_DuelistBuff.1H
8	1H_Finesse_1	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	()	()	BlueprintGeneratedClass'/Game/TLR/blueprints/Buffs/1H_FinesseBuff.1
9	1H_PiercingLunge_1	1.000000	0.200000	1.700000	2.000000	-10.000000	0.000000	1.000000	()	()	None
10	1H_PiercingLunge_2	1.100000	0.300000	1.900000	2.000000	-10.000000	0.000000	1.000000	()	()	None
11	1H_Requiem_1	1.500000	0.700000	3.000000	3.000000	-20.000000	0.000000	1.000000	()	()	None
12	1H_Riposte_1	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	()	()	BlueprintGeneratedClass'/Game/TLR/blueprints/Buffs/1H_RiposteBuff.1
13	1H_RiposteProc_1	1.000000	0.500000	1.500000	0.000000	0.000000	0.200000	1.000000	()	()	None
14	1H_WaterEdge_1	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	()	()	BlueprintGeneratedClass'/Game/TLR/blueprints/Buffs/1H_WatersEdgeBı
15	2H_Attack_1	1.500000	0.000000	0.100000	1.000000	10.000000	0.000000	1.000000	()	()	None

AbilityParametersScalable DT in the engine

Characters, abilities and buffs were then built to to populate their values from the respective data tables based on a unique ID. This was originally FName-based, but I later implemented custom C++ logic to allow dropdown lists for all the major datatable-driven variables in editor details and graph nodes to reduce the probability of human error and speed up the workflow.


Add Character from Reserve to Party
Target is Shore Game Instance
Target
self
Character ID
renhe
RenHealerJoinable
RenHealerJoinable
RenHealerJoinable_c2
Remove Character from Unions to Reserve
Target is Shore Game Instance
Target
self
Character ID
Search
Ashya

An example of custom editor node with data table-populated dropdown selection and filter logic

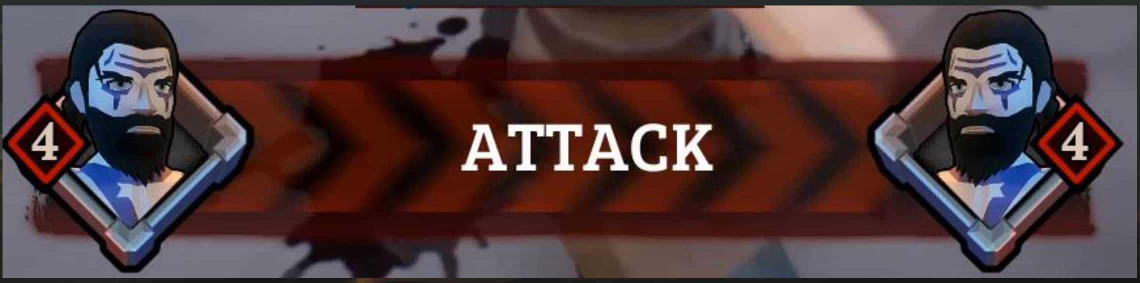
PROBLEMS TO SOLVE

Of course, there were many challenges to overcome during the development, from the very first steps of battling “access none” reference errors as I was learning the ropes in UE4, to learning to profile and optimize my logic to make it run without hitches on Nintendo Switch hardware.

At one point, about a year into the development, a major engine bug related to BP structures caused me to lose days of progress. Because of it, I had to reimplement most of the game’s structs in C++, which also required numerous changes in actor classes (including having to re-implement the base classes for characters and abilities in C++). But in the end, the game became better for it, making it easier than ever to expand the logic and add new functionality.

At a different time, as I was battling a crash when loading a particularly large level on Switch, I realized that using hard references instead of soft ones for assets in data tables resulted in every single character model being loaded 100% of time, taking ~500MB of memory. This resulted in a refactor that reduced the memory footprint of the character table to only 7MB, both fixing the crash issue I was experiencing and noticeably improving the loading times of the game.

Finally, one of the funniest bugs was encountered when working on ability reassess logic for characters in combat. Normally, if a character had a healing ability queued during Order phase, but no valid targets to use it on during action resolution, they’d automatically switch to an offensive ability against an eligible target so as not to waste the action. However, something went wrong with the alignment check when searching for a target, and to our surprise, all healers in the game suddenly started to practice self-flagellation.



CONCLUSION

Current version of Shores Unknown features:

- More than 50 combat encounters, ranging from bandits to a giant undead crow-man
- Player parties of up to 5 characters with fully customizable equipment clashing with enemy squads sometimes as large as a dozen of enemies
- 10 distinct ability trees (groups) that can be learned by player characters
- 300 abilities and 200 buff effects – from simple sword pokes to comets falling from the sky, minion summons and battlefield-wide ice storms
- Game logic optimized to run at stable 30 fps (not exceeding 33.3ms on game thread) on Nintendo Switch hardware

It's hard to overstate how empowered I felt working with tools provided by Unreal Engine 4, as it enabled me to unify my experiences as a player and game designer and use my technical knowledge to bring my vision of the game to life. The result of my work is an original combat system that, although inspired by industry heavyweights, now stands on its own and offers a truly unique experience. And it also exists as proof that it's, indeed, possible for all the logic in a 20+ hour long RPG to be implemented by a single technically minded designer almost entirely in Blueprint.