OULUN AMMATTIKORKEAKOULU

Venla Jokikokko

THE POTENTIAL OF UNREAL ENGINE 5 IN GAME DEVELOPMENT

Exploring the Capabilities of the Unreal Engine

THE POTENTIAL OF UNREAL ENGINE 5 IN GAME DEVELOPMENT

Exploring the Capabilities of the Unreal Engine

Venla Jokikokko
Bachelor's Thesis
Spring 2023
Information Technology
Oulu University of Applied Scienc

ABSTRACT

Oulu University of Applied Sciences
Information Technology

Author(s): Venla Helena Jokikokko
Title of the thesis: The Potential of Unreal Engine 5 in Game Development
Thesis examiner(s): Jouni Juntunen
Term and year of thesis completion: Spring 2023 Pages: 71

Independent game developers are challenged to develop high-quality games that can compete in the market as the demand for interactive and visually stunning games continues to grow. With a powerful toolkit and intuitive workflow, a large asset library and community resources, advanced graphics and physics capabilities together with cross-platform development support, Unreal Engine is a powerful and versatile solution to this challenge. This thesis explored the objectives and benefits of using the Unreal Engine. Utilizing the engine capabilities to create high-quality, engaging games and creating immersive and visually stunning experiences was the focus of this thesis.

An Unreal Engine-powered game development project programmed in C++ was used in the thesis. Based on the results, it was evident that Unreal Engine provides an effective solution for independent game developers. It includes creating successful and profitable games that can lay the groundwork for future projects. In addition, the optimization of game assets and characters for Unreal Engine 5 were evaluated in the thesis. The findings demonstrate the importance of utilizing optimization techniques in game development as well as the essential role of Unreal Engine and C++ in this process.

In conclusion, this thesis demonstrated the benefits and potential of working with Unreal Engine as a new independent game developer. It also provided insights into game development and the use of Unreal Engine as a game engine.

Keywords: Game Development, Unreal Engine, C++ programming, Independent game development, Game engines, Game asset creation, and Game optimization.

CONTENTS

1 INTRODUCTION

As a role-playing game with roots in Finnish and Nordic mythology, Tuonela is a game loosely based on the national Finnish epic, Kalevala, and takes inspiration from Finnish and Nordic mythology. During the game, the player will encounter a variety of horrors and monstrosities that are traditionally found in Finnish and Nordic mythologies. As a player, you are in the position of being the only mortal who has managed to escape the underworld of Tuonela alive. The mission you are given is that of a suitor hoping to attract the attention of the daughter of the Queen of the Underworld. Will that be enough for you to cross the precarious bridge to return to the underworld? Or do you have different intentions in mind? As you return to this once-picturesque landscape, it is up to the player to escape the horrors that await you.

The team working on this game demo consists of the author (of this thesis) and Daniel Carey, a long-time friend, and fellow game developer enthusiast. We came together to develop this demo because one of us has a passion for environmental artistry and level design. The other has technical knowledge of gameplay programming as well as a passion for character design and creation. Our aim is to bring relatively unknown Finnish mythology to the forefront so that a larger audience can learn and appreciate it. The goal of this thesis is to demonstrate how we used Unreal Engine 5 as our dedicated game engine for development. The demo itself will not be completed upon completion of this thesis. Instead, it will continue as this is a lengthy project that will still need a lot of time to be completed. The reason why Unreal Engine 5 was chosen as a game engine is that it is free and supports high-quality graphics. Furthermore, Unreal Engine uses C++ as its primary language. As an aspiring game developer, learning C++ is very helpful since most big studios develop their games with it. There is a large developer community for Unreal Engine that has also been of assistance when some issues arise during development.

In Finnish mythology, Tuonela is the gateway to the underworld. We used that term in naming the game Tuonela as it is symbolic of the underworld in mythology. A previous version of Tuonela had been made with Unreal Engine version 4, which was more blueprint-based when it came to the mechanics and systems of the game. The goal is to create a well-balanced role-playing game of old-time charm with some more modern styles of role-playing game methodologies, as well as some additional features such as puzzle solving. In addition, our environmental design has undergone significant changes since migrating to Unreal Engine 5. The Tuonela demo consists of two

levels: MAIN_village, where most of the gameplay will be, and MAIN_gateway, which is meant to serve as a teaser for the full game content of Tuonela. It is planned to be a 30 to 45 minutes long game demo that would be published on Steam upon its completion.

2 UNREAL ENGINE

Developed by Epic Games, Unreal Engine is a game engine that enables creators in all industries to create next-generation 3D content and experiences with freedom, fidelity, and flexibility. It prides itself on being the most advanced and open-source tool for real-time 3D graphics. Besides providing tools for games, Unreal Engine also allows a wide variety of industries to take advantage of the engine, most notably the film and television industries. The Unreal engine was first developed and released in 1998 for a first-person shooter game that was made only for PC. Besides PC, Unreal now supports multiple platforms such as PlayStation 4 and 5, Xbox One, Xbox Series X, and Nintendo Switch. The engine itself is written in C++, which is also used as a programming language when implementing functionalities and features for the games. There is also a visual programming language called blueprints if a more visual presentation of logic is preferred by the developer. (Epic Games. 2004-2022ai.)

The minimum hardware and software specifications mentioned here are for Unreal Engine 5. For the Windows system, the minimum hardware requirements are Windows 10, 64-bit version 1909 revision. The processor should be Quad-core Intel or AMD, 2.5GHz or faster. It is recommended that the computer has at least 8GB of memory and a DirectX11 or 12 graphics card with the latest drivers installed. The minimum software requirements are DirectX runtime and for the development of Visual Studio version 2019 or 2022. When developing for iTunes, it must be version 12 or higher. A macOS environment requires the latest macOS Ventura operating system, quad-core Intel, or AMD, 2.5GHz or faster, and 8GB of memory. The video card must be a Metal 1.2 compatible graphics card. For developing with macOS XCode 13.4.1 is required. Lastly, Ubuntu 22.04 with Quad-core Intel or MSD 2.5 GHz or faster is recommended for Linux development. Minimum memory of 32 GB RAM, NVIDIA GeForce 960 GTX or higher video card driver, and video RAM of 8 GB or more. RHI version needs to be Vulkan either AMD 21.11.3 or higher or NVIDIA 515.48 or higher. (Epic Games 2004-2022j.)

2.1 Getting started with Unreal Engine

Unreal Engine cannot be installed on your computer without the Epic Games launcher. The Epic Game Launcher can be downloaded from the Unreal Engine website. Additionally, users must create epic game accounts that can be used to log into the epic game launcher. Users can create Epic Games accounts by logging in with their email or other associated social media accounts such as Facebook, Google, Xbox Live, PlayStation Network, Nintendo, Steam, or Apple accounts. Windows, macOS, and Linux are supported (Unreal Engine 2004-2023k.). Using the Epic game launcher, install Unreal Engine by clicking on the Unreal Engine tab that is highlighted (FIGURE 1). A new installation window will appear where you can choose where to install Unreal Engine. Platform support, starter content, debug symbols, engine source code, and some other features can be chosen to install along with Unreal Engine (Epic Games 2004-2022j.).

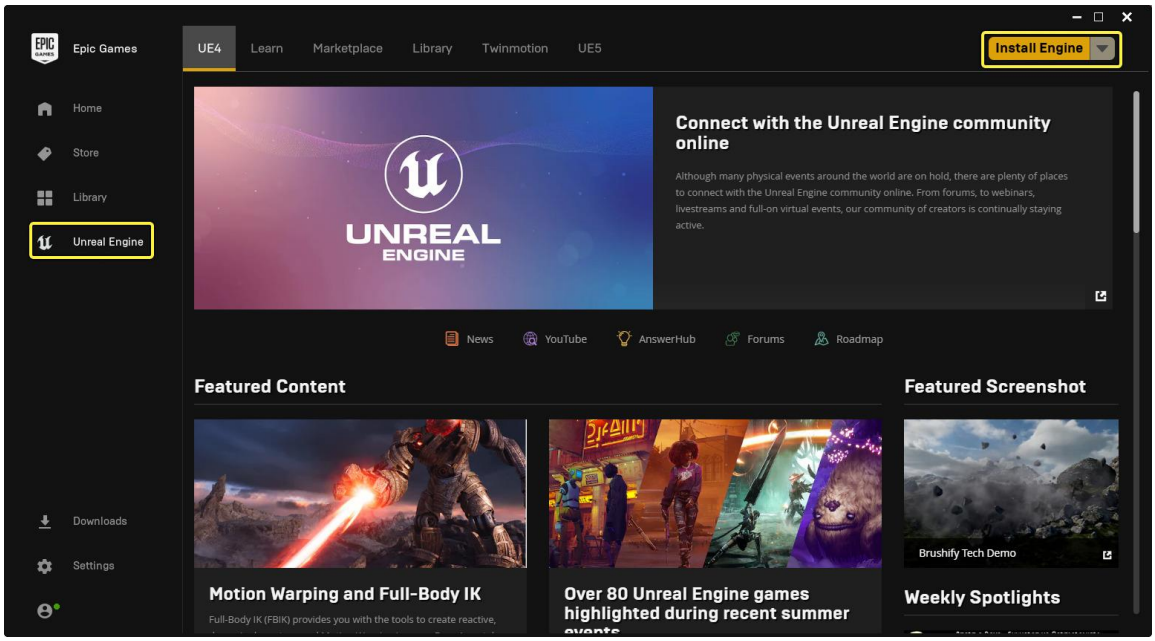


FIGURE 1. Install Unreal Engine (Epic Games 2004-2022l.).

The different game engine versions can be added to the "Library" tab at the top of the launcher after Unreal Engine is successfully installed. Unreal Engine supports multiple versions of the engine to be used at the same time as seen in the figure below. The new engine can be installed by clicking the plus icon next to the “Engine version” text which is highlighted in yellow. (Epic Games 2004-2022l.)

Once the engine is launched the project can be created from the Unreal Engine project browser view. In yellow box number one, (FIGURE 2) you will see different development category options. If you click on category games, readymade templates will be shown next to the development categories as indicated by yellow box number two. Readymade templates have features and functionalities already implemented for faster project implementation. This includes options such as First-person, third-person, top-down, vehicle, handheld augmented reality, or virtual reality templates. However, blank projects can also be created if needed. Once the preferred template is chosen developers can choose if they prefer Blueprint or C++ as the development method and what platform the initial development will be done on. Also, preset quality projects should be initialized. Furthermore, there are checkboxes to choose if starter contents should be included and raytracing turned on. Yellow box number three indicates these options. The project location and name can also be configured in yellow box number four (FIGURE 2). (Epic Games- 2004-2022d.)

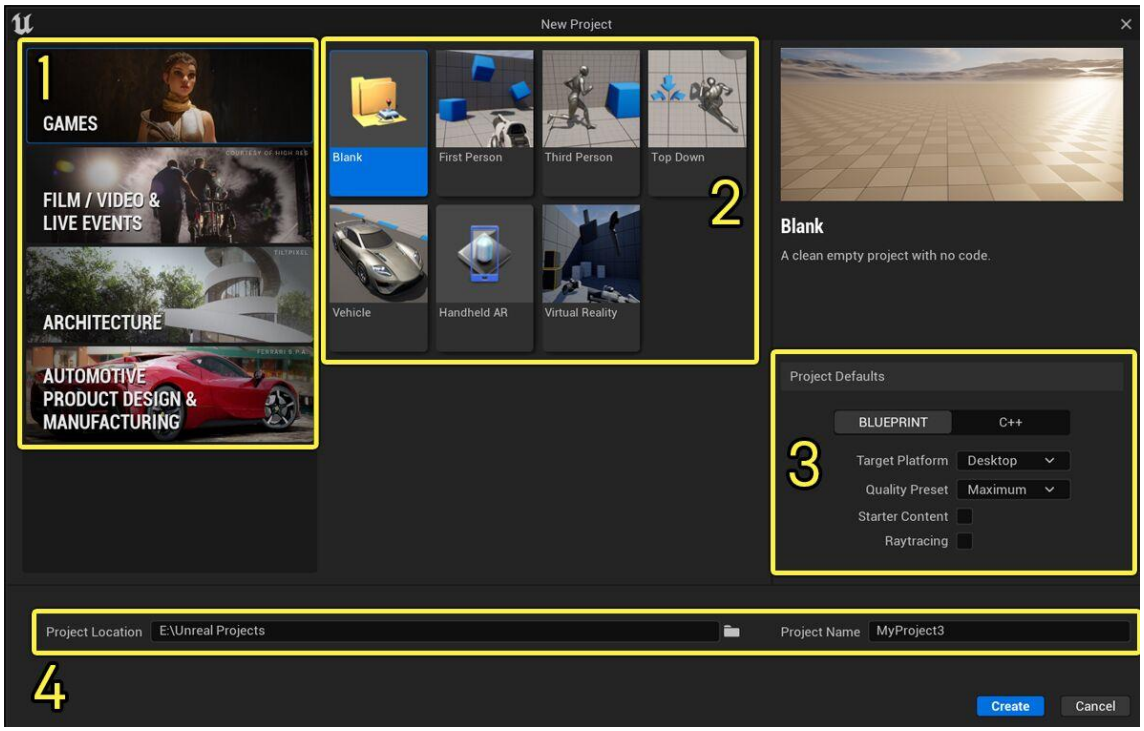


FIGURE 2. Creating a New Project. (Epic Games. 2004-2022d.)

The Unreal Engine 5 project interface level editor has seven common elements that are seen in figure 3 below. These elements consist of different windows that can be moved around in the editor interface freely and placed whenever users prefer them to be in the level editor. The Unreal Engine 5 arranges the interface by default when the project is created, and this interface arrangement is looked at in more detail. Starting from number one to number seven, the first window on the editor

is a menu bar with editor-specific functionalities and commands (FIGURE 3). The second window marked as number two is the main toolbar. It has shortcuts to the most used tools as well as a button to enter play mode. When the play button is pressed the game is run inside the Unreal Engine 5 editor. The most dominant feature of the interface is the viewport as indicated in number three. It is essentially just a view of a level that is being created. In the level viewport, static meshes, and actors can be panned and rotated as well as scaled. The content of the viewer can be displayed in two different ways. From the perspective view, content can be displayed from different angles in 3D. The orthographic view looks down on the level from one of the main axes in 2D view. Number four is a content drawer button that can be used to toggle to show all the assets in the project (FIGURE 3). Number five is a bottom toolbar that has shortcuts to the command console, output holds the status of source control and provides functionality for derived data. Window number six is called the outliner which has the hierarchical tree of all the game objects of the current game level that is seen in the viewport. The outliner panel enables you to quickly hide or show actors by clicking the eye icon. You can also access the context menu for the actor to perform more specific actions such as create, move, and delete content. Lastly, windows indicated as number seven is the details panel that will appear on the interface when an actor on the viewport or outliner is selected. The details panel will feature all the actor properties. For example, position in the level, static mesh that the actor has been assigned to, what kind of material it uses physics settings, and much more. (Epic Games. 2004-2023ad.).

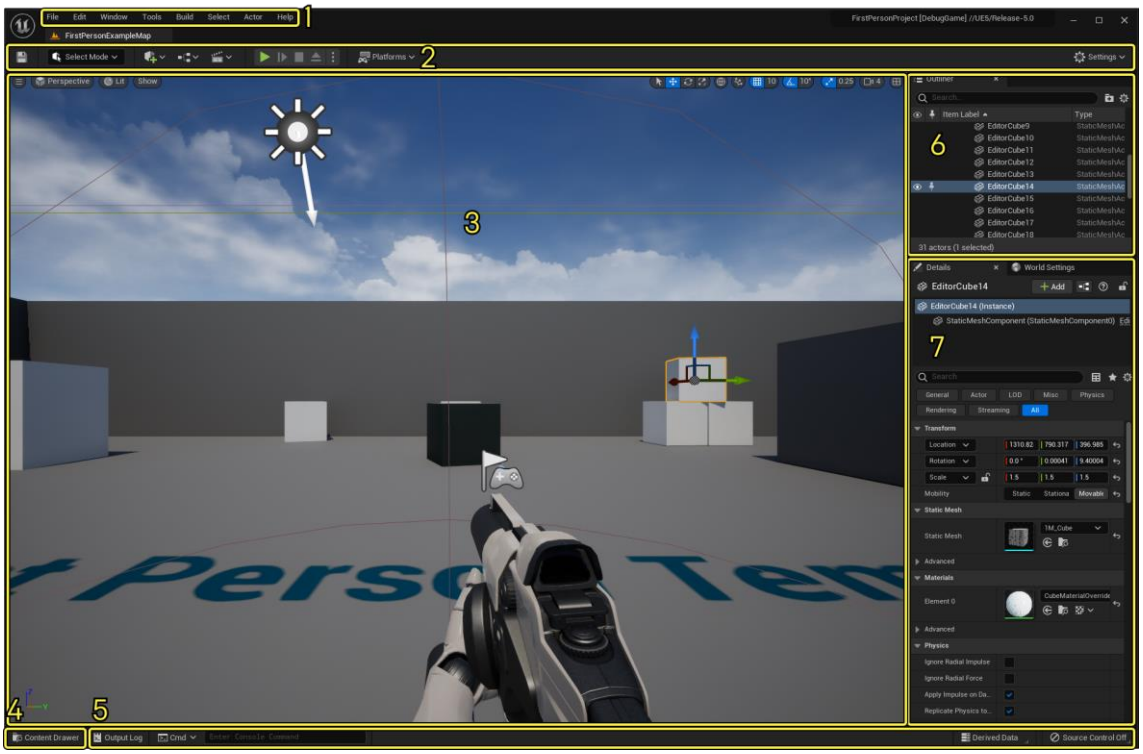


FIGURE 3. Unreal Editor Interface. (Epic Games. 2004-2023ad.)

After examining the general layout of the level editor's interface, it becomes clear that there is an instrumental part of the interface that can be used for creating, editing, and modifying game actors. Below is figure four, which shows the elements inside the main toolbar. From left to right, the save button is indicated as box number one (FIGURE 4). Next to it as number two is a select mode where modes can be switched quickly between different content editing types in the level. Inside the third box, there are three different content shortcuts. A plus icon allows us to create static meshes for the level. Hierarchical pictures let us design blueprints and access them. Slate pictures allow us to create a level sequence or master sequence for cinematics. Next, indicated as number four, are all the play mode controls for starting, pausing, stopping, and ejecting the game that runs in the editor (FIGURE 3). The game project can be configured, deployed, and prepared for different platforms in section number five. Lastly, number six is where you can access editor settings, view-port level settings as well as game behavior settings. (Epic Games. 2004-2023l.).

FIGURE 4. Unreal Editor Interface. Main toolbar (Epic Games. 2004-2023ad.)

Using the viewport, actors can be selected as we mentioned previously. The actor's details panel will be displayed. In this panel, the user can modify many different types of global as well as actor settings and properties that can be configured for the actor. (FIGURE 5). (Epic Games. 2004-2023ad.).

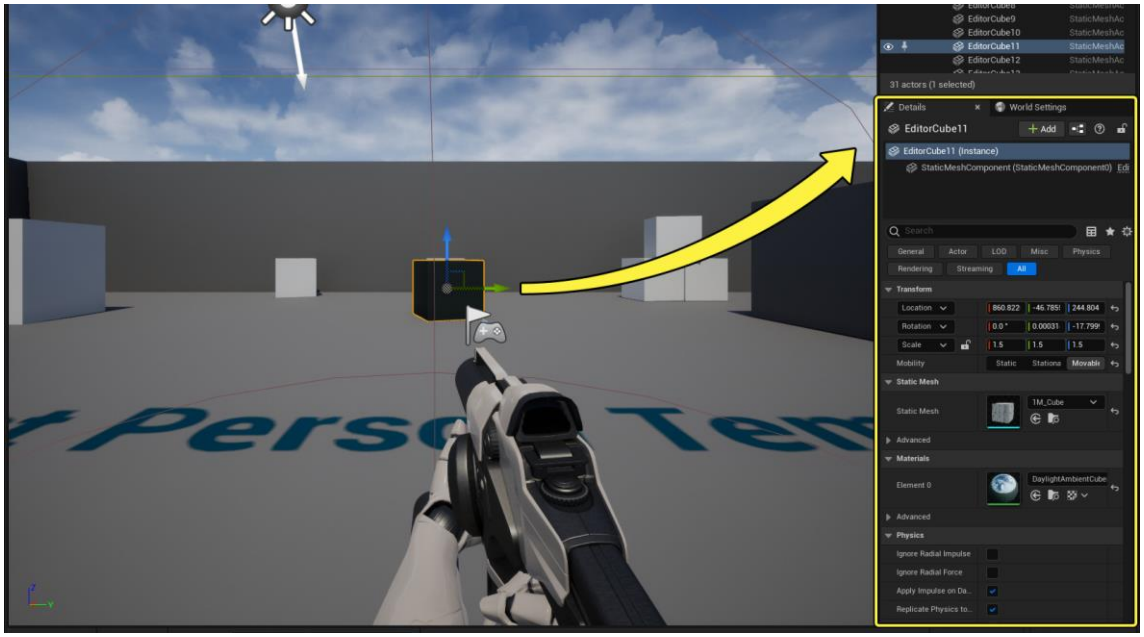


FIGURE 5. Unreal Editor Interface. Details panel (Epic Games. 2004-2022ad.)

From gameplay functionality to the assets and materials in the game, the tools, editors, and systems that are provided by Unreal Engine are utilized to develop the game. Not all the tools and editors come as built-in options in a game engine, but they need to be added to the project as a plugin. Nevertheless, the following editors are the most frequently used and have already built into the engine. As in gameplay levels, the level editor is the primary window where a level is designed, and the world is visually constructed. Different types of geometry, actors, blueprints, and objects are added to do so. Static mesh editor can be used to manipulate properties of the static meshes as well as set up a level of details for that specific static mesh. This will affect how meshes are displayed when the game is running. In addition, a static mesh editor can be used to preview static mesh, the collision of the static mesh as well as how that mesh is UV mapped. The Material editor is the Unreal Engines editor interface where materials are created and edited. Materials are applied with static meshes to change how they look.

Blueprint editors are accustomed to working with unreal engines and visual scripting systems to design gameplay elements without needing to write C++ code. However, preferred gameplay elements can be done purely with C++ programming and minimize the need for blueprints. The physics editor is used to create assets that use skeletal meshes. When game assets need physical elements like collisions or deforming properties to be present. A behaviour tree editor is a visual node system that is utilized to script artificial intelligence systems' behavioural patterns based on actors. Niagara editor is used to create emitters, which are a special effect of multiple particle systems. UMG UI editor is designed to create interface graphics. UI editor also used visual tools when building different elements. Sequence editor enables the creation of in-game cinematics such as cut scenes. It has a multi-track editor so that the scene can be made of different tracks that have different elements such as animations, transforms that move around the scene, and audio effects. Animation editor can be used to edit several kinds of animation assets such as skeletal assets, meshes as well as animation blueprints. With the Control Rig Editor, you can rig characters and animate them directly in the game engine. With the control rig editor, there is no need to use external tools to animate and rig game assets. Sound cue editor combines a mix of several different sounds and makes a single output that is saved as a sound cue. (Epic Games. 2004-2022z.).

Creating and importing assets such as static meshes or textures to your Unreal Engine 5 projects can be done in a content drawer/content browser. This is where the folder structure of the project can be found. When the content browser is open there is an add button in the left corner of the window where new assets can be added (FIGURE 6). Besides importing assets outside the project this also enables the creation of other basic assets such as blueprints, levels, and materials. There is also advanced asset creation where a developer can create several customized elements for the game. These elements include animations, artificial intelligence, advanced blueprints, cinematics, editor utilities, foliage, FX, Gameplay, input, materials, media, miscellaneous, paper 2D, physics, sounds, textures, and user interface elements. At the top of this popup window, there is an option to import content from outside of the engine. Additional assets can also be dragged and dropped directly into the content folder for import. Regardless of the chosen import method, the import options menu will appear once the import starts. The import options window can look different depending on what type of assets are being imported. A different setting for import can be adjusted in this import options window. (Epic Games. 2004-2022k.).

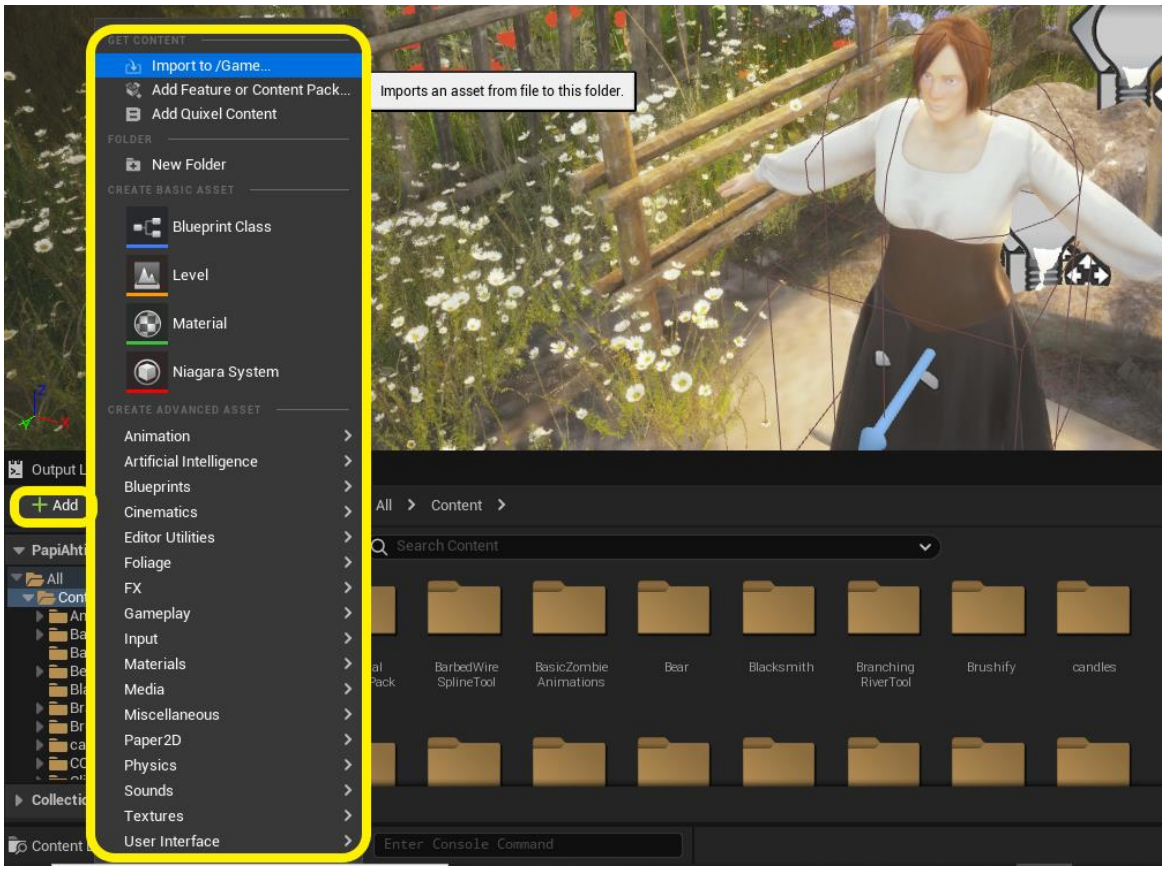


FIGURE 6. Tuonela project screenshot of how to import assets to game.

2.2 License agreements

When creating products with Unreal Engines, creators must follow license agreements. Unreal Engine end-user licenses are for creators who use the engine internally or create projects for customers. An EULA license applies to Epic Games technologies downloaded to a computer when Unreal Engine is installed. Starter content templates, game engine code, and examples are included. However, content that is downloaded from the Unreal Engine Marketplace is not under a EULA license. When the revenue received from the project does not exceed the royalty threshold under the EULA license agreement. The initial gross revenue of $1 million is royalty-free. After that, Epic Games requires 5% of the product royalty. Products that are completely exempt from royalty payments are products that do not require engine code to run and do not use started content. (Epic Games. 2004-2022aj.).

A content license agreement is applied to content made available by Epic Games under the licensed content agreement. This agreement includes content that is downloaded from the Unreal Engine marketplace or Quixel Megascans library or the learning tab of the epic game launcher. A

license agreement allows developers, publishers, or distributors to distribute video games when they are part of the Unreal Engine project. Under this license agreement, users are restricted from reverse engineering, translating, or decompiling, and any type of distribution on a standalone basis is forbidden. Projects must add value beyond licensed content if the project is distributed. Lastly, any violation of applicable law will not be permitted. Different plans are available for different types of distributors. A personal plan does not require users to pay for a plan if the commercial product generates less than $100,000 gross revenue annually for two consecutive years. The indie plan allows an entity to use assets under a license agreement in product releases if annual gross revenue falls under $2,000,000. This is not funded by a publisher or controlled by a game studio. Any funds raised by fundraisers are also included (Epic Games. 2004-2022f.).

To release a version that can be distributed, the game needs to be packaged. The game package can be created in the project launcher editor of Unreal Engine which can be found under tools. A new custom launch can be created once the project launcher has been selected. Different settings are available for the release project, and how these settings should be adjusted is heavily dependent on the type of project. The project build needs to be set for shipping as well. The project content cooking settings will determine which platform the project is being deployed on, what maps should be deployed, and which version number is being released. Once these steps and settings have been configured, the project can be deployed (Epic Games. 2004-2022t.).

2.3 Unreal Engine compared with other game engines

As Unreal Engine is not the only game engine on the market that provides developers with an easy way to create games. Where do free-to-use game engines differ from each other? Because of this, Tuonela chose Unreal Engine over all the other options available on the web. While one of the most obvious answers would be that Unreal Engine is one of the most popular engines, there were also a few technical reasons why the demo was made with Unreal Engine. Hence, if developing with Unreal Engine is an option, we do not delve too deeply into how different the minimum hardware requirements are. These other game engines should also run just fine on the hardware that they are intended to be used on. For the purposes of this thesis, only game engines that can generate exclusively or additionally 3D games, rather than just 2D games, will be compared with Unreal Engine. This is because the demo is done with 3D graphics.

When comparing free-to-use game engines **Unity** is most often compared to the Unreal Engine. Both engine types support cross-platform development. Unity uses C# for development while Unreal Engine uses C++. Unity is often praised for being much easier to learn and use with its intuitive user interface. In comparison, C# is considered a bit easier language. Unity has refined the 2D system that Unreal Engine lacks. Therefore, if the developer made a 2D game Unity would be much better suited for that type of development. However, Unity can also be used to develop 3D games but its graphical qualities on that front are significantly less impressive than Unreal Engine. Still, Unity has an extensive toolbox for development that also Unreal Engine offers so if the developer does not require AAA-quality games Unity is a much easier option for someone who is just starting out in game development. Unity similarly has a royalty system in place if a developer generates income from the game that is developed. Additionally, Unity has different engine products depending on whether the developer is a big studio or a small indie developer (Unity Technologies 2022.).

Godot is another cross-platform game engine that offers both 2D and 3D development capabilities. Unreal Engine is not the most suitable game engine for making 2D games. Therefore, Godot could be a viable alternative to Unity if you mainly want to make 2D games. Godot also has a simple interface that is friendly to inexperienced users. However, like Unreal Engine development is done with C++, GDScript, and visual scripting tools. A most notable difference is that Godot comes with its own IDE. Graphical possibilities are abundant with Godot's BSDF system, GLSL shader language, post-processing effects, and support for MSAA and FXAA. Unreal Engine's graphical capabilities are dominant when it comes to 3D games and AAA-quality graphics. Godot is free to use, so even if a developer makes money from games developed with it, there will not be any royalty payments. (Linietsky, Manzuri, and contributors. 2007-2022.).

As with other engines, **CryEngine** provides cross-platform solutions for developers. It offers a Sandbox solution where developers can create a large seamless world without the need for baking when exporting the project. As part of their rendering process, they offer dynamic global illumination with a PBR workflow that provides excellent graphical quality. This is one of the few engines that can compete with Unreal Engine for environment creation. Similarly, Unreal Engine CryEngine is much more focused on making 3D games even though there are possibilities to create 2D games. However, the engine is not the most intuitive for this type of development. CryEngine uses C++, Lua, and C# when developing. Similarly, to Unreal Engine, the learning curve compared to other game engines is very steep and for less experienced developers this might not be the most suitable

option. Furthermore, learning multiple languages simultaneously can be overwhelming and intimidating. (Crytek GmbH 2022.).

3 UNREAL ENGINE 5

Unreal Engine 5 is an open-source game engine like its predecessor Unreal Engine 4. The Unreal Engine developer community can make changes to the source code through GitHub. Due to Unreal Engine 5's newness, there are still some features that are in the experimental or beta stages. It is not recommended to use these features in production. As of now, Unreal Engine 5 offers several fully supported features, and it promises to add more in the future. The software provides freedom, fidelity, and flexibility when creating 3D content and experiences. A variety of options are available with Unreal Engine 5 as a platform for your game.

Unreal Engine 5 brings cutting-edge, ground-breaking visuals to the user so that the game world can be more dynamic and expansive than ever before. This can be achieved with new features developed for Unreal Engine 5 like Nanite and Lumen. Epic Games has also improved its animation and modeling features to be more usable for the artists as well as faster to use than its predecessors. The last most notable change that can be seen in Unreal Engine 5 is a new kind of editor user interface. It has been made to be more flexible and faster to work with. Features that will be excluded are those that are still in the beta or experimental stages. This is because they are not feasible to be used in the game demo since Epic Games does not recommend using beta or experimental features in production (Epic Games 2004-2022g).

3.1 Lumen and Nanite

Lumen Global Illumination and Reflections also referred to as Lumen, offers artists the possibility to create illumination and reflections that are more realistic. Angle changes or any changes in the environment can change lighting and reflection direction and Lumen should mimic this to a realistic extent (FIGURE 7). The rendering system uses indirect specular reflections to bounce off from the environment to capture even the smallest details. In Unreal Engine 4 lighting was baked into textures with precomputed lights. This often could take some time depending on the number of textures and objects placed in the scene. However, with Lumen baking lighting is not needed and lighting changes can be seen in real-time. Lumen supports Ray Tracing (2004-2022n.).

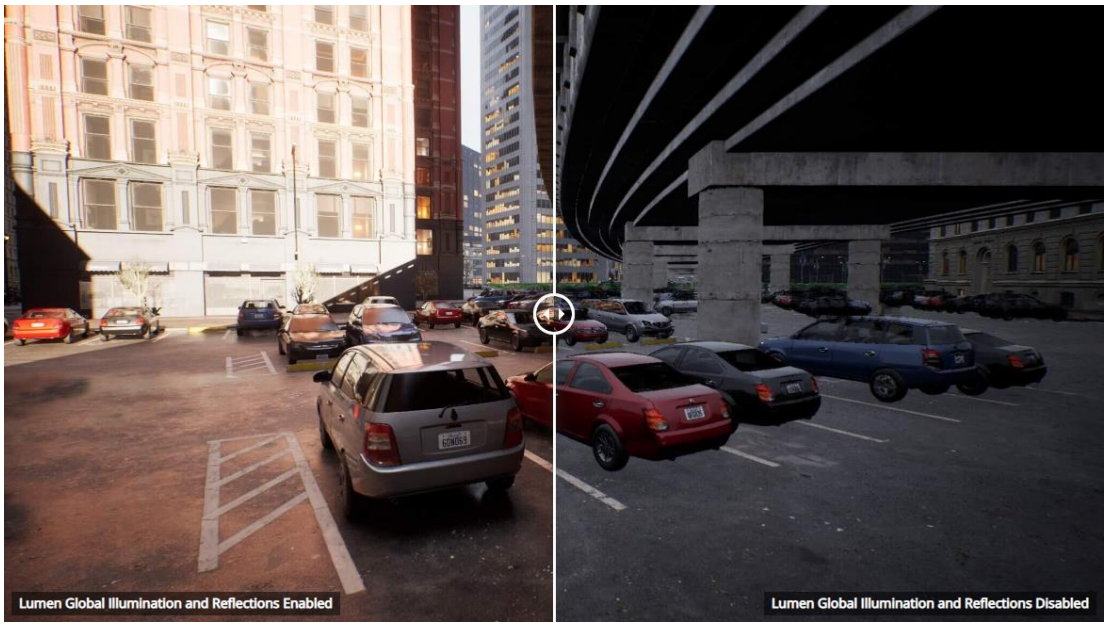


FIGURE 7. Lumen Global Illumination and Reflections comparison. (2004-2022n.).

Nanite Virtualized Geometry allows developers to create games that have a lot of geometric details. It allows importing art that is made of millions of polygons from sculpts to photogrammetry scans. As a developer who also makes some of the game assets, there is always a concern about the polygon amount on the game assets that are imported into the engine. Too much detail can often be too much for the game engine and it will result in low latency during gameplay. Nanite will stream only the details that can be seen currently. The workflow will be improved as textures will no longer need to be baked to normal maps or levels of detail will be manually adjusted.
Temporal super-resolution together with Nanite can produce next-generation games with a high level of quality. It will up a sample that is built into the Unreal Engine (Epic Games 2004-2022o.).

In the Unreal Engine 5 editor, developers can create control rigs for their characters. Animation mode itself has also seen improvement so that the interface is more animation friendly. This has been achieved by adding filtering tools for animation control and properties for management purposes. There are also tools in place to aid with the animation workflow such as the posing tool from which poses can be saved and reused. Motion trail tool for visualizing keyframe movements. Epic games have added space switching that allows re-parenting controls without the need for complex logic for it. It can be used inside the sequencer or found rebuilt in the control rig asset. Creating control rig assets has become production ready in Unreal Engine 5 and it has lots of improvements in graphics such as creating functions and arrays that will make it easier to work and manage with the control rigs. The new control rig also offers python scripting for those who wish to automate

their workflow with control rigging inside Unreal Engine 5. Another notable animation related change is IK Rig and IK retargeted. A more efficient method of creating IK solvers and defining target goals. These target goals can be controlled during runtime. Therefore, the retarget manager that served a similar purpose in Unreal Engine 4 has been deprecated (Epic Games 2004-2022ag.).

3.2 Local Exposure and Post-motion Blur

Local exposure processing automatically adjusts exposure on the map which can be controlled via parameters. Local exposure processing is applied to preserve details of highlights and shadows in addition to the existing global exposure systems that Unreal Engine 5 has. This feature is useful when projects have dynamic lighting in a wide range. Using local exposure processing in a scene will help to avoid overexposure and completely dark shadows. The documentation for Unreal Engine 5 gives an example of games with dynamic time-of-day systems. Using this type of system, it is easy to find scenes that are both under-exposed and over-exposed. Such as indoor scenes with bright outside light coming from a door or a window which will lead to problems with gameplay (Epic Games 2004-2022ag.).

Post-motion blur translucency was developed to help improve the quality of in-world UI elements that suffer in terms of quality when they are rendered with translucent passes like motion blur and temporal anti-aliasing. Post-motion blur translucency provides a way to avoid using these two methods when creating transparent materials, which improves how rendered materials appear (FIGURE 8). (Epic Games 2004-2022ag.).

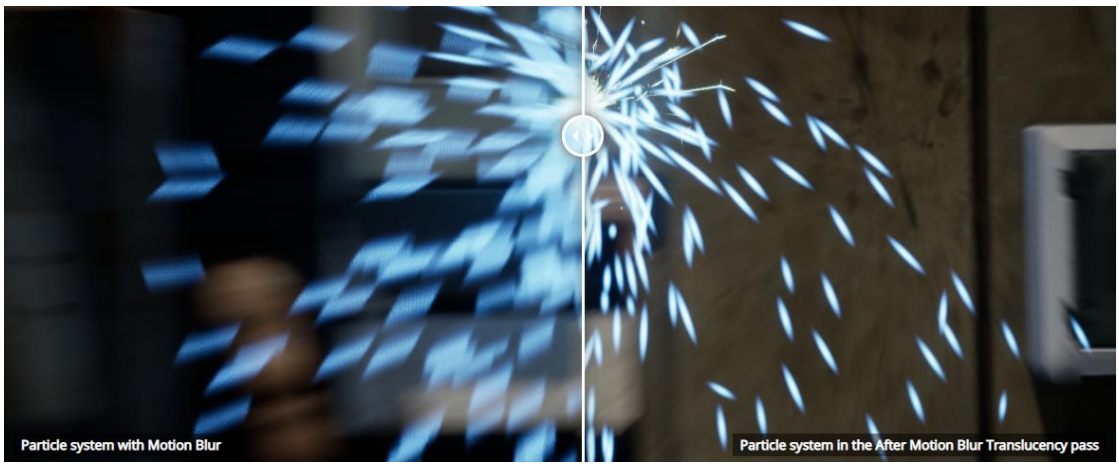


FIGURE 8. Unreal Engine 5 Post-Motion Blur Translucency (Epic Games 2004-2022g.).

3.3 World partition

World partitioning is a management and streaming system that can be used in the editor as well as during runtime. It enables developers to create a world on a single persistent level without the need for dividing it into countless sublevels for streaming and for data connection reduction. A 2D grid will allow developers to divide the world into different areas. This 2D editor handles which areas, along with their data, are loaded and when. This way it is possible to make large worlds without fear of slow loading or memory handling issues. This feature has already seen a lot of improvement from the early access state by providing default map types for the open world. In addition, it added custom shapes for streaming sources, support for landscape brushes and automated batch processes, and generating and modifying data at world partition levels. For the newly introduced world partition feature, Unreal has also developed data layers that are designed to load world data during runtime and editor activation and deactivation. Data layers can organize the editor or create different types of scenarios that one level might have such as day and night. One file per actor feature was added to the world partition system so that collaboration is easier in levels. The level editor saves individual actors in their own files rather than in the group. Developers will only need to check out actors that they need from source control, not the entire level. Data management and streaming systems are also new features for the world partition. This feature removes the need to manually divide levels into sublevels to manage to stream and reduce data connections. With the world, partition game exists in a single persistent level. The level is split by using a 2D grid and these grid areas are managed by the world partition editor window. (Epic Games 2004-2022ag.)

Another notable improvement from the previous version of Unreal Engine 5 is the ability to collapse nodes in the material editor. This allows the user to group selected nodes into functions thus making them easier to interact with. Collapsed nodes can also be reverted if needed. Another improvement is an automated grid-based hierarchical level of detail system that optimizes static meshes that are displayed outside the loaded area. Meshes are generated by mimicking the original actor's geometry and then simplified. This way performance can be boosted, and memory usage reduced (Epic Games 2004-2022ag.).

In comparison to its predecessor Unreal Engine 4, Unreal Engine 5 offers several upgrades and added features. Despite a lot of changes to the engine, Epic Games has made migration a painless process for developers with the built-in conversion process. Migrating the Unreal Engine 4 project

to the Unreal Engine 5 project starts with downloading the Unreal Engine 5 engine. Once downloaded, the user can use the Epic Game launcher with Unreal Engine 5 selected. Project files can be opened with the Unreal Engine 5 launcher even though they have been created with the Unreal Engine 4 game engine. Migration can be performed as a copy of the original project or convert-in-place which will convert an existing project without creating a new one. There is also an option to try to open the project as it is and skip the conversion process (Epic Games 2004-2022af.).

4 TUONELA

Tuonela is being developed exclusively for Windows PC. The reason why the demo was developed with Unreal Engine 5 is that Unreal Editor offered needed graphical capabilities as well as advanced features in terms of gameplay capabilities and world-building properties. As part of the environment for the project, Epic Games' free-to-use assets for Unreal Engine 5 assisted with the creation of foliage and environmental materials as well as some environmental assets. Due to the project being worked on by two developers, both found Plastic SCM to be the best-suited option among Perforce Helix Core and GitHub for source control.

Tuonela gameplay design follows very traditional role-playing game gameplay where the player takes on the role of the protagonist of the game and follows a story as the character travels through the game's quest line. An imperative part of the gameplay is to level up the character and gain points that will unlock new skills and abilities for the character. Leveling up will also make the character stronger. This is because there will not be different armor or weapons to switch between stronger or weaker types of equipment. It is possible to improve the quality of existing equipment. Tuonela is not meant to be too challenging but balances compelling storytelling with RPG elements. In addition, we want to implement some puzzle elements into the gameplay to phase the game forward for the player.

4.1 Level Design

During the process of designing the level and narrative for the Tuonela demo, it was vital to keep the design scale as small as possible. Considering two people were going to be working on the demo, it would be a lot more achievable. As well as getting a demo done, it was also extremely crucial not to get stuck in the development of a much larger game than it needed to be. The game design document was created for the purposes of documenting different design choices as well as other game elements such as how much different items would give points to the player, how much experience would be gained defeating different enemies, what kind of abilities different kinds of enemies would possess, what kind of currency the game uses, how many quests there are in the world and how quests progress, what kind of abilities the player has, how new abilities are gained and much more. An essential part of the environment and level design is to have a functioning

ecosystem. In the future, some of the narratives and technical designs will depend on foraging and hunting animals. As the Game design document name suggests this document consists of all the design solutions for the game.

The demo takes place in the settings pictured below (FIGURE 9). The small village represented by light blue area number two is an area where most of the non-playable characters can be found residing. This village is surrounded by a field marked by yellow number one, an old graveyard that has a museum marked with green number three. Finally, a bond that was sullied a long time ago is represented by the pink square and the number four; additionally, you can find catacombs under the village. These are accessible from the museum but also from the small entrance to the valley indicated by the red square and number five. A lot of gameplay and combat will take place in areas like fields and the valley as well as the catacombs. The end demo area is the sullied pond in number four. The player is led there at the end of the main quest at the end of the demo.

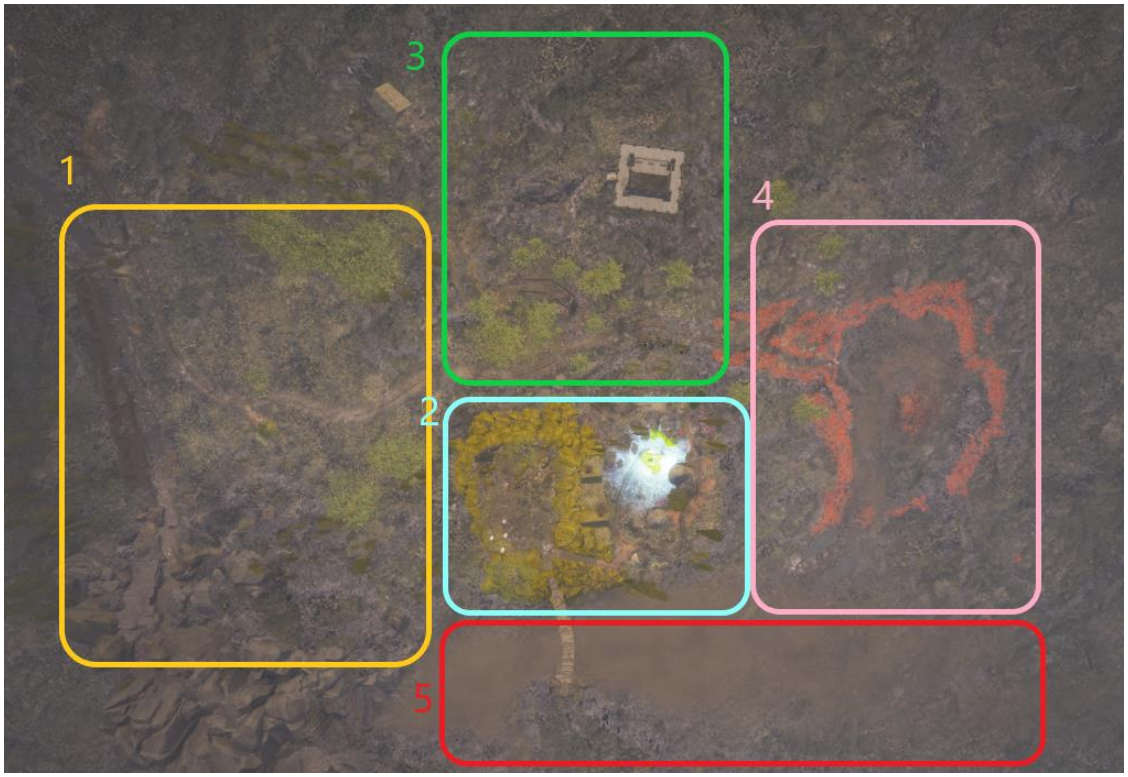


FIGURE 9. The top-down perspective of different areas where the demo takes place.

4.2 Narrative Design

Tuonela has three scenarios designed for gameplay purposes: one main quest and two side quests. During the main quest, the player follows the story of the protagonist, who flees Tuonela and wants to return to Pohjola to propose to Louhi's daughter. Through the quest, the player gains power and prestige. The protagonist is already on thin ice if the person who is alive does not leave Tuonela. However, he escapes and now wants to find his way back to Tuonela. This will take the player to the end of the demo while leaving open possibilities of what a full game around this same setting and lore could be like. Two side quests are also extremely significant for the demo. While side quests serve the phasing purpose, they also add to the player's knowledge of the mythology of Kalevala. They are not just fillers for the game but rather an interesting part of Finnish mythology that can be explored. The second side quest will tease the bestiary where the player gets hold of a book of different kinds of mythological beasts that are traditionally known in Finnish mythology. In the demo, the player will only be able to find and defeat one of the beasts. However, the bestiary book will include pictures and descriptions of other mythological creatures that could be included in a more extensive game. Bestiary is obtained through a quest that takes you underground, under the village (FIGURE 10).

FIGURE 10. Underground area found under the village.

Lastly, the second side quest is related to the story of Aino which is in Kalevala. We wanted to include more puzzle elements in this side quest. The player will begin the task of solving the mysterious drowning of a girl named Aino. The story does not have the happiest ending. However, the

player will proceed through some detective work to conclude that Aino has not been murdered but she has taken her own life. In the original story of Kalevala, Aino will turn into a fish and in our game demo, we would like to hint at this as well. We would like to emphasize to players that this is a fantasy world and a game scenario based on mythology. Most NPCs are in the village, which is a small part of our demo. However, it is a key element of the game, located in the middle of the demo map (FIGURE 11).

FIGURE 11. Entrance to the village is located middle of the game demo map.

Cinematic storytelling with Tuonela has been planned to be implemented in two ways. In the future we would like this to be an open-world game and players would be free to explore and interact with the world as they wish and make it as immersive as possible. As we did not have the possibility of creating expensive fancy mock-ups with real-life actors in the demo, we will handcraft all the main quest scene cinematics as well as the side quest cinematics for the demo. However, if the game gets any larger, this would be way too much work. Therefore, we also made initial plans for a procedural camera system. This would have logic behind it that would generate cutscenes based on the player character and nonplayable characters so that it would choose animation randomly from the array. There would also need to be some elements of camera lens angle variables implemented differently depending on what kind of environment a nonplayable character would be surrounded by. This would be based on whether there are any possible obstructions between the camera lens and the scene. The procedural camera will probably not be implemented in the demo just yet but the plan for it will be included in the game design document for the future and only hand-crafted cutscenes will be used at this moment to save time.

5 TUONELA TECHNICAL DESIGN

We compiled a list of the technical design choices we made for the Tuonela project using C++ in the sections below. In some instances, we also used Blueprint, an Unreal Engines visual scripting language to convey information to UI elements. These design choices were just a few of many and are just a few among some of examples. The file structure of an Unreal Engine project can vary depending on the project, but there are some common conventions that are followed. The "Content" folder contains all the assets that are used in the game, including 3D models, textures, audio files, and other media. As part of the Tuonela project, we made a lot of subfolders to make it easier to organize. For example, we have an NPC folder where you can find more subfolders for each character such as Hildrus (one of the non-playable character names), and then in the NPC folder, there are even more subfolders where you can find textures, character rigs, animations, and skeletal mesh of that character. This is one of the examples and there are often a lot of subfolders to organize different types of files and media. The "Source" folder contains all the C++ code that makes up the game's logic and functionality. Within the "Source" folder, there are "Private" and "Public" subfolders, which separate the code that is used internally by the game engine from code that is exposed to designers and other external developers. Both folders then have the same subfolder that separates files depending on the purpose for which they are used in the game. For example, there is a subfolder for the default time class as well as the child classes of that item. The "Config" folder contains various configuration files for the game, such as settings for the engine, input bindings, and other options. The "Intermediate" and "Saved" folders contain intermediate and saved files during compiling, running, and packaging. It is imperative to note that this is not the only way to arrange files and folders, and different projects and teams may have different conventions.

Unreal Engine code is typically structured into modules, classes, and blueprints. C++ is used for creating the core engine functionality while blueprints are used for creating gameplay and game logic. Classes can interact with blueprints and engine code, allowing for a flexible and efficient development process. In Unreal Engine, parent-child classes in C++ classes are used to define the functionality of different Actors or Components in the game. A parent class is a class that serves as a template for one or more child classes. There can be as many child classes as needed to be derived from one parent class (FIGURE 12). A child class can inherit the functionality of the parent class and add, or override functionality as needed. One common use of parent-child classes in Unreal Engine is for code reuse and organization. For example, you might have a parent class that

defines basic functionalities for all characters in the game such as movement and collision detection. In the demo, we also use the parent-child relationship to our advantage by creating child classes for different types of characters such as player characters and enemies as well as items and weapons.

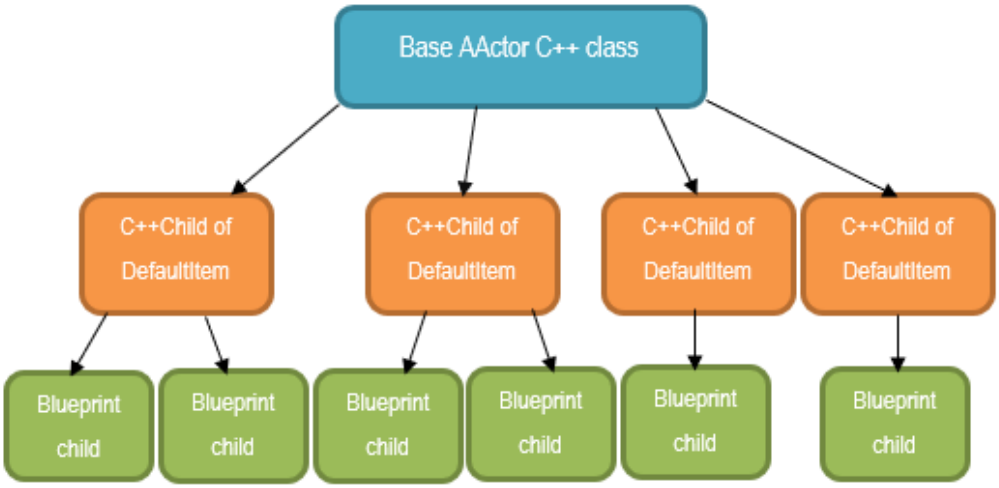


FIGURE 12. How Tuonela code is structured for code reuse and organization AActor class example.

A lot of Tuonelas' technical design elements are drawn from traditional role-playing game systems; this will be a more surface-level look at how we bridge narrative and environmental design with technical design at a high level. The following is a brief overview of some of the most influential game systems in the demo and just a mere glimpse of how we execute technical design choices. The game contains several mix-and-match elements that we intend to make work with traditional roleplaying game style by combining puzzle elements with some platforming, which is something that many big game studios do with their games today. There are a few mechanics that can be found in Unreal Engine 5 templates that are not mentioned, such as the character movement system, nor is every nuance of the game being displayed; just some key programming concepts for the game to show some technical design choices for the demo. It is also assumed that there is a basic understanding of C++ fundamental concepts.

5.1 Player Character

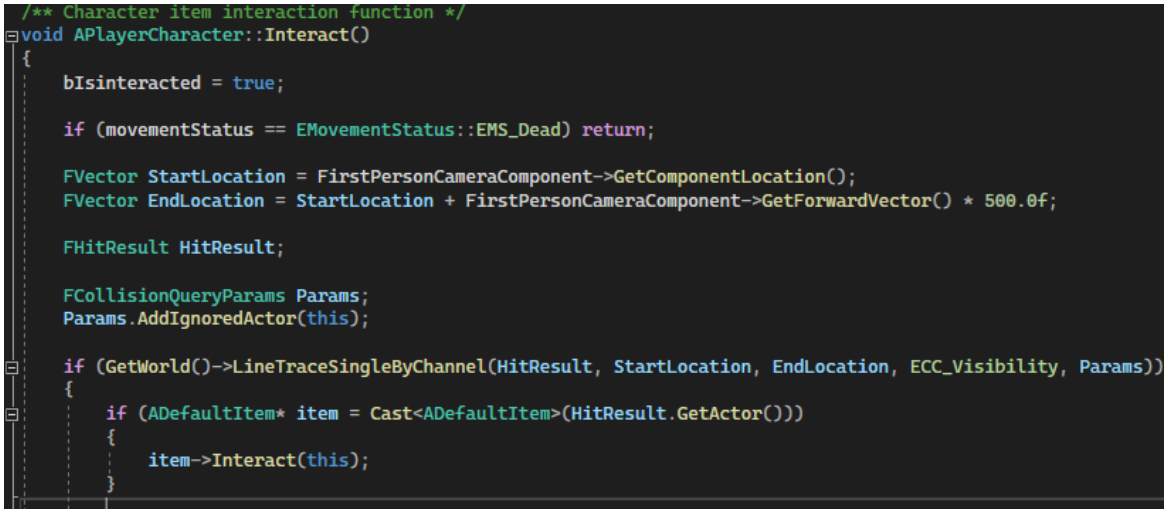
One of the most significant aspects of a player's character is their stats. This is because many of these stats will also carry over to other classes such as skills, weapons, and items. Our current system consists of the Attribute Component which is a component that allows actors to have attributes such as health, mana, or stamina (FIGURE 13). It provides a way for actors to have a current value, a base value, and a max value for an attribute. In addition, you can also modify the attribute's value over time. In Tuonela, attribute components are not only used to determine players' health and stamina but they are also used by enemies to determine their health and other attributes. In addition, they are used to handle other attribute functionalities like calculating the current health of the actor for example. When you organize your code, it makes your game more efficient, flexible, and maintainable.

```
private:
    UPROPERTY(EditDefaultsOnly, Category = "Stats")
    float Health = 80.0f;

    UPROPERTY(EditDefaultsOnly, Category = "Stats")
    float MaxHealth = 100.0f;

    UPROPERTY(EditAnywhere, Category = "Stats")
    int32 CurrentLevel = 1;

    UPROPERTY(EditAnywhere, Category = "Stats")
    int32 SkillPoints = 0;

    UPROPERTY(EditAnywhere, Category = "Stats")
    float StaminaAmount = 50.0f;

    UPROPERTY(EditDefaultsOnly, Category = "Stats")
    float MaxStamina = 100.0f;
```

FIGURE 13. Stats in UAttributeComponent.h file that can be attached to actors.

A major aspect of gameplay mechanics in the demo is the interaction system, which is a major part of the gameplay system. Since puzzle elements were incorporated into the narrative of the game, efficient interaction with the world around the player is extremely relevant. In addition, many of the interactions with non-playable characters are instrumental in moving the story forward. To begin with, we implemented a collision volume system for interacting with the game. To interact with an object, a player must be within the collision range of the object to do so. However, from the first person's perspective, it was not a very feasible solution. This was because it was often challenging to place yourself within the collision sphere range from a first-person perspective. Instead, we used ray tracing from the players' camera component to draw a line from where the player looked to the

object to be interacted with (FIGURE 14). This was determined by the FVector variable called StartLocation. In addition, FVector as EndLocation was determined so that there would be limited reach. There is also a struct called FHitResult which contains information on the hit point such as the impact point (Unreal Engine 2004-2022h.). Finally, the FCollisionQueryParams struct called Params defines the parameters that are passed to the collision function to ensure that the player character is ignored when a collision occurs with objects (Unreal Engine 2004-2022g.). This was created for safety purposes and probably is not necessary for a first-person view. All the above-mentioned structures can be found by default in the actor class and do not need to be defined in the header file beforehand.



```
/** Character item interaction function */
void APlayerCharacter::Interact()
{
    bIsinteracted = true;

    if (movementStatus == EMovementStatus::EMS_Dead) return;

    FVector StartLocation = FirstPersonCameraComponent->GetComponentLocation();
    FVector EndLocation = StartLocation + FirstPersonCameraComponent->GetForwardVector() * 500.0f;

    FHitResult HitResult;

    FCollisionQueryParams Params;
    Params.AddIgnoredActor(this);

    if (GetWorld()->LineTraceSingleByChannel(HitResult, StartLocation, EndLocation, ECC_Visibility, Params))
    {
        if (ADefaultItem* item = Cast<ADefaultItem>(HitResult.GetActor()))
        {
            item->Interact(this);
        }
```

FIGURE 14. Interact function in PlayerCharacter CPP file.

After creating the required variables inside the if statement, the final step was to implement interaction functionality using the ray trace function that draws a ray trace. The function takes in previously created variables as well as the collision channel that these hits will be registered on. In this case, the selected channel is ECC_Visibility (Unreal Engine 2004-2022ak.). A cast can be done in separate if statements for the different actors to be included in the HitResult struct to be interactable objects. The game uses the default item as a base for all interactable items in the game. Additionally, the default item contains a virtual void function called Interact (FIGURE 15), which can and will be overridden in child actor classes. Items that are used in the game are all child classes of the DefaultItem class. This same logic is used for non-playable character interactions as well.

```
UFUNCTION()
virtual void Interact(APlayerCharacter* PlayerCharacter);
```

FIGURE 15. Interact function inside DefaultItem.h Actor Class.

5.2 Items

DefaultItem is the parent AActor class for all game items including consumables, weapons, quest items, and puzzle items. Therefore, DefaultItem has many virtual void functions meaning that they can be used in child classes inheritably. The interact function was already mentioned as one of them. In addition to the interaction function, there are other functions such as the use function. This is what happens when a player uses an item such as gaining health or stamina. By default, mesh components are found in child classes. Another significant part of the DefaultItem was the *Item-Structure* which contained all the item data (FIGURE 16). The item data holds information that can be modified and personalized within child classes. This information can be used in a variety of gameplay scenarios such as inventory and shop logic.

```
USTRUCT(BlueprintType)
struct FItemData
{
    GENERATED_BODY()

    UPROPERTY(EditDefaultsOnly, BlueprintReadWrite)
    TSubclassOf<class ADefaultItem> ItemClass;

    UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category = "Item")
    class UTexture2D* ItemImage;

    UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category = "Item")
    int32 ItemCost;

    UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category = "Item")
    FString ItemName;

    UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category = "Item", meta = (MultiLine = true))
    FString ItemDescription;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Item")
    int32 MaximumStackAmount;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Item")
    int32 CurrentStackAmount = 1;

    UPROPERTY(EditdefaultsOnly, BlueprintReadOnly, Category = "Item")
    int32 ItemID;
};
```

FIGURE 16. Screenshot of the ItemStrcut.h structure and what data is stored in items.

Based on the child class of default items, item blueprints expose all the item data, which can be modified to fit each item, such as name and cost. In addition, it contains some information for other

purposes such as inventory logic such as maximum stack amount and current stack amount (FIGURE 17).

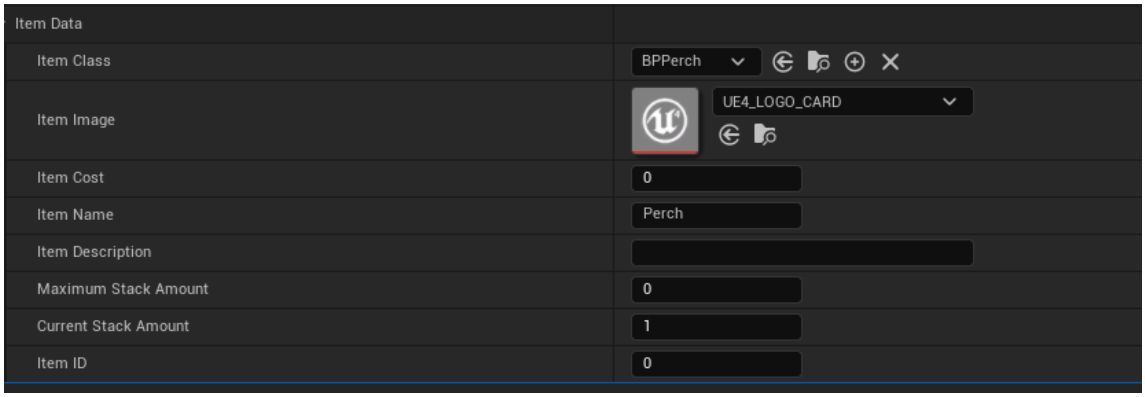


FIGURE 17. Example of child class blueprint BPPerch and ItemData that can be modified.

As a result, the primary weapon is a child of the default item. However, it will not behave similarly to items that can be used in the inventory. However, there are cases that weapons need some of the functionalities that the default item has such as the interaction function as well as some components such as mesh. There are no different varieties of weapons that can be switched between. The player can only switch between a primary weapon or a hatchet type of weapon. The primary weapon, however, has different stats that can be upgraded along the way. During programming, the fact that the primary weapon would be a static element was taken into consideration. The primary weapon can be equipped or unequipped. However, it will not appear in the inventory as a usable object. Instead, it can only be seen in the menu where it can be upgraded in exchange for gold as well as resources for the upgrade. It is critical for the primary weapon to contain upgradeable components and the ability to inflict damage upon contact with the enemy. In Tuonela case we created UBoxComponent for a simple collision called ComabtCollision (Unreal Engine 2004-2022aa.) to determine which part of the weapon would inflict damage to the enemy (FIGURE 18). The hatchet weapon is then derived from the primary weapon as a child class. This is because it still requires the usage of the same logic for example how to deal damage to enemies. However, in the child blueprint of the hatchet class, we will create some logic for the throw mechanics of the hatchet. As mentioned before classes can have as many children as needed.

```
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Combat")
UBoxComponent* CombatCollision;
```

FIGURE 18. CombatCollision variable in DefaultWeapon.h.

The blueprint for class-default weapon combat collision can be found in the blueprint and it can be adjusted according to the need as to where the damage should be caused relative to the weapon mesh (FIGURE 19). As can be seen in the picture, the collision has been moved to the blade of the axe. This is because it would be logical to have a damage impact point when attacking.

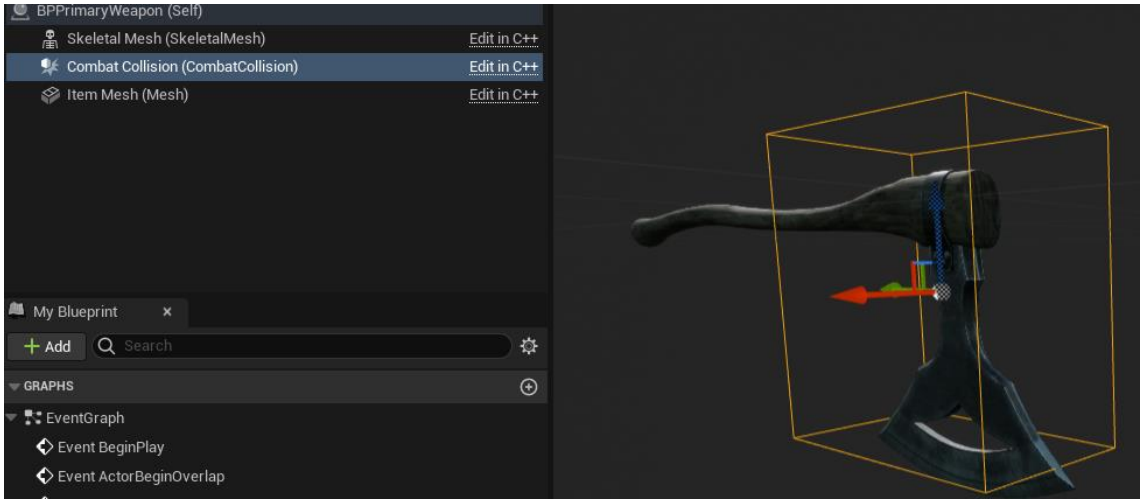


FIGURE 19. CombatCollision in Blueprint made from DefaultWeapon class.

In Unreal Engine 5, collision channels are used to define how different objects in a scene interact with each other when they come into contact. They allow for fine-grained control over which objects can affect each other and how. On the BeginPlay of the *DefaultWeapon* class, it is declared that the *CombatCollision* will use the built-in Unreal Engines *OnComponentBeginOverlap* and *OnComponentEndOverlap* function macro called *AddDynamic*, which will bind the *CombatOnOverlapBegin* and *CombatOnOverlapEnd* events to the *CombatCollision* so that they will be called whenever the enemy encounters the *CombatCollision*. Since *CombatCollision* is a primitive component, by default it contains a lot of different types of collision functions that can be seen being set with the ECollisionChannel, ECollisionEnabled, and ECollisionResponse built-in enums (FIGURE 20). These enums respond to a different setting that will help the game to understand how the collision volume should behave. For example, what should happen when the collision volume is activated? With what collision volume should and should not be colliding? In the Tuonela case, we use collision channels to determine a place where to draw a ray trace when colliding with the enemy (Unreal Engine 2004-2022ah.).

```cpp
void ADefaultWeapon::BeginPlay()
{
    Super::BeginPlay();

    CombatCollision->OnComponentBeginOverlap.AddDynamic(this, &ADefaultWeapon::CombatOnOverlapBegin);
    CombatCollision->OnComponentEndOverlap.AddDynamic(this, &ADefaultWeapon::CombatOnOverlapEnd);

    CombatCollision->SetCollisionEnabled(ECollisionEnabled::NoCollision);
    CombatCollision->SetCollisionObjectType(ECollisionChannel::ECC_WorldDynamic);
    CombatCollision->SetCollisionResponseToAllChannels(ECollisionResponse::ECR_Ignore);
    CombatCollision->SetCollisionResponseToChannel(ECollisionChannel::ECC_Pawn, ECollisionResponse::ECR_Overlap);
}
```

FIGURE 20. DefaultWeapon.CPP BeginPlay function

In Unreal Engine, the OnComponentBeginOverlap event is a built-in event that is called whenever two components collide with each other for the first time. In this event, the default delegates are another actor the component overlaps with, another component the component overlaps with, the index of the body of the other component that overlaps, the bFromSweep flag indicating if the overlap is from a sweep, SweepResult flag indicates if the overlap is from a sweep. OtherActor refers to the actor with whom the component overlaps. OtherComp refers to the actor with whom it overlaps. OtherBodyIndex points to the body of the other actor with which it overlaps. bFromSweep indicates whether the overlap is from a sweep. The sweep result structure is used if the overlap is from a sweep. This event is generally triggered to detect when an object enters or leaves the collision volume of another object. It can be used to trigger various gameplay events such as the spawning of objects or triggering animations. In the below figure (FIGURE 21), it is difficult to see all the parameters, but these are needed for the overlap events. You can find these delegates if you search for the UPrimitiveComponent from the Engine of your project visual studio solution files. In Unreal Engine, *BoxTraceSingle* is a function that allows you to check for collisions based on the created box collision. It takes in several parameters such as the start and end point of the trace, the extent of the bounding box, and any actors or components to ignore during the trace. It then returns information about any actors or components that were hit by the trace, such as their location and normal (Unreal Engine 2004-2023b.). *WeaponTraceStart* and *WeaponTraceEnd* scene components are created so that in the child blueprint we can determine the exact start and end point of the trace.

```cpp
void ADefaultWeapon::CombatOnOverlapBegin(UPrimitiveComponent* OverlappedComponent, AActor* OtherActor,
{
    const FVector Start = WeaponTraceStart->GetComponentLocation();
    const FVector End = WeaponTraceEnd->GetComponentLocation();

    TArray<AActor*> ActorsToIgnore;
    ActorsToIgnore.Add(this);

    FHitResult WeaponHit;

    UKismetSystemLibrary::BoxTraceSingle(
        this,
        Start,
        End,
        FVector(5.0f, 5.0f, 5.0f),
        WeaponTraceStart->GetComponentRotation(),
        ETraceTypeQuery::TraceTypeQuery1,
        false,
        ActorsToIgnore,
        EDrawDebugTrace::ForDuration,
        WeaponHit,
        true
    );
```

FIGURE 21. Box trace from the weapon when an overlap event happens.

Hatchets are another weapon we have which is an object that works as a ranged weapon. Hatchets can be flung at enemies, but they will be destroyed upon impact. This type of weapon has much more complicated logic to implement physics for these hatchets than for the primary weapon. This is because it is not just a static object that is held, but also thrown. If thrown objects need to simulate the physics of an object that has been thrown. In addition, there was a limited number of hatchets for the player to have. Therefore, there needed to be logic that would keep track of how many hatchets were left and if the player could fire a hatchet at a given moment. However, how this weapon inflicts damage on the enemy is the same as a primary weapon by using ray tracing to detect when the enemy is being hit.

5.3 Base Enemies

As for the enemies, we created a base character class called BaseEnemy from which all the enemies found in a level will be derived. It is still undetermined if we use this base class for the boss enemy or if we create a separate boss enemy character class. In the character class, we need to determine collision channels in the constructor class (FIGURE 22). This is because our weapons collision trace traces specific channels. We need to enable them on enemy meshes so that it knows when the weapon collides with the enemy. So, we collision channel is set to block the visibility channel. Furthermore, we modify the collision channel on the enemy's mesh and capsule component to ignore the player's camera on both components since we worked with a first-person shooter where there is a high likelihood that the camera will collide with the enemy components. Finally, we

needed to set enemies to mesh to generate overlap events by using a boolean and setting it to true. This boolean can be found by default in all character classes.

```
// Sets default values
ABaseEnemy::ABaseEnemy()
{
    // Set this character to call Tick() every frame.  You can turn this off to improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

    /** Collision channel presests that will detect when the collision channels of the weapon collides enemy mesh */
    GetMesh()->SetCollisionObjectType(ECollisionChannel::ECC_WorldDynamic);
    GetMesh()->SetCollisionResponseToChannel(ECollisionChannel::ECC_Visibility, ECollisionResponse::ECR_Block);

    /** Enemy mesh ignores players camera when colliding with it */
    GetMesh()->SetCollisionResponseToChannel(ECollisionChannel::ECC_Camera, ECollisionResponse::ECR_Ignore);

    /** Set enemys mesh to generate overlap events */
    GetMesh()->SetGenerateOverlapEvents(true);

    /**Enemy capsule component ignores player camera */
    GetCapsuleComponent()->SetCollisionResponseToChannel(ECollisionChannel::ECC_Camera, ECollisionResponse::ECR_Ignore);
```

Figure 22. Collision pre-sets in BaseEnemy CPP construct function.

We used Unreal Engine interfaces to determine when a box trace collides with an enemy character that implements the interface by casting to the interface in our BaseWepaon.CPP. Interface classes are useful for ensuring that a set of (potentially) unrelated classes implement a common set of functions. This is very useful in case some game functionality may be shared by large, complex classes that are otherwise dissimilar. We have a Character class for an enemy and an Actor class for a base weapon. So, all these classes need shared functionality but have no common ancestor other than the interface with the common function. In this case, an interface is recommended (Unreal Engine 2004-2023m.).

5.4 Combat system

The player character uses left mouse button click input to play an array of random attack animations from an animation montage when performing melee attacks. Collision on ray tracing logic that was explained previously is applied to detect when the weapon collides with the enemy mesh. This is helpful for knowing when the time is right to play sounds that correspond to hitting something. Particle systems for blood spurting for example. However, one of the most critical systems is how to inflict damage on the enemy. In Unreal Engine, the TakeDamage function can be used to deal damage to objects or actors (FIGURE 23). Custom damage classes can be created through it. It can be called on any actor with a health component and the ability to take damage, such as a player character or an enemy. The function takes several parameters, including the amount of damage to be applied, the actor or object responsible for the damage, and any optional flags or

modifiers. In our case, the TakeDamage function was overridden in our base enemy class. In addition, ApplyDamage was added to the base weapon and hatchet class. (Unreal Engine 2004-2023a.).

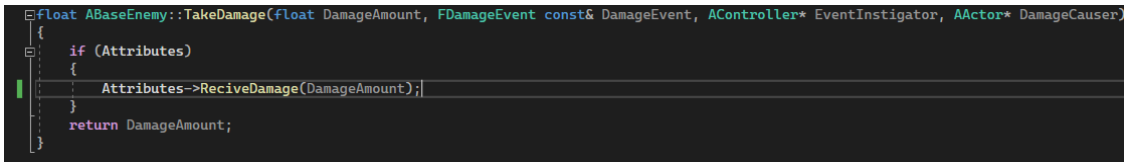


FIGURE 23. TakeDamage function in BaseEnemy CPP that uses Attribute components Receive Damage function on enemy character.

The *ApplyDamage* function in Unreal Engine is a function that can be used to deal damage to an object or actor in a game (FIGURE 24). It is like the T*akeDamage* function, but ApplyDamage is a *BlueprintCallable* function and can be called from Blueprint or C++ code. This is because it does not need the object or actor to have a health component, and it returns the amount of damage dealt. There are several parameters included in the *ApplyDamage* function such as base damage that we have set as a variable in our weapon header file, which determines the damage to apply. This is followed by DamageType, which specifies what damage is being inflicted, and Instigator, which is an optional parameter that specifies the actor responsible for causing the damage. *HitInfo*, which is an optional parameter that describes the point of impact and any additional information about the hit, such as the type of bone hit. DamagePreventionFlags is an optional parameter that specifies whether to prevent damage. The function returns the amount of damage that was applied to the object or actor. This can be useful for game logic, such as tracking how much damage was dealt with or determining if an object or actor was destroyed.

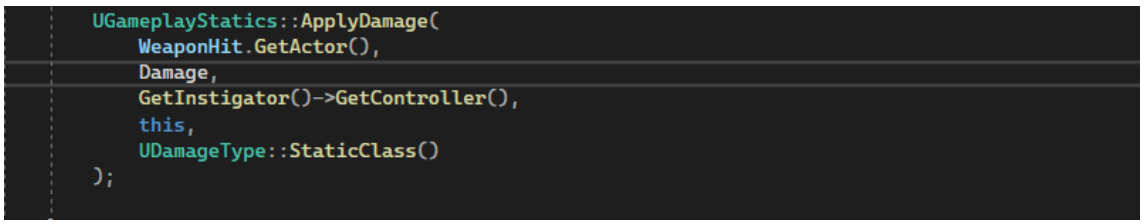


FIGURE 24. ApplyDamage function that is called by using UGameplayStatic to deliver appropriate damage amount from the equipped weapon.

Ranged attacks will consist of throwing a hatchet, a small axe-like weapon that will be destroyed upon impact, as well as different kinds of spells. Both ranged attacks have limitations, such as how many hatchets the player can carry at once. The hatchet weapon itself does not require further

design, but we have been prototyping its behavior with blueprint logic, rather than coding C++ code. It is therefore much easier to understand why vectors and rotators behave the way they do after creating a blueprint version first. In the future, we aim to make this part of combat more C++-oriented, but heavier. However, currently, there is a working prototype in the character blueprint class. While the hatchet-throwing logic is blueprint-based, the damage inflicted will still follow the C++ collision and ray-tracing logic. About magic attacks, we are still considering how they should look. This includes how they should look, what kind of animations should be created, and what the effects should be like. To give some examples - will the player gain some attributes, and/or can the player attack enemies with these spells? The only thing that we are sure of is that we would like to come up with something more creative for the magic system. This is not the regular fireball, ice attack type thing that you usually see in roleplaying games.

5.5 Inventory system

An inventory system in Unreal Engine is a way to manage and track the items a player character has in their possession. Items and other collectibles can be found in Tuonela. There are a few different ways to implement an inventory system in Unreal Engine. One common approach is to use an array or a list to store items in the inventory. Each item can be represented by a structure or a class that contains information such as the item's name, description, and sprite. The inventory class can then have functions to add, remove, and check for items in the array. This method was what we used in the demo, and we chose to use TArray which is dynamically sized (Unreal Engine 2004-2023w.). The player character also holds a collection of several different functions created for different inventory functions. The array of inventory items is derived from the item data structure that was seen previously. In addition, we created various functions that can handle inventory data. These functions include adding items to the inventory, updating the widgets that show inventory items, and removing them from the UI after a player uses them. In addition to the UI, another significant aspect of an inventory system is the user interface (UI) that displays the inventory to the player. This can be done using Unreal Engine's built-in UI system, such as UMG widgets. The inventory UI can show the player's current items, and allow them to interact with them, such as selecting or equipping an item. It is also worthwhile to note that the inventory system can be expanded with features such as weight limit, item stack, item durability, item unique identification, and many more. The shop system we have in place is a very common and basic type of inventory

system. Interacting with a character who owns a shop is also a part of the player characters' header file functions.

The function *AddInventoryItem* uses a for each loop and a reference to the structure of *ItemData* to loop through the inventory items array. This is an inventory that the player is assigned to. We have ItemClass, a subclass of *DefaultItem,* that child blueprint class can be assigned in the blueprint as we saw previously. A comparison is made between the coming item data class and the item data class that may or may not exist in the player's inventory. If an inventory item already exists the current stack amount will be increased, and the item will be added. Newly entered inventory items are traced with the boolean *bIsNewItem.* If the item is a newly created inventory item, it is added using TArray and its function *Add()* which adds a new item at the end of the array (Unreal Engine 2004-2023v.). There is an *AddItemAndUpdateInventoryWidget* blueprint implementable event that the UI elements can use to understand this logic (Unreal Engine 2004-2023ab.). The event does not contain any logic written in code, but simply feeds the information to an event that is referenced in the blueprint for the UI information. This function can be used for items placed in the world as well as for items that are in the non-playable character shop (FIGURE 25).

```
void APlayerCharacter::AddInventoryItem(FItemData ItemData)
{
    bool bIsNewItem = true;

    for (FItemData& Item : InventoryItems)
    {
        if (Item.ItemClass == ItemData.ItemClass)
        {
            ++Item.CurrentStackAmount;
            bIsNewItem = false;
            break;
        }
    }
    if (bIsNewItem)
    {
        InventoryItems.Add(ItemData);
    }
    if (InventoryItems.Num())
    {
        AddItemAndUpdateInventoryWidget(InventoryItems[InventoryItems.Num() - 1], InventoryItems);
    }
    else
    {
        AddItemAndUpdateInventoryWidget(FItemData(), InventoryItems);
    }
}
```

FIGURE 25. Add Inventory Logic in PlayerCharacter.CPP file.

The *Interaction* function in the DefaultItem class in the CPP file uses *AddInventoryItem* logic to add items to the player character's inventory. When the player interacts with an item in the game world,

the *AddItemInventory* function, which takes in the *ItemData* and adds item information to the player's inventory array (FIGURE 26). In child C++ classes that inherit from the DefaultItem class, the Interaction function is then overridden. This tells the player specific item information depending on which item the player is interacting with. It is imperative to note that in each child class, the *Super::Interact()* function needs to be called for the implementation and logic of the Interaction function to be carried over from the parent class to the child class, as intended (Unreal Engine 2004-2023u.). However, in weapon classes, *Super::Interact()* is not called because we do not want them to behave like inventory items when interacting. Instead, the Interaction function is overridden in these classes to provide custom behaviour specific to weapons.

```
void ADefaultItem::Interact(APlayerCharacter* PlayerCharacter)
{
    if (PlayerCharacter)
    {
        PlayerCharacter->AddInventoryItem(ItemData);
    }
    Destroy();
}
```

FIGURE 26. DefaultItem Interact function logic for adding items to players' inventory array.

The *RemoveItem* function removes items from the inventory as its name implies. It has some default parameters defined as *ItemSubclass*. Integer *AmountToRemove* and boolean *UseItem*. In a similar fashion to adding items to inventory, we use a for loop to iterate through the *InventoryItems* array that exists in the inventory that the player has. The items found in item class and item subclasses are compared to determine the use of the right item. Using the bool *UseItem* and the reference to the DefaultItem actor class with the item subclass that was assigned in child blueprints, if the item is something the player can use, it will be made available. If multiple items exist in the stack, one will be deleted from the current stack. If the stack amount reaches zero, we call the RemoveAt function that TArrays has and find the index of the item to be deleted in the function (Unreal Engine 2004-2023x.). To remove it completely from the inventory, meaning the item will be removed visually as well as from the array of items that the player possesses. Lastly, we call in the if-else statement the blueprint function *AddInventoryAndUpdateInventroyWidget* with the *Num()* function which returns which element number we are executing our logic at (Unreal Engine 2004-2023i.) in the if statement if there are changes in UI meaning we need to eliminate an item completely from the inventory we return the inventory and removed old value else we will just update the inventory to correspond with current stack amounts (FIGURE 27).

```cpp
void APlayerCharacter::RemoveItem(TSubclassOf<ADefaultItem> ItemSubclass, bool UseItem, uint16 AmountToRemove)
{
    uint8 Index = 0;

    for (FItemData& Item : InventoryItems)
    {
        if (Item.ItemClass == ItemSubclass)
        {
            if (UseItem)
            {
                if (ADefaultItem* ItemCDO = ItemSubclass.GetDefaultObject())
                {

                    ItemCDO->Use(this, false);
                }
            }

            Item.CurrentStackAmount -= AmountToRemove;

            if (Item.CurrentStackAmount <= 0)
            {
                InventoryItems.RemoveAt(Index);
            }

            break;
        }
        ++Index;
    }
    if (InventoryItems.Num())
    {
        AddItemAndUpdateInventoryWidget(InventoryItems[InventoryItems.Num() - 1], InventoryItems);
    }
    else
    {
        AddItemAndUpdateInventoryWidget(FItemData(), InventoryItems);
```

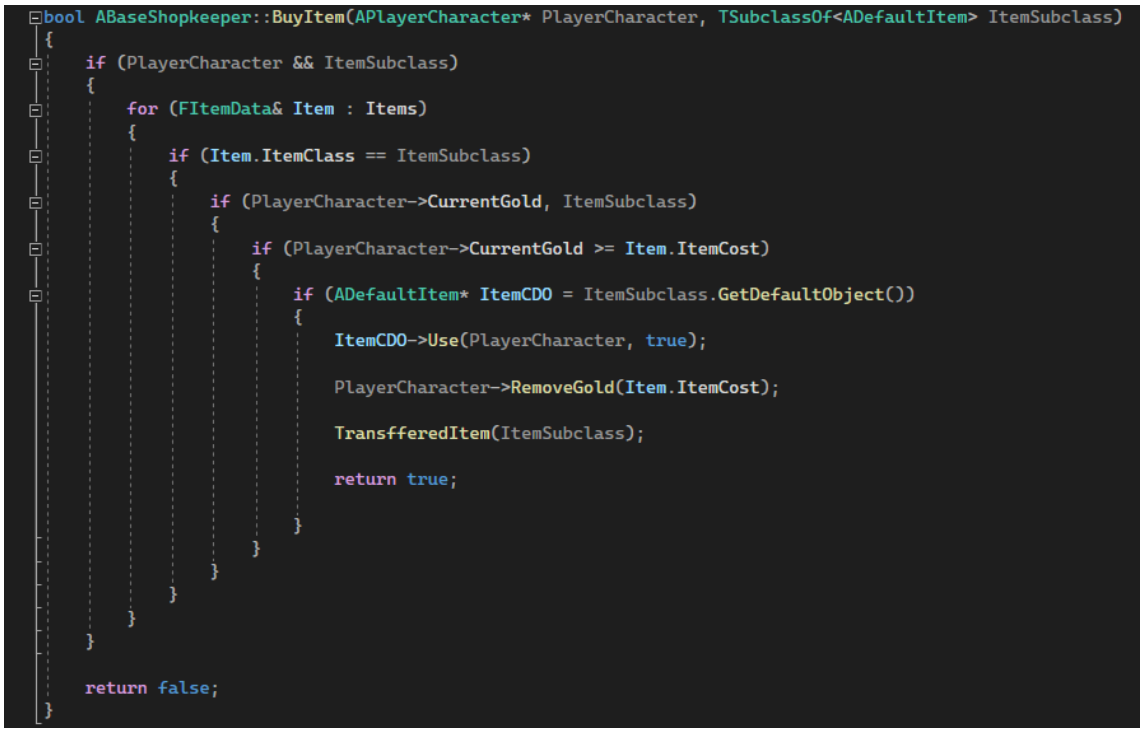
FIGURE 27. RemoveItem function in player character CPP file.

The AddItemFromShop function is used when the player interacts with the shop of non-playable characters. Again, we define some default parameters such as ItemSubclass. This helps us to find that specific item that the player interacts with. Then we have the BaseShopkeeper character class and the boolean IsShopItem. Our first step is to check that the item dealt with is or is not in the shopkeeper's inventory. If it is in the player's inventory, we will execute logic to remove it from our own inventory. If it is the shopkeeper, we call a function from BaseShopkeeper character class *BuyItem()* which is a boolean function that transfers the item from the shopkeeper's inventory to the player's inventory based on parameters of PlayerCharacter and *ItemSubclass*. The AddItemAndUpdateInventoryWidget function is called for the same reason as the RemoveItem function to feed information to UI widgets accordingly (FIGURE 28).

```
void APlayerCharacter::AddItemFromShop(TSubclassOf<ADefaultItem> ItemSubclass, ABaseShopkeeper* ShopKeeper, bool IsShopItem)
{
    if (ItemSubclass)
    {
        if (!ShopKeeper)
        {
            RemoveItem(ItemSubclass, true);
        }
        else
        {
            ShopKeeper->BuyItem(this, ItemSubclass);

            if (InventoryItems.Num())
            {
                AddItemAndUpdateInventoryWidget(InventoryItems[InventoryItems.Num() - 1], InventoryItems);
            }
            else
            {
                AddItemAndUpdateInventoryWidget(FItemData(), InventoryItems);
            }
        }
    }
}
```

FIGURE 28. The AddItemFromShop function is found in the PlayerCharacter CPP file.

As part of its *BuyItem* function, the *BaseShopkeeper CPP* character class file performs several checks inside a for loop to determine whether the player has enough gold to buy the item. In addition, it determines whether the item matches the item in the shopkeeper's inventory of the item that the player intends to buy. After all the checks pass, the function calls two other functions to remove the amount of gold that the item costs from the player's character with the simple function that can be found in the player character CPP file as well as “transfer” the item from the shopkeeper's inventory to the player's inventory (FIGURE 29).



```
bool ABaseShopkeeper::BuyItem(APlayerCharacter* PlayerCharacter, TSubclassOf<ADefaultItem> ItemSubclass)
{
    if (PlayerCharacter && ItemSubclass)
    {
        for (FItemData& Item : Items)
        {
            if (Item.ItemClass == ItemSubclass)
            {
                if (PlayerCharacter->CurrentGold, ItemSubclass)
                {
                    if (PlayerCharacter->CurrentGold >= Item.ItemCost)
                    {
                        if (ADefaultItem* ItemCDO = ItemSubclass.GetDefaultObject())
                        {
                            ItemCDO->Use(PlayerCharacter, true);

                            PlayerCharacter->RemoveGold(Item.ItemCost);

                            TransfferedItem(ItemSubclass);

                            return true;

                        }
                    }
                }
            }
        }
    }

    return false;
}
```

FIGURE 29. BuyItem function in ShopKeeper CPP file.

The *TransferItem* function is just for each loop that checks for correspondence between the Item class and the item's class. It then removes the item at the index chosen by the player. This is either subtracting the stack amount or removing the item completely if it is the last item on the stack (FIGURE 30).

```
void ABaseShopkeeper::TransfferedItem(TSubclassOf<ADefaultItem> ItemSubclass)
{
    uint8 Index = 0;

    for (FItemData& Item : Items)
    {
        if (Item.ItemClass == ItemSubclass)
        {
            --Item.CurrentStackAmount;

            if (Item.CurrentStackAmount <= 0)
            {
                Items.RemoveAt(Index);
            }

            break;
        }
        ++Index;
    }

    APlayerCharacter* PlayerCharacter = Cast<APlayerCharacter>(UGameplayStatics::GetPlayerCharacter(GetWorld(), 0));

    OpenShop(PlayerCharacter);
}
```

FIGURE 30. TransferredItem function in ShopKeeper CPP file.

Because we call *Use()* function and TransferItem function it will then visually look like a player just moved inventory item from shop keeper inventory. *Use()* function works similarly to the Interact function by feeding ItemData to the function as well as the *IsInShop* boolean value to determine that the item is in the shop. If all the conditions are passed item is moved to the inventory (FIGURE 31).

```
void ADefaultItem::Use(APlayerCharacter* PlayerCharacter, bool IsInShop)
{
    if (IsInShop && PlayerCharacter)
    {
        PlayerCharacter->AddInventoryItem(ItemData);
    }
}
```

FIGURE 31. Use a function that handles the addition of the item from the shop inventory to the players' inventory.

In the player character blueprint, the C++ event AddItemandUpdateInventory is called which was used in all the functions related to inventory. The widget which is responsible for inventory as well as shop UI functionality called shop inventory uses values that the C++ code delivers through the functions that are responsible for adding items to inventory, removing items from inventory, and

updating existing item information in the inventory. A widget's elements of the inventory system include many blueprint functions seen in a blue header. As the name suggests, they handle additions and updates of the UI elements in both the player inventory widget as well as shop keepers inventory widget (FIGURE 32).

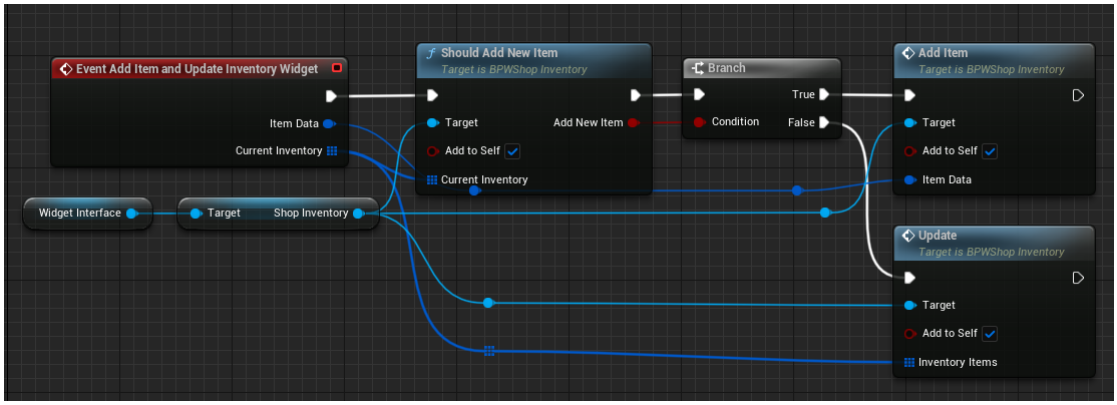


FIGURE 32. AddItemAndUpdateInventoryWidget called in the player character child blueprint class.

5.6 UI Design and logic

We are in the process of testing the UI/UX design now meaning that a lot of UI elements in the demo are still very bare-boned and not much-styled elements in the game. UI/UX elements are being tested to ensure that they display the correct information visually and that they respond to code logic correctly. Additionally, we have a basic layout of the main menu, but it is still in very much in the design stages. It will still take a lot of design and work to make the UI elements look right and fit the game's aesthetics. UI design is planned to be minimalist and especially the gameplay screen is aimed to look as clean as possible without much of a UI element so that the player can appreciate the game environment as well as merge themselves into the game without many distractions (FIGURE 33).

FIGURE 33. Prototyping player HUD UI in the gameplay view.

At present, our UI elements are heavily reliant on visual scripting languages rather than C++ programming. In the Unreal Engine, UI widgets can be customized to display a variety of information, such as text, images, and other graphical elements. For example, a text widget can be used to display a string of text on the screen. An image widget can be used to show a picture or other graphic. To display information about an item, we use the item data structure that we created in C++ as a data source. This is a data source that holds the information we want to display in the inventory widget. This can be done in Unreal Engine's UI editor or by using blueprints or C++ code. Once the widget is bound to the data source, any changes to the data will automatically be reflected in the widget on the screen. This allows you to create dynamic UI elements that can change based on the state of the game or application.

6 MIGRATION FROM UNREAL ENGINE 4 TO UNREAL ENGINE 5

When Tuonela was first created it was made to be using unreal engine 4 as the time of its creation Unreal Engine 5 was not yet released. It was just a beta version that was not yet a stable option to be used. However, when Unreal Engine 5 was officially released there was no question to convert the project from Unreal Engine 4 to Unreal Engine 5. When the conversion was made the copying method was chosen for the Tuonela. This was because so much work had already done a lot of work on the demo and the possibility of conversion turning terribly wrong and destroying our meticulous work.

6.1 Lumen, Nanite, and World Partition System

When a new project is created with Unreal Engine 5 the new Lumen lighting system is active by default; however, the project was migrated from Unreal Engine 4 to Unreal Engine 5, and some additional steps were required for Lumen to be active in Tuonela. In the figure below can be seen what parameters on the project setting needed to modify for Lumen to work to its full potential (FIGURE 34). This is what we did with Tuonela to get the Lumen lighting system working for the project. (Epic Games 2004-2022b.).

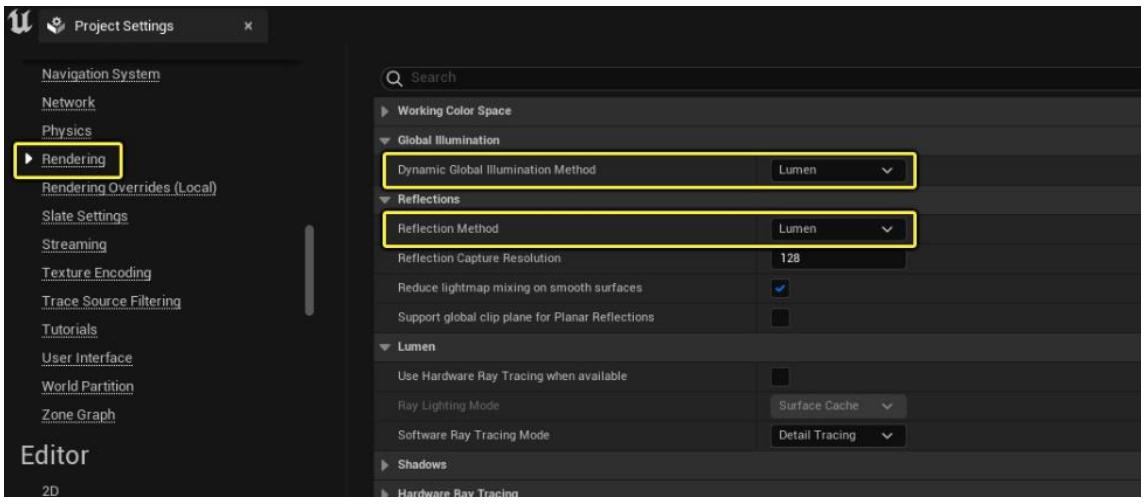


FIGURE 34. Unreal Engine 5 rendering project setting panel (Epic Games 2004-2022b.).

A second new feature implemented with Unreal Engine 5 was the use of Nanite for static meshes. There were many reasons why Nanite seemed to be a logical option to be implemented. This is

because it enables world geometry to be much more complex than before. In addition, frame budgets would not be reliant on the polycounts of the meshes, draw calls, or memory usage. Another major benefit was that LOD is automatically handled by Nanite, so there is no need for manual setup for individual meshes and this way loss of quality is non-existent. However, Nanite works best on static meshes so skeletal meshes like characters and rigged items should still be optimized traditionally by baking details into normal map textures. It can be enabled by selecting "Build Nanite" in the mesh menu when importing assets or by selecting already existing assets and left clicking on Nanite to enable it (FIGURE 35). In Tuonela, it was primarily accomplished by selecting existing static mesh assets. (Unreal Engine 2004-2022o.)

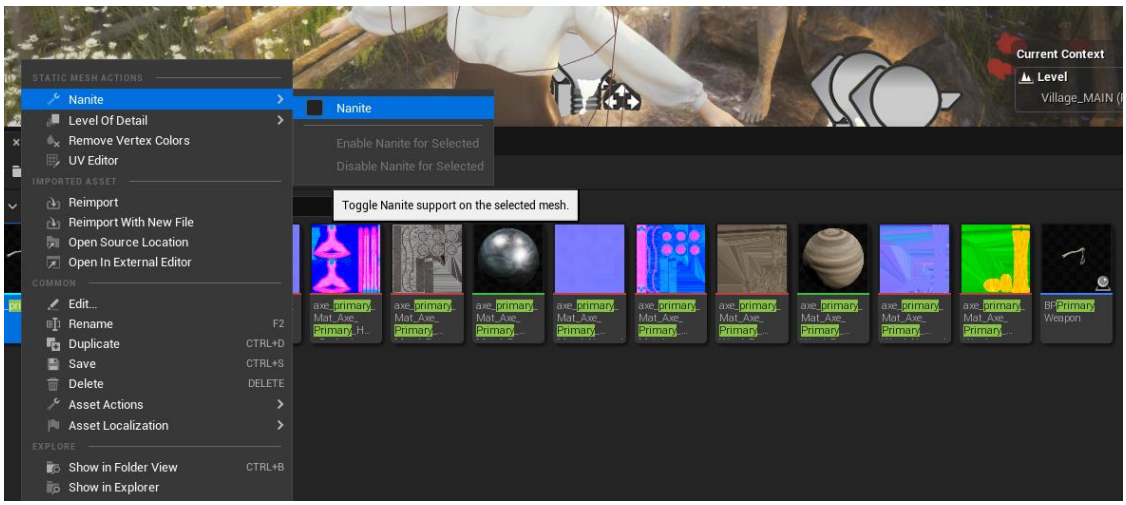


FIGURE 35. How to use nanite in existing static meshes.

The Level to World partition in Unreal Engine 5 is a system that divides the game world into smaller parts, called levels. These parts can be loaded and unloaded as the player moves through the world. This allows for more efficient use of memory and processing power, as only the levels that the player is currently interacting with need to be processed and rendered. This behavior is implemented differently in Unreal Engine 4 by using collision volumes along with level streaming volumes and several levels to navigate between the areas. Each level in Unreal Engine 5 is a self-contained unit that contains all the necessary data for that specific area of the game world, including geometry, textures, lighting, and gameplay elements. When the player moves from one level to another, the engine seamlessly loads the new level and unloads the previous one, without any interruption to the player's experience. The Level to World partition also allows for more efficient use of the GPU. This is because it can focus rendering only on the visible levels rather than having to render the entire game world at once. To convert existing levels to world partition levels, access the tools

menu of the Unreal editor and under the world, partition menu selects the converted level. This will pop up an asset dialogue where the user can choose which level to convert. Once the level is selected, a second window will open that has the conversion setting options in it. In the demo, we left everything as default. However, we did make a note that the variable InPlace could be checked. This would convert the existing level, but if it is left blank it will create a copy of the level. As we have a lot of big levels and we have already worked on our current levels a lot, we rather make a copy. This is so that we do not destroy any of the work already done. The process was a bit tedious since you needed to convert level one at the time. However, this will benefit us in the long run when starting to optimize game performance when playing.

6.2 Enhanced Input system

In Unreal Engine 5.1, Unreal introduced a newly enhanced input system, which deprecated the more traditional axis and action mapping input systems. In contrast to regular action mappings, enchanted action mappings provide much greater control over inputs without overcomplicating the way systems work. To use enhanced inputs with Unreal Engine 5.0 there needs to be separate plugins as well as changes made to the project settings input. Nevertheless, if the project is converted to 5.1 all these settings will be automatically considered. However, adjustments to the input types, as well as code, need to be made if the project originates from Unreal Engine 4. This is because projects created with previous versions still rely on an older input system. Enhanced input will continue to work with older input codes since it uses the same capabilities and child classes as older input. Due to the possibility that input systems could very well change in the future, it is prudent to convert to enhanced input systems already at this stage of development. Enhanced input actions differ from the original inputs in the way that they do not require changes to the code. Instead, they are the bridge between the system and the code. (Unreal Engine 2004-2022e.). As a result, updated input action mappings were created (FIGURE 36). A new mapping context needs to be assigned to the character and in the demo, we created a new UInputMappingContext* variable was created for the player character and exposed it to the blueprint so that the newly created input mapping context could be assigned to it (Unreal Engine 2004-2023ac.).

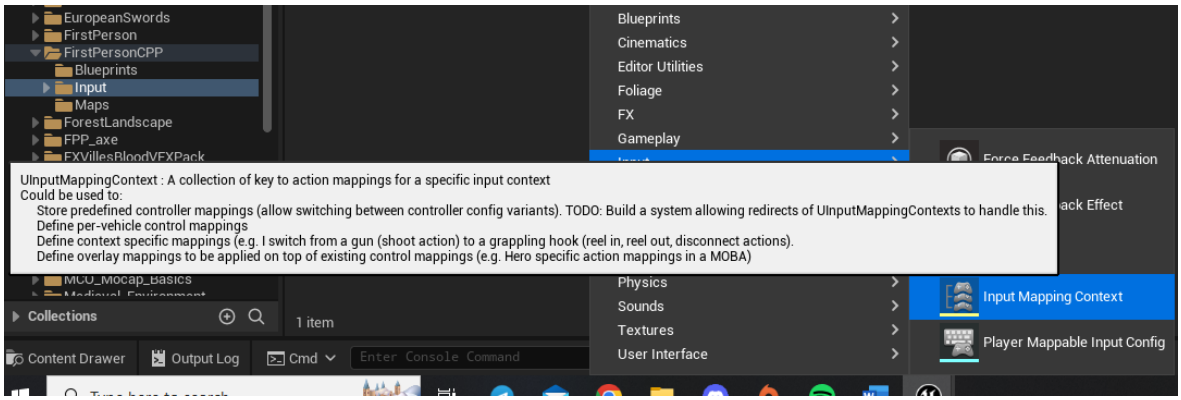


FIGURE 36. Tuonela Adding New Input Action Mapping Context for Player character that contains all action inputs and axis that were previously defined on project settings.

As well as mapping context, there is a streamlined way to create player inputs. In previous Unreal Engine versions, all the inputs were previously defined in the project settings but after Unreal Engine version 5.1 inputs are not recommended to be defined in project settings anymore (Unreal Engine 2004-2023ac.). Inputs are created using the newly added Input option under the same menu as when setting up an input mapping context (FIGURE 37).

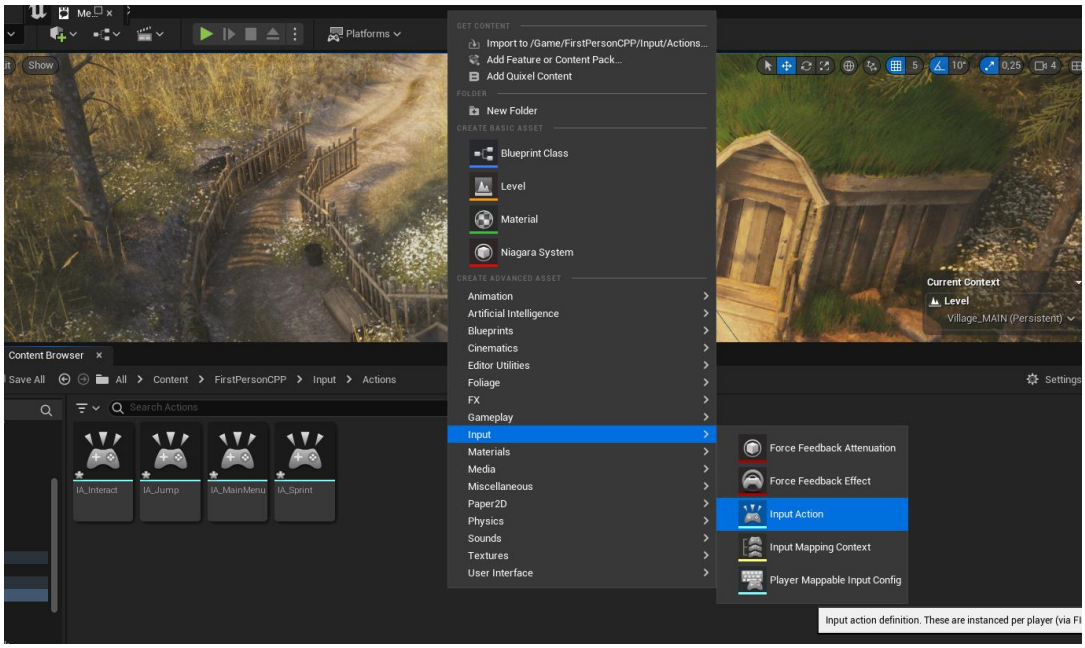


FIGURE 37. A new way of creating individual input actions in Unreal Engine project.

The input action needs to be added to the mapping context for the theme to work with the character assigned using the mapping context. The mapping context offers much greater control over the behaviour of input actions. Previously forward and backward movements were created as separate input actions. However, in the updated input system all of these can be included in one input action.

With the different modifiers that can be added multiple times to keystrokes, it is possible to determine how each keypress behaves. This is shown in the following action mapping context that uses the Move input action as one of the examples (FIGUER 38).

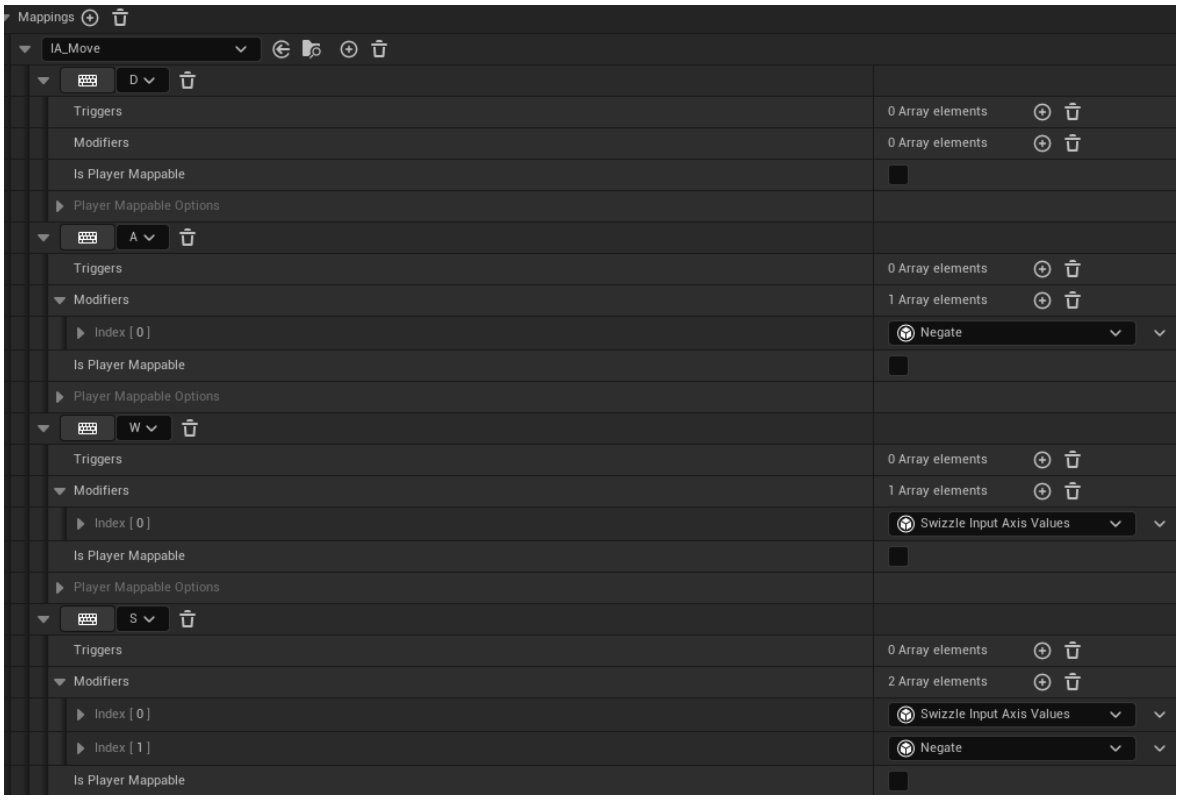


FIGURE 38. New Asset type InputMappingContext that hold all the input mapping actions which were previously found from the project setting.

Although some code still needs to be added to the C++ class to get the actual movement, there is significantly less programming required than in previous engine versions. This is because there are no character input methods. All that will be needed for the logic of the character moving left/right with the A, D, W, and S keys are shown below (FIGURE 39).

```
/** New enchaded input system move back and forth */
void APlayerCharacter::Move(const FInputActionValue& Value)
{
    const FVector2D MovementVector = Value.Get<FVector2D>();

    const FVector Forward = GetActorForwardVector();
    AddMovementInput(Forward, MovementVector.Y);

    const FVector Right = GetActorRightVector();
    AddMovementInput(Right, MovementVector.X);
}
```

FIGURE 39. New Move function for implementing logic for character movement.

7 TUONELA GAME ASSETS DESING

For the game assets, Tuonela used a variety of pre-made assets that can be found in Quixel or the Unreal Marketplace. However, there are also unique assets in the demo. Notably, most of the characters were made by the developers as well as many of the environment pieces that are specific to the game are created by the developers. Game assets are created from scratch whenever we could not find suitable assets in the two places mentioned. We use several third-party programs when creating assets, such as Blender and Zbrush, which are 3D modeling tools that we prefer to use. If you need bake details for assets, the Marmoset Toolbag is preferred, and if you need texturing, Adobe substance painter is a reliable 3rd party tool. For character creation, a combination of Reallusion Character Creator, Marvelous Designer, and one of the 3D programs is used.

7.1 Readymade Game Assets Origin

The Tuonela demo was designed using some Quixel scans, and they were notably mostly used to design foliage patterns, cliffs, and rocks, and to create some textures for the environment (FIGURE 40). A few other ready-made meshes were included in the game as well, such as crates and barrels, that fit into the environment of the game and were added. Quixel mega scans are the largest 3D scan library. People who are developing with the Unreal Engine have unlimited access to this library. All the assets are high quality and standardized so that they are ready to use. It is part of Epic Games. There is no license cost for Quixle Megascans assets in games intended to be published from the free library (Epic Games. 2022q.).

FIGURE 40. Quixel assets range from foliage to trees and rock cliff formations as seen in the picture.

There is a fee associated with larger libraries and productions. Even for free library assets, active subscriptions are required so that users can agree on the usage and license terms that these assets fall under. (Epic Games 2022r.). In the Unreal Engine marketplace, anyone can sell their assets, blueprints, or audio files for use in the game engine. Epic Games reviews all submissions for the marketplace first before they become available for purchase. Our project includes several different assets that we purchased from the marketplace. These assets range from animations to different kinds of static as well as skeletal meshes to sounds and VFC and particle systems.

7.2 Custom Game Assets

As of right now, most of our game characters are planned as assets that we have created ourselves. There are several tools that are used when creating custom characters. These include Character Creator, Blender, Marvelous Designer, and Adobe Substance Painter. Each of these pieces of software has a specific role to play in the creation process, and each of them has its own advantages. When creating characters, we use a combination of programs for time-saving purposes and to provide more advanced features than Blender which is software that could be solely used to create characters. It has all the features you need from modeling to rigging to baking in place. All the software mentioned above requires a subscription, except Blender.

Character Creator is used to creating custom character bases. It contains features for quickly adjusting 3D character models to specific preferences. It also allows for the automatic creation of a

level of detail for the character model. This saves a lot of time from starting to model a character base from scratch. The model can be exported as either FBX or OBJ once it is ready. There are other export options as well but for our purposes, these two are the ones that are used. The model is first brought into Blender to check that the model dimension is correct and that the model does not have any other possible issues. If the character is a human, once all of that is confirmed it is opened in Marvelous Designer so that creating clothing for a character can begin. For monster-type characters, there is often a lot of additional modeling needed to be done to make them look less like humans, and not all monsters in the game use the human model as a base.

Marvelous Designer is 3D clothing design software that enables users to create realistic garments for use in animation, film, and video games. The software enables users to create patterns, create and edit garment details, and preview the draping and movement of the fabric. Some of the key features of Marvelous Designer are garment creation. This allows users to draft and alter patterns for a wide variety of garments, including shirts, pants, and dresses. Textile simulation allows users to simulate the draping and movement of fabric, providing a realistic look and feel to the garments. 2D & 3D garment editing in both 2D and 3D, making it simple to design and fine-tune patterns and details (FIGURE 41). Dynamic simulation enables users to produce animation-ready garments by simulating the movement of fabrics in real-time. 3D Posing characters allow users to pose garments on 3D characters, making it easy to create realistic renderings. Marvelous Designer allows users to import and export garments to and from other 3D software in our case Blender. Users can design dynamic and realistic folds, wrinkles, and creases on the cloth surface using the tools. With Live Simulation, users can see designs come to life in real time, enabling them to create better designs

before rendering. The interface is highly customizable, enabling users to tailor the software to their specific needs and workflow (CLO Virtual Fashion Inc. 2023.).

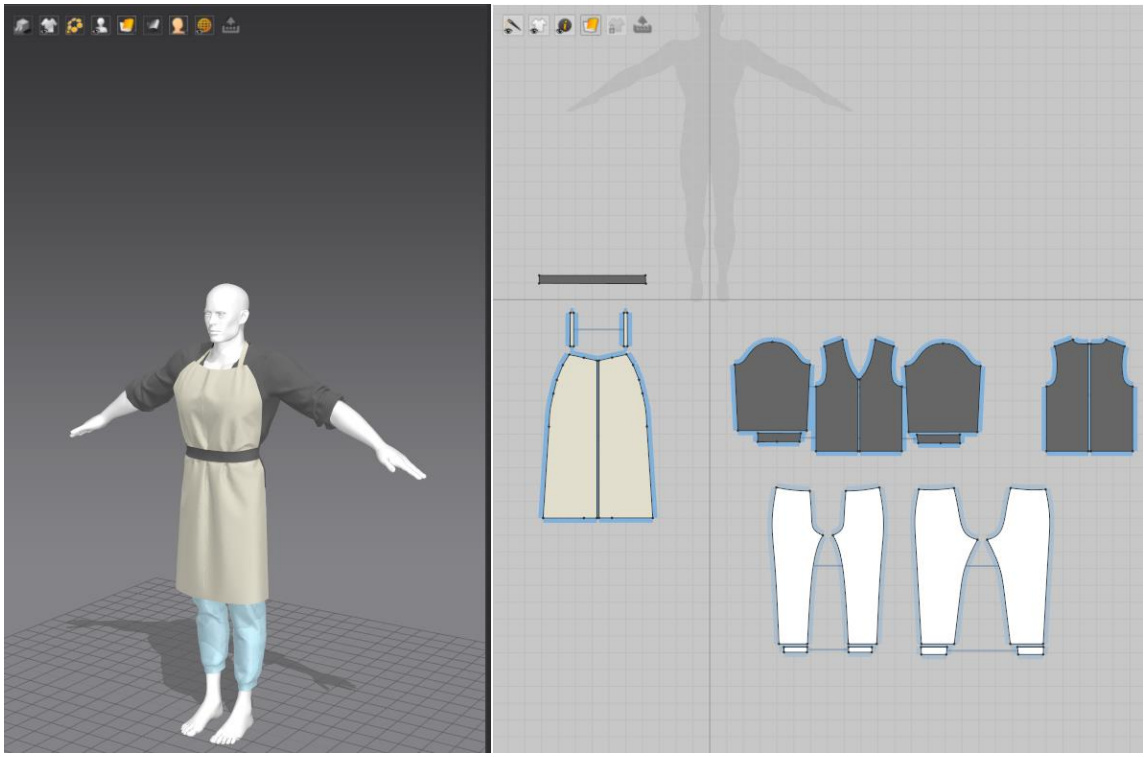


FIGURE 41. Marvelous Designer view of character clothing creation using the Character Creator model.

It is worthwhile to note that some character clothing items are still modelled in Blender such as shoes. There are other items that do not benefit from fabric simulation such as jewellery. After the clothes have been created, they can be imported into Blender as OBJ files. Personally, I prefer using blender's add-on auto rig pro for retopology. Retopoflow 3 provides a set of tools for retopologizing 3D models. The process of retopologizing involves simplifying and reducing the polygon count of a model while maintaining its overall shape and details (FIGURE 42). This add-on automates and streamlines common retopology tasks to make this process more efficient. The Retopoflow 3 add-on simplifies the process of retopologizing 3D models in Blender through its powerful and versatile features. It can save a lot of time and effort in the retopologizing process, but it requires some knowledge of Blender and retopology techniques (Orange Turbine 2022. Product, RetopoFlow - Retopology Toolkit for Blender).

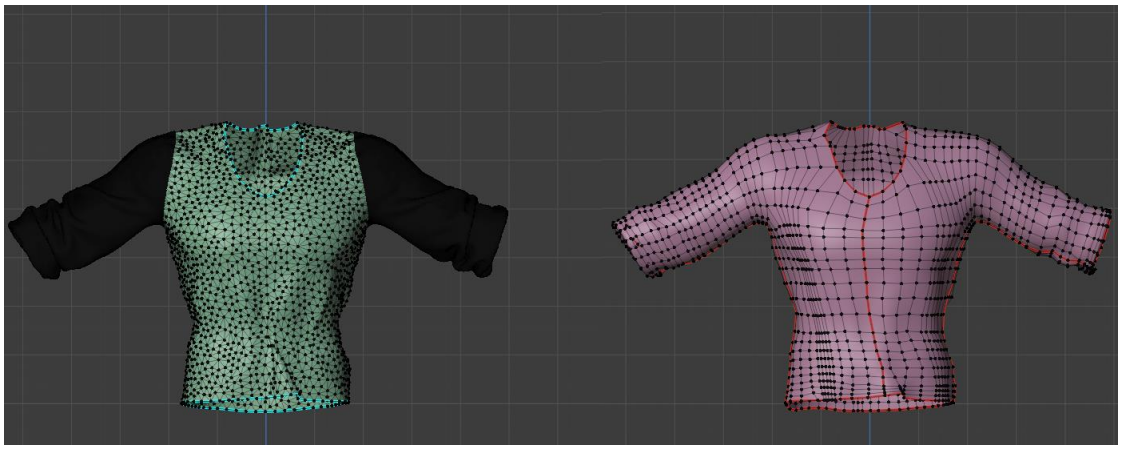


FIGURE 42. The left high poly model and the right low poly model were created by retopologizing.

For rigging purposes, we use Auto Rig Pro which is a Blender add-on that allows users to build a rig quickly and easily for their 3D models. The add-on provides a user-friendly interface for creating character rigs, including options for controlling the body, face, and fingers (FIGURE 43). It also includes a variety of advanced features, such as support for custom bone shapes and the ability to create inverse kinematic chains (Blender Market, LLC 2023a.). The Auto Rig Pro add-on is designed to save users time and effort when rigging characters in Blender. In combination, we use yet another blender add-on Voxel Heat Diffuse Skinning that provides a method for skinning characters in a more physically accurate way, by simulating heat diffusion on the model's surface. It works by using a "heat source" to propagate heat across the surface of the model, which in turn deforms the mesh in a way that mimics the way real-world muscles and skin would move and stretch. This results in more realistic skin deformation, especially in areas such as the armpits and elbows where traditional skinning methods can produce unwanted results. The add-on includes various features to simplify the process, such as the ability to paint heat sources directly onto the model and to adjust the amount of heat diffusion. It also supports weight painting and vertex groups, so that users can fine-tune the deformation in specific areas of the model (Blender Market, LLC 2023b.). Utilizing retopologized meshes, these two programs produce an impressive rig and skinning result.

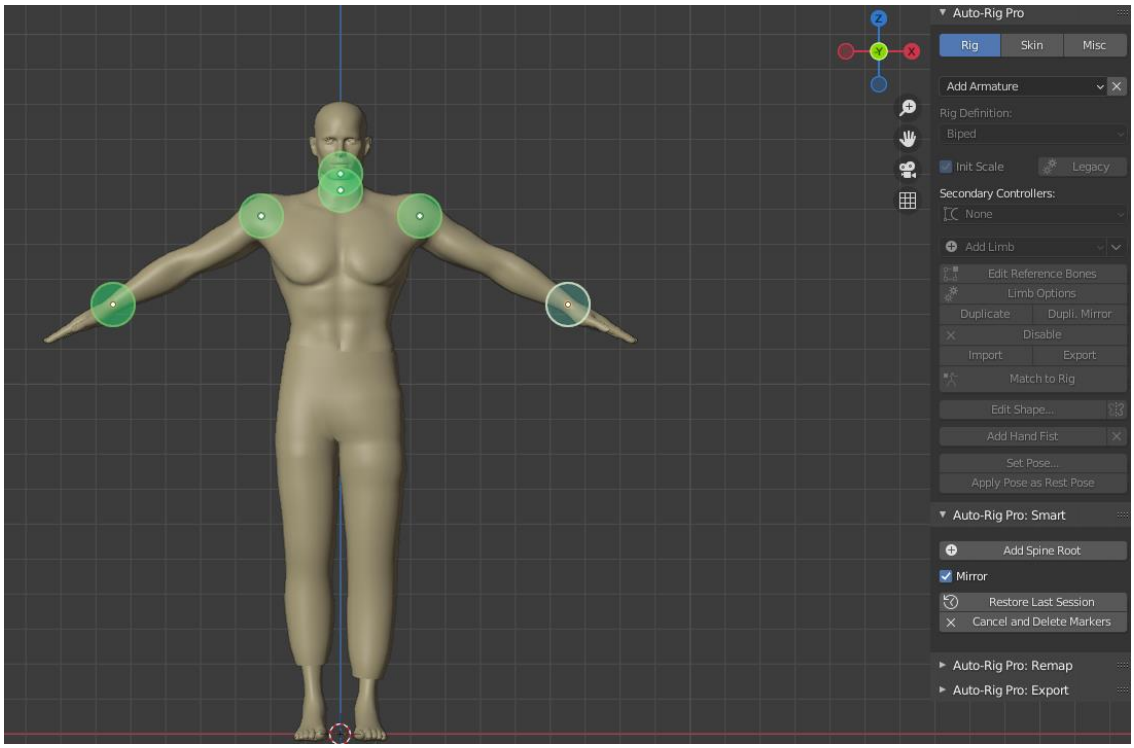


FIGURE 43. Rigging fast with Rig fast with the Smart feature of the auto-rig pro.

However, the goal of the Tuonela demo is also to create control rigs for all the characters inside Unreal Engine 5. This is done using the enhanced rigging system called the control rig that engine version 5 has to offer (FIGURE 44). It is a node-based system that allows you to create complex animation setups using a visual scripting interface. This allows for more flexibility and control over your animations, and can also be used for automating repetitive tasks, such as footstep generation (Unreal Engine 2004-2023ae.). A lot of the animations that we have been using in Tuonela for the player character and non-playable characters are purchased from the Epic Games marketplace and sometimes even when trying to retarget them as closely as possible they still will look a bit off and need adjustments in the end. However, with improved animation tools and rigging, we can hopefully do some custom animations as well as fix or improve animations that we have purchased elsewhere.

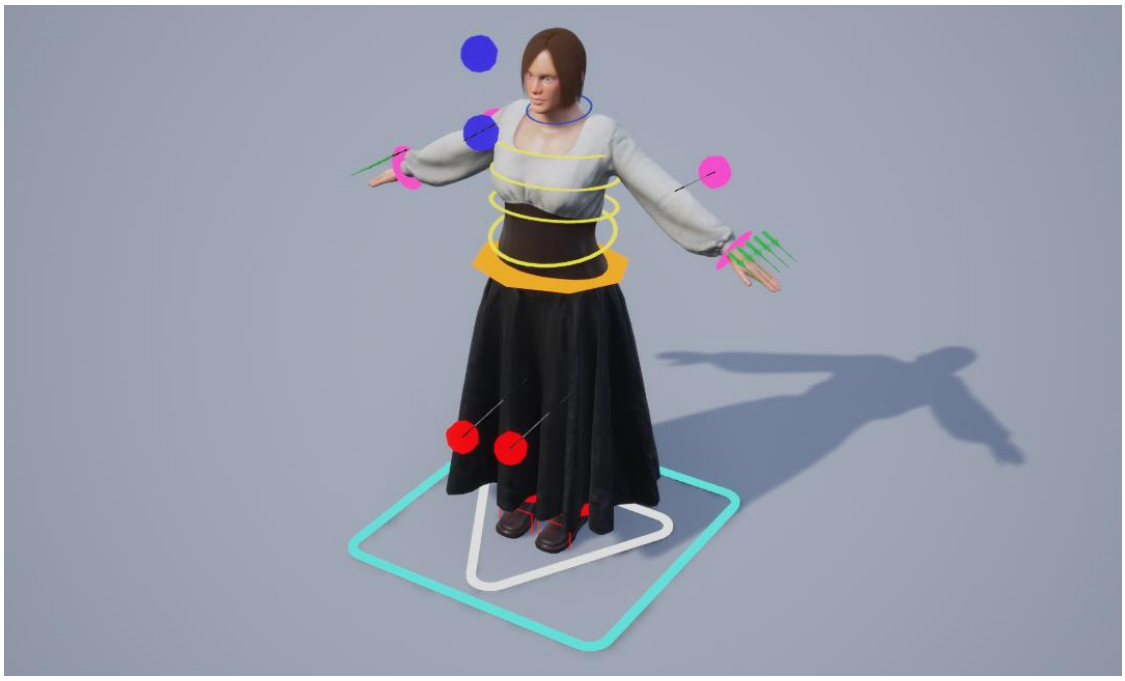


FIGURE 44. Control rig made in Unreal Engine 5 that can be used for animating inside the engine.

Marmoset Toolbag is real-time 3D rendering software that can embed lighting and other effects into textures for use in game engines (Marmoset N/A.). When baking textures from large poly models to small poly models, you must create a high-poly model with all the desired details. This is done by creating a high-poly model that will be included in the final game or application. Therefore, our clothing from Marvelous Designers and other top-quality models are large poly models with a large polygon count and our retopology is the low poly model to bake textures into. The small poly models need to be UV mapped before they can be used for baking. The high poly model is used as the source model and the low poly model as the target model for the baking process. UV mapping is the process of projecting a 3D model's geometry onto a 2D plane so that a 2D texture can be applied to the 3D model. The UV map is essentially a representation of the 3D model's surface, flattened out into a 2D image (Blender, 2023.). This allows textures to be painted or applied to the model in 2D image editing software and then mapped back onto the 3D model. There are a variety of UV mapping techniques in Blender but for our purposes, we chose unwrapping which involves creating seams for the character before unwrapping can be done. It's imperative to note that the high-poly model should be well-sculpted and have enough detail for the desired results. I consider Marmoset to be one of the most effective baking tools. This is because the program itself offers so much more control over the baking process than others with superb results.

Adobe Substance Painter is used for texturing. 3D low poly model is imported into Substance Painter by Creating a new project: Once your model is imported, create a new project. The textures that were baked in the Marmoset toolbag are added to the model in the texture settings. Once you have a base texture applied, you can start painting your model using the various brushes and tools available in Substance Painter (FIGURE 45). You can also add multiple layers of paint to create more complex textures. We won't delve into much detail about how to texture characters according to the game style. This is because there are many different styles and ways to texture, and our goal is to strive for realism. Once textures are finished, they will be exported as Unreal Engine 4 SSS (Packed) or Unreal Engine 4 (Packed) materials. Most materials are exported as Unreal Engine 4 SSS. However, if there is a need for an opacity map, such as hair textures, then Unreal Engine 4 is used since it supports exporting opacity channels. Texture sizes and file types can be configured in the export settings. Export creates 3 different texture images from the base color, normal map, and multi-texture image that contains occlusion, roughness, and metallic maps of the textures. These are the textures that can be directly used in Unreal Engines materials.

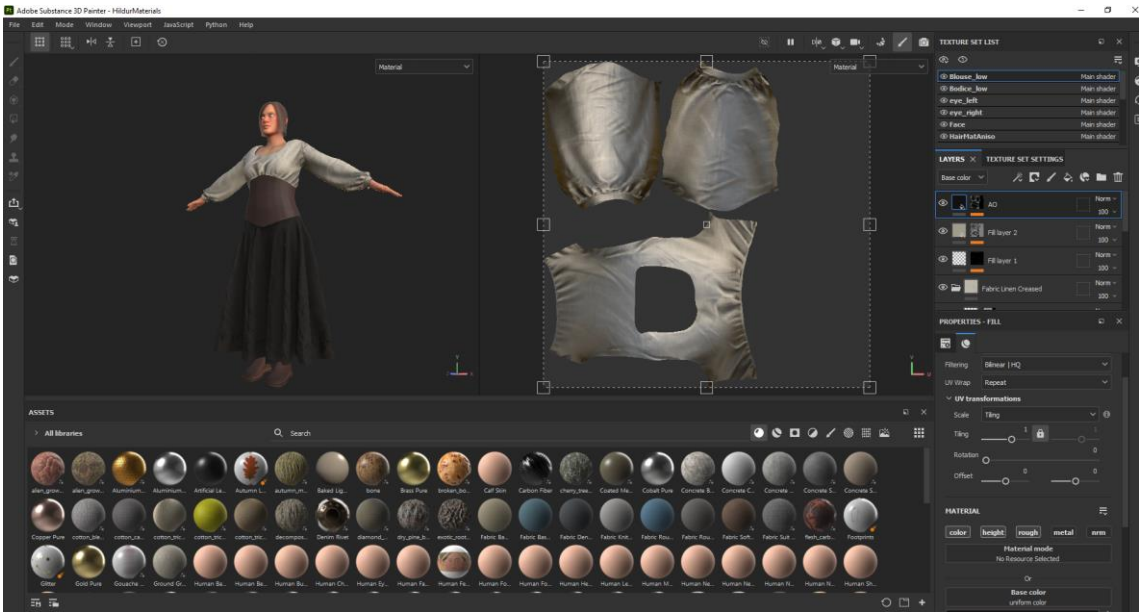


FIGURE 45. Adobe substance painter texturing view.

Once character creation in Blender is complete and the needed textures for the character are created it is saved as an FBX file. However, since Auto-Rig Pro is used for character rigging when exporting there is a special option that comes with the Auto-Rig Add-on to save the character as an auto-rig pro fbx file. Exporting a 3D model with Auto-Rig Pro to Unreal Engine involves a few steps that are helpful to take when exporting. First, the armature needs to be selected for auto rig pro export to work. Using the Unreal Engine humanoid rig makes working with the rig in the engine

easier. Model export settings can be adjusted to use the Unreal Engine humanoid rig. Other options include adding IK bones, animating IK bones, renaming bones for an unreal engine, using a mannequin axis, and root motion. If there are animations created in Blender, they can also be exported. In the misc section, smoothing groups are changed from normal to face since if left as normal, Unreal Engine will throw a warning message. There are many different settings and options that can be used when exporting models and animations, and you may want to experiment with different settings to find what works best for you.

We will not go into much detail about how the environmental art for the demo is done in this thesis. This is because our environmental artist is not the author of this thesis. Character modeling can be applied to environment assets as well, as creating 3D models, doing UV mapping for texturing, and exporting textures are similar procedures. Mostly there is no need for rigging practices for environmental art since objects just need to be static meshes. Note that environmental artists use two different 3D modeling programs to create their custom assets, namely Zbrush and Blender. In addition, the same programs are used for baking and texturing as for character baking and texturing.

8 OPTIMIZATION

Optimization of the demo is one of the most critical steps in the development process. This ensures that the demo performs well during gameplay and that the audience has a pleasant experience when interacting with it. Our aim is to make a game that can be played on a wide range of devices and can run on a wide variety of platforms so that it can be played by as many people as possible.

8.1 Asset Optimization

Some of the optimizations were done by taking advantage of Unreal Engine 5 enhanced features such as --nanite to improve game performance but traditional methods such as baking high-level details from large poly count models to retopology 3D models. The beloved figure is the starting point of our environment optimization process. We try to actively develop while keeping performance in mind. However, since both developers have been learning about the Unreal Engine while working on this demo there have been some teachable moments in terms of optimization. As can be seen, GPU usage is 53.95 ms, and frames are 53.65 ms, which is way too high for optimal performance (FIGURE 46).

FIGURE 46. Screenshot of the current GPU and Frame cost.

Some of the steps that were taken in terms of environmental optimization were to adjust LOD bias for less significant textures so that GPU memory would be saved. Due to load bias, the texture loading process is limited in terms of image resolution. For example, if there is an image that is 2048 in resolution, the LOD bias set to 1 would stop texture loading at a resolution of 1024. (Unreal Engine 2004-2022c.). GPU memory was also saved by turning off shadows for small assets in the environment, such as grass, bushes, and other foliage. In terms of trees, which are the most prominent feature of our environment, the setting "Force Volumetric" was turned off since it is an efficient method to save GPU power while also enhancing the overall quality of our environment. (Unreal Engine 2004-2022al.). It was discovered that particle systems had a large quad overdraw of 10 as shown in the figure below (FIGURE 47). We investigated the valley area where there is a lot of fog as particles. Therefore, our environment artist began learning about how LODs could be used to accommodate particle systems.

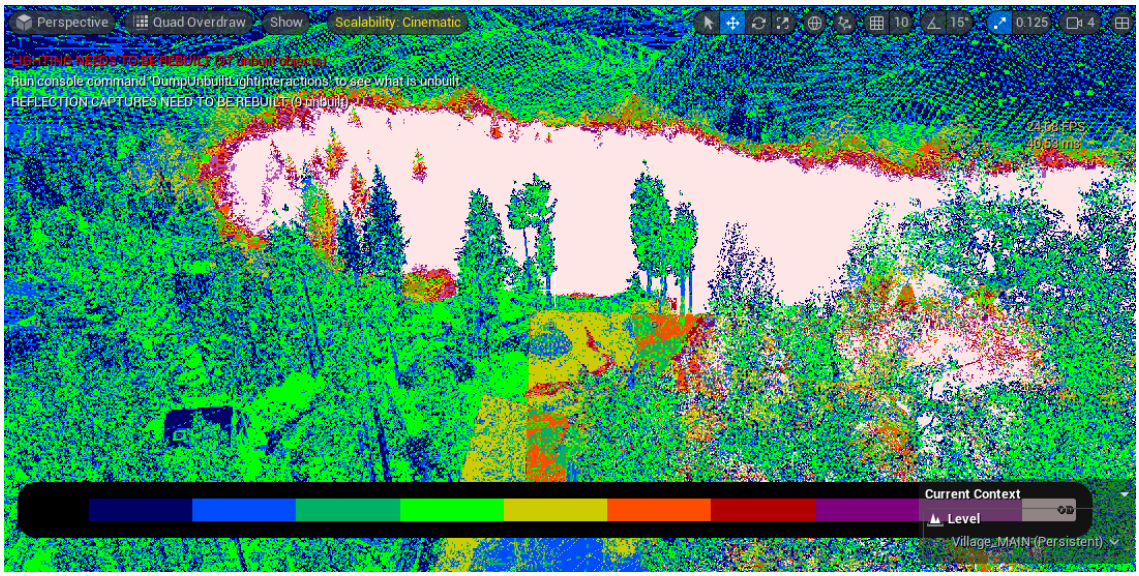


FIGURE 47. Tuonela Quad Overdraw Pictured.

Unfortunately, the demo had some problems with Lumen and how it affected the performance of the game. This was very unfortunate considering that one of the most anticipated features of Unreal Engine 5 was Lumen support in the demo. It is obvious that the price of a Lumen as seen in the figure below is too high (FIGURE 48). Since this is not something that the Unreal Engine documentation tackles, we turned to the Unreal development community to find out if other developers had faced the same problems. Luckily, we did find a thread that was made by one of the team members who is actively working on the Lumen system with Unreal Engine 5. The thing that came up in many

discussions was that the Nanite system does not yet support ray-traced shadows and it is recommended to still use virtual shadow maps. So, measures were taken to turn off the shadow system from tray racing from the Lumen system and it significantly improved performance and lowered the GPU cost. Hopefully Unreal will at some point correct these issues in future releases so that Lumen can be used to its full potential in the future. We still will keep Lumen active and hopefully can take advantage of the reflection system that it offers.

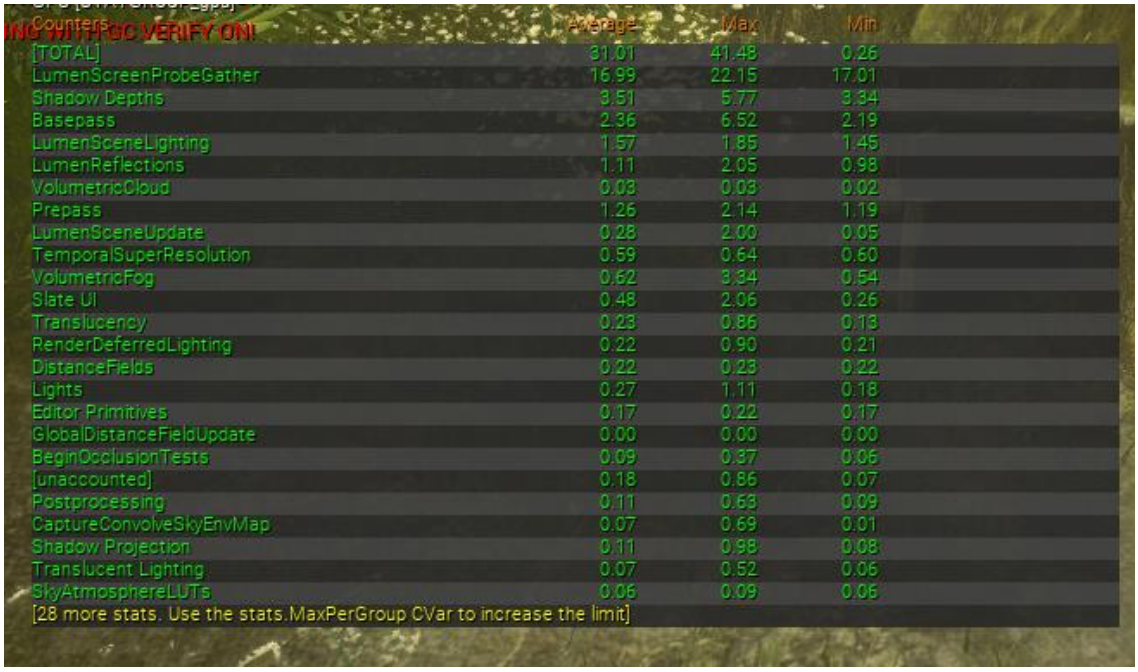


FIGURE 48. Screenshot of Lumen cost

Optimizing characters for Unreal Engine involves several steps, including reducing the polygon count, utilizing textures efficiently, and applying LODs (level of detail) to decrease the amount of detail in the character as it gets further away from the camera. One way to reduce the polygon count is to use Blender to reduce the geometry of the character. This can be accomplished by removing unnecessary polygons or by implementing a technique called retopology to generate a revised, lower-polygon version of the character. Another way to optimize characters is to apply textures efficiently. This can be done by creating smaller textures, compressing textures, or applying texture atlases to group multiple textures together (Unreal Engine 2004-2023y.) We have already talked about using LODs (level of detail) to decrease the amount of detail in environment assets and the same can be added to the character. It works similarly by creating multiple versions of the character with varying levels of detail. Then the engine switches between them based on the distance from the camera (Unreal Engine 2004-2022c.). Unreal Engine also allows you to use a physics asset which allows you to define the physical properties of the character and use them for

physics simulation. This can help optimize characters (Unreal Engine 2004-2023p.). It is critical to note that optimizing characters is an iterative process and may require some trial and error to find the right balance between quality and performance.

8.2 Technical design optimization

Since Unreal Engine is based on the Unreal Engine code base, it is very easy to optimize code for performance and make sure it is performance-driven. It is vital to note that the copy values of big assets should not be used as a reference to these assets. It is possible to pass assets into variables and functions in two different ways besides passing them as a copied value: either by using a reference or by using a pointer. A reference is indicated by & and a pointer is indicated by *. These will affect how objects and assets are loaded into memory (Unreal Engine 2004-2023s.).

Now, these are some of the optimization strategies that we have currently been implementing for the demo. It is relevant to mention, however, that we are actively looking for other ways of improving game performance in the technical as well as asset design areas. It is one of our main priorities to ensure that the game runs smoothly for players, and that is why it will be one of our main priorities.

9 DISCUSSION

Unreal Engine 5 can be used to create visually stunning and high-performance games and interactive experiences. Nanite virtualized geometry technology and the Lumen global illumination system offer unparalleled levels of detail and realism. Improved workflow and performance enhancements also make it easier for developers to implement their ideas. Moreover, Unreal Engine 5 supports next-generation consoles and platforms allowing developers to reach a wider audience and deliver immersive experiences across multiple platforms. It is a powerful tool for developers looking to push the boundaries of what is possible with interactive media with Unreal Engine 5.

When programming with Unreal Engine, developers have the option to use either C++ or Blueprints, Unreal Engine's visual scripting system. C++ is a powerful and versatile programming language that allows low-level access to the engine and provides complete control over the game's systems. This is ideal for experienced programmers who want to create custom game mechanics, optimize performance, and create complex systems. On the other hand, Blueprints is a more accessible option for those who are less experienced with programming or who prefer a more visual approach to development. Blueprints allow developers to create game logic by connecting nodes together in a flowchart-like structure rather than writing code. This makes it easier to prototype and test ideas quickly, and it's a simple way to get started with game development.

Unreal Engine's documentation and community resources are also very robust, which makes it easy to find answers to any questions that may arise during the development process. Additionally, the engine's built-in debugging and profiling tools allow developers to quickly identify and fix performance issues. Overall, Unreal Engine provides a flexible and versatile environment for game development that can accommodate a wide range of programming skill levels and workflows. Whether a person is an experienced programmer or just getting started with game development, Unreal Engine has the tools and resources you need to bring your ideas to life.

While all that is true, it has still been a steep learning curve as an individual trying to grasp even a fraction of the potential that Unreal Engine offers. Individuals and independent developers often do the work of a larger team. For example, I have learned practices from coding game logic to learning how to use other complicated software. These include creating custom characters with complicated rigs, clothing, and animations for Unreal Engine. Once you learn the basic nuances of developing

with Unreal Engine, nothing happens in a day or even a few weeks. In terms of how our combat system is supposed to work in the final demo, I still have a lot to learn about 3D math and vector math. We have been developing for over a year just for an hour-long demo. We will keep working on Tuonela and hopefully, in the future, we can see a demo up on Steam for everyone to try.

The demo itself is not playable yet. It is missing a lot of functional UI design elements and a lot of the game mechanics are not fully developed yet and is just in the first stages of testing them in terms of high-level functionalities. In addition, there are still a lot of environmental design elements missing especially in the underground area. This is where a lot of quests and puzzle elements for quests take place. Finally, most of the characters are still in the first art pass, so they lack the quality textures and animations that will be included. From this point on, we will keep returning to the game mechanics and improving them, as well as adding both playable characters and non-playable characters, and enemies, through their second or third passes to fit them into the game aesthetics. In addition, a lot of characters and enemies have not even been created yet. Lastly, there are still a lot of environmental elements that need to be added to the game level. That the quests can take place and that there will not be unfinished areas in the demo. In addition, we will need to do some investigating and educate ourselves better with UI design and/or perhaps find a person who could create custom elements and understands better game UI design. This is because right now we mostly use any default design that Unreal Engine has to offer.

REFERENCES

Blender 2023. Editors, Blender Manual 3.4, UV Editor, UVs Explained. Search date 17.01.2023. https://docs.blender.org/manual/en/latest/editors/uv/introduction.html

Blender Market, LLC 2023a. Blender Market, Auto-Rig Pro. Search Date 17.01.2023. https://blender-market.com/products/auto-rig-pro?ref=46

Blender Market, LLC 2023b. Blender Market, Voxel Herat Diffuse Skinning. Search Date 17.01.2023. https://blendermarket.com/products/voxel-heat-diffuse-skinning

CLO Virtual Fashion Inc. 2023. Product, Features. Search Date 15.01.2023. https://www.marvelousdesigner.com/product/keyfeature

Crytek GmbH 2022. Features. Search date 31.12.2022. https://www.cryengine.com/features

Epic Games 2004-2023a. AActor::TakeDamage. Search date 26.01.2023. https://docs.unrealengine.com/4.27/en-US/API/Runtime/Engine/GameFramework/AActor/TakeDamage/

Epic Games 2004-2023b. BoxTraceByChannel. Search Date 22.01.2023. https://docs.unrealengine.com/5.0/en-US/BlueprintAPI/Collision/BoxTraceByChannel/

Epic Games 2004-2023c. Creating and Using LODs. Search date 03.01.2023. https://docs.unrealengine.com/5.0/en-US/creating-and-using-lods-in-unreal-engine/

Epic Games. 2004-2022d. Creating a New Project. Search date 19.11.2022.
https://docs.unrealengine.com/5.1/en-US/creating-a-new-project-in-unreal-engine/

Epic Games 2004-2023e. Enhanced Input. Search date 02.01.2023.
https://docs.unrealengine.com/5.0/en-US/enhanced-input-in-unreal-engine/

Epic Games. 2004-2022f. Epic Content License Agreement. Search date 13.11.2022.
https://www.unrealengine.com/en-US/eula/content

Epic Games 2004-2023g. FCollisionQueryParams. Search Date. 04.01.2023.
https://docs.unrealengine.com/5.1/en-US/API/Runtime/Engine/FCollisionQueryParams/

Epic Games 2004-2023h. FHitResult. Search Date. 04.01.2023.
https://docs.unrealengine.com/4.26/en-US/API/Runtime/Engine/Engine/FHitResult/

Epic Games 2004-2023i. FScriptMapHelper, Functions. Search Date 20.01.2023.
https://docs.unrealengine.com/5.1/en-US/API/Runtime/CoreUObject/UObject/FScriptMapHelper/

Epic Games. 2004-2022j. Hardware and software Specifications. Search date 19.11.2022.
https://docs.unrealengine.com/5.1/en-US/hardware-and-software-specifications-for-unreal-engine/

Epic Games 2004-2022k. Importing Assets Directly. Search date. 30.12.2022.
https://docs.unrealengine.com/5.0/en-US/importing-assets-directly-into-unreal-engine/

Epic Games. 2004-2022l. Installing Unreal Engine. Search date 19.11.2022.
https://docs.unrealengine.com/5.1/en-US/installing-unreal-engine/

Epic Games 2004-2023m. Interfaces. Search Date 22.01.2023.

https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/GameplayArchitecture/Interfaces/

Epic Games 2004-2022n. Lumen Global Illumination and Reflections. Search date 25.09.2022. https://docs.unrealengine.com/5.0/en-US/lumen-global-illumination-and-reflections-in-unreal-engine/

Epic Game 2004-2023o. Nanite Virtualized Geometry. Search date. 02.01.2023 https://docs.unrealengine.com/5.0/en-US/nanite-virtualized-geometry-in-unreal-engine/

Epic Games 2004-2023p. Physics Asset Editor. Search Date 15.01.2023 https://docs.unrealengine.com/5.0/en-US/physics-asset-editor-in-unreal-engine/

Epic Games. 2022q. Quixel. Search date 29.9.2022.
https://quixel.com/megascans

Epic Games 2022r. Quixel. Plans & Pricing, What is the difference between licenses? Search date 29.9.2022.
https://help.quixel.com/hc/en-us/articles/115000599329-What-is-the-difference-between-the-licenses-

Epic Games 2004-2023s. Referencing Assets. Search Date. 08.01.2023. https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/ProgrammingWithCPP/Assets/ReferencingAssets/

Epic Games 2004-2022t. Releasing Your Project. Search date 30.12.2022. https://docs.unrealengine.com/5.0/en-US/preparing-unreal-engine-projects-for-release/

Epic Games 2004-2023u. Super. Search Date 15.01.2023 https://docs.unrealengine.com/5.1/en-US/API/Runtime/CoreUObject/UObject/UObject/Super/

Epic Games 2004-2023v. TArray::Add. Search Date 15.01.2023 https://docs.unrealengine.com/4.27/en-US/API/Runtime/Core/Containers/TArray/Add/1/

Epic Games 2004-2023w. TArray:Arrays in Unreal Engine. Search Date 08.01.2023 https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/ProgrammingWithCPP/UnrealArchitecture/TArrays/

Epic Games 2004-2023x. TArray, Functions, Remove At. Search Date 19.01.2023 https://docs.unrealengine.com/4.27/en-US/API/Runtime/Core/Containers/TArray/

Epic Games 2004-2023y. Texture Format Support and Settings. Search Date 15.01.2023. https://docs.unrealengine.com/5.0/en-US/texture-format-support-and-settings-in-unreal-engine/

Epic Games 2004-2022z. Tools and Editors. Search date 11.12.2022. https://docs.unrealengine.com/5.1/en-US/tools-and-editors-in-unreal-engine/

Epic Games 2004-2023aa. UBoxComponent. Search Date. 05.01.2023. https://docs.unrealengine.com/5.1/en-US/API/Runtime/Engine/Components/UBoxComponent/

Epic Games 2004-2023ab. UFunctions, Function Specifiers. Search Date 15.01.2023. https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/GameplayArchitecture/Functions/

Epic Games 2004-2023ac. UInputMappingContext. Search Date 12.01.2023. https://docs.unrealengine.com/4.26/en-US/API/Plugins/EnhancedInput/UInputMappingContext/

Epic Games. 2004-2022ad. Unreal Editor Interface. Search date. 26.11.2022. https://docs.unrealengine.com/5.1/en-US/unreal-editor-interface/

Epic Games 2004-2023ae. Unreal Engine 5.0 Documentation, Control Rig. Search Date 17.01.2023. https://docs.unrealengine.com/5.0/en-US/control-rig-in-unreal-engine/

Epic Games 2004-2022af. Unreal Engine 5.0 Migration Guide. Search date 27.09.2022. https://docs.unrealengine.com/5.0/en-US/unreal-engine-5-migration-guide/

Epic Games. 2004-2022ag. Unreal Engine 5.0 Release Notes. Search date 25.09.2022. https://docs.unrealengine.com/5.0/en-US/unreal-engine-5.0-release-notes/

Epic Games 2004-2023ah. UPrimitiveComponent. Search Date 05.01.2023. https://docs.unrealengine.com/5.1/en-US/API/Runtime/Engine/Components/UPrimitiveComponent/

Epic Games. 2004-2022ai. Unreal Engine. Unreal Engine. Search date 27.10.2022. https://www.unrealengine.com/en-US/

Epic Games. 2004-2022aj. Unreal Engine. Unreal Engine End User License Agreement. Search date 29.10.2022. https://www.unrealengine.com/en-US/eula/unreal

Epic Games 2004-2023ak. UWortld::LineTraceSingleByChannel Search Date 04.01.2023. https://docs.unrealengine.com/5.0/en-US/API/Runtime/Engine/Engine/UWorld/LineTraceSingleByChannel/

Epic Games 2004-2023al. Volumetric Lightmaps: Volumetric Lightmap versus Indirect Lightmap Cache. Search Date 03.01.2023. https://docs.unrealengine.com/4.27/en-US/RenderingAndGraphics/Lightmass/VolumetricLightmaps/

Juan Linietsky, Ariel Manzuri, and contributors. 2007-2022. Features. Search date 31.12.2022. https://godotengine.org/features

Marmoset N/A. Toolbar, Baking. Search Date 17.01.2023.
https://marmoset.co/toolbag/baking/

Orange Turbine 2022. Product, RetopoFlow - Retopology Toolkit for Blender. Search Date 17.01,2023.
https://orangeturbine.com/downloads/retopoflow

Unity Technologies 2022. Products & Features | Unity. Search date. 31.12.2022. https://unity.com/products-features