FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Graph editor in virtual reality
Student:	Ivan Menshikov
Supervisor:	Ing. Petr Pauš, Ph.D.
Study program:	Informatics
Branch / specialization:	Web and Software Engineering, specialization Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2023/2024

Instructions

Cílem práce je navrhnout a implementovat prototyp editoru a zobrazovače grafů ve virtuální realitě (VR) pomocí Unreal Enginu (UE). Prototyp by měl umět načíst a uložit grafy z externího souboru, měl by mít možnost přidávat, mazat a editovat uzly a hrany grafu. Prototyp optimalizujte pro rozsáhlejší grafy (tisíce uzlů).

1. Analyzujte možnosti využití VR v Unreal Enginu.
2. Anazylujte, jak implementovat grafy pomocí UE.
3. Pomocí technik softwarového inženýrství navrhněte prototyp aplikace včetně uživatelského rozhraní.
4. Prototyp implementujte.
5. Proveďte vhodné testy.

Electronically approved by Ing. Michal Valenta, Ph.D. on 7 October 2022 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Graph editor in virtual reality

Ivan Menshikov

Department of Software Engineering
Supervisor: Ing. Petr Pauš, Ph.D.

January 5, 2023

Citation of this thesis: Menshikov Ivan. *Graph editor in virtual reality.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Contents

List of Figures

List of Tables

List of Code Listings

I would like to thank my supervisor, Ing. Petr Pauš, Ph.D., for the guidance, encouragement, and advice he has provided. I am also sincerely grateful to my family and friends for their support.

Declaration

In Prague on January 5, 2023 . . . .

Abstract

Virtual reality technology can be used to represent and solve problems involving relationships between objects in a more immersive and interactive way. This thesis presents the design and development of a standalone VR software product that visualizes graphs and provides tools for performing operations on them. The product, which targets the Oculus Quest 2 headset and was built using Unreal Engine 4, allows users to examine, alter and interact with the graph in a more intuitive manner. The system includes a procedural generation method for rendering large graphs efficiently and various optimizations for high performance. A system of tools and a user interface have also been implemented. Testing was conducted to ensure the product's functionality and performance, and it was demonstrated to be a practical tool for visualizing and working with graphs in VR with potential applications in education and research.

Keywords graphical application, graph, virtual reality, Oculus Quest 2, Unreal Engine 4, procedural mesh generation, user interface

Abstrakt

Technologii virtuální reality lze využít k zobrazení a řešení problémů týkajících se vztahů mezi objekty, a to více pohlcujícím a interaktivním způsobem. Tato práce představuje návrh a vývoj samostatného softwarového produktu pro VR, který vizualizuje grafy a poskytuje nástroje pro provádění operací s nimi. Produkt, který je určen pro náhlavní soupravu Oculus Quest 2 a byl vytvořen pomocí Unreal Engine 4, umožňuje uživatelům zkoumat, měnit a interagovat s grafem intuitivnějším způsobem. Systém obsahuje procedurální metodu generování pro efektivní vykreslování velkých grafů a různé optimalizace pro vysoký výkon. Implementován byl také systém nástrojů a uživatelské rozhraní. Bylo provedeno testování, které zajistilo funkčnost a výkonnost produktu, a ukázalo se, že se jedná o praktický nástroj pro vizualizaci a práci s grafy ve VR s potenciálním využitím ve vzdělávání a výzkumu.

Klíčová slova grafická aplikace, graf, virtuální realita, Oculus Quest 2, Unreal Engine 4, procedurální generování, uživatelské rozhraní

List of abbreviations

2D	Two-dimensional
3D	Three-dimensional
API	Application programming interface
AR	Augmented reality
CPU	Central processing unit
FPS	Frames per second
GPU	Graphics processing unit
OS	Operating system
RAM	Random-access memory
UI	User interface
UX	User experience
VR	Virtual reality
XR	Extended reality

Introduction

A graph can be both a visualization of an idea that comes to you and an expression of this idea's essence. When dealing with a particular type of problem, we sometimes need to describe or represent some sort of relationship between objects. By treating these objects as vertices and relationships between them as edges, we can solve a wide variety of problems using the entire rich arsenal of graph theory algorithms: finding a path from one object to another, identifying connected components, calculating shortest paths, and much more. Today, discrete mathematics and graph theory are applied to our everyday lives and used in a lot of important and interesting research and corporate applications: from network security to navigation applications, from scheduling or delivery route problems to vaccine development, from digital image processing to kidney donor matching, and far more.

During the solving of problems related to graph theory, we tend to draw vertices as points on paper and connect them with lines denoting relations. We do, however, live in a 3D world, and there are a huge number of cases where it is much clearer and more convenient to represent a corresponding graph in 3D rather than in 2D. With the development of technology and the appearance of computers in our lives, we have the ability to visualize graphs using various websites and applications, which are mainly focused on 2D rendering. Compared to drawing on paper, these solutions provide the ability to display much larger graphs as well as, not least, the visualization of graph theory algorithms, which can be very helpful, for example, in education. Nevertheless, regarding 3D visualization, such tools cannot boast the convenience of working with graphs as well as the clarity of algorithms due to the fact that, in this case, all of our perception is limited to the monitor, and all manipulation and interaction with objects is limited to the keyboard and mouse.

From my point of view, a much more practical and convenient representation of graphs in 3D space can be achieved with the use of VR technology. It has been available for decades, but it is only now that products based on it have become available and are in demand by ordinary users. The main feature of VR is immersion, which means being immersed in a virtual 3D space and being able to interact with some of its elements. This solution will allow users to see graphs more clearly and conveniently and will also help them engage more deeply with the problem to be solved. With more features and proposed tools, the same can be applied to the visualization of graph theory algorithms.

Speaking of the above-mentioned education, VR has important advantages as a learning tool, which can also be applied in the context of graph theory learning. The first is focus. In a virtual world, users are not distracted by anything and are completely focused on exploring the appropriate objects or events. Next, we have illustrativeness. VR allows you to explore objects that are very difficult or impossible to see in real life, such as other planets or black holes, how brain neurons work, the life of microorganisms, and much more. Participation and engagement are also important to me as a student. Thanks to immersiveness, a student can not only read about events in, for example, a history book but also become a participant in those events. With the help of VR and gamification, it is possible to make learning about laws, theorems, and subjects more entertaining.

After discussion with the supervisor, Ing. Petr Pauš, Ph.D., the practical result of this work can be demonstrated and used during the teaching of the Algorithms and Graphs course at the Faculty of Information Technology (CTU).

This thesis' structure is based on the software development life cycle process. The first to be examined and analyzed will be general VR application development techniques, concepts, and tools needed to complete the assignment, as well as the assignment itself. Assignment analysis will consist of gathering and analyzing specifications as well as identifying use cases. This is followed by the design and architectural engineering of the resulting application, where each of its necessary parts will be examined in detail. Then the implementation part is presented. Finally, the Results chapter describes both the results of testing the created system and the conclusions of this work, along with ideas and plans for the future.

Goals

The main goal of this thesis is to apply the principles of software engineering to design, develop, and test a complete standalone software product that can visualize graphs in VR and offers tools for performing certain operations on such graphs. The resulting system should be able to accept user requests for operations on a graph and, based on those requests, render and alter a relevant graph in 3D space with the use of VR technology. Designing an alternative way to represent graphs and interact with them is the essence of this goal.

One of the goals is to define a possible representation of a graph in 3D space. This graph representation will allow users to change it and interact with it in a more understandable and convenient way. It will also be designed in a way that makes it easy to save and retrieve from an external file. The proposed representation of graphs stored in an external file will also be presented in a human-readable form for users, which will also give them the ability to read and edit the file outside the resulting application.

The next goal is to design a system that will serve as an environment to visualize the proposed representation of graphs in VR and provide users with tools to work with them. To achieve this, aspects of the development of VR applications will be considered and analyzed. These considerations include user comfort, content optimization, and the limitations of VR platforms.

Another goal is to build the system so that it is capable of rendering and visualizing huge graphs while simultaneously accepting and processing user queries without interruption. In addition, a number of optimizations will be presented for all of the parts of the entire application to achieve high system performance. Consequently, all these improvements will help the system reach the level of performance it needs to render and process a number of very large graphs.

Chapter 1

VR application development

This chapter examines the use of VR technology in the development of applications, which is capable of simulating vision and creating a 3D environment in which the user is immersed while browsing or experiencing it. The first section discusses the key differences between the development of classical desktop applications and the development of VR applications, as well as the challenges that developers have to deal with. When it comes to developing video games or graphical applications in general, a game engine can be a great framework to use as a basis. One of the most popular game engines, Unreal Engine, is described in the second section. In the third section, we talk about OpenXR, a standard for making XR applications, and how it works with Unreal Engine.

1.1 Development and design challenges

The design of a VR application is very similar to the design of a graphical desktop application because, in both cases, we are dealing with an interactive 3D experience. However, there are several key differences that create serious challenges and problems that need to be overcome during the development and design of VR applications. Due to immersiveness, which is the most important feature of VR, developers need to pay special attention to the effect of presence, non-linear narrative, non-nauseating movement, and graphic optimization.

1.1.1 Camera perspective

The *camera perspective* [1] in a 3D world is the first thing a developer will face when designing a VR application. It should be determined before the start of development since its subsequent change can substantially alter both the gameplay and the overall concept of the game being developed. If we develop a 3D game for a classic desktop environment, we have quite a lot of freedom in deciding where to place a camera that allows the user to see a virtual world. There are choices for first-, second-, and third-person views, as well as top-down or isometric perspectives. On the other hand, VR technology aims to gain a maximum sense of presence, which is achieved with a headset device, an example of which is shown in Figure 1.1.

The headset is a standalone device that provides the ability to view and manipulate a virtual world. It shows a separate image to each eye, which our brain combines, giving us the impression that we are in the same 3D world as our real one. Some of the devices are equipped with camera and sensor systems to detect a body in space, track hands, handle mixed reality operations, etc. There are many VR headsets on the market, and the main manufacturers are HTC, Microsoft, Oculus (acquired by Meta), and Sony.

Figure 1.1 Oculus Quest 2 headset [2]

As a result, the first-person view is favored for use in VR applications. This viewpoint is the one that most successfully preserves the sensation of being in a virtual world. It also works well for fast-paced games that rely on quick thinking, and it helps users experience unusual situations and events by putting them in the shoes of the characters who must go through these events.

1.1.2 Performance

The next issue that could arise during the development of a VR application is *performance optimization*. Due to the resource limitations of wearable devices and the high computational cost of real-time rendering, VR applications often face performance bottlenecks, and performance optimization plays an important role in VR software development.

There are two main categories of VR performance concerns: CPU and GPU issues. The simulation logic, state management, and creation of the visual scene are typically handled by the CPU. The GPU usually takes part in texture sampling and shading for the meshes in scenes. It is critical to identify whether CPU or GPU load is to blame for a performance issue and to optimize code accordingly. In general, the optimization of VR applications can be very different from that of traditional software, as VR involves more elements to render in real-time, such as 3D graphics, animations, etc. Specifically, with VR, every frame must be drawn twice, once for each eye. It means that every draw call is issued twice, every mesh is drawn twice, and every texture is bound twice.

As a result, this kind of problem is one of the most complex and often requires a lot of time and effort to solve. Unoptimized VR applications cause visual and orientational discomfiture for many users, which subsequently leads to poor reviews and ratings.

1.1.3 Locomotion

The other problem is to find a solution or concept of movement in a virtual world, which is also called the *"VR locomotion problem."* [3] When developing a VR application, game designers must solve the problem of moving through large virtual spaces that are larger than the space available in the real world.

The Cybershoes or omnidirectional treadmills like Virtuix Omni, Kat Walk, and Infinadeck are some of the solutions. Those are systems that keep the user in a spot while allowing him to walk through infinitely sized VR environments. Such a system requires additional devices, which are currently quite expensive and also take up available space. However, it removes any spatial limitations, enabling free VR locomotion of arbitrary size or shape, and, speaking of the immersiveness mentioned above, those are the best solutions to maintain presence in VR.

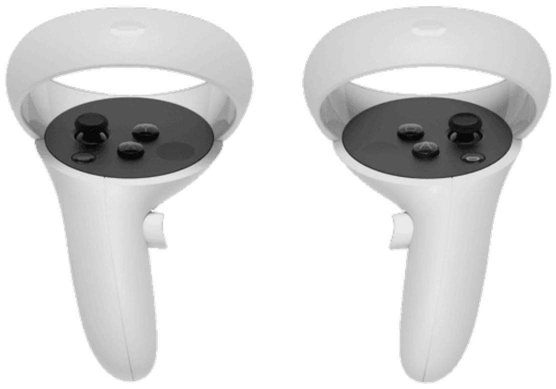


Figure 1.2 Oculus Quest 2 controllers [2]

There are also numerous software solutions to this problem that make use of controllers, an example of which is illustrated in Figure 1.2. Controllers are often already bundled with a headset, so these solutions do not require any additional devices. In classic desktop games, the user's movement in space is traditionally realized via WASD or arrow keyboard keys. A similar experience can be achieved with the use of thumbsticks on controllers, which is one of the simplest solutions from an implementation point of view. However, this solution is commonly used as an optional one due to the fact that it can cause discomfiture or nausea among some users. For this reason, there is another type of movement that aims at the idea of teleportation, which allows the user to move in a 3D scene without a sense of motion because of the instantaneous movement. Despite the fact that it can quite significantly interfere with the sense of presence, it is one of the most popular solutions.

1.1.4 Control limitations

Another thing that differs in the development of VR applications is that users have to use *controls only available on controllers*, which are much smaller in number compared to normal keyboards. Because of this, if an application needs to use more complex and sophisticated manipulation elements, developers need to model and adjust them appropriately so that they can be used to manipulate objects in a 3D world using VR controllers.

An example of a good control model is a text input request. When the user tries to enter text in a VR application, a virtual keyboard should appear that they can interact with by pointing controllers at its buttons. In this case, the use of a physical keyboard is a bad choice because users have to remove and put on their headsets every time they want to see a keyboard.

1.2 Unreal engine

A game engine is a software development environment that can be used as a basis for creating games or graphical applications in general. It is a great solution that can reduce the time and complexity of development and focus on the idea itself. One of the most important factors in game engines is the interoperability between the different game systems available. In essence, they provide developers with a number of components and tools that allow them to create their games faster and with less effort. The degree to which a tool assists developers varies, depending on the engine. What will also be different is the compatibility of an engine with various genres of games. For example, an engine that is perfect for creating first-person shooter games may not be optimal or suitable for creating strategy games at all. However, the most popular modern game engines provide developers with almost all the tools they need and are suitable for developing games of different genres. The main features that most game engines provide are the physics engine, animation, 2D and 3D graphics rendering, collision detection, artificial intelligence, sounds, networking, memory management, and user interface creation.

1.2.1 Brief overview

Unreal Engine [4] is a game engine developed and maintained by Epic Games. The first game based on this engine was the first-person shooter Unreal, released in 1998. Although originally designed for first-person shooters, it has been used in various genres of games over the years and has been adopted by other industries, notably the film and television industries. The latest version, Unreal Engine 5, was shipped in April 2022. One of the main goals of its creation was to achieve exceptional photorealism *comparable to real life* [5]. Since its predecessor was released in 2014, Epic Games has rejected the idea of distributing the engine on a monthly subscription model in favor of a royalty model for commercial use, meaning that Unreal Engine is completely free for non-commercial use. Today, for commercial use, Epic waives their royalty margin for games until developers have earned $1,000,000 in gross worldwide revenue under certain conditions, as defined in the *Unreal Engine End User License Agreement* [6].

The engine is written in C++ and provides a high-performance, powerful, and voluminous 2D/3D application development toolkit for the majority of commonly used operating systems and platforms, such as desktops, mobile devices, consoles, and VR. To simplify porting, the engine uses a modular system of independent components and plugins. Plugins are collections of code and data that developers can easily enable or disable on a per-project basis. Plugins can add runtime gameplay functionality, add new or modify built-in engine features, create new file types, etc. Many existing engine subsystems were designed to be extensible using plugins. The engine supports various modern rendering systems (Direct3D, OpenGL, Pixomatic), sound (EAX, OpenAL, DirectSound3D), speech recognition, voice synthesis, network modules, and a wide variety of input devices.

1.2.2 Development methods

Among the main advantages of Unreal Engine are its versatility and accessibility. It can be used by experienced developers and beginners who want to create a game for the first time. The point is that the engine provides two methods to create gameplay elements: traditional *C++* and *Blueprints*. Blueprints is the Unreal Engine's *visual scripting system* [7]. Instead of writing code line by line, developers can do everything visually: drag-and-drop nodes, set properties in the interface, and connect them, with little programming knowledge required for a beginner. The gameplay APIs and the framework classes are available for both systems and can be used individually, but if combined appropriately, they allow for a more maintainable and extensible application architecture. It is common for C++ programmers to add the base gameplay systems that level designers can then build upon or use to create their own custom gameplay elements for one level or the entire game.

Several essential aspects of the *Balancing Blueprint and C++* [8] documentation can be strongly emphasized. Development with Blueprints is usually faster because of its clarity, relative ease of use, and intuitiveness. Also, storing data inside Blueprint classes is much simpler and safer. On the other hand, C++ offers significantly faster runtime performance, broadly speaking, because executing each individual node in a Blueprint is slower than executing a line of C++ code. Also, C++ has more engine functionality exposed to it, which allows users to interact directly with the engine's source code and tools.

1.3 OpenXR standard

Initially, VR application development was quite complex due to the need to support and handle the many popular VR devices at the time. Today, thanks to XR fragmentation solutions with cross-platform standardization and API interfaces, the process of developing game projects for different XR platforms is significantly simplified. Developers do not need to use separation of concerns and write a distinct module or separate code to support a particular device. It does not even matter which VR headset or controllers you use during development, thanks to the emulation of any input mapping using interaction profiles.

1.3.1 Outline

VR support in Unreal Engine has been available since version 4 in 2012. The first supported device was the Oculus Rift DK1. Currently, the main standard for accessing and working with XR in Unreal Engine is *OpenXR* [9], developed by Khronos Group and introduced in 2019. It is also one of the most popular XR standards in general. It aims to provide developers with a convenient way to create applications with the support of VR devices, and it simplifies VR development by enabling developers to reach more platforms while reusing the same code. The standard provides an API aimed at application developers targeting VR or AR hardware, as depicted in Figure 1.3. With this standard, developers can create an immersive experience in Unreal Engine that can run on any system that supports the OpenXR APIs. The following platforms have released OpenXR runtimes and are currently supported in Unreal Engine: Steam VR, Windows Mixed Reality, Oculus VR, Samsung Gear VR, and Google Daydream.

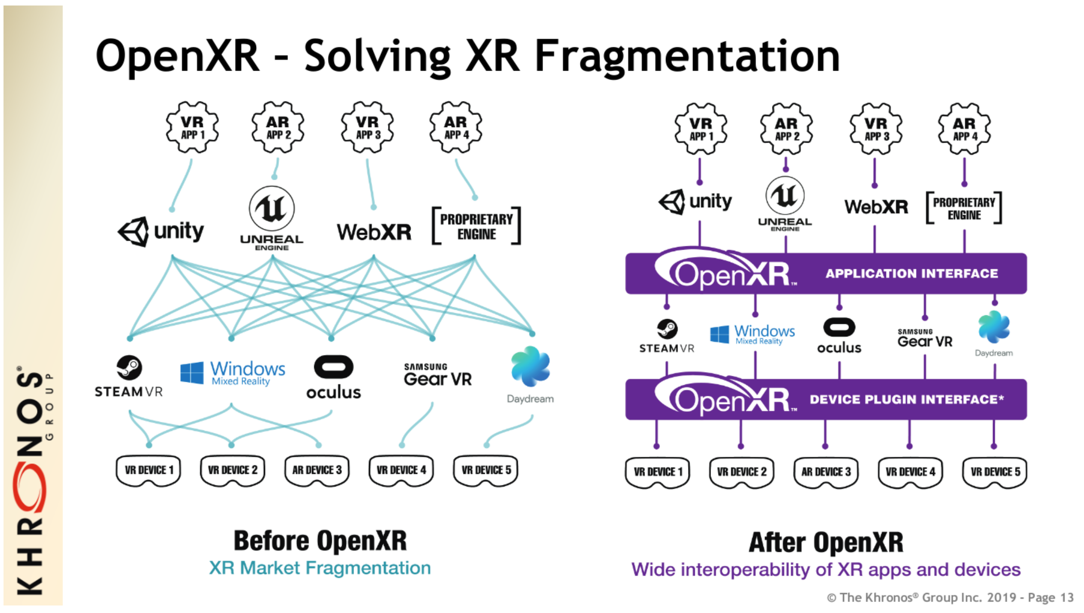


Figure 1.3 OpenXR standard for solving XR fragmentation [10]

The OpenXR runtime uses interaction profiles to support a variety of hardware controllers and provide action bindings for whichever controller is connected. When emulating action or axis controller mappings, the OpenXR runtime chooses controller bindings that closely match the user's controller. Because it provides this kind of cross-compatibility for developers, they should only deal with bindings for controllers they can test with and plan to support in the game being developed. Any bindings specified for a controller define the actions that are connected to that controller.

1.3.2 Plugins

As mentioned in Section 1.2.1, Unreal Engine uses a modular system of independent plugins and components. To start developing OpenXR projects, the *OpenXR plugin* [11] should first be enabled. This plugin supports extension plugins so that functionality can be added to OpenXR without relying on engine releases. The plugins currently available in Unreal Engine to extend the OpenXR plugin include:

- *Oculus OpenXR* — used to support Oculus headsets and their features. The plugin includes essential resources and provides additional functionality for more advanced work with Oculus devices.
- *XRVisualization* — contains additional resources that can be used for headsets, controllers, and hand mesh visualization. This plugin can recognize XR devices and, based on that, gives the right resources and shows the right 3D model for the device that is currently being used.
- *OpenXRHandTracking* — enables applications to locate the individual joints of hand tracking inputs in real-time. It also provides functionality to render hands in XR experiences and interact with virtual objects using hand joints. Data from sensors on the position of bones are returned as structures with information about each element for further processing.
- *OpenXRMsftHandInteraction* — defines a new interaction profile driven by directly tracked hands for near- and far-field interactions. This plugin is often used to recognize certain gestures. Due to the convenience of using interaction profiles, processed gestures can be treated as one of the input sources.
- *OpenXREyeTracker* — a plugin for getting eye gaze input from an eye tracker to enable eye gaze interactions. An eye tracker is a sensory device that tracks the eyes and accurately maps what the user is looking at. With this extension, an application can discover if the XR runtime has access to an eye tracker, bind the eye gaze pose to the action system, determine if the eye tracker is actively tracking users' eyes, and use the eye gaze pose as an input signal to build interactions.

Chapter 2
Graphs

This chapter introduces the concept of a graph, which is an essential part of this thesis. A brief history of the origin of graphs and the terminology used in this thesis are described. Existing solutions for representing graphs in computer memory are also analyzed for use in the design part of this thesis.

2.1 Brief history and terminology

Gottfried Wilhelm Leibniz wrote in a letter to Christiaan Huygens:

> “*I am not content with algebra, in that it yields neither the shortest proofs nor the most beautiful constructions of geometry. Consequently, in view of this I consider that we need yet another kind of analysis, geometric or linear, which deals directly with position, just as algebra deals with magnitude...*”

The mathematician Leonhard Euler, referring to the *geometry of position* notion in this letter, in an article on the solution of the famous *Königsberg bridge problem* [12], dated 1736, which is shown in Figure 2.1, was the first to apply the ideas of graph theory to prove certain assumptions, which makes him the father of graph theory who discovered a *concept of a graph.* In his work, he did not use any graph theory terms, nor did he use drawings of graphs.

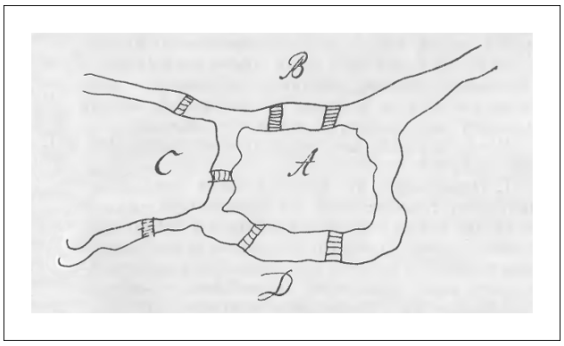


Figure 2.1 The Königsberg bridge problem [12]

At the beginning of the 20th century, the Hungarian mathematician Denes König was the first to suggest calling configurations of nodes and connections “graphs” and to study their general properties. The graph-theoretic definitions required for this work are presented [13].

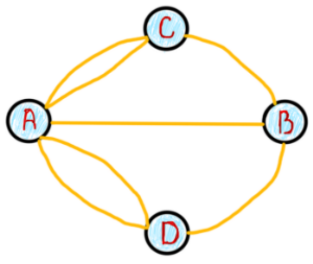


Figure 2.2 Graph representing the Königsberg bridge problem

▶ **Definition 2.1.** *A **graph** $G = (V, E)$ is a discrete structure consisting of two sets V and E:*

- *The elements of V are called vertices (or nodes);*
- *The elements of E are called edges, and each of them has a set of two vertices associated with it.*

$V(G)$ denotes the set of vertices of the graph G, and $E(G)$ denotes the set of edges. The number of vertices $|V(G)|$ and edges $|E(G)|$ determine the *order* and *size* of the graph G. Typically, the vertices of a graph have labels represented by literals such as a, b, c, etc. or integers such as $1, 2, 3$, etc. An edge is denoted as (a, b) or ab or a, b where a and b are the vertices to which the edge is connected. The vertices a and b are called the *endpoints* of the edge (a, b).

▶ **Definition 2.2.** *A **self-loop** is an edge that joins a single endpoint to itself.*

▶ **Definition 2.3.** *A **proper edge** is an edge that joins two distinct vertices.*

▶ **Definition 2.4.** *A **multi-edge** is a collection of two or more edges with identical endpoints.*

Most of the theoretical graph theory is concerned with *simple* graphs. This is in part because many problems regarding general graphs can be reduced to problems about simple graphs.

▶ **Definition 2.5.** *A **simple graph** is a graph that has no self-loops or multi-edges.*

▶ **Definition 2.6.** *A **trivial graph** is a graph consisting of one vertex and no edges.*

▶ **Definition 2.7.** *A **null graph** is a graph whose sets of vertices and edges are empty.*

▶ **Definition 2.8.** *A **weighted graph** is a graph in which a number is assigned to each edge.*

Adding and removing vertices and edges from a graph are called *primary operations* because they serve as the basis for other actions, called *secondary operations.*

▶ **Definition 2.9.** *The operation of **adding the vertex** v to the graph $G = (V, E)$, such that $v \notin V$, yields the new graph $G \cup \{v\}$ with the set of vertices $V \cup \{v\}$ and set of edges E.*

▶ **Definition 2.10.** *The operation of **removing the vertex** v from a graph $G = (V, E)$ removes not only the vertex v but also every edge along which v is an endpoint. $G - v$ denotes the resulting graph.*

▶ **Definition 2.11.** *The operation of **adding the edge** uv to the graph $G = (V, E)$, joining the vertices u and v ($\{u, v\} \subset V$), yields the new graph $G \cup \{uv\}$ with the set of vertices V and set of edges $E \cup \{uv\}$.*

▶ **Definition 2.12.** *The operation of **removing the edge** uv from the graph $G = (V, E)$ removes only this edge. $G - uv$ denotes the resulting graph.*

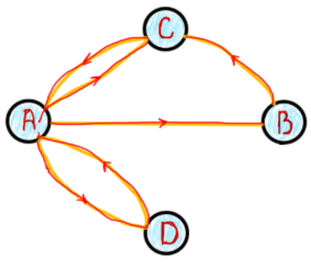


Figure 2.3 Digraph representing the potential solution to the Königsberg bridge problem

An *undirected* graph has no orientation at its edges. In an undirected graph, an edge between the vertices a and b is denoted by (a, b) or (b, a). The edges of a graph can be oriented, in which case an arrow indicates the direction, as shown in Figure 2.3. Such graphs are called *directed graphs* or *digraphs*. An edge (a, b) in such a graph indicates that this edge is directed from vertex a to b which is shown by $(a, b) \in E$ but does not mean that $(b, a) \in E$.

▶ **Definition 2.13.** *A* ***directed graph*** *is a pair* $G = (V, E)$ *comprising:*

- V*, a set of vertices;*
- $E \in \{(x, y) \mid (x, y) \in V^2,\ x \neq y\}$*, a set of edges (also called directed edges) represented as ordered pairs of vertices.*

2.2 Common representations in memory

In the previous section, the terms and definitions of graph theory were introduced. In this section, different representation techniques for an undirected graph in computer memory [13] will be analyzed for further processing. The data structures that can be used and the complexity of each method will be discussed.

There is no unambiguous standardized solution for storing graphs in computer memory. However, there are different ways to optimally represent a graph, depending on the density of its edges, the type of operations to be performed, and the ease of use. Later, a combination of the following methods with additional improvements will be used to achieve more efficient results.

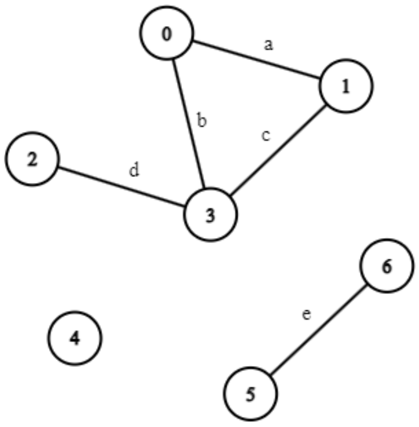


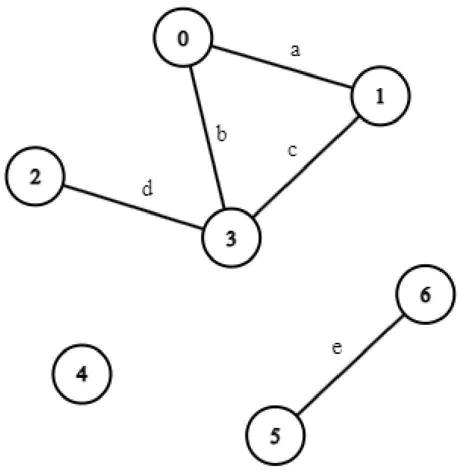
Figure 2.4 Graph used for analysis of graph representations in memory

For example, an undirected graph $G = (V, E)$, $|V| = 7$, $|E| = 5$, as shown in Figure 2.4, can be assumed. The set V consists of vertices $\{0, 1, 2, 3, 4, 5, 6\}$. The set E consists of edges $\{(0, 1), (0, 3), (1, 3), (2, 3), (5, 6)\}$. The edges are also labeled a, b, c, d, e for convenience.

2.2.1 Adjacency matrix

This is the most common way to represent a graph, but it takes up the most memory. It is appropriate for use if the number of edges approaches V^2. A two-dimensional matrix of size $|V| \times |V|$ is used to store edges. If the vertices x and y are adjacent, the element at $[x, y]$, $x \in V$, $y \in V$ equals 1, otherwise it equals 0. In the case of undirected graphs, the matrix is symmetric with respect to the main diagonal, and the sum of each row and each column is equal to the degree of the vertex.

Several improvements can be applied to reach slightly more efficient representations. In the case of simple graphs, a boolean type with values *true* or *false* can be used instead of integer numbers as matrix elements to save space. For a more advanced way, a bitfield of size $|V|$ can be used as a row of a matrix, where each bit denotes whether the vertices are adjacent. This improvement makes it possible to get close to the information-theoretic lower bound for the minimum number of bits needed to represent all n-vertex graphs [14].



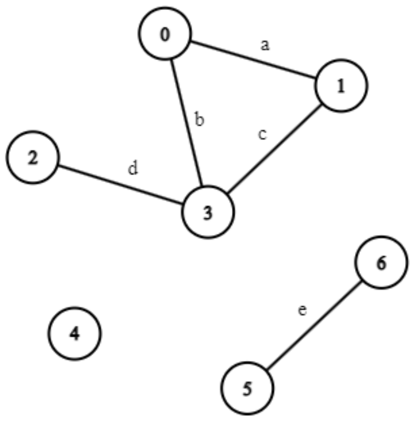
$$\begin{array}{c|ccccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
0 & * & 1 & * & 1 & * & * & * \\
1 & 1 & * & * & 1 & * & * & * \\
2 & * & * & * & 1 & * & * & * \\
3 & 1 & 1 & 1 & * & * & * & * \\
4 & * & * & * & * & * & * & * \\
5 & * & * & * & * & * & * & 1 \\
6 & * & * & * & * & * & 1 & *
\end{array}$$

Figure 2.5 Adjacency matrix

- *Space complexity*: $\mathcal{O}(|V|^2)$
- *Edges iteration complexity*: $\mathcal{O}(|V|^2)$
- *Complexity of iterating all vertices adjacent to one edge*: $\mathcal{O}(|V|)$
- *Complexity of verifying if two vertices are adjacent*: $\mathcal{O}(1)$

2.2.2 Incidence matrix

This graph representation is also memory-consuming. It is appropriate for use if the number of edges is small. To store edges, a two-dimensional matrix of size $|V| \times |E|$ is used. Each column of such a matrix has one edge with opposite vertices incident to this edge having values of 1, otherwise having a value of 0. Thus, the sum of numbers in each column equals 2, and the sum of numbers in a row equals the degree of a vertex.



	a	b	c	d	e
0	1	1	*	*	*
1	1	*	1	*	*
2	*	*	*	1	*
3	*	1	1	1	*
4	*	*	*	*	*
5	*	*	*	*	1
6	*	*	*	*	1

Figure 2.6 Incidence matrix

- *Space complexity*: $\mathcal{O}(|V| \cdot |E|)$
- *Edges iteration complexity*: $\mathcal{O}(|V| \cdot |E|)$ — even though each edge is stored in its own column, all the numbers in the column must be checked to find out the vertices associated with it
- *Complexity of iterating all vertices adjacent to one edge*: $\mathcal{O}(|V| \cdot |E|)$
- *Complexity of verifying if two vertices are adjacent*: $\mathcal{O}(|E|)$ — it is sufficient to iterate over the columns in order to find two 1s

2.2.3 Edges enumeration

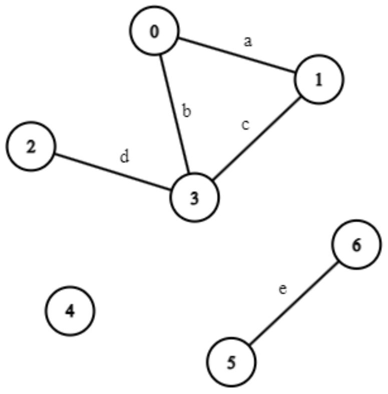
By definition, a graph is a topological model that consists of a set of vertices and a set of edges that connect them. Therefore, the “simplest” way to represent a graph is to define the set of vertices and to enumerate the elements from the set of edges.

Usually, in this structure, the vertices are not stored separately. Only their quantity n is specified, and they are numbered from 0 to $n - 1$ automatically. The vertices themselves are stored in an array or set that enumerates the edges connecting the given vertices. To store the color, weight, or other characteristics of the vertices, additional arrays can be used for each criteria.

This is a very memory-efficient approach: each edge is stored once, whereas in all previous alternatives, each edge is usually written twice. However, when searching for vertices in the list of edges, it is necessary to perform two checks where both the first and second vertex are compared.

- *Space complexity*: $\mathcal{O}(|E|)$
- *Edges iteration complexity*: $\mathcal{O}(|E|)$
- *Complexity of iterating all vertices adjacent to one edge*: $\mathcal{O}(|E|)$
- *Complexity of verifying if two vertices are adjacent*: $\mathcal{O}(|E|)$

The list of edges can be grouped by vertices, which speeds up the search for adjacent vertices. Thus, another method can be obtained that is similar to this one but differs in the way the elements are stored.



```
n = 7

{
 {0, 1},
 {0, 3},
 {1, 3},
 {2, 3},
 {5, 6}
}
```

Figure 2.7 Edges enumeration

2.2.4 Adjacency list

This structure represents a collection of unordered lists. Each unordered list within an adjacency list associates each vertex in a graph with a collection of its neighboring vertices. It is more suitable for sparse graphs in which most pairs of vertices are not connected by edges. For this type of graph, an adjacency list is significantly more space-efficient than the adjacency matrix. The use of space in the adjacency list is proportional to the number of edges and vertices in the graph, whereas for the adjacency matrix stored in this way, the space is proportional to the square of the number of vertices.

The next significant difference between adjacency lists and matrices is the efficiency of their operations. The neighbors of each vertex in the adjacency list can be efficiently enumerated in time proportional to the degree of the vertex. In the adjacency matrix, this operation takes time proportional to the number of vertices in the graph, which can be much higher than the degree. On the other hand, the adjacency matrix lets us check in constant time whether or not two vertices are adjacent.

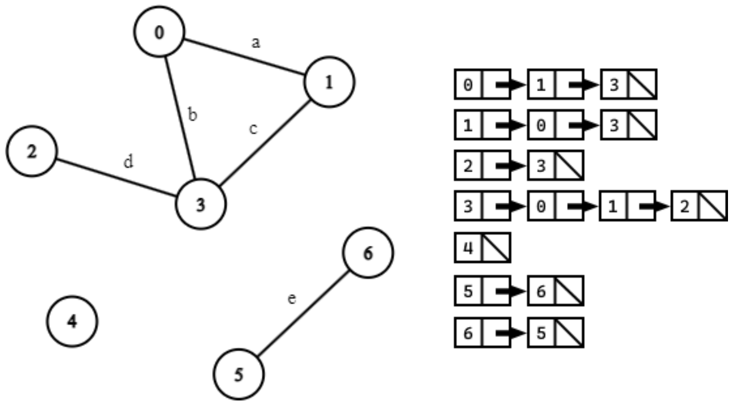


Figure 2.8 Adjacency list

- *Space complexity*: $\mathcal{O}(|V| + |E|)$
- *Edges iteration complexity*: $\mathcal{O}(|V| + |E|)$
- *Complexity of iterating all vertices adjacent to one edge*: $\mathcal{O}(|V|)$
- *Complexity of verifying if two vertices are adjacent*: $\mathcal{O}(|V|)$

Chapter 3
Assignment analysis

This chapter aims to examine and analyze the assignment specifications of this thesis. The first section describes the assignment and the clarifications that were used. The second section presents an existing solution related to the assignment. The third section focuses on the gathering and analysis of requirements. In the fourth section, the various use cases that were identified are discussed.

3.1 Work assignment

The assignment, as it was formulated and translated into English, is:

> The goal of this thesis is to design and implement a prototype of a graph editor and viewer in Virtual Reality (VR) using Unreal Engine (UE). The prototype should be able to read and save graphs from an external file and should be able to add, remove, and edit the vertices and edges of a graph. It should also be optimized for large graphs (thousands of vertices).

3.1.1 Agreed limitations

After a number of consultations with the thesis's supervisor, Ing. Petr Pauš, Ph.D., and with his subsequent approval, it was decided to make the following changes, or, to be more precise, clarifications, to the specification of the assignment:

1. The prototype will be developed for desktop hardware and will only be supported on Windows (64-bit) platforms.
2. The prototype will aim to support Oculus VR headsets and will be tested specifically on the Oculus Quest 2 headset, shown in Figures 1.1 and 1.2.
3. Only trivial and simple finite labeled graphs and operations on them will be supported.
4. The prototype will be more focused on performance than on graphics quality.

These clarifications will serve as criteria that limit the scope and complexity of the practical part of this thesis and will be taken into account in the upcoming requirements analysis.

3.1.2 Added functionality

It was also agreed that the resulting project would have the following features and improvements:

1. The prototype will allow users not only to edit a graph imported from an external file but also to create a new graph.
2. In addition to the above graph operations, the prototype will also give users the option to change the colors of the vertices.
3. The prototype will be capable of supporting weighted graphs.
4. Although the prototype is meant to work with Oculus VR, it had to be made so that it would be easy to add support for other VR devices in the future.

3.2 Related work

Before analyzing the requirements, a detailed search of already existing solutions for this problem was performed. An open-source project called *3D Force-Directed Graph in VR* was discovered, with the code available on GitHub [15] under the MIT license.

This project provides a web component to represent a graph data structure in VR. This web component is written in JavaScript and can be used in web development projects by importing it as a module. The web component is capable of displaying graphs imported from a JSON file. The structure of JSON files used to store graphs is also specified in the GitHub repository. It supports directed, undirected, colored, and weighted graphs. There is the ability to use custom vertex geometry and detect vertex collisions. Asynchronous loading for larger graphs is also provided.

However, this project is not a standalone application and allows for changing the loaded graph only programmatically. This means that this functionality can only be implemented by developers who use this module in their projects. During the development of my application, this project inspired me. In particular, the JSON file structure used to store a graph and the way this web component visually displays an imported graph, which is shown in Figure 3.1.

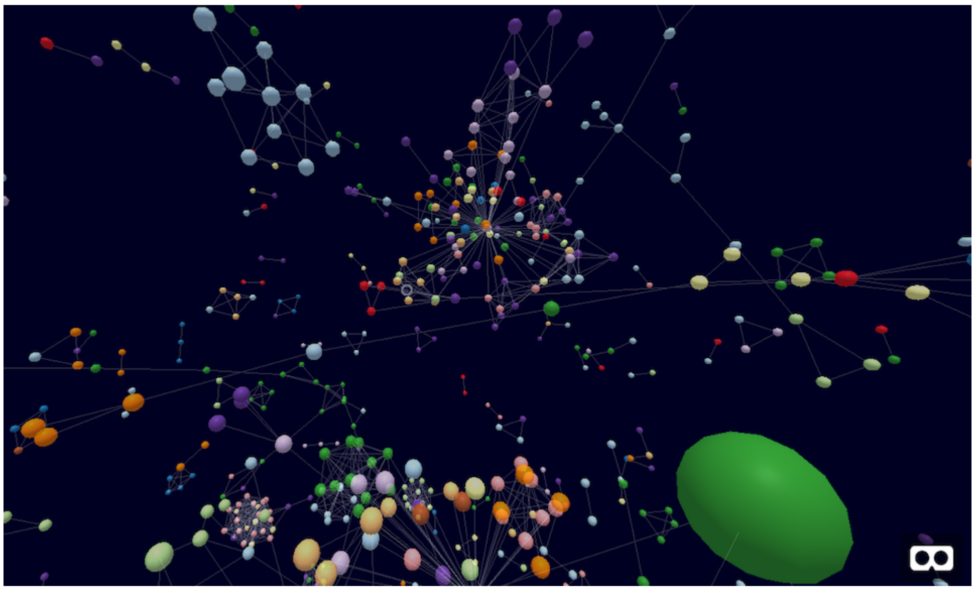


Figure 3.1 3D Force-Directed Graph in VR [15]

3.3 Requirements specification

The requirements for the given specification need to be obtained and analyzed before continuing with the direct design and development of an application that satisfies the criteria specified in the assignment. During this phase, we will be able to specify the limits of the resulting system as well as the constraints that will be placed upon it.

3.3.1 Functional requirements

Several functional requirements have been singled out for consideration following an in-depth examination of the assignment specification. The criteria produced as a result account for all of the goals of the assignment and provide all the functionality that is required to complete it. The following is a comprehensive listing of all key functional requirements:

F1 – Graph visualization In addition to the basic requirements for graph operations, the application must support the ability to render multiple graphs in a scene. Each graph in the scene will be treated as a separate object with its own set of objects with which the user can interact. The scene itself is represented by a large space. When the application starts, the scene is empty, that is, no graph is located on it. The rendering of the graph will be redone each time, depending on the operation that was performed on it.

F2 – User movement The graphs that are being shown can be placed in different regions of the scene. The application must give users the ability to roam freely about the scene in order to examine and interact with the individual graphs in a more comfortable way. Using the controller, the movement will be accomplished by teleporting to a specified point. This type of movement was chosen based on the analysis of the common available VR locomotion systems, described in Section 1.1.3. It should also be allowed to rotate the user's camera with the controller to compensate for the fact that the user's head can only turn in a certain range.

F3 – Import graph The application will make it possible to add a graph to the scene by reading its structure from an existing external file. The application will provide the user with a list of all files that contain a graph structure. It will not check the entire disk for such files, but only one specific folder is created to store files for import and export. This check will not be automatic and continuous and will be performed after the user requests for the list to be refreshed. Only files with the JSON extension will be shown, regardless of whether or not they have a correct graph structure. All other files in the folder will be ignored. The graph will be added to the scene only after the user has decided which file to import. A valid graph structure must be present in such a file, and the application needs to ensure that the file satisfies this condition before proceeding with the actual deserialization and visualization of the object. This valid structure must contain at least one vertex. All vertex labels must be unique in the scope of the parent graph, and all edges must connect only declared vertices.

F4 – Export graph The application will not only allow users to import graphs but also provide the opportunity to save a graph located in the scene to an external file. The graph structure that the user selects for export will not be altered and will be serialized in the exact same form as presented in the scene. When imported after being exported, the graph should maintain its original appearance. After the user selects the graph, it will always be exported to a new file. This file must be created in a shared folder that serves the purpose of storing files related to graph import and export. The folder should be next to the application's executable file and must be created in the event of an absence. In order to prevent name collisions with other files in the output folder, the name of the new file will be generated pseudo-randomly.

F5 – Create object Users will have the option to manually build graphs using the application. To create a new graph, the user must initially set up the scene with at least one vertex. To fulfill this objective, the ability to create vertices and edges will be made available. This feature must be able to create not only new graphs but also new vertices and edges and add those to existing graphs that are already present in the scene. The user will be able to choose which graph receives the addition before it occurs. The definitions of the graph operations defined in Section 2.1 will be followed in the process of adding objects to a graph. The default settings will be applied to all newly created objects. Therefore, the vertices will be rendered with a default color and labeled with an incremental value when they are added to the graph, starting from 0. The process of adding vertices and edges will continue until the user decides that the selected graph is done.

F6 – Edit object The application will provide the capability to alter the attributes of graphs and their objects that are located in the scene. Depending on the object with which the user chooses to work, different sets of editable properties will be presented to them. For the graphs, it will be possible to change the color of all their vertices. For the edges, it will be possible to change their weight. For the vertices, it will be possible to change their colors. After a change is made to any of the properties, the result will immediately be applied to the object that has been selected. It will be possible to undo the modifications that were made and return the object to the state in which it was before the user made their choices. The manipulation of graphs is another component of this requirement. It is necessary for the application to include functionality that allows users to adjust the location of a specific object in the scene.

F7 – Remove object It is possible to remove graphs and any items associated with them from the scene. Only the objects chosen by the user will be removed. Also, the option to select an entire graph should be available. All its vertices and edges will be selected after this action. The removal of content will not be performed automatically, but rather only at the user's request by pressing the button in the UI. The definitions of the graph operations defined in Section 2.1 will be followed in the process of removing objects from a graph. After the removal of the final vertex of a graph, the graph itself will be eliminated likewise.

3.3.2 Non-functional requirements

In addition to gathering and examining functional requirements, non-functional requirements were also included in the scope of the project. Such requirements specify criteria that can be used to judge the operation of the system, rather than specific behaviors. The following is a comprehensive listing of all key non-functional requirements:

NF1 – Portability The following 64-bit Windows operating systems will be supported by the application: Windows 10, Windows 11. The connected Oculus Quest 2 VR device will be used to interact with the application. It must also be easily scalable to support other VR devices in the future.

NF2 – Performance The application needs to have good performance when visualizing larger graphs to prevent users from experiencing simulation sickness. The system must be optimized enough to visualize at least 2 graphs with 5,000 vertices and 5,000 edges each at the same time and at least 10 graphs with 1,000 vertices and 1,000 edges each at the same time while maintaining optimal performance. Optimal performance for Oculus Quest 2 targets a 90 FPS rate, as provided in the *VR and Simulation Sickness* guideline [16]. This frame rate translates to $1000/90 = 11.11$ ms, which is the maximum time that it should take to render a frame.

NF3 – Usability Since the application is intended for VR, its design must take into account the best practices to provide a positive user experience in VR, as described in Section 1.1.

When using the application, users should not experience any negative health effects. The user's experience interacting with the application must be understandable, efficient, and easy to recall.

3.4 Use cases

To provide a comprehensive specification of the functional requirements, various use cases and scenarios of the application have been compiled. The constructed use case model is depicted in Figure 3.2.

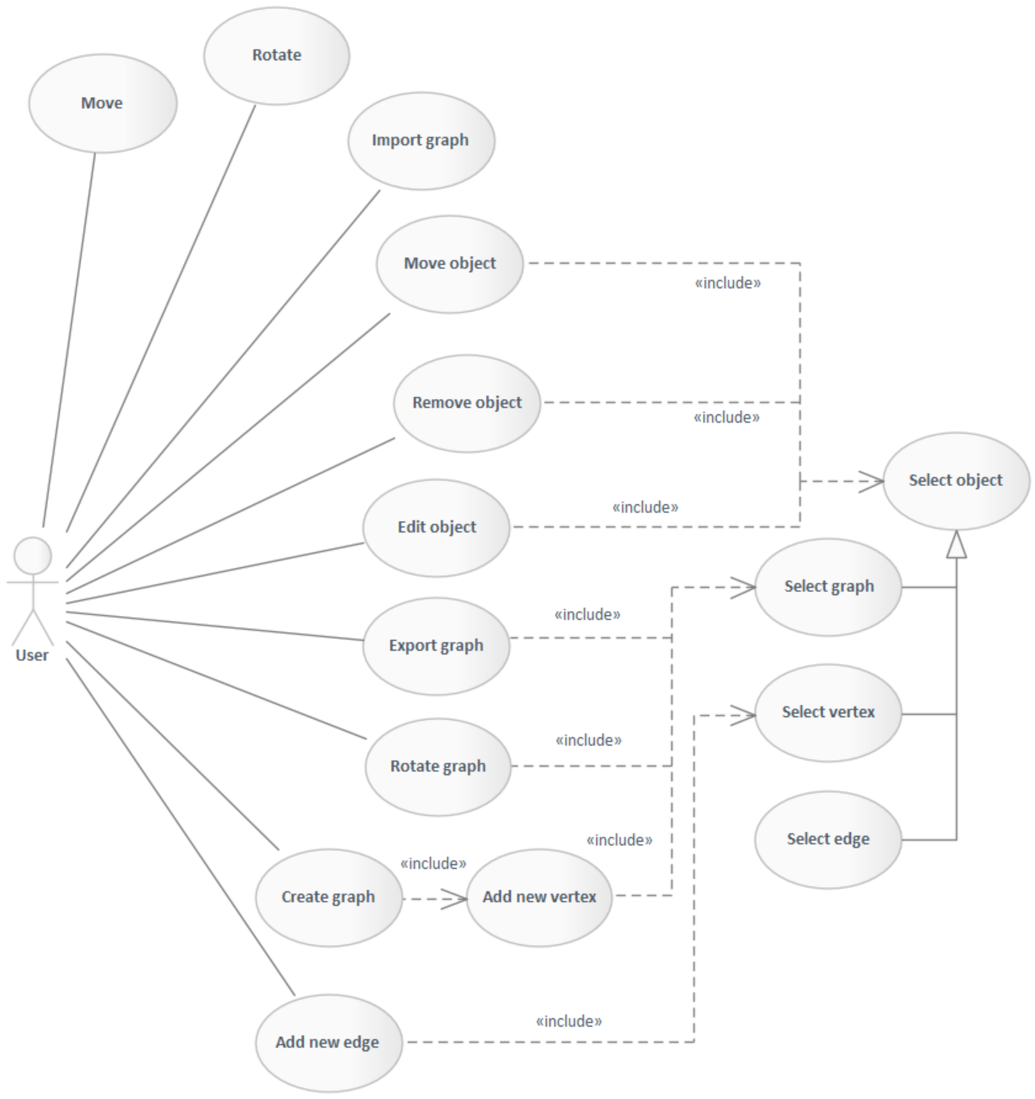


Figure 3.2 Use case diagram

The components of the model, as well as the scenarios for each use case, are outlined in the following list.

UC1 – Import graph Provides the ability for the user to add a graph by reading its structure from an external file.

1. This scenario starts when the user makes the decision to import a graph from a file into the scene in order to continue working with it.
2. The user is presented with a list of all accessible JSON files that are located in the shared folder for import and export purposes. If the folder cannot be located, the system will create it automatically.
3. The user chooses a file from the list that has been suggested. The system will deserialize the object after reading the contents of the specified file.
4. If the process is successful, the system will visualize the newly read graph in the scene and assign a distinct color to it so that it can be easily identified. In the event that the system cannot read the specified file or successfully deserialize the object, an error message will be displayed to the user in the UI.
5. The user confirms the result of the operation.
6. From Step 2, the scenario continues until the user decides not to import a new graph.

UC2 – Select object Provides the ability for the user to select an object to which they want to apply operations subsequently. This use case is essentially a generalization of the following three use cases: **UC2.1 – Select graph, UC2.2 – Select vertex, and UC2.3 – Select edge**. The only thing that differentiates these use cases from the scenario described below is the type of object that will be selected.

1. This scenario starts when the user makes the decision to select a specific object in the scene to start working with it.
2. After pointing to the object they want to select in the scene, the user presses a button on the controller to confirm the choice.
3. The system marks the object as selected and draws it again in a different, unique color for identification.

UC3 – Export graph Provides the ability for the user to save a graph to an external file.

1. This scenario starts when the user decides to save the structure of a graph by exporting it to an external file.
2. The user selects a desired graph in the scene (include UC2.1 – Select graph).
3. The system creates a new JSON file with a unique name and places it in the shared folder for import and export purposes. The user-selected graph structure will be serialized and exported to the new file.
4. If the process is successful, the system will display the path to the new file in the UI. If the system cannot create the new file or serialize the chosen object, an error message will show up in the UI.
5. The user confirms the result of the operation, and the system deselects the chosen graph.
6. From Step 2, the scenario continues until the user decides not to export a graph.

UC4 – Create graph Provides the ability for the user to build a new graph.

1. This scenario starts when the user makes the decision to create a new graph in the scene.
2. The system will create a new graph that is temporarily empty and will be marked as selected automatically.
3. The user adds vertices to the selected graph (include UC5 – Add new vertex).

UC5 – Add new vertex Provides the ability for the user to add a new vertex to a graph.

1. This scenario starts when the user makes the decision to add a new vertex to a graph.
2. The user selects the desired graph in the scene (include UC2.1 – Select graph).
3. The user receives a preview of the new vertex, which is attached relatively to the controller. They will be able to move this preview, as well as push or pull it, to indicate the location of the new vertex in a 3D space.
4. After the location has been indicated and confirmed through the use of the controller, the system will create the new vertex that will be located in this position and add it to the selected graph. The new vertex will be drawn in a default color, and its label will be assigned a value next to the maximum vertex label in the graph.
5. The scenario continues from Step 3 until the user decides not to add the new vertex, at which point the desired graph will be deselected.

UC6 – Add new edge Provides the ability for the user to add a new edge to a graph.

1. This scenario starts when the user makes the decision to add a new edge to a graph.
2. The user selects two vertices in the scene (include UC2.2 – Select vertex).
3. The system checks if the two selected vertices belong to the same graph and if there is no existing edge that connects them. If these conditions are met, the system will create the new edge that connects the two selected vertices.
4. The two desired vertices will be deselected, and the scenario will go back to Step 2 until the user decides not to add the new edge.

UC7 – Rotate graph Provides the ability for the user to rotate a graph.

1. This scenario starts when the user makes the decision to rotate a specific graph.
2. The user selects the desired graph in the scene (include UC2.1 – Select graph).
3. Using the thumbstick on the controller, the user can tell the system to rotate the chosen graph in a certain direction.
4. The scenario continues from Step 3 until the user decides not to rotate the desired graph, at which point it will be deselected.

UC8 – Move object Provides the ability for the user to move an object.

1. This scenario starts when the user makes the decision to move a specific object.
2. The user selects the desired object in the scene (include UC2 – Select object).
3. The system updates the location of the selected object based on the movements of the controller made by the user.
4. The scenario continues from Step 3 until the user decides not to move the desired object, at which point it will be deselected.

UC9 – Edit object Provides the ability for the user to edit the properties of an object.

1. This scenario starts when the user makes the decision to edit the properties of a specific object.
2. The user selects the desired object in the scene (include UC2 – Select object).
3. The user changes the required properties from a list of all the object properties that can be modified. Any property change is immediately reflected by the system on the selected object.

4. The edited object will be saved if the user decides to apply the altered properties. If that is not the case, the system will restore the object to the state it was in before the user made their choices.
5. The scenario continues from Step 3 until the user decides not to edit the desired object, at which point it will be deselected.

UC10 – Remove object Provides the ability for the user to remove objects.

1. This scenario starts when the user makes the decision to remove specific objects.
2. The user selects the desired objects in the scene (include UC2 – Select object).
3. After receiving confirmation from the user, the system will proceed to remove the objects from the scene. If this does not occur, the system will deselect the desired objects.

UC11 – Move Provides the ability for the user to move to a specified location within the scene.

1. This scenario starts when the user enters the camera movement mode.
2. The user receives a preview of the teleportation point, which is relatively attached to the controller. The user will be able to move this preview, as well as push or pull it, to indicate the location of the teleportation point in a 3D space.
3. Once the user has chosen the location and confirmed it with the controller, the system will teleport them there with a smooth screen fade effect to prevent feelings of sickness.
4. The scenario continues from Step 2 until the user leaves the camera movement mode.

UC12 – Rotate Provides the ability for the user to rotate in the selected direction.

1. This scenario starts when the user indicates and confirms the direction of rotation with the thumbstick on the controller.
2. The system turns the user 45 degrees in the specified direction with a smooth screen fade effect to prevent feelings of sickness.

Chapter 4

Graph representation

This chapter provides representations of the graph structure used in different aspects of the application. The first section presents a compiled representation domain model, on the basis of which the following representations will be designed. The second section aims at a visual representation of 3D graphs within the scene. The third section discusses a designed schema for an external file, describing the graph structure for importing and exporting.

4.1 Composition

When designing a graph representation that will be used in various aspects of the application, it is vital to take into account all the characteristics and attributes that are associated with the common graph structure and its components. A domain model was created based on the analysis of the assignment and the requirements for the application in terms of the supported graph types and classes. The model illustrates the structure of the general graph representation, as well as its attributes, components, and relations between them. The domain model itself is shown in Figure 4.1.

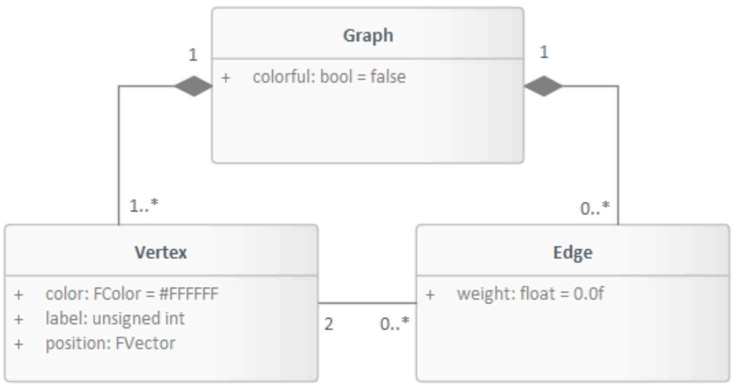


Figure 4.1 Graph representation domain model

The resulting structure consists of the graph structure itself, represented as a collection of vertices and edges. The vertex and edge structures cannot exist independently of the parent graph structure and will be removed if the parent is removed. A graph can have any number of vertices and edges but must have at least one vertex. If a graph contains edges, each of them must connect exactly two vertices of the same parent graph.

The graph structure has only the attribute “colorful,” which will be applied in conjunction with the attribute “color” of the descendant vertices. These attributes are necessary for the realization of colored graphs. The “colorful” attribute is used to enable and disable the colors of all vertices in the graph, allowing the colors assigned to them to be preserved. The two attributes of the vertex are “label” and “position,” which denote its label and position in the 3D scene, respectively. The only attribute of the edge structure is “weight,” which is necessary for the realization of weighted graphs.

4.2 Visual

Since the application will contain a graphical component, it is important to consider the design of the visual graph representation. The design is based on the related work described in Section 3.2 and is carried out taking into account the convenience of the user when interacting with 3D graphs. The resulting representation is described using an example of a graph depicted in the model sketch in Figure 4.2.

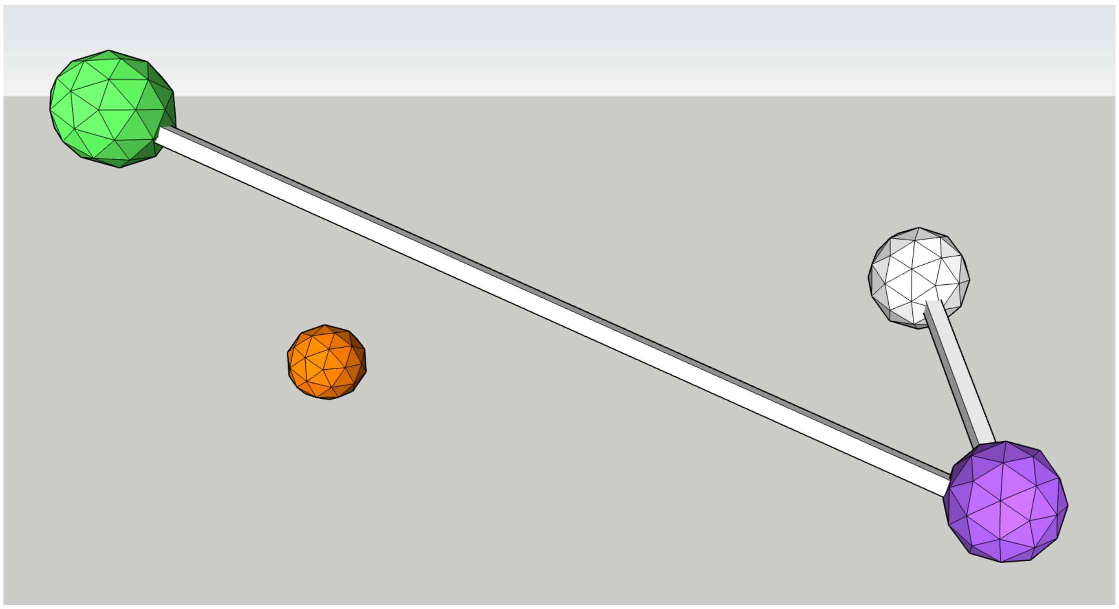


Figure 4.2 Sketch of the visual graph representation

This example shows the graph $G = (\{0, 1, 2, 3\}, \{(0, 1), (0, 2)\})$. This graph is colored and its vertices $\{0, 1, 2, 3\}$ are assigned the colors purple, white, green, and orange, respectively. All of the graph’s and its components’ attributes will not be shown on the model itself. Instead, they will be shown in the UI, which will be described in Section 8.3.

For the visual aspect, the vertices of the graphs will be rendered as *1-order icospheres* [17]. This model was chosen with a focus on performance to reduce the load on the GPU. It only has 42 vertices and 80 triangles, which is not a lot for a good approximation to a sphere. The edges will be rendered as *parallelepipeds* connecting vertices of the graph. It allows users to interact with edges and is also efficient for rendering since it has only 8 vertices and 12 triangles.

4.3 External file schema

In order for the application to fulfill the functional requirements, it needs to be able to read graphs from and write graphs to external files. To successfully achieve these requirements, it is necessary to choose the right format for a file that will hold a 3D graph representation and to design a file's schema.

An external file will be in JSON format rather than binary in order to give users a legible form of graph that can be altered outside of the application. This format provides the ability to describe objects in a way that is easy for humans to understand while still being a lightweight structure. The extensive adoption of the format across the industry also has an impact. As a consequence of its widespread use, there are tools for interacting with this format included in the frameworks and standard libraries of many contemporary programming languages.

To provide a foundation for consideration of the designed file schema, the example that is supplied in Code Listing 4.1 will be used. This particular example relates to the graph $G = (\{0,1,2,3\}, \{(0,1),(0,2)\})$, the model for which was illustrated earlier in Figure 4.2.

Code Listing 4.1 JSON graph representation example

```
{
    "colorful": true,
    "vertices": [
        {
            "label": 0,
            "position": "X=437.110 Y=225.097 Z=50.000",
            "color": "#1F00FF"
        },
        {
            "label": 1,
            "position": "X=748.975 Y=345.263 Z=260.000"
        },
        {
            "label": 2,
            "position": "X=504.859 Y=-437.557 Z=460.000",
            "color": "#00FF39"
        },
        {
            "label": 3,
            "position": "X=969.929 Y=-452.031 Z=260.000",
            "color": "#FF0500"
        }
    ],
    "edges": [
        {
            "from": 0,
            "to": 1,
            "weight": 0.5
        },
        {
            "from": 0,
            "to": 2,
            "weight": 2.0
        }
    ]
}
```

The root of the schema is a single JSON object that represents the graph structure. This is necessary because only one graph will be contained in one file, as discussed in the functional requirements in Section 3.3.1. This object, along with any others that follow, are composed of properties that represent key-value pairs. There is only one “vertices” property that is necessary for this object. The “colorful” and “edges” properties are considered to be optional, which means that they are not required to be presented in a file.

The optional “colorful” property accepts boolean values of true or false. In the example above, the property is set to true, which will allow the application to render the vertices using the colors that have been provided for them in the “color” property. If it was false or not specified, the vertices would be shown in the standard white color, but they would keep the colors that were set for them in case the value was later changed to true.

The “vertices” property is assigned a value consisting of an array of JSON objects that each stand for one of the graph’s vertices. This property is required, which means that the array must have the declaration of at least one vertex in order to contain a legitimate graph structure. The “label” and “position” properties of the JSON vertex object are required, while the “color” property is optional. The objects in the array can be specified in any order with any label value. For the graph structure to be valid, the labels of the vertices must be unique in the scope of the parent graph.

The mandatory “label” property is set as an integer number that denotes the label of a vertex. This property is intended to be a convenient way to specify vertices when declaring edges. Some consideration was also given to the potential for relative vertex identification. It would be possible to use the indices of the vertices in the array rather than the labels. However, this method is not convenient for the user because it would be necessary to manually count the vertex index and, in the case of adding a vertex at the beginning or middle of the array, reassign all specified edges.

The “position” is the next required property of a vertex. This property determines where the vertex is located in the 3D scene. It possesses a string value in the format shown in Code Listing 4.1. This format was selected because of its compactness and ability to save space in a file. It is also used internally in Unreal Engine when serializing and deserializing instances of the FVector class, which is a representation of a vector in a 3D space composed of X, Y, and Z components with floating point precision.

The “color” property gives the user the ability to customize the color of each individual vertex. It has a string value that specifies the color using a hexadecimal encoding of the RGB color model. This representation was selected for the same reasons as the position representation: to reduce the size of the resulting file and to simplify the process of serializing and deserializing instances of the FColor class in Unreal Engine, which stores a color with 8 bits of precision per channel. This property is optional and may not be specified by the user in a file. As demonstrated by the vertex with label 1 in Code Listing 4.1 and in Figure 4.2, if the “color” property for a vertex is not set, the application will use white as the default color.

The “edges” property contains an array of JSON objects that represent undirected edges. Since a valid graph structure may not contain edges, this property is optional and may not be specified by the user or may have an empty array. The “from” and “to” properties of the edge object are required, and their purpose is to identify related vertices on the basis of their labels. This specification might also be used to denote directed edges in later versions of the application. The objects in the array can be specified in any order and can contain any label values. However, to be valid for the graph structure, undirected edges must be unique and contain valid labels, which means that they must be declared at the vertices of the same graph.

The “weight” property stores a floating-point number that indicates the weight of a certain edge. Additionally, it is not required, and the user only needs to provide it when they want to deal with weighted graphs.

Chapter 5
Application architecture

This chapter superficially provides a general view of the resulting application architecture. In its basic form, the design process of the application may essentially be split up into two parts. The first section refers to the first part, which is related to the representation of graphs in the scene and includes both their visualization and management. The second section refers to the second part, which is concerned with the design of the user's interaction with graphs, involving the addition of new elements, the removal of existing ones, or the application of various operations on them. The third section aims to describe the overall composition of the system as well as the relationships between all its modules.

In this chapter, we will begin to dive deeper into the architectural design of the application itself. All of the requirements, both functional and non-functional, that are outlined in Section 3.3, as well as all of the use cases that are presented in Section 3.4, serve as the foundation for the whole architectural design of the application. It is also important to note that the design was created with the intention of providing structures and interfaces that may be easily and conveniently used during subsequent implementation.

5.1 Graph visualization

Providing the user with a visual representation of 3D graphs is one of the most essential features of the application. For this purpose, it is necessary to design a part of the system that will be responsible for storing and processing graphs in memory and providing an accordant presentation with which the user can subsequently interact. Since storing and accessing graph objects and their visualization are themselves rather complex parts of the application, it was decided to use separation of concerns to split these functionalities into corresponding modules. In this way, two modules will be used to represent graphs within the application, providing a backend and a frontend for them.

The graphs backend is the first module that will be considered. This module will introduce an *entity system* that will manage graph objects and the attributes associated with those objects in the computer memory during the time that the application is being used. It will also provide a convenient interface for accessing certain objects in order to perform operations on them or modify the attributes of those objects. The architecture of this module will be described in more detail in Section 7.1.

The graphs frontend is the next module that needs to be contemplated. This module will supply the application with a *graph rendering system*, the purpose of which will be to create, redraw, and delete actors located in the scene that are responsible for rendering graph objects according to their attributes. This module will, in essence, be a visual reflection of the backend. It will offer users an interactive layer of abstraction through which they can examine and interact with entities themselves. The architecture of this module will be described in more detail in Section 7.2.

5.2 User interaction

The next most significant feature of the application is the user's ability to modify and interact with objects that are present in the scene. According to the assignment requirements, the user's interaction with the application will be carried out through the use of connected VR devices. The VR headset will act as both an input and an output device, while the controllers themselves will function as input devices for the system. All features of the application that give the user the ability to interact with the system and its elements will be realized in the third and final module of the application, which will be described in more detail in Chapter 8.

This module will supply a number of actors that are tasked with the responsibility of providing the user's representation in the scene and communication with other components of the system. When the user launches the application, Unreal Engine's functionality will be used to link these actors with the user's connected VR devices. As soon as the devices are linked, the input will be mapped, and the actors will simultaneously begin to get the device data. These data will include the positions of the controllers in the world, the position of the headset, the buttons that have been pressed, the degree to which triggers have been pressed, and so on. Because of this, the actors will be able to visualize corresponding device models in the scene and notify another component of the system about the user's actions.

This component will be a *tool system*, which will also be realized in this module. It will consist of a set of tools, each of which will be responsible for providing appropriate functionality regarding operations and interactions with graphs. This system will, in essence, serve as a communication provider between the user and the application. With the help of these tools, the user will be able to designate exactly what operation they want to apply to a selected graph within the scene. This system was chosen because it has the useful property of *extensibility*, which makes it much easier to add a new functionality in the future simply by adding an appropriate tool for this. The system itself will be designed as a component that, based on VR device data received from user actors, will transfer commands to the graphs backend module. The commands that are transmitted will be determined by a tool that the user currently uses.

The system will offer a total of six different tools to interact with the various types of objects present in the scene. This toolset was designed according to the functional requirements outlined in Section 3.3.1, and each functionality is reflected as a separate tool. Importing, exporting, creating, altering attributes, manipulating regarding position in the scene, and removing from the scene are the purposes that each of these serves. The functionality provided by a tool is represented by a predetermined set of operations that make up the tool itself. When the user "equips" a tool, they can choose a specific action and change the parameters that control that action. All of this will be handled by the user through the corresponding UI.

Since the application is intended for VR, its design must take into account the best practices for providing a positive user experience in VR. Due to the control limits described in Section 1.1.4, the user's interaction with the application will be realized through the UI to create a good control model. The UI will be designed as a component attached to the left controller. It will appear to the user in the form of a frame that is made up of a series of tabs and windows. The administration of the tool system will be moved to a separate main tab, the window of which will provide a list of all the tools that can be used. Choosing one of the available tools will command the system to activate that tool, and then the current tab will display the appropriate window with the actions and configurations for the activated tool. After activation of a tool, all user actions related both to the UI and to the interaction with graphs within the scene will be addressed by the tool system to the active tool.

5.3 Overall composition

As a result, the application will have a modular architecture consisting of three communicating and interconnected modules. To minimize the overall complexity of the system and to improve extensibility and maintainability of the code, the architectural design of each module will be based on the rule “low coupling, high cohesion.”

To provide a more simplified view of the overall composition of the system architecture, a model was created based on the superficial descriptions of the modules proposed in Sections 5.1 and 5.2, respectively. This model describes the structure of the application and the relationships between its modules. It was also created to understand the flow of internal processes for storage, visualization, interaction, and graph processing. In this way, it shows the collaboration and sequential communication of all modules of the application, which in the future will allow it to fulfill all functional requirements and realize all possible use cases. For better illustration, the model shows a structure containing three graphs. The model itself is shown in Figure 5.1 below.

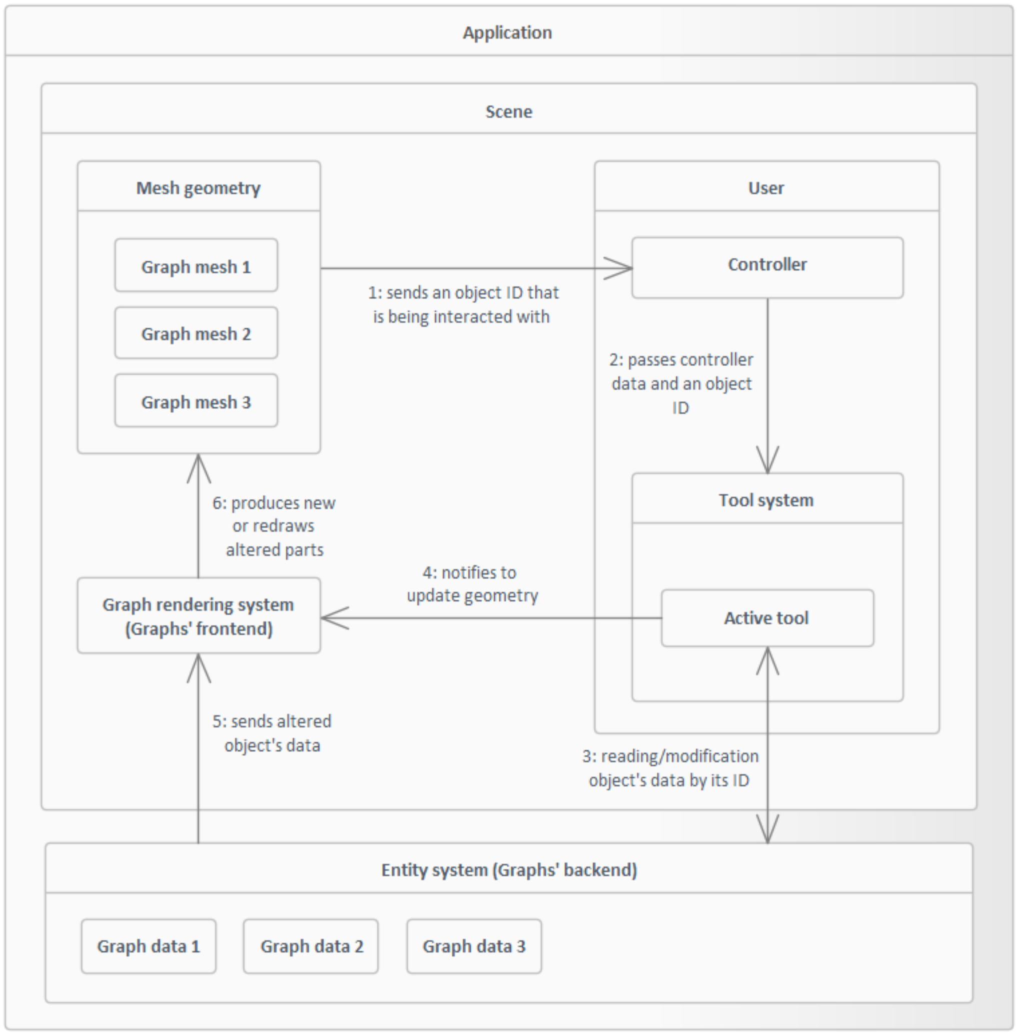

Application
Scene
Mesh geometry
Graph mesh 1
Graph mesh 2
Graph mesh 3
User
Controller
1: sends an object ID that is being interacted with
2: passes controller data and an object ID
Tool system
Active tool
6: produces new or redraws altered parts
4: notifies to update geometry
Graph rendering system (Graphs' frontend)
5: sends altered object's data
3: reading/modification object's data by its ID
Entity system (Graphs' backend)
Graph data 1
Graph data 2
Graph data 3

Figure 5.1 Application structure and process flow

The scene will be the most important part of the application, at least from the user's point of view. In essence, it will be a workspace and a container for all components that have a visual component or produce it in some way, thereby being the highest layer of abstraction for the user. Since the entity system itself neither has visualization functionality nor is involved in its creation, its existence will be taken out of the scene. Thus, the scene and the entity system will make up the top layer of the application.

Two actors will be present in the scene. The first is a user representation that is responsible for showing an image on the headset, processing data from the VR devices, and working with the internal tool system. This actor will consist of many subcomponents. One of them is a controller's representation, which in essence will be a reflection of the real VR controllers of the user in the virtual scene of the application. Its main task will be to detect interactions with objects in the scene and send data to the tool system, which is the next important component of the actor.

The following scenario of the process of interaction between controllers and objects in the scene will be considered the primary one. Every frame, a laser will be emitted from the user's primary virtual controller, giving them the ability to select a desired object in the scene by pointing at it. If the laser detects an object's collision, the virtual controller will get its identifier and notify the tool system accordingly. It is also worth noting that all input data received from the real VR controllers will also be sent by their virtual representations to the tool system for further processing and reaction by the tools to button presses, thumbstick movements, etc.

The tool system will take care of receiving interacting objects' identifiers, their processing depending on an active tool, and the overall management of all tools. Hereinafter, "active tool" means a tool that the user has "equipped." Activation of a tool allows the system to redirect information about input data from controllers to it and to "filter" identifiers based on its functionality. So, for example, if an active tool implies working only with graph vertices, the system will not transfer an identifier of a whole graph or an edge when it is obtained. Based on its settings, object's identifier, and controller's input data, an active tool will perform its corresponding operation on the object.

This will be accomplished by querying the object's data from a repository and changing it accordingly. In the role of repository will be the entity system, responsible for storing and managing all object data and supplying them based on received identifiers. After or during a change to the object data, an active tool will give a command to a second actor in the scene responsible for rendering objects based on the changes made to the entity system.

After receiving the command with the object identifier, the rendering system will request its changed data from the entity system. On the basis of this, the system will redraw the object's geometry in the scene, taking into account its changed properties. It will also save the object's identifier in the geometry collision "metadata" so that it can be interacted with user controllers in the future.

Chapter 6

Graph visualization: a naive approach

This chapter is an introduction to the architectural design of graph visualization modules. Its goal is to discuss the problems with a naive method to visualize and represent 3D graphs within the scene, as opposed to the systems described in the next chapters. The first section introduces Unreal Engine in a little more detail and discusses the actor system it provides. The second section describes a method for rendering small-sized graphs using the internal actor system. The third section discusses the problems with the proposed method when visualizing larger graphs.

6.1 Actor system

To understand the reasons for the actions taken in the subsequent design, it is necessary to understand the structure of Unreal Engine concerning the system of actors it offers.

Regardless of what exactly is provided in the scene, it should be a game object. A *game object* can be thought of as anything within the scene that must either be drawn, updated, or both updated and drawn on each frame. Despite the fact that it is referred to as a "game object," this does not automatically imply that it must be represented by a conventional object in the object-oriented sense. Some games make use of standard objects, but the majority of them rely on composition or other approaches that are significantly more difficult. The basic building block in Unreal Engine that represents a game object is UObject.

UObject [18] is the base class for almost everything in Unreal Engine. From this class, the vast majority of objects that are created in the world or just in memory are inherited, including objects in the scene, components, different types for working with data, and others. Although lighter than its descendant classes, the class itself is quite functional. For example, it contains many useful events, such as changing the values of variables in the editor and the basic functions for the network, which are not active by default. Also, one of its main features is working with the garbage collector in Unreal Engine in a way that is approximately similar to how, for example, in Java, the garbage collection system is implemented with the base class Object. Objects created by UObject class exist only in memory and cannot be presented in the scene. For this purpose, Unreal Engine has a number of object types called actors.

Their representative is *AActor* [19] class, which inherits from UObject and uses all its standard functions. AActor is a type of UObject that is designed to play a role in the overall gameplay experience. It is in charge of all high-level behaviors in a game, and it is the basis of every object in the scene, including players, enemies, doors, walls, player start location, player camera, etc.

Actors can be manually placed in the scene by a level designer, or they can be created dynamically during runtime via gameplay systems. They can also be destroyed explicitly through gameplay code via C++ or Blueprints, or by the standard garbage collection mechanism when the level is unloaded from memory. All objects that can be placed into a level extend from this class. Even a class like GameMode, which is commonly used to specify game rules, is an actor, though it does not have a "real" position in the world.

This is due to the fact that actors, despite being positioned within the scene, do not truly have their own transform, which is a collection of location, rotation, and scale properties. Actors are responsible not only for their own behaviors but also for the organization of a hierarchy of *actor components* through the process of specialization and composition. Components are essentially a specialized kind of object that can be attached to actors or other components and used to share common behaviors, for example, with other actors. In Unreal Engine, it is possible both to create your own components by inheriting from the corresponding component base class and to use one from a large library of built-in components. For example, a USceneComponent can be used to provide an actor with transformation in the scene, and a UStaticMeshComponent allows a piece of geometry to be used as a sub-object for another actor.

Unreal Engine comes with a variety of different basic actors, all of which are descended from the AActor base class and divided into types. For instance, among the types of gameplay actors that can be highlighted is APlayerStart, which allows the location of the player's spawn point to be specified. However, the most important types of actors in the scope of this chapter are mesh and geometry actors.

AStaticMeshActor [20] is one of the most widely used actors of this type. This actor is able to display a corresponding mesh in the scene by making use of a 3D model that has been saved in an asset file for the project as the basis for its work. It is also important to clarify that this actor has its own transform and that the word "static" in its name refers to the fact that the geometry of a displayed mesh cannot be modified in runtime. This actor also provides many useful features when working with static meshes. Among these features are setup of sockets for attaching to other actors or components, and, importantly, assignment of both simplified and complex collisions.

Collisions are needed to detect or determine the exact position of the intersection or overlap of an object with other objects. They are often used when working with the physics engine, but they can also be useful in conjunction with traces. The point is that Unreal Engine allows to get an instance of an object located in the scene when its collision intersects with one of several possible trace types, including box trace, capsule trace, sphere trace, and, most importantly, line trace. For example, it can be used to highlight objects in the game when the player is looking at them: a line will be traced from the player's camera in the forward direction every frame, and when it intersects with a collision of an object, its color will change.

To apply a specific color to an object, AStaticMeshActor provides a material assignment feature. A material is an asset that can be applied to a mesh to control its visual look. It defines the type of surface an object is made of and gives the ability to adjust its color, emission, opacity, and other properties. When thinking about a material on a high level, it is usually easier to think of it as the "paint" that is applied to a mesh.

6.2 Method description

Taking into account the actor system in Unreal Engine, which was talked about in Section 6.1, a naive method of visualizing graph objects in the scene can be designed. This method entails creating an individual actor for each vertex and edge of graphs added to the scene, providing visualization through a corresponding static mesh, and allowing interaction through a simple and appropriate collision. Also, the standard implementations of AActor and AStaticMeshActor in Unreal Engine can be extended to store attributes of vertices and edges in objects themselves by applying the inheritance mechanism.

In this way, a separate application module can be created that would be responsible both for storing graphs and their objects and for their visualization in the scene. Since a graph does not have a visual part, but instead is just a container with references to objects of its vertices and edges, it can be represented in the scene by an actor inherited from the AActor class. A graph vertex could be represented by an actor that was inherited from the AStaticMeshActor class and had all the necessary attributes for a vertex described in Section 4.1. An edge of a graph could be represented in a similar way. The following model has been created in order to better comprehend this structure, as illustrated in Figure 6.1.

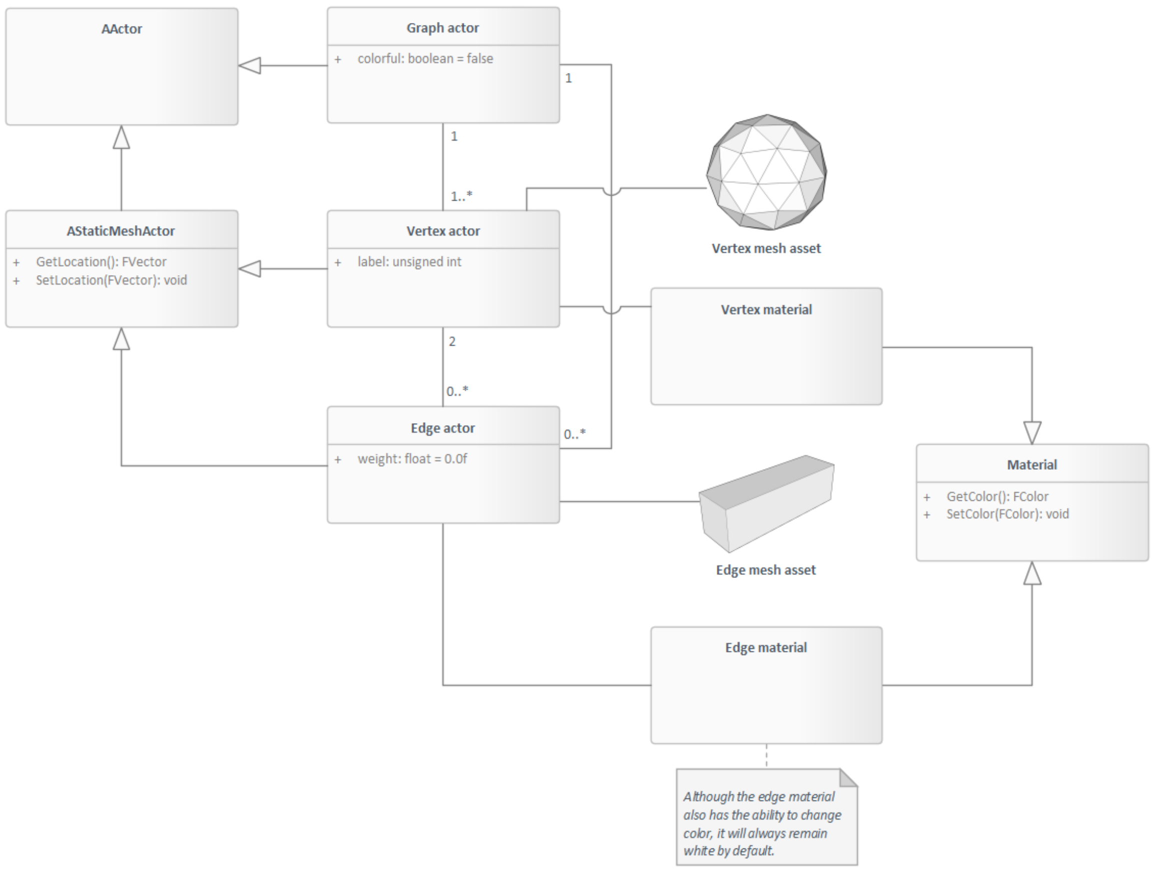


Figure 6.1 Structure of the module for visualizing graphs using the actor system

This model is a highly simplified representation of a graph structure for visualization that could be realized using the Unreal Engine functionality. It is based on the graph representation domain model described in Section 4.1. Note that the "position" and "color" attributes have been removed from the vertex structure because the first of them is already contained in the inherited AStaticMeshActor class, and the second can be obtained from the associated material. Since this approach involves creating a separate actor for each object in the scene, we will end up with a rather large number of actors when importing or creating multiple graphs. Consider, for example, a graph with 20 vertices and 5 edges. When importing this graph, 1 corresponding graph actor will be created in the scene. Then, 20 more actors for each vertex and 5 actors for each edge will be created and subsequently linked with the parent graph actor. In total, 26 actors will be needed to represent this graph in the scene. A number of improvements to this method can also be applied to significantly reduce the number of actors.

For example, each vertex and edge can be represented not by a separate actor but by a component, or, to be more precise, by the UStaticMeshComponent that would be attached to a parent graph actor. In this way, the representation of this graph in the scene would be exactly the same, but with only 1 actor with 25 subcomponents instead of 26 actors. This improvement also prevents manually removing vertices and edges when removing their parent graph actor. Since components cannot exist without an actor, they will be automatically deleted by Unreal Engine when the actor is deleted.

Both of these methods are among the easiest ways to realize the representation of 3D graphs in the scene with the functionality provided by Unreal Engine. They also fit the overall application structure described in Chapter 5 and can be used in conjunction with the user interaction module. However, in both cases, this approach has some disadvantages, which will be discussed in the next section.

6.3 Problems and performance issues

The approach and one of its possible improvements proposed in Section 6.2 are, in essence, simple to implement and perfectly suitable for the visualization of small graphs. However, when it comes to graphs with 1,000 or more vertices and edges, when using this method, we can and we *will* face performance issues. The point is how the static mesh actors and components are actually rendered.

In the development of graphical applications, there is a term such as *draw call*. It stands for CPU/GPU communication and tells how many objects are being drawn to the screen. In essence, a draw call is the CPU activity that prepares drawing resources for the graphics card and comprises all the information needed to inform the GPU about textures, states, shaders, rendering objects, buffers, etc. For the CPU, it is quite "expensive" to translate all of the information previously mentioned into GPU hardware commands.

Each actor and component in the scene produces 1 draw call per frame. What emerges from this is that since a draw call is required for each different material, with a variety of unique actors in the scene and multiple different materials, the number of draw calls raises accordingly. Since CPU work takes time to translate this information into GPU hardware commands, we will encounter performance issues when rendering a number of large graphs with a high number of draw calls involved. So, for example, 10 graphs, each consisting of 1,000 vertices and edges, can be represented as 20,010 actors or 10 actors and 20,000 components. Their rendering implies 20,000 draw calls. Moreover, since the application uses a VR headset for output, the number of draw calls is multiplied by 2 (for each eye), as described in Section 1.1.2. This gives us a total of 40,000 draw calls *each and every frame*. In practice, the number of renderings can be less because the *occlusion culling algorithm* [21] can be applied to the visual objects in the scene. It allows the system to not render objects that are not visible to the user, thereby providing better performance. Nevertheless, one of the main goals to achieve the necessary effectiveness is to reduce the number of draw calls.

It will also be a problem that it is very difficult to work with actors when performing parallel or asynchronous tasks. For example, it is impossible to create actors in Unreal Engine on a separate thread, which could be very useful for asynchronous object deserialization when importing a large graph. This can be solved by using the deferred actor creation method, but this can significantly increase the complexity of the code.

The complexity of the code is also affected by the architecture of the proposed method itself. The problem is that this approach involves both the storage and visualization of objects. As a consequence, this module will lack *maintainability and extensibility.* This leads to high connectivity with the other module of the application and the complexity of fixing existing functionality or adding new functionality.

All of the above problems, as well as many others, will be solved by using a procedural mesh generation method, which will be described in the next chapter.

Chapter 7

Graph visualization: a procedural approach

This chapter examines and describes the part of the application responsible for visualizing graphs using procedural generation. Since object data cannot be stored in the geometry itself, the first section describes the entity system designed to store data. The second section provides the procedural rendering approach and the graph rendering system designed for it, along with improvements to achieve the necessary efficiency in visualizing a variety of large graphs.

The typical way to show something is to render a mesh with a specific material. Unreal Engine has a few built-in meshes of simple shapes, including a cube and a sphere. Other meshes can be purchased, downloaded, or made yourself, and then imported into a project. All of them can be represented in the scene by the corresponding actors or components, which were described in Chapter 6. However, it is also possible to create a mesh on-demand at runtime via code. Such meshes are known as procedural because they are generated via code using specific algorithms, rather than being modeled by hand. This method will provide both an opportunity to merge the geometry of objects in the scene, thereby minimizing the number of draw calls, and a huge scope for ideas and improvements.

7.1 Entity system

In the overall design of the application, it was decided to use separation of concerns and split the representation of graphs in the application into a backend and a frontend. First of all, this is caused by the fact that, with the procedural approach, the geometry created in the scene does not have the ability to store object data. The point is that, unlike the naive approach, which gave each object its own actor in the scene that was in charge of storing and displaying data, the procedural approach involves producing partial or whole geometry without having corresponding objects in the scene. In addition, this partition into modules will allow the application architecture to use separation of concerns, thereby allowing both object data modification and visualization of that data to be performed *separately*. This will lead to a huge number of advantages, among which the extensibility of the system and the possibility of parallelizing the work of its modules can be mentioned.

Therefore, before proceeding with the graph rendering approach, it is necessary to design a system capable of storing and managing object data for subsequent rendering. In addition to storing data, this system must also provide a common and convenient interface for accessing particular graphs, as well as the components of those graphs and their attributes.

Numerous systems have already been designed and developed to perform this task. One of the most popular is an *Entity Component System* [22]. It is an architecture that focuses on data and separates data into components, identities into entities, and behavior into systems. In essence, it treats every object in the world as a unique entity defined by properties it holds. Unfortunately, Unreal Engine does not provide functionality for this system. Also, this system might be great for developing more extensive games and applications, but its use for graph representation purposes may be redundant. This is due to its complexity and the features it provides, many of which will not be needed in our case and will only add overhead in terms of complexity and performance of the resulting application. For these reasons, based on this system, we will design our own.

In this way, a separate module will be created in the application that provides the *entity system*. The use of polymorphism has been initially considered as a way to store multiple types of objects in a single container. Such a solution would allow us to identify all objects using a single number, which is the index of an object in a container. However, this method implies the allocation of each object on a heap and subsequent *runtime type identification* [23], which, with a large number of objects, will significantly affect performance. With this method, we will also lose the efficiency of iterating only the necessary object types, which will be useful for visualization. For example, if we needed to iterate all vertices, we would have to iterate the entire container, which, in addition to vertices, may store dozens of graph and thousands of edge objects.

Therefore, it was decided to split the storage area in the system into three containers. Each of these containers will store objects of a corresponding type from all possible types of objects in the application, that is, graphs, vertices, and edges. This solution will allow us to get rid of the allocation of each object on a heap. This will also allow access to not only a specific object, as in the case of a single container, but also to all objects of a certain type.

After the separation of the storage area, there were two questions: how exactly stored objects would be organized in memory and how identifiers for accessing them would be produced. An operation of accessing an object should have $\mathcal{O}(1)$ complexity and, preferably, be cache-friendly for the CPU, because this operation will be one of the most frequent to be performed. So, for example, when changing a graph position, the corresponding tool from the tool system in *each frame* will access the graph object, get identifiers of all its vertices, then access the data of these vertices by their identifiers, and change their positions accordingly. Initially, the possibility of identifying vertices by their labels was considered. However, this method is not possible because all objects in the application must be *uniquely identified*. Labels of vertices in a graph are unique only within the scope of the graph itself, i.e., two graphs in the application can have vertices with the same labels.

One of the simplest and cache-friendly data structure is *contiguous array*, which places objects in memory one after another and provides access to them with $\mathcal{O}(1)$ complexity. In that case, the object identifier could be a pair of two numbers, the first indicating the object type to determine which of the three arrays to access, and the second being the actual index of the object in the array. Using array data structures is quite possible and easy to implement, however, as long as we do not have to deal with the removal of objects, which would lead to their fragmentation in memory and the subsequent invalidation of their identifiers. In that case, it would be necessary to access each object and update identifiers for other objects in their data, which would be time-consuming with a large number of objects.

The resulting solution will be a data structure called *sparse array* [24]. This structure is the opposite of the aforementioned regular array, which is inherently dense. The point is that objects in a dense array are placed one after another without “gaps” in memory. When an object is removed from the beginning or middle of an array, it shifts objects to the left, which can lead to invalidation of identifiers. A sparse array, on the other hand, has the ability to have “gaps” in memory. This means that when elements are removed from the beginning or the middle, there will be no shifting at all, and an empty space is left which will be taken by new elements added later. Unreal Engine’s implementation of a sparse array is provided by the *TSparseArray* [25] class.

Among the disadvantages of this realization is the higher memory consumption compared to a dense array because it allocates memory for a whole range of indices. So, for example, to add two objects in memory with indices 0 and 50, it will allocate memory for 50 elements. However, this will not happen in our case since we assume the sequential addition of elements. This means that a new element will be added to the first free position of the array, and if there is not one, it will be added to the end. Even though a big "gap" can be left after removing a lot of elements from the beginning or middle of an array, the implementation of TSparseArray allows for fast iteration over objects. The internal implementation of TSparseArray stores elements in a regular dynamic array, which implies the contiguousness of elements in memory. However, with high fragmentation of elements, this contiguousness can be lost, which also leads to a loss of memory cache friendliness. Despite this, maintaining object indices when removing their neighbors saves us from manually updating them in the scope of all modules, which, compared to cache misses, would be much more time-consuming.

Using this data structure will allow us to have $\mathcal{O}(1)$ complexity for all the operations we need. These operations include adding elements to the first free position or to the end, accessing them, and removing them from any place while preserving the validity of identifiers. Figure 7.1 below shows a model of the entity system structure. As an example, the model shows the storage of 2 graphs, 6 vertices, and 4 edges.

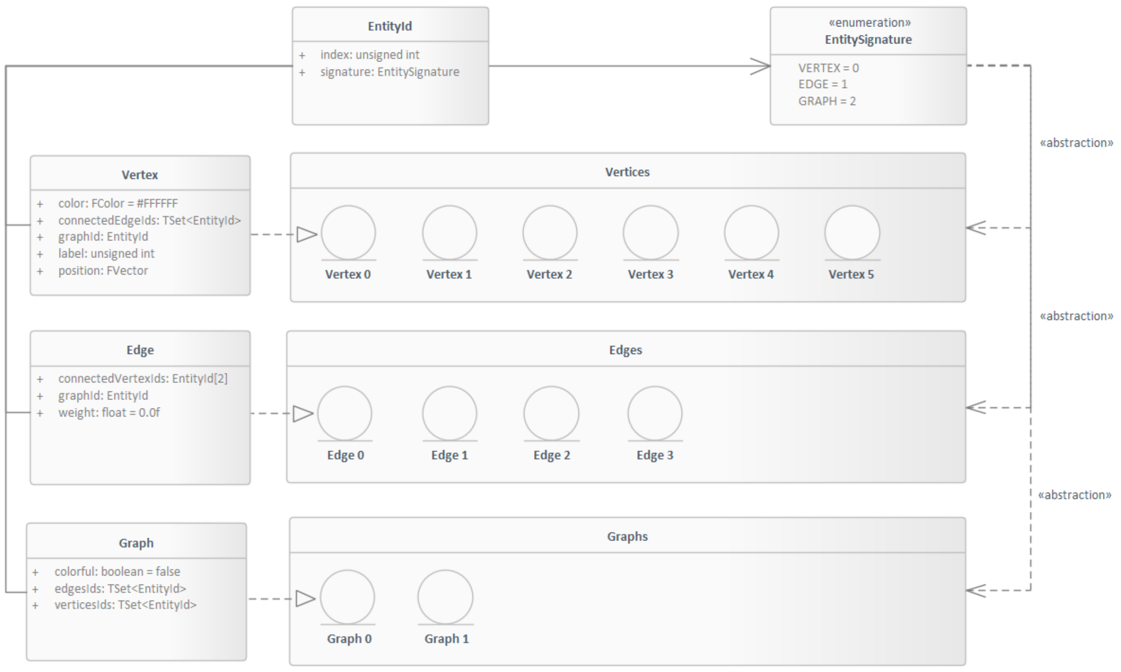


Figure 7.1 Structure of the entity system module

The system will consist of three sparse arrays, each responsible for storing entities of the corresponding type. Each entity is an instance of the corresponding class. Entities in the application will be identified using an *EntityId* structure, which will consist of two elements: an entity signature and an index. The signature is an abstraction for representing the type of entity and can be used to identify both the type of class from which an entity is instantiated and the sparse array in which an entity exists. The index is a number that only indicates where an entity is located within the corresponding sparse array.

In addition to the entity storage structure itself, the class structures representing these entities can also be seen in Figure 7.1. These structures were designed based on the common graph representations in memory discussed in Section 2.2 and the graph representation domain model described in Section 4.1. Similarly to the definition of a graph, its class will consist of two sets. The first set stores identifiers of its vertices entities, and the second set stores identifiers of its edges entities, respectively. This relationship is also represented in the reverse direction, that is, each vertex entity will have an identifier of its parent graph, which is the same as in the case of an edge entity. In addition, a vertex stores a set of identifiers of adjacent edges, and an edge stores two identifiers of vertices that it connects.

With this structure, we have to correctly associate elements with each other when we add new elements and correctly remove their identifiers when they are removed. However, in contrast to this, Unreal Engine's *TSet* [26] class, which is a hash set, allows for fast object iteration and provides addition, deletion, and search operations with $\mathcal{O}(1)$ complexity.

As a result, this entity system structure will be relatively memory-consuming. Despite this, it is designed so that the most frequent operations in the application regarding finding a desired object, adding and removing objects, and iterating over all objects of a desired type would be as fast as possible.

7.2 Graph rendering system

Using the procedural generation approach, we will be able to generate geometry for graphs in fewer draw calls than in the case of the naive approach using the actor system. The point is that with the naive approach described in Chapter 6, each entity in the scene would be represented as an independent actor, which implies a direct proportionality between the number of actors in the scene and the number of draw calls per frame.

Procedural generation, on the other hand, for rendering vertices and edges implies generating meshes using algorithms in code instead of using corresponding models from assets, as in the case of actors. This approach not only provides a huge variety of different ways to render and the possibility to parallelize the rendering process, but also increases the scalability of the entire system. Thus, adding functionality, for example, to draw directed edges implies only adding geometry to the beginning or the end of the edge geometry. However, in the case of the actor system, such an example would be much more difficult to implement. Either the model asset of the edge actor would have to be replaced during runtime, or one or two additional actors would have to be created and attached to the beginning and end of the edge actor.

Before proceeding with the design of the entity rendering module, we have to understand how the geometry rendering process works and what data are passed within the draw call. Unreal Engine provides a separate plugin for procedural generation with a corresponding *component* [27] for this purpose. However, this component is experimental in engine version 4.27 and has relatively high memory consumption and render thread CPU time. For these reasons, we will use a more efficient component that comes with the open-source *RuntimeMeshComponent* [28] plugin. The code for this plugin is available on GitHub [29] under the MIT license. Therefore, the process of rendering geometry that follows will be related to the workflow of this component and may be different in some places from common rendering pipelines in graphics applications.

3D models are nearly commonly represented using *polygonal meshes* [30] in computer graphics applications. In its most basic form, a mesh is made up of a collection of vertices, polygons, and, if desired, a variety of other vertex and polygonal properties. The complexity of a mesh can range from minimal to extremely high depending on factors such as rendering quality, speed, resolution, etc. Graphics and game developers employ a wide range of mesh processing algorithms for a variety of applications, including creating, simplifying, smoothing, remapping, and transforming meshes. A polygonal mesh is a set of vertices and polygonal elements that collectively define a 3D geometric shape. The simplest mesh representation thus consists of a vertex and a polygon lists. Polygons are often defined in terms of triangular elements. Since triangles are always both planar and convex, they can be conveniently used in several geometrical computations such as point inclusion tests, area and normal calculations, and interpolation of vertex attributes.

The RuntimeMeshComponent sends information about the mesh's vertices to the GPU using a vertex buffer that has information about each vertex. An index buffer represents a list of polygons that is laid out as a contiguous list of indices into the vertex buffer, three at a time, representing the three points of a triangle. The front side of each polygon is shown by putting the vertices in the normal direction of the outward face in counterclockwise order. The distinction between the front and back faces of a polygon becomes important in lighting computations and culling operations. For illustration, Figure 7.2 shows the vertex and index buffers for generating a square.

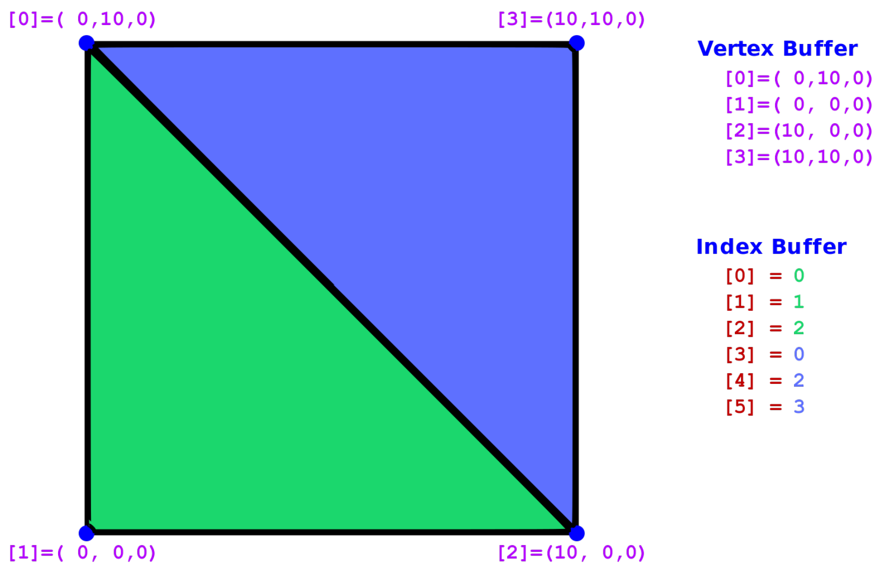


Figure 7.2 Example of square mesh composition [31]

In addition to vertex positions, the vertex buffer also contains additional data, including color, normal, and texture coordinates for each vertex. All of them are also created on the CPU and are then brought to and processed by the GPU when generating meshes. Thus, vertex color can and, in our case, will be used in the corresponding material to colorize graph vertex entities. Vertex normals are used in lighting processing to properly render shadows on meshes. In our case, they will not be used, since the system design carried out in Section 8.1 implies the use of a non-directional light source, which allows us to avoid generating normals for the mesh and, as a consequence, to get rid of shadows. This will improve the performance of both CPU and GPU render threads. The use of texture coordinates will also be discarded since the graph visual design does not imply texture mapping.

To organize geometry rendering and the use of materials, the plugin uses a partitioning system into *levels of detail* [30] and sections. Thus, each component can have up to eight levels, and each of them can be divided into sections with the corresponding materials. Although levels of detail are very useful in reducing the complexity of geometry depending on how far away a mesh is from the camera, this feature will not be used in the scope of this work. A section, in essence, can be thought of as a rendering pipeline for a certain part of the geometry that has one assigned material. This section system has the ability to work in parallel at the CPU level of rendering, which was not possible when using the system of actors. Reasonable use of sections will entail a huge performance gain in terms of both the number of draw calls and the effective parallelization of rendering.

There are several types of collisions in the RuntimeMeshComponent plugin. In this work, a complex type of collision will be used, the generation of which is similar to the geometry generation for meshes. Compared to a simple collision, one of its limitations is its inability to be used in the simulation of physics for an object, but we do not require it anyway. On the other hand, a complex collision can be generated more efficiently. Most importantly, polygons of this collision type have the ability to store indices of the geometry triangles they represent. This will allow us to obtain an identifier of an entity by the intersection of the line trace with one of its geometry triangles, which will be used as the basis for user interaction with entities in the scene.

Although complex collision is made from a triangular mesh, its generation performs in a slightly different way. Its distinguishing feature from geometry is that it is not visible to the user and no materials are applied to it. As a consequence, its creation and processing are done on the CPU without being partitioned into levels of detail and sections, which makes it impossible to parallelize the process of its generation. Despite this, specific optimizations will be applied to reduce the number of collision calculations during the design of the tool system. So, for example, in the case of changing the colors of graph vertices, recalculating their collisions is redundant and will not happen because this operation does not change the geometry topology of a mesh. Their lack of a visual component also saves us from generating colors, normals, and texture coordinates.

7.2.1 Approach description

Now, we may proceed with the design of the system to visualize graph entities. In this way, a separate module will be created that will produce the corresponding geometry in the scene based on data stored in the entity system. It will reflect the entity system as a visual representation with which the user can subsequently interact. The module will provide a *graph rendering system* actor that will work in conjunction with the numerous instances of the URuntimeMeshComponent and the URuntimeMeshProvider classes provided by the plugin.

The URuntimeMeshProvider deals with generating geometry and collisions on the CPU side and then passing them to the assigned URuntimeMeshComponent, whose job is to transfer the necessary data to the GPU to render geometry. This is done by copying the buffers into the *GPU memory* [32] and then clearing them on the CPU side. In the case of collisions, its job is to transfer the collision data to the physics engine, which will autonomously handle their processing, baking, etc. A provider can consist of several sections. Even though sections are usually set up for each material in a mesh, the fact that all graph vertices and edges will use the same material allows us to use a section to render *only a certain part* of a geometry.

The essence of the approach is that each section will contain a certain number of entity identifiers from the entity system. This allows us to distribute the number of sections relative to the number of entities to draw, thus minimizing draw calls. One section produces 1 draw call (2 for VR) per frame. This does not mean, however, that we should use a single section to draw all graphs' vertices and edges using only 1 draw call. When determining the number of sections to draw, it is important to find and consider the balance between CPU and GPU load. With a single section, the load on the GPU will be minimal, which is not the case with the CPU. This will be especially noticeable when the user interacts with at least one of the objects in the scene. The point is that any change to an entity that is rendered in a section causes all other entities in that section to be redrawn. So, in the case of a single section for all entities, changing any object will force a redraw of absolutely all the objects in the scene. With a large number of entities, this would impose a huge and excessive load on the CPU.

There are a variety of possible ways to organize these components and objects, depending on the required functionality and performance level. The following is a step-by-step look at how the module's architecture was designed, as well as improvements that were made to get the required level of performance when rendering multiple large graphs.

7.2.2 Two sections for all entities

Initially, it was decided to split the process of rendering vertices and edges into two corresponding sections. Such a solution will give us the possibility both to render the geometry for all types of entities in parallel and to render only a certain type. Since collisions have no sections, the structure that implies one provider with two sections entails generalizing their collisions. This means that while only one type of entity can be redrawn, both types will have their collisions recalculated once at least one of them is notified that it has been changed. For this reason, it was decided to use two sections, but with two providers instead of one. Both providers will have their own single section, so we have the possibility to redraw the geometry in parallel and to recalculate collisions only for a certain type.

The model shown in Figure 7.3 shows the module structure with this partition. As an example, the entity system stores 2 graphs. The first one consists of 3 vertices and 1 edge. The second consists of 4 vertices and 3 edges. For the model to be compact, the contents of the entity system have been hidden.

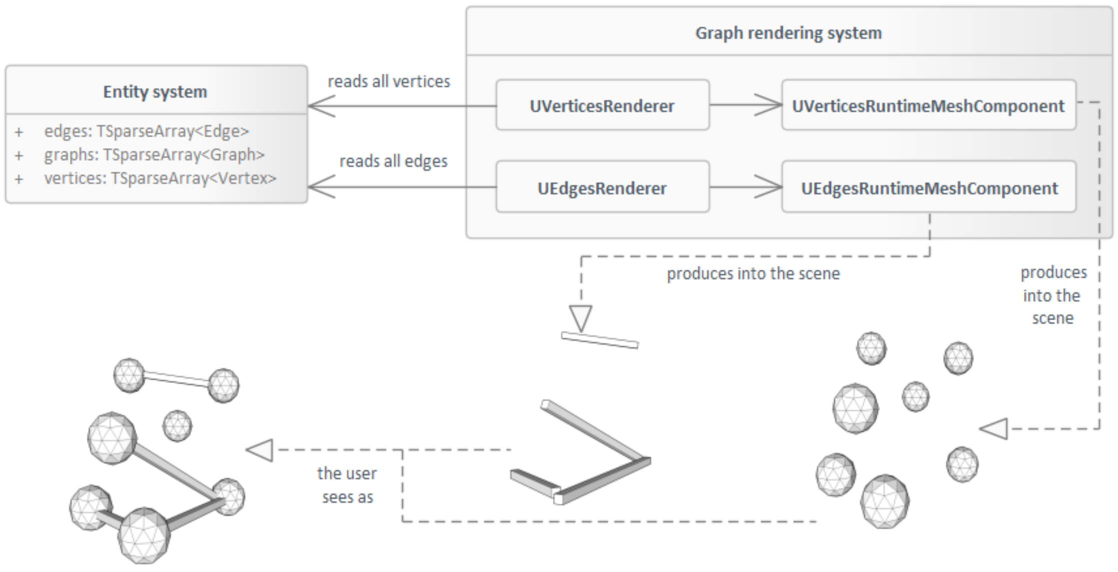


Figure 7.3 Graph visualization: two sections for all entities

The graph rendering system consists of two components and two providers assigned to them for rendering vertices and edges, respectively. To render all vertices, the UVerticesRenderer provider will be used in conjunction with the UVerticesRuntimeMeshComponent. Similarly, the pair UEdgesRenderer and UEdgesRuntimeMeshComponent will be used to draw all the edges.

A provider's job is to get a command to update the geometry or collisions of entities of its type. After receiving the command, the provider reads the data of all entities of the same type and, based on that data and the command, sends the generated data to the assigned component. The component will then send the received data to the GPU memory in the case of geometry updates or to the physics engine in the case of collision updates.

All graph vertices will appear in the scene as a "solid" geometry mesh with icospheres rendered on the CPU and visualized by the GPU during the first draw call. Similarly, square cuboids for the edges of graphs will be generated as a "solid" geometry mesh during the second draw call. Thus, despite the fact that the entire mesh is essentially divided into two parts, the user will perceive the mesh as a composition of vertices and edges presented as separate objects. As a result, this representation matches the visual representation of a graph described in Section 4.2.

Due to the fact that there are two independent sections rendering each type of entity, we are able to update geometry for either one type of entity or for both types simultaneously thanks to section parallelization. Also, thanks to the separation into two providers, we have the ability to update the collision for both entity types and for entities of a particular type. Despite the fact that the number of draw calls will be equal to 2 (4 for VR), when rendering a large number of entities, there will still be a large load on the CPU.

Consider a vertex color update as an example. In essence, this action means that only this vertex needs to be redrawn with a new color. However, when a section is designed to render all entities of the same type, this action will cause the geometry of all vertices to be redrawn, which will be redundant and expensive for the CPU when the number of vertices is large.

The situation worsens when trying to manipulate the position of an edge. This action requires redrawing not only the edge itself but also the vertices connected to it. As well as their collisions. Although we are able to generate geometry for vertices and edges in parallel, this action requires reading the data of all vertices and edges and then rendering them twice each (for geometry and collisions). To minimize this CPU overhead when rendering multiple graphs, a solution described in the next section was adopted.

7.2.3 Two sections for each graph

To reduce the CPU load, it was decided to go further and divide the architecture of the rendering system in proportion to the number of graph entities. Thus, for each graph in the scene, there will be an actor with the same two components and two providers for rendering vertices and edges. The improvement is that each of these actors' providers will only have to deal with rendering the vertices and edges of one associated graph. So, if the user interacts with, say, one vertex of a graph, then all the vertices of just this graph will be drawn again. As a result, the complexity of both rendering geometry and calculating collisions will be reduced.

Complexity will not be limited by the total number of vertices and edges, as it was in the case of two sections for all entities, but only by the number of vertices and edges in a corresponding graph. This will reduce the load on the CPU but increase the load on the GPU due to the higher number of draw calls. If for all entities in 2 sections we constantly had 2 draw calls (4 for VR), now their number would be linear $n \cdot 2$ ($\cdot 2$ for VR), where n is the number of graph entities and 2 denotes the vertices and edge providers. In this way, rendering 10 graphs with any number of vertices and edges will produce 20 draw calls (40 for VR), which is a very acceptable load on the GPU in this case. Also, note that when using this method, we will not lose the possibility of parallelizing sections, since graphs in the scene are independent of each other.

This structure, in addition to improving the distribution of rendering resources among entities within individual graphs, will provide a more unified interface between the graph rendering system and the entity system. Since there is no limit on how many graphs can be loaded into the application, when a graph is imported or created, a corresponding actor with two components and providers will be created at runtime for its graph entity. In the same way, when a graph entity is removed, the actor with all its components will also be removed. This leads to the fact that the graph rendering system will depend directly on the sparse array where the graph entities are stored. To unify the interfaces of the modules, the actors involved in drawing graph entities can be arranged in a sparse array, similar to the method of storing entities in the entity system.

The main idea behind a more unified interface is that we can use *the same identifier* to access graph entity data in the entity system and to send a command to a rendering actor of this graph to update its visualization or collision. Likewise, to redraw vertices or edges, we can access a graph renderer actor using an identifier of a parent graph included in vertex and edge entities and send a command to a provider of an appropriate type. All of this will be possible because a graph rendering actor is created, exists, and is removed along with the graph entity it represents in the scene, which leads to identical populations of the sparse arrays in which they are stored during runtime.

The model shown in Figure 7.4 shows the module structure with this partition. The entity system contains the same 2 graphs, 7 vertices, and 4 edges as in the previous example. For the model to be compact, the containment of vertex and edge entities in the entity system has been hidden.

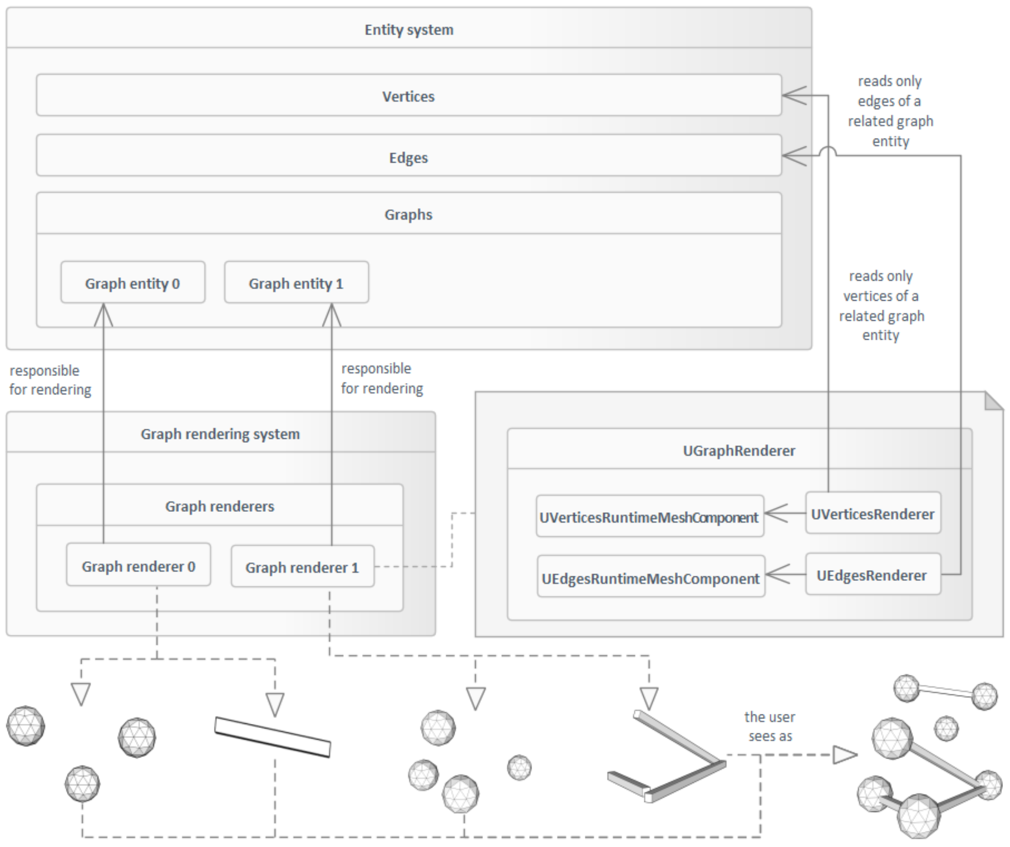

Entity system
Vertices
Edges
Graphs
Graph entity 0
Graph entity 1
reads only edges of a related graph entity
reads only vertices of a related graph entity
responsible for rendering
responsible for rendering
Graph rendering system
Graph renderers
Graph renderer 0
Graph renderer 1
UGraphRenderer
UVerticesRuntimeMeshComponent
UVerticesRenderer
UEdgesRuntimeMeshComponent
UEdgesRenderer
the user sees as

Figure 7.4 Graph visualization: two sections for each graph

In this model, we can see that the graph rendering system for this partitioning consists of a single sparse array of UGraphRenderer actor instances. The number of instances equals the number of graph entities in the entity system, and each instance has the same array index as the graph entity it renders. Each instance of UGraphRenderer contains two providers and two components, which were discussed in the previous section. They are still tasked with drawing all vertices and edges. However, in this case, only all the vertices and edges of a graph entity that their actor holder is related to.

This proposed structure allows for the recalculation of geometry and collisions for multiple graphs and entity types, as well as for a specific type within a specific graph. In addition, when interacting with multiple graphs at once, the types of modified entities will be rendered in parallel. For example, removing an edge from graph entity 0 and a vertex from graph entity 1 will cause three sections to work. In the case of graph 0, the edge section within its actor will be involved. In the case of graph 1, the vertex section within its actor will be involved. It will also involve the edge section because, in this graph, any vertex is the endpoint of some edges. According to the primary operations on graphs, described in Section 2.1, removing a vertex implies removing all edges connected to it. Thanks to the structure of sections in the providers and the plugin's functioning, the work of these sections will occur in parallel.

For comparison, consider rendering 100 graphs, each with 100 vertices and 100 edges. In the case of the previous method, where only two sections were considered, the number of draw calls would have been 2 (4 for VR), and the complexity of rendering geometry or collisions during interaction would have been 10,000, since interaction with a single entity would involve redrawing all entities of the same type. With this structure, the number of draw calls would increase to 100 (200 for VR), and the complexity of rendering would significantly reduce to 100.

In summary, this partitioning contributes to more efficient load balancing between CPU and GPU compared to the structure described in the last section. It implies a relatively small increase in the number of draw calls but provides the ability to render only a graph whose components have been altered. This structure is perfectly suitable for rendering numerous small graphs, such as 100 graphs with 100 vertices and edges each, which implies rendering of 20,000 objects in total. Nevertheless, when rendering the same number of objects but with a different number of graphs and their topologies, we will still face performance problems on the CPU side. So, for example, this structure will be ineffective for the visualization of a graph with 10,000 vertices and 10,000 edges, or for the visualization of 2 graphs, each with 5,000 vertices and 5,000 edges. The latter is one of the non-functional requirements for the application described in Section 3.3.2.

The final and effective solution to be used in the resulting application will be the partitioning described in the next section.

7.2.4 Chunked rendering

The solution, which in result will fulfill the non-functional requirement concerning the graph visualization performance, will be the partitioning of the graph renderers into so-called "chunks." In the graph renderer, a chunk is a block that contains a certain number of vertex and edge identifiers that it will be responsible for rendering. This solution is harder to implement because each chunk will be an actor and must be created, stored, and deleted based on how many vertices and edges are in a graph. However, its complexity is due to the fact that the design of this method emphasizes efficiency in performing time-consuming tasks.

Less time-consuming tasks such as importing or removing a large number of objects from the scene can be optimized by means of parallelization or asynchronization since they are performed only by a user command, such as pressing a trigger. Real-time manipulation of objects is a much more time-consuming task that involves changing the entity's position data relative to the position of the controller, recalculating the geometry, and rendering it afterwards. All of these subtasks have to be handled on every frame. Looking ahead, this solution will provide the necessary efficiency for this and many other tasks.

The point is that the relative complexity of the implementation is compensated by the ability to arbitrarily balance the load on the CPU and GPU. *The number of vertex and edge identifiers in a chunk* (hereinafter "chunk capacity") governs this balance. The chunk capacity is inversely proportional to the number of chunks in the graph and, consequently, inversely proportional to the number of draw calls and directly proportional to the complexity of rendering geometry and collisions. Thus, the smaller the chunk capacity, the more chunks will be created, resulting in more draw calls on the GPU side but less complexity in rendering objects on the CPU side. If the chunk capacity is too small, it will lead to the problem we had with the naive actor system approach, which was discussed in Section 6.3. A chunk capacity that is too high will lead to the situation with the 2-section approach for all vertices and edges, which was discussed in Section 7.2.2. This solution allows us to adjust this balance by changing just a number.

Speaking of efficiency, the complexity of rendering all entities of the same type in a chunk with one draw call will be $\mathcal{O}(k)$, where k is the capacity of the chunk. The total number of draw calls per frame will be $\lceil \frac{v}{k} \rceil + \lceil \frac{e}{k} \rceil$ ($\cdot 2$ for VR), where v and e represent the total number of vertices and edges in the scene. This expression has been obtained based on the fact that each chunk will consist of 2 providers (for vertices and edges, respectively), and each of these providers will render at most k entities of the corresponding type.

In this work, the capacity of a chunk will be *fixed* and set to provide the necessary efficiency to visualize 10,000 vertices and 10,000 edges in VR. This number of components was derived from one of the non-functional requirements described in Section 3.3.2. Figure 7.5 below shows a graph showing the dependence of CPU and GPU load on the chunk's capacity k.

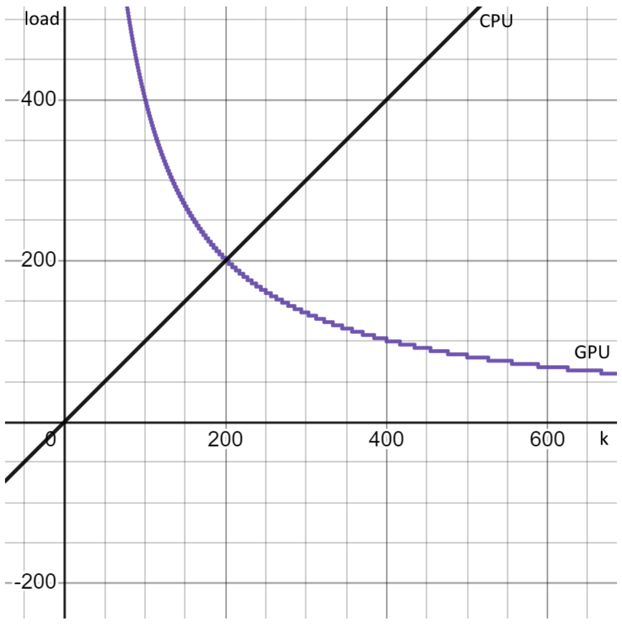


Figure 7.5 Dependence between chunk capacity and CPU and GPU load

The black color represents the function $y = k$. This denotes the relationship between k and the complexity of rendering all entities of the same type on the CPU side. The purple color represents the function $y = 2 \cdot (\lceil \frac{10000}{k} \rceil + \lceil \frac{10000}{k} \rceil)$. This denotes the relationship between k and the number of draw calls per frame when drawing 10,000 vertices and 10,000 edges on the GPU side in VR.

The graph demonstrates that as k decreases, the less the load is on the CPU and the more it is on the GPU. Correspondingly, the higher the k, the greater the load on the CPU and the lower the load on the GPU. It can be established that the required rendering performance will be achieved with an approximate equality of load between CPU and GPU. At $k = 200$, the graph indicates that the CPU and the GPU will be loaded equally. Thus, the capacity of a chunk will be fixed at **200**, which assumes that a chunk renders at most 200 vertices and 200 edges. The rendering efficiency achieved by using this value will be tested during the application performance testing phase provided in Section 10.2.

It should be noted that the data shown in the graph are only speculative, and the identification of the load functions on the CPU and GPU was made *without* taking into account the load on the CPU in the preparation of draw calls. This means that in practice, with a large number of draw calls, a high load may be observed not only on the GPU side but also on the CPU side.

In addition to its high efficiency and many positive qualities inherited from the previous two approaches, this solution also provides a number of possible ideas and improvements that could be implemented in the future. Among them is the use of chunks with variable capacity, which will allow the optimal load balance to be adjusted in runtime based on the number of vertices and edges in the scene. Also, the number of draws can be reduced with the occlusion culling algorithm. This will be possible due to the proper distribution of graph vertices and edges into chunks. A chunk will form a bounding box consisting of vertices and edges that it renders, which will later be used by the algorithm to draw only those chunks that are visible to the user. For the algorithm to work efficiently, it is necessary to distribute entities into chunks so that each chunk forms a bounding box as compactly as possible.

In this work, however, this algorithm will not necessarily be used in the most effective way, and the graph vertices and edges will be distributed into chunks *as they are added to the graph*. This will considerably simplify the process of creating new chunks and adding identifiers to them.

Summarizing all of the previously described aspects of graph rendering and entity systems, the work of such a set of modules will enable the efficient addition, removal, processing, and visualization of a large number of graphs with many vertices and edges.

Figure 7.6 shows a detailed model of how these modules work together. For clarity, the entity system stores two graphs: $G_0 = (\{0, 1, 2\}, \{(0, 1)\})$ and $G_1 = (\{3, 4, 5, 6\}, \{(4, 5), (5, 6), (3, 6)\})$. According to this, the graph rendering system has two graph renderers, each of which consists of chunks. It is also worth noting that the capacity of the chunks in this example is 2, which implies that each chunk is responsible for rendering at most 2 vertices and 2 edges of the corresponding graph.

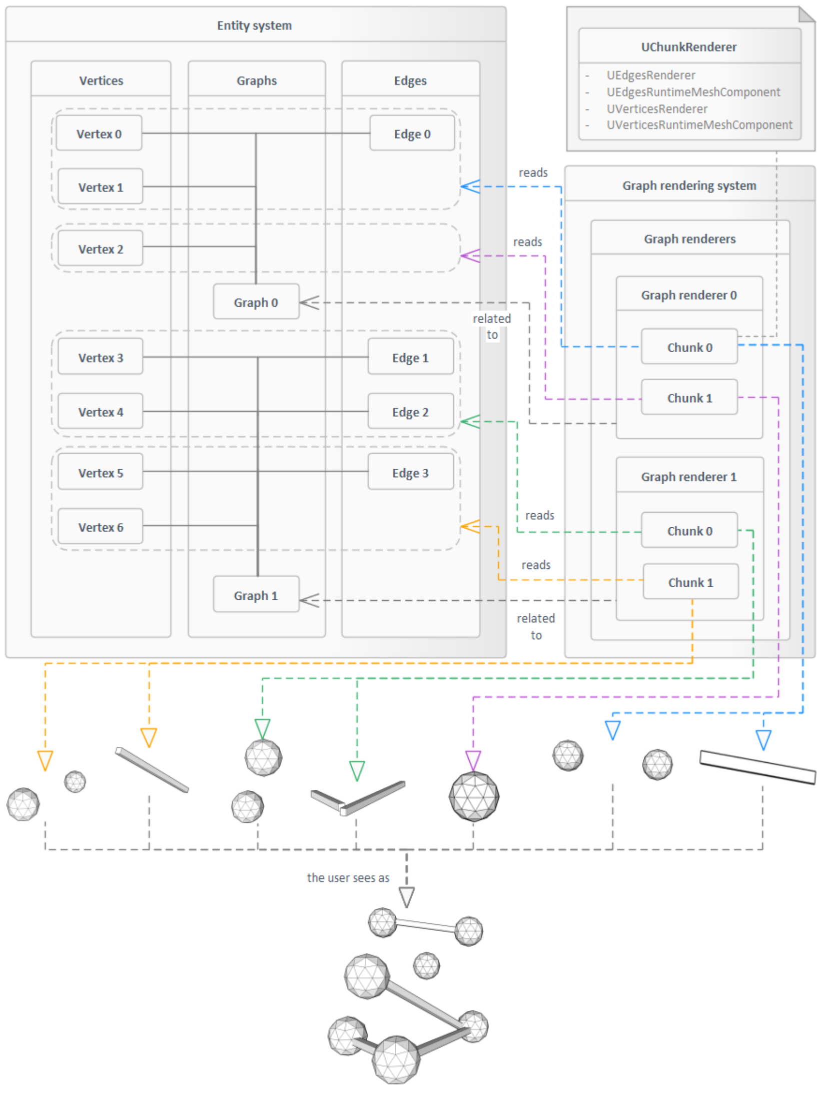


Figure 7.6 Collaboration of the entity system and the graph rendering system to represent two graphs

Chapter 8

User interaction with the application

This chapter aims to describe the designed solution regarding the interaction of the user with the application. The first section covers the design of the scene, which in essence will be a workspace for the user. The second section describes the structure of the corresponding module and its components. The third section aims at designing the user interface. The fourth section discusses the designed tool system for working with graphs in the scene.

8.1 Scene

A graph, whether it was created locally or imported from an external file, must be visualized within the application. Because it intends to make use of VR technology and the positions of the graph's vertices are established in 3D, the application will need to present the user with a 3D environment.

One of the most important concepts in game development is a *game world*. In Unreal Engine, a world serves as a repository for all of the individual game levels that come together to form the final game. It is also responsible for the streaming of levels and the generation of dynamic game elements. A gaming area designed to contain and manage everything a player may see and interact with, such as geometry, objects, particles, and so on, is commonly referred to as a *game level*. In essence, the only game level of the application will be the 3D scene that has been mentioned numerous times before.

The scene itself does not require architectural design, as its creation and deletion, as well as the removal of all objects contained in it when closing the application, will be automatically performed by the functionality of Unreal Engine. It is also important to note that, since the specification of the assignment does not require the declaration of scene sizes or the management of objects that are located outside its boundaries, Unreal Engine will also be responsible for providing this functionality on its own internally. However, it is worth considering everything about the visual representation of the scene itself.

As is the case with the X and Y components of the positions of the graph's vertices, the Z component is not constrained by anything. Consequently, the Z-axis of the scene in which graphs will be located will not be bounded either, for example, by a ground. This indicates that it is vital to provide an appropriate background for the scene in order to effectively immerse the user in the application. To simulate that the user is located in an "unconstrained" environment in which they can work with graphs, it was decided to use a space setting. Thus, the scene will be presented as an empty space with a black background and images of stars on it.

In the process of designing the scene, lighting is also worth considering. It will be supplied by a stationary diffused light source that will match the designed space environment and also have an impact on the overall performance of the application. The fact is that by using this undirectional type of lighting, we have the opportunity to completely eliminate rendering shadows at vertices and edges of graphs. This will free up CPU and GPU resources, which will affect performance in a better way.

8.2 User representation

A separate *Player module* will be provided to represent the user in the scene and provide all possible functionality regarding working with a graph and moving around the scene. This module will consist of a set of actors and components responsible for the corresponding tasks. It is important to note that this module and all its components were designed based on the functional and non-functional requirements specified in Section 3.3 and the use cases detailed in Section 3.4. Further in this chapter we will consider the most important components of the module, which include their structures and workflow.

8.2.1 Pawn

One of the main components of this module is the *VRPawn*, inherited from the *APawn actor* [33] provided by Unreal Engine. Its structure and workflow are shown below in Figure 8.1.

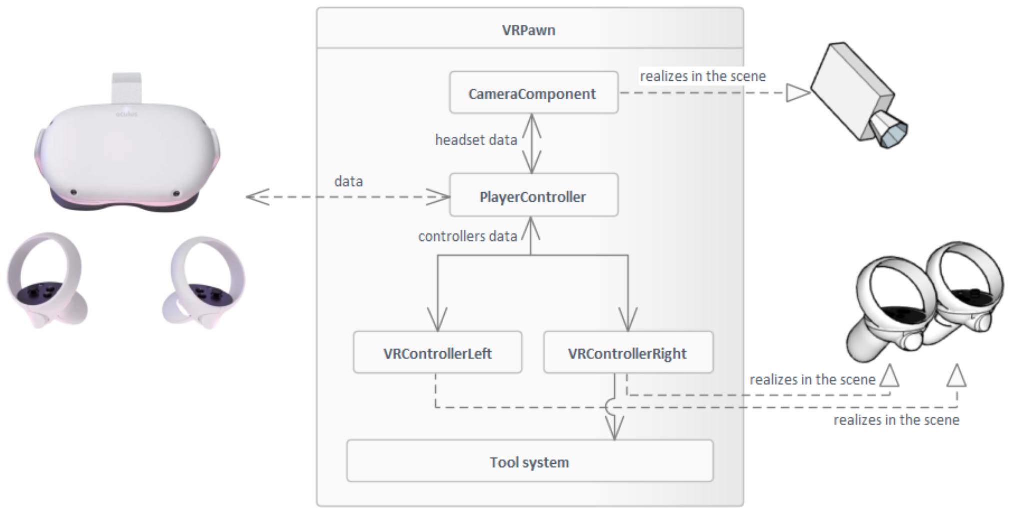


Figure 8.1 Structure and workflow of the VRPawn actor

This actor will be responsible for representing the user in the scene. This involves not only the visual representation but also the interaction of the user with both the scene and the elements in it. In essence, it will be the fundamental to which the other various components will be attached. The core of its work will be the *PlayerController actor* [34], also provided by Unreal Engine. Among the many PlayerController's responsibilities are to accept both input from devices, and commands to change the transformation of the user in the scene. So, for example, when it receives a certain position or rotation, it is able to move or rotate the parent pawn in the scene accordingly. This will make it possible to fulfill one of the functional requirements regarding user teleportation and rotation.

Concerning the input data, PlayerController is able to receive information from input devices and transform it into actions depending on the defined input mappings. In our case, the input devices will be the VR helmet and the controllers of Oculus Quest 2. Therefore, the Oculus OpenXR plugin will be used to process their inputs and transform them into specific actions, as described in Section 1.3. As an example, suppose that we have a "RightTriggerPressed" action defined in the input mappings whenever the trigger press factor of the right controller is at its maximum value. For each frame, PlayerController will receive a current trigger press coefficient from the real device. As soon as the coefficient reaches its maximum value, the "RightTriggerPressed" action will be created and passed to the parent pawn, to which we are subsequently able to react.

In addition to receiving input and creating actions, PlayerController also receives data about the positions of the VR headset and controllers. Then, based on the source of these data, it will send them to the following three components: CameraComponent, VRControllerLeft, and VRControllerRight. It was decided to turn these objects into components (they will be inherited from USceneComponent) for several reasons. First, it will improve the cohesion of their work. Second, this structure makes it clear that neither the camera nor the controllers can exist without a parent pawn, which makes sense. And third, most importantly, their positions and rotations will be calculated *relative to the pawn*. Since the user can not only rotate his head and arms, but also move around the room, the camera and the controllers will have their own position and rotation in the scene. Moreover, their relativity will preserve their transformation during teleportation or rotation of the parent pawn.

Unreal Engine's *CameraComponent* [35] is a component that is in charge of representing the user's point of view within the application. In essence, this component in the scene will be a reflection of the physical VR helmet, thereby being the user's "eyes." The user's "hands" will be the VRControllerLeft and VRControllerRight components. Thanks to the Oculus OpenXR plugin, which was described in Section 1.3.2, they will appear in the scene as Oculus Quest 2 controller models, reflecting the user's physical controllers. The design of these components and a more detailed description of them will be provided in the next two sections. The tool system associated with the right controller component will be presented later in Section 8.4.

It should also be noted that in the model shown in Figure 8.1, the connections between the physical VR devices, the PlayerController component, and the camera and controller components are bidirectional. This is because PlayerController is capable of not only receiving data from the devices but also sending data. In fact, the physical headset as well as the controllers are both input and output devices. Thus, the data containing the output image will be sent from the CameraComponent to the physical headset through PlayerController. The output data for the controllers can contain, for example, vibration, which can improve the user's tactile sensations and immersion in the application.

8.2.2 Right controller

The right controller will be considered the primary one, and thanks to it, the user will be able to *designate* with which objects they want to interact. As demonstrated in Figure 8.2, this will be accomplished by directing the laser to the appropriate object in the scene. Thus, every frame a blue beam will be emitted from the front of the controller. Due to practical constraints, the beam's *maximum* length will be capped at *15,000 unreal units*, which is equivalent to 150 meters. This length has been designated as "maximum" due to the beam's ability to alter it. To provide more immersiveness, the user should feel as though they are interacting with "physical" objects in the scene. This requires that the beam do not go through anything it is directed at. Simply put, the beam will "shoot" out as far as possible until it collides with the target, at which point its length will be reduced to the distance between the controller and the target. The implementation of the beam itself will not contain any internal logic since its task is essentially just to be a visual representation of where and what the user's controller is pointing at.

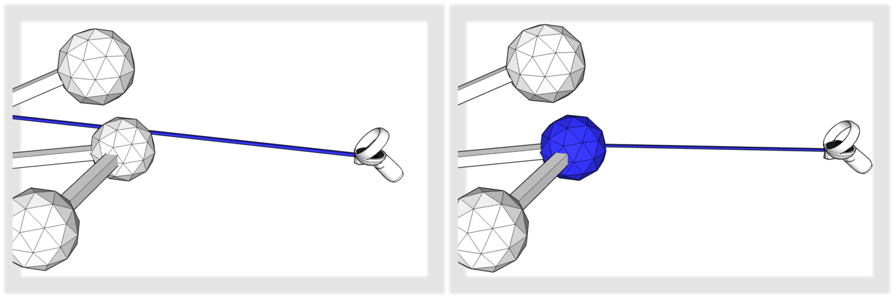


Figure 8.2 The use of a laser to identify an entity

In detail, the task of the controller will be to use Unreal Engine's built-in features to produce a *line trace*. The line-tracing process is represented by "shooting" out an invisible ray, which will detect geometry between two points and, if geometry is hit, return hit result data. Both the beam and its accompanying trace will be continuously and uniformly projected in the same direction for each frame. To minimize potential mistakes when working with floating-point numbers when the distance between the trace end and the object collision is exceedingly small, the trace length will be constant and equal to the maximum beam length.

The hit result data contains a lot of useful information both about the tracing process and about the collision of the object that collided with it. It includes an impact point, a reflection normal, a collision triangle index, and much more. Only the information regarding the point of impact, including the position in the 3D scene, will be essential in the scope of the controller's activity. It will be used to determine the distance between the controller and the object, allowing the beam length to be adjusted accordingly.

In this way, the user will be able to designate the vertices and edges in the scene with which they want to interact. However, this will not be enough to meet all functional requirements since some use cases require interaction with the entire graph, and not just a single vertex or edge. Since a graph entity itself does not have a visual representation, to solve this problem, it was decided to design a so-called *selection modes* model, whose operation is shown in Figure 8.3.

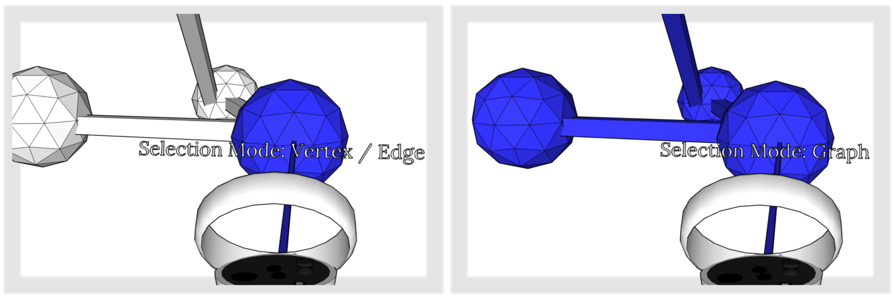


Figure 8.3 Model of selection modes

This model will be activated by pressing and holding the controller's grip and altered by tilting the thumbstick left or right. It will consist of two modes: a vertex/edge and a graph. These modes will be represented by a circular buffer and will be used to identify exactly with which type of entity the user wants to interact. The default value will be "vertex/edge," whereby pointing the laser at a vertex or edge will cause that entity to be highlighted. When set to "graph," hovering the laser over a vertex or an edge will select all components of the parent graph of that entity. This was designed with a bias toward the fact that when the user is pointing at a component of a graph, it is impossible to uniquely identify what they want to interact with: a component of a graph or the graph itself.

Also, the right controller will be used to interact with the UI, which will be described in Section 8.3. This will be possible thanks to the data from the line trace, which also contains the type of actor it collided with. By this type, we will be able to determine when the user interacts with an entity mesh from a provider in the scene or with the UI itself, in which case the device input data received by PlayerController will be redirected by the right controller to the UI.

In summary, it is important to note that the controller itself will not be engaged in retrieving data about a hit entity from the entity system or highlighting entities in blue, as shown in Figures 8.2 and 8.3. This was mentioned and shown only for a more convenient and clear illustration of its work. The controller's tasks will only be to manage the laser, retrieve hit result data from the line trace, and provide the ability to change the selection mode model that is attached to it. All of these data will be sent to the tool system described in Section 8.4. Only then will the tool system be able to use these data to actually work with entities and highlight them.

8.2.3 Left controller

The left controller will be considered the secondary one. Its main task is to provide functionality regarding changing the user's transformation in the scene. The controller thumbstick will be used to *rotate* the user around their axis. Turning the stick to the left or right will turn the pawn in the corresponding direction by 45 degrees, as indicated in one of the use cases. Before the actual rotation, a fading effect will be applied to the camera to prevent the user from experiencing any discomfort due to possible motion sickness. Additionally, a vibration of the controller will occur in response to the user's rotation.

In addition to rotating, the user will be able to *teleport* to the location indicated by the controller for further exploration. A user-friendly method to indicate the desired teleportation destination in a 3D space is essential to achieve this goal. This will be the movement of the teleportation preview in relation to the controller. To put it simply, a sphere will be attached to the end of the laser that is emitted from the controller and will serve as a pointer for where the user will be teleported. This teleportation method is depicted in Figure 8.4 for simplicity of comprehension.

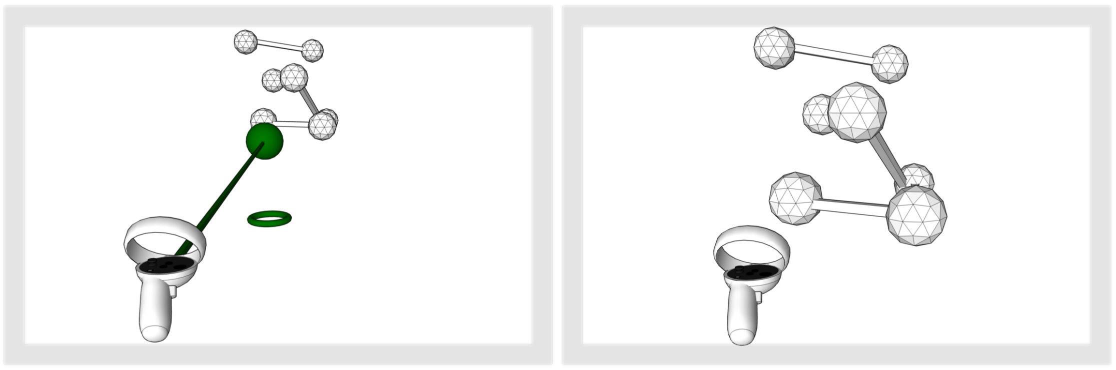
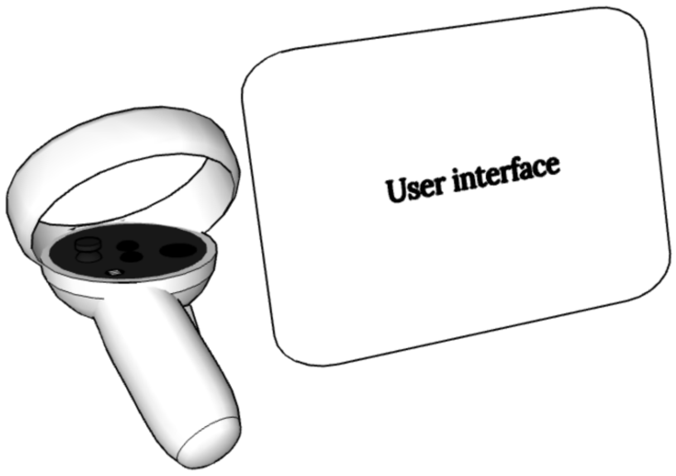


Figure 8.4 Teleportation using controller-based position indication

In order to avoid distracting the user while they are using the application, the preview should not be visible at all times. Thus, it will not be visible unless the user enters a *movement state* by pressing and holding the controller grip. The movement state exposes the capacity to employ the vertical deflection axis of the thumbstick, in addition to preserving the ability to rotate through a horizontal deflection of the stick. By tilting the stick forward and backward, the user can adjust the distance to the spot by pushing or pulling the preview. Pressing the trigger while the controller is in the movement state will initiate the teleportation action, which, like the rotation action, will be followed by the fading camera effect and the controller vibration.

The second, but no less important, functionality of the left controller is to be the "backbone" to which the UI frame is attached. The UI frame and the left controller are components of the system that provide visual representations. Thanks to this, we can take advantage of the Unreal Engine functionality of attaching components to each other, which will allow us to get rid of manually updating the frame position relative to the controller position. In this way, the frame will always appear to the right of the controller, as shown below in Figure 8.5.



■ **Figure 8.5** Attachment of the user interface to the left controller

It is important to note that the UI will be constantly visible to the user, and its presence in Figure 8.4 has been hidden since the main focus has been on the preview of the teleportation. The design of the UI itself will be provided in the next section.

8.3 User interface

One of the most important and integral parts of the application is the user interface. The UI presented as a window attached to the left hand in the application could potentially provide a unique and immersive way for users to interact with the application. This type of UI could allow users to access and manipulate various features and functions of the application in a more natural and intuitive way, as it would be similar to using their own hand to perform tasks.

It is important to consider common UI/UX principles when designing any user interface. In the case of the UI presented in the VR application, it is important to consider both general UI/UX principles and principles specific to VR. Some of the general principles to consider include:

Simplicity: Keep the interface simple and easy to understand, with clear and concise labels and instructions.

Consistency: Use consistent design elements and patterns throughout the interface, such as color, typography, and layout.

Clarity: Make sure the interface is easy to read and understand, with high contrast and legible text.

Functionality: Ensure that the interface is functional and easy to use, with intuitive controls and feedback.

User comfort: Consider the user's comfort and ensure that the interface does not cause any discomfort or motion sickness.

By following these principles, we will be able to create a UI that is intuitive, user-friendly, and visually cohesive, and that does not distract the user from interacting with the graph objects in the scene. This can help to enhance the overall user experience and ensure that the interface is effective and enjoyable to use.

The structure of the designed UI will consist of three windows, as shown in Figure 8.6.

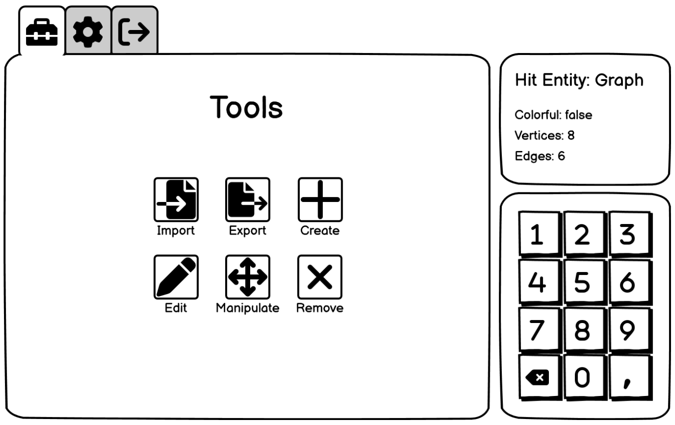


Figure 8.6 User interface: overall structure

The information window, located in the top right, is designed to provide the user with information about the entity to which their right controller is directed. This window is necessary because it is technically more difficult to display information about an entity in the form of an object within the scene. The contents of the window will change depending on the type of entity to which the controller is directed. For example, if the controller is directed at a graph, the window will display the number of vertices and edges in the graph, as well as whether it is colorful or not. If the controller is directed at a vertex, the window will display the label and color hex code of the vertex. If the controller is directed at an edge, the window will display the labels of the vertices it connects and the value of its weight.

The keyboard window, located in the bottom right, is a type of interaction element that will be used to enter a floating-point number. It was necessary due to the limitations of VR devices, as described in Section 1.1.4. This window will typically be hidden, and will only be shown when the user changes the weight of an edge in a graph.

The main window, located on the left, is the primary element of user interaction with both the application and the tool system. It will consist of three tabs: tools, settings, and exit. The settings and exit tabs are secondary, and their contents are shown in Figure 8.7.

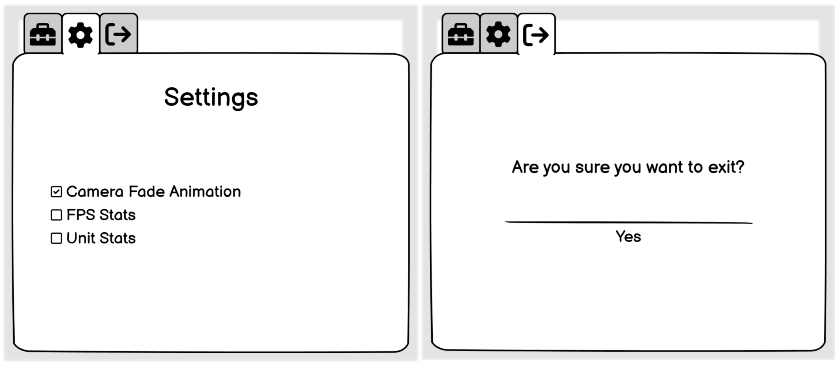


Figure 8.7 User interface: secondary tabs

The settings tab, located on the left, consists of three checkboxes. The first checkbox, "Camera Fade Animation," is used to enable and disable the smooth camera fade animation when rotating and teleporting the user. It will be enabled by default, but disabling it may be useful for users with stronger vestibular apparatus to speed up movement through the scene without the fade animation. The other two checkboxes, "FPS Stats" and "Unit Stats," are provided to display technical information about the application's functioning, and are not necessary for the average user. These checkboxes will be responsible for turning on/off the display of FPS on the screen and information about the rendered geometry, including the number of polygons on the screen and the number of draw calls. It is important to note that these settings will be saved between launches and exits of the application, thanks to the corresponding Unreal Engine functionality by using the internal configuration .ini files.

The exit tab, located on the right, serves to close the application. It's panel is presented as a confirmation window to ensure that the user is sure they want to close the application. The "Yes" button immediately closes the application without preserving the work done in the scene.

The tools tab, shown in Figure 8.8, is considered to be the primary tab.

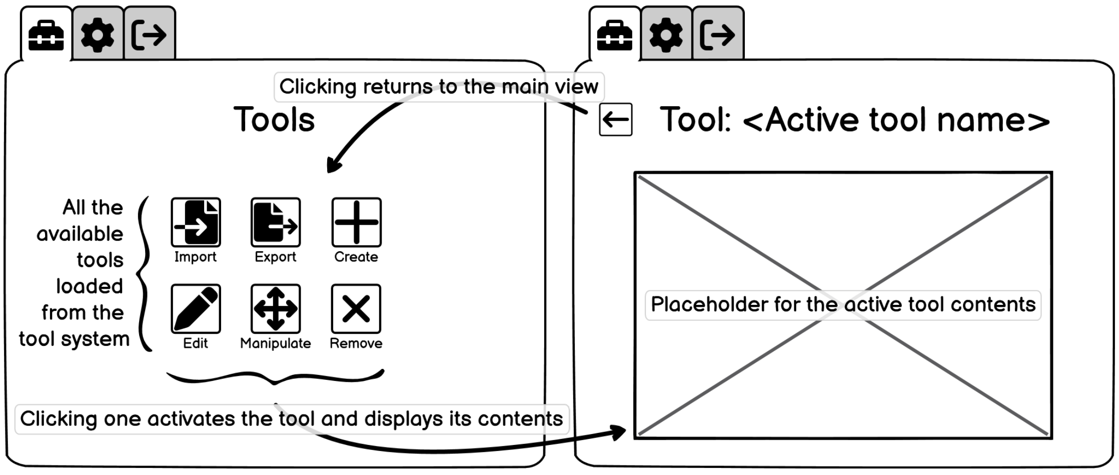

Clicking returns to the main view
Tools
Tool: <Active tool name>
All the available tools loaded from the tool system
Import
Export
Create
Edit
Manipulate
Remove
Placeholder for the active tool contents
Clicking one activates the tool and displays its contents

Figure 8.8 User interface: tools panel functioning

The tools panel in the main UI window is designed to work in conjunction with the tool system and display all the tools available for use, as shown on the left. The panel will contain six buttons, each representing a different tool. When the user clicks on one of these buttons, it will set the corresponding tool in the tool system as "active." After clicking the button, the panel will change its contents to the active tool window, as shown on the right. Pressing the "Back" button deactivates the tool and returns to the main tools panel.

The active tool window will provide various interaction items, such as buttons, lists, and selectors, which the user can use to customize the functionality of the tool. The contents of the active tool window will be directly related to the functionality provided by the tool. A detailed description of the UI of each tool window, along with their workflows, is provided in Appendix A for a more in-depth examination.

The tools panel itself will not contain any predefined tool buttons or their windows. Instead, it will function as a placeholder for the corresponding windows and elements. This means that the tool buttons for selecting the active tool will be added to the panel at runtime when the application is started. This is done by accessing all the tools in the tool system and reading their data, which includes the name of the tool and the icon that will be displayed for it. The data of the tool itself will also contain a link to an asset with its pre-designed window. Thus, clicking on one of the tools will cause the buttons of all the tools to be hidden and show the window with the contents of the active tool.

This structure is much more complex to implement than creating a predefined hierarchy of windows in a single asset. However, it will significantly increase the extensibility of the system and the convenience of adding new functionality. To add a new tool, it will be enough to implement the tool itself, link it to the icon and its window asset, and add it to the tool system. The panel will "automatically" display a button for the new tool and show its window when it is activated by pressing the button. This allows the panel to be easily updated with new tools as they are added to the tool system.

8.4 Tool system

The tool system is an integral part of the application, responsible for providing a range of tools that the user can select and activate to perform various tasks and operations within the virtual environment. The tool system serves as a conduit between the user and the application functionality, receiving and processing data from the user's actions and translating them into appropriate commands for the entity system and the graph rendering system.

The tool system consists of many tools, each representing a different graph operation functionality. These tools include: import graph, export graph, create graph, edit attributes of entities, manipulate positions of entities, and remove entities from the scene. The functionality of these tools was defined during the definition of the functional requirements described in Section 3.3.1 and most of the application use cases described in Section 3.4.

Figure 8.9 shows the structure of the tool system as well as its operation and connection with the right pawn controller and the entity and graph rendering systems.

To put it simply, the job of the tool system is to provide all the available tools on the UI and allow the user to select one of them, thereby activating the tool. Once a tool is activated, all the data received from the right pawn controller will be filtered and processed by the tool system for the active tool. Based on this data and the functionality it provides, the active tool will send appropriate commands to the entity system to change their data accordingly and request the graph rendering system to redraw these entities to update their geometry in the scene according to the altered data in the entity system.

Going into detail, the tool system will receive data from the right pawn controller every frame. These data consist of physical controller input data, as well as hit result data, which contain information about the geometry of the entity pointed to by the controller laser and the currently set selection mode. Based on these data, the tool system can obtain the identifier of the entity at which the user is pointing with the controller.

To ensure that the active tool is capable of performing the desired operation, each tool has a definable set of entity types that it can work with. For example, a manipulator tool can work with any type of entity, while the export tool only works with graph entities, since it is not possible to export a single vertex or edge. The import tool does not work with any specific entity, since all interaction with this tool is done through the user interface by selecting a file from a list.

When the tool system obtains an entity identifier, it checks if the entity's type is supported by the active tool. If it is, the system passes the controller data and the entity identifier to the active tool. In addition, it will also highlight this entity in the scene in blue. The highlight color is used to visually indicate that the entity is being interacted with, as shown in Figures 8.2 and 8.3. Also, the information window will be updated so that it shows the information about the selected entity.

The active tool processes the received data and sends appropriate commands to the entity system and the graph rendering system to perform the desired operation. This may involve changing the attributes or position of the entity, or removing it from the scene entirely. The entity system and the graph rendering system are responsible for updating the data and geometry of the entities in the scene according to the commands received from the active tool.

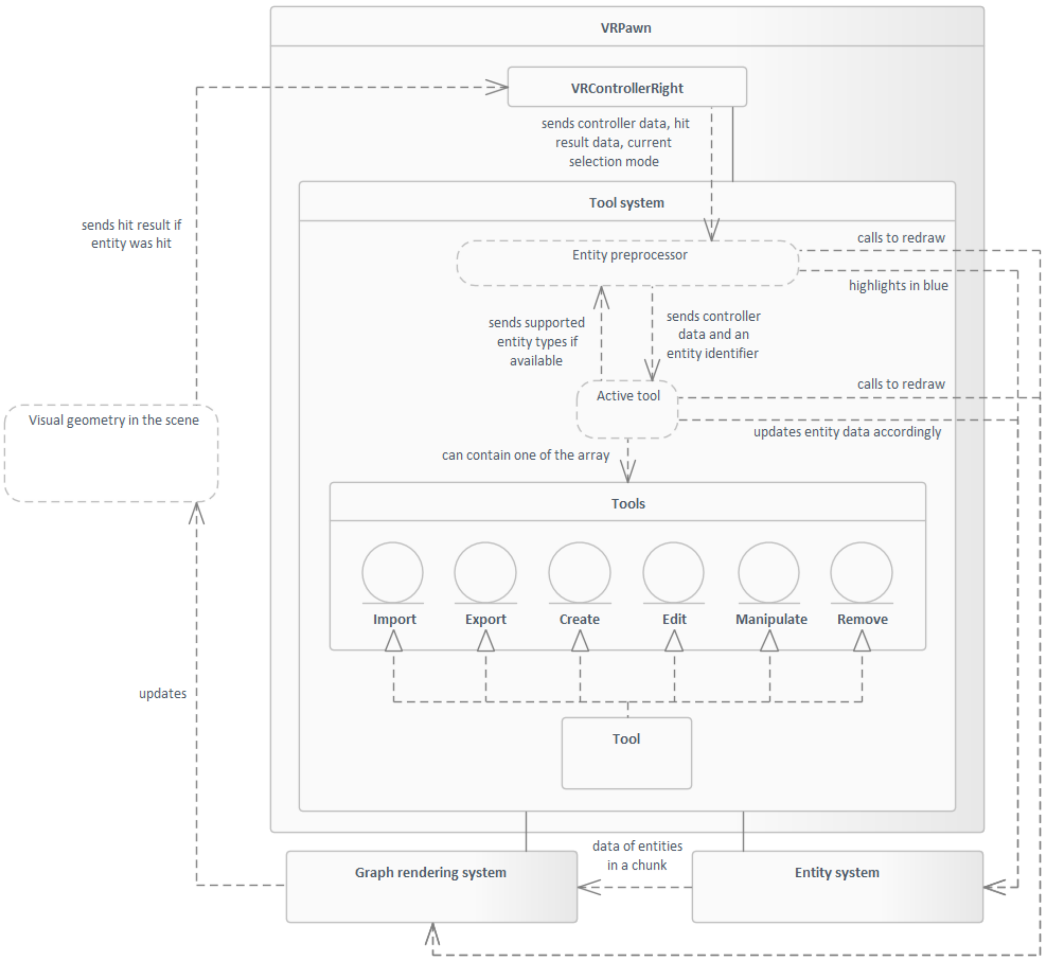

VRPawn
VRControllerRight
sends controller data, hit result data, current selection mode
Tool system
sends hit result if entity was hit
Entity preprocessor
calls to redraw
highlights in blue
sends supported entity types if available
sends controller data and an entity identifier
calls to redraw
Visual geometry in the scene
Active tool
updates entity data accordingly
can contain one of the array
Tools
Import
Export
Create
Edit
Manipulate
Remove
updates
Tool
data of entities in a chunk
Graph rendering system
Entity system

Figure 8.9 Structure and workflow of the tool system

Each tool in the system is designed to be intuitive and easy to use, with clear and concise labels and instructions. Tools are organized into a hierarchy, with all available tools being grouped together. This organization helps to keep the interface organized and easy to navigate, and allows the user to quickly and easily access the tools they need. A detailed description of the UI of each tool window, along with their workflows, is provided in Appendix A for a more in-depth examination.

It is also important to note that while no specific operation is being performed, the tool system will simply highlight the selected entity in the scene to indicate that it is being viewed. It means that even when no tool is active, the tool system will continue to process data from the right pawn controller. This allows the user to view information about the selected entity in the information window without the need to activate a specific tool. To facilitate this, the tool system will not filter the entity type received as a result of data processing, and will accept any entity type that is received.

As a result, this tool system was designed with extensibility in mind, making it easy to add new functionality to the system in the future. By implementing new tools and adding them to the tool system, it is possible to expand the range of operations that the application can perform. This allows the application to remain flexible and adaptable as the needs of the user change over time.

Chapter 9

Implementation

This chapter presents the implementation of a graph editing and viewing application in virtual reality, including the implementation of the scene and user pawn, the visualization of graphs, and the tool system and user interface. The scene and user pawn provide the virtual environment for the application, while the graph visualization implements the rendering and interaction with graph entities in the scene. The tool system and user interface enable the user to perform various operations on the graphs, such as creating, deleting, and modifying vertices and edges. The chapter also describes the challenges and solutions encountered during the implementation process.

9.1 Preface

Prior to beginning the implementation process, it was critical to carefully consider which version of Unreal Engine to use, as well as the desired development style and C++ version. After careful consideration, it was decided to use *Unreal Engine 4.27* for this work. This particular version was chosen due to the fact that the author of this work has significantly more experience with this version, as well as the fact that newer versions such as 5 and 5.1 include features (such as Lumen and Nanite) that are not necessary for the scope of this project.

In addition to the game engine version, it was also necessary to decide on the C++ version and development style to be used. It was decided to develop the application *entirely* in C++ without using the Blueprints visual scripting tool. This decision was made due to the advantages of using C++ described in Section 1.2.2, including the ability to have all of the application logic in one place as well as the ability to easily view and modify the code without having to open the Unreal Engine editor. The version of C++ used for development was *C++ 17*, which is the latest version officially supported in Unreal Engine 4.27.

The project structure was divided into two parts: logic and data. The logic of the application was written in C++, while all of the application's content, such as materials, static meshes, UI layouts, and so on, was stored as assets in the Content folder of the project and could only be opened and changed using the Unreal Engine editor. This separation was made to improve the ease of working with content in the editor and maintain code cleanliness. By storing content in the form of assets, it is possible to continue using pre-made assets, like UI layouts, and manipulate them in the code. This approach allows for a more organized and efficient workflow.

To keep track of the development process and ensure that the code was organized and well-maintained, the version control system Git was used. A remote repository was also created on GitHub to store the code and make it easily accessible. This repository is available at: `https://github.com/menshiva/graphs`. The *master* branch contains the latest added functionality and fixes, while other branches were created for each significant new feature or fix. These branches were created from the master branch at the start of development for a particular feature and were then merged back into the master branch once the feature had been implemented and manually tested. To track progress, milestones were also defined before the implementation to mark the completion of various parts of the application or significant fixes.

Each of these milestones represented a pre-release version of the application, and all of them (including the final release version) were uploaded to the *Releases* section of the GitHub repository. This section allows users to download pre-built bundles of the application, including an executable, and provides a way to review the development history of the project. All applications releases are available at: **`https://github.com/menshiva/graphs/releases`**.

To begin the development process, an empty project was created from a blank template and given the name **Graphs** as the working title for both the project and application. The project was then configured for development on a 64-bit version of Windows, and various settings were adjusted to optimize the project for VR use. The first step in the development process was to create the foundation of the application: the scene.

9.2 Scene

The scene serves as a workspace for the user and an environment where various types of actors can be created, including the user's pawn. In this way, it was implemented as a Level asset containing several predetermined actors to provide the visualization, which was described in Section 8.1. This Level asset was selected in the project settings as the default, essentially making it the starting point for the creation of actors when the application starts up. Among the predefined actors are the following:

- *PlayerStart*: an actor that allows to set the user's spawn point when creating the scene.
- *SkyLight*: a scene component that is a stationary source of diffused light.
- *SkySphere*: an actor that represents a simulation of unbounded space.

To create an immersive environment for the user, a SkySphere actor was added to the scene. This actor, as well as PlayerStart, is placed at the origin of the scene coordinate system. The authors of the engine created this actor to allow developers to quickly create their own sky box, which will be the background of the environment. The actor was copied from the Engine folder, which contains pre-developed content that can be used by developers. It is based on a sphere mesh with a material applied to it, which can be changed in the editor or during runtime to control the appearance of the Sun, the height of the background ground, the brightness of the stars, etc. All settings for this material are controlled through the SkySphere actor that uses it.

In this way, the copied SkySphere actor was brought into the scene and adjusted to enable night mode, hide the moon and background ground, and increase the brightness of the stars. The size of the mesh was also increased to 400 unreal units, which is a sufficiently large size to prevent the user from getting ahead of the background. These modifications allowed for the creation of an outer space environment around the user, as shown in Figure 9.1.

9.3 User pawn

After the scene was set up and ready for interaction, the next step was to implement a pawn that would be responsible for handling input from the VR controllers and headset and displaying the corresponding representation of the user in the scene. This required enabling the OpenXR and Oculus OpenXR plugins. To do this, the project's Plugins directory was opened, and both the plugins were enabled. This allowed the project to use the Oculus VR runtime and the OpenXR API, providing access to all the necessary functions to work with VR.

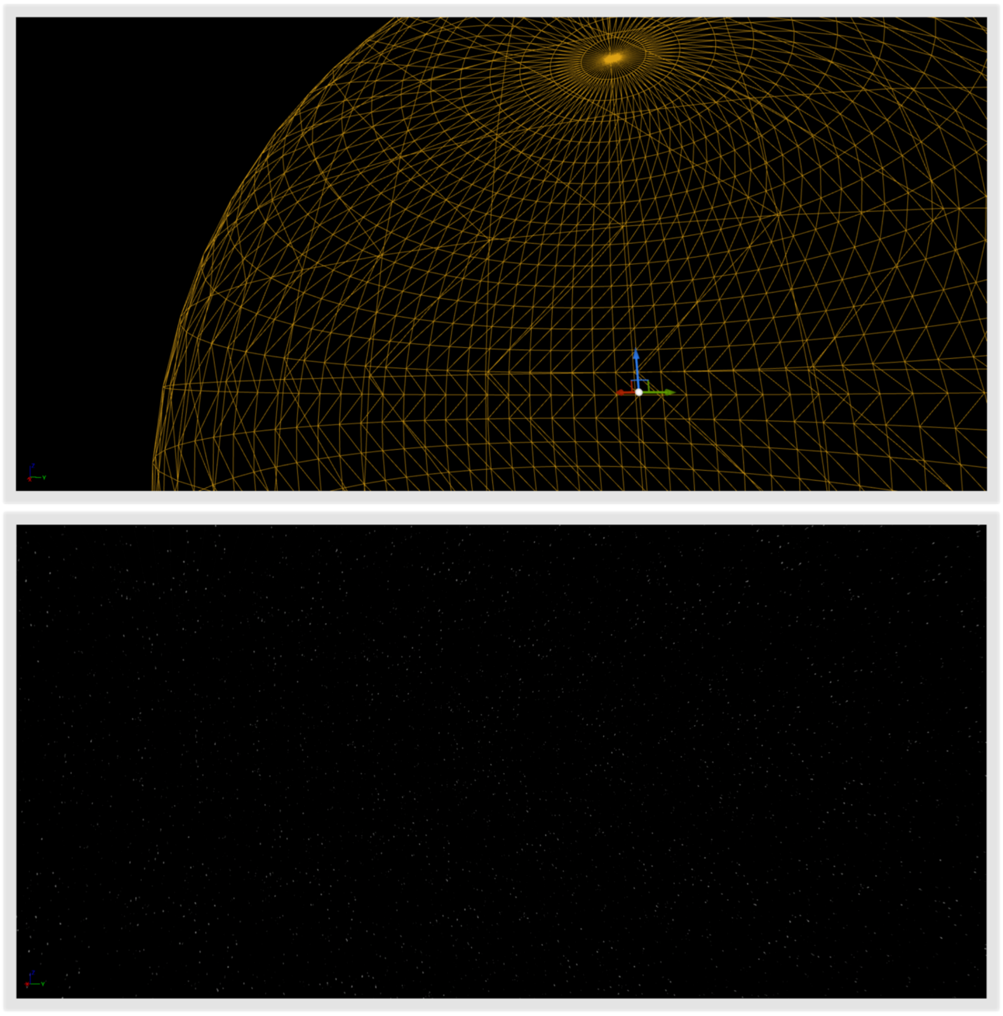


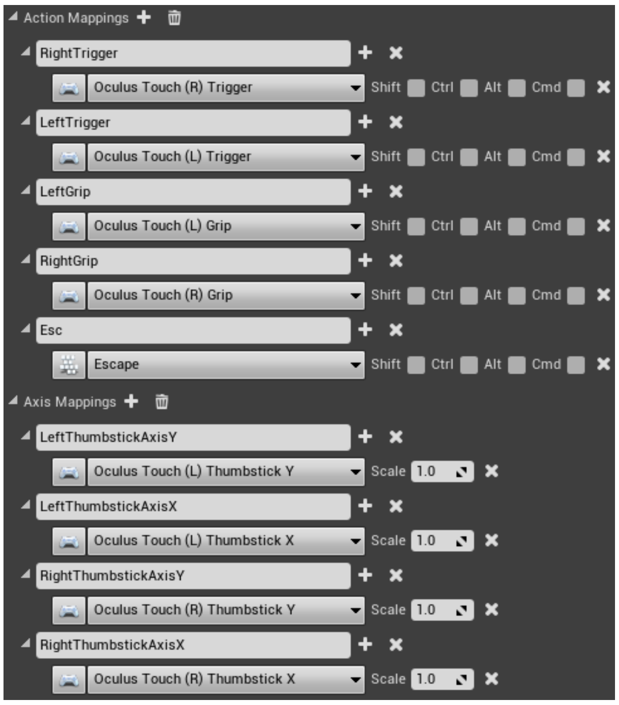
Figure 9.1 Scene of the application. The Figure contains the appearance of the sky actor outside its boundaries in the editor with brush wireframe view mode enabled (top) and the view inside the boundaries of the actor from the user's perspective (bottom).

Once the plugins were enabled, the next step was to create input mappings for controller actions. This was done using the Unreal Engine Input System, which allows developers to define input actions and axis mappings in a separate Action Mappings and Axis Mapping section in the Input settings. Based on the user interactions with the application in Chapter 8, all actions that could be taken to solve this problem were put together, including interactions with the control elements of the controllers. These actions were named and assigned to the mapping groups:

- *Action*: used to trigger events such as pressing, releasing, double clicking, etc.
- *Axis*: used to trigger a binding event for each frame, as well as to output a floating-point value set by specific keys, buttons, control sticks, or mouse inputs.

These defined mappings are shown in Figure 9.2 and will be referenced later in the code to bind input events to gameplay logic.

After defining the necessary input actions and axis mappings, the next step was to implement the pawn class and all of its components.


Action Mappings
- RightTrigger
  - Oculus Touch (R) Trigger — Shift Ctrl Alt Cmd
- LeftTrigger
  - Oculus Touch (L) Trigger — Shift Ctrl Alt Cmd
- LeftGrip
  - Oculus Touch (L) Grip — Shift Ctrl Alt Cmd
- RightGrip
  - Oculus Touch (R) Grip — Shift Ctrl Alt Cmd
- Esc
  - Escape — Shift Ctrl Alt Cmd

Axis Mappings
- LeftThumbstickAxisY
  - Oculus Touch (L) Thumbstick Y — Scale 1.0
- LeftThumbstickAxisX
  - Oculus Touch (L) Thumbstick X — Scale 1.0
- RightThumbstickAxisY
  - Oculus Touch (R) Thumbstick Y — Scale 1.0
- RightThumbstickAxisX
  - Oculus Touch (R) Thumbstick X — Scale 1.0

Figure 9.2 Defined input mappings for the application

The AVRPawn class is the main pawn class in the application, responsible for representing the user in the scene. It is inherited from the APawn class, which provides a lot of useful data and functions for handling the pawn's position and orientation, as well as input events through the APlayerController class.

AVRPawn has several components attached to it, including a camera and two controller objects for the left and right hands. The camera is a UCameraComponent that is responsible for rendering the view that the user sees on the VR headset. The controller objects are instances of UVRControllerLeft and UVRControllerRight, which are inherited from the UVRControllerBase class. These controllers handle input events from the VR controllers and provide various functions, such as vibrations and laser pointer functionality.

One of the important components of the UVRControllerBase is the MotionController component, which is responsible for tracking the user's hand movements and updating the position of the VR controllers accordingly. The MotionController component is attached to the pawn's root component and receives input from the VR controllers through the PlayerController component.

In this way, the UVRControllerBase in the constructor gets information from the inheritors about which controller they represent. Based on this information, it will be determined not only to which of the sources the internal motion controller will be attached, but also the visual representation of the attached controller. By default, Unreal Engine will try to load a static mesh model that is compatible with the device that is driving the motion controller. This can help make it easier and quicker to visualize the connected controller in the application.

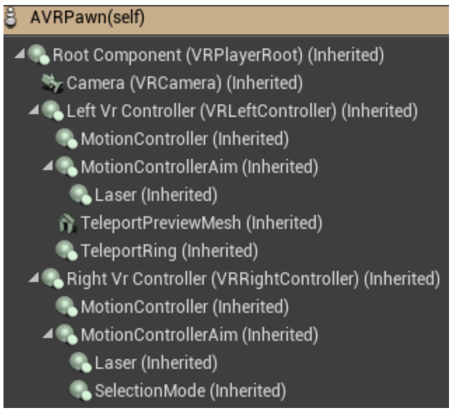
In addition to MotionController, there will be another variation, MotionControllerAim. While the regular motion controller was used to attach to the controller input source, which included its position in the real world, and to visualize the VR controller mesh, the aim motion controller will only be a socket for attaching other components.

The aim controller, unlike the regular one, is not located in the center of the VR controller but near the trigger. Thanks to this, components such as the laser or selection mode can be attached to it, and as a result, their position will be updated by the internal functionality of the

Unreal Engine.

AVRPawn also has a USceneComponent called RootComponent, which serves as the root component for all of the other components attached to the pawn. This allows for easier manipulation of the position and orientation of the pawn as a whole.

The above hierarchy of actors and components can be displayed in a more convenient and clear way in the Components tab of the Unreal Engine editor, as shown in Figure 9.3.



```
AVRPawn(self)
  Root Component (VRPlayerRoot) (Inherited)
    Camera (VRCamera) (Inherited)
    Left Vr Controller (VRLeftController) (Inherited)
      MotionController (Inherited)
      MotionControllerAim (Inherited)
        Laser (Inherited)
      TeleportPreviewMesh (Inherited)
      TeleportRing (Inherited)
    Right Vr Controller (VRRightController) (Inherited)
      MotionController (Inherited)
      MotionControllerAim (Inherited)
        Laser (Inherited)
        SelectionMode (Inherited)
```

Figure 9.3 Hierarchy of components in the AVRPawn class

To make the pawn class the default pawn for the application, the project's GameMode class was updated to specify the AVRPawn as the default. This ensures that the pawn of AVRPawn class will be spawned and controlled by the user when the application is run in VR mode.

After creating the component hierarchy, it was necessary to bind the input events to the controllers. To do this, the SetupPlayerInputComponent function of the pawn ancestor was overridden in the AVRPawn class. This function will be called as soon as the PlayerController is assigned to the VR pawn instance, thereby defining the ideal moment to bind action and axis mappings to the pawn functionality.

Since the mappings were defined mainly for controllers, it was decided to pass the responsibility for binding a particular mapping to the appropriate controller. This means that when the SetupPlayerInputComponent function is called in the pawn, it will call functions to bind mappings for each of the controllers. Therefore, the left virtual controller will be responsible for binding mappings only to the left real controller, and the right virtual controller will be responsible for binding mappings only to the right real controller.

The following Code Listing 9.1 shows an example of code calling the mappings of the left controller to implement teleportation. At the moment when the left controller's grip is pressed, a sphere preview appears, indicating the teleportation point. When the trigger is pressed, the pawn is directly teleported to the position where the preview is located. For convenience, the classes AVRPawn and UVRControllerLeft are provided on the single list.

Thus, the use of mappings and automatic controller visualization in this implementation will bring high extensibility to the application in terms of adding support for new VR devices. For example, to process input data and visualize controller meshes for new VR devices, all that is required is to add the controls of the new controller to the appropriate action or axis mapping group.

With all of AVRPawn components in place and all the input mappings bind and implemented, the AVRPawn class can now be used as the pawn for the resulting application. The player will be able to use the controllers to interact with the scene and the camera will render the scene from the perspective of the player, as shown in Figure 9.4.

Code Listing 9.1 Example of teleportation implementation using left controller input mapping

```
class AVRPawn final : public APawn {
public:
    // receives a PlayerInputController as an argument, which is
    // used for binding
    virtual void SetupPlayerInputComponent(
        UInputComponent *PIC
    ) override {
        // pass input controller to both of the VR controllers
        LeftVrController->SetupInputBindings(PIC);
        RightVrController->SetupInputBindings(PIC);
    }
private:
    UVRControllerLeft *LeftVrController;
    UVRControllerLeft *RightVrController;
}

class UVRControllerLeft final : public UVRControllerBase {
public:
    // this function is declared in UVRControllerBase for taking
    // the responsibility for binding a particular mapping to the
    // appropriate controller
    virtual void SetupInputBindings(UInputComponent *PIC) override {
        // on grip pressed
        BindAction(Pic, "LeftGrip", IE_Pressed, [this] {
            // show the laser, ring, and sphere mesh
            SetLaserActive(true);
            TeleportRing->Activate();
            TeleportRing->SetVisibility(true);
            TeleportPreviewMesh->SetVisibility(true);
            // additionaly, vibrate this controller
            PlayActionHapticEffect();
        });

        // on grip released
        BindAction(Pic, "LeftGrip", IE_Released, [this] {
            // hide the laser, ring, and sphere mesh
            SetLaserActive(false);
            TeleportRing->Deactivate();
            TeleportRing->SetVisibility(false);
            TeleportPreviewMesh->SetVisibility(false);
        });

        // on trigger pressed
        BindAction(Pic, "LeftTrigger", IE_Pressed, [this] {
            if (IsLaserActive()) {
                // teleports the parent pawn to the location of
                // the preview, in world space
                GetVrPawn()->Teleport(
                    TeleportPreviewMesh->GetComponentLocation()
                );
                // vibrate this controller
                PlayActionHapticEffect();
            }
        });
    }
private:
    UStaticMeshComponent *TeleportPreviewMesh;
    UNiagaraComponent *TeleportRing;
}
```

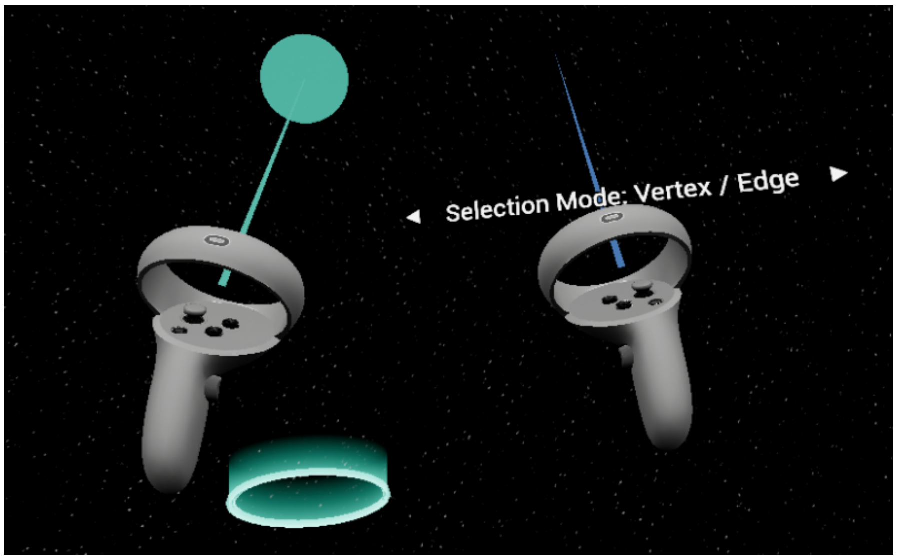


Figure 9.4 Controllers representation from the user's point of view (both grips are pressed)

9.4 Entity system

After a successful implementation of the user pawn, it was time to start developing the graph visualization in the scene. Since the designed architecture consisting of the backend and the frontend implies that the frontend will be a visual reflection of the backend, it was decided to start development with the graph backend, i.e., with the entity system.

To begin with, an entity signature was defined and created along with an entity identifier. The signature is required to define the entity type and is represented in the code as an EntitySignature enum with the underlying type of 8-bit unsigned integer. This enum contains values starting from 0 for each of the possible entity types in the application, that is, *vertex*, *edge*, and *graph*. The *size* value has also been declared. It represents the number of possible entity types, and is also considered an invalid type.

The entity identifier was implemented as an EntityId class consisting of a pair of a 32-bit unsigned integer and a signature value. The first field, "Index," denotes the index of an entity in one of the sparse arrays in the entity system. The second field, "Signature," not only defines the entity type but also forms a unique entity identifier along with the index within the scope of the application. With the structure of the entity system as designed, it is impossible to create a unique identifier without a signature. So, if vertex and edge entities have been added to their respective sparse arrays, they will have an index of 0, which violates the uniqueness.

Their declaration in part is shown in Code Listing 9.2.

EntityId contains the *NONE()* function in addition to the constructor and unspecified functions in the listing. It is a static function that returns a singleton specifically defined to denote an invalid identifier. It will be used in cases such as indicating an error when an entity is created or deserialized or to signify that the trace line of the right controller has not detected a collision with any entity.

However, the main interest here are the *Hash(EntityId)* and *Unhash(uint32_t)* functions. These functions are used to generate a hash value for a given EntityId, and to compute the original EntityId that was used to generate the given hash, respectively. It was decided to choose *Cantor pairing and unpairing functions* [36] as hash and dehash functions. Since EntityId is a unique pair of two numbers, this function can be used to encode them into a single number. The decision was made based on several features of this function:

Code Listing 9.2 Entity signature and identifier declarations

```cpp
// ---------------- EntityId.h ----------------
enum EntitySignature : uint8_t {
    VERTEX = 0,
    EDGE,
    GRAPH,
    SIZE
};

struct EntityId {
    static EntityId NONE() {
        static EntityId NoneId(-1, SIZE);
        return NoneId;
    }

    EntityId(
        const uint32_t Index,
        const EntitySignature Signature
    ) : Index(Index), Signature(Signature) {}

    static uint32 Hash(const EntityId Id) {
        return Utils::CantorPair(Id.Index, Id.Signature);
    }

    static EntityId Unhash(const uint32 Hash) {
        const auto [Index, Signature] = Utils::CantorUnpair(Hash);
        return EntityId(
            Index,
            static_cast<EntitySignature>(Signature)
        );
    }
private:
    uint32_t Index;
    EntitySignature Signature;
};

// ---------------- Utils.h ----------------
constexpr static uint32_t CantorPair(
    const uint32_t X,
    const uint32_t Y
) {
    return (X + Y) * (X + Y + 1) / 2 + Y;
}

static std::pair<uint32_t, uint32_t> CantorUnpair(
    const uint32_t Val
) {
    // Calculate the value of the upper bound of the
    // pair of integers
    const auto T = static_cast<uint32_t>(
        floorf(-1.0f + sqrtf(1.0f + 8.0f * Val)) / 2.0f
    );
    // Calculate and return the pair of integers
    return {
        T * (T + 3) / 2 - Val,
        Val - T * (T + 1) / 2
    };
}
```

1. *One-to-one correspondence*: The Cantor pairing function creates a one-to-one correspondence between pairs of integers and integers. This means that every pair of integers maps to a unique integer and every integer maps back to a unique pair of integers. It will be useful when creating a hash or dehash.

2. *Compact representation*: The Cantor pairing function represents pairs of integers using a single integer. This can be useful when it is necessary to store or transmit pairs of integers and to use a compact representation. Specifically, it will be useful when storing the EntityId in collision metadata in the scene. This will allow the identifier to be stored despite the fact that the metadata only supports storing a single number.

3. *Easy to implement*: The Cantor pairing function is relatively simple to implement, as it only involves a few arithmetic operations. This makes it easy to use in a wide range of applications.

Next came the implementation of the entity types themselves. In the big picture, their implementation is no different from what was described when designing the entity system in Section 7.1. However, it is worth clarifying that such fields as *bool IsHit* and *FColor OverrideColor* were added to the vertices and edges entities.

The IsHit flag has been added so that when rendering their geometry, it is possible to determine if an entity has been hit by a right controller trace and to render a color specifically defined as the selection color. In the case of vertices, it would help to keep the color defined by the user in the Color field. In the case of edges, this was done for code consistency.

The OverrideColor field in both entity types is intended to specify a secondary color. This will be used, for example, in the remove tool, when selecting an entity by the right controller trigger marks it red as the one to be removed. In the case of vertices, it will also help to keep the user-defined color.

Speaking of the implementation of the entity system, it was implemented as a singleton class storing three sparse arrays, one for each of the possible entity types. A small snippet of its implementation is shown in Code Listing 9.3. As can be seen, this implementation uses the template specialization of the private GetStorage methods. This gets rid of a lot of condition checks in each of the public methods of the system, greatly improving the clarity and readability of the code. Also, it eliminated the need to provide a base class for all types of entities and got rid of runtime type information and completely eliminate the use of dynamic_cast.

9.5 Renderers

Before starting to develop the graph renderer, the RuntimeMeshComponent plugin repository was cloned [29] and added to the project as a submodule. By adding this plugin to the Build.cs file of the main graph application module, the plugin was successfully integrated into Unreal Engine.

First, the URendererBase class was created. This class is a base class for vertex and edge providers. It inherits from the URuntimeMeshProvider provided by the plugin and generalizes the provider logic for the vertex and edge renderers. This was done with the bias that each URuntimeMeshProvider must be properly configured, i.e. its section must be created, a section must be assigned material, each section must have parameters configured, and so on. Since the difference between vertex and edge providers is only in the mesh and collisions they generate, all the common logic of the two providers is covered in the base class to avoid code duplication.

In the base class, the overridden function Initialize reads an asset with material for a section, creates a single section, and then configures it. For example, the visibility flag will be set to true and the flag responsible for casting shadows using mesh normals will be set to false. Besides that, the class also calculates the bounding box of the generated geometry to apply levels of detail and an occlusion culling algorithm. In our case, one level of detail will be used, and the occlusion culling algorithm will be applied to the entire graph.

Code Listing 9.3 Excerpt of the implementation of the entity system

```
class ES {
public:
    template <typename EntityType>
    using StorageImpl = TSparseArray<EntityType>;
private:
    static ES &GetInstance() {
        static ES Singleton;
        return Singleton;
    }

    template <typename EntityType>
    StorageImpl<EntityType> &GetStorage() {
        UE_LOG(LogTemp, Fatal, TEXT("Undefined␣type!"));
        return StorageImpl<EntityType>();
    }

    template <>
    StorageImpl<VertexEntity> &GetStorage<VertexEntity>() {
        return Vertices;
    }

    template <>
    StorageImpl<EdgeEntity> &GetStorage<EdgeEntity>() {
        return Edges;
    }

    template <>
    StorageImpl<GraphEntity> &GetStorage<GraphEntity>() {
        return Graphs;
    }

    StorageImpl<VertexEntity> Vertices;
    StorageImpl<EdgeEntity> Edges;
    StorageImpl<GraphEntity> Graphs;
};
```

The ChunkRenderer class is basically a place to store identifiers for a certain number of entities. It has two providers, one for vertices and one for edges, that are in charge of rendering those entities. The GraphRenderers class is the storage for ChunkRenderers. Suppose a vertex needs to be redrawn. The instance of the GraphRenderers class receives the command to redraw the entity and its identifier. Then it searches for which ChunkRenderer is responsible for rendering the vertex with the received identifier.

This flow should be optimized as much as possible because this sequence of actions, when using, for example, the manipulator tool, will be called every frame. Code Listing 9.4 shows the excerpt of the internal implementation of the GraphRenderer class.

Code Listing 9.4 Excerpt of the GraphRenderers class implementation

```
class AGraphsRenderers final : public AActor {
private:
    UPROPERTY()
    TSet<AGraphChunkRenderer*> AllChunks;

    TSparseArray<TSet<AGraphChunkRenderer*>> GraphsChunks;
    TSparseArray<AGraphChunkRenderer*> VerticesChunksLookup;
    TSparseArray<AGraphChunkRenderer*> EdgesChunksLookup;
};
```

Considering this implementation, it can be seen why the graph visualization frontend has been referred to as a reflection of the backend several times within this paper. Its structure also consists of three sparse arrays. However, in this class, they store references to created chunk renderers. This solution does not save memory, but it allows for the identification of a chunk responsible for rendering an entity with complexity $\mathcal{O}(1)$. It is also important to note that using TSet to store all the created chunks is due to the fact that each chunk is an actor. TSparseArray does not have the ability to work with the garbage collector, in contrast to TSet.

So, for example, when adding a new vertex to a graph, all the chunks will be enumerated. When a free chunk is found or created, this vertex will be added to it. Then in VerticesChunksLookup using the index of the vertex identifier, the link to the chunk responsible for its rendering will be added.

The next important task for the URendererBase class is to lock and unlock the critical section. Since the generation of each section is asynchronous, the main application thread can change entity data while being read by providers to generate meshes. For this, a critical section and structure called RenderData have been defined. This structure is the same for both vertex and edge providers. It contains three arrays for identifiers, positions, and colors of entities for which a mesh or collision must be generated.

The ChunkRenderer will be involved in the generation of RenderData for the corresponding provider when requested to render any type of entities. This action takes place on the main thread and will not affect the read-write race condition. After the data for rendering is generated, it is moved and saved in the provider to be rendered. And it is at this phase that the critical section must be locked because while the data is being moved, the asynchronous rendering of the previous data by the provider may occur.

Regarding the procedural generation process, a single material, shown in Figure 9.5, was used for both vertex and edge sections. This material is extremely simple and does nothing but return the color received as input.

The provider receives a render command from the chunk, along with render data, as part of the process. Render data for vertices include their positions (vertex centers) and colors. The vertex provider goes through all the positions in the render data and, in their place, generates first-order icosphere geometry vertices and indexes, along with a color for each geometry vertex. The icosphere is generated by copying a predefined pair of vertex and index arrays. These arrays

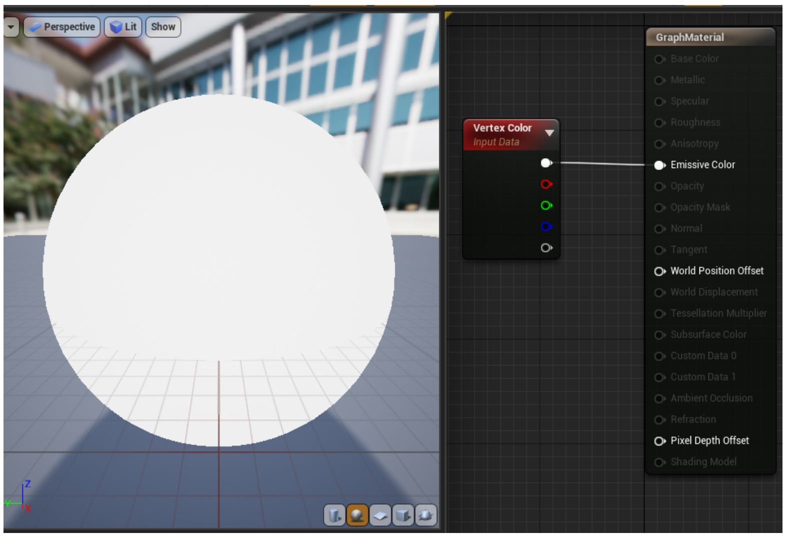


Figure 9.5 Material for vertex and edge providers

were manually generated using an algorithm [37] and hardcoded inside the vertex provider. After copying the icosphere, all the vertices of its geometry are moved to the position of the graph vertex.

The vertex collisions are generated in a similar way, but without color and with a zero-order icosphere. In addition, each collision triangle of one entity stores its identifier using the Cantor pairing function. This is necessary so that when the right controller traces, it is possible to retrieve the entity identifier from the triangle.

For the edge provider, its mesh generation is slightly different from that of the icosphere. Since an edge is rotated, it is generated in runtime by rotating the vector and saving positions. Thus, one face of the edge will be generated. Its positions will be moved to the first vertex to which the edge connects, scaled by the defined value, and then this face will be copied and moved to the second vertex to which the edge connects.

Therefore, with the correct layout of chunks, renderers, and providers and the correct implementation of locking and unlocking the critical section and the procedural generation algorithm, the first result was obtained, as shown in Figure 9.6. This figure shows the graph with eight vertices and eight edges, which was used in manual testing during the development.

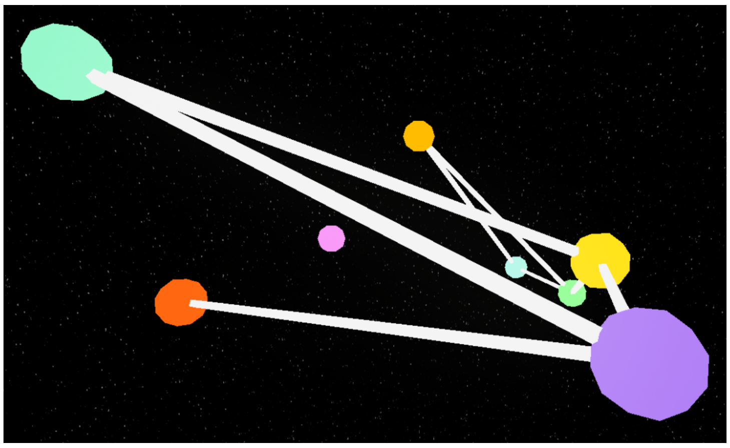


Figure 9.6 Visualized colorful graph with eight vertices and eight white edges

As can be seen, the edges of this graph are drawn in white. This is because the edges do not have their own specific color. Despite this, thanks to the procedural generation method, there is quite a lot of flexibility in controlling single vertices, indices, and geometry colors. Since the color in the material is assigned to each geometry vertex, when generating graph edges, the color of the first graph vertex can be applied to the first face geometry vertices, and to the second face geometry vertices, the color of the second graph vertex, respectively. This will result in smooth interpolation of the edge color from the first vertex of the graph to the second vertex, as can be seen in Figure 9.7.

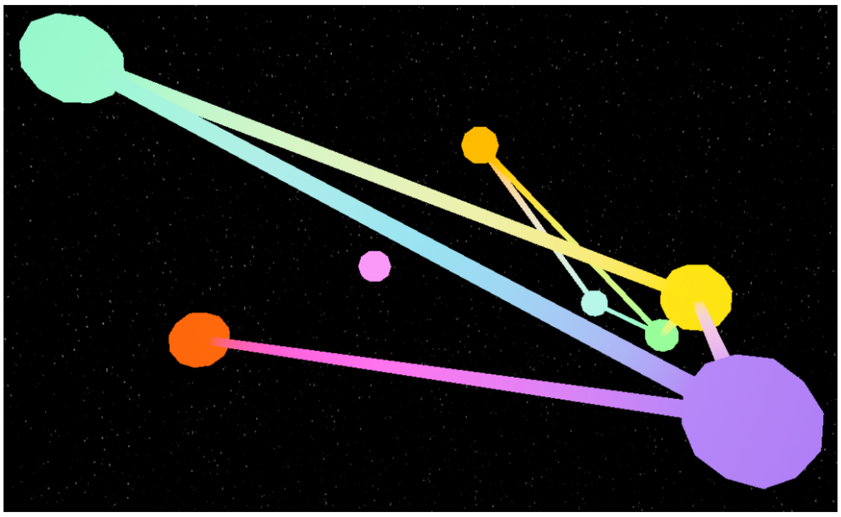


Figure 9.7 Visualized colorful graph with eight vertices and eight colored edges

Although such functionality is not a functional requirement for the application, this feature was demonstrated to the thesis supervisor, Ing. Petr Pauš, Ph.D., and subsequently *approved*.

In addition, by disabling mesh generation and activating the *pxvis collision* console command, the generated collision can be examined in detail. As can be seen in Figure 9.8, it resembles the mesh itself from afar, despite the fact that much fewer geometry vertices and indices are involved in its generation.

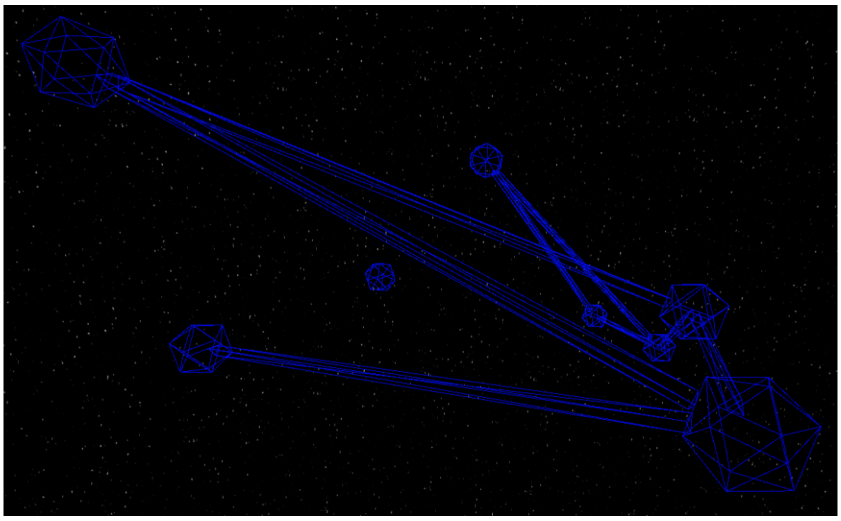


Figure 9.8 Visualized collision of the graph with eight vertices and eight edges

9.6 Tool system and user interface

Based on the UI design done in Section 8.3, the UWidgetComponent was attached to the left controller. This class provides a surface in the 3D environment on which to render widgets normally rendered on the screen. Because of this, making a UI for VR with WidgetBlueprint assets will not be any different from making a regular UI.

In this way, using the Unreal Motion Graphics UI Designer (hereafter UMG), the menu asset was created as shown in Figure 9.9. It is essentially the foundation for laying out all the other UI assets. It was decided to make the entire UI in a dark style to give more resemblance to the designed space.

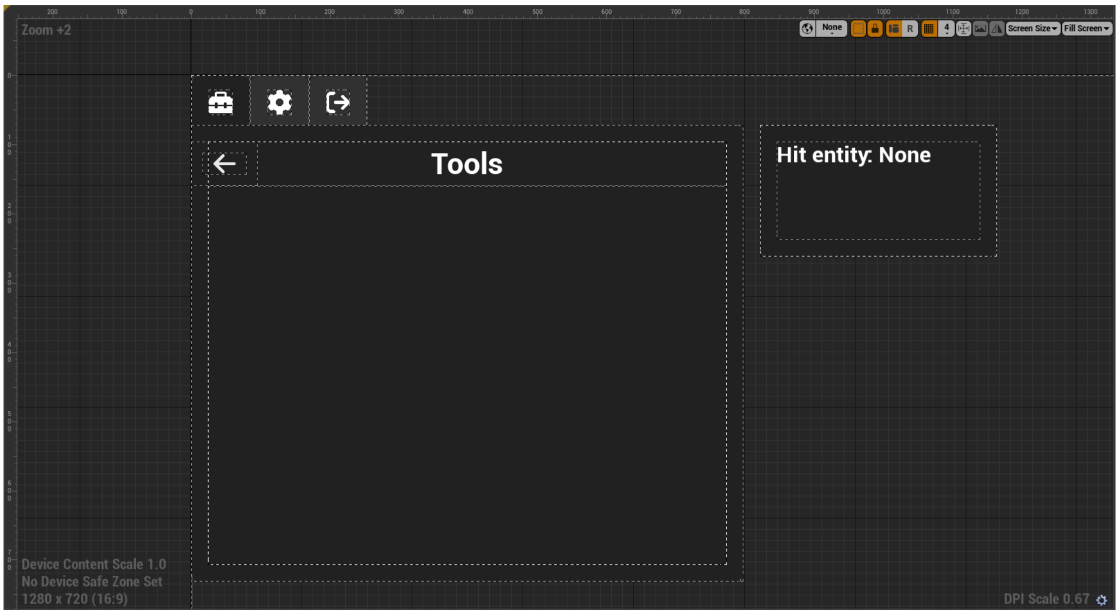


Figure 9.9 Menu in the designer window

It is worth noting the strengths and weaknesses of UMG. One of the strengths of UMG is the ability to inherit widgets. Due to this feature, a separation of the UI logic in C++ code and the visual part in the widget asset has been implemented.

It works by creating a new class in the code inherited from UUserWidget. Then some functionality can be added to this class. In UMG, a new widget must be created and inherited from the class created in the code. After that, it is possible to use logic written in C++ inside the widget. Moreover, once any widget is declared in an ancestor class field and given a name, the widget of the same type with the same name in the descendant widget will be related to each other. Thus, it is sufficient to change some properties of the widget in the code, and then these changes will be automatically reflected in the descendant widget as well. This feature is widely used in the practical part of the work.

Among the weaknesses of UMG is its quite limited library of pre-made widgets. Therefore, to provide the necessary elements of interaction with the UI in a homogeneous style, the following widgets were created: a text button, an icon button, a tool button, a checkbox, a list item for an import tool, and an option selector. The developed widgets were used throughout the entire UI of the application, providing the necessary interaction elements in a uniform style.

As for the tool system, it was implemented based on the design provided in Section 8.4. Each tool was implemented based on the application requirements listed in Section 3.3.1 and the design in Appendix A. In addition, it should be noted that the *RapidJSON* [38] library was used in the implementation of import and export tools to work with JSON objects.

Chapter 10
Results

This chapter presents the findings from user testing and the performance measurements of the resulting application. The user tests were conducted to assess the usability and user experience of the application, while the performance measurements evaluated the application's ability to handle a specified number of objects in the scene. The chapter concludes with a summary of the work and suggestions for potential improvements and new functionality that could be implemented in the future.

Before providing information about the user tests, it is also worth noting that some form of testing was also carried out throughout the development of the application. Thus, during the development of each of the main parts of the application, its functionality was tested and verified to see if it fulfilled certain functional and non-functional requirements or was somehow involved in their fulfillment. In addition, during development, a large number of different *assert* macros (the equivalent in Unreal Engine is a *check* macro) were written into the code. This greatly helped to detect and diagnose unexpected or invalid runtime conditions during development and determine if the flow of the implemented functionality worked as originally conceived and described in the design phase.

10.1 User tests

Conducting user tests is an important step in the development of any application. It allows developers to gather valuable feedback from users and identify any problems or issues that may exist in the application. The goal of conducting user tests within the scope of the work was to verify the functionality of the application and identify any issues or difficulties users may have while using the application and to gather feedback on the overall user experience.

A total of 4 participants were recruited for the user tests. The test participants were divided into two groups: experienced and inexperienced. The experienced group consisted of individuals who had previously worked with VR and had some experience with it. The inexperienced group consisted of individuals who had never worked with VR before and who were trying it for the first time.

The tests were conducted 2 days a week apart. The first day was spent familiarizing inexperienced users with VR, working in it, and using the Oculus Quest 2 headset. This was followed by an introduction to the functionality of the developed application for both experienced and inexperienced participants, during which the users were observed as they interacted with the application and recorded any problems or difficulties they encountered. Participants were asked to complete a series of tasks using the resulting application, such as creating and importing graphs as well as viewing and interacting with existing graphs. The test scenarios on the first day for both groups were the same and consisted of the following tasks:

1. Create a graph using the create tool and add vertices and edges to it.
2. Edit the properties of the created graph and its components using the edit tool.
3. Move/rotate the created graph using the manipulation tool.
4. Export the graph using the export tool.
5. Remove some of the vertices and/or edges and then the entire graph using the remove tool.
6. Import the previously exported graph using the import tool and ensure that it has the same structure as before the removal.

The testing conducted on the second day aimed to test the application's ability to recover the graph work done on the first day. It also included a series of tests during which an observation of the behavior of the participants was made. The test scenarios on the second day consisted of the following tasks:

1. Import the graph created and exported on the first day and make sure that its representation is identical to that on the first day.
2. Take the headset off, minimize the application, and change the structure and/or properties in the JSON file of the graph just imported.
3. Import a new graph from the modified file and make sure that the application correctly reflected the changes made or correctly pointed out the JSON file structure error.
4. All from the first day's test scenario.

As can be seen, the tests performed 2 days a week apart differ from each other only in some aspects and could have been performed in one day. However, it is important to clarify that this division of the testing process was carried out using a gap of one week to test *ease of learning* of the participants on the first day regarding work with the resulting application, as the second day of familiarization with the functionality was *not* carried out.

The observations made on the participants while using the application consisted of testing two main aspects: functionality and usability. The first aspect covers the main functionality of the application, such as creating and editing graphs, working with external files, interacting with graphs, and viewing graphs in VR. The goal of the second aspect was to assess the overall usability of the application and identify any issues or problems that users might encounter while using it. The time it took the participants to complete each task was recorded, as well as any problems or difficulties they encountered while performing the tasks. Users have also been asked to provide verbal feedback on their experience using the application, as well as any suggestions they had for improvement.

Overall, the user testing process was a **success**. All participants were able to complete the tasks assigned to them using the application without any major issues on both the first and second days of testing. The results of the user tests showed that both experienced and inexperienced users were able to complete the tasks in a relatively short amount of time and without encountering any major problems or difficulties. The experienced users were able to complete the tasks slightly faster than the inexperienced users, which was expected, as they had some prior experience with VR. However, the difference in the time it took for both groups to complete the tasks was not significant.

The results of the observation with respect to the functional aspect of the application were positive. The user satisfaction survey showed that all users were satisfied with the application's functionality and found it very useful, especially in the field of education. The results of the observation regarding the usability aspect were also generally positive. Users reported that they found the application easy to use and that they did not encounter any significant usability or performance problems while using it.

However, a few areas for improvement were identified based on user feedback. Some participants did experience minor usability issues while using the application. Among these are:

1. Difficulties in distinguishing between the color defined to indicate the highlighted entities and the color specified by the user himself. This happens only in some rare cases, such as when the color of the vertex approximates the blue color of the selection.
2. The relative difficulty in getting used to switching selection modes on the right controller. All participants were able to adjust to this solution, and their user experience improved over the course of the testing, particularly on the second day. Nevertheless, on first use, users found the solution relatively difficult to get used to.
3. The information window on the left controller is not in the most convenient location. This was noted in cases where the participant was pointing the right controller at an entity to examine its data. To do this, they had to use both hands, aim at an entity with the right controller, and pay attention to the information window on the left controller while doing so. The majority of participants noted that this solution was “not the most comfortable.”

All comments, suggestions for improvements, and new functionality additions received during the participant survey were noted, addressed, and will be resolved in future iterations of the application. These problems, along with ideas to improve the application, will also be discussed in Section 10.3.

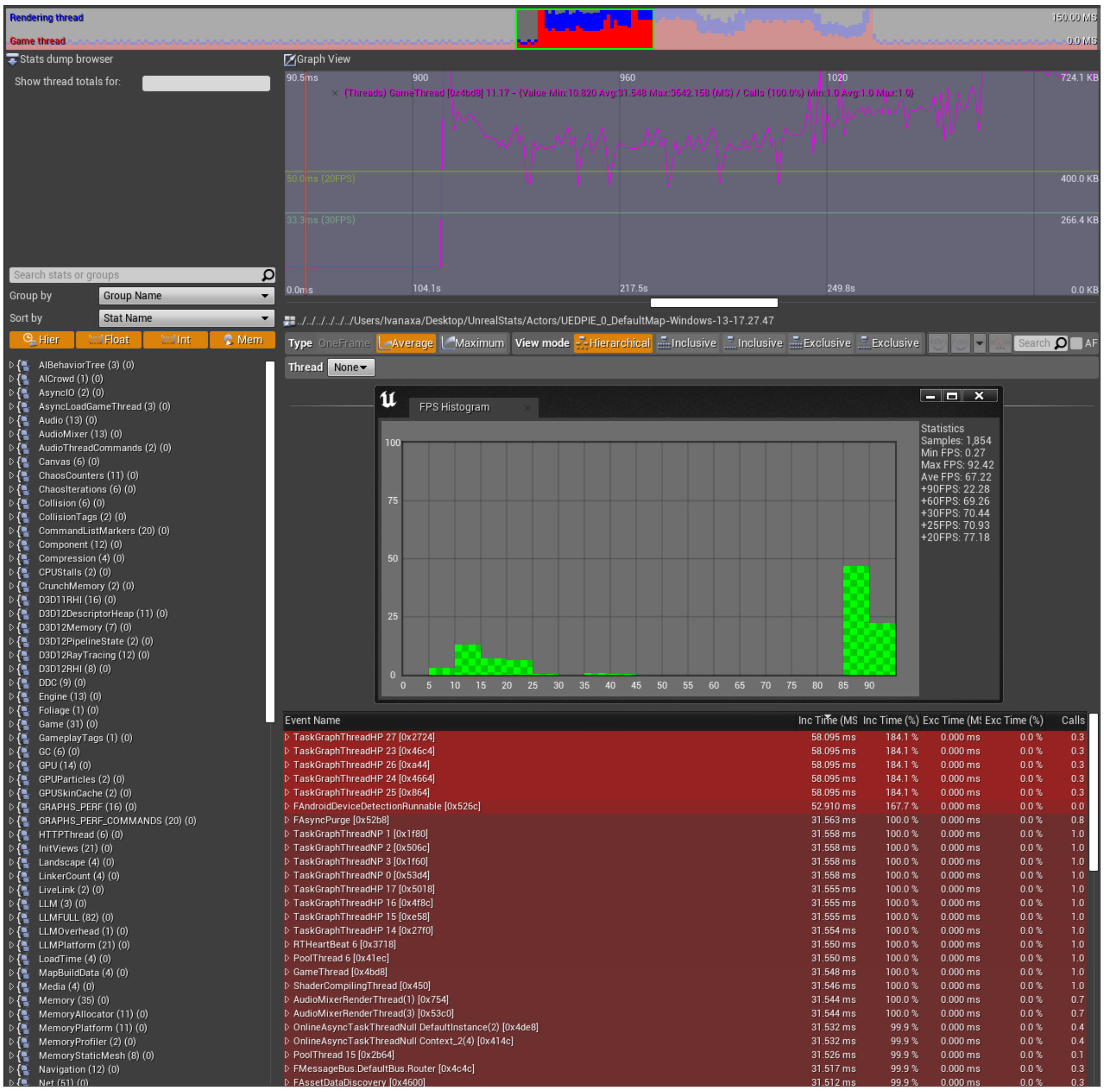
In conclusion, the user testing of the resulting application showed that the application is generally easy to use and that it fulfills all of its functional and non-functional usability requirements. Users found the application to be intuitive and easy to understand and did not experience significant issues or problems while using it.

10.2 Performance tests

Performance tests are used to measure the performance of the entire application or component. These tests can help identify bottlenecks, optimize performance, and ensure that the system or component can handle expected workloads. As part of this work, performance tests were conducted to ensure that the application meets one of the non-functional requirements to visualize at least 2 graphs with 5,000 vertices and 5,000 edges each at the same time and at least 10 graphs with 1,000 vertices and 1,000 edges each at the same time while maintaining 90 FPS.

The testing was performed using the *Stats system* [39] in Unreal Engine to capture session statistics while rendering several graphs of different sizes and manipulating them. The results of each of the sessions were analyzed in the Profiler window in the engine editor. Figure 10.1 shows an example of what the results of a session capture look like. The window displays a lot of useful information, including a graph view where the load on the game and render threads throughout the session can be observed, as well as an FPS histogram.

Before testing, the application was built in the “Development Editor Win64” configuration and run. Testing was carried out as follows. After the application was launched, the *Stat StartFile* console command was used to start collecting statistics. Next, graphs of the same size were sequentially imported into the scene until no performance problems were detected. Immediately after each graph was imported, a manipulator tool was applied to it. It was chosen because it is the tool that involves changing the graph entity data and redrawing it each frame, which was an excellent test of the application’s ability to redraw a graph while interacting with it.



```
../../../../../../Users/Ivanaxa/Desktop/UnrealStats/Actors/UEDPIE_0_DefaultMap-Windows-13-17.27.47
```

```
AIBehaviorTree (3) (0)
AICrowd (1) (0)
AsyncIO (2) (0)
AsyncLoadGameThread (3) (0)
Audio (13) (0)
AudioMixer (13) (0)
AudioThreadCommands (2) (0)
Canvas (6) (0)
ChaosCounters (11) (0)
ChaosIterations (6) (0)
Collision (6) (0)
CollisionTags (2) (0)
CommandListMarkers (20) (0)
Component (12) (0)
Compression (4) (0)
CPUStalls (2) (0)
CrunchMemory (2) (0)
D3D11RHI (16) (0)
D3D12DescriptorHeap (11) (0)
D3D12Memory (7) (0)
D3D12PipelineState (2) (0)
D3D12RayTracing (12) (0)
D3D12RHI (8) (0)
DDC (9) (0)
Engine (13) (0)
Foliage (1) (0)
Game (31) (0)
GameplayTags (1) (0)
GC (6) (0)
GPU (14) (0)
GPUParticles (2) (0)
GPUSkinCache (2) (0)
GRAPHS_PERF (16) (0)
GRAPHS_PERF_COMMANDS (20) (0)
HTTPThread (6) (0)
InitViews (21) (0)
Landscape (4) (0)
LinkerCount (4) (0)
LiveLink (2) (0)
LLM (3) (0)
LLMFULL (82) (0)
LLMOverhead (1) (0)
LLMPlatform (21) (0)
LoadTime (4) (0)
MapBuildData (4) (0)
Media (4) (0)
Memory (35) (0)
MemoryAllocator (11) (0)
MemoryPlatform (11) (0)
MemoryProfiler (2) (0)
MemoryStaticMesh (8) (0)
Navigation (12) (0)
Net (51) (0)
```

```
Statistics
Samples: 1,854
Min FPS: 0.27
Max FPS: 92.42
Ave FPS: 67.22
+90FPS: 22.28
+60FPS: 69.26
+30FPS: 70.44
+25FPS: 70.93
+20FPS: 77.18
```

Event Name	Inc Time (MS	Inc Time (%)	Exc Time (MS	Exc Time (%)	Calls
TaskGraphThreadHP 27 [0x2724]	58.095 ms	184.1 %	0.000 ms	0.0 %	0.3
TaskGraphThreadHP 23 [0x46c4]	58.095 ms	184.1 %	0.000 ms	0.0 %	0.3
TaskGraphThreadHP 26 [0xa44]	58.095 ms	184.1 %	0.000 ms	0.0 %	0.3
TaskGraphThreadHP 24 [0x4664]	58.095 ms	184.1 %	0.000 ms	0.0 %	0.3
TaskGraphThreadHP 25 [0x864]	58.095 ms	184.1 %	0.000 ms	0.0 %	0.3
FAndroidDeviceDetectionRunnable [0x526c]	52.910 ms	167.7 %	0.000 ms	0.0 %	0.0
FAsyncPurge [0x52b8]	31.563 ms	100.0 %	0.000 ms	0.0 %	0.8
TaskGraphThreadNP 1 [0x1f80]	31.558 ms	100.0 %	0.000 ms	0.0 %	1.0
TaskGraphThreadNP 2 [0x506c]	31.558 ms	100.0 %	0.000 ms	0.0 %	1.0
TaskGraphThreadNP 3 [0x1f60]	31.558 ms	100.0 %	0.000 ms	0.0 %	1.0
TaskGraphThreadNP 0 [0x53d4]	31.558 ms	100.0 %	0.000 ms	0.0 %	1.0
TaskGraphThreadHP 17 [0x5018]	31.555 ms	100.0 %	0.000 ms	0.0 %	1.0
TaskGraphThreadHP 16 [0x4f8c]	31.555 ms	100.0 %	0.000 ms	0.0 %	1.0
TaskGraphThreadHP 15 [0xe58]	31.555 ms	100.0 %	0.000 ms	0.0 %	1.0
TaskGraphThreadHP 14 [0x27f0]	31.554 ms	100.0 %	0.000 ms	0.0 %	1.0
RTHeartBeat 6 [0x3718]	31.550 ms	100.0 %	0.000 ms	0.0 %	1.0
PoolThread 6 [0x41ec]	31.550 ms	100.0 %	0.000 ms	0.0 %	1.0
GameThread [0x4bd8]	31.548 ms	100.0 %	0.000 ms	0.0 %	1.0
ShaderCompilingThread [0x450]	31.546 ms	100.0 %	0.000 ms	0.0 %	1.0
AudioMixerRenderThread(1) [0x754]	31.544 ms	100.0 %	0.000 ms	0.0 %	0.7
AudioMixerRenderThread(3) [0x53c0]	31.544 ms	100.0 %	0.000 ms	0.0 %	0.7
OnlineAsyncTaskThreadNull DefaultInstance(2) [0x4de8]	31.532 ms	99.9 %	0.000 ms	0.0 %	0.4
OnlineAsyncTaskThreadNull Context_2(4) [0x414c]	31.532 ms	99.9 %	0.000 ms	0.0 %	0.4
PoolThread 15 [0x2b64]	31.526 ms	99.9 %	0.000 ms	0.0 %	0.1
FMessageBus.DefaultBus.Router [0x4c4c]	31.517 ms	99.9 %	0.000 ms	0.0 %	0.3
FAssetDataDiscovery [0x4600]	31.512 ms	99.9 %	0.000 ms	0.0 %	0.3

Figure 10.1 Profiler window

If performance problems were detected, the session recording was end with *Stat StopFile* command, and the application was closed. Then a new session was started during which graph visualization testing of larger graphs was conducted. This was repeated for all generated graphs. In general, these tests were aimed at determining the maximum number of graphs of the same size that can be displayed simultaneously in the scene while maintaining 90 FPS.

To generate large graphs, a small script was written in Python. The code for this script is provided in Appendix B. Graphs with the following numbers of vertices and edges were generated for testing:

- 1,000 vertices and 1,000 edges
- 5,000 vertices and 5,000 edges
- 10,000 vertices and 10,000 edges
- 15,000 vertices and 15,000 edges

The Oculus Quest 2, VR headset with the following specifications, was used for testing:

- CPU: Qualcomm® Snapdragon XR2
- GPU: Qualcomm® Adreno™ 650
- RAM: 6 GB

To be more informative during testing, the maximum refresh rate of the headset was set to 120 Hz.

The testing was conducted on an HP OMEN GT13 personal computer and a Lenovo Legion 7 laptop. Personal computer configuration:

- CPU: Intel® Core™ i7-10700K CPU @ 3.80GHz 3.70 GHz
- GPU: NVIDIA GeForce RTX™ 3070
- RAM: 32 GB
- OS: Windows 11 Home, x64, 22H2 22621.963

Laptop configuration:

- CPU: AMD Ryzen™ 7 5800H
- GPU: NVIDIA GeForce RTX™ 3060 Laptop
- RAM: 40 GB
- OS: Windows 10 Pro, x64, 22H2 19045.2364

The test results are presented in Table 10.1.

Table 10.1 Performance test results

Graph	Personal computer: maximum quantity	Personal computer: commentary	Laptop: maximum quantity	Laptop: commentary
1,000 vertices 1,000 edges	> 50	120 FPS stable; only from graph 52 dropped to 110 FPS	24	at 24th graph FPS drops from 120 to 70-80
5,000 vertices 5,000 edges	7	rare drops to 90 FPS from 5th graph; from 7th graph stable 60 FPS	4	varies between 90-120 FPS; at 4th graph FPS drops to 60-70
10,000 vertices 10,000 edges	4	rare drops to 60 FPS when manipulating 4th graph	2	2nd graph manipulation occurs at 60 FPS
15,000 vertices 15,000 edges	2	stable 60 FPS when manipulating 2nd graph	1	rare drops to 50-60 FPS

Based on this data, it can be noted that the application not only fulfills the non-functional requirement for performance but is also able to render much more graphs than specified of even larger size. It is also worth noting that the data was obtained during testing of the Development Editor version of the application. In the Shipping version of the application for release, the result may be even better.

10.3 Conclusion and future work

Although the application has already achieved a high and required level of functionality and performance, there are still many areas for further improvement and development. The following list presents with a potential future work directions for this application, including fixes, improvements and new features:

1. Dynamic chunk size
2. Distribution of entities by chunk, taking into account the effectiveness of occlusion culling
3. Several levels of detail for the geometry
4. Support for directed and mixed graphs
5. Handle self-loops in graphs
6. Make the entity highlighting with the glow
7. Allow the information window to float
8. Support for various VR devices
9. Visualization of graph theory algorithms

In conclusion, the goal of this thesis was to design, develop, and test a standalone software product that can visualize graphs in VR and offers tools for performing certain operations on such graphs. To achieve this goal, various aspects of VR application development and graph representation were analyzed and considered.

The system was designed to represent graphs in 3D space and provide users with tools to work with them in a VR environment. The design of the system considered user comfort, content optimization, and the limitations of VR platforms. Additionally, the system was designed to be able to render and visualize huge graphs while simultaneously accepting and processing user queries without interruption.

The proposed representation of graphs was designed to be easily editable and manipulable in VR, as well as being easy to save and retrieve from an external file. The resulting system is able to accept user requests for operations on a graph and, based on those requests, render and alter a relevant graph in 3D space with the use of VR technology. The system was designed to provide an alternative way to represent and interact with graphs, making it more convenient and understandable for users.

Several optimizations for the entire application were presented to achieve high system performance. With these improvements, the system was able to reach and go beyond the level of performance needed to draw and process a number of very large graphs.

Through the development and testing process, it was demonstrated that the resulting system is able to effectively visualize and manipulate graphs in VR, providing users with a new and convenient way to work with them. It is believed that the resulting system has the potential to be a useful tool in a variety of fields, including education and research, where the visualization and manipulation of graphs is important.

Overall, the successful completion of this project demonstrates the feasibility of using VR technology to represent and interact with graphs in a more intuitive and engaging way. It is hoped that the resulting system will make a valuable contribution and be of use to a wide range of users.

Appendix A
Tools design

This document contains a detailed description of the functionality and operation of all the tools provided in the application. Each tool is carefully designed to meet the functional requirements of the application and the use cases defined in the analysis phase in Chapter 3, ensuring that they are easy to use and effective in performing their intended tasks.

The design of each tool includes a full description of its functionality and operation, as well as the design of its user interface and a process diagram. This information is essential to understand how each tool works and how it can be used to achieve the desired results.

A.1 Import

The import tool is an important part of the application, as it allows the user to bring graph structures from external files into the virtual environment for further manipulation and interaction. The tool is designed to be user-friendly and easy to use, providing a simple and intuitive way for the user to select a file from the shared folder and import a graph into the scene with which they can begin to work.

A.1.1 Main window

Figure A.1 shows the main window of the import tool, which contains a list of all available JSON files located in the shared folder for import and export purposes. This window is presented to the user when the import tool is activated and allows the user to select a file to import. The list is presented in a scrollable container and displays the name of each file. The user can select a file by clicking on its name in the list.

The main window is designed to be intuitive and easy to use, with a clear layout that makes it easy for the user to find and select a file to import. It is also designed to be responsive and dynamic, with the ability to refresh the list of available files at any time. This ensures that the user always has the most up-to-date information about the files available for import, making it easy to keep track of new and modified files.

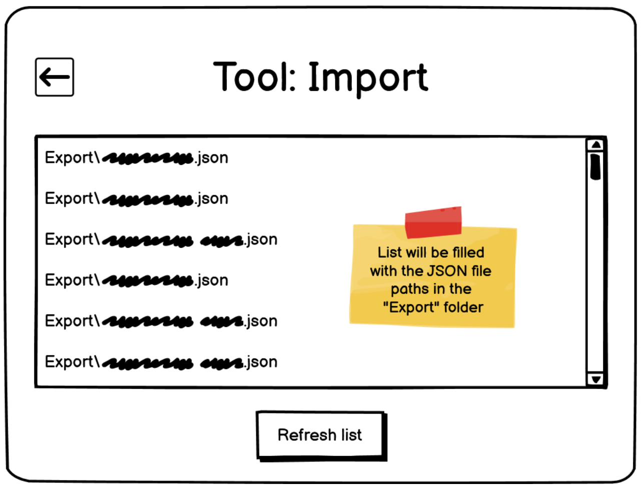


Figure A.1 Import tool: main window

A.1.2 Loading window

The loading window of the import tool, shown in Figure A.2, is presented to the user when the tool is reading the contents of the selected file and attempting to deserialize the object. This window is designed to provide the user with visual feedback that the tool is still working, as the import process may take some time, especially for larger graphs.

The loading window consists of a single element, an infinite progress bar, which is displayed in the center of the window. This progress bar is designed to rotate continuously, intending to keep the user informed and provide a sense of progress.

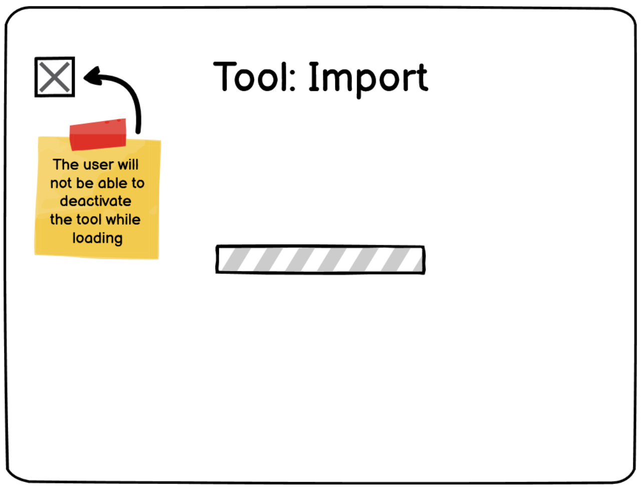

Tool: Import
The user will not be able to deactivate the tool while loading

Figure A.2 Import tool: loading window

A.1.3 Result window

The result window of the import tool is presented to the user after the tool has finished processing the selected file and attempting to deserialize the object. This window is designed to provide the user with information about the result of the import process, as well as the option to confirm the result and return to the main window with the list. The result window can be in one of two states: a success state or an error state.

If the graph has been successfully imported without any problems, it will be immediately drawn in the scene and the window will display a message indicating that the import was successful, along with a green OK button that the user can click to confirm the result and close the window. The result window in the success state is shown in Figure A.3.

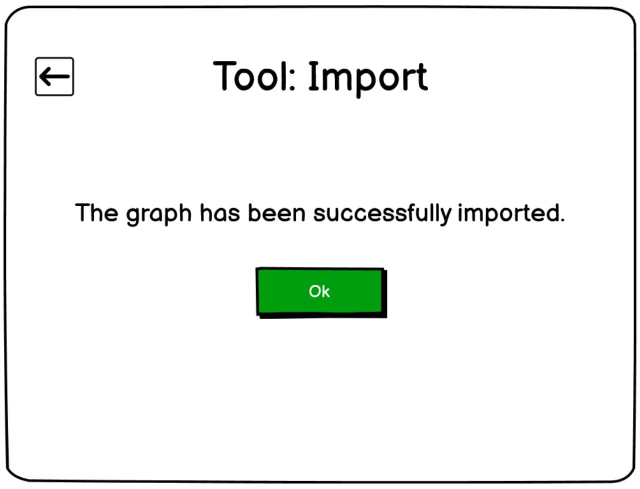


Figure A.3 Import tool: result window in the success state

If there was an error during the import process, the window will display a message indicating that an error occurred, along with a description of the error and a red OK button that the user can click to confirm the result and close the window. The result window in the error state is shown in Figure A.4.

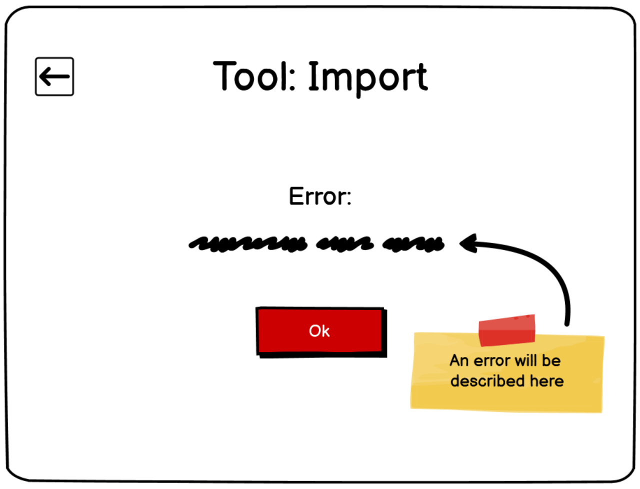
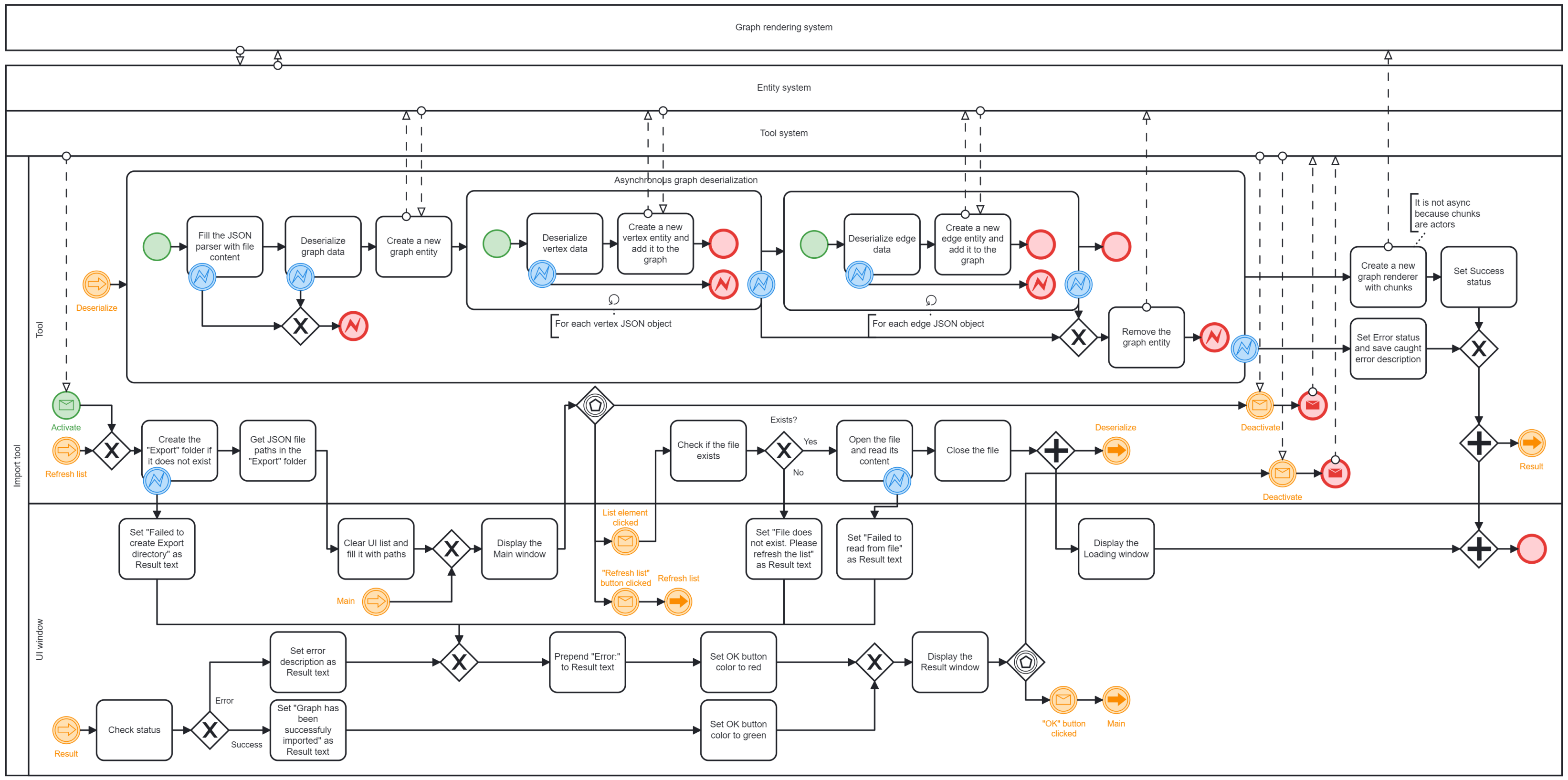


Figure A.4 Import tool: result window in the error state

A.1.4 Process model

The business process model illustrated in Figure A.5 shows the functioning of the import tool and steps involved in the process of importing a graph into the virtual environment.


Graph rendering system
Entity system
Tool system
Import tool
Tool
UI window
Asynchronous graph deserialization
Deserialize
Fill the JSON parser with file content
Deserialize graph data
Create a new graph entity
Deserialize vertex data
Create a new vertex entity and add it to the graph
For each vertex JSON object
Deserialize edge data
Create a new edge entity and add it to the graph
For each edge JSON object
Remove the graph entity
It is not async because chunks are actors
Create a new graph renderer with chunks
Set Success status
Set Error status and save caught error description
Result
Activate
Refresh list
Create the "Export" folder if it does not exist
Get JSON file paths in the "Export" folder
Check if the file exists
Exists?
Yes
No
Open the file and read its content
Close the file
Deserialize
Deactivate
Deactivate
Set "Failed to create Export directory" as Result text
Clear UI list and fill it with paths
Display the Main window
List element clicked
"Refresh list" button clicked
Refresh list
Main
Set "File does not exist. Please refresh the list" as Result text
Set "Failed to read from file" as Result text
Display the Loading window
Set error description as Result text
Prepend "Error:" to Result text
Set OK button color to red
Display the Result window
Error
Success
Result
Check status
Set "Graph has been successfuly imported" as Result text
Set OK button color to green
"OK" button clicked
Main

■ **Figure A.5** Import tool: process model diagram

A.2 Export

The export tool is a feature that allows users to save the structure of a graph from the virtual environment to an external file. It allows users to save their work and make it possible to reuse it later, which is especially useful when working on large or complex graphs that take a lot of time to build. With this tool, users can save their progress and come back to it at a later time without having to recreate the entire graph from scratch.

Despite the fact that the export tool's functionality is diametrically opposed to that of the import tool, their sets of windows are identical, and their contents differ only in a few places.

A.2.1 Main window

Figure A.6 shows the main window of the export tool, which is used only to prompt the user to select a graph in the scene that they wish to export. It does not display a list of files like the main window of the import tool, as the export process always creates a new file with a randomly generated name in the shared folder for import and export. In order to select a graph, the user will use their right controller to point at the desired graph and press the trigger to confirm the selection.

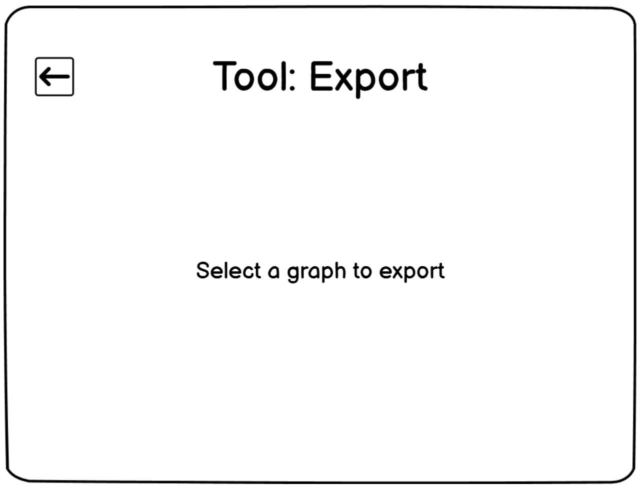


Figure A.6 Export tool: main window

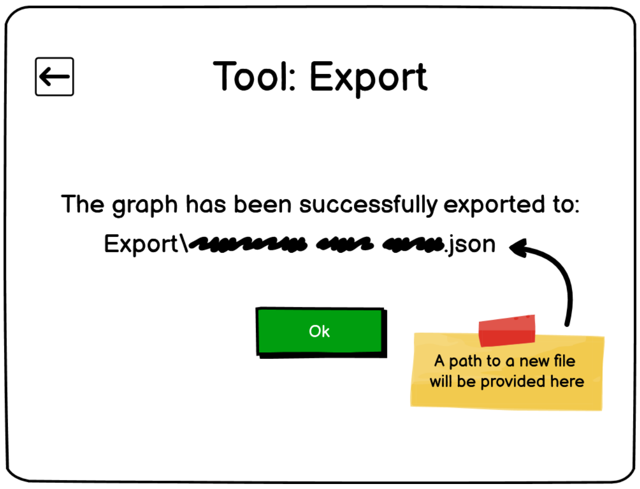
A.2.2 Loading window

The loading window of the export tool is identical to the loading window of the import tool, which was described in Section A.1.2.

A.2.3 Result window

The result window of the export tool is displayed to the user after the export process has been completed. It is used to inform the user of the outcome of the process and provide any additional information necessary. Depending on whether or not the export was successful, the result window will be displayed in one of two states.

If the export was successful, the window will show the result message and the OK button to confirm the result. It will also display the name of the file that was generated and in which the serialized structure was written. This information is provided to allow the user to easily locate the file for future use or reference. The result window in the success state is shown in Figure A.7.


Tool: Export

The graph has been successfully exported to:

Export\.json

Figure A.7 Export tool: result window in the success state

If the export was not successful, the window will show "Error" and a description of the error that occurred. The structure of the window in the error state will be the same as that of the import tool provided in Figure A.4.

A.2.4 Process model

The following business process model provided in Figure A.8 outlines the process of how the export tool functions within the application, from the activation of the tool and the initiation of the export process by the user to the final confirmation of the result.

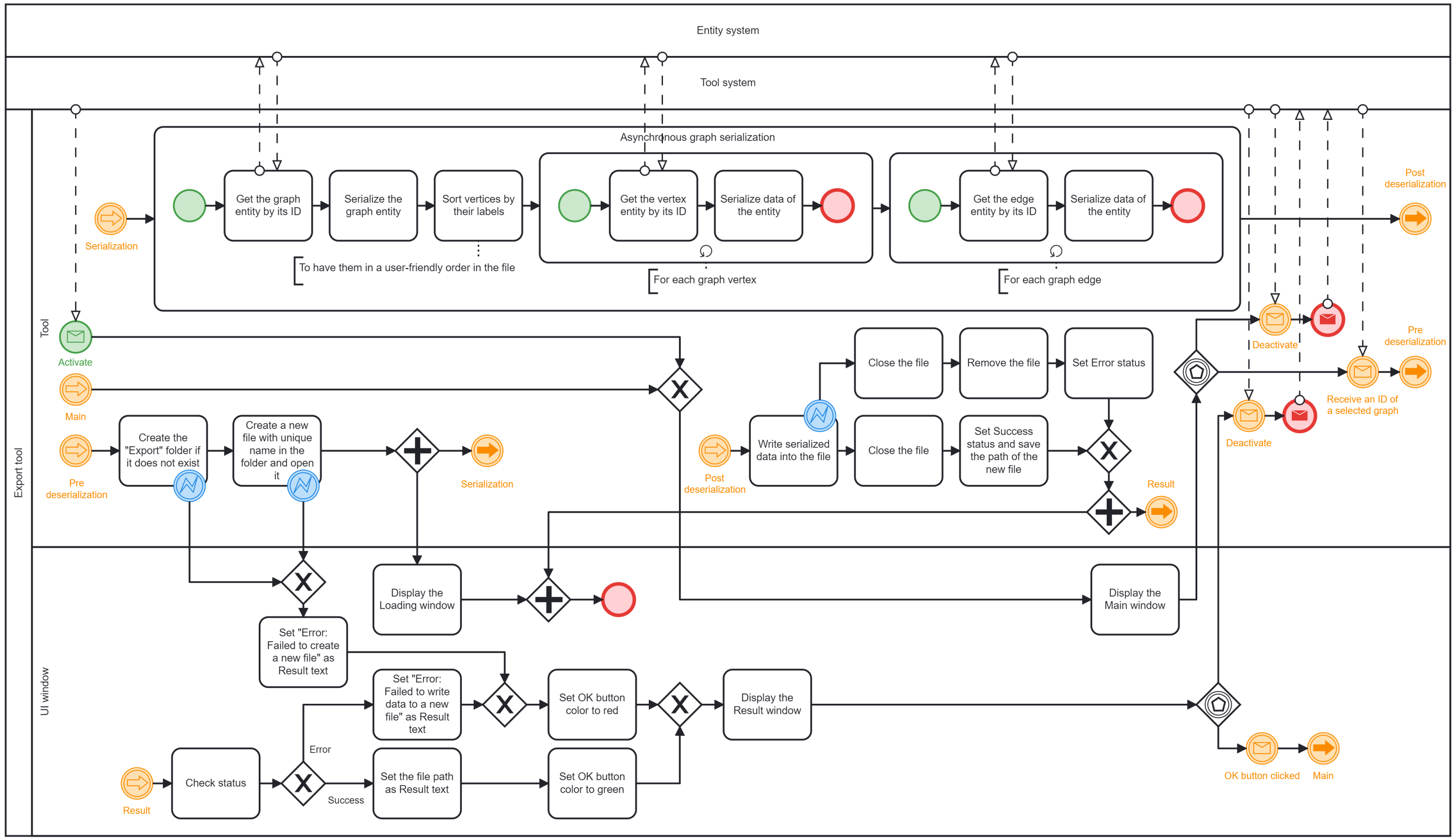

Entity system
Tool system
Export tool
Tool
UI window
Asynchronous graph serialization
Serialization
Get the graph entity by its ID
Serialize the graph entity
Sort vertices by their labels
To have them in a user-friendly order in the file
Get the vertex entity by its ID
Serialize data of the entity
For each graph vertex
Get the edge entity by its ID
Serialize data of the entity
For each graph edge
Post deserialization
Activate
Main
Pre deserialization
Create the "Export" folder if it does not exist
Create a new file with unique name in the folder and open it
Serialization
Post deserialization
Write serialized data into the file
Close the file
Remove the file
Set Error status
Close the file
Set Success status and save the path of the new file
Result
Deactivate
Deactivate
Pre deserialization
Receive an ID of a selected graph
Display the Loading window
Display the Main window
Set "Error: Failed to create a new file" as Result text
Set "Error: Failed to write data to a new file" as Result text
Set OK button color to red
Display the Result window
Error
Success
Result
Check status
Set the file path as Result text
Set OK button color to green
OK button clicked
Main

Figure A.8 Export tool: process model diagram

A.3 Create

The create tool is designed to allow users to manually build graphs in the application. It offers three distinct modes of operation, each of which is intended for creating a specific type of entity.

A.3.1 Main window

The create tool's user interface consists of a single main window. Its main part is a circular selector, which allows users to choose the type of entity they want to create. There are three options available: vertex, edge, and graph.

In vertex creation mode, shown in Figure A.9, the user is prompted to choose a graph in the scene to which they want to add a new vertex (left). After the selection is done, the user will be prompted to specify the position in the scene for a new vertex (right). This will be done with a vertex preview attached to the laser of the right controller, which the user can move. Clicking on the trigger of the right controller will create a vertex at the desired position and add it to the selected graph. This action can be performed until the user changes the current mode or deselects the graph. For this purpose, a Deselect button is provided on the window, clicking on which will cause the graph to be deselected.

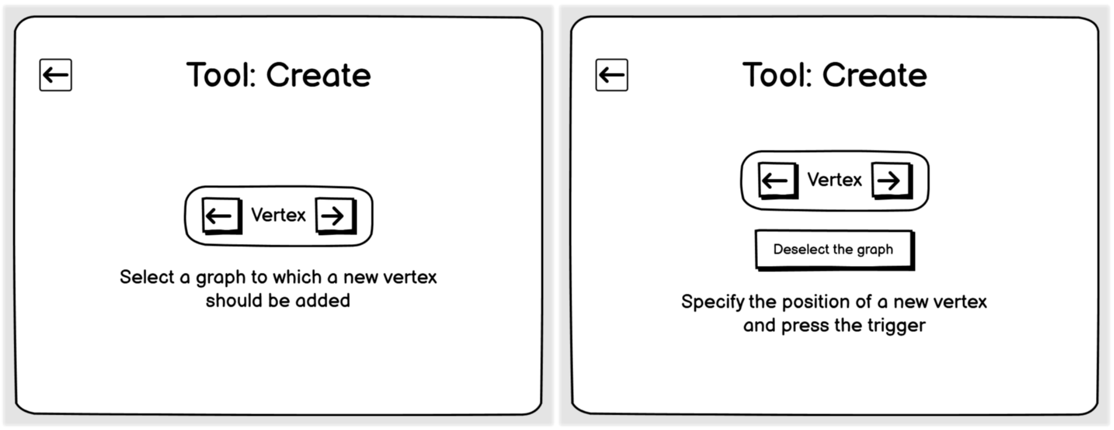

Tool: Create
Vertex
Select a graph to which a new vertex should be added
Tool: Create
Vertex
Deselect the graph
Specify the position of a new vertex and press the trigger

Figure A.9 Create tool: main window in vertex creation mode

In edge creation mode, shown in Figure A.10, the user is prompted to select a first vertex of a future edge in the scene (left). After the selection is done, the user will be prompted to select a second vertex (right). After the first vertex is selected, the window will also show a Deselect button to reset the edge creation. This mode, however, implies the validity check of the edge. So, the edge cannot connect two vertices that are already connected and the two selected vertices must be from the same graph. After user selection of a second vertex and successful validation, the graph will be redrawn taking into account the new edge. In the event of unsuccessful validation, the second vertex selection will be reset.

In graph creation mode, shown in Figure A.11, the user is prompted to specify the position in the scene for the first vertex of a new graph. As in the case of vertex creation mode, this will be done using the provided vertex preview attached to the laser of the right controller. Once the position is chosen, the tool will create an empty graph entity and add it a new vertex placed at the specified position. The newly created graph will then be automatically selected, and the tool will switch to the vertex creation mode so that the user can immediately start adding additional vertices to the graph.

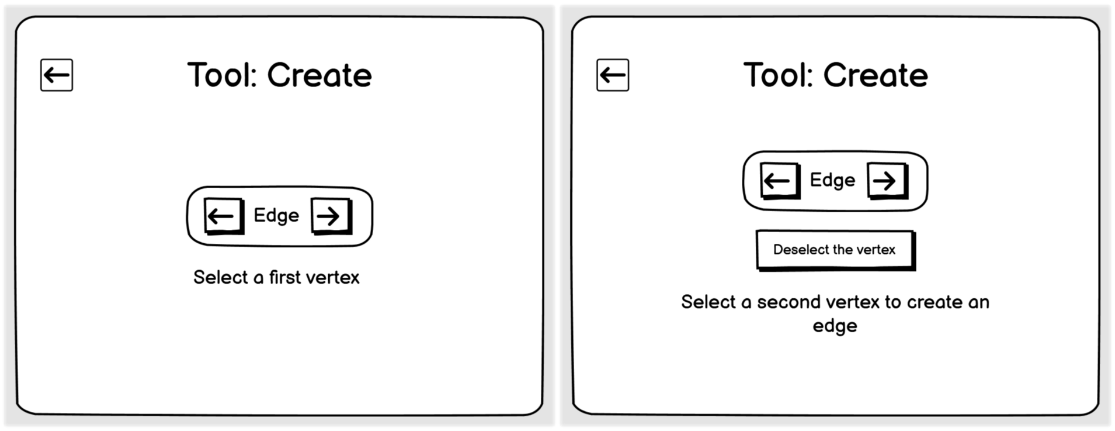


Figure A.10 Create tool: main window in edge creation mode

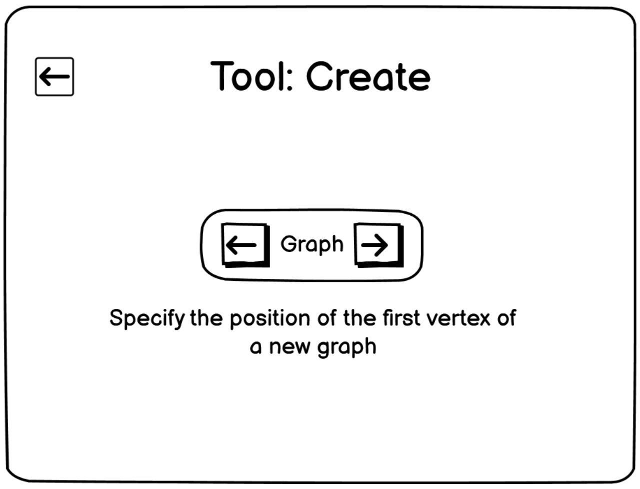
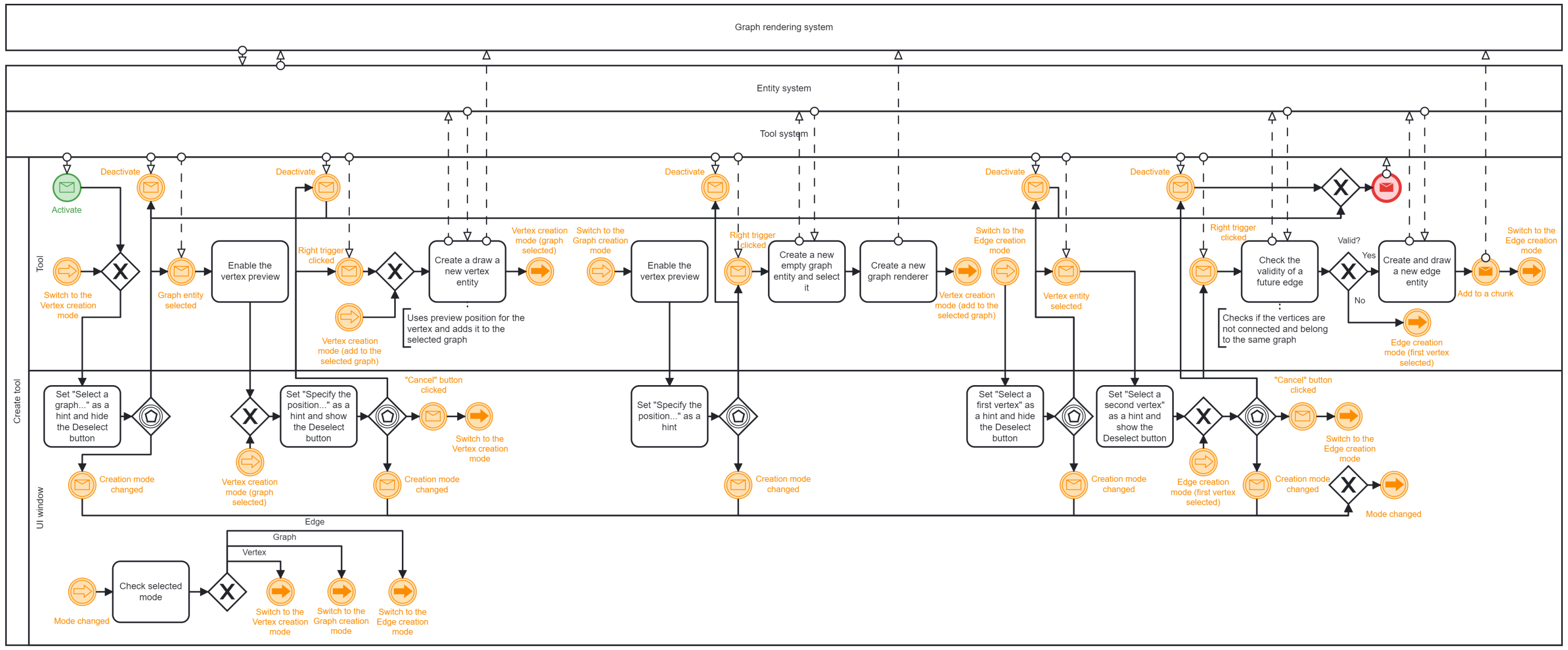


Figure A.11 Create tool: main window in graph creation mode

A.3.2 Process model

The following business process model provided in Figure A.12 represents the process of the create tool functioning. The model shows the steps involved in each mode of the create tool, including the creation and addition of new entities to a graph, the user's actions, and the tool's responses.



Graph rendering system
Entity system
Tool system
Create tool
Tool
UI window
Activate
Deactivate
Switch to the Vertex creation mode
Graph entity selected
Enable the vertex preview
Deactivate
Right trigger clicked
Vertex creation mode (add to the selected graph)
Create a draw a new vertex entity
Uses preview position for the vertex and adds it to the selected graph
Vertex creation mode (graph selected)
Switch to the Graph creation mode
Enable the vertex preview
Deactivate
Right trigger clicked
Create a new empty graph entity and select it
Create a new graph renderer
Vertex creation mode (add to the selected graph)
Switch to the Edge creation mode
Deactivate
Vertex entity selected
Deactivate
Right trigger clicked
Check the validity of a future edge
Checks if the vertices are not connected and belong to the same graph
Valid?
Yes
No
Create and draw a new edge entity
Add to a chunk
Switch to the Edge creation mode
Edge creation mode (first vertex selected)
Set "Select a graph..." as a hint and hide the Deselect button
Creation mode changed
Vertex creation mode (graph selected)
Set "Specify the position..." as a hint and show the Deselect button
"Cancel" button clicked
Switch to the Vertex creation mode
Creation mode changed
Set "Specify the position..." as a hint
Creation mode changed
Set "Select a first vertex" as a hint and hide the Deselect button
Creation mode changed
Set "Select a second vertex" as a hint and show the Deselect button
Edge creation mode (first vertex selected)
"Cancel" button clicked
Switch to the Edge creation mode
Creation mode changed
Mode changed
Edge
Graph
Vertex
Mode changed
Check selected mode
Switch to the Vertex creation mode
Switch to the Graph creation mode
Switch to the Edge creation mode

Figure A.12 Create tool: process model diagram

A.4 Edit

The edit tool is an application feature that allows users to alter the attributes of graph, vertex, and edge entities in the scene. When using the edit tool, the user first uses the right controller to select an entity in the scene. The user interface of the edit tool will consist of a series of windows, depending on the type of object selected. All of the three windows will contain a list of all the object's editable properties, which the user can modify as needed. In all three windows, changes made to the properties of an object are immediately applied and can be saved or discarded using the Save and Cancel buttons.

A.4.1 Main window

Thus, when the tool is activated, the main window will be displayed with a prompt to select an entity in the scene. The design of this window is shown in Figure A.13.

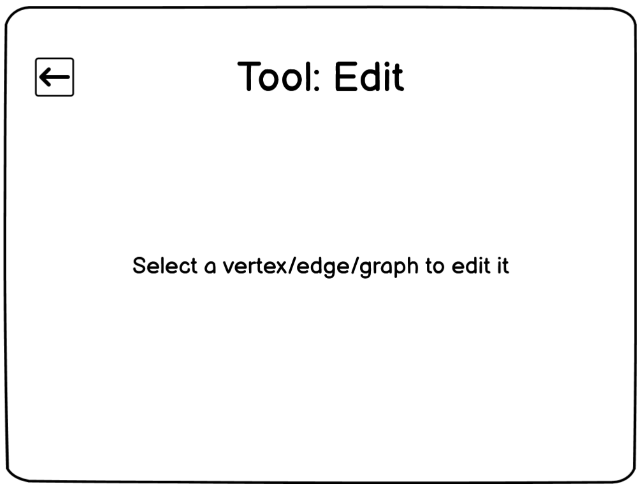


Figure A.13 Edit tool: main window

A.4.2 Vertex window

For a vertex object, the edit vertex window is displayed. Depending on whether the parent graph is colorful or not, this window will display either a prompt that the parent graph is not colorful or a field to change the color of the vertex, which is shown in Figure A.14.

The field consists of a number of buttons, each of which displays a preview of a pre-defined color. By pressing the button, the color is applied immediately to the selected vertex. In addition, the field also has a button for applying a randomly generated color to the vertex. It is worth noting that such a solution to change the vertex color by using buttons with predefined colors was adopted in accordance with the fact that Unreal Engine does not provide any built-in widget to change colors, such as a color picker.

A.4.3 Edge window

For an edge object, the edit edge window shown in Figure A.15 is displayed. It contains only the field for changing the weight of the selected edge. When this window is displayed, the field will be pre-filled with the current value of the selected entity's weight. Focusing on this edit field will display a virtual numeric keypad window for entering a floating point number.

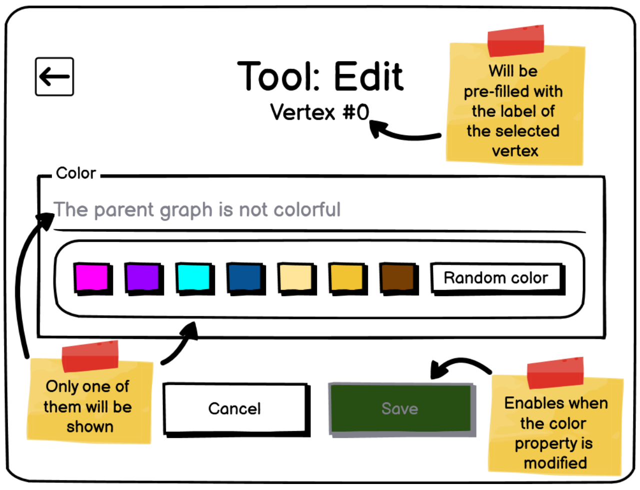

Tool: Edit
Vertex #0
Will be pre-filled with the label of the selected vertex
Color
The parent graph is not colorful
Random color
Only one of them will be shown
Cancel
Save
Enables when the color property is modified

Figure A.14 Edit tool: edit vertex window

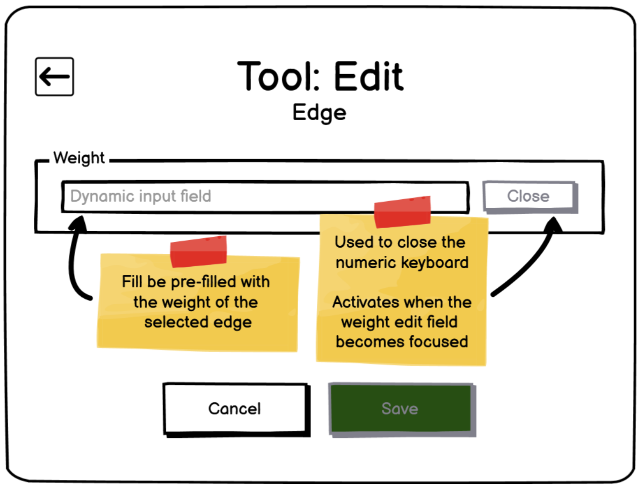

Tool: Edit
Edge
Weight
Dynamic input field
Close
Fill be pre-filled with the weight of the selected edge
Used to close the numeric keyboard
Activates when the weight edit field becomes focused
Cancel
Save

Figure A.15 Edit tool: edit edge window

A.4.4 Graph window

The edit graph window shown in Figure A.16 will be displayed if the graph entity is selected. It allows users to change the colorful property of the selected graph through a tick button. If the colorful property is set to true, this window expands to provide more options to change the color of all vertices in the graph. This includes a field similar to the one contained in the edit vertex window, with buttons for applying a predefined or randomly generated color. In this window, however, a predefined color will be applied not to one vertex, as it was in the case of the edit vertex window, but to all vertices in the selected graph. This feature was specifically designed for the user's convenience by applying a single color to all of the graph's vertices.

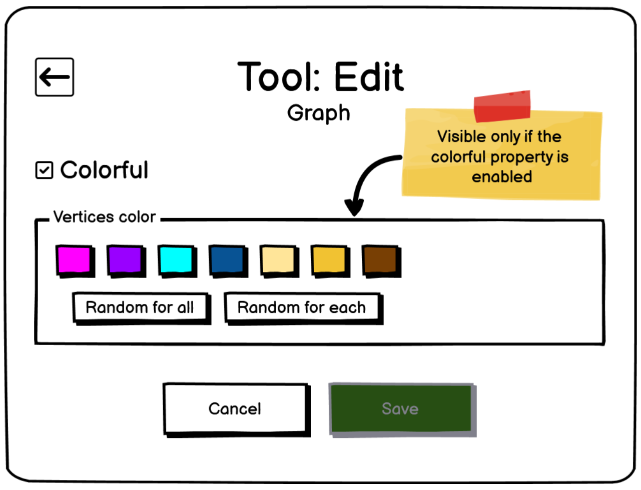


Figure A.16 Edit tool: edit graph window

A.4.5 Process model

The following business process model provided in Figure A.17 represents the functionality of the edit tool. It consists of three different activity flows, one for each of the possible entity types that can be edited: vertex, graph, and edge. Each activity flow will follow the same basic logic, with the user first selecting an entity and then making changes to its properties using the tool's user interface. The flow will then branch based on whether the user decides to save their changes or cancel them, with the former resulting in the changes being applied to the entity and the latter resulting in the entity being restored to its previous state.

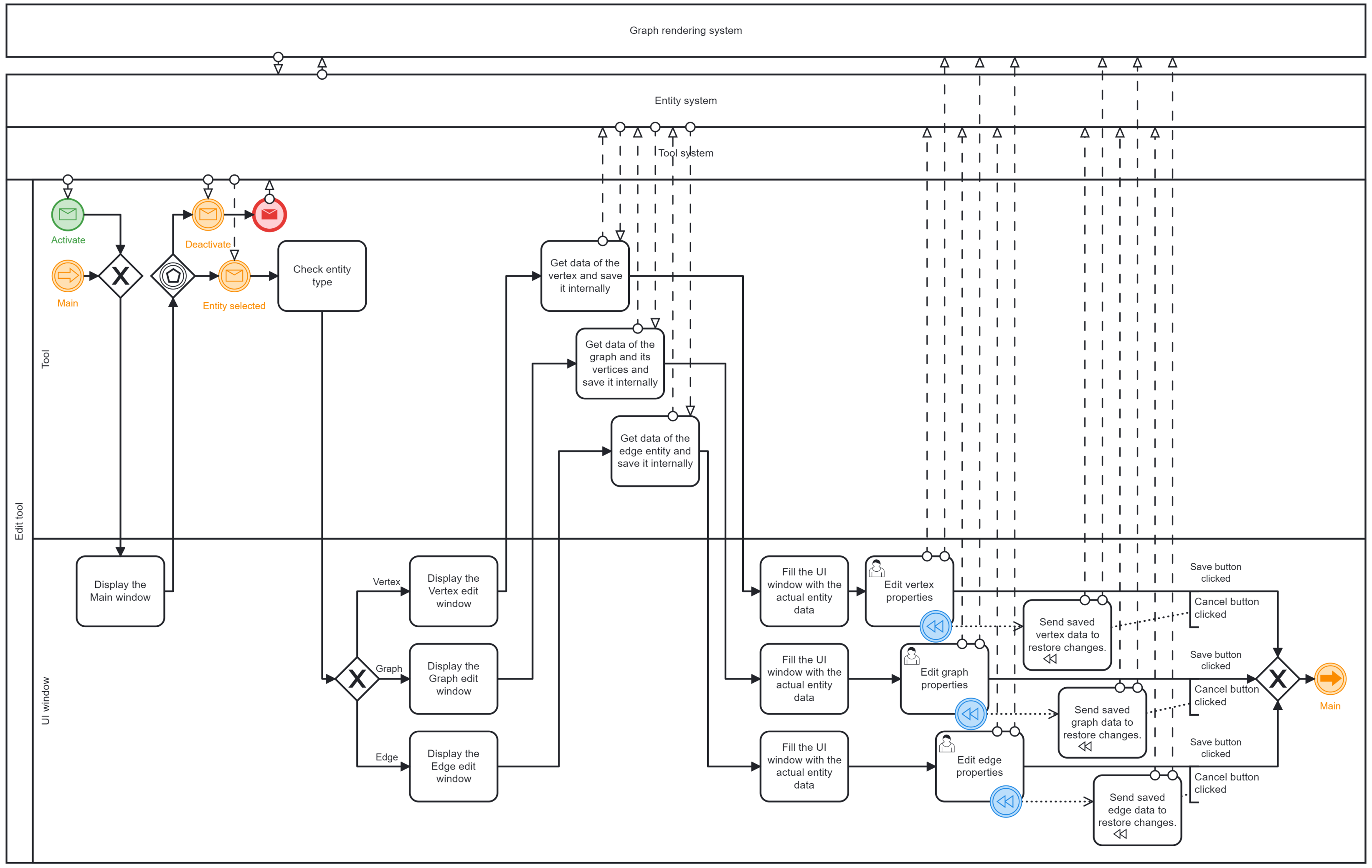

Graph rendering system
Entity system
Tool system
Edit tool
Tool
UI window
Activate
Main
Deactivate
Entity selected
Check entity type
Get data of the vertex and save it internally
Get data of the graph and its vertices and save it internally
Get data of the edge entity and save it internally
Display the Main window
Vertex
Graph
Edge
Display the Vertex edit window
Display the Graph edit window
Display the Edge edit window
Fill the UI window with the actual entity data
Fill the UI window with the actual entity data
Fill the UI window with the actual entity data
Edit vertex properties
Edit graph properties
Edit edge properties
Send saved vertex data to restore changes.
Send saved graph data to restore changes.
Send saved edge data to restore changes.
Save button clicked
Cancel button clicked
Save button clicked
Cancel button clicked
Save button clicked
Cancel button clicked
Main

Figure A.17 Edit tool: process model diagram

A.5 Manipulate

The manipulation tool is designed to enable users to interactively manipulate objects in the scene. With this tool, users can move objects around in the 3D space or rotate graphs around their centers. The tool provides a user interface that allows users to select the desired manipulation mode and interact with the objects using the right controller. The tool updates the positions and orientations of the manipulated objects in real-time, ensuring that their connections and representations are correctly updated.

A.5.1 Main window

The user interface of the manipulation tool consists of a main window with a circular selector and a prompt for the user. The selector allows the user to choose between two manipulation modes: movement and rotation.

In movement mode, users can move entities in the scene by selecting them and using the right controller to control the position of the selected entity. The user interface with the use of this mode is shown in Figure A.18. When this mode is activated, the user is prompted to select an entity in the scene. Once the entity is selected and the controller's trigger is pressed, the position of the entity will change every frame based on the movement of the right controller. The selected entity can also be moved closer or farther away using the thumbstick on the controller.

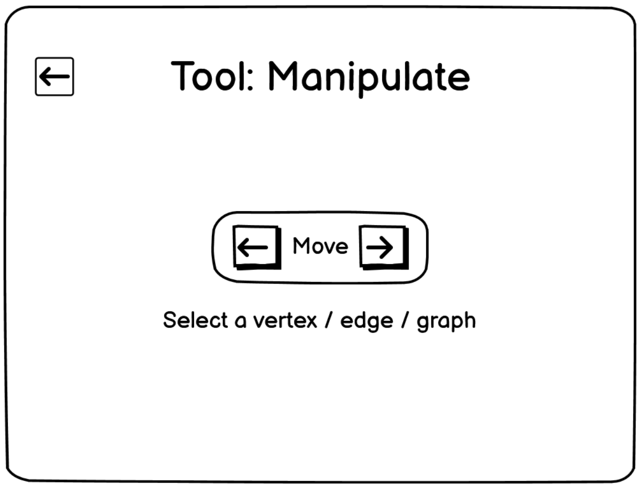


Figure A.18 Manipulate tool: main window in movement mode

Thus, moving a vertex will cause its position to be changed. It is important to note that when the position of a vertex is changed, not only is its position updated, but the vertex and any associated edges are also redrawn to correctly render their connection. Moving an edge will also cause the positions of any vertices it connects to be changed. Moving a whole graph will change the positions of all its vertices.

In rotation mode, users can rotate the selected graph by moving the thumbstick on the right controller. The user interface with the use of this mode is shown in Figure A.19. When this mode is activated, the user is prompted to select a graph entity in the scene. Once a graph is selected, its center is calculated on the basis of the average position of all its vertices. The user can then use the thumbstick on the controller to rotate the graph around the Y-axis or the Z-axis, with the base of the rotation at the calculated center of the graph. Moving the thumbstick forward or backward will rotate the graph around the Y-axis, while moving it left or right will rotate it around the Z-axis.

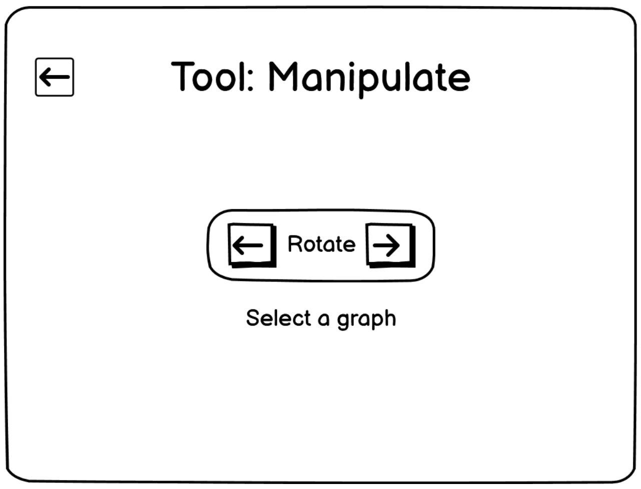


Figure A.19 Manipulate tool: main window in rotation mode

A.5.2 Process model

The following business process model provided in Figure A.20 will provide a concise overview of the functionality of the manipulate tool. It will provide a visual representation of the different processes involved in manipulating objects in the scene. This includes the selection of a component or graph and the manipulation of its position or rotation. The model will also show the different paths that the process can take depending on the user's actions and the results of the manipulation.

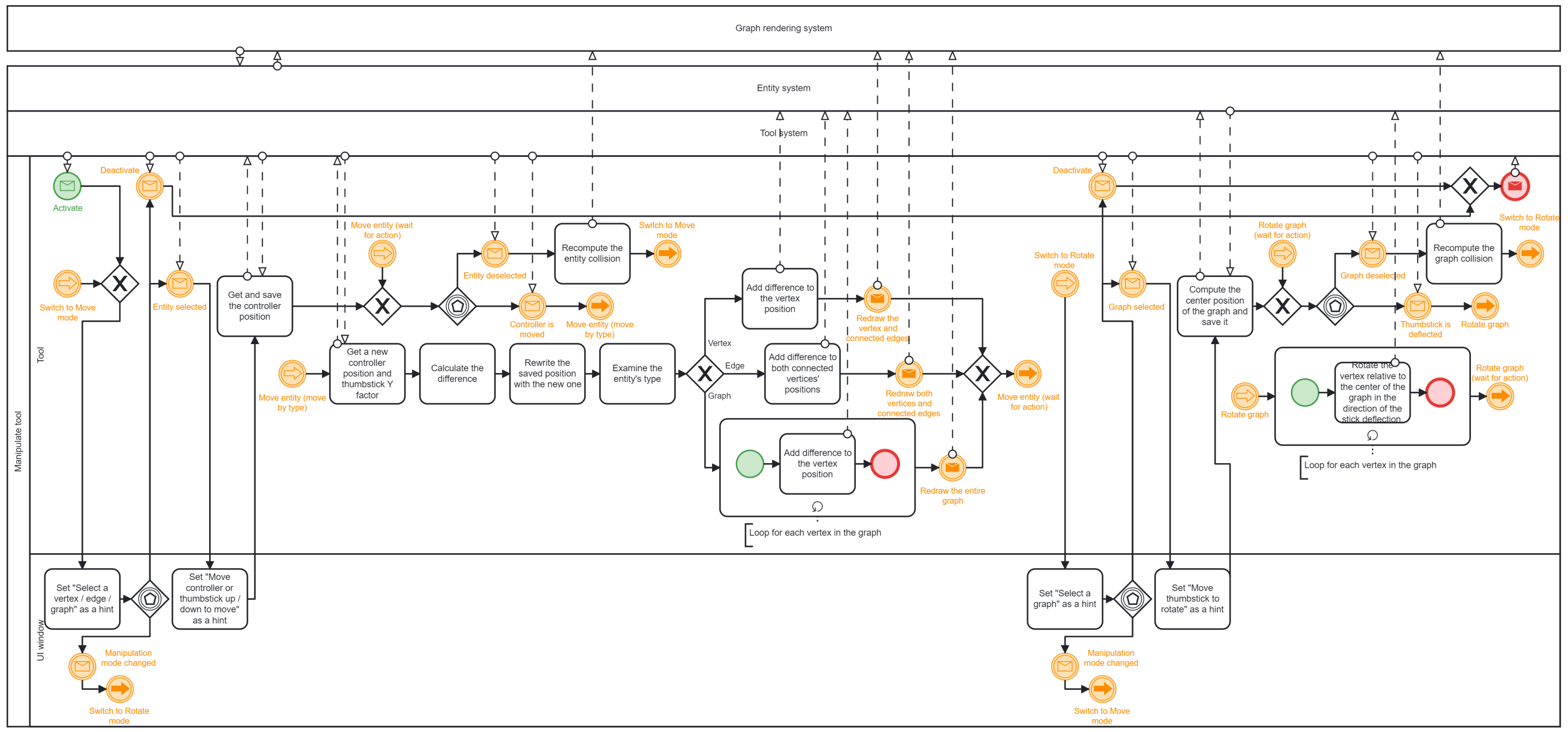

Graph rendering system
Entity system
Tool system
Manipulate tool
Tool
UI window
Activate
Deactivate
Switch to Move mode
Entity selected
Get and save the controller position
Move entity (wait for action)
Entity deselected
Controller is moved
Move entity (move by type)
Recompute the entity collision
Switch to Move mode
Move entity (move by type)
Get a new controller position and thumbstick Y factor
Calculate the difference
Rewrite the saved position with the new one
Examine the entity's type
Vertex
Edge
Graph
Add difference to the vertex position
Redraw the vertex and connected edges
Add difference to both connected vertices' positions
Redraw both vertices and connected edges
Add difference to the vertex position
Redraw the entire graph
Loop for each vertex in the graph
Move entity (wait for action)
Set "Select a vertex / edge / graph" as a hint
Set "Move controller or thumbstick up / down to move" as a hint
Manipulation mode changed
Switch to Rotate mode
Deactivate
Switch to Rotate mode
Graph selected
Compute the center position of the graph and save it
Rotate graph (wait for action)
Graph deselected
Thumbstick is deflected
Rotate graph
Recompute the graph collision
Switch to Rotate mode
Rotate graph
Rotate the vertex relative to the center of the graph in the direction of the stick deflection
Rotate graph (wait for action)
Loop for each vertex in the graph
Set "Select a graph" as a hint
Set "Move thumbstick to rotate" as a hint
Manipulation mode changed
Switch to Move mode

Figure A.20 Manipulate tool: process model diagram

A.6 Remove

The remove tool is an essential component of the application, allowing users to delete unwanted objects from their scene. With the remove tool, users can selectively remove vertices, edges, or entire graphs from the scene, ensuring that their visualizations remain clear and organized. The remove tool's user interface is designed for simplicity and ease of use, with a clear prompt for selecting objects and straightforward remove and deselect buttons for confirming or canceling deletion actions.

A.6.1 Main window

The user interface of the remove tool, shown in Figure A.21, consists of a single window that prompts the user to select the object they want to delete. The window also contains two buttons: Deselect and Remove. These buttons are initially inactive until at least one object is selected.

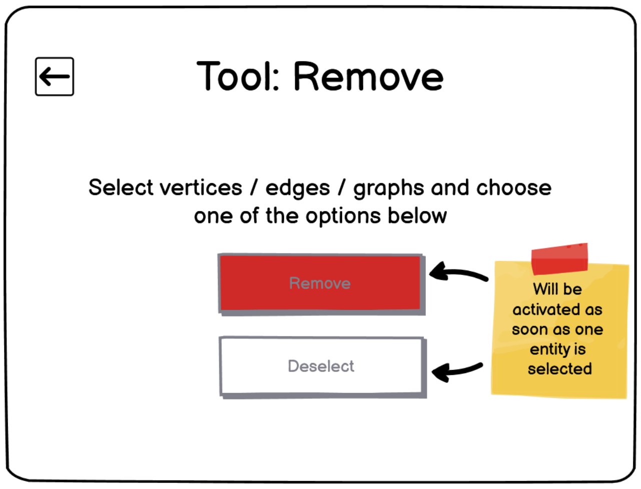

Tool: Remove
Select vertices / edges / graphs and choose one of the options below
Remove
Deselect
Will be activated as soon as one entity is selected

Figure A.21 Remove tool: main window

To delete an object, the user first selects it by pointing the laser right controller at the object and pressing the trigger button. The selected object will be highlighted in red to indicate that it has been selected. The user can select multiple objects at once by continuing to select them with the controller, or they can deselect an individual entity by re-selecting it. Once all desired objects are selected, the user can either deselect them by clicking on the Deselect button, or confirm the deletion of the selected objects by clicking on the Remove button.

Deleting a vertex will also delete any edges connected to it. If the deleted vertex is the last one in a graph, the graph entity will also be deleted. Deleting an edge will only delete its entity. Deleting an entire graph will delete its entity as well as the entities of all its vertices and edges.

The remove tool is designed to be user-friendly, allowing users to select and delete multiple objects at once using the selection modes on the right controller. For example, if a scene contains a graph with many vertices and edges, and the user only wants to keep a few of them, they can switch to graph selection mode on the right controller and select the entire graph. They can then switch back to vertex/edge selection mode and unselect the vertices they want to keep. This allows the user to easily delete the objects they do not need while keeping the ones they do.

A.6.2 Process model

In order to better understand the functionality of the remove tool, it is helpful to visualize its process through a business process model, which is provided in Figure A.22. This model will outline the steps involved in selecting and removing entities from the scene, as well as provide a clear overview of the tool's capabilities and limitations.

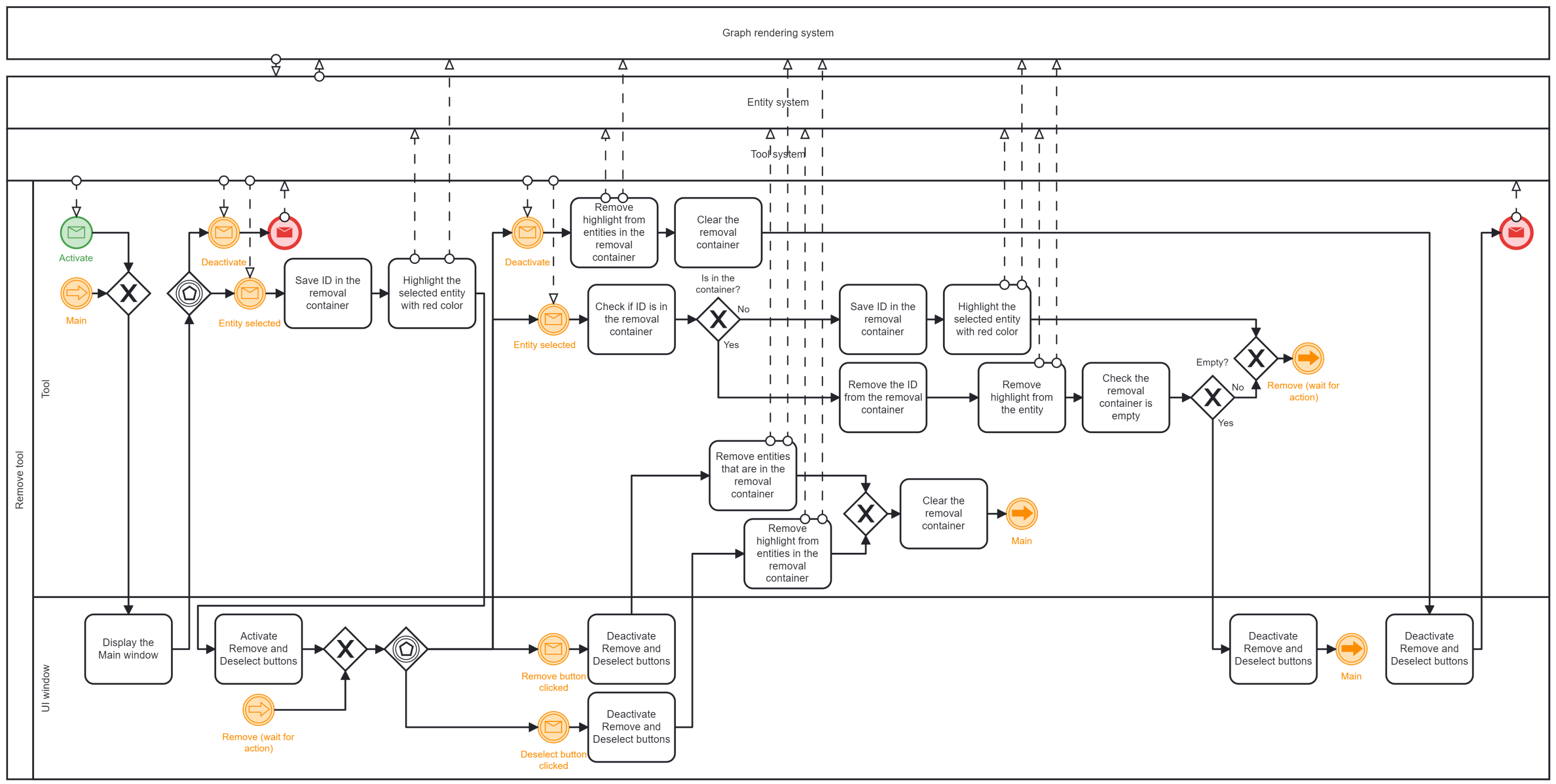

Graph rendering system
Entity system
Tool system
Remove tool
Tool
UI window
Activate
Main
Deactivate
Entity selected
Save ID in the removal container
Highlight the selected entity with red color
Display the Main window
Activate Remove and Deselect buttons
Remove (wait for action)
Deactivate
Entity selected
Remove highlight from entities in the removal container
Clear the removal container
Check if ID is in the removal container
Is in the container?
No
Yes
Save ID in the removal container
Highlight the selected entity with red color
Remove the ID from the removal container
Remove highlight from the entity
Check the removal container is empty
Empty?
No
Yes
Remove (wait for action)
Remove entities that are in the removal container
Remove highlight from entities in the removal container
Clear the removal container
Main
Remove button clicked
Deactivate Remove and Deselect buttons
Deselect button clicked
Deactivate Remove and Deselect buttons
Deactivate Remove and Deselect buttons
Main
Deactivate Remove and Deselect buttons

Figure A.22 Remove tool: process model diagram

Appendix B

Graph generator

This document provides the code for the script used to generate JSON files with graphs of given sizes. The script was written in Python during the testing phase of the application. The script code is shown in Code Listing B.1.

It uses the NumPy and json modules. The number of vertices required for generation is set in the “vertices_num” variable. The number of edges is set in the “edges_num” variable. To limit random positions of vertices, a bounding box with extent, configurable in the variable “vertices_coordinates_max,” is used.

Code Listing B.1 Graph generator script code

```
import numpy as np
import json


def main():
    vertices_num = 30000
    edges_num = 30000
    # maximum bounding box extent in which vertices
    # will be located
    vertices_coordinates_max = 6000

    vertices = []
    # generate vertices positions within the box with
    # size vertices_coordinates_max
    vertices_coordinates = (2 * np.random.rand(vertices_num, 3) - 1)\
                           * vertices_coordinates_max
    for i in range(vertices_num):
        coordinate = vertices_coordinates[i]
        x, y, z = float("{:.3f}".format(coordinate[0])),\
                  float("{:.3f}".format(coordinate[1])),\
                  float("{:.3f}".format(coordinate[2]))
        vertex_dict = {
            "label": i,
            "position": f"X={x}␣Y={y}␣Z={z}"
        }
        vertices.append(vertex_dict)

    edges = []
    vertices_indices_set = set()
    for i in range(edges_num):
        ids = np.random.randint(0, vertices_num, size=2)
        ids_tup = (ids[0], ids[1])
        # generate a random edge until it becomes unique
        while (ids[0] == ids[1]) or (ids_tup in vertices_indices_set)\
               or ((ids[1], ids[0]) in vertices_indices_set):
            ids = np.random.randint(0, vertices_num, size=2)
            ids_tup = (ids[0], ids[1])
        vertices_indices_set.add(ids_tup)
        edge_dict = {
            "from":   int(ids_tup[0]),
            "to":     int(ids_tup[1])
        }
        edges.append(edge_dict)

    graph_dict = {
        "vertices": vertices,
        "edges": edges
    }

    json_object = json.dumps(graph_dict, indent='\t')
    out_file = f"Test_{vertices_num}_Vertices_{edges_num}_Edges.json"
    with open(out_file, "w") as outfile:
        outfile.write(json_object)


if __name__ == '__main__':
    main()
```

Bibliography

1. GORISSE, Geoffrey; CHRISTMANN, Olivier; AMATO, Etienne Armand; RICHIR, Simon. First- and Third-Person Perspectives in Immersive Virtual Environments: Presence and Performance Analysis of Embodied Users. *Frontiers in Robotics and AI*. 2017, vol. 4. ISSN 2296-9144. Available from DOI: `10.3389/frobt.2017.00033`.
2. *Oculus Quest 2: Full Specification* [online]. VRcompare [visited on 2022-12-12]. Available from: `https://vr-compare.com/headset/oculusquest2`.
3. CHERNI, Heni; MÉTAYER, Natacha; SOULIMAN, Nicolas. Literature review of locomotion techniques in virtual reality. *International Journal of Virtual Reality*. 2020, vol. 20. Available also from: `https://www.researchgate.net/publication/340250772_Literature_review_of_locomotion_techniques_in_virtual_reality`.
4. *Unreal Engine* [online]. Epic Games [visited on 2022-12-12]. Available from: `https://www.unrealengine.com/en-US`.
5. *UE5 etchū-daimon station* [online]. subjectn [visited on 2022-12-12]. Available from: `https://www.youtube.com/watch?v=2paNFnw1wRs&ab_channel=subjectn`.
6. *Unreal Engine End User License Agreement* [online]. Epic Games [visited on 2022-12-12]. Available from: `https://www.unrealengine.com/en-US/eula/unreal`.
7. *Overview of Blueprints Visual Scripting* [online]. Epic Games [visited on 2022-12-12]. Available from: `https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Blueprints/Overview`.
8. *Balancing Blueprint and C++* [online]. Epic Games [visited on 2022-12-12]. Available from: `https://docs.unrealengine.com/4.27/en-US/Resources/SampleGames/ARPG/BalancingBlueprintAndCPP`.
9. *OpenXR - High-performance access to AR and VR* [online]. Khronos Group [visited on 2022-12-12]. Available from: `https://www.khronos.org/openxr`.
10. INSKO, Brent. *Standardizing All the Realities: A Look at OpenXR* [Paper presented at the Khronos BOFs conference, Los Angeles California]. 2019. [visited on 2022-12-12]. Available from: `https://www.khronos.org/assets/uploads/developers/library/2019-siggraph/OpenXR-BOF-SIGGRAPH-Jul19.pdf`.
11. *OpenXR Plugin* [online]. Epic Games [visited on 2022-12-12]. Available from: `https://docs.unrealengine.com/4.27/en-US/SharingAndReleasing/XRDevelopment/OpenXR`.
12. FLEISCHNER, H. *Eulerian Graphs and Related Topics*. Elsevier Science, 1991. ISSN. ISBN 9780080867908. Available also from: `https://books.google.cz/books?id=cDktskC2vQcC`.

13. GROSS, J.L.; YELLEN, J.; ZHANG, P. *Handbook of Graph Theory.* CRC Press LLC, 2017. Discrete Mathematics and Its Applications. ISBN 9781138199668. Available also from: https://books.google.cz/books?id=vc5RvgAACAAJ.

14. TURÁN, György. On the succinct representation of graphs. *Discrete Applied Mathematics.* 1984. ISSN 0166-218X. Available from DOI: https://doi.org/10.1016/0166-218X(84)90126-4.

15. ASTURIANO, Vasco. *3D Force-Directed Graph in VR* [online]. [visited on 2022-12-12]. Available from: https://github.com/vasturiano/3d-force-graph-vr.

16. *VR and Simulation Sickness* [online]. Epic Games [visited on 2022-12-12]. Available from: https://docs.unrealengine.com/4.26/en-US/SharingAndReleasing/XRDevelopment/VR/DevelopVR/ContentSetup/#vrandsimulationsickness.

17. POPKO, E.S.; KITRICK, C.J. *Divided Spheres: Geodesics and the Orderly Subdivision of the Sphere.* CRC Press, 2021. ISBN 9781000412437. Available also from: https://books.google.cz/books?id=Mmc1EAAAQBAJ.

18. *Unreal Engine: Objects* [online]. Epic Games [visited on 2022-12-12]. Available from: https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/ProgrammingWithCPP/UnrealArchitecture/Objects.

19. *Unreal Engine: Actor System* [online]. Epic Games [visited on 2022-12-12]. Available from: https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/ProgrammingWithCPP/UnrealArchitecture/Actors.

20. *Unreal Engine: Static Mesh Actors* [online]. Epic Games [visited on 2022-12-12]. Available from: https://docs.unrealengine.com/4.27/en-US/Basics/Actors/StaticMeshActor.

21. *Efficient Occlusion Culling* [online]. Nvidia [visited on 2022-12-12]. Available from: https://developer.nvidia.com/gpugems/gpugems/part-v-performance-and-practicalities/chapter-29-efficient-occlusion-culling.

22. PUPIUS, R. *SFML Game Development By Example.* Packt Publishing, 2015. ISBN 9781785283000. Available also from: https://books.google.cz/books?id=_zjlCwAAQBAJ.

23. LIPPMAN, S.B.; LAJOIE, J.; MOO, B.E. *C++ Primer: 5th Edition.* Addison-Wesley, 2012. ISBN 9780321714114. Available also from: https://books.google.cz/books?id=a4YPBQAAQBAJ.

24. TEWARSON, Reginald P. *Sparse Matrices.* Elsevier Science, 1973. ISSN. ISBN 9780080956084. Available also from: https://books.google.cz/books?id=aU2fYX_TbZ8C.

25. *Unreal Engine: TSparseArray documentation* [online]. Epic Games [visited on 2022-12-12]. Available from: https://docs.unrealengine.com/4.27/en-US/API/Runtime/Core/Containers/TSparseArray.

26. *Unreal Engine: TSet documentation* [online]. Epic Games [visited on 2022-12-12]. Available from: https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/ProgrammingWithCPP/UnrealArchitecture/TSet.

27. *Unreal Engine: Procedural Mesh Component documentation* [online]. Epic Games [visited on 2022-12-12]. Available from: https://docs.unrealengine.com/4.27/en-US/API/Plugins/ProceduralMeshComponent/UProceduralMeshComponent.

28. *Runtime Mesh Component* [online]. Koderz [visited on 2022-12-12]. Available from: https://runtimemesh.koderz.io/.

29. *Runtime Mesh Component: GitHub* [online]. Koderz [visited on 2022-12-12]. Available from: https://github.com/TriAxis-Games/RuntimeMeshComponent.

30. LUEBKE, D.; REDDY, M.; COHEN, J.D.; VARSHNEY, A.; WATSON, B.; HUEBNER, R. *Level of Detail for 3D Graphics*. Elsevier Science, 2003. Morgan Kaufmann series in computer graphics and geometric modeling. ISBN 9781558608382. Available also from: `https://books.google.cz/books?id=M-gl4aoxQfIC`.

31. *Runtime Mesh Component: Quick start* [online]. Koderz [visited on 2022-12-12]. Available from: `https://runtimemesh.koderz.io/start.html`.

32. GILLIS, Alexander S. VRAM (video RAM). *TechTarget* [online]. 2021 [visited on 2022-12-12]. Available from: `https://www.techtarget.com/searchstorage/definition/video-RAM`.

33. *Unreal Engine: Pawn documentation* [online]. Epic Games [visited on 2022-12-12]. Available from: `https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/Framework/Pawn`.

34. *Unreal Engine: Player Controller documentation* [online]. Epic Games [visited on 2022-12-12]. Available from: `https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/Framework/Controller/PlayerController`.

35. *Unreal Engine: Camera documentation* [online]. Epic Games [visited on 2022-12-12]. Available from: `https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/Framework/Camera`.

36. PIGEON, Steven. *Pairing Function* [online]. [visited on 2022-12-12]. Available from: `https://mathworld.wolfram.com/PairingFunction.html`.

37. KAHLER, Andreas. *Creating an icosphere mesh in code* [online]. [visited on 2022-12-12]. Available from: `http://blog.andreaskahler.com/2009/06/creating-icosphere-mesh-in-code.html`.

38. THL A29 LIMITED, a Tencent company; YIP, Milo. *RapidJSON library* [online]. [visited on 2022-12-12]. Available from: `https://rapidjson.org`.

39. *Unreal Engine: Stats system overview* [online]. Epic Games [visited on 2022-12-12]. Available from: `https://docs.unrealengine.com/4.27/en-US/TestingAndOptimization/PerformanceAndProfiling/StatCommands/StatsSystemOverview`.

Contents of attached medium

```
readme.txt ........ a brief description of medium contents
src
  impl ............ source codes of the implementation
  thesis .......... source of the thesis in LaTeX
  exec
    Engine ........ engine libraries and an installer for prerequisites
    Export ........ directory for import and export of external graph files
    Extra ......... graph generator script
    Graphs ........ compiled and packaged application build
    Graphs.exe .... application executable file
text .............. text of the thesis
  thesis.pdf ...... text of the thesis in PDF
```