Philipps-Universität Marburg

Fachbereich 12 - Mathematik und Informatik

Master Thesis

Dynamic Insertion of 3D Objects from CAD Files into Unreal Engine

Matija Mišković

September 2022

Supervisor:
Prof. Dr. Thorsten Thormählen

Research Group Graphics and Multimedia Programming

Declaration of Originality

I, Matija Mišković (Computer Science Student at Philipps-University Marburg, Student-ID: 3139015), confirm that the submitted thesis is original work and was written by me without further assistance. Appropriate credit has been given where reference has been made to the work of others. The thesis was not examined before, nor has it been published. The submitted electronic version of the thesis matches the printed version.

Marburg, 29. September 2022

Matija Mišković

Abstract

Viele der in der Computergrafik verwendeten 3D-Modelle werden mit Hilfe der Dreiecksnetze repräsentiert. ... (max. 1 Seite)

Abstract

text text text text text text text text text text text text text text text text text text text text text text (exakte englische Übersetzung der deutschen Kurzfassung)

Table of Contents

Table of Contents **I**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Goals . . . 2
1.3 Thesis Structure . . . 2
1.4 Related Works . . . 3

2 Unreal Engine **5**
2.1 Unreal Engine Basics . . . 6
2.2 C++ and Blueprints . . . 8
2.3 Networking . . . 11

3 Dynamic 3D Object Insertion **14**
3.1 Loading and Parsing CAD Files . . . 14
3.1.1 File Loading and Sharing . . . 14
3.1.2 Parsing Wavefront OBJ and STL . . . 16
3.2 Runtime Mesh Generation . . . 19
3.3 Object Interaction . . . 24
3.3.1 Grabbing and Translating . . . 24
3.3.2 Scaling and Rotating . . . 25
3.3.3 Duplication, Expansion, Resetting and Deletion . . . 25
3.3.4 Interacting with Individual Components . . . 26

4 Results and Evaluation **29**
4.1 CAD Runtime Importer and Presenter . . . 29
4.2 Comparison to Related Works . . . 29
4.3 Shortcomings and Possible Improvements . . . 29

5 Conclusion **31**

Bibliography **32**

List of Abbreviations **36**

List of Figures **38**

List of Tables **39**

List of Algorithms **41**

Listings **43**

1 Introduction

The topic of this thesis the dynamic insertion of 3D objects, defined in computer assisted design(CAD) Files, into an Unreal Engine program while it is running. Especially important for the project are why this might even a problem and how it can actually be realized. For these purposes an Unreal Engine plug-in was developed which enables such functionality and an additional Unreal Engine program which implements the plug-in and can be used to present and interact with the objects in a multi-user desktop or virtual reality environment.

1.1 Motivation

Virtual reality(VR) is a relatively new field which is constantly seeing a lot of interest and innovation for all the new possibilities in opens up in software development and user interaction. In recent years VR has been used in many companies in various industries such as the engineering, architecture and healthcare and this number keeps on growing. One such company is Inosoft [].
Inosoft is a software development firm in Marburg which was founded in 1993 and has since worked and consulted over two thousand projects for various companies including Viessmann, CSL Behring, Sanofi and many more. They are also very interested in VR and have been working in the field since 2016. Inosoft was also interested in establishing a working relationship with the Phillips University Marburg. As such they reached out with some very interesting projects in the field of VR. Among them was designing and developing a concept to dynamically insert and interact with objects from CAD files in a running Unreal Engine environment.
There are definitely certain scenarios where this could be a very useful tools. As an example, let's take a software Inosoft developed which is used to train workers in a digital production plan while the physical building was being built. This quite a handy tool and has helped quite a bit [**?**]. Slight problems arise when things about the model need to be added or changed. First the changes need to be implemented in Unreal, packaged for standalone use and then redistributed to everyone who needs to use them. It would be a lot simpler if the program could simply open a file and add the new or update objects without ever having to change the version of it.
Another use-case where this could be useful is in collaborative design or presenting 3D models. Instead of having to make the scene and import everything beforehand and distribute this version of the program, simply having a program that can open a file and have the model appear for everyone involved could save a lot of time and effort.
So seeing as there are uses for this technology it makes sense to look into how it could be

done and what the limitations are, as well as looking into why this isn't already officially part of Unreal Engine.

1.2 Goals

The main goal of this work is to develop an efficient and user-friendly plug-in which will make it possible to load 3D objects from the most common CAD formats during the runtime of an Unreal Engine program. Additionally another software will be developed to use the plug-in and allow simple interactions with these objects in a normal desktop window as well as in a virtual reality environment.

Efficiency

The developed plug-in should be capable of handling large amounts of data seeing as the models which can be found in CAD files can be incredibly large, containing thousands or millions of vertices and polygons. If the plug-in were to effect the runtime performance in a significant way, such as causing stutters or freezing the program all together, it would severely worsen the user experience and invalidate the whole point of the program.

Expandability

The field of computer assisted design is very wide and there are countless programs and formats for all the varying use-cases in which it is being used. That is why creating one solution for all of those is incredibly complicated and way out of the scope and possibilities of this project. Instead it is much better to concentrate on creating a simple to use and understand system which can then be further improved upon and adjusted for the concrete cases of clients or projects.

1.3 Thesis Structure

In Chapter 2 the Unreal Engine will be clarified and explained. Seeing as this is both the tool which is being used for development as well as being the software for which the plug-in is being developed, an understanding of how it works and what its limitations are is needed in order to better grasp the project and what problems might arise. It is a rather expansive tool so not everything will be covered, only the more basic aspects and the concrete parts which play a role for this project. Then, in Chapter 3, the plug-in will be analysed, starting of with how the files are parsed and into what sort of form they are transformed in order to be used. After that comes the actual mesh generation mesh, how it is achieved and where extra attention is required. In Chapter 3.3 it will be illustrated in what ways users can interact with the newly created objects, either using mouse and keyboard or a virtual reality headset. In Chapter 4 the developed programs will be presented, evaluated and compared to similar software to see where its strengths and weaknesses are. Lastly in Chapter 5 the reached goals and some possible further projects and improvements will be discussed.

1.4 Related Works

When it comes to this topic there are unfortunately not that many similar works. When it comes generally importing CAD files into Unreal Engine, Datasmith definitely needs to be mentioned.
Datasmith is an official set of tools and plug-ins created by Epic Games, the developers of Unreal Engine, to simplify and streamline the process of importing various CAD formats into the engine []. It is important to note that the main focus of Datasmith is to make the process of transferring a model from a CAD software into the Unreal Engine editor during development smoother and more efficient []. Nonetheless amongst the many features it has it does also contain a plug-in for loading the models in runtime. This plug-in is still in being developed, even upon installation there are clear warnings that the software is still in a beta, so there are some missing features and there also haven't seemingly been many updates to it since the initial research for this project started [].
Outside of Datasmith there are a handful of small plug-ins that can be found which handle this topic, most importantly glTFRuntime [] and Runtime FBX Import []. They were developed by a small team and a single person respectively and are available to be bought in the Unreal Marketplace. The strengths and weaknesses of these tools, as well as Datasmith, will be discussed in further detail later in order better evaluate the programs developed for this project.

2 Unreal Engine

The Unreal Engine is a 3D graphics video game engine, first created for the first person shooter Unreal in 1998 [**?**]. Originally written mostly by Tim Sweeney, the founder of Epic Games, it has since grown an incredible amount and become one of the most popular game engines on the market, only perhaps beaten by Unity []. It has also had many versions since its initial release, first with Unreal Engine 2 in 2002 and then with version 3 in 2006 []. Up until recently Unreal Engine 4, released in 2014, was the latest version but April of 2022 saw the official release of Unreal Engine 5. All of the versions were written in C++ enabling great performance as well as portability, so that the engine is currently supported on a wide range of desktop, console, mobile and even virtual reality platforms [].

In its more than 25 year history the Unreal Engine has been used to create a vast number of incredibly popular and critically acclaimed games such as Fortnite, Hellblade and the Bioshock series, only to name a few. Even though the main use-case has remained video game development, the engine has seen wide adoption in many other industries as well. In film making it can be used to create virtual sets that can be rendered in real time on large LED screens and lighting systems while also tracking around actors and objects using the camera's movement. Epic Games worked with the Industrial Lights and Magic of division Lucasfilm to develop their StageCraft technology [**?**], first used in filming the television show The Mandalorian [**?**]. Outside of these creative fields due to its wast functionalities and ease of use, it has been used to create virtual reality tools to explore building and car designs, as well pharmaceutical drug molecules [**?**].

For the purposes of this project Unreal Engine 4.27 was used and this is the version that will be described unless specified otherwise. Although this technically isn't the latest version and the development of this project started around the same time as version 5 was officially released, there were multiple reason as to why this decision was made. First of all, pretty much any new software release tends to bring with it a number of bugs and quirks which need to be discovered and fixed first. This doesn't always have to be the case but a lot of developers will wait for the software become more ironed out before using it. That is if they even want to use UE5. There are many programs already written in earlier versions of it and not every one of those might truly require the new features UE5 brings with it so the update might not even occur. Also the initial research for the project, which also included learning how Unreal works and how to use it, was done months before the launch.
On the other hand Unreal Engine 4 is a very mature tool which has been used and improved for years now. There are also many sub-versions of it but the decision was

made to use the latest one, 4.27.2, as it should be UE4 at its best and also due to the excellent compatibility between it and earlier version of UE4.

2.1 Unreal Engine Basics

Developing a video game is quite a complicated process and requires various features in order to create a cohesive experience. As Unreal is primarily a game development engine it also has to support many of these functionalities. In total there are more than dozen editors for levels, materials, meshes,physics and user-interface, to just name a few, but for the purposes of this project only a few are of relevance. These are the level and blueprint editors.

The level editor is the primary editor where the levels are created and modified by placing, transforming and editing properties of objects. This is also the default screen Unreal shows when creating or opening a project and what that looks like is shown in figure 2.1. As can be seen in the figure, in the centre of the screen is the level itself. Above it is a toolbar for managing project settings, code, plug-ins and as well a play button which can be used to launch the game inside of the editor for testing purposes. On the bottom the content browser which display all of the assets which are part of the project can be found. This includes meshes, materials, code as well as project plug-ins. On the left is a toolbar for placing various built in objects and on the right all of the objects in the current world, as well as details about the currently selected object can be seen. This is also generally the screen where a developer would import any external assets into the engine directly or through one of the engines importer tools.

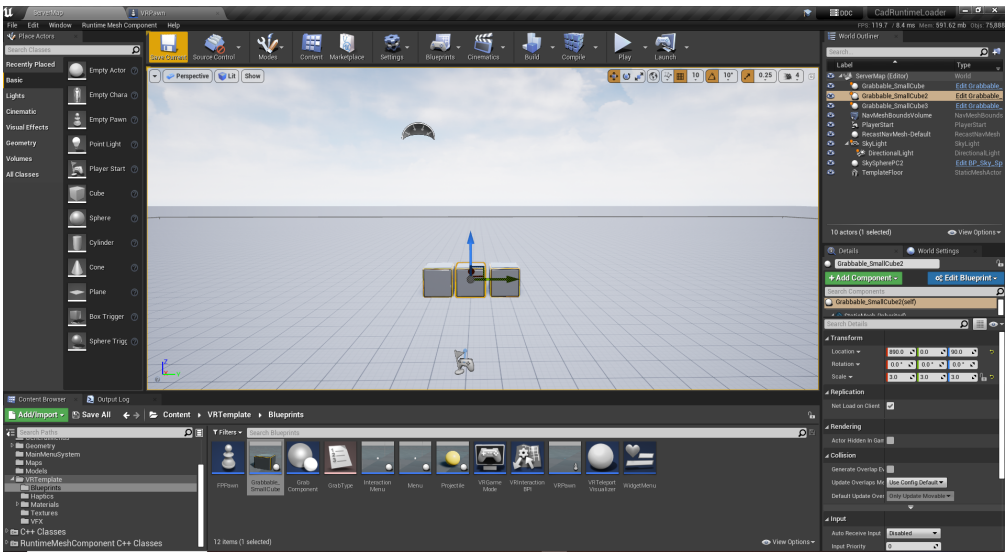

Figure 2.1: Example of what the Level Editor looks like for a project

Actors and Components

In Unreal Engine all of the objects that can be placed inside of a level are called Actors. This includes everything from meshes to particle systems to even the players starting location. This is partially due to Unreal Engines object-oriented nature so having all objects inherit from one base class, in this case Actor, is quite beneficial. Actors can be created and destroyed through code and support 3D transformations like translation, scaling and rotation.

In order to add functionality to an Actor, so called Components are used. Components can offer varying functionalities such as creating sounds, light or movement and once they are added to an Actor, the Actor can access these features and use them for its own purposes. It is important to note that a Component cannot exist on its own and an instance of a Component has to be attached to an instance of an Actor. It doesn't have to be directly attached though, a Component itself can also have several subcomponents. So the Components are what actually makes an Actor what it is supposed to be. One way to think about this is a house. All of the walls, floors, lights and other parts would be the Components, while the house in its entirety is the Actor.

When an Actor is placed inside a level it gets a world transformation which describes the Actors location, scale and rotation in comparison to the world origin. A Component along with that also gets a relative transform, which are again the same values as the world transformation but this time relative to the origin of its parent object. The world transform of a component can be calculating by adding the relative transformation to the parents world transformation. This is very important to keep in mind when components are moved around in a scene.

Pawns and Controllers

Amongst all of the Actor subclasses, there are two which need to be especially highlighted. These are Pawns and Controllers and they form the basis of user-interaction in Unreal Engine.
The Pawn class is the base class for all Actors which can be controlled by a user or through AI. A Pawn determines what a user looks like visually and how they interact with their surroundings either through collision or other physical means. Generally for Pawns that will be controlled by users a further subclass called Character is used. A Character has the additions of a Character Movement, Capsule and Skeletal Mesh Component. The Character Movement Component enables various means of moving like walking, flying, swimming for a character in a scene. It assumes that collision class is vertically-oriented capsule, described in the Capsule Component, and uses this for movement collision. The Skeletal Mesh simply allows for the use of more complex animations which require some sort of skeleton. What this looks like inside the editor can be seen in Figure 2.2. This is the basic template Unreal provides for a third person character. Aside from the already mentioned components, it also has an Arrow Component, which

shows what direction is forwards for the character, and a Camera Component that represents where the view of a user will be in a project.

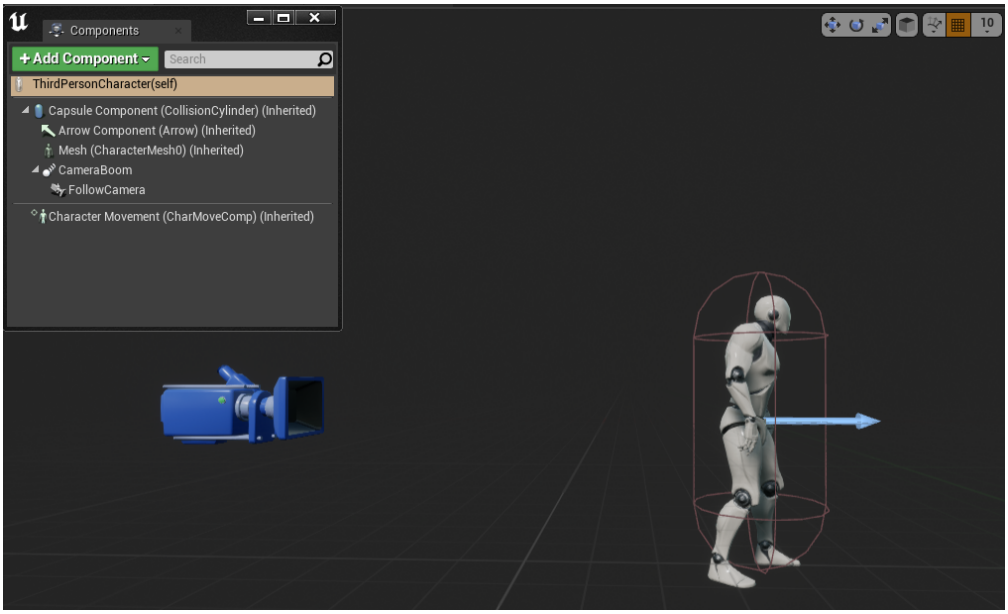

Figure 2.2: Template for third person character

While some functionalities can already be implemented inside of a Pawn, this alone is not enough to get user inputs. For that an additional Actor called a Controller is necessary, specifically a Player Controller. A Player Controller is a non-physical Actor that functions as an interface between a human user and a Pawn. Generally speaking there is a one-to-one relationship between a Player Controller and a Pawn. There are cases where this doesn't have to be the case but for the purposes of this project that is not of interest. This relationship doesn't mean that a Controller can only ever posses one Pawn, just that it can only do one Pawn at a time. The process of gaining controller over a Pawn is called possessing and losing control is called unpossessing. This, alongside slight differences between classes, is why it is important to properly choose whether certain functionalities should be implemented in the Pawn or the Controller.

2.2 C++ and Blueprints

Now that the most important design elements have been explained, the next step is to explain how writing code in Unreal Engine works. When it comes to this regard, Unreal has a rather unique combination of programming tools with C++ and its own Blueprint Visual Scripting system. As already mentioned, Unreal is written in C++ so it makes sense that it would also be used for its programming. What is important to note is the fact that it is not pure C++ that is actually used. Rather, Unreal Engine

has developed its own extensive C++ API, also known as simply Unreal C++, tailored for game development build upon normal C++. This API provides libraries for common game development features as well many built-in classes, functions and utilities. The idea behind this is to have a fully functioning framework that makes the developing process a lot simple and faster than it would've been using standard C++. An excellent example of this is the fact the Unreal C++ support multiplayer and network replication on a core engine level.

The other way to program in Unreal Engine is the Blueprint Visual Scripting system, more commonly referred as simply Blueprints. This system is a relatively new addition to Unreal as it was first released with the launch of Unreal Engine 4. It was meant to be a replacement for the previous Kismet scripting system which was quite complicated and very outdated. Blueprints themself, like many visual scripting languages, use an object-oriented approach for developing. It is a very powerful and flexible tool which is meant to allow designers to create impressive gameplay elements without needing to know how to program. As such it has access to almost all the same frameworks and APIs C++ has. This means that whole games and projects could be made only using blueprints. Likewise the same could also be done with only C++ but such approaches are generally not advised. There is a reason after all why Unreal specifically has both tools and they each have their purposes in development. C++ is advantageous when designing base systems for a project and for writing performance critical features. On the other hand, Blueprints shine when they are used to design the behaviour and incorporate it into the rest of the program. Another great benefit of Blueprints is that, due to their simplicity, allow for rapid prototyping and then these prototypes can easily be translated into C++ if the increased performance is necessary.
Due to this unique mix of tools, a typical workflow for creating features would look as follows. First a C++ programmer would create a new class, add the required features and properties and then make sure that they can properly be accessed in the Blueprints. An example of a header for such a class is shown in Figure 2.3. As can be seen, the code

```
1   #include "MyObject.generated.h"
2
3   UCLASS(Blueprintable)
4   class UMyObject : public UObject{
5       GENERATED_BODY()
6
7       MyUObject();
8
9       UPROPERTY(BlueprintReadWrite, EditAnywhere)
10      float PropertyExample;
11
12      UFUNCTION(BlueprintCallable, Category = "Example")
13      void FunctionExample();
14  };
```

Figure 2.3: Example Unreal C++ Class Header

does resemble normal C++ header code with a few special lines. Most important are the macros that can be found in the 3rd, 9th and 12th line of code. These special lines of code are used to describe the class, property or function in the line below them. This description is used by the Unreal Editor to determine if and how these objects should be presented in Blueprints. As an example the property is set to "BlueprintReadWrite" so that it can be read and modified in Blueprints. There are many specifiers that can be used depending on what the desired outcome is and it is important that these are used properly to mitigate possible problems.

Once that part of the development is done, the new class can be used in the engine. The class itself, seeing as it is C++ code, can't be directly worked with in the Blueprint editor. Instead a new Blueprint class needs to be made that inherits from the C++ class and then that new class can be opened in the editor.
As already mentioned, Blueprints are a visual scripting system, which means that the code isn't represented through text but rather with nodes which are connected among each other. An example as to how this looks like in the Blueprint editor is shown in Figure 2.4.

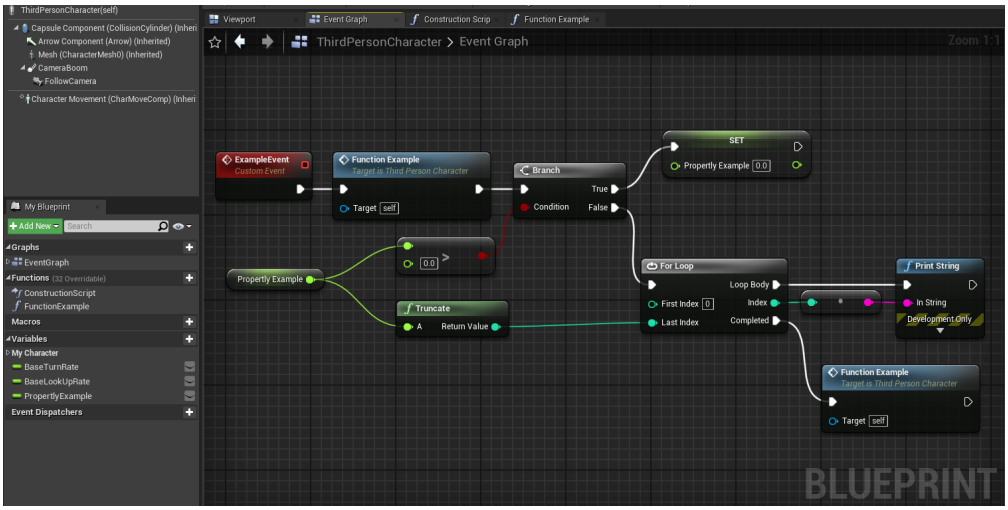

Figure 2.4: Example of Blueprint Code in the Blueprint Editor

In the top left side of the screen the components of the current actor can be seen again. Underneath that is an overview of all the functions, graphs and variables contained in the blueprint. In the centre of the editor is the event graph of the class, this is where all the various events that can happen to it are handled. In the example shown it needs to be noted that the view is heavily zoomed into a single event in order to better see the individual nodes. Typically there will be many events all across the graph and also sub-graphs to keep the code readable.
The execution flow of the code is represented by the white line connecting the nodes. The nodes themselves have varying functions and are accordingly colour coded for better

understanding. Red nodes represent the starting point of an event. This can be triggered by many actions including collision, player inputs or other events. Blue nodes are either functions or event calls and green nodes are usually used for getting values. Lastly grey nodes represent macros or flow control nodes. This is where the typical programming tools such as if conditions, for and while loops can be found. Depending on if they are needed, input and output pins can respectively be found on the left and right side of a node. These are connected using lines that automatically match the colour of the value, which are also colour coded, and can only be connected to other pins of the correct type. All in all, these properties and features make using Blueprints quite simple and almost play-like, which makes them accessible to a wider audience, while staying quite powerful.

2.3 Networking

Everything that was discussed so far mostly relates to what happens in a single instance of our program. Nonetheless, as already mentioned, networking is a big part of Unreal Engine and is required to understand the steps that are needed in order to create a multi-user project.

When the program runs in a standalone mode, all of the objects that make it up exist on the local machine which is running the program and only that machine. For a network multi-user program, Unreal Engine uses a so-called client server model. One computer acts as a server and hosts a session that can be joined by other user as clients. The server is what connects all of the different users and enables their communication with each other. The instance running on the server is the true, authoritative world instance. In order words this is where the multiplayer is actually happening. The clients only have copies of this world. The server dictates the clients what Actors exist, how they should behave and values their variables should have. The clients then use this information to approximate what is happening on the server in their own copy of the world. The clients only really control the Pawn and Player controller that they are assigned to. One thing to note is that while a copy of a Pawn exists in every instance of the program, the Player Controller only exists for the owning player and server. This means that a local Player Controller is completely unaware of the existence of other Player Controllers.

In total there are three network modes in which an Unreal project can run in: standalone, client and server. For the server mode there is a further classification into listen and dedicated servers. A listen server represents a user hosting a session through their local machine. This means that they function both as a server and a client simultaneously. The benefit of this is its relative simplicity, especially with Unreals already existing tools and support for many popular online subsystems such as Google, Amazon and Steam. A big downside of this approach is the extra load that is put on the server machine, as it also has to handle user-relevant features like graphics. Also the user that hosts the server can get a slight advantage from the non-existent latency, although this is only relevant in specific use-case.

A dedicated server on the other hand runs "headlessly", which means it does not have to render any visuals and isn't controlled by anyone locally. This means that most of the resources available to the machine can be used for hosting and moderating the program. Unfortunately this requires a separate computer with its own network connection, not to mention a lot of complex work to be properly configured.

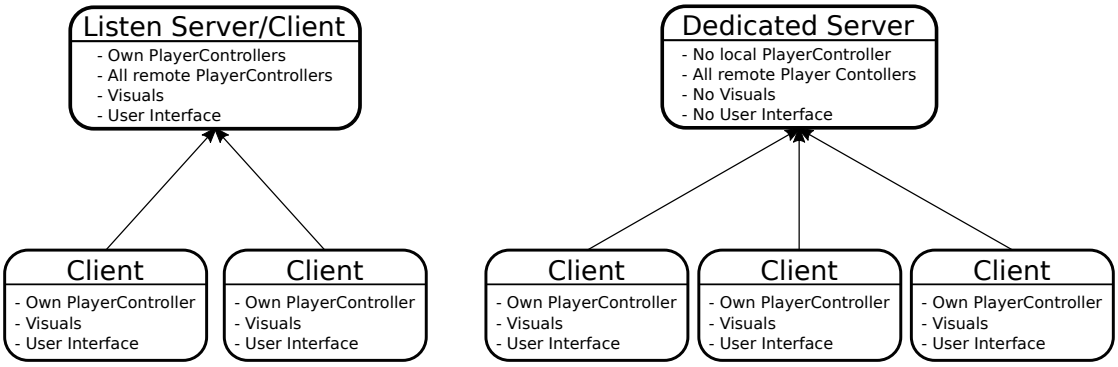

Figure 2.5: Example to demonstrate differences between a Listen and Dedicated server

The actual information sharing and interaction between users through a server is done through replication and remote procedure calls. In order to use replication for a variable, its boolean specifier "Replicated" needs to be set to true. Now when the variable's value is changed on an authoritative Actor, usually on the server, the change is automatically sent to the connected remote copies of the Actor. Likewise if a variable is changed locally in a clients instance, this will not be replicated to the server or other clients. This doesn't mean that every variable needs to be replicated, as that could cause problems in network traffic. Rather it is important to use replication carefully and replicate only variables that require it.
Remote procedure calls(RPCs) are functions which are called from a machine but executed remotely on another machine. They are also known as replicated functions. In order for a function to become an RPC, the keywords server, client or multicast need to be added to its definition. The server and client keywords simply mean that the function should be executed on specifically on the server or client. Multicast means that a function will be called on every instance of an object. Another peculiarity of RPCs is that they have no return value. In order to achieve that another RPC is needed which returns the output from the remote to the local machine.
All of this is only a small section of all the features that Unreal Engine is capable of but for the purposes of this project this should suffice as an introduction to the engine and make understanding the rest of the work easier.

3 Dynamic 3D Object Insertion

In order to realize the insertion of a 3D object from a CAD file, a plug-in with this functionality was developed simply called CADRuntimeImporter(CRI). Alongside it a standalone Unreal prototype project, named CADRuntimePresenter(CRP), was made that incorporates CRI in order to demonstrate how it can be used for a multi-user desktop and VR environment. The whole mechanism can be split into three major sections: opening and parsing the files, generating the objects and lastly user interaction with said objects. How all of that was implemented and what sort of advantages and disadvantages these approaches have, will be discussed in this chapter.

3.1 Loading and Parsing CAD Files

3.1.1 File Loading and Sharing

The first step in creating an object in runtime is naturally opening the desired file and getting the required data from it in runtime. As Unreal Engine is written in C++, it is not surprising that opening up a file isn't too much of an issue. What makes this simpler is the fact that Unreal also offers this in their FileHelper class with the functions LoadFileToString() and LoadFileToArray(). The first function can load a text file into a string, while the other loads binary files into an array of bytes. This is only directly available in C++ and therefore had to be exposed to Blueprints. As this functionality is more of a utility, it isn't the best idea to attach it to a specific object. Luckily for such purposes Unreal offers Blueprint Function Libraries. This is just a special type of Unreal C++ class in which only static functions can be defined. These functions then become available to be used in any Blueprint without needing any instanced objects.

Something that is slightly more complicated is actually choosing the file. Here Unreal does technically offer the ability to open a file dialog but this is a strictly developer only module and can't be used in finished products. Even neglecting that, it wouldn't work in VR so a separate solution would have been needed anyway. Due to this a file picker in CRPs UI had to be written. The end result of that can be seen in Figure 3.1. The design is rather simple but offers all the necessary functionality, especially that it is compatible with VR and only displays supported file formats.

For a single user this would be enough, they could choose a file and then it could be parsed for object generation. Complications arise once there are multiple users involved. If a user were to open a file in such a scenario, the object would appear only in the world of that user and not for other users or even on the server. This could cause many issues

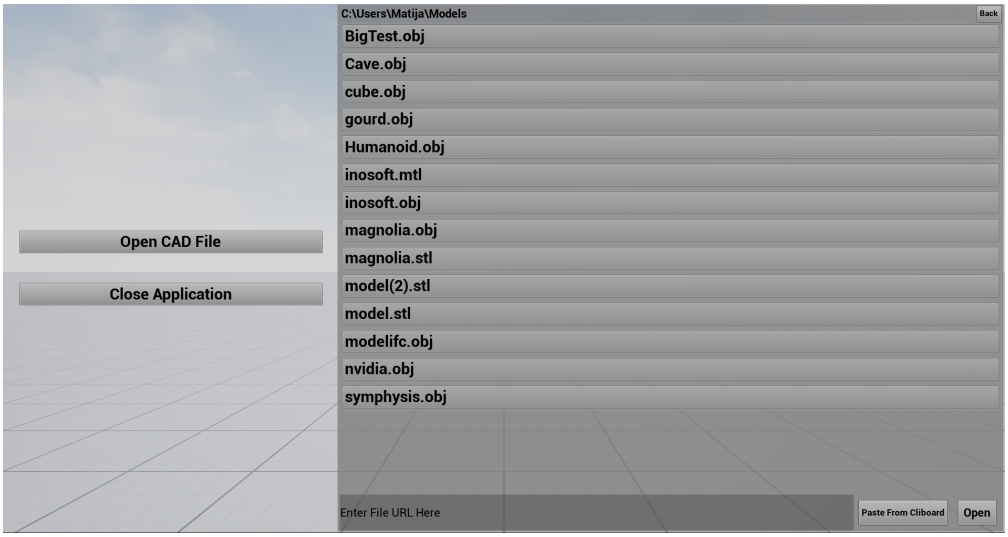

Figure 3.1: File Picker for CRP

since the clients world would not match servers, which is the authoritative instance. The problem lies in the fact that the new object needs to be created on every client and on the server. In order for that to happen every machine needs access to the required data, not just the client who has the file available on their machine.

One way of sharing this data is saving it in a replicated variable and having it handled by the automatic replication system. This does actually work but it has one fatal flaw which makes using it not viable. That is the size limit of replicated variables. The size limit for arrays is 64 kilobytes and for strings it seems to about the same. For smaller files this would be a perfect solution but unfortunately CAD files tend to be too big for it.
So instead this problem was solved by having the file be uploaded to a server and then be downloaded by the rest of the clients and server. For such purposes Unreal offers the HTTPModule interface, which uses the popular and powerful libcurl library, to create HTTP requests. A file could then be uploaded using a POST request and downloaded using a GET request. For this a simple file server which can handle such requests was written in python. The only problem is that the clients and server need to know where and what file was uploaded in order to make the correct request. This is where replication comes in handy. The client that uploads the file can tell the server where it was uploaded through a replicated string, which is most likely going to be within the size limit, and then the server can propagate that information to the rest of the clients which then make the GET requests.
Seeing as for most users only the link to the file matters, this means that the client that wants to open a file doesn't even need to have it locally on their machine. Instead they can simply input a link to a service like Dropbox or Google Drive and have everyone

download those files. The UI for that is also part of the file picker and can be seen at the bottom of Figure 3.1.
There are technically other libraries and plug-ins that could be used to enable file sharing with more complicated protocols but that wasn't necessary for this prototype project. The HTTPModule is simple to use while offering all the needed functionalities and avoids having to rely too much on third-party libraries.

3.1.2 Parsing Wavefront OBJ and STL

Once every instance of the program has the desired file, the next step can begin which is parsing the data. What data is available and how it is stored can vary heavily from format to format. Generally they will all have the vertices that define the mesh but outside of that colour, material or anything else isn't guaranteed. This is the case because CAD formats tend to be highly specific for their uses cases, as well as proprietary for the CAD software they were developed for. This makes supporting many CAD formats quite difficult, especially those that aren't well documented or don't even have publicly available documentation. Considering the scope of this project, spending too much time writing parsers for as many formats as possible was not feasible.
Instead the decision was made to use well known and widely supported formats, like OBJ and STL. One of the biggest benefits of this approach is the already existing support that these formats have. Many CAD softwares support exporting to one of these and even if they don't, there are probably tools with which the files can be converted. This saved a lot of time in the development, as only a few parsers had to be written and for those that had to be implemented, the process was fairly simple due to all the existing resources on the formats.

Wavefront OBJ

Wavefront OBJ, or simply OBJ, is a geometry definition file format developed by Wavefront Technologies for their Advanced Visualizer animation packages. It is a neutral, open source format which has been widely adopted and has good import and export support from almost all CAD software.
Another reason why OBJ was chosen is the fact that is can be directly read through any text editor. This helped out a lot in early stages of development where the primary goal was to prove that the concept worked. Being able to see and read the data made it simpler to write a parser in the first place, as well as comprehending what was going on with the data at later stages.
The format represents 3D geometry in the form of vertex positions, vertex normals, texture coordinates, polygonal faces and groups of faces. These geometries can also use materials indirectly through referencing materials defined in a separate MTL file. Every entry in an OBJ file is represented through a single line, starting with an identifying tag followed by the value of the entry. What these entries can look like is represented in Table 3.1.
In order to parse their values, most of the entries can be regarded one-by-one. Vertices,

Tag	Example Value	Definition
v	0.2 0.3 0.5	3D Vector representing 3D Vertex
vn	1.0 0.5 0.0	3D Vector representing 3D Normal
vt	0.5 0.25	2D Vector representing Texture Coordinate
f	1/1/2 2/2/5 3/3/5	Polygonal Face made from Vertices, Normals and Textures
usemtl	Stone	Defines what Material should be used for following faces
g	Door	Defines a polygon group

Table 3.1: Relevant types in OBJ format

normals and texture coordinates are vectors and can be saved in separate vector arrays in the order in which they appear. How these and the rest of the values are used will be explained later, for now it's only important how they are saved.
Faces, materials and group are more complicated as they define how the rest of the data is put together to make the object. A face represents a polygon defined through lists of vertex, texture and normal indices in the format "vertexIndex/textureIndex/normalIndex", as can be seen in Table 3.1. It is important to note that these are the indices of the values and not the values themselves. Also the indices for vertices, texture coordinates and normals are separate and based on when the entry was defined in the file starting with one. So both a vector and normal can have their respective index be 1. The polygons themselves tend to be triangles but can also have more sides. This isn't ideal as later for generating the mesh, only triangles are supported but there is a simple way to solve this. The vertices are listed in a counter clockwise order so that any polygon can be represented through a triangle fan, as is shown in Figure 3.2. So faces that define polygons with more than 3 edges are replaced through multiple triangular faces.

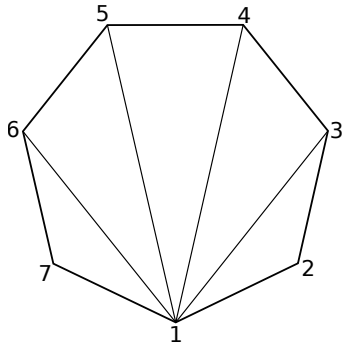

Figure 3.2: Polygon represented through a Triangle Fan

A group determines what faces are combined in order to make a component of the larger object. The "usemtl" tags tells what material should be used on the faces and that material is used until the next tag appears. The actual materials are defined in an MTL file, which works similarly to an OBJ, just with different tags and values. Some of the more important entries can be seen in Table 3.2. This MTL file of course also needs to be shared to every client. The values themselves are saved in a Map where the keys are

the material names and the values are arrays of the values.

Tag	Example Value	Definition
newmtl	Stone	Defines the name of a new Material
Ka	1.0 1.0 1.0	Ambient colour
Kd	0.0 0.0 0.0	Diffuse Colour
Ns	1.0	Specular Exponent
d	0.5	Transparency

Table 3.2: Relevant types in MTL format

As the information of these values heavily relies on each other, they needed to be combined and saved in an array where every array entry represents one component of the whole object. The entry consists of the face values and marks where materials need to be switched.
The only major problem with OBJ is the fact that the coordinates don't have units, meaning they can't be scaled to properly represent the designed size. Instead everything is scaled with same factor so that at least the scales between objects made in the same scene stay the same. Also the created objects can later be scaled by users to better resemble the desired size.

STL

STL is a file format native to the CAD software created by 3D Systems in 1987 []. The format has gained a lot of support in many different software packages, especially for 3D printing software where it is one of the default formats. STL files only describe the surface geometry of a three-dimensional object with out any additional information about colours, materials or groups. The geometry is described in raw, unstructured triangles defined by a normal and three vertices. The way a triangle is defined in the file is shown in Figure 3.3.

```
facet normal x_n y_n z_n
      outerloop
            vertex x_1 y_1 z_1
            vertex x_2 y_2 z_2
            vertex x_3 y_3 z_3
      endloop
endfacet
```

Figure 3.3: A triangle facet represented in STL

While the structure suggests that multiple possibilities are possible for loops, the facets can only be triangles. In order to represent a full model the triangles are simply listed one after the other. In order to parse it, the vertices and normals are saved in arrays and faces are generated to point what vertices make up the triangles. This is important for generating the mesh later. While it is a simpler format compared to OBJ, it is excellent

when only the shape is relevant. It does unfortunately suffer from the same scaling problems as OBJ and that is handled in the same way as well.
Overall with these two formats the majority of the most common and well-known CAD formats are covered. Ideally more formats would be covered and specific parsers for every format would be written to get the most accurate results but for the limitations of this project, this is a very practical solution.

3.2 Runtime Mesh Generation

After the files have been parsed and the needed data was extracted, the actual process of creating a new 3D object starts. For that, a way to tell Unreal to generate a mesh is required and Unreal does in fact support such a feature in the form of so called Procedural Mesh Components(PMC). These components can be created in runtime by giving them the required mesh data and letting them generate themselves. In the first phases of this project it was explored as to how viable using these components were, seeing as they initially seemed to be exactly what was needed. Getting it to function took a bit of work, mostly due to Unreal Engines generally bad documentation, but the first results with small objects were quite promising. Unfortunately the problems started to arise with larger objects where the performance would drop significantly or the Unreal editor would simply crash. The reason for this lies within the procedural mesh components themselves. As the name implies, these components are supposed to be generated by some form of procedure which generally won't create nearly as many vertices as a large CAD file can. On top of that the underlying architecture for a real procedural system for expensive geometrical operations would be quite different to that of a runtime gameplay framework []. PMC is also a relatively late addition to Unreal Engine, first appearing in version 4.8 in 2015. So even though it is designed for a quite similar purpose, it is different enough that it couldn't be used.

Due to this a different method of getting Unreal to generate meshes had to be found and during this time a the most important question for this project came up. Doesn't Unreal technically already support runtime mesh generation? Let's say, as an example, there was an Actor that had some sort of static mesh attached to it. Unreal could spawn in this new Actor without a problem. So shouldn't it be able to create such an Actor from data parsed from a file?
A definitive answer for that question couldn't be found but through some research a few speculations can be made. The biggest reason for this probably stems from the main purpose of Unreal Engine. Unreal has always and will probably always be a video game engine. As such it is designed and optimized for use cases that happen in video games. In a game, every asset and model is carefully crafted and placed in a level. These objects are already known to the game and are packaged within it in an optimized state. The game knows exactly what to do with these and where to load them. That is a part of the core functionality of Unreal. But it isn't exposed directly to a developer because adding externals models to a game isn't generally required. Why should a game depend

on the user having some specific type of file to load? Those need to be within the game itself. Not to mention the problems and security issues an external file could cause.
Another factor that comes into play are the hardware limitations that existed throughout most of Unreal's lifespan. Computers and gaming consoles weren't always as powerful as they are now, especially consoles tend to be very underpowered machines. As such a lot of work went into optimizing games so that they would run smoothly on their target platforms. Even nowadays with modern systems that are way faster, optimizing is a big part of game development. Especially important for that is managing what is loaded in and when as the systems can have limited memory. Some games will use a loading screen to hide the loading, other games might use a gameplay sequence that slows down the player so the game has time to load in the next assets. There are even games that would restart the console they were running on without letting the player know when the memory was full []. How these assets were stored also played a big role. Some games would use the same asset but with different colours to save space or some games saved the same asset multiple times so it could be accessed from storage quicker []. CAD files on the other hand tend to be rather big and just creating the files is already a very demanding job which requires good hardware. Due to all of this, the idea of just creating a new external model using Unreal's functionality isn't of value to game development and isn't directly exposed.
However Unreal isn't just a game engine any more. As already mentioned it has gained popularity in various fields and those have very different demands compared to games. This has led to many new additions to Unreal, most importantly Datasmith. Datasmith is a set of tools and plug-ins developed by Epic Games with the goal of streamlining the process of importing CAD files into the engine during development. It is a relatively new addition to Unreal as it was first released around 2016. CAD objects are very different to objects used in game development, they focus on creating geometry for manufacturing and production while game objects are more focused on looking a certain way and being optimized. Due to this, it is clear that this addition isn't meant for game development. But it wasn't until August of 2021 with the release of Unreal Engine version 4.27 and with it the release of the Datasmith Runtime plug-in that it gained the ability to import meshes in runtime. This plug-in, as it is still very new, is in beta and still being worked on. Most importantly it shows that with access to the mesh generation functionality of Unreal it is possible to create meshes from external data in runtime.

But the demand for such a functionality has existed for a lot longer and Unreal's existing solution with PMC wasn't good enough, which led to the development of Runtime Mesh Component (RMC). RMC is a third-party plug-in developed that exposes the mesh generation capabilities of Unreal in a much more efficient and feature-rich way compared to PMC. It promises 50-90 % lower memory usage and 30-100 % lower render thread CPU time [] compared to PMC. These claims were checked and the results do match the expected improvements. It is also completely free and has been used in many projects even in larger companies []. This is why it was finally decided to use RMC for the purposes of this project. While ideally this project wouldn't need to rely on an unofficial plug-in, trying to recreate what is available with RMC, which has been around for more

than 6 years and had more than 40 people contribute it, is not a feasible endeavour. Instead, for the limitations of this project, it is much wiser to use this tool and apply its capabilities for the purposes of generating CAD models.

Generating a CAD Model

Before a model can be generated, there needs to be an Actor to which the components can be attached to. Technically this could be any Actor, even the Player Character, but it makes more sense to create a specific Actor for these purposes. In the plug-in such an Actor is defined in the CRIObject Blueprint class. Actors of this class are also going to be where the values from the parsed files will be stored, as well as the Actors that call the function to generate the new components on themselves.
In order to spawn in a new Actor of this class, the Unreal SpawnActor function can be used. This function just needs to know what class should be spawned and where. The first parameter is easy but the second one has a few more options. Depending on what is needed all these Actors could be spawned in the same place or a user could enter the coordinates. For the purposes of CRP it was decided that the location of the new Actor is going to be where the user adding the object is looking. For this the users camera rotation is taken as a forward vector, multiplied by a distance and added to the players location. The distance is based on the size of the spawned object so that collision is avoided but it also isn't spawned to far from the user. Also this spawning process happens as soon as clients start downloading the CAD files so that once the files are parsed, there is a place to save the data. The Actors will be in the world but won't be visible or interactive as they don't have any components yet.
Once the data has been saved, flags in the Actor get set in order to notify it that generating can begin. This is done because there is no guarantee that the CAD and material file will be downloaded in any specific order and generating before all the needed data is there would cause problems. The material flag is also only used if the file uses it and the user specified that it should be created with materials.
The whole model isn't generated all at once because this could cause the program to stutter or freeze for the duration of the process for larger files. This happens due to this running on the same thread as the main thread, which means that nothing is rendered until this is finished. Instead each component gets generated separately from the rest in the order that they are defined in the file. This is realized through the tick event that Actors can possess. A tick event is simply an event that gets called every frame or in some other interval. As the components themselves are comparatively small, generating one doesn't cause a significant performance hit that could freeze the main thread. So basically by doing this, the performance cost of generating the model gets spread across every component and becomes practically negligible. Another benefit of this is the fact that the components themself start appearing one after the other in the world, creating an interesting-looking animation. Ideally multithreading would be used for even better results but Unreal is rather specific about that due to its built-in garbage collection [].
With every tick the GenerateMeshComponent function is called and a counter is kept as to know what components have already been dealt with. The function is implemented

in C++ as the performance is necessary but in order to call it from Blueprints it was implemented in a Blueprint Function library, just like the file reading function. It could also have been implemented as a native function of the CRIObject class but, as this is supposed to function as a plug-in, it didn't make sense to limit it to one specific class. This way the functionality can be used in more ways depending on what the user needs. One such way is implemented in CRP and will be demonstrated later.
How the function works can be seen in Algorithm 1 which shows the process in a simplified pseudocode.

```
Input Actor, ComponentIndex, Vertices, TextCoords, Normals,
        ComponentData, Materials, UseCollision;
RMC = new CustomRuntimeMeshComponent()
RMC.ID = ComponentIndex
RMC.AttachTo(Actor)
RMC.UseCollision = UseCollision
Center = FindBoundingBoxMiddle()
RMC.SetRelativeLocation(Center)
Vertices.Translate(−Center)
RMC.CenterVertex = Center
Secitons = GetComponentSections(ComponentData)
SectionMaterials = GetSectionMaterials(Materials)
for i := 0 to Sections.Length do
    Faces = GetFacesForSection(Sections[i])
    Material = SetupSectionMaterial(ComponentMaterials[i])
    RMC.CreateSection(i, Vertices, Faces, Normals, TexCoords, Material)
```

Algorithm 1: Pseudocode for generating a Mesh Component

As can be seen the function takes quite a few inputs but all of those necessary. The first step is instancing a new Runtime Mesh Component object. In this case it is a slightly customized subclass of RMC that contains a few more variables that can be quite useful. One of those being the index of component, which is used when interacting with components. After that the component is attached to the Actor generating it as it cannot exist in a scene on its own. Based on user wishes the mesh can be created either with or without collision. Next the a bounding box is generated from the vertices of the component and the centre of that box is determined. This is done because when a component is added to an Actor, its relative location is (0,0,0) so it's in the centre of the Actor. The mesh on other hand will be placed where the vertices are. Once the whole mesh is generated it will look exactly the way it should but there will be problems with certain interaction. Let's take rotation as an example. Since the position of the component is technically in the centre of the object, rotating it will cause the mesh to rotate around the whole object instead of itself. In order to fix that, the component is placed where the centre of the mesh will be and the vertices are translated the same amount just in the opposite direction. These two translations cancel each other out

so that the whole mesh ends up looking unchanged. Meanwhile the position of the component matches the centre of the mesh so that rotating and scaling work properly. Alternatively the component could be moved and translated in specific ways to simulate such a rotation but this is many times more complicated and unnecessary. The location of the centre is also saved in the custom class as it is required for some interactions that will be explained later.
Then the component is split into sections and the material values for the sections are extracted. An RMC can contain one or multiple meshes and these are regraded as sections. In this case the mesh is split according to the materials seeing as a section can only use one material. This way a component can have multiple meshes, instead of having to create a component for every material change. Lastly for every section, all the indices defining its faces are saved in one array and the appropriate material for it is setup. In Unreal a completely new material cannot be created at runtime. Instead an already existing material is needed as the parent for an instance of a dynamic material. Such a dynamic material can be create during gameplay and the values that define it can be altered as well. There is just a slight problem du to the plug-ins current limitations. The materials in Unreal are physically based rendering(PBR) materials, while the MTL uses Phong shaded materials. These two ways of rendering are vastly different and there isn't one true way of converting between them. That's why the diffuse colour, specular exponent and opacity are used to approximate the appropriate values for the PBR material. How well this works depends on what material is supposed to be represented. This is slightly unfortunate but what is more important is the fact that an appropriate material could technically be created. If a new format, that supports PBR, were to be added the whole mesh generating approach would still be the same just with the correct values. Also technically two parent material are needed, one for opaque and one for translucent materials because of the way Unreal handles opacity.
Finally the RMC takes all of the inputs that it needs and creates a section. This is then repeated for every section of every component until the whole mesh has finished generating. The model will appear in the world in the position where the Actor was spawned and the result of such a process can be seen in Figure 3.4.

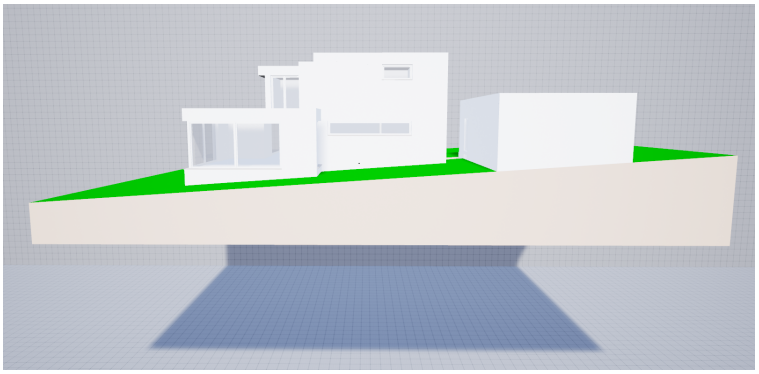

Figure 3.4: Loaded CAD Model in Unreal Engine

3.3 Object Interaction

While being able to generate 3D models on its own is very useful, it would be severely limited if it just stood in a place and the users had no way of doing anything with it. That is why it is important to take a look at how it possible to interact with these objects. There are many ways in which this can be done so the presented solutions are just what was implemented in CRP to demonstrate some essential and some interesting interactions that can probably find use in most projects.
Almost all the interaction was written in the Player Controller, while movement and looking around is handled in the Pawns. This was done because the Pawns have to be different in desktop and VR mode and that would mean writing pretty much the same code in two. Instead by having one Controller class and checking what mode is used, makes for much cleaner code and a more unified experience between the two modes.

3.3.1 Grabbing and Translating

One of the most basic but also important interactions a user can have with an object is grabbing and moving it around in the world. This is triggered when the user presses the grab button and stays as long as that button is held. Depending on the mode, this button is either the left mouse button or a trigger on a VR controller. When the button is pressed an RPC is called onto the server, where all of the interactions happen. This RPC creates traces a line in the direction that the user is pointing at and checks if any objects get hit by this line. The pointing direction for desktop is where the user is currently looking at, marked for the user as a circular crosshair in the UI. For VR it's simply where the controller, whose trigger was pressed, is pointing at. What this looks like in Blueprints is shown in Figure 3.5.

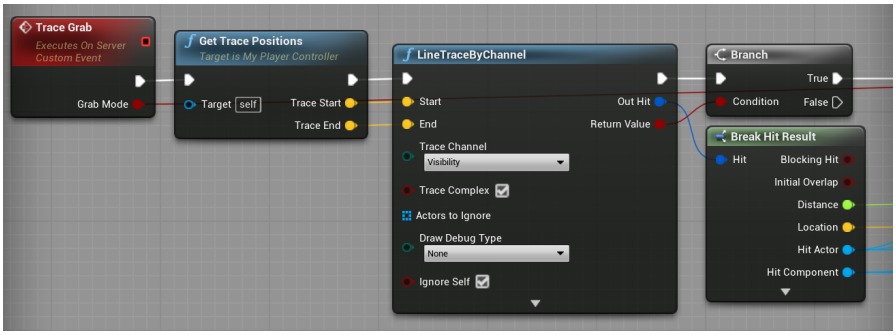

Figure 3.5: Tracing Grabs in Blueprints

In case there was a hit, a result is created which contains which Actor and Component were hit and where. It is then checked that the correct class of Actor was hit and that the Actor isn't already being grabbed. If these checks are passed, a flag on the Actor is set to let other users knows that it is currently being grabbed. While this is the case, the Actor will be moved to where the user is pointing at the distance that the Actor

had when it was grabbed. This location is additionally slightly offset by the position where it was grabbed. If this isn't wasn't done, once the Actor got grabbed it would instantly move so that the middle of it was where the user is pointing, causing a rather weird looking effect. Instead the location where the hit occurred is taken, transformed to the local coordinate system of the Actor and used as an offset when moving. This way it looks like the user is grabbing the Actor by the part where they intended to. While grabbing, the Actor they can also increase and decrease the distance to the Actor as well as rotate it around its z-axis. This is either done through the mouse wheel and two keys on the keyboard or through the stick on the VR controller. As this all is happening on the server and the Actors movement is replicated, it will also be moved in every instance. Once the grab button is released, another RPC is called that changes the flag on the Actor and ends the grab. Overall this creates a very intuitive way of grabbing.

3.3.2 Scaling and Rotating

The next ways of interacting after translation are logically scaling and rotation. For these, and the rest of the interactions, a small menu was created for the user-interface which makes these functionalities available. This interaction menu can be seen in Figure 3.6. It appears when a user presses the right mouse button or secondary trigger on a VR controller while pointing at an Actor. On a desktop window the menu is located on the right side and is used with a cursor. This cursor captures the mouse movement while the menu is open so that the users view doesn't move along with it. In VR the menu appears above the other VR controller and is used with the initial controller.
The scaling and rotation are done through sliders and the input fields next to them. After one of the values is changed, an RPC gets called that tells the server to rotate or scale the selected Actor the desired amount. This then automatically gets replicated to the other clients.

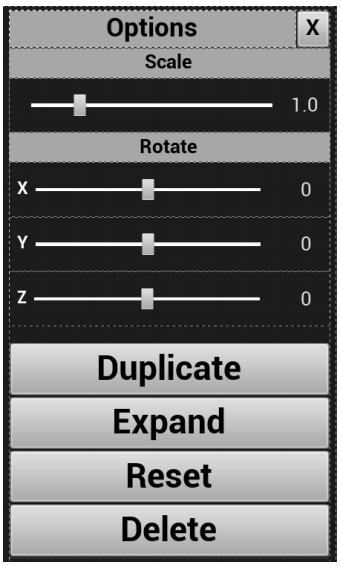

Figure 3.6: Interaction Menu

3.3.3 Duplication, Expansion, Resetting and Deletion

Duplication, as the name says, creates a duplicate of the selected Actor. There are a lot of cases when multiple instances of an Actor are required and instead of having to open the same file for each of those times, it can be done with the click of a button. This uses the fact that the original Actor contains the necessary data so no further file sharing is needed to create a new one. The new object doesn't contain the data, in order to save on memory. Instead it has a reference to the original so that in case it also gets

duplicate, it can point to the necessary data.
Expansion aims to give the user a better look at the individual components and how they make up the whole Actor. This is done by translating the relative positions of the components in the direction of their centre. By doing this, the components remain in the relative areas they are supposed to be while creating enough space to differentiate between them.

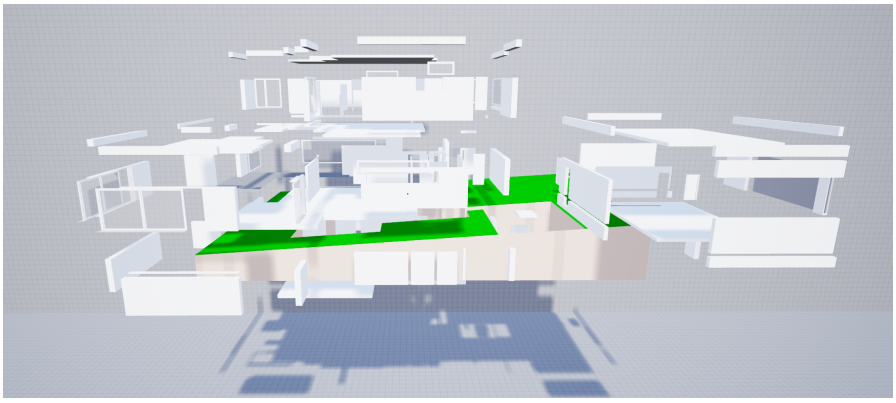

Figure 3.7: Example for an expanded Actor

Resetting is done to undo any sort of changes done to the Actor and put in back into the state it was when it was generated. This means the scaling for every component is set to 1, rotation set to (0,0,0) and relative location to the centre location saved in the components themself.
If objects can be added during runtime, they should also the option to remove them. Compared to creating, deleting is a lot simpler as Unreal already has functions for that. When deleting an Actor it is also checked if that is the original instance of it and if there are copies of it in the world. If that is the case, the variables are transferred to a copy and the references for all the copies are updated.

3.3.4 Interacting with Individual Components

While being able to interact with it the Actor as a whole is already useful, a lot more could be achieved if users could interact with every component individually. For that a separate mode can be toggled with the press of a button. This mode uses the fact that the hit of a line trace also returns what component was hit to determine the component to interact with. But for the user it might now always be clear which component is which. So while the user is in this mode the component currently being pointed is highlighted. This effect is achieved through changing the material on the component with a dynamic material instance and changing it back to it's usual material once it isn't being targeted any more.The result of this can be seen in Figure 3.8.

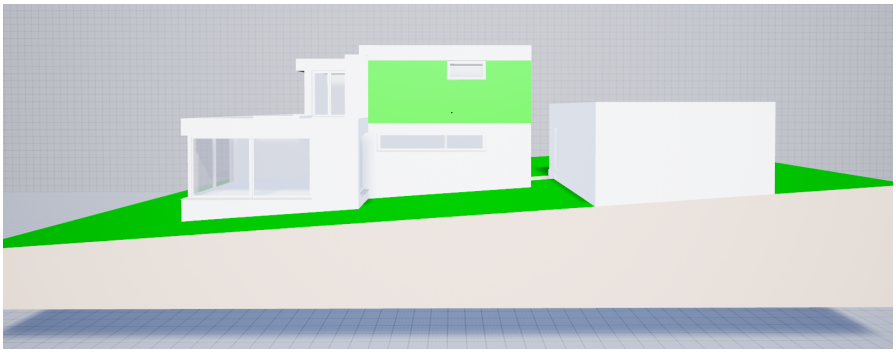

Figure 3.8: Highlighting a Component in CRP

With this component the user can interact in almost all of the same ways they could with an Actor. Only expansion and duplication are not supported. The first one due to it requiring multiple components and the other one in order to avoid having two identical components attached to the same Actor.
When it comes to translating the component a special feature is implemented. Even though the component can be moved out of it's initial space, it is still part of the Actor so if the Actor were to move, so would the component. Unreal does offer the ability to turn this off through a function called DetachFromComponent. While it may seem that the component now would be independent of the Actor, this is not the case. It purely means that the Component isn't moved along the Actor. Meaning if the Actor were to be delete, the Component would be gone as well. Truly detaching and reattaching a Component is very finicky and not recommended. Instead, the developed mesh generating functionality can be used to solve this problem. The component is simply replaced by a new copy of it that is attached to a different Actor. For this Actor a specific class was made that is designed to hold exactly one Component. This class can call upon the Generate Mesh Function, as that is a Blueprint Function Library, and generate the specific component needed by using the index saved in the original. This is done once the component is released from being grabbed. The original component is moved away, turned invisible and has its collisions turned off. From the users perspective it doesn't actually look like anything happened but they now have an instance of that component that can exist completely separately from the original Actor. The user can interact with this newly generated component in all the ways they usually could. Important to note is that when the reset function is used here, the component actually deletes itself and return the original to the way and position it is supposed to be. For the user this just looks like the component was placed back where it was.
Overall this suite of interactions demonstrates the main ways of interaction and some additional ones are possible with the newly generated objects. There are definitely countless many more that could be implemented but for the purposes of this prototype, this is quite sufficient.

4 Results and Evaluation

In diesem Kapitel sollen die Ergebnisse dieser Diplomarbeit diskutiert werden.

4.1 CAD Runtime Importer and Presenter

As mentioned

4.2 Comparison to Related Works

4.3 Shortcomings and Possible Improvements

5 Conclusion

In diesem Kapitel sollen zunächst die erreichten Ziele diskutiert und abschließend ein Ausblick auf mögliche, weiterführende Arbeiten gegeben werden.

Bibliography

[CEI01] Carl Erikson D. M., III W. V. B.: Hlods for faster display of large static and dynamic environments. *University of North Carolina at Chapel Hill* (2001).

[Cha08] Charpentier F.: *Nvidia Cuda: Das Ende der CPU?* Technical report, Tom's Hardware, Jun 2008. http://www.tomshardware.com/de/CUDA-Nvidia-CPU-GPU,testberichte-240065.html (20.08.2009).

[Cla76] Clark J. H.: Hierarchical geometric models for visible surface algorithms. *Communications of the ACM 19*, 10 (Oct 1976), 547–554. http://design.osu.edu/carlson/history/PDFs/clark-vis-surface.pdf (09.09.2009).

[DFMP98] De Floriani L., Magillo P., Puppo E.: Efficient implementation of multi-triangulations. In *VIS '98: Proceedings of the conference on Visualization '98* (Los Alamitos, CA, USA, 1998), IEEE Computer Society Press, pp. 43–50.

[Eck99] Eckert M.: *Von-Neuman Architektur.* Technical report, TecChannel, sep 1999. http://www.tecchannel.de/server/prozessoren/401364/so_funktioniert_ein_prozessor (19.08.2009).

[Eis06] Eiserle M.: Progressive techniken in der computergrafik. *Universität Rostock* (2006). http://vcg.informatik.uni-rostock.de/assets/publications/theses_sem/SA_Eiserle2006.pdf (15.09.2009).

[ESV99] El-Sana J., Varshney A.: Generalized view-dependent simplification. *Computer Graphics Forum 18*, 3 (1999), 83–94.

[Fos95] Foster I.: *Designing and Building Parallel Programs.* Addison Wesley Pub Co Inc, Reading, MA, USA, 1995.

[Har08] Harris M.: *Parallel Prefix Sum (Scan) with CUDA.* Programming guide, NVIDIA, 2008.

[Hop96] Hoppe H.: Progressive meshes. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1996), ACM, pp. 99–108.

[Hop97] Hoppe H.: View-dependent refinement of progressive meshes. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer*

graphics and interactive techniques (New York, NY, USA, 1997), ACM Press/Addison-Wesley Publishing Co., pp. 189–198.

[Hop98] Hoppe H.: Efficient implementation of progressive meshes. *Computers & Graphics 22*, 1 (1998), 27–36.

[HSH09] Hu L., Sander P. V., Hoppe H.: Parallel view-dependent refinement of progressive meshes. In *I3D '09: Proceedings of the 2009 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2009), ACM, pp. 169–176.

[JNS08] John Nickölls Ian Buck M. G., Skadron K.: *Scalable Parallel Programming*. Programming guide, UNIVERSITY OF VIRGINIA, 2008.

[Lit08] Littschwager T.: *Neue Grafik-Generation: Alle Details*. Technical report, Chip, Sep 2008. http://www.chip.de/artikel/Neue-Grafik-Generation-Alle-Details_32708718.html (19.08.2009).

[Lue01] Luebke D. P.: A developer's survey of polygonal simplification algorithms. *IEEE Computer Graphics and Applications 21*, 3 (May/Jun 2001), 24–35. http://www.cs.virginia.edu/~luebke/publications/pdf/cg+a.2001.pdf (11.09.2009).

[MB00] Mrohs Bernd C. R.: Progressive meshes - eine einführung. *test* (Jul 2000). http://www.mrohs.com/publications/Bernd%20Mrohs,%20Christian%20Raeck%20-%20Progressive%20Meshes.pdf(29.07.09).

[MGK03] Michael Guthe P. B., Klein R.: Efficient view-dependent out-of-core visualization. *University of Bonn, Institute of Computer Science II* (2003). http://www.uni-marburg.de/fb12/informatik/homepages/guthe/files/guthe-2003-efficient (12.09.2009).

[Nah02] Nahmias J.-D.: Real-time massive model rendering. *University College London* (Sep 2002). http://www.cs.ucl.ac.uk/research/equator/papers/Documents2002/Jean-Daniel_Nahmias/Massive_Model_Rendering.htm (29.07.09).

[NVI07] NVIDIA: *NVIDIA CUDA Compute Unified Device Architecture*. Programming Guide Version 1.0, NVIDIA Corporation, Santa Clara, CA, USA, 2007.

[NVI08] NVIDIA: *NVIDIA GeForce GTX 295*, 2008. http://www.nvidia.de/object/product_geforce_gtx_295_de.html (02.10.2009).

[NVI09] NVIDIA: *OpenCL Programming Guide for the CUDA Architecture*. Programming Guide Version 2.3, NVIDIA Corporation, Santa Clara, CA, USA, 2009.

[OLG*05] Owens J. D., Luebke D., Govindaraju N., Harris M., Krüger J., Lefohn A. E., Purcell T. J.: A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports* (Aug 2005), pp. 21–51.

[Paj01] Pajarola R.: Fastmesh: Efficient view-dependent meshing. *Computer Graphics and Applications, Pacific Conference on 0* (2001), 0022.

[PD04] Pajarola R., DeCoro C.: Efficient implementation of real-time view-dependent multiresolution meshing. *IEEE Transactions on Visualization and Computer Graphics 10*, 3 (2004), 353–368.

[Rißß99] Risska V.: *Test: Intel Core i7 920, 940 und 965 Extreme Edition.* Technical report, Computerbase, Sep 1999. http://www.computerbase.de/artikel/hardware/prozessoren/2008/test_intel_core_i7_920_940_965_extreme_edition/ (20.08.2009).

[Tro01] Troger C.: Levels of detail. *Institute of Computer Graphics and Algorithms Vienna University of Technology* (2001). http://www.cg.tuwien.ac.at/courses/Seminar/SS2001/lod/troger_paper.pdf (09.09.2009).

[WIK] WIKIPEDIA: *Octree.* http://de.wikipedia.org/wiki/Octree (29.07.2009).

List of Abbreviations

ALU Arithmetic Logic Unit
BTF Bidirektionalen Textur Funktion
CPU Central Processing Unit
CU Control Unit
CUDA Compute Unified Device Architecture
FLOPs Floating Point Operations Per Second
FPU Floating Point Unit
GPGPU General Purpose Compution on Graphics Processing Unit
GPU Graphics Processing Unit
HLOD Hierarchische Level of Detail
IFS Indexed-Face-Set
LOD Level of Detail
MIMD Multiple Instruction Multiple Data
OpenCL Open Computing Language
OpenGL Open Graphics Library
PCAM Partitionierung Kommunikation Agglomeration Mapping
PM Progressive Meshes
SFU Spezial Funktion Units
SIMD Single Instruction Multiple Data
SIMT Single Instruction Multiple Threads
SLI Scalable Link Interface
SP Streaming-Prozessoren
SM Streaming-Multiprozessoren
TPC Textur Prozessor Clustern
VBO Vertex Buffer Object

List of Figures

2.1 Unreal Engine Level Editor 6
2.2 Template for third person character 8
2.3 Example Unreal C++ Class Header 9
2.4 Example Blueprint Code 10
2.5 Difference between Listen and Dedicated Server 12

3.1 CAD Runtime Presenter File Picker 15
3.2 OBJ Polygon to Triangle Fan 17
3.3 STL Triangle facet 18
3.4 Loaded CAD Model in Unreal Engine 23
3.5 Tracing Grabs in Blueprints 24
3.6 Interaction Menu instead of UI Editor 25
3.7 Example for an expanded Actor 26
3.8 Highlighting a Component in CRP 27

List of Tables

3.1 ObjTypes 17
3.2 MTL Types 18

List of Algorithms

1 Pseudocode for generating a Mesh Component 22

Listings