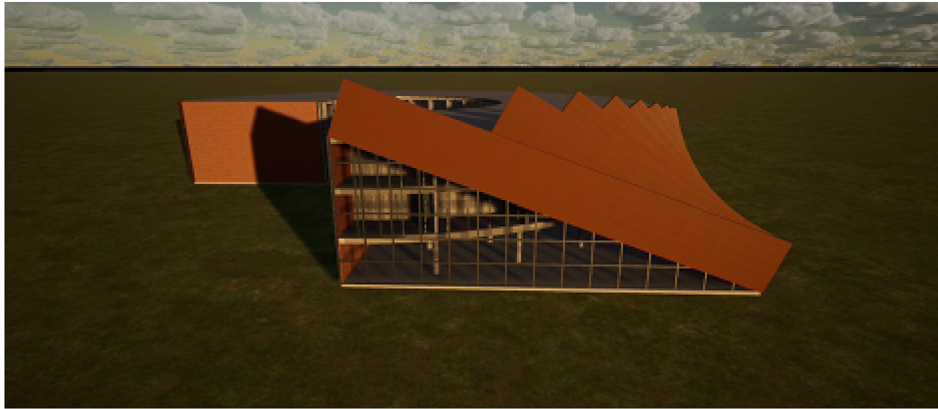


Game Engines for Algorithmic Design

Ricardo de Lemos Filipe

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. António Paulo Teles de Menezes Correia Leitão

Examination Committee

Chairperson: Prof. Maria Luísa Torres Ribeiro Marques da Silva Coheur
Supervisor: Prof. António Paulo Teles de Menezes Correia Leitão
Member of the Committee: Prof. João António Madeiras Pereira

January 2021

Acknowledgments

First of all, I would like to deeply thank my family and friends, specially my parents for their support, care and encouragement over the years.

During these past five years at IST, I met incredible people and I am lucky to call some of them friends. A special thank you to Nuno, Lucas, Miguel, Guilherme e Francisco. I would like also to thank the members of the Algorithmic Design for Architecture Group (Guilherme Santos, Inês Pereira, Renata Castel Branco, Inês Caetano, Helena Martinho, Gonçalo Araújo, Pedro Alfaiate, Pedro Ramos, Rui Ventura, Sara Garcia), for all the feedback and cooperation in the writing of articles and in the realization of presentations.

Last but not least, I would like to express my deepest gratitude and appreciation to my supervisor, Professor António Leitão, for having accepted to guide my dissertation and for all the monitoring provided during this long process.

Abstract

With the advancements in technology and computers, new tools and techniques were developed in architecture. Architects started using digital modeling tools, like Computer-aided Design (CAD) and Building Information Modeling (BIM) applications. By using these tools, architects can design three dimensional models. A new approach was also developed, an algorithmic approach. In an algorithmic approach, the architect writes an algorithm that generates the digital model. Visualization tools are important in an algorithmic approach, because they help the architect write the algorithm and allow architects to give a subjective evaluation of the design aesthetic. Typical visualization tools, such as CAD and BIM applications, can only provide a low fidelity dynamic view. These applications can generate high-fidelity renders, but they require a large amount of time to render. This wait time hinders the architect's productivity and thought chain. Additionally, these applications have performance issues, when they are saturated with the geometry fed by an algorithmic description. Game engines, contrary to CAD and BIM applications, can adapt the digital model to be visualized in real time and they also provide navigation systems. These qualities make game engines excellent visualization tools. For this reason, we explore the use of game engines that can generate high-fidelity renders in real time as visualization tools. This solution can generate the digital model and adapt the model for real time rendering with high fidelity. We evaluate the image quality and performance of our solution by comparing it with another visualizer.

Keywords

Algorithmic Design; Game Engine; Interactive Visualization; High-fidelity render.

Resumo

O avanço da tecnologia e desenvolvimento dos computadores permitiu a criação de novas técnicas e ferramentas na arquitetura. Os arquitetos passaram a ter acesso a ferramentas de modelação digital, como ferramentas Computer-aided Design (CAD) e Building Information Modeling (BIM). Usando estas ferramentas, os arquitetos conseguem desenhar modelos em três dimensões. Com o avanço da tecnologia surgiu uma abordagem algorítmica. Na abordagem algorítmica, o modelo digital é gerado por um algoritmo. Nesta abordagem, ferramentas de visualização são importantes, pois ajudam na escrita do algoritmo e permitem o arquiteto avaliar o design do seu modelo. As ferramentas tipicamente usadas em arquitetura, como ferramentas CAD e BIM, são bastante limitadas, apenas permitem visualizar em tempo real com baixa fidelidade. Estas aplicações são capazes de gerar imagens com fidelidade, mas requer uma elevada quantidade de tempo, diminuindo a produtividade do arquiteto. Adicionalmente, estas aplicações mostram mau desempenho num contexto de design algorítmico onde é gerado um elevado número de geometria. Motores de jogos, contrariamente às aplicações CAD e BIM, adaptam o modelo digital para ser visualizado em tempo real e fornecem sistemas de navegação. Esta qualidade torna-os uma excelente ferramenta de visualização. Devido a isto, nós exploramos o uso de um motor de jogo capaz de gerar imagens com alta fidelidade como ferramenta de visualização para design algorítmico. A nossa solução é capaz de gerar o modelo e adaptá-lo para visualização em tempo real com alta fidelidade. Avaliámos a qualidade das imagens geradas pela nossa solução e o seu desempenho comparativamente com outro visualizador.

Palavras Chave

Motor de Jogo; Visualização Interactiva; Design Algorítmico; Imagem de Alta Fidelidade.

Contents

1 Introduction **1**

1.1 Algorithmic Design . . . 3

1.2 Problem . . . 4

1.2.1 Goals . . . 4

1.2.2 Photo-realistic renders . . . 5

1.2.3 Interactivity . . . 6

2 Related Work **7**

2.1 Luna Moth . . . 9

2.2 OpenSCAD . . . 10

2.3 ArchiCAD . . . 11

2.4 Revit . . . 11

2.5 Rhinoceros 3D . . . 12

2.6 SketchUp . . . 12

2.7 Unreal Engine . . . 13

2.7.1 Photo-realism in UE . . . 14

2.7.1.A Physical-Based Shading . . . 14

2.7.1.B Illumination . . . 16

2.7.1.C Reflections . . . 16

2.7.1.D Shadows . . . 16

2.7.1.E Ray tracing . . . 17

2.7.2 Generating Geometry in UE . . . 17

2.7.3 Additional Architecture Visualization Features . . . 18

2.8 Unity . . . 18

2.9 Comparison . . . 20

3 Methodology and Implementation **23**

3.1 Architecture Design Requirements . . . 25

3.1.1 Optimizations . . . 27

3.2 Model creation 29
3.2.1 Brushes 29
3.2.2 Primitive Method 31
3.2.3 Optimizations 33
3.3 Render Process 34
3.3.1 Textures 34
3.3.2 Rendering 37
3.4 Navigation 38

4 Evaluation **39**
4.1 Image Fidelity 41
4.1.1 Materials 42
4.1.2 Lights 44
4.2 Performance 46
4.2.1 Generate Model 46
4.2.2 Generate Cinematics 47
4.2.3 Interactivity 47
4.3 Summary 48

5 Conclusion **51**
5.1 Future Work 54

List of Figures

1.1 In the image we can see the three photo-realism components: lights, shadows, and materials. The walls and the roof are illuminated by global illumination while the green ball is illuminated by local illumination and casts a shadow. The roof has a green tone because the sphere has some reflective material and is reflecting its color onto the roof. 5

2.1 Image of a model generated by Luna Moth 10
2.2 Image taken from Realspace, using ray tracing 14
2.3 Image taken from ”Rebirth”, a cinematic rendered in real-time, powered by Unreal Engine and produced by Quixel. 15
2.4 Sequencer Editor gives users the ability to create in-game cinematics with a specialized multi-track editor. 18
2.5 Render produced in Unity 19

3.1 Component-and-Connector view of our solution 26
3.2 Unified Modeling Language sequence, showing how the communication between Unreal Engine (UE) and Khepri works. 28
3.3 Creating a slab with brushes. 29
3.4 Slab created with brushes. 29
3.5 Slab created by brushes with a square tile texture. 30
3.6 The projection of a texture into a cube. 31
3.7 Slab created by the brushes method (above) and the primitive method (below). Note, in the top image, that the texture is divide into different areas. This problem does not occur the bottom image. 32
3.8 In the left image, we can see a cylinder created with brushes between tiles. In the right image, we can view a cylinder generated with the primitive method. 33
3.9 A geometric texture applied to a big surface. In the left image, we can see the texture. In the right image, we can view the texture applied to a huge surface. 35

3.10 Numeric texture using tile-blending. In the left image, we can see the original texture. In the right image, we can view the texture applied to a larger surface using tile-blending. . . 36
3.11 In the left image, we can see the texture without blending between tiles. In the right image, we can view the texture using blending. 36
3.12 Blending process colored with low and high blending respectively. 37
3.13 Stochastic texture synthesized. 37

4.1 Shader parameters in Unity and UE 42
4.2 Creating a virtual texture as input for a brick material shader. 42
4.3 Render showing material quality in Octane, Unity and UE. 43
4.4 Material using the same textures rendered in Unity and UE 44
4.5 Render in V-Ray, Unity and UE showcasing reflection capabilities in each visualizer. . . . 44
4.6 Reflection in UE when we cover the screen with an orange block. 45
4.7 Scene showcasing Unreal Engine light reflection and diffraction limitations. On the right, we have the scene rendered in Unreal Engine and on the left we have the scene rendered in V-Ray. 45
4.8 Scene showcasing UE global illumination limitations. On the right, we have the scene rendered in UE and on the left we have the scene rendered in V-Ray 46

List of Tables

2.1 Comparison between Algorithmic Design (AD) visualization back-ends, based on features used by architects. 20

3.1 Time it takes to create objects in UE with and without concurrency, and with optimized concurrency. The results are shown in seconds and all of the objects created are exactly the same. 28
3.2 Triangulation algorithms in the library polypartion and their complexity without and with holes where n is the number of vertices and h is the number of holes. 31
3.3 The time it takes on average to create a cube with the brush method and primitive method. 33
3.4 The time, in seconds, it takes to generate the Isenberg Business Innovation Hub's model when using different methods. 34

4.1 Time Unity and UE take to generate the digital model of Isenberg Innovation Hub, Astana National Library, and a model that can be increased exponentially 46
4.2 Time it take to generate cinematics in Unity and UE while using Khepri method and our method 47
4.3 The average number of frames per second that UE and Unity have when using game view and editor view in different viewpoints 48

Acronyms

AD Algorithmic Design

BIM Building Information Modeling

BRDF Bidirectional Reflectance Distribution Function

BSP Binary Space Partitioning

CAD Computer-aided Design

CSG Constructive Solid Geometry

NURBS Non-Uniform Rational Basis-Spline

PBR Physical-based Rendering

PBS Physical-based Shading

RPC Remote Procedural Call

TCP Transmission Control Protocol

UE Unreal Engine

VIM Virtual Information Modelling

VR Virtual Reality

1

Introduction

Contents

1.1 Algorithmic Design 3
1.2 Problem 4

Architectural designs and design processes have been influenced by the digital era. Computer-aided Design (CAD) and Building Information Modeling (BIM) applications are digital tools used by architects to create building designs, increasing productivity, the quality of the presention images, and the production of technical documentation. This evolution to the digital medium has allowed architects to develop more complex designs.

In the digital design process, an architect uses a set of digital tools that are capable of 3D modeling, analysis, rendering, 2D drawing, optimization, etc. CAD and BIM applications are the most commonly used for 3D modeling [1] and rendering, as they provide a digital way to model a 2D or 3D view of a building. Render engines, like V-Ray, are also used to create renders in architecture. Additionally, the BIM paradigm goes even further to complement the digital model with various relevant metadata, such as material costs and quantities, to support other related activities such as construction and fabrication [2]. As for the analysis tools, they are typically used to perform simulations in order to infer a building's performance according to structural, thermal, lighting, cost, or other requirements. Different analysis tools must be used to study these different criteria, such as Radiance for lighting evaluation, EnergyPlus for thermal evaluation, and Robot for structural evaluation. The usage of these multiple tools to construct an architectural project might impose an inefficient, repetitive, and tiresome workflow. Furthermore, as the project grows, changes become costlier. This happens not only because of the manual work required to remodel the building's design, but also because of the propagation of these changes to the respective different analysis models tied to the building design.

1.1 Algorithmic Design

The need to use various tools and making multiple changes can be problematic when creating complex architectural designs. Algorithmic Design (AD) came to mitigate this problem. AD is a design approach based on the creation of models through algorithms [3]. Unlike traditional architecture design approaches, with AD, the architect does not create the digital building model directly. Instead, the architect writes the program that generates the digital model through a combination of geometric, mathematical, and symbolic representations [4]. This allows the architect to be able to create more complex geometry, automate repetitive tasks, and explore new alternative designs with low effort. Easily achieving these design alternatives is possible because the entities in the project are logically connected and changes are easily propagated [5].

The algorithmic approach, if applied correctly, is capable of describing the same model to different tools, by adapting the algorithmic description to the modeling operations of each tool [6].This approach follows a workflow. The workflow starts with coupling an AD tool with CAD tools where the architect can visualize the design. After the design concept, the model is sent to the analysis tools for performance

evaluation. Finally, after the evaluation process finishes, having potentially improved performance, the design is further developed either in CAD or in BIM.

1.2 Problem

In an AD approach, the architect does not model a design directly in CAD and BIM applications, but instead creates a parametric program that generates a design model in the intended applications (e.g., for visualization or analysis) [4]. However, creating such a program is not a trivial task. Coding complex designs demands an additional effort for the architect, who might not be very proficient at programming. This leads not only to additional errors, such as coding mistakes, along with design mistakes, but also to a disconnection between what is being written and what effectively is going to be generated as a result. The latter aspect is particularly important because of how crucial visualization is for architecture. Only by visualizing their designs can architects give a subjective evaluation of their designs' aesthetics.

Unfortunately, currently used visualization tools, such as CAD (e.g., AutoCAD and Rhinoceros) and BIM (e.g., Revit and ArchiCAD) applications, have performance issues as a project grows in scale. This is particularly severe in the case of AD , because it enables the quick generation of large amounts of geometry without much effort. Moreover, AD allows us to reconstruct whole designs by simply changing its parameterization, leading to further deceleration in the design workflow. This will greatly affect the design production process since, as a project starts to grow, each change will take longer to verify and design errors might proliferate. On later stages of the design process, high-quality renders needs to be generated for design presentations to clients. In our experiments, this stage may take days, sometimes even weeks, to accomplish, even on a specialized rendering workstation [7].

1.2.1 Goals

Our main goal with this thesis is to overcome the problems caused by the use of AD alongside a CAD or BIM application, as the model generated by these applications, most of the time, is not suitable for navigation or visualization because they provide a restrictive real-time visualization and perform badly with complex models [8]. These applications were designed for interactive use and often become a liability with regards to performance with the considerable amount of data generated by the AD approach. These applications prove to be unacceptable to an AD workflow because their performance problems delay the visualization of the generated designs, thus making AD harder than necessary. To this end, we will explore the use of a game engine as a real-time visualization tool for AD, something that has already been explored successfully in the past [7].

The reason why game engines are a good solution for real-time visualization [9] is that they employ different techniques to simulate reality while having real-time rendering in mind. One example of these

techniques is mipmap. This technique uses a sequence of textures with progressively less resolution. The selection of the texture to apply is based on the distance between the camera and the object, providing better performance while maintaining the illusion of realism.

Due to advances in technology, game engines have become more complex and sophisticated. New techniques permit a better simulation of reality, allowing games to achieve more photo-realistic results, while only using relatively weak graphics environments with textured maps and artistry [10, 11]. This capability of creating close to photo-realistic results in real time allows not only the quick generation of renders, but also the capability of real-time navigation, as well as direct interaction with the building elements, like opening a door.

Our goal in this dissertation is to use a game engine to create high-fidelity renders rapidly and also serve as navigation tool.

1.2.2 Photo-realistic renders

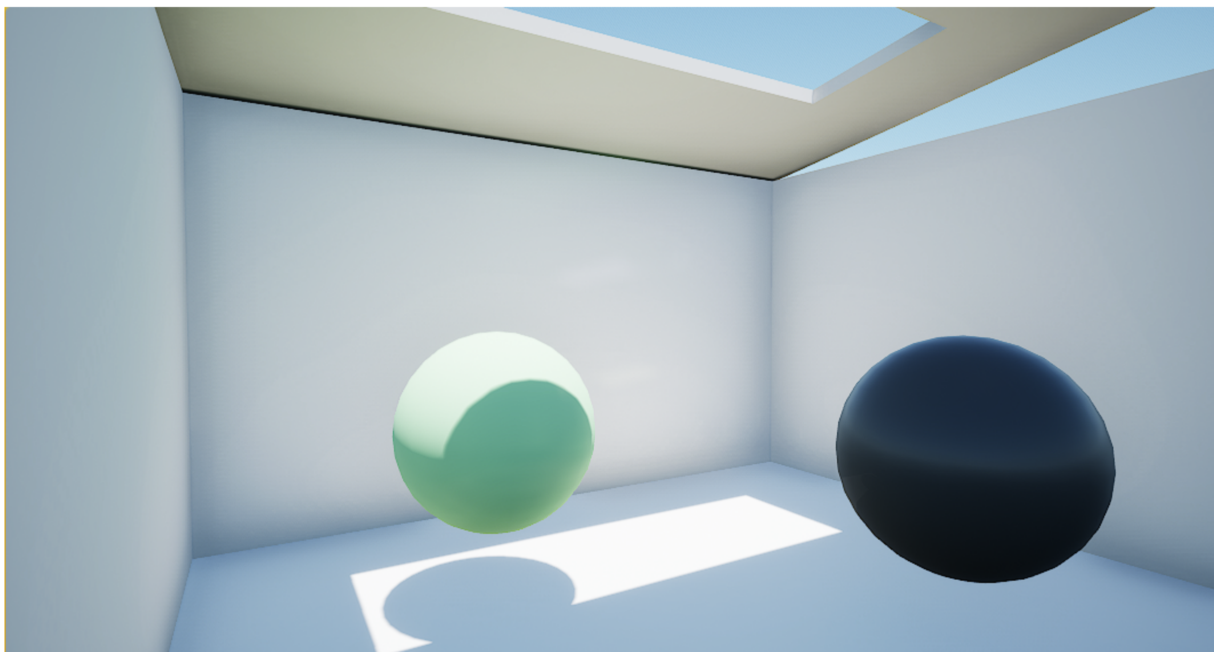


Figure 1.1: In the image we can see the three photo-realism components: lights, shadows, and materials. The walls and the roof are illuminated by global illumination while the green ball is illuminated by local illumination and casts a shadow. The roof has a green tone because the sphere has some reflective material and is reflecting its color onto the roof.

One of our objectives is to provide a high-fidelity visualizer. In other words, a close to photo-realistic visualizer. However, defining a photo-realistic image is hard because it requires a person to define what makes something look real. In computer graphics [11, 12], this can be defined as mimicking real life physics. To that end, four components are needed, as seen in figure 1.1, where the quality of these

components define how real an image looks like:

Lights: makes a two dimensional model look three dimensional, through shading;

Shadows: provide visual cues to the viewer about object placement;

Interactions: simulates interactions between objects and with the environment.

Materials: composed of textures that define the model's appearance and how the model interacts with light.

In computer graphics, to simulate real life light physics, light is divided into two parts: local illumination, which is responsible for calculating light interaction coming directly from a source; and global illumination, simulating the light coming from reflection, diffusion, or refraction.

1.2.3 Interactivity

Navigating in a 3D model requires a well-defined navigation system. We proposed three different navigation systems: free camera, walk mode, and virtual reality. In free camera, the user will be able to fly around the scene and pass through any object. In walk mode, the user will be able to walk around in the scene as a person and possibly collide with other objects. Virtual Reality (VR) is like walk mode but with VR equipment.

For the purpose of developing a real-time high-fidelity visualization tool that satisfies the requirements imposed by the AD approach, in the next section, we will study existing visualizers aimed at architectural design.

2

Related Work

Contents

2.1 Luna Moth 9
2.2 OpenSCAD 10
2.3 ArchiCAD 11
2.4 Revit 11
2.5 Rhinoceros 3D 12
2.6 SketchUp 12
2.7 Unreal Engine 13
2.8 Unity 18
2.9 Comparison 20

In this section, we will explore existing visualization solutions and describe advantages and disadvantages of each solution, analyse the impact game engines have had on architecture visualization, and analyse the techniques used by Unreal Engine (UE) to generate geometry and achieve high fidelity rendering.

Before comparing visualization tools, it is important to clearly define the characteristics we are analysing. As mentioned in the objectives, we want to achieve high-fidelity visualization, interactivity, and real-time rendering, and, as such, we will be focusing more on the tool's capabilities in these areas. All the tools mentioned below are capable of receiving a set of modeling operations describing a model and generate it.

2.1 Luna Moth

Luna Moth is a web application designed for AD [13]. The main objective of this application is to increase productivity during early stages of the architectural design process. Luna Moth removes the need to install and update software since it can be used inside a web browser. Furthermore, the user does not need to transfer projects between computers since the information can be saved remotely. Another key feature that Luna Moth has is interactivity, since an architect can change a parameter and immediately receive feedback, which helps showing the relation between the program and the model. Luna Moth is also capable of traceability, since selecting parts of the model makes the application show which part of the code created those parts and vice versa.

Luna Moth is a useful tool during early stages of the architectural design process due to its ability to provide fast feedback and create an environment where an architect can rapidly test different variations. Luna Moth currently uses Three.js[1], a JavaScript library that uses WebGL. Three.js supports local illumination, global illumination, shadows, and realistic materials. However, Luna Moth only uses Three.js to do local illumination, using the Phong shading model, and only uses a simple matte material. This decision significantly reduces the rendering quality, as can be seen in figure 2.1.

In terms of navigation, Luna Moth only supports free camera movement. Moreover, adding new navigation systems that require collision detection is not trivial due to the fact that Three.js does not support collision detection. Performance is also an issue: even though Luna Moth is more responsive than other native desktop applications such as AutoCAD [13], these can render frames faster than web applications [14]. Due to interactivity being so related to performance in real-time rendering, as low frame rates ruin the user experience [15], we can conclude that Luna Moth is not a sufficiently good visualization tool for more complex models.

Because Luna Moth is designed to be a tool for the design process, it also allows the exportation

[1] Three.js Documentation, `https://threejs.org/docs/`. Last accessed 23 Dec 2019

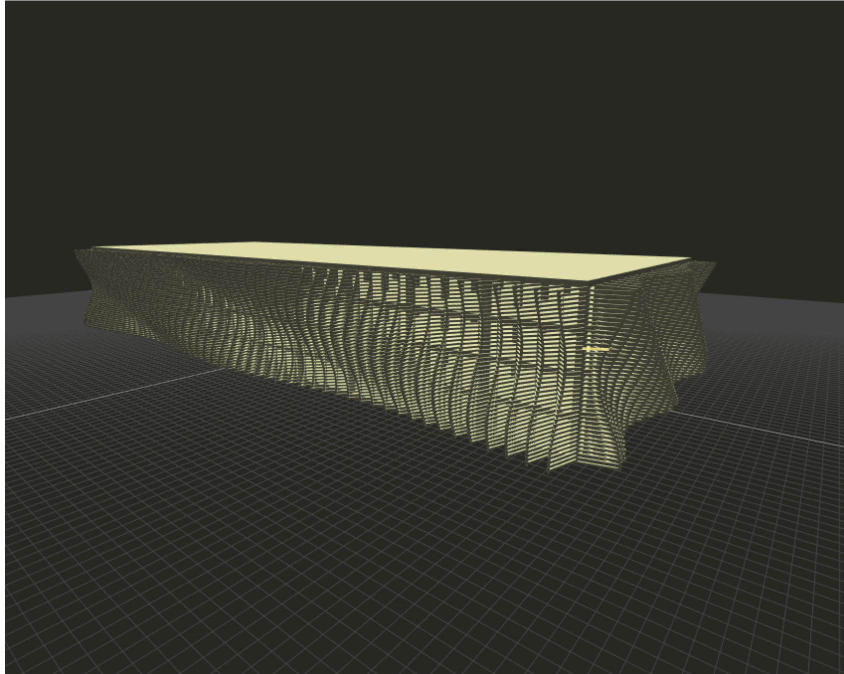


Figure 2.1: Image of a model generated by Luna Moth

of the model into a CAD application, like AutoCAD. The exportation is done by sending requests to the target CAD application and creating the corresponding shapes. This allows the user to regenerate the model in the CAD tool with better visualization and rendering capabilities.

2.2 OpenSCAD

OpenSCAD is an AD application with the goal of reducing the architect's waiting time for visual feedback of changes and providing a visual way to help them in the programming task. Similarly to Luna Moth, OpenSCAD uses scripts which specify geometric primitives, such as cubes, cylinders, and spheres, and defines how they are modified and combined through Constructive Solid Geometry (CSG). OpenSCAD allows the creation of 3D models of parametric designs that can be easily adjusted by changing the parameters. In OpenSCAD, a user can highlight an object and see the part of the program that generated it, which can help the user understand what object is being changed.

OpenSCAD provides a view using Phong shading model, which does not provide a high-quality view but allows to quickly generate and render the model. Unlike Luna Moth, OpenSCAD supports simple materials.

2.3 ArchiCAD

ArchiCAD is a BIM tool developed by Graphisoft. What makes it stand out from other CAD applications is the fact that it uses data-enhanced parametric objects. ArchiCAD can be used to produce 2D models, 3D models, technical documentation, and renders. Because ArchiCAD is a BIM application, it is capable of storing data and structural information within the model, allowing to run analyses on the modeled building.

Similarly to other BIM applications, ArchiCAD is capable of rendering visually appealing rendered results, but it has its limitations. BIM applications' use case is aimed at interactive usage, where they can be sufficiently performant. However, when used in the context of AD, they suffer from slowdowns as a project becomes saturated with the geometry fed by an algorithmic description. ArchiCAD has two main views: one with a simplified view of the model of the design, with simplified materials, shadows, and lighting; and another view for the generation of high-quality static renders. If the architect wants to see that view in high-quality, he must wait for the rendered result. This wait time hinders the architect's productivity and thought chain.

ArchiCAD does not natively support real-time rendering with high fidelity but this becomes possible with Twinmotion and Lumion. Twinmotion is an UE real-time visualizer capable of doing real-time rendering with high fidelity, and supports the three navigation systems that are described in section 1.2.3, as well as synchronisation with CAD and BIM applications. However, Twinmotion does not allow any interaction with objects and has limited control over quality level, like the level of detail of materials. Lumion is a real-time visualizer capable of generating high-fidelity renders for CAD and BIM applications. It includes an interactive interface, a weather system, and a resourceful library of Physical-based Rendering (PBR) materials and assets. Lumion does not support VR navigation or walk mode, it only supports free camera navigation. Both Twinmotion and Lumion have import mechanisms to link with CAD and BIM applications. Since these visualizers are mainly focused for high-fidelity real-time renders, they might not scale well with large projects.

2.4 Revit

Autodesk Revit is an architectural design and documentation software created by Autodesk. Revit is designed to support BIM, allowing to accurately design and document complex buildings, while also being capable of doing analysis on buildings. Revit defines objects as individual paremetric 3D shapes called families, representing walls, doors, and windows, among other things. Similar to ArchiCAD, it can create 2D and 3D models of the buildings. Revit also has the same limitations other BIM tools have. Revit performance does not scale well with the high amount of geometry created with AD and it only provides a simplified dynamic view. The biggest difference between Revit and ArchiCAD is the simpler

interface and the support Revit has for other Autodesk applications.

VIM AEC is a real-time visualizer based on Unity for Revit BIM models that introduces the concept of Virtual Information Modelling (VIM). In contrast with BIM, VIM aims to join the best of both worlds by integrating the interactiveness of game engines along with the model-enriching feature of BIM's metadata. Thus, resulting in this improved BIM concept capable of fast visualization. However, VIM AEC lacks features compared to Twinmotion and Lumion, such as a weather control and an asset library, realistic lighting.

2.5 Rhinoceros 3D

Rhinoceros 3D, or Rhino 3D, is a CAD application. Rhino 3D geometry is based on the Non-Uniform Rational Basis-Spline (NURBS) mathematical model, which has as main focus the creation of mathematically precise curves and free-form surfaces. NURBS contrast with polygon mesh-based models, commonly used in game engines and other rendering applications, where a model is represented by vertices, edges, and faces. Rhino 3D supports scripting with different programming languages: RhinoScript, a textual programming language, based on Visual Basic; Python; and Grasshopper, a visual programming language. Like BIM tools, CAD applications are aimed for interactive usage. However, when used in the context of AD, they perform poorly because of the high amount of geometry generated by an algorithmic description. Rhino 3D is no exception to this limitation.

Unlike ArchiCAD and Revit, Rhino 3D is capable of doing real-time rendering natively using OpenGL, but it does not support PBR materials, reducing the fidelity of the rendered image. Even though this can be fixed by using plugins, it is still a disadvantage over game engines. Another disadvantage is only allowing a simple free camera navigation system. Rhino is also capable of doing offline rendering with high quality using V-Ray, an offline rendering software.

2.6 SketchUp

Sketchup is a CAD application owned by Trimble Navigation. In Sketchup, 3D, the user inputs points and faces to create 3D shapes, contrary to Rhino 3D. The user can use these shapes to quickly create sketches of models which can be useful in early stages of the architectural project where the architect wants to quickly explore different designs.

Sketchup provides a dynamic view with simplified materials, light, and shadows. This application can also create cinematics and renders with low fidelity. Sketchup only supports free mode navigation, but with the use of SketchyPhysics, a plugin that simulates physics, Sketchup can also support additional navigation system. Sketchup is not designed for this purpose so it might not scale well with a large

amount of objects.

2.7 Unreal Engine

In architecture, game engines have been considered for visualization [16]. Game engines not only allow efficient rendering and interaction with architecture models, but they are also portable, meaning an architect can use different platforms to show their project. A study [17] showed that the usage of real-time rendering engines increases productivity and predicts the adoption of similar solutions will increase. Moreover, the use of real-time rendering engines also reduces the need for physical prototypes or mock-ups, and allows the client to quickly visualize the project during the design phase.

UE is a game engine developed by Epic Games. Initially, it was only meant to be used for game development, but with continuous development and releases of newer generations, its usage was broadened. The 4th generation of UE has become a complete suite of creation tools for game development, architectural and automotive visualization, and other applications. UE is written in C++ and it is open source. UE4 can be scripted with C++ or with a visual language called Blueprints. UE has the following qualities [2]:

Performance: the entire engine is written in C++, which is a programming language known for its performance;

Usability: UE also has a visual programming language that simplifies the scripting process for artists and also provides an easy to use interface;

Photo-realism: UE has an advanced shader system for materials and supports local and global illumination;

Low cost: UE is free to use and it offers a vast assets library for free;

Documentation: UE provides rich documentation with examples;

Open Source: currently, the entire C++ source code for UE is available and is included in the installation.

There are a lot of success stories about the use of game engines for real-time rendering, especially UE. Hellmuth, Obata + Kassabaum (HOK), a global design and architecture firm[3], used Datasmith, a tool from UE, and UE to aggregate CAD data from different sources and use the real-time rendering to help decision making and support presentations to the client. Another example is DevelopWise, an Australian real estate luxury apartment specialist. They develop Realspace [4], an application powered

[2] https://docs.unrealengine.com/en-US/index.html. Last accessed 23 Dec 2020

[3] https://www.unrealengine.com/en-US/spotlights/hok-architectural-visualization-aggregate-iterate-communicate. Last accessed 23 Dec 2019

[4] https://www.unrealengine.com/en-US/spotlights/real-time-technology-helps-buyers-fall-in-love-with-unbuilt-homes. Last accessed 23 Dec 2019

by UE that allows clients to explore the house before it is built. The render quality can be observed in figure 2.2.

Figure 2.2: Image taken from Realspace, using ray tracing

In the following subsections, we will explore UE's capabilities to do renders with high fidelity, geometry generation, and other interesting features for architecture visualization.

2.7.1 Photo-realism in UE

UE uses a set of computer graphics techniques to achieve photo-realism in real time. In figure 2.3, we can see an image taken from "Rebirth", a cinematic produced by Quixel. Quixel is a company that takes scans from various objects and landscapes, using them to create realistic textures. In this subsection, we are going to explain how it works and why it achieves such good results.

2.7.1.A Physical-Based Shading

The first technique we are going to talk about is Physical-based Shading (PBS) [10, 11]. Traditionally, light interactions were done through shading models and punctual lighting. Even though such a process easily implements PBS , it still did not take into consideration real life physics and provided poorer results. As such, PBS was further developed with the main objective of simulating light by doing approximations of the Bidirectional Reflectance Distribution Function (BRDF). BRDF [11] is a function that describes how

Figure 2.3: Image taken from "Rebirth", a cinematic rendered in real-time, powered by Unreal Engine and produced by Quixel.

much light interacts with opaque objects. One problem with PBS was the complexity for the artists to use it, but Disney [18, 19] developed a physics-based shading model whose main focus was to maintain the artist's control over the final product by simplifying user controls. Following Disney's success, other companies started using similar approaches to achieve the same results. One of these companies was Epic Games, developer of UE, although their approach had some differences, particularly regarding how real-time performance was achieved [10].

In PBS, this function is divided into two components: a diffuse component that represents the amount of light that is diffused, and a specular component that describes the specular reflection. In UE, the diffuse component is obtained using Lambert's model and the specular component using the Cook-Torrance microfacet specular model. In the Cook-Torrance model, a surface is composed of a collection of microfacets which define how rough the surface is, and the specular component is calculated by three other sub-components: a normal distribution function that describes the orientation of microfacets in a given point, a geometry function that calculates the self-shadow created by the microfacets, and finally, a Fresnel function that is used to simulate how light interacts with surfaces in different angles. Another property that PBS takes into consideration is energy conservation, i.e., the radiance, and the quantity of light, where light coming from the source cannot be lower than the light reflected.

Materials are an essential part of PBS because they define the physical properties of the object. In UE, these properties are base color, metallic, roughness, and cavity, where all these properties are described using textures. There is still one last PBS component, which is Image-based Lighting. Light does not only come from light sources, but it also comes from the environment. To reach this goal in UE, this is done by analysing the pixels of the image (for example, from skybox, which is an image representing the background), then converting then to radiance, and finally, applying the BRDF. This last technique

is important to simulate the light coming from the atmosphere and the surrounding environment. It is also possible to reflect the surrounding environment with this technique, improving the visual fidelity. The reflection is done through a series of mipmaps, i.e., pre-calculated images with progressively lower resolution, and based on the roughness of the surface, where a lower resolution image would be used instead.

2.7.1.B Illumination

Global illumination, in UE, is achieved through a tool called Lightmass. The objective of this tool is to create lightmaps, which are textures containing the effects of casting light sources onto static objects. Lightmass can only use stationary light sources and does not support changes in light intensity. Lightmass is also capable of translucent shadows. Translucent shadows are created when light passes through a translucent object. This can be used when light goes through stained glass, since it will cast a colored shadow. Screen space global illumination is another feature of UE that is used to complement lighmaps and create more realistic light interactions by adding dynamic light sources on emissive surfaces.

2.7.1.C Reflections

Unfortunately, Lightmass does not do reflections. To solve this problem, another set of techniques is used: cube mapping, screen space reflection, and planar reflection. Cube mapping is a technique used to map the environment onto faces of a cube. The environment is projected onto each face of the cube and the information is stored in a texture, aptly called cube map. In UE, a cube map can be imported or generated through a reflection captured by an object. This technique only takes into consideration the environment. Extending reflections to the rest of the objects in the scene requires extra work. Screen space reflection and planar reflection are techniques used to calculate reflections. Screen space reflection uses screen space information to calculate reflections, only being capable of reflecting what is on the screen. Planar reflection offers a more realistic solution by taking in consideration information off-screen. However, planar reflection requires the insertion of a special object and rendering the scene again from the direction of the reflection. Planar reflection is more adequate for big and high-reflective surfaces, while screen space reflection is a more generic technique.

2.7.1.D Shadows

Shadows are also dependent on the type of light used. Higher shadow quality can be achieved while using stationary and static light, but it also depends on what the architect wants to do with UE. For example, the architect might want to change the types of lights. If the architect wants to do a render, the

better option would be to use static light because it provides better global illumination and good shadow quality. However, if they prefer interaction with the model, static light would be a bad option because it does not take into consideration non-static objects.

2.7.1.E Ray tracing

Ray tracing [20] is a technique that has been used in photo-realistic visualization in offline renderers for a long time. Ray tracing allows more realistic shadows, reflections, and global illumination, but it requires considerable computation and is not possible to do in real-time. However, with the advancement of software and hardware, it is now possible to attain similar results with real-time tracing. The only disadvantage is that real-time ray tracing requires specific hardware that supports this new technology.

2.7.2 Generating Geometry in UE

During the process of developing a game, it is necessary to create 3D models of characters, spaces, and environments for the game though a process called modeling [9]. Modeling is done by model designers using 3D modeling applications, such as 3ds Max, Blender, and Maya. However, modeling requires time and programmers need models during the design stage to prototype and test concepts. One solution to this problem is to develop temporary models that are replaced in latter stages of the development when the artist finishes modeling. To circumvent this potential issue, game engines started developing features that allow the creation of simple geometry. In UE, this feature is enabled by CSG. This tool is interactively used through the interface of the correspondent application.

To use CSG , the user only needs to place brushes that define different shapes, with different characteristics. Brushes are objects responsible for generating shapes that are created by BrushBuilder, a class that defines the shape that will be drawn. These shapes have parameters that can be manipulated. For example, if we want to create a cube, we would place a brush in the desired position and select as the builder the CubeBrushBuilder. One advantage of using these brushes is that they already support geometric Boolean operations, meaning that it is possible to unite or subtract models in a quick and simple way. With the use of geometric Boolean operations, architects can create complex geometry by combining simple geometry. However, brushes have a cost. They are heavier on the system than models created by modeling applications. The reason why brushes are less efficient than imported models is because they are generated dynamically and can constantly be changed. To increase their performance, UE allows their conversion into static models that can no longer be changed, similarly to models created by modelling applications.

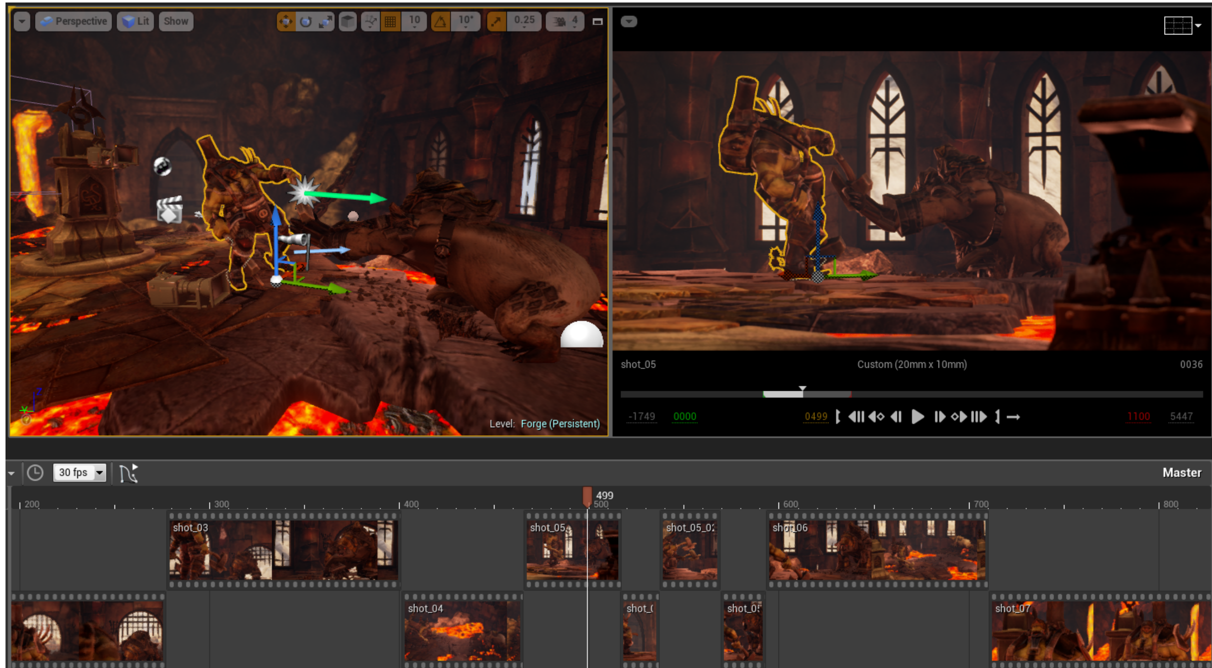


Figure 2.4: Sequencer Editor gives users the ability to create in-game cinematics with a specialized multi-track editor.

2.7.3 Additional Architecture Visualization Features

There are additional tools that have been developed in UE with the goal of helping visualizing architectural projects. These tools are also considered in this research as they can contribute towards improving the quality of the final solution. The first tool is called SunPosition. SunPosition can simulate light interactions with the atmosphere and allows sun position-based longitude, latitude, date, and time. With this tool, architects can easily show the clients how the building will look like during different hours of the day. The second tool is the Sequencer Editor. This tools allows the user to create cinematics rendered in real time, offering a multi-track editor where the user can place sequences, as can be seen in figure 2.4. Sequences are real-time rendered scenes that a user programmed. They can contain model animations, transformations, and sounds. With this feature, architects can create complex cinematics that would only be possible with video editors.

2.8 Unity

Unity is a cross-platform game engine. Unity can be scripted with two different languages: C Sharp, and UnityScript, a version of JavaScript. Because Unity is a game engine, it already has components with the purpose of rendering and simulating real life physics. Unity also has a well-developed virtual store, the Unity Asset Store. The store provides the users with a library of assets of their choice. These assets

can be, for example, physics-based materials, furniture, or shaders, which can help achieve a higher visual quality.

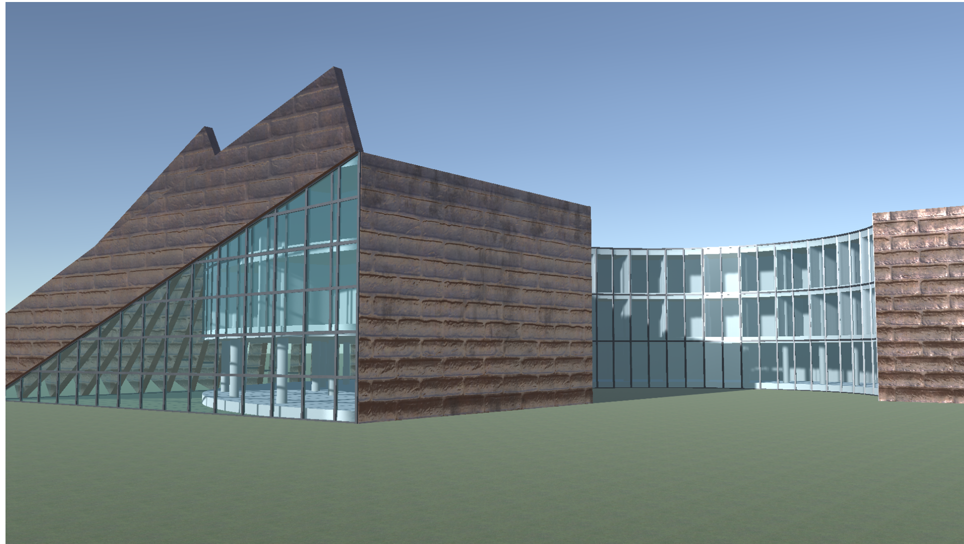


Figure 2.5: Render produced in Unity

Unity provides a standard shader[5] that is capable of rendering materials like stone, wood, glass, plastic, and metal. This standard shader also uses PBS to simulate the light interactions. For a novice user, this shader can be useful because he does not need to have extensive knowledge of computer graphics to achieve a realistic visualization, although this not always happens [21]. Sometimes it is required for the user to develop a shader that better suits their needs. UE has a better solution for this problem: materials incorporate the shader and can be edited through visual programming, making it possible to develop a shader that has a more realistic look for each material. Another disadvantage of Unity is that it requires a third-party library to do Boolean operations. In Unity, there is a tool called ProBuilder that can use CSG to generate models, similarly to Brushes in Unreal.

Unity can use ray tracing to calculate high quality global illumination, reflections and shadows, but this requires specific hardware that supports this technology. Global illumination in Unity can also be calculated with less quality by using lightmaps and probes. Alas, Unity requires the meshes in the scene to be static. Unity is incapable of doing translucent shadows without real-time rasterisation. In terms of navigation, Unity provides a good experience because, once again, it is a game engine. It has a physics engine and is capable of managing user input, which allows to automatically detect collisions and to easily develop the three different navigation systems mentioned previously. Moreover, Unity, when compared with classic CAD/BIM visualization solutions, shows significant improvements in performance [7] due to the techniques used in game engines to achieve real-time rendering. However, in complex environments with a lot of meshes, like models developed with AD , the performance is lower

[5] https://docs.unity3d.com/Manual/index.html. Last accessed 23 Dec 2019

Tool Name	Real-Time rendering	High-fidelity render	Produces Documentation	Support Analysis	Navigation
Luna Moth	$\surd$	χ	χ	χ	χ
OpenSCAD	$\surd$	χ	χ	χ	χ
ArchiCAD	χ	offline render	$\surd$	$\surd$	χ
Revit	χ	offline render	$\surd$	$\surd$	χ
Rhino 3D	$\surd$	offline render	χ	χ	χ
SketchUp	$\surd$	χ	χ	χ	χ
Unreal	$\surd$	$\surd$	χ	χ	$\surd$
Unity	$\surd$	$\surd$	χ	χ	$\surd$

Table 2.1: Comparison between AD visualization back-ends, based on features used by architects.

when compared to UE [21] which reduces its quality of interactivity [15]. A scene in Unity is composed by game objects [7], which represent every object in scene. Game objects have components that define what the game object does. Unity was extended to support the Khepri AD tool by creating a game object that is responsible for managing user interactions and generating the model. This game object is necessary because Unity does not allow concurrency while manipulating a scene since it requires some component to balance it. Otherwise, the user would not be able to interact with the scene. The model creation is done through primitive game objects that can be placed and transformed in the scene to construct entities, or through complex geometries as custom elements built using Khepri. In figure. 2.5, we present a building rendered in Unity and algorithmically described in the Khepri AD tool.

2.9 Comparison

In this section, we compare the visualization tools mentioned previously based on their capabilities and their purpose during the design process. In Table 2.1, we present a summary of our analysis. Both Luna Moth and OpenSCAD offers features like traceability and interactivity that speed up the scripting process. Rhino 3D and SkecthUp present features that are useful during the early stages of the architectural design process. This means that an architect can explore different designs quickly due to these tools' capabilities to generate a model through scripting. This means that an architect can explore different designs quickly due to these tools' capabilities to generate a model through scripting. BIM tools, such as ArchiCAD and Revit, are capable of storing data and structural information within the model, with the purpose of doing analysis on the building, producing technical documentation, and creating a workflow with less room for human error.

All of the tools mentioned previously are incapable of doing high fidelity render in real time , or do not perform well with high amounts geometry which is common in AD projects. Without high-fidelity rendering in real time, architects have to use offline rendering to generate renders with high quality, and offline rendering is not a quick process, requiring several hours or even days to produce results. This

process becomes even more painful in the case of AD projects, as these can be easily altered, which requires the regeneration of the entire model. This problem coupled with high rendering times makes the AD approach slow and undesirable. Furthermore, offline rendering is limited, since it only offers images and videos, while real-time rendering can also offer interactivity with the model. This is where game engines like Unity and Unreal stand out, because they offer real-time rendering and interactivity while being able to run in computers, phones, and consoles.

However, Unity is not a perfect solution, particularly when compared with UE. Unity uses a more simplistic approach to materials that does not support virtual textures which can limit the material complexity. Unity also has poor global illumination without real-time rasterisation when compared to UE. The latter uses better lightmap calculation techniques which create translucent shadows. However Unity and Unreal can produce high quality global illumination when they use real-time rasterisation techniques. Moreover, in complex environments, UE can have better performance than Unity [21], which is an important aspect because poor performance can degrade the interactive visualization. However Unity and Unreal can both achieve the same level of high fidelity rendering when using real-time ray tracing. Additionally, UE has more features designed for architecture visualization and UE is an open source solution, making the process of adapting such tools to architect' needs simpler.

From this analysis, we can conclude that UE is the tool that better fits our requirements. UE is capable of doing high-fidelity renders while simultaneously allowing a high level of interactivity with the model. As such, we will use UE as a visualization tool for AD.

3

Methodology and Implementation

Contents

3.1 Architecture Design Requirements 25
3.2 Model creation 29
3.3 Render Process 34
3.4 Navigation 38

Our goal with this research, as mentioned in the introduction, is to develop a tool that provides a photo-realistic view, in real time, of a complex AD model. Real-time rendering is an important quality for the AD workflow because delaying the visualization makes AD frustrating, and it also helps the architects share their vision. Real-time visualization requires the use of an application capable of real-time rendering with high quality. For this reason, we decided to use UE, because it shows good performance rendering complex models compared to other applications. Also, UE is a game engine capable of doing rendering in real time, something that is not common place in typical visualizers for AD tools.

Our first step while developing this tool was to extend UE to support the AD approach. In the AD approach, an architect does not directly describes the digital model, specifying instead an algorithm that describes the digital model. In our solution, we specifically targeted the Khepri AD tool. We chose Khepri because the tool we developed is meant for later stages of project development. In latter stages of development, the project already went through the modelling and design process and the architect requires visualizers to make aesthetics decisions and generate renders. Khepri is capable of generating a model in different tools based on the project's stage. This quality requires that the architect describe only a single algorithm, instead of one for each intervening tool. The workflow of our solution will be the following: (1) an architect describes the model through an algorithm using Khepri, and (2) Khepri will then generate the model in UE. This workflow requires the creation of a plugin in UE, which serves as an interface for Khepri. In figure 3.1, we can see the architecture of our solution.

In the following sub-sections, we will delve into the proposed architecture in more detail, specifically looking at the model generation, rendering processes and navigation systems. We will also explain and justify the decisions made. All tests in following sections are done in a machine with following hardware: i7-4770, Nvidia 960GTX and 16GB RAM.

3.1 Architecture Design Requirements

Before diving into the details of our architecture, it is important to identify what are the requirements of our solution. The requirements are the following:

1. A user must be able to generate a digital model through algorithms;
2. The tool must be able to pause and resume generating the model at any time;
3. Our solution must provide an AD interface to UE ;
4. Our tool must be able to create close to photo-realistic renders in real time.
5. Our tool should be modifiable.

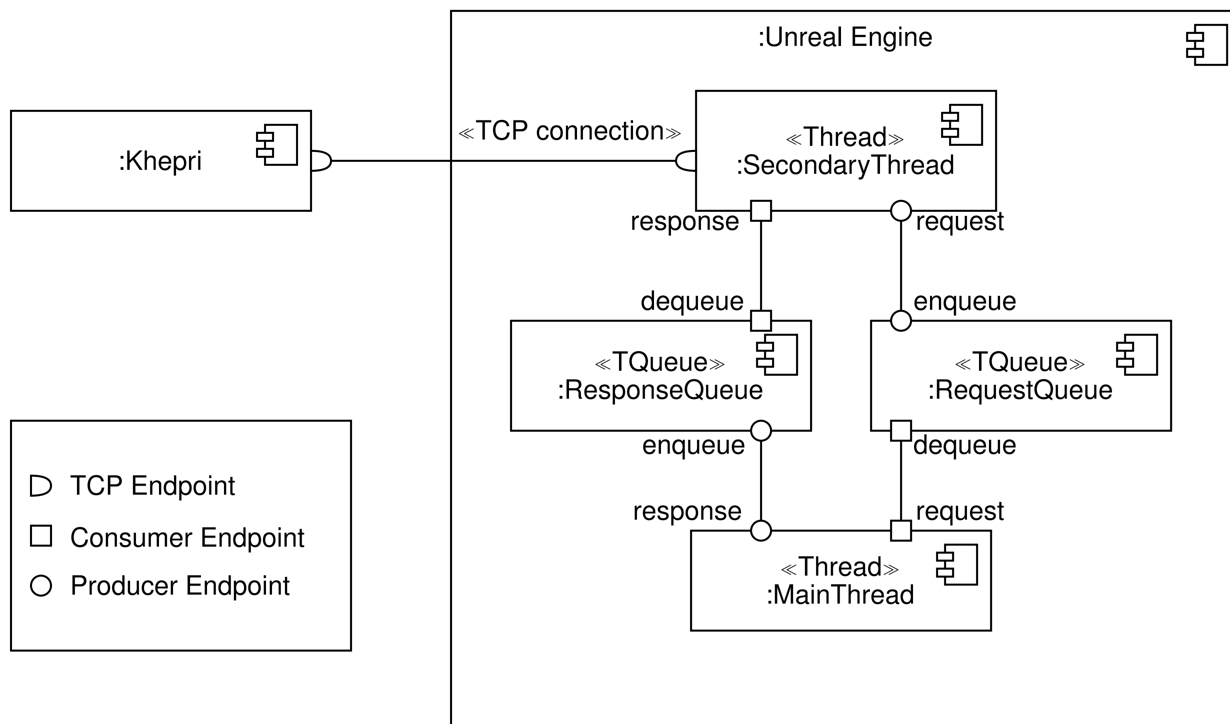


Figure 3.1: Component-and-Connector view of our solution

To fulfill our first requirement, we used Khepri as the AD tool in which the user describes the algorithms responsible for generating the model. This tool communicates through a Transmission Control Protocol (TCP) channel with UE and it makes remote procedure calls to generate the digital model in the game engine. Khepri can communicate with different visualization tools, providing different communication channels for different tools. This presents a big quality because some of these channels can be reused for other tools. In our case, we decided to reuse the communication channel used to communicate with ArchiCAD , since both UE and ArchiCAD share the same programming language, C++.

Still, the communication channel is only the first step in allowing communication between UE and Khepri. We also need to create a Remote Procedural Call (RPC) server in UE, which allows a set of functions to be called remotely. This lead us to develop a plugin for UE. This plugin is responsible for receiving all remote calls and executing them. The plugin is also responsible for managing and storing information about objects requested by Khepri and for translating this information between Khepri and UE .

With the use of RPC communication, UE has to wait for requests from Khepri.However, this wait must not block UE or it would not be able to respond to user inputs, like moving or rotating the camera. This creates a conflict between our second and third requirements. In order to allow the user to interact with the visualization tool, while it waits for requests from Khepri, we decided that the communication

channel should be handled in a separate thread, different from the main thread that handles the user interaction.

This new thread will be responsible for receiving and translating information between Khepri and UE. This thread then forwards all requests to the main thread. In UE, only the main thread has permission to access UE's memory space, meaning only the main thread is capable of creating new objects in the scene. Figure 3.1 shows a view of our solution. This new thread solves the conflict between our two aforementioned requirements, because it can listen for new requests while the main thread is free to respond to user input. It also makes it possible to stop and continue the process of generating a digital model without closing the communication channel. With the use of concurrency, there is also a newly added overhead due to the need for threads to communicate with each other. With this in mind, we analyse the cost of employing the concurrent approach and evaluated the viability of this solution. To measure this cost, we stored timestamps before sending and after receiving information in arrays, outputting these values after the operation is completed. This way, we can reduce the additional overhead created by obtaining the timestamps and outputting them. With this test, we can observe that, on average, the time it takes to send a message between threads is less than one millisecond, which is an acceptable delay.

In our solution, the background thread does not directly forward the request to the main thread. Instead, it creates an object around it (called operation), that represents the request. This allows the background thread to simplify the request, which reduces the time it takes for the main thread to respond. For example, when the background thread receives a request to create an object, before creating the operation, it calculates the pitch, roll, and yaw. This calculation simplifies the request because the main thread no longer needs to do this calculation. Another advantage of using operations is that the background thread can respond to some requests. For example, if Khepri sends a request asking how many actors are in the scene, the background thread can respond to this request by counting how many creation requests were done.

3.1.1 Optimizations

The communication channel that is used between UE and Khepri is intended for synchronous calls. When the RPC client sends a request, it expects to receive a response back. Another important detail is that the main thread only checks for requests after rendering a frame. Both these characteristics limit UE to execute a single operation per frame, because the main thread does not wait for new operations coming from Khepri after executing one. This quality increases the interactivity, but it also increases the time it takes to generate a digital model. These qualities go against an smooth AD workflow. An architect requires a tool capable of quickly generating the model so they can equally quickly visualize their project and make decisions.

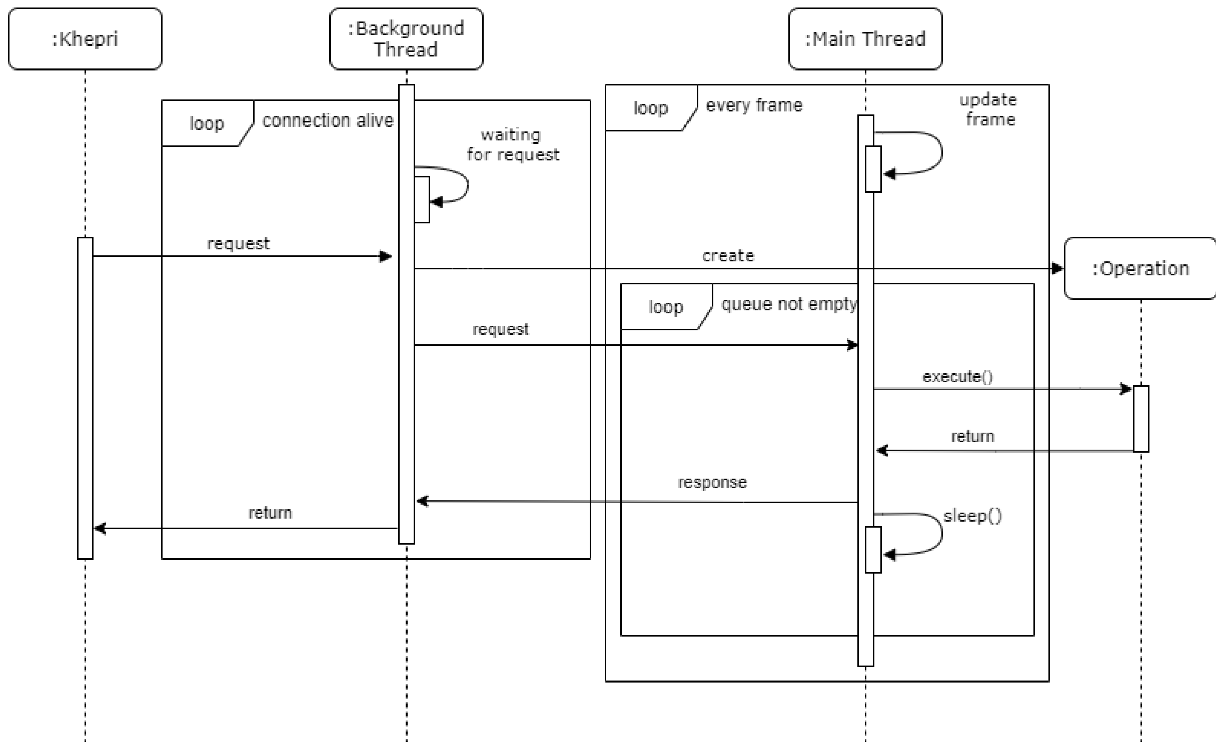


Figure 3.2: Unified Modeling Language sequence, showing how the communication between UE and Khepri works.

In figure 3.2, it is possible to visualize how we solved this issue. After the main thread executes an operation, it is put to sleep for a short amount of time. This sleeps forces the main thread to wait for another Khepri's request and skip a frame, saving time when the sleep time is shorter than time it takes to generate a frame. We used different periods of time based on the time Khepri takes to respond. We increase the sleeping period exponentially until it is greater than the time it took for Khepri to receive the last request. The sleeping period cannot grow exponentially to infinity because we limited it to 32 milliseconds. Higher values would not make much sense because we want UE to be interactive. Clearly, this sleep time incurs a small overhead. However, this overhead does not affect interactivity with UE because it only affects UE while UE is receiving requests, and is extremely small compared to the time saved by skipping a frame.

Number of Operations	Time	Without Concurrency	With Concurrency	With Optimized Concurrency
100	Total	0.5446	26.9905	0.7243
	Per Operation	0.0054	0.2699	0.0072
1000	Total	5.3868	240.3395	5.8926
	Per Operation	0.0054	0.2403	0.0059

Table 3.1: Time it takes to create objects in UE with and without concurrency, and with optimized concurrency. The results are shown in seconds and all of the objects created are exactly the same.

In table 3.1, we can observe that the results achieved by using this optimized concurrency are similar

to the ones that did not use concurrency. With this optimization, we can have fast model generation and allow high level interaction with UE.

3.2 Model creation

Before identifying and explaining the methods used to generate the digital model, it is important to know how UE represents geometric objects and how it organizes them. The geometric objects that collectively represent the model are polygon meshes, composed of a set of vertices and faces. All geometric objects are placed in a scene. In this scene, there can be multiple types of objects, and not just geometric objects. All the objects that can be inserted in a scene are called actors. In our solution, we are required to create different actors based on Khepri's requests.

One of the challenges that we found during the implementation is the inconsistency between geometric descriptions in Khepri and UE. The problem exists because the descriptions are independent from the tool where the model will be generated. This means some descriptions might not be suitable for UE. One example of this is unit scale: in UE, one unit is one centimeter, while in Khepri, one unit is one meter. To solve this issue, we made the background thread translate the descriptions, as mentioned in the previous section.

In the following subsections, we will describe two different methods that are used in our solution to create geometry. Furthermore, we also explain an optimization implemented in our solution.

3.2.1 Brushes

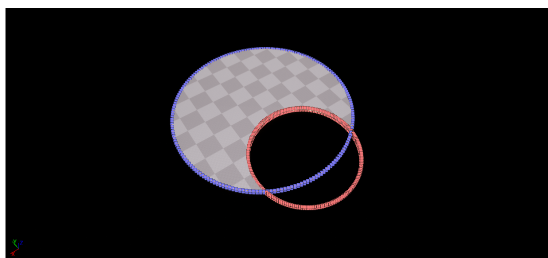


Figure 3.3: Creating a slab with brushes.

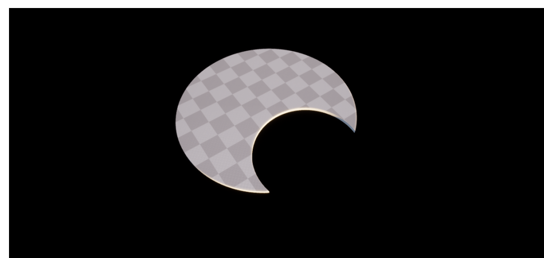


Figure 3.4: Slab created with brushes.

In our first method, we used brushes to generate geometry. Brushes are special actors and they are associated with a builder responsible for generating geometry. Builders are associated with different geometric primitives, which can be cubes, cylinders, cones, pyramids, etc. Coincidentally, Khepri primitives are also based on these geometric primitives. This means that we can develop different builders for each geometric primitive required by Khepri and make it possible to generate the correspondent digital model in UE. Additionally, we can perform Boolean operations between brushes, which means we can

create complex geometry by using geometric primitives. We used this property to build slabs. Slabs are extruded surfaces, as the one that can be seen in figure 3.4. These surfaces can have holes and they are defined through a sequence of vertices. We can use Boolean operations between brushes to simplify the construction of slabs, as can be observed in figure 3.3, where the red circle is being subtracted from the blue circle.

In our implementation, we developed a builder for each primitive required by Khepri. Due to the fact that brushes perform badly in real-time rendering, we converted brushes into static mesh actors. Static mesh actors are actors that are used to represent meshes. Additionally, these meshes cannot be manipulated in real-time because they have to be static and they also have to be stored in memory. In our solution, we also created a special builder capable of converting a mesh back to a brush. Using this builder, we can do Boolean operations between static meshes.

Figure 3.5: Slab created by brushes with a square tile texture.

Unfortunately, brushes cannot correctly generate non-convex surfaces. This is a serious problem, because it creates meshes with poor texture mapping in non-convex faces. In figure 3.5, there is an example of this issue. In the middle of the image, there is an inconsistency that is created due to poor texture mapping. The reason why this happens is because brushes use Binary Space Partitioning (BSP) to store data and BSP was not developed with non-convex surfaces in mind, instead dividing non-convex surfaces into multiple independent convex surfaces. Applying a texture to a mesh requires a two dimensional image to be projected onto a three dimensional object. This process is called UV mapping, where U and V are the 2D axes. An example of this process can be seen in figure 3.6. The separation into multiple convex surfaces makes UV mapping irregular, because each convex surface is mapped separately. This problem made us look for a different solution that provided a higher level of control in

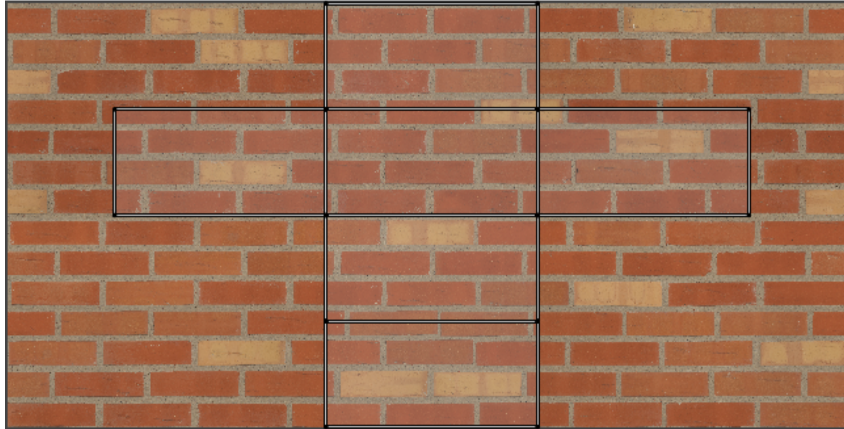


Figure 3.6: The projection of a texture into a cube.

UV mapping while generating geometry.

3.2.2 Primitive Method

Our objective is to provide a realistic view of an architecture project. For this reason, it is important that generated meshes have the correct UV mapping. UV mapping, when done correctly, allows textures to have the desired appearance when applied to a mesh. To achieve this, we took a more primitive approach where we have more control during the mesh creation process. The approach is based on the use of a structure called FRawMesh. With this structure, we can describe the vertices and polygons that compose a face of a mesh. Furthermore, we can also map textures correctly by providing the UV coordinates for each vertex. By using this method, we also need to calculate normals, tangents, and cotangents for each vertex. Since the UV mapping is different for each primitive, we created different builders based on primitives, similarly to the brush approach. All the faces of a 3D object in UE have to be triangulated, decomposed into polygons with three vertices. Brushes did this automatically but, with the FRawMesh approach, we also have more control over the triangulation process. This allows us to use different algorithms capable of doing triangulation and this means we can now use algorithms that are capable of doing triangulation of non-convex faces, allowing us to create objects, such as slabs, without using Boolean operations.

Algorithms	Complexity	Complexity with holes
Triangulation by partition into monotone polygons	$O(n \log n)$	$O(n \log n)$
Dynamic programming algorithm	$O(n^3)$	$O(h * n^2 + n^3)$
Triangulation by ear clipping	$O(n^2)$	$O(h * n^2 + n^2)$

Table 3.2: Triangulation algorithms in the library polypartion and their complexity without and with holes where n is the number of vertices and h is the number of holes.

We used a lightweight C++ library called polypartition to triangulate the non-convex surfaces. This li-

brary contains three different algorithms: ear clipping, optimal triangulation in terms of edge length using dynamic programming algorithm, and triangulation by partition into monotone polygons. Their complexity can be seen in table 3.2. Because we prioritize model generation speed, we chose triangulation by partition into monotone polygons.

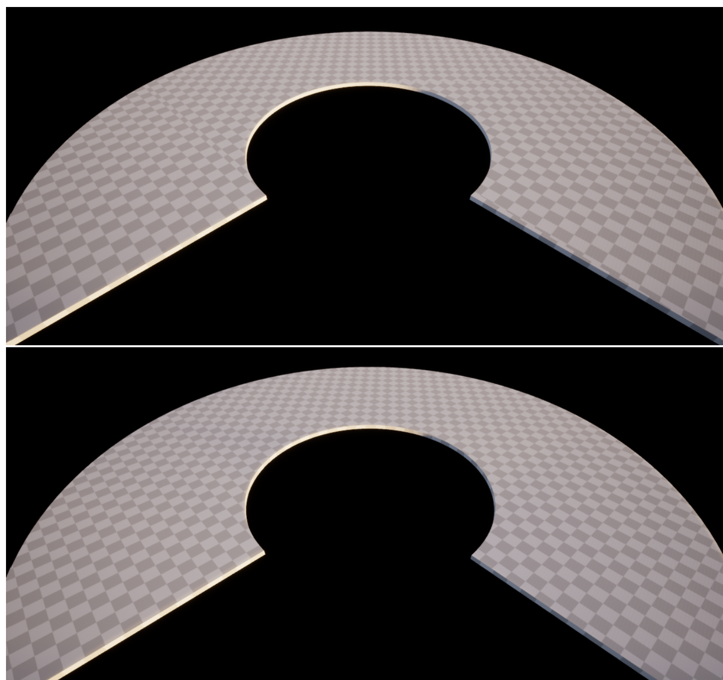


Figure 3.7: Slab created by the brushes method (above) and the primitive method (below). Note, in the top image, that the texture is divide into different areas. This problem does not occur the bottom image.

This more primitive approach solves the issue created by non-convex surfaces, because now we can map the mesh correctly. In figure 3.7, we can observe slabs created by the brush method and the primitive method. In the top image, we can observe the problems that brushes have, while in the bottom image, we can see that the texture has the expected result.

This new approach also shows to be faster than the previous one. We created 50 cubes using both methods. It took, on average 166 milliseconds to generate each cube while using brushes. Meanwhile, it took, on average, 110 milliseconds to generate each cube when using the primitive method. The results show that the more primitive approach can create meshes faster, which results in a larger number of operations per second. The only disadvantages of this approach is that it is not able to do Boolean operations and requires more effort to map textures.

Lastly, generated cylinders also look more realistic while using the primitive method. In UE, cylinders are represented through a prism with a large amount of sides. When using brushes, each side is mapped independently, which makes cylinders look unrealistic. However, when using the primitive method, we can map the texture correctly by mapping all sides together. In figure 3.8, we can see a cylinder created

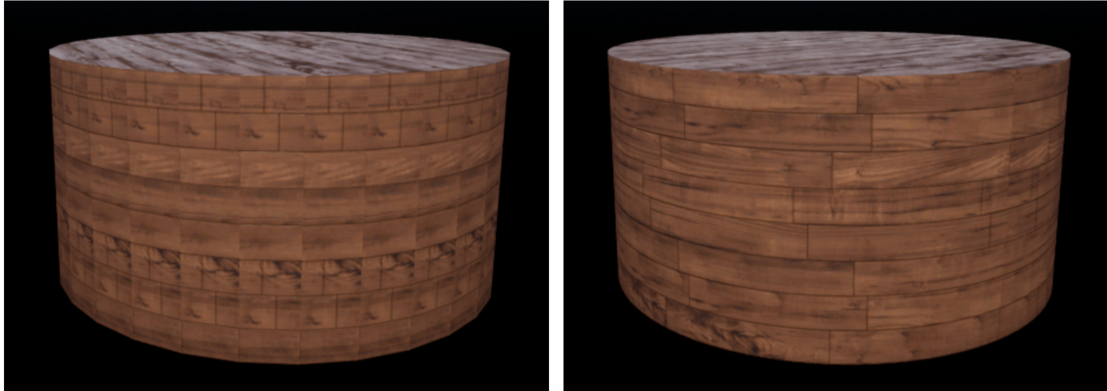


Figure 3.8: In the left image, we can see a cylinder created with brushes between tiles. In the right image, we can view a cylinder generated with the primitive method.

by each method.

3.2.3 Optimizations

Both previous methods are used to generate meshes. As you can see in the table 3.3, this process has a significant cost in both methods. The brush approach can be divided into two phases. In the first phase, the brush is inserted in the scene with the desired parameters. In the second phase, it is converted into a static mesh actor. The primitive method can also be divided into two phases. The first phase is the mesh generation process, and the second is the insertion of a static mesh actor with the created mesh into the scene. As a reminder, this last phase is necessary because brushes are designed to be a temporary tool.

Method	First phase	Second Phase	Total
Brush Method	0.058	0.052	0.110
Primitive Method	0.003	0.166	0.169

Table 3.3: The time it takes on average to create a cube with the brush method and primitive method.

As we mentioned previously, we are using static meshes to represent models and all static meshes are kept in memory. This means that it is possible to reuse meshes that were already created. In our solution, we created a mechanism to reuse cuboids and cylinders. The reason why we do not do it for the other primitives is because they have arrays as parameters, which makes it harder to find similar meshes because they can have different sizes and values. Additionally, these primitives are also less used and have smaller chances of repeating.

To rapidly find similar cuboids and cylinders, every time a cuboid or cylinder is created, e cache its name along with the its parameters. With this in mind, we just have to look for a mesh with a certain name based on parameters. If the mesh already exists, we just create a static mesh actor with the

already created mesh. If the mesh does not exist, we create the mesh before inserting the actor. This process might incur an overhead, because we always have to search for the mesh before creating it. To measure the cost of this overhead, we inserted a timestamp before and after the search. We performed 50 searches and recorded the timestamps. With this test, we observed that the overhead is less than one milliseconds and barely has any impact in performance. This means that the time it takes to create a cube becomes close to the time it takes to create an actor, if the cube was already generated.

To verify the gain of our optimization in real architectural projects, we measured the time it took to generate a digital model using the brushes and the primitive methods with and without our optimization. We used as an example the Isenberg Business Innovation Hub building (designed by BIG Architects), shown in figure 2.5. The results can be found in the table 3.4. The optimization showed a significant gain in performance.

	Brushes	Primitive method	Brushes with cache	Primitive method with cache
Time to generate Isenberg	552.69	323.45	263,44	121.94

Table 3.4: The time, in seconds, it takes to generate the Isenberg Business Innovation Hub's model when using different methods.

3.3 Render Process

In this sub-section we will explore a technique called texture synthesising which allows a person to use a small sample texture to generate a bigger texture. We explore UE material capabilities to do texture synthesis. Finnally, we also explain the process used to generate cinematics in UE.

3.3.1 Textures

As we mentioned in a previous section, good texture quality is essential to create photo-realistic renders. Because textures are images, they have limited size, which means, for large surfaces, texture may look less realistic. In this section, we will discuss techniques that can be applied to solve this issue.

A texture is an image projected onto a model. This process requires mapping the texture in the model. However, an image will inevitably be too small to cover all of the surfaces of a model. There are two solutions for this problem. The first solution is stretching the image until it has the required size, which makes a texture look different based on the surface's size. The second solution uses tiling. As we can see in figure 3.9, the image is repeated to increase the texture's size [11]. Tiling solves the issue of a texture looking different on different surfaces, but creates repetitive patterns. This repetitive pattern gives the texture an unrealistic feel as such patterns do not existing in real life.

To solve this issue, one must resort to algorithms that will use the original texture image and synthesize a bigger version of the initial image [22]. These algorithms create new tiles with variations and blend them. To perform this task, it is important to consider the type of the texture. We can divide textures into two different groups: (1) textures with stochastic origins, and (2) textures with geometric patterns, like tiling floors and brick walls.

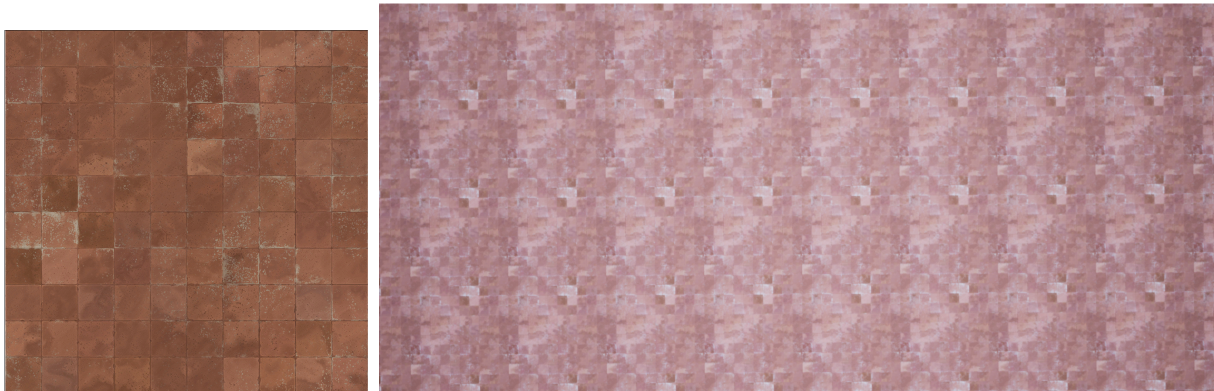


Figure 3.9: A geometric texture applied to a big surface. In the left image, we can see the texture. In the right image, we can view the texture applied to a huge surface.

For textures with geometric patterns, it is important to maintain the geometric pattern [23]. For stochastic textures [24], we must only consider generating tiles with similar appearance and blend them. Because the different types of textures have different needs, it is important to use different algorithms for different types of textures. In our solution, we use shaders for this purpose.

For geometric textures, we use a tile blending approach. The tile blending approach divides a texture in sections, called tiles, and distributes them randomly. In our implementation, the texture is divided into rectangles. The size of the rectangle is dependent on number of columns and rows of tiles. In figure 3.10, we can see the result of this shader. In the image, we used a texture that is a colored number grid. On the left, we can see the original texture; on the right, we can see the texture applied to a larger surface when we use the shader. As we can see in the image, there is no longer obvious patterns and the numbers are no longer organized numerically. In certain scenarios, when tiles are distributed randomly, inconsistencies can be created. This is observable in the figure 3.11. Because bricks have different colours in different tiles, it creates inconsistencies when two tiles with different colours are placed next to each other. The inconsistency is created due to the abrupt change in colour. A possible solution is to blend the tiles. In our solution, the blending is done through linear interpolation and it hide abrupt change in colour. In the shader, the blending has parameters, so the user can find the best values for a certain texture.

For stochastic textures, we also use a tile blending technique. However, this technique uses three different types of tiles. The first two types are squared tiles, with different offsets, and the third type

Figure 3.10: Numeric texture using tile-blending. In the left image, we can see the original texture. In the right image, we can view the texture applied to a larger surface using tile-blending.

Figure 3.11: In the left image, we can see the texture without blending between tiles. In the right image, we can view the texture using blending.

is based on a rotation. The shader also supports hiding texture patterns by blending tiles with different blending levels. We can see the results of using different blending levels in figure 3.12 where the different types of tiles are represented by different colours.

Unlike the previous shader, this one uses three different tile types that hide texture patterns. In figure 3.13, we can observe an example of the use of this shader. Both these shaders are only meant to be used in extremely large surfaces where noise textures are not large enough. Noise texture are generated images created by algorithms, like perlin noise, that can also be used to hide repetition patterns.

In UE, shaders are incorporated with materials. This allows to create specialized shaders for certain types of materials. A material can have instances, each of which can have different textures and parameters while using the same shader. We created two different materials, one for geometric textures and another for stochastic textures. With this approach, an architect can easily add new materials just by creating a new instance with a texture and the correct parameters. These shaders were developed

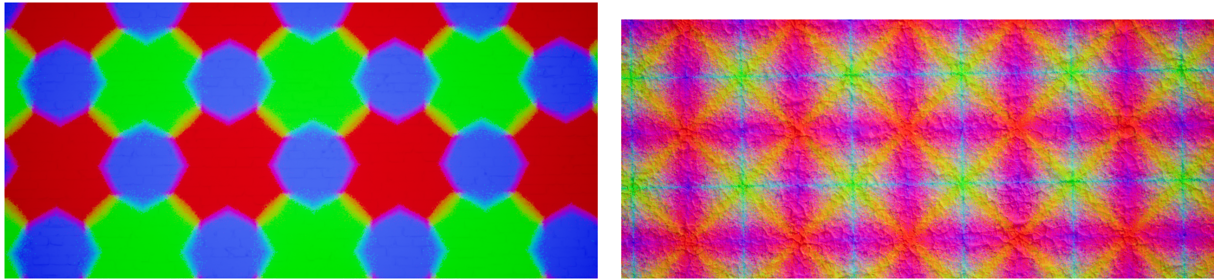


Figure 3.12: Blending process colored with low and high blending respectively.

Figure 3.13: Stochastic texture synthesized.

to explore the UE material capabilities and show what UE can do while only using UE's standard render pipeline. For this reason, they will not be taken into consideration in our evaluation.

3.3.2 Rendering

One of our main objectives is to allow architects to create realistic renders to communicate their design. Consequently, it is important to present their model in an appealing manner. As such, it is important to explore what UE can achieve and take advantage of its qualities. In UE, there is a tool called sequencer that allows the users to plan and generate cinematics. The sequencer provides a timeline and the user can change properties of actors in the scene during this time line. These properties can be actor position, rotation, visibility, among others. With the use of the sequencer, architects can create cinematics where characters interact with the digital model. This is possible by changing the proprieties of the different actors in the scene. For example, during a cinematic we can open doors by changing the rotation of the door's actor across time.

To create a simple cinematic, we only need a name for the sequencer and an actor that represents

the camera. Fortunately, Khepri also has the same requirements. The only difference is that Khepri expects to create a render every time it sends a rendering request. We can solve this by storing camera position and rotation every time this request is sent and render the frame. This way, we maintain the functionality Khepri expects and also save every camera position in a sequencer. The architects can latter edit this cinematic as they wish through this sequencer.

In UE, creating renders is not a simple task. It requires users to use a specialized view that copies the camera's proprieties. The properties are focal length and aperture. Without this view, we would not have as much control over image resolution. The specialized view is created every time render request is received. Because this specialized view takes some time to create, we also developed another method where the cinematic is only generated after receiving every camera position in the sequence.

3.4 Navigation

One of our objectives with our solution is to also explore different navigation systems in UE. We developed three different navigation systems: free camera, walk mode, and VR mode. The navigation system can be changed through the game engine interface.

In the next chapter, we evaluate our solution in terms of image quality and performance when compared with other visualizers.

4

Evaluation

Contents

4.1 Image Fidelity 41
4.2 Performance 46
4.3 Summary 48

In later stages of the development of an AD project, architects require a view of the digital model with high fidelity which allows to make aesthetics decisions and generate renders that share their vision. For this purpose, architects look for visualization tools. As we mentioned in chapter 2, game engines can create high-fidelity renders, provide navigation systems that allow architects to quickly view the model, and they can also generate multiple frames per second.

To evaluate our solution, we compared it with another AD visualization tool that is meant to be used during the same stages of AD project development as our solution, which is Unity [7]. Unity can generate digital models and renders faster than typical CAD and BIM visualization tools. Both of those qualities make Unity an excellent visualization tool for later stages of project development.

In the following section, we compare our approach using UE with the approach using Unity, specifically comparing render image fidelity, model generation performance, and render production performance. To evaluate image fidelity, we compare the limitations in materials and light interactions that our approach and the approach using Unity have. We do not evaluate shadows due to both UE and Unity using the same techniques to create shadows. To evaluate performance, we measure the time UE and Unity take to generate renders and the digital model. We also take in to consideration how UE and Unity react to complex models by measuring frames per second. All of the tests in the following sections were performed in a machine with the following hardware: Intel® Core™ i7-4770, Nvidia GeForce GTX960 and 16GB RAM.

4.1 Image Fidelity

As mentioned before, one of our objectives is to provide a close-to-realistic view of the digital model to help the architect make decisions. For this reason, it is important to know the advantages and disadvantages that our solution has when compared with another real-time visualizer with fidelity. We evaluate the quality of materials and lights from our approach and the approach used in Unity.

In this section, all the renders done in Unity and UE used dynamic shadows to create shadows and they did not use lightmaps. The reason why we did not take into consideration lightmaps is because this process can take multiple hours, even on simple models, which goes against the purpose of tools whose goals is to provide a view of the digital model quickly. We only took in consideration the appearance of the digital model right after the model is generated. We also used an offline renderer to create renders as a reference for correct light interactions. For this purpose, we used a plugin in UE that allows Octane, an offline renderer, to render scenes from UE. For the following renders, we used the digital model of the Isenberg Business Innovation Hub.

4.1.1 Materials

Materials are a set of textures that define the appearance of an object. The types of a material's textures depend on the shaders' inputs. In the approach used in Unity, there is a limitation on what material shaders can use. This happens because the approach using in Unity does not adapt the model's UV, which makes Unity only able to use shaders independent from UV mapping. In figure 4.1, on the left, we can see all of the parameters of a standard shader independent from UV. This limits the variety of shaders that can be used, which can make some materials look less realistic, because we cannot use the most appropriate shaders. This problem can make some materials in Unity have less detail than materials in Octane and UE , as we can see in figure 4.3.

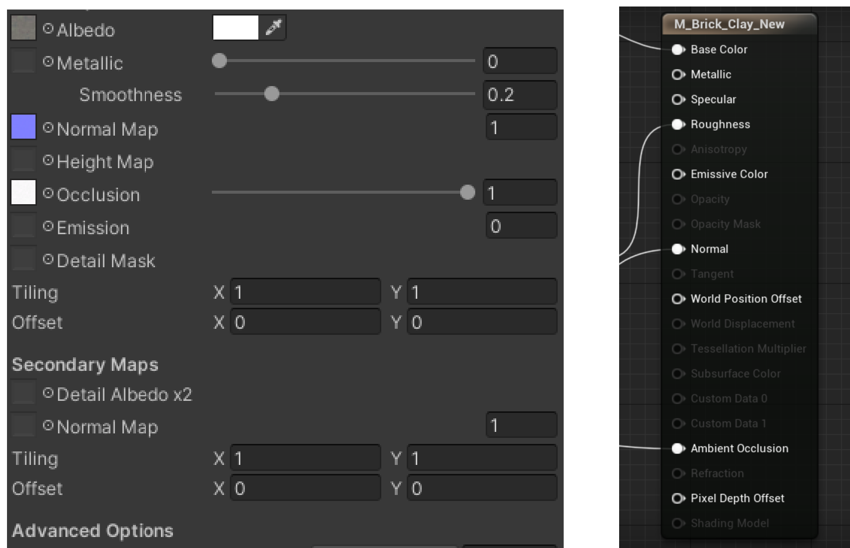


Figure 4.1: Shader parameters in Unity and UE

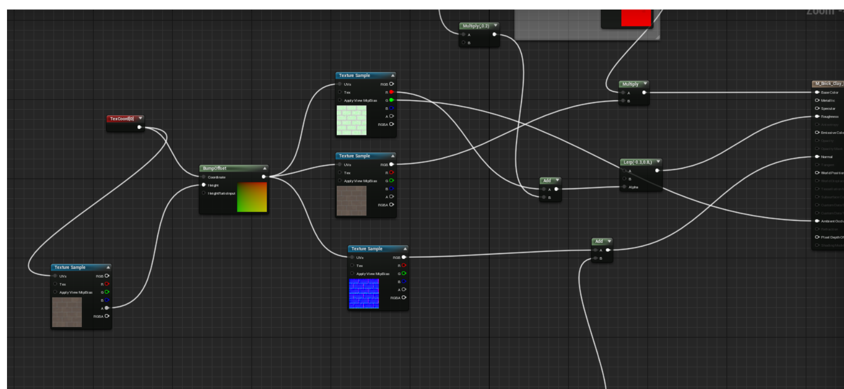


Figure 4.2: Creating a virtual texture as input for a brick material shader.

In UE, materials have a special property that allow the extension of shaders through Blueprints, a visual language in UE. In figure 4.1, on the right, we can see the parameters that the default material's

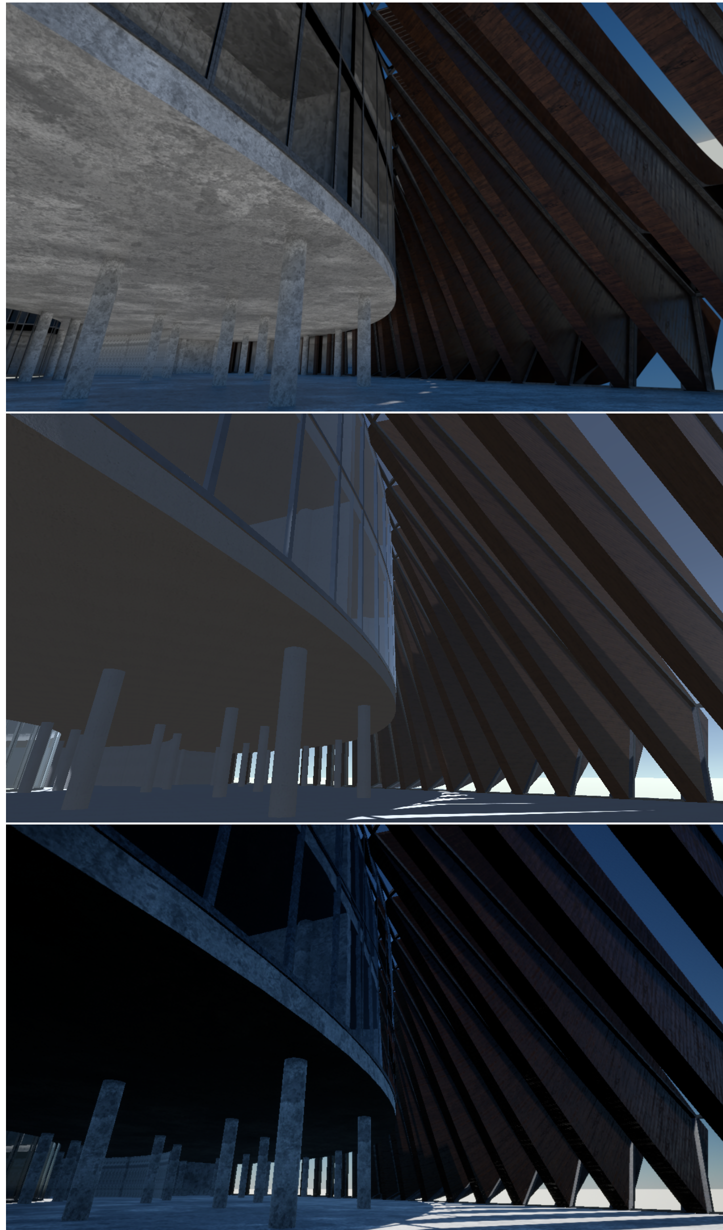


Figure 4.3: Render showing material quality in Octane, Unity and UE.

shader in UE can receive and how the parameters are similar in UE and Unity. However, with the use of Blueprints, we can create virtual textures that result from operations between textures which allow more complex materials, like we did in section 3.3.1. In figure 4.2, we can see the shader used for bricks, a material. In this shader, the diffuse texture results from operations between four textures which cannot easily be done with the shaders available in Unity.

Unity's shaders also do not have a parameter for the roughness texture that adds extra detail. In figure 4.4, we can see that the bricks in Unity appear like a smooth texture, contrary to UE. Additionally,

the bricks look smaller in Unity, even though UE and Unity are using the same textures. This happens because UE can scale textures with Blueprints while Unity cannot. However, this can be solved by creating a shader in Unity that can take in to consideration roughness textures and scaled textures.

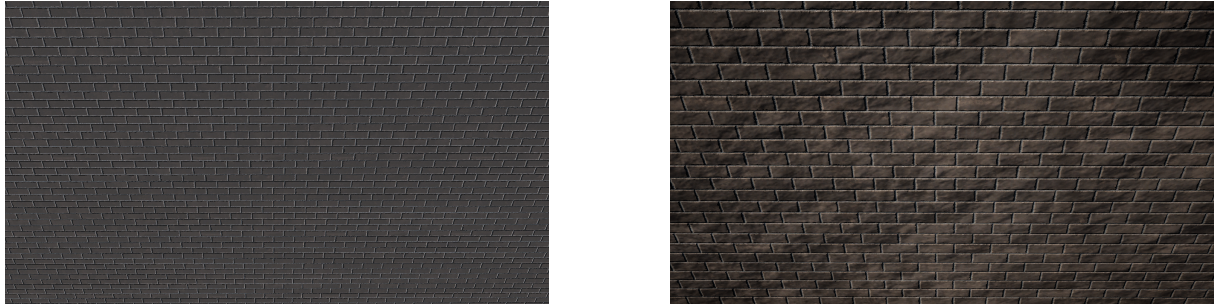
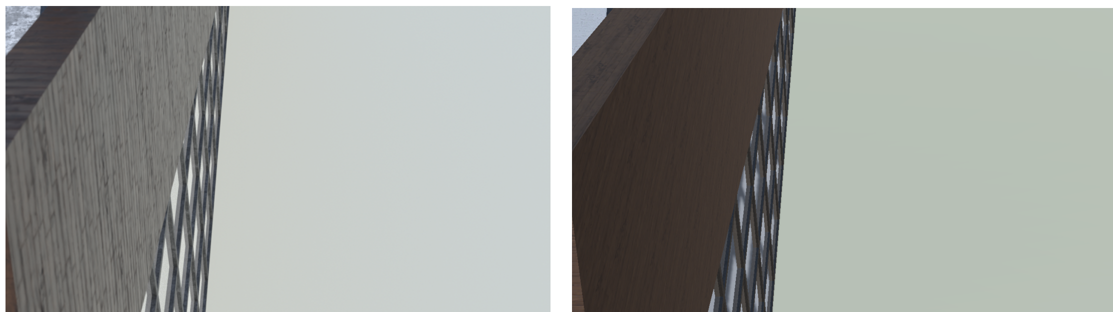


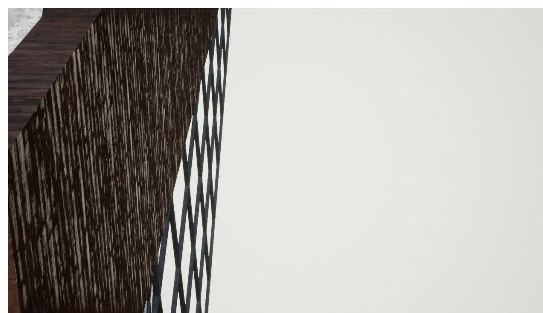
Figure 4.4: Material using the same textures rendered in Unity and UE

4.1.2 Lights



(a) V-Ray

(b) Unity



(c) Unreal Engine

Figure 4.5: Render in V-Ray, Unity and UE showcasing reflection capabilities in each visualizer.

Both UE and Unity are using directional lights and an object that simulates light interactions with the atmosphere. In figure 4.5, we can see the capability to do reflections done by UE and Unity. To

observe reflections, it is necessary to have a sufficient angle of reflection for the material. We can obtain this information by using renders done with offline renderers, because offline renderers can use more complex techniques like path tracing. In figure 4.5, we can see that only UE can create reflections. This happens because UE uses screen-space reflections and the reflected object is on the screen. Unity cannot use screen-space reflections with the standard render pipeline. It requires the high-definition render pipeline which is not supported by the approach that was used. In figure 4.6, UE renders do not have reflections because the ground is not on the screen.

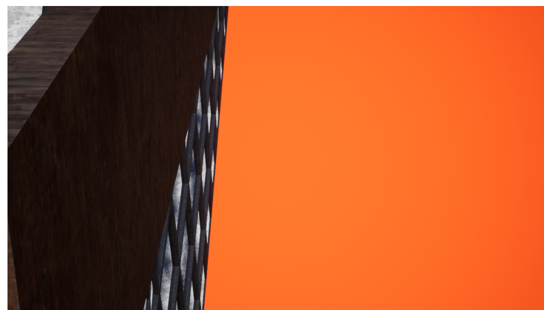


Figure 4.6: Reflection in UE when we cover the screen with an orange block.

Game engines cannot use complex techniques used by offline renderers, creating fidelity issues. We created example scenes in UE and rendered them with V-Ray and UE. In figure 4.7, we can see that the inner reflections on the chairs and light diffraction on the glass ornament do not exist in UE.

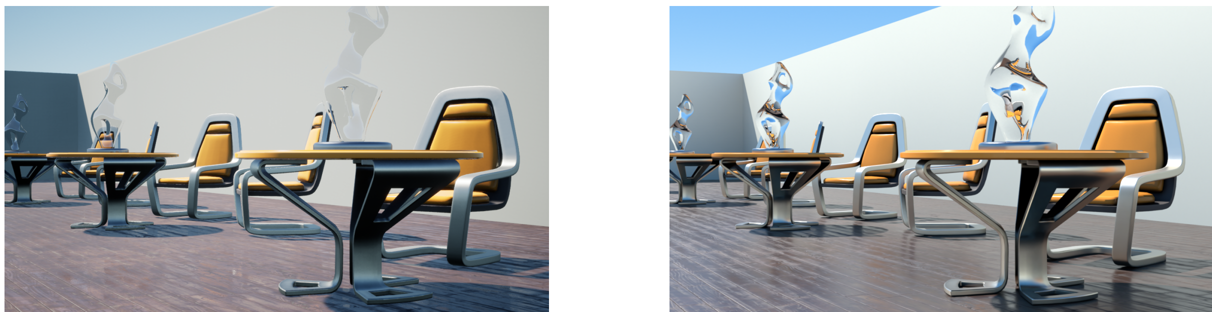


Figure 4.7: Scene showcasing Unreal Engine light reflection and diffraction limitations. On the right, we have the scene rendered in Unreal Engine and on the left we have the scene rendered in V-Ray.

Global illumination is also impossible to simulate in game engines without using lightmaps. In scenarios where we have a non-directional light in the scene, having a poor or non-existent global illumination can create low-fidelity results, as we can see in the figure 4.8. In the figure, we used a source of light to illuminate a room through a pink glass. In the UE render, the room is completely dark due to lack of global illumination. V-Ray simulates global illumination and lights the entire room up.

Some of these limitations could have less impact if we used ray tracing, but due to lack of hardware support, we were unable create renders using this technique.

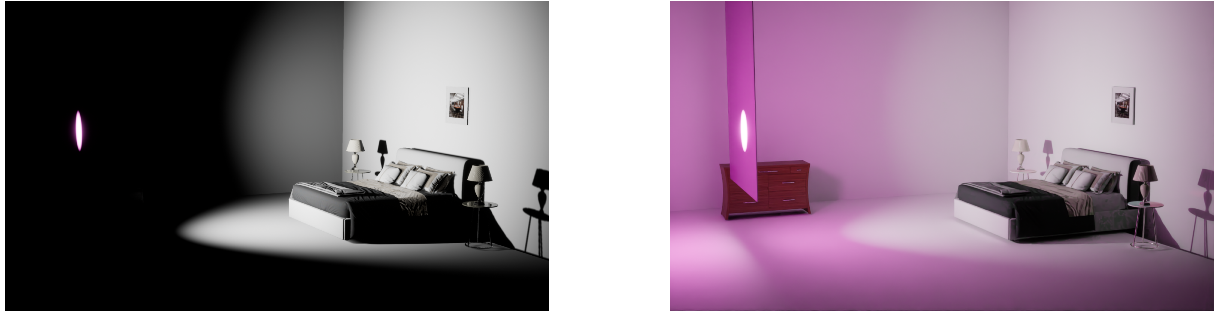


Figure 4.8: Scene showcasing UE global illumination limitations. On the right, we have the scene rendered in UE and on the left we have the scene rendered in V-Ray

4.2 Performance

One of our objectives in this research was to adapt a game engine to support AD. In AD, it is important for a visualization tool to be fast to allow an architect to quickly view the digital model. For this reason, we will measure the performance of our solution and compare other visualization tools. We measured the following qualities: the time it takes to generate a digital model, the time it takes to generate a cinematic, and interactivity with the application.

4.2.1 Generate Model

We compared the time it takes to generate our solution with Unity, another tool that is meant to be used in the same stages of AD project development. We generated the digital model of the Isenberg Business Innovation Hub, the digital model of the Astana National Library, and a model that can be increased exponentially. Both the Isenberg Business Innovation Hub and the Astana National Library are projects modeled using an AD approach, and they are composed by two thousand and twenty eight thousand primitive elements, respectively.

Model	Unreal Engine	Unity
Isenberg Business Innovation Hub	00:02:01.943	00:00:02.843
Astana National Library	01:12:58.196	00:00:29.305
Exponential model (n=1)	00:00:03.511	00:00:00.001
Exponential model (n=3)	00:00:26.634	00:00:00.114
Exponential model (n=4)	00:02:02.486	00:00:00.397

Table 4.1: Time Unity and UE take to generate the digital model of Isenberg Innovation Hub, Astana National Library, and a model that can be increased exponentially

In table 4.1, we can see that Unity can generate the model much faster than UE. One of reasons this happens is because the approach using Unity does not generate geometry, contrary to our solution. The last model is only composed of cylinders and cubes that are of the same size. This means our solution will use the cache a lot and we can decrease the impact of generating similar geometry multiple

times unnecessarily. However, as we can see in table 4.1, Unity is still much faster when inserting game objects than UE is when inserting actors. This is one of the biggest limitations of UE and it will always make the model generation slower in UE.

4.2.2 Generate Cinematics

We generated cinematic of the Astana National Library's digital model in UE and Unity. In UE, we used two different methods to render cinematics. In the first method, UE will generates a frame every time Khepri sends a request. We called it UE Khepri method. In the seconds method, UE stores the

Cinemeatic	Unity	Unreal Engine with Khepri method	Unreal Engine with our method
Tracking line	00:00:16.858	00:04:43.992	00:00:13.996
Tracking patio entrance	00:00:18.016	00:04:20.474	00:00:08.052
Enter Patio	00:01:11.020	00:17:28.633	00:00:32.107
Library	00:01:22.208	00:20:23.421	00:00:31.833
Exterior	00:04:06.122	01:04:05.893	00:02:25.216
Panning Patio	00:00:31.651	00:08:36.950	00:00:20.900

Table 4.2: Time it take to generate cinematics in Unity and UE while using Khepri method and our method

camera position every time Khepri send a request and then we generate the entire cinematic. In this last method, we will add the time it takes to generate the cinematic and the time it takes for Khepri to send all of the camera positions. We generated multiple cinematics due to the optimization techniques present in game engines. We did so knowing different performance values will be obtained depending on the scene viewpoint. Therefore, pre-defined pathways in various places of the scene will be followed to form a render sequence. In table 4.2, we can see that UE, when using the first method, is much slower than Unity, but when we used the second method, we could generate cinematics much faster.

4.2.3 Interactivity

In AD, projects can have a large amount of geometry which can hinder interactivity in some visualization tools. Game engines are the solution to this problem because they use a set of techniques to adapt the model for real-time visualization, unlike typical CAD and BIM applications. For this reason, we decided to evaluate how our solution reacts to the generation of complex models with a lot of geometry. We will compare our solution with Unity.

Unity and UE provide two views: a view for scene editing, where the user can change the geometry's properties and add new objects in the scene, but only allows navigation in free mode; and another view, called game view, that does not allow changes but allows using different navigation systems. We generated the Astana National Library model in UE and Unity and placed the view at different viewpoints.

Viewpoint	**Unreal Editor/Game Mode**	**Unity Game Mode**	**Unity Editor Mode**
Profile view	16.34 $\pm$ 1.28 fps	18.76 $\pm$ 1.74 fps	7.00 $\pm$ 0.80 fps
Top view	14.22 $\pm$ 1.10 fps	16.44 $\pm$ 1.34 fps	5.87 $\pm$ 0.61 fps
Inner to Patio	32.89 $\pm$ 2.45 fps	23.26 $\pm$ 2.09 fps	10.18 $\pm$ 1.13 fps
Library	99.00 $\pm$ 5.77 fps	65.79 $\pm$ 4.99 fps	21.69 $\pm$ 2.08 fps
Inner view	81.30 $\pm$ 5.48 fps	56.17 $\pm$ 3.28 fps	18.11 $\pm$ 1.94 fps
Stairs	59.52 $\pm$ 3.79 fps	53.19 $\pm$ 3.56 fps	16.03 $\pm$ 1.33 fps

Table 4.3: The average number of frames per second that UE and Unity have when using game view and editor view in different viewpoints

In table 4.3, we can see the results each view has at different viewpoints. UE shows the same level of performance in the editor view and in the game view, contrary to Unity, where the editor view has major performance issues when compared to game mode. However, UE and Unity show similar results while using game view. At viewpoints where a larger amount of geometry has to be rendered, like the profile view and the top view, Unity has better performance. Meanwhile, at viewpoints where a smaller amount of geometry is rendered, UE has better performance.

4.3 Summary

In this chapter, we observed that our solution can generate higher fidelity renders than Unity, because UE can use more complex materials. Our approach generates geometry with correct UV which allows us to use a larger amount of shaders. However, the decision of generating more faithful geometry comes with the cost of slower model generation.

Khepri's approach to generating cinematics also showed to be restrictive because it made UE, our solution, much slower. When we used our approach, where UE generates the entire cinematic in one go, we can generate cinematics faster, even when compared with Unity. UE also allows more creativity in cinematics by allowing the user to animate objects in the scene.

In terms of interactivity, both UE and Unity show similar results when generating complex models which makes both applications good candidates for visualization of AD projects. However, Unity is only good when using the game view which hinders Unity as a visualization tool because only in the editor view the user can edit scenes and use the game engines' tools to add more details.

With these results in mind, we can conclude that UE is the preferred tool when a user wants to have a high-fidelity view, or wants to generate cinematics. However, Unity seems more useful for AD projects in earlier stages of development where the architect is still testing different designs and needs to regenerate the digital model multiple times. The time it takes to generate geometry in UE is a big disadvantage and might be hinder the architect's productivity. Our solution, like all AD visualization tools, has compromises, but by using an AD approach, the architects can select the most suitable visualization

tool for their needs.

5

Conclusion

Contents

5.1 Future Work 54

Designing complex buildings requires the architect to use several tools in order to accomplish various tasks, such as 3D modeling, analysis evaluation, and rendering. The usage of all these tools leads to a tiresome and error-prone process. Algorithmic Design (AD) presents itself as the solution by automating this process.

AD is becoming a more common practice in architecture. Architects using this approach can quickly change a model with no effort and generate alternatives in a short amount of time. Unlike traditional architecture design approaches, with AD, the architect does not create the digital building model directly, but instead writes an algorithm that generates the digital model. This allows the architects to automate repetitive and time consuming tasks, and create complex geometry that would be extremely hard to create by traditional means.

The process of developing the algorithm is not a trivial task. For this reason, visualization tools are used to reduce this program-design disconnection and aid the design process in several ways. Unfortunately, typical visualization tools cannot easily handle the amount of geometry generated by AD programs or provide high-fidelity real-time rendering. High-fidelity views of the digital model are essential for architects when they want to make design decisions or share their vision but many AD only provide a simplified view of the digital model. If architects want higher fidelity renders, they must resort to Computer-aided Design (CAD) or Building Information Modeling (BIM) tools. However, these renders can take large amounts of time to finish. Additionally, CAD and BIM tools do not perform well with the large amounts geometry generated by the AD approach, which also hinders the architect's productivity. Unlike the previous tools, game engines are designed to handle large amounts of geometry and generate renders in real time, which makes them excellent candidates for AD visualization tools.

For this reason, the Unity game engine was previously adapted for AD. This tool allows architects to generate digital models and cinematics quickly while also allowing the generated model to be viewed in real time. Unfortunately, Unity is not able to achieve high-quality renders due to limitations in its materials shaders. Unreal Engine (UE) is another game engine that provides a more flexible material system capable of generating close to photo-realistic renders in real time.

We propose using UE as an AD visualization tool. Our solution is composed of: (1) a RPC communication channel between UE and an AD tool; (2) a plugin that translates requests from the AD tool to UE, in order to generate the digital model; and (3) a set of navigation systems that allow the user to view the digital model from different perspectives. We also explored how materials work and we extended cinematic creation tools that exist in UE for AD.

To evaluate our solution, we compared the image fidelity that it can generate with another real-time visualizer and an offline renderer. We also evaluated the performance of our solution by measuring: the time it takes to generate models, the time it takes to generate cinematics and the interactivity level of our solution. Then, we compared our measurements with measurements in another visualizer. Results

show that UE can use more complex materials than Unity, it can generate cinematics faster than Unity, and it also provides higher level interactivity than Unity. However, UE is slower to generate the digital model than Unity, the time it takes to generate the in UE can be more than hundred time larger.

In summary, our solution is able to explore the use of a game engine capable of high-fidelity rendering in real time for AD. We achieve this by extending UE with the use a plugin and a communication channel. Our solution shows more flexibility that previous solutions, such as Unity, with regards to how materials are implemented, leading to higher image fidelity. However, our solution is not optimal for quick view of the model due to the time it requires for the model to be generated. Our solution, like all AD visualization tools, has compromises, but by using an AD approach, the architects can select the most suitable visualization tool for their needs.

5.1 Future Work

UE is constantly evolving and newer versions are being released every few months. These newer versions come with new features that can be interesting to explore, in the context of AD. For example, in a future version of UE, there will be a feature that allows the user to create water bodies through splines, which are already supported in Khepri. Additionally, newer versions might deprecate the current application programming interface, so our plugin might need maintenance to work in newer versions.

We will also continue to explore newer ways to optimize the model generation and actor creation in UE, since quick model generation is an important quality in an AD visualization tool. This is currently one important limitation of UE and greatly reduces the productivity. One possibility would be to explore procedural mesh actors. These actors can generate geometry in run time, which might reduce the time it takes to generate the model, but can greatly impact the performance of UE.

Currently, Khepri is not prepared to animate the model, but this would be an interesting extension to add in Khepri. This feature would allow a user to automate animations through the algorithmic description as well and generate more complex cinematics.

Finally, we will also explore UE's ray tracing capabilities. These techniques will allow the generation of higher fidelity renders in real time, but since we currently do not have compatible hardware, we cannot test this technology.

Bibliography

[1] B. Kolarevic, *Architecture in the Digital Age: Design and Manufacturing*, 1st ed. Taylor Francis, 2003.

[2] K. Kensek and D. Noble, *Building Information Modeling: BIM in Current and Future Practice*, 1st ed. Wiley, 2014.

[3] D. J. Gerber and M. Ibañez, *Paradigms in Computing: Making, Machines, and Models for Design Agency in Architecture*, D. J. Gerber and M. Ibañez, Eds. eVolo, 2015.

[4] R. Woodbury, *Elements of parametric design*, 1st ed. Routledge, 2010, vol. 1.

[5] Mark Burry, *Scripting Cultures: Architectural design and programming*, M. Burry, Ed. Hoboken, NJ, USA: John Wiley Sons, Inc., jan 2013.

[6] R. Castelo Branco and A. Leitão, "Integrated algorithmic design: A single-script approach for multiple design tasks," *eCAADe 35 - Design Tools - Theory*, vol. 1, no. September, pp. 729–738, 2017.

[7] A. Leitão, R. Castelo-Branco, and G. Santos, "Game of Renders," *Intelligent and Informed - Proceedings of the 24th International Conference on Computer-Aided Architectural Design Research in Asia, CAADRIA 2019*, vol. 1, pp. 655–664, 2019.

[8] A. W. Pelosi, "Obstacles of utilising real-time 3D visualisation in architectural representations and documentation," *New Frontiers - Proceedings of the 15th International Conference on Computer-Aided Architectural Design in Asia, CAADRIA 2010*, pp. 391–398, 2010.

[9] Carlos Martinho, P. Santos, and R. Prada, *Design e Desenvolvimento de Jogos*, C. Martinho, P. Santos, and R. Prada, Eds. FCA, 2013.

[10] B. Karis, "Real Shading in Unreal Engine 4," *Acm Siggraph 2013*, pp. 1–21, 2013.

[11] T. Akenine-Möller, E. Haines, N. Hoffman, A. Pesce, M. Iwanicki, and S. Hillaire, *Real-Time Rendering, Fourth Edition*, 4th ed. CRC Press, 2018.

[12] J. A. Ferwerda, “Three varieties of realism in computer graphics,” in *Human Vision and Electronic Imaging VIII*, B. E. Rogowitz and T. N. Pappas, Eds., vol. 5007. SPIE, jun 2003, p. 290.

[13] P. Alfaiate and A. Leitão, *eCAADe 2017*, vol. 2, pp. 511–518, 2014.

[14] M. Z. Khan and M. M. Hashem, “A Comparison between HTML5 and OpenGL in Rendering Fractal,” in *2nd International Conference on Electrical, Computer and Communication Engineering, ECCE 2019*. Institute of Electrical and Electronics Engineers Inc., apr 2019.

[15] M. Claypool and K. Claypool, “Perspectives, frame rates and resolutions: It's all in the game,” in *FDG 2009 - 4th International Conference on the Foundations of Digital Games, Proceedings*. New York, New York, USA: ACM Press, 2009, pp. 42–49.

[16] A. S. Augsburg, “Realtime Interactive Architectural Visualization using Unreal Engine 3 . 5 Masterarbeit Realtime Interactive Architectural Visualization using Unreal Engine 3 . 5,” no. March 2013, 2016.

[17] “A Forrester Consulting Thought Leadership Spotlight Commissioned By Epic Games Real-Time Rendering Solutions: Unlocking The Power Of Now,” Tech. Rep., 2018.

[18] I. Sadeghi, H. Pritchett, H. W. Jensen, and R. Tamstorf, “An artist friendly hair shading system,” *ACM SIGGRAPH 2010 Papers, SIGGRAPH 2010*, vol. 29, no. 4, pp. 1–10, 2010.

[19] B. Burley, “Physically Based Shading at Pixar,” *Acm Siggraph*, pp. 1–27, 2012.

[20] E. Haines and T. Akenine-Möller, *Ray tracing gems: High-quality and real-time rendering with DXR and other APIs*, 1st ed., E. Haines and T. Akenine-Möller, Eds. Apress Media LLC, feb 2019.

[21] A. Šmíd, “Comparison of Unity and Unreal Engine,” no. May, p. 69, 2017.

[22] L. Liang, C. Liu, Y. Q. Xu, B. Guo, and H. Y. Shum, “Real-time texture synthesis by patch-based sampling,” *ACM Transactions on Graphics*, vol. 20, no. 3, pp. 127–150, jul 2001.

[23] P. Bhat, S. Ingram, and G. Turk, “Geometric texture synthesis by example,” in *ACM International Conference Proceeding Series*, vol. 71. New York, New York, USA: ACM Press, 2004, pp. 41–44.

[24] T. Deliot and E. Heitz, “Procedural Stochastic Textures by Tiling and Blending,” *GPU Zen 2*, 2019.