Game Engines for Algorithmic Design

Ricardo de Lemos Filipe
ricardo.l.filipe@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

January 2021

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

1. Introduction

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 [14], increasing productivity, the quality of presentation images, and the production of technical documentation. Furthermore, 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 2D drawing, 3D modeling, analysis, optimization, and rendering. The need to use various tools and make 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 [8]. Unlike traditional architectural design approaches, with AD , the architects do not create the digital model directly. Instead, they write the program that generates the digital model, through a combination of geometric, mathematical, and symbolic representations [20]. This allows the architect to be able to create more complex geometry, automate repetitive tasks, and explore new alternative designs with low effort.

However, creating such a program is not a trivial task. Coding complex designs demands an additional effort from the architect, who might not be very proficient at programming. This leads not only to additional errors, such as coding mistakes along with geometric 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 aesthetics evaluation on them. 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 [10].

This discrepancy between code and model 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 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 may 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 [15].

1.1. Goals

Our main goal with this dissertation is to overcome the problems caused by the use of AD alongside a CAD or BIM application, as the models generated by these applications, most of the time, are not suitable for navigation or visualization because they provide a restrictive real-time visualization and perform badly with complex models [17]. These applications were designed for interactive use ; however, they often become impracticable 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 high-fidelity real-time visualization tool for AD. Our goal in this dissertation is to use a game engine to create high-fidelity renders rapidly and also serve as navigation tool.

2. Related Work

In this section, we will explore existing visualization solutions for AD and describe advantages and disadvantages of each solution. We will also analyse the techniques used by game engines to achieve high fidelity rendering.

2.1. AD Visualization Tools

Although the area of visualization is broad, this research focus on the visualizers aimed for architectural design and also those related to AD. As mentioned in the goals section, we want to achieve high-fidelity visualization, interactivity, and real-time rendering and, as such, we will focus more on the tool's capabilities in these areas. All the tools mentioned in the following subsections are capable of receiving a set of modeling operations describing a model and generate it.

2.1.1 CAD and BIM applications

There are two paradigms for producing and visualising 3D models in architecture: CAD and BIM. Both allow the creation and modification of a design by means of a computer, and their goal is to increase design productivity and quality. With these tools, an architect can freely create, explore and visualize their designs either in 2D or 3D space.

What makes BIM applications stand out from CAD applications is the fact that 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 [12]. The BIM paradigm uses data-enhanced parametric objects that allow data and structural information to be stored within the model. ArchiCAD and Revit are two examples of BIM applications. Rhino3D and SketchUp are two examples of CAD applications.

CAD and BIM applications are capable of rendering visually appealing rendered results but they have their limitations. When used for direct modeling, 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 [10]. Natively, both CAD and BIM provide two main views, one with a simplified view of the model, 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.

Real-time rendering with high fidelity is possible with Twinmotion and Lumion. Twinmotion is an Unreal visualizer capable of doing real-time rendering with high fidelity, and supports the three navigation system that are described in section 1, as well as synchronising 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 render for CAD and BIM applications. It includes an interactive interface, a weather systems, and a resourceful library of Physical-Based Rendering (PBR) materials and assets. Lumion does not support Virtual Reality (VR) navigation or walk mode, it only supports free camera. Since these visualizers are mainly focused for high fidelity real-time renders, they might not scale well with large projects.

2.1.2 Luna Moth

Luna Moth is a useful web-based AD 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 which reduces significantly the render quality.

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 [3], the latter can render frames faster than web applications [13]. Because interactivity is so related to performance in real-time rendering, as low frame rates ruin user experience [7], we can conclude that Luna Moth is not a sufficiently good visualization tool for more complex models.

2.1.3 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.

2.1.4 Game Engines

In architecture, game engines have been considered for visualization [4, 1]. Unlike the previous applications, game engines are optimized to generate renders in real time. Game engines employ different techniques to simulate reality [6]. 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 a better performance while maintaining the illusion of realism. By using these techniques, game engines can perform much better in environments with large amounts of geometry than the previous visualization applications.

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 [11, 2]. 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.

Most popular game engines nowadays are Unity [2] and Unreal Engine (UE) [3]. These engines carry all the features that make up a good visualizer, such as: (1) they have an active community, composed by developers and users that constantly improve the tool; (2) they are updated on a regular basis, to augment the game engine tool with the latest algorithms, such as ray tracing [9]; (3) they have high quality real-time visualizers, as they support PBR materials, lighting, shadowing and many other effects; (4) there is a multitude of assets, present either on each respective asset store or user-imported; (5) they include a physics engine to allow interactions that obey the laws of physics;

This means that game engines capabilities satisfy our needs of using an application capable of rendering in real-time with high fidelity and also perform well with large amounts of geometry. Therefore, they are a good candidate for our solution.

In the following subsections, we will explore the different game engines techniques used to create high-fidelity results.

Physical-Based Rendering

The first technique we are going to talk about is PBR [11, 2]. Traditionally, light interactions were done through shading models and punctual lighting. Even though such a process easily implements PBR, it still did not take into consideration real life physics and provided poorer results. As such, PBR was further developed with the main objective of simulating light by doing approximations of the Bidirectional Reflectance Distribution Function (BRDF). BRDF [2] is a function that describes how light interacts with opaque objects. One problem with PBR was the complexity for the artists

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

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

[3] https://docs.unrealengine.com/en-US/index.html. Last accessed 23 Dec 2019

to use it, but Disney [18, 5] developed a physics-based shading 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, owner of the Unreal Engine, although their approach had some differences, particularly regarding the procedure to achieve real-time performance [11].

In PBR, 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.

Materials are an essential part of PBR because they define the physical properties of the object. In game engines, these properties are base color, metallic, roughness, and cavity; all these properties are described using textures.

Global Illumination

Global illumination is a set of algorithms that simulate the light coming from reflection, diffusion or refraction. In games engines, global illumination is achieved through lightmaps, which are textures containing the effects of casting light sources onto static objects. However, lightmaps can only be calculated when light sources are stationary.

Reflection

Unfortunately, game engines can not create reflection using ligthmaps. 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. 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 into 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.

3. Methodology

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.

UE stands out from other game engines, because it uses a more advanced techniques to create materials and to calculate light interactions than other game engines[19], like Unity. Additionally, UE has already features designed for architecture visualization and UE is an open source solution, making the process of adapting such tools to architect's needs simpler.

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

3.1. Architecture

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 describe 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.

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 1, we can see the architecture of our solution.

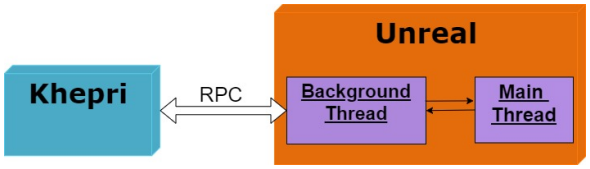

Figure 1: Component-and-Connector view of our solution

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. The communication

channel is only the first step in allowing communication between UE and Khepri. We also need to make UE Remote Procedural Call (RPC) server, which requires developing a set of functions that can 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. This new thread makes it possible to pause and resume generating the digital model without closing the communication channel.

In our solution, the background thread does not directly forward the request to the main thread. Instead, it creates an object called operation, that represents this request. This allow 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. This would be impossible if the background thread just forwarded the request to the main thread, because the background could not distinguish the requests and store the state of the UE.

3.2. Model Generation

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 requests.

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 Brush

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 do 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 2.

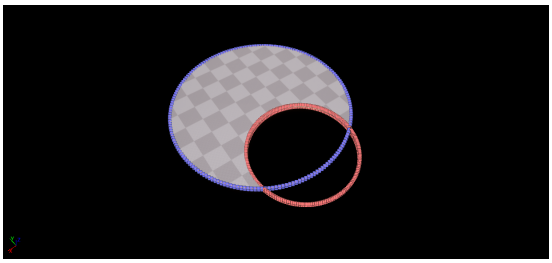

Figure 2: Creating a slab with brushes.

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.

While using our brushes, we found a problem. Brushes cannot generate correctly non-convex surfaces. This is a big problem, because

our main objective is to create a realistic representation of the digital model and the incorrect generation creates meshes with poor texture mapping in non-convex faces. The reason why this happens is because brushes use Binary Space Partitioning (BSP) 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 into a three dimensional object. This process is called UV mapping, where U and V are the 2D axes. 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 UV mapping while generating geometry.

3.2.2 Primitive Method

r 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.

This more primitive approach solves the issue created by non-convex surfaces, because now we can map the mesh correctly. This new approach also shows to be faster than 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, we can see a cylinder created by each method.

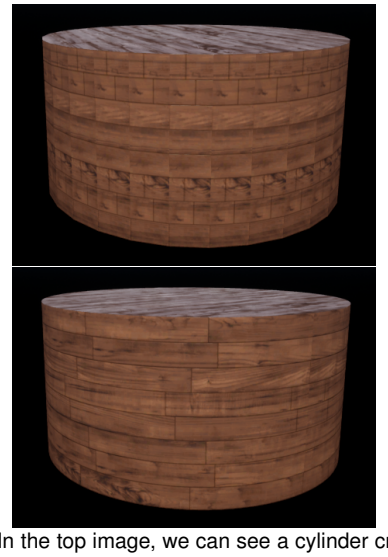

Figure 3: In the top image, we can see a cylinder created with brushes. In the bottom image, we can view a cylinder generated with the primitive method

3.2.3 Optimizations

Both previous methods have to generate meshes. This means that the same mesh can be created multiple times, this process has significant cost in both methods.

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 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 Ar-

chitects). The results can be found in the table 1. The optimization showed a significant gain in performance.

	Without Cache	With Cache
Brushes	552.6	263,4
Primitive	323.4	121.9

Table 1: The time, in seconds, it takes to generate the Isenberg Innovation Hub's model when using different methods.

3.3. Generate Renders

One of our main objectives is to allow architects to create renders. As such, it is important to explore what UE can achieve and allow the user 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, aperture, and focus distance. 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 section, we evaluate our solution in terms of image quality and performance when compared with other visualizers.

4. Evaluation

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 section 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[15]. 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 dig-

ital 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 depends on the shaders' inputs. In the approach used in Unity, there is a limitation on what material shaders can be used. 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. 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.

In UE, materials have a special property that allow to extend shaders through blueprints, a visual language in UE. With the use of blueprints we can create virtual textures that result from operations between textures, which allow to create more complex materials. For example we can do texture synthesis which is a process of algorithmically generating a bigger texture image from a smaller digital sample [16].

Unity uses a simpler approach to materials that does not support virtual texture as input for shaders which makes it harder or impossible to accomplish complex materials.

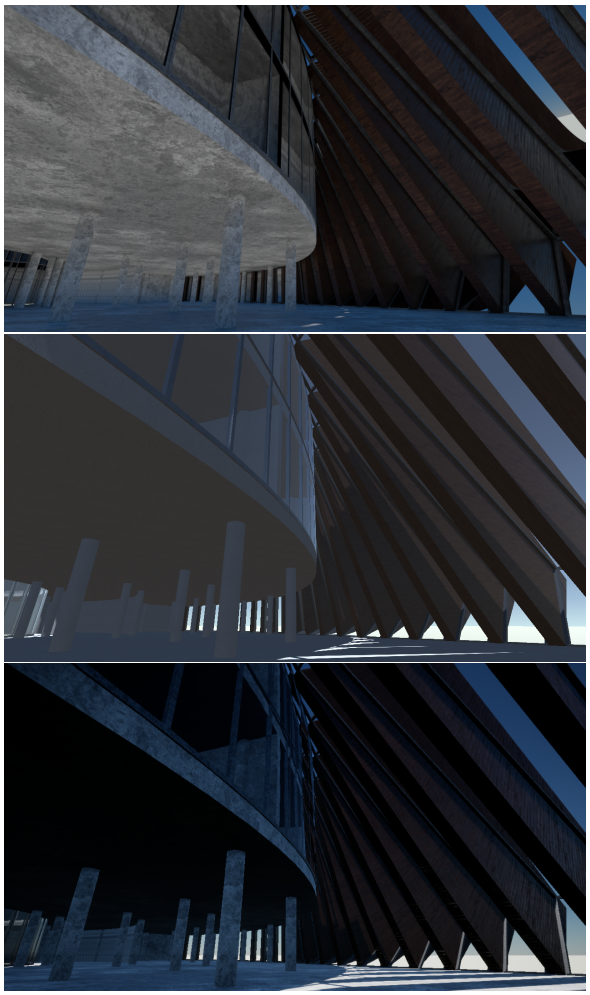

Figure 4: Render showing material quality in Octane, Unity and UE.

Figure 5: Scene showcasing Unreal Engine global illumination limitations. On the right, we have the scene rendered in Unreal Engine and on the left we have the scene rendered in V-Ray

4.1.2 Light

Both Unreal and Unity have limitations with reflections and diffraction when compared with offline rendering. However, Unreal can natively do screen-space reflection while Unity requires the usage of a high-end render pipeline.

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 5. 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 support from hardware, we were unable create renders using this technique.

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 applica-

tion.

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 BIH	00:02:01.9434	00:00:02.8436
Astana National Library	01:12:58.1964	00:00:29.3057
Exponential model (n=3)	00:00:00.1146	00:00:26.6347
Exponential model (n=4)	00:00:00.3971	00:02:02.4862

Table 2: Time Unity and Unreal take to generate the digital model of Isenberg Business Innovation Hub (BIH), Astana National Library, and a model that can be increased exponentially

In table 2, 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 2, 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 Astana National Library digital model in UE and Unity. In UE, we used our method to render cinematics. In our method, UE stores the 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 3, we can see that UE when it uses our method is much faster than Unity.

Cinemeatic	Unity	Unreal Engine with our method
Tracking line	00:00:16.85	00:00:13.99
Patio entrance	00:00:18.01	00:00:08.05
Enter Patio	00:01:11.02	00:00:32.10
Library	00:01:22.20	00:00:31.83
Exterior	00:04:06.12	00:02:25.21

Table 3: Time it take to generate cinematics in Unity and Unreal while using Khepri method and our method

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.

Viewpoint	Unreal	Unity Game View	Unity Editor View
Profile view	16.34 fps	18.76 fps	7.00 fps
Top view	14.22 fps	16.44 fps	5.87 fps
Inner to Patio	32.89 fps	23.26 fps	10.18 fps
Library	99.00 fps	65.79 fps	21.69 fps
Inner view	81.30 fps	56.17 fps	18.11 fps
Stairs	59.52 fps	53.19 fps	16.03 fps

Table 4: The average number of frames per second that Unreal and Unity have when using game view and editor view in different viewpoints

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. In table 4, 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. Discussion

In this section, 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.

When we used our approach where UE generates the entire cinematic in one go, we can generate cinematics faster than 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 visualizers, has compromises, but by using an AD approach, the architects can select the most suitable visualization tool to their needs.

5. Conclusions

In architecture, Algorithm Design (AD) is becoming more commonly used. In this approach, the architect describes the model through algorithms. By using AD, architects use visualization applications to help them relate parts of the algorithms with the generated geometry. Furthermore, throughout the AD project development, the architect needs a high-fidelity view to make design aesthetics decisions.

Unfortunately, commonly used modeling applications, such as Computer Aided-Design and Building Information Modeling, cannot fulfill architect's needs concerning the visualization of complex AD projects. These applications have performance issues when saturated in geometry and take a large amount of time to generate high fidelity renders.

For this reason, we explore using a game engines, the Unreal Engine (UE), as a high-fidelity real-time visualization tool for AD. Game engines use techniques that accelerate the rendering process which allows a higher level of interaction than other visualization applications and high-fidelity renders in real time. 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.

References

[1] A Forrester Consulting Thought Leadership Spotlight Commissioned By Epic Games Real-Time Rendering Solutions: Unlocking The Power Of Now. Technical report, 2018.

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

[3] P. Alfaiate and A. Leitão. Luna Moth: a web-based programming Environment for Generative Design. *eCAADe 2017*, 2:511–518, 2014.

[4] A. S. Augsburg. Realtime Interactive Architectural Visualization using Unreal Engine 3 . 5 Masterarbeit Realtime Interactive Architectural Visualization using Unreal Engine 3 . 5. (March 2013), 2016.

[5] B. Burley. Physically Based Shading at Pixar. *Acm Siggraph*, pages 1–27, 2012.

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

[7] 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*, pages 42–49, New York, New York, USA, 2009. ACM Press.

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

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

[10] M. Johansson. Real-time rendering of large building information models: Current state vs. state-of-the-art. 2012.

[11] B. Karis. Real Shading in Unreal Engine 4. *Acm Siggraph 2013*, pages 1–21, 2013.

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

[13] 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.

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

[15] 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*, 1:655–664, 2019.

[16] 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*, 20(3):127–150, jul 2001.

[17] 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*, pages 391–398, 2010.

[18] I. Sadeghi, H. Pritchett, H. W. Jensen, and R. Tamstorf. An artist friendly hair shading system. *ACM SIGGRAPH 2010 Papers, SIGGRAPH 2010*, 29(4):1–10, 2010.

[19] A. Šmíd. Comparison of Unity and Unreal Engine. (May):69, 2017.

[20] R. Woodbury. *Elements of parametric design*, volume 1. Routledge, 1st editio edition, 2010.