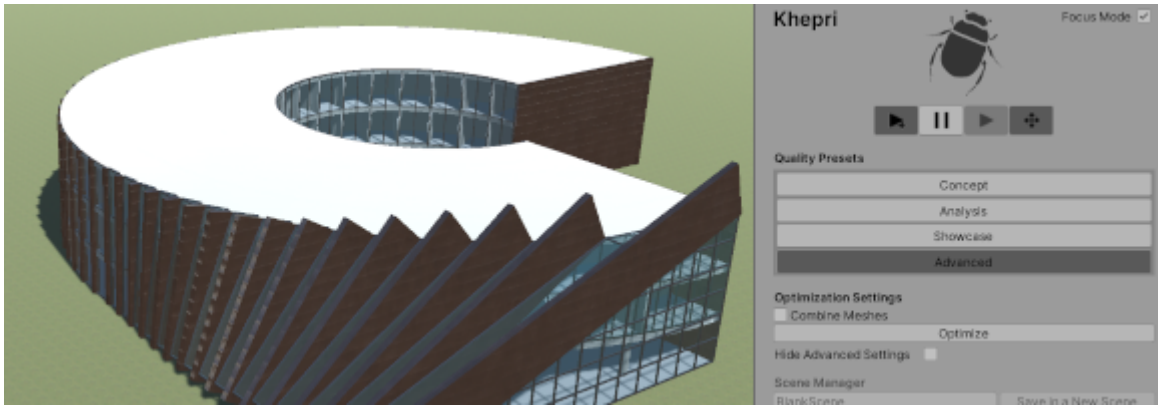
TÉCNICO LISBOA



Fast Interactive Visualization for Algorithmic Design

Guilherme Jorge dos Santos

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Dr. António Paulo Teles de Menezes Correia Leitão

Examination Committee

Chairperson: Prof. Dr. David Manuel Martins de Matos
Supervisor: Prof. Dr. António Paulo Teles de Menezes Correia Leitão
Member of the Committee: Prof. Dr. João António Madeiras Pereira

October 2020

Acknowledgments

I would like to thank my parents for the support they have given me throughout these university years. Without them, I would not have the money, food, or even clean clothes to survive all these years. Without them, I would not be at this point in life, writing such an important document.

I deeply appreciate the time and patience that my friends and members of the Algorithmic Design for Architecture research group sacrificed to help me during the development of this thesis and I apologize for making you read and review this long document.

I would also like to acknowledge my dissertation supervisor Prof. António Leitão for his insight, support, awesome moments of debugging my problems, and for sharing the knowledge that made possible the creation of this document.

Last but not least, I appreciate everybody in the Instituto Superior Técnico (IST) community that had to deal with my terrible handwriting when correcting my tests and exams. Without their effort, I would not be here writing this document.

To each and every one of you – Thank you.

Abstract

Nowadays, with the increasing use of Computer-Aided Design and Building Information Modeling applications in architecture, more complex digital architectural projects can be developed. This complexity, however, has been increasing to the point where the aforementioned applications no longer suffice. In order to better assist the demanding workflow of the architectural design process, Algorithmic Design, a programming approach to design, comes into play.

Nevertheless, Algorithmic Design constitutes a learning challenge for many practitioners. To overcome this issue, the industry relies on the introduction of immediate visual feedback into the programming workflow, allowing designers to quickly visualize the impact of changes made to their programs, considerably smoothing the learning curve. However, previous solutions for visualization do not offer the satisfactory performance required by the Algorithmic Design workflow. Hence, we propose to develop a fast interactive visualizer that can be used during the design process. To this end, we developed a visualizer based on a Game Engine. Game Engines are capable of handling complex scenes with high performance, and are thus suitable for the fast generation of the complex models that result from Algorithmic Design.

Moreover, we propose an integration with current state-of-the-art visualization technology, namely Virtual Reality. With it, we can immerse the architects in their design creations, enhancing decision-making and design communication even further.

Keywords

Algorithmic Design; 3D Graphics; Real-Time Visualization; Game Engine; Virtual Reality

Resumo

Hoje em dia, com o uso crescente de aplicações de Desenho Assistido por Computador e Modelação de Informação de Construção em arquitetura, é possível desenvolver projetos arquitetónicos mais complexos. No entanto, esta complexidade tem vindo a aumentar até ao ponto em que as aplicações anteriormente referidas são insuficientes. Contudo, existem abordagens que auxiliam as tarefas exigentes do processo de projeto arquitetónico, como é o caso do Design Algorítmico, uma abordagem programática para o design.

Infelizmente, o Design Algorítmico, apresenta uma difícil curva de aprendizagem para muitos profissionais. Para ultrapassar este problema, a introdução de uma solução de *feedback* visual imediato no fluxo de trabalho para visualizar rapidamente o impacto de mudanças feitas no código suaviza consideravelmente a curva de aprendizagem. No entanto, soluções anteriores de visualização não oferecem o desempenho exigido pelo fluxo de trabalho do Design Algorítmico. Para resolver este problema, propomos o uso de um visualizador rápido e interativo durante o processo de design. Para esse fim, desenvolvemos um visualizador baseado em motores de jogo, uma vez que estes são capazes de lidar com cenas complexas com um bom desempenho. Neste trabalho, propomos tomar proveito desses benefícios num contexto arquitetónico, mais especificamente para o Design Algorítmico.

Propomos também a integração desta solução com tecnologias de visualização recentes, nomeadamente a Realidade Virtual. Desta forma, os arquitetos poderão ficar imersos dentro dos seus modelos, melhorando a tomada de decisão e a apresentação dos seus designs.

Palavras Chave

Design Algorítmico; Gráficos 3D; Visualização em Tempo Real; Motor de Jogos; Realidade Virtual

Contents

1 Introduction **1**

1.1 Algorithmic Design . . . 3

1.2 Problem . . . 4

1.3 Goals . . . 6

2 Related Work **7**

2.1 Existing Visualizers . . . 9

2.1.1 CAD and BIM . . . 9

2.1.2 Luna Moth . . . 11

2.1.3 Twinmotion . . . 12

2.1.4 Lumion . . . 13

2.1.5 VIM AEC . . . 14

2.1.6 Unity Reflect . . . 14

2.1.7 Game Engines . . . 15

2.2 Acceleration Algorithms . . . 16

2.2.1 Rasterization or Ray-tracing . . . 16

2.2.2 Visibility Culling . . . 17

2.2.3 Level of Detail . . . 18

3 Proposed Solution **21**

3.1 Unity Backend . . . 23

3.2 Standard Features . . . 24

3.2.1 Operations . . . 25

3.2.2 Assets . . . 27

3.2.3 Navigation . . . 31

3.2.4 UI . . . 32

3.3 Advanced Features . . . 35

3.3.1 Day and Night System and Scene Illumination . . . 35

3.3.2 Visibility Culling and Level of Detail . . . 38

3.3.3 Additional Performance Acceleration . . . 41
3.3.4 Traceability . . . 43
3.3.5 Layers . . . 45
3.3.6 Scene Manager and Standalone Build . . . 47
3.3.7 Per Project Assets . . . 50
3.3.8 Virtual Reality . . . 51
3.3.9 Interactive Mode . . . 55

4 Evaluation **57**

4.1 Performance Benchmarks . . . 59
4.1.1 Occlusion Culling Benchmark . . . 64
4.1.2 LOD Benchmark . . . 65
4.1.3 Design Merge Benchmark . . . 68
4.2 Practicability Analysis . . . 69

5 Conclusion **73**

5.1 Limitations and Future Work . . . 76
5.2 Contributions . . . 77

List of Figures

1.1 Market Hall variations 4
1.2 Algorithmic Design workflow 5
1.3 Programming a design idea challenge 5

2.1 Rhinoceros interface 10
2.2 Luna Moth interface 12
2.3 Unity Reflect's AR mode 15
2.4 Z-buffer overdraw problem 17
2.5 Visibility culling techniques 18

3.1 Khepri architecture overview 23
3.2 Project structure overview 25
3.3 Client and server interaction 25
3.4 Boolean operations 27
3.5 Assets comparison 28
3.6 Material application challenge 29
3.7 Material application challenge solution 29
3.8 Unity backend user interface 32
3.9 Start Khepri communication interface 33
3.10 Quality and performance trade-off 34
3.11 Quality and performance trade-off interface 35
3.12 Configurations interface 36
3.13 Day and night interface 37
3.14 Illumination settings interface 37
3.15 Occlusion Culling settings interface 39
3.16 Level of detail settings interface 40
3.17 Level of detail example 41

3.18 Traceability flowchart . . . 44
3.19 Object outline example . . . 45
3.20 Object outline modes . . . 46
3.21 Layer coloring . . . 47
3.22 Edit mode generation . . . 49
3.23 Scene Manager interface . . . 49
3.24 Standalone Build interface . . . 50
3.25 Virtual Reality interface . . . 52
3.26 SteamVR input interface . . . 53
3.27 Object selection in Virtual Reality . . . 53
3.28 Live Coding in Virtual Reality example . . . 55
3.29 Interactive Mode interface . . . 56

4.1 Astana render . . . 60
4.2 Pre-defined evaluation routes . . . 61
4.3 Image quality comparison . . . 63
4.4 Astana transparency . . . 64
4.5 Astana variation beams . . . 66
4.6 Unity profiler . . . 67
4.7 Structure variations . . . 70
4.8 LCVR case study . . . 71

List of Tables

4.1 CineRender benchmark 62
4.2 Unity Backend benchmark 62
4.3 Occlusion Culling benchmark 64
4.4 Level of detail benchmark 65
4.5 Level of detail benchmark of Astana's variation 66
4.6 Design Merge benchmark 68
4.7 Design Merge and pointlights benchmark 69

Acronyms

IST Instituto Superior Técnico

CAD Computer-Aided Design

BIM Building Information Modeling

AD Algorithmic Design

VR Virtual Reality

IDE Integrated Development Environment

UI User Interface

GE Game Engine

PBR Physically Based Rendering

AEC Architecture, Engineering and Construction

AR Augmented Reality

UI User Interface

LOD Level of Detail

CSG Constructive Solid Geometry

VIM Virtual Information Modeling

LCVR Live Coding in Virtual Reality

FPS Frames per Second

ANL Astana National Library

1

Introduction

Contents

1.1 Algorithmic Design 3

1.2 Problem 4

1.3 Goals 6

The digital era has greatly influenced architecture and its design process. A number of digital tools, namely Computer-Aided Design (CAD) and Building Information Modeling (BIM) tools, are used nowadays to design buildings, increasing productivity, and the production of technical documentation. This evolution to the digital medium has also led to advancements in the complexity of the designs [Hensel and Nilsson, 2016].

Nowadays, the digital design process of an architectural creation depends on the execution of several tasks, such as 3D modeling, analysis, and rendering, which require different tools [Castelo Branco and Leitão, 2017]. Among those tools, CAD and BIM applications are the most prominent ones for 3D modeling and rendering, as they provide a digital way to model a 2D or 3D representation of a building. Additionally, the BIM paradigm further complements the digital model with relevant metadata, such as material costs and quantities, to support other related activities such as construction and fabrication [Kensek and Noble, 2014]. As for the analysis tools, they are used to perform simulations to infer a building's performance regarding specific criteria. The architectural criteria might be structural, thermal, lighting, cost, or other requirements. Different analysis tools are used to study these different criteria, such as Radiance[1] for lighting evaluation, Energy Plus[2] for thermal evaluation, and Robot[3] for structural evaluation.

The usage of these multiple tools to construct an architectural project often imposes 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 restructure the building's design, but also because of the propagation of these changes to the multiple different analysis models tied to the building design.

1.1 Algorithmic Design

The need for several tools and constant changes proves to be a setback in creating complex architectural designs. The Algorithmic Design (AD) approach was developed specifically to solve these issues [Castelo Branco and Leitão, 2017]. It allows for the creation of arbitrarily complex parametric models and the automation of tasks involving multiple tools.

AD is the development of architectural designs through the use of algorithmic and mathematical descriptions. The end result is a program that generates a model of the design for either visualization tools, such as CAD or BIM tools, or, inclusively, a model for analysis tools, such as Radiance or Robot. This algorithmic and mathematical nature of the program brings advantages to the architectural design process, such as: (1) the ability to create an abstract description of a building that can be represented in

[1] Radiance: https://www.radiance-online.org/
[2] Energy Plus: https://energyplus.net/
[3] Robot: https://www.autodesk.com/products/robot-structural-analysis/overview

different tools, (2) the ability to automate and generate complex geometry, (3) the ability to parameterize the model's description, bringing flexibility to the design, and (4) the ability to adapt the parametric data along with the results of an analysis tool to find the optimal design according to a given design criteria [Castelo Branco and Leitão, 2017].

With the AD methodology, architects can create complex parametric building, which can be easily analyzed and changed according to their needs. Figure 1.1 shows an example of the parameterization capabilities in action.

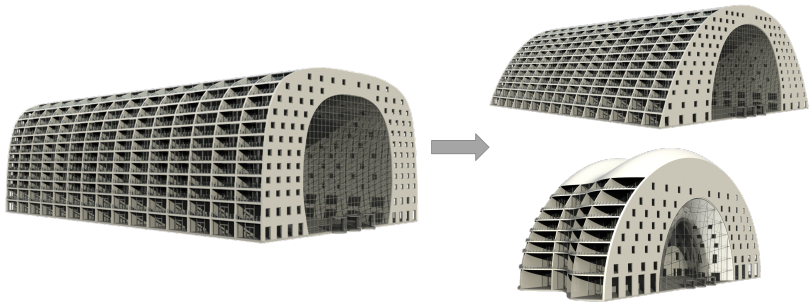


Figure 1.1: The Market Hall, in Rotterdam, designed using the AD approach, in a parametric fashion. Depicted on the left, is the original design, and, on the right, two design variations, generated by applying simple changes to the parameters' values. Source: [Leitão et al., 2013]

Alongside with this novel methodology, an extension of it was proposed, named Integrated AD [Castelo Branco and Leitão, 2017]. This extension suggests a workflow that complements AD with the required tools to satisfy the various phases of the design process, represented in Figure 1.2. This workflow starts with the coupling of an AD tool with CAD tools at initial stages to program the draft of the design. After the design concept takes shape, the model is sent to the analysis tools for performance evaluation. At this stage, the AD tool will generate the specific analytic model for the respective analysis tool. When the evaluation is performed and is adapted to improve the performance, the final design is ready to be generated on a BIM tool, where a more detailed model is created.

1.2 Problem

An AD approach requires that the architect, instead of modeling a design directly in CAD and BIM applications, creates a parametric program that generates a design model. However, writing such a program is not a trivial task. Coding complex designs demands 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 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 make

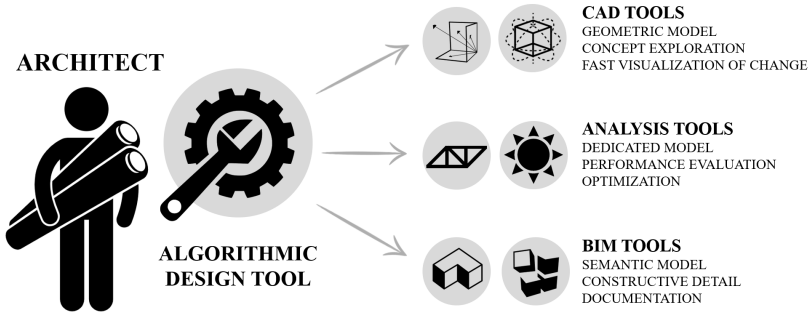


Figure 1.2: The AD workflow, containing design concept exploration in CAD tools, followed by performance evaluation with analysis tools, and a final detailed design model for construction using BIM tools. Source: [Leitão et al., 2019]

a subjective aesthetics evaluation. Additionally, injecting algorithmic logic can hinder the creativity of an architect when designing a building [Castelo Branco and Leitão, 2017]. Figure 1.3 illustrates this disconnection problem that the AD approach imposes on an architect who is new to this approach.

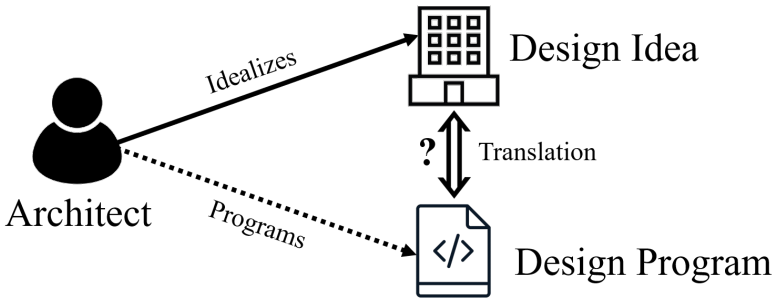


Figure 1.3: An architect can easily idealize a design idea. However, describing it in terms of a computer program poses a challenge, hence making this approach unappealing as it may be seen as an unnatural way to design.

One could benefit from the use of visualization tools to complement the design programming, lessening this disconnection. But unfortunately, currently used visualization tools, such as CAD (e.g., AutoCAD and Rhinoceros) and BIM (e.g., Revit and ArchiCAD) applications, suffer from performance issues as an AD project grows in scale [Johansson and Roupé, 2012]. This is particularly severe due to the nature of AD, which enables the quick generation of large amounts of geometry without much effort, easily overwhelming the visualization tool. Such a harsh slowdown on a visualizer 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.

The last stage of the architectural design process, where high-quality renders need to be generated for design presentation to clients, is also in need for improvements. From our experiments, we have found that this stage, with complex designs, may take days, sometimes even weeks, to accomplish, even on a specialized rendering workstation [Leitão et al., 2019].

1.3 Goals

Typically, architects following the AD methodology resort to CAD or BIM applications to serve as visualizers for the results of their AD programs. However, these applications were designed for interactive use and often become a liability in terms of performance with the considerable amount of data generated by the AD approach. These applications prove to be insufficient when dealing with large scale AD descriptions because their performance problems delay the visualization of the generated design, thus making AD harder than necessary.

To mitigate these effects, the architect needs to have immediate visual feedback of the design generated by the AD program, throughout the different phases of the project development. This feedback allows them to freely experiment with the design, thus expressing their ideas and concepts, but also to promptly correct mistakes in the AD program as they arise and fine tune the design's parameters with ease [Moloney and Harvey, 2004]. This reduces the distance between the architect's idealized design and the design program's results, increasing program comprehension [Rugaber, 1997], and thus making programming a less daunting task. Additionaly, it also allows for a better aesthetics judgment and decision making [Ashour and Kolarevic, 2015]. For instance, during the analysis stage, architects may be presented with several variations of their designs to choose from. Being able to quickly visualize each variation in detail allows them to make more informed and faster design decisions.

To this end, a fast and interactive visualizer is required. This visualizer needs to have good visual quality to compete against CAD and BIM tools, while still providing better performance. Additionally, with good interactivity we are able to introduce several benefits that were not previously present in the AD's process. For instance, the possibility of real-time navigation on the design along with direct interaction with the building elements, like opening a door.

However, we predict that the development of a visualizer, tailored for architectural designs, capable of maintaining a high enough frame rate to guarantee good interactivity, can be a difficult task, because of how detailed designs can be and how their complexity can vary and scale indefinitely. Therefore, ultimately, our main goal is to provide a visualizer that is capable of scaling better in performance, than the aforementioned CAD and BIM based visualizers, when faced with large and complex designs.

For the purpose of developing a visualizer that satisfies the requirements imposed by the AD approach, in the next section, we will study existing visualizers aimed at architectural design. This will be followed by an exploration of various acceleration algorithms, commonly used by real-time visualizers, with the goal of improving rendering speed without significantly deteriorating visual quality.

2

Related Work

Contents

2.1 Existing Visualizers 9
2.2 Acceleration Algorithms 16

This section is split into two subsections focused on exploring concepts to enrich our visualizer. Section 2.1 is where we are going to explore existing visualizers, focusing on those used in architecture in order to obtain knowledge on what kind of features are important to include in our own visualizer. In section 2.2, we will be exploring some of the technical knowledge on acceleration and optimization algorithms used in real-time visualizers. These algorithms are crucial to enable fast visualization.

2.1 Existing Visualizers

Although the area of visualization is broad, this research focuses on the visualizers aimed for architectural designs and also those related to AD. Considering that the AD methodology is recent, only a few visualizers were specifically made for it. For this reason, we also investigate solutions that fall outside the architectural context. This way, we can assimilate a broader spectrum of ideas into our fast and interactive visualizer for AD.

We will start by investigating the most important tools for architecture, CAD and BIM, which are not only visualizers but also encompass means for the creation of design models. Afterwards, we will discuss in more detail its falloffs, which motivated the goals of this thesis.

2.1.1 CAD and BIM

For production and visualization of 3D models, the architectural industry is divided into two paradigms: CAD and BIM.

Both consist in the creation and modification of a design by means of a computer. Their goal is to increase design productivity and quality, as well as assist in the production of technical documentation. With these tools, an architect can freely create, explore, and visualize their designs, either in 2D or 3D space.

BIM is a more specialized type of CAD, encompassing construction logic and collaboration information into the design primitives. BIM covers not only the geometry, but also the spatial relationship of elements, geographic information, and the building components' quantities and properties. This brings us to the concept of BIM families, which are a group of parametric building components commonly used. Inside a component's family there can be several variations of that said component, for instance, a wall family encompasses wall variations of different sizes, shapes, colors, materials, etc. By using BIM applications, such as Revit[1] and ArchiCAD[2], an architect can save a lot of time during the modeling of the design, since these applications already provide built-in libraries with detailed family elements to choose from.

[1] Revit: https://www.autodesk.com/products/revit/overview
[2] ArchiCAD: https://www.graphisoft.com/archicad/

A BIM model can provide additional benefits that could not be achieved by CAD tools alone, such as automatic generation of detailed technical documentation, which helps to better manage the cost and the life cycle of a design. Furthermore, having the information to support related activities, such as construction, in a single file format, leads to improved communication about the design with all stakeholders of the project, rather than just architects [Kensek and Noble, 2014].

All these benefits end up making BIM models essential in an architectural project, at a cost of added complexity when compared to the CAD counterpart.

While the BIM paradigm focuses on enriching the design with additional information, CAD embraces simplicity, thus allowing its designs to take any shape the architect desires without being constrained by the construction details and various other BIM requirements. As such, creating and manipulating designs in CAD applications, such as AutoCAD,[3] Rhinoceros,[4] and Sketchup,[5] is a contrastingly easier and faster process, hence architects typically tend to opt for said applications in the early design stages.

Both CAD and BIM applications show multiple views of the design model with various options to manipulate it in their interface, as illustrated in Figure 2.1. Additionally, both kinds of applications are

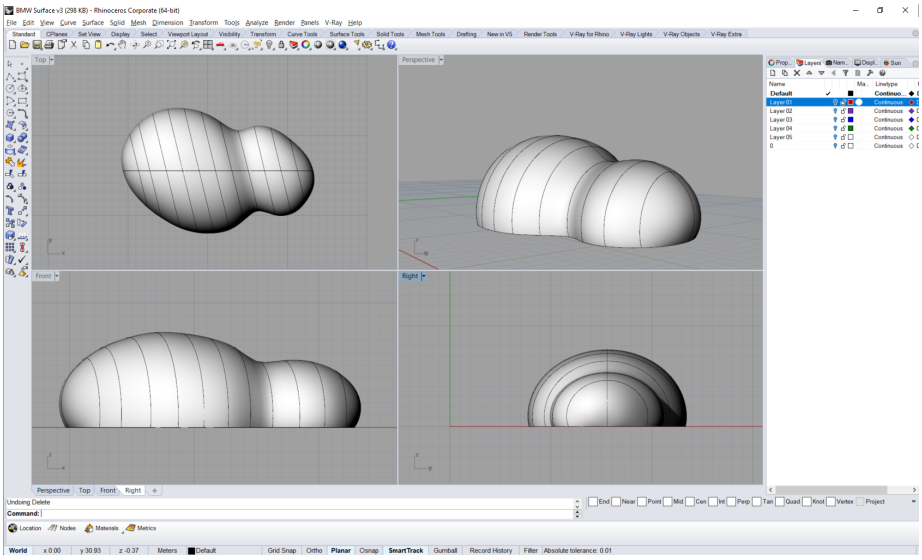


Figure 2.1: A screenshot of the Rhinoceros 5 interface, where four different views of the design are presented.

capable of offering visually appealing rendered results, but, despite serving their purpose by modernizing the current architectural design process, they have their limitations. Their 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 gets bloated with the geometry fed by an algorithmic description. Natively, both CAD and BIM tools have 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

[3] AutoCAD: https://www.autodesk.com/products/autocad/overview
[4] Rhinoceros: https://www.rhino3d.com/
[5] Sketchup: https://www.sketchup.com/

renders. Only the former supports a form of free-fly navigation, where the user can fly anywhere and pass through solid objects, but not only is this hard to use, because of the keyboard shortcuts required just to position the camera in a desired spot, but it also only displays a low fidelity view of the scene. If an architect wants to see that view in high quality, they must wait for the rendered result. This wait time hinders the architect's productivity and train of thought. We aim to tackle this problem by developing a visualizer with good navigation capabilities alongside a good visual representation of the model.

Currently, if the architect intends to visualize its design in real time and high quality, external tools or plugins must be used. An example of a plugin would be Enscape. Enscape[6] is a plugin for Revit, SketchUp, Rhinoceros, and ArchiCAD, that is capable of rendering a design model in real time and provides various navigation capabilities to explore it. It also enables a Virtual Reality (VR) visualization of the models through the use of VR headsets, such as the Oculus Rift,[7] in order to immerse the user into a virtual representation of the design model. While inside the model, the user can navigate around and control various aspects, such as the illumination level, time of day, the elements' materials, etc. By allowing the architects to be present in their design, it gives them a better visual feedback than any other type of visualization. The topic of VR in the context of architecture will be further discussed in section 3.3.8.

This plugin is insufficient for our needs since it only links directly to CAD and BIM applications, which creates an inefficient workflow for an AD user. A user that wishes to navigate in a design model created by AD must first generate it in a, for instance, CAD application, and only afterwards can the model be transferred to this plugin. This behaviour is undesirable as it leads to slowdowns as a project grows in scale. Additionally, as it lacks quality control mechanisms, such as control over the quality of the lighting of the scene, its performance will not scale properly for large designs.

Other external tools will be discussed in the following subsections, but first, we will explore Luna Moth, one of the few fast visualizers for AD.

2.1.2 Luna Moth

Luna Moth is a web-based Integrated Development Environment (IDE) for AD [Alfaiate and Leitão, 2017, Alfaiate et al., 2017]. Figure 2.2 illustrates this tool's User Interface (UI) along with a usage example. The problems that led to the development of this tool are very similar to ours: aiming to reduce the architect's waiting time for visual feedback of the changes, and provide a visual way to help them with the programming task.

It integrates with and improves the AD workflow by adding the following benefits: (1) portability, by taking advantage of the predominance of web technologies, it provides a cloud-based IDE that can be

[6] Enscape: https://enscape3d.com/
[7] Oculus Rift: https://www.oculus.com/

accessed anywhere, (2) interactivity, by providing an immediate feedback visualizer that displays the design model as it is being written and on every change to its parameters, and (3) usability, by providing an IDE that is simple, easy to use, capable of manual interactions with the program parameters, using sliders, and with the visualizer, by having the ability to trace from which instructions the design model elements came from and vice-versa. All of these qualities aim to aid the programming task, especially the latter. We consider traceability to be the strongest point of this application as it greatly increases program comprehension [Rugaber, 1997], particularly, the relation between what the user writes in the design code and the expected rendered result.

Although a performance study of how this tool fares with complex models was not performed, its visualizer implementation lacks acceleration techniques. Additionally, it only provides limited navigation capabilities, and lacks the ability to add assets, such as materials, to the design. Nevertheless, it is an important object to study as it is one of a few fast visualizers developed primarily for AD.

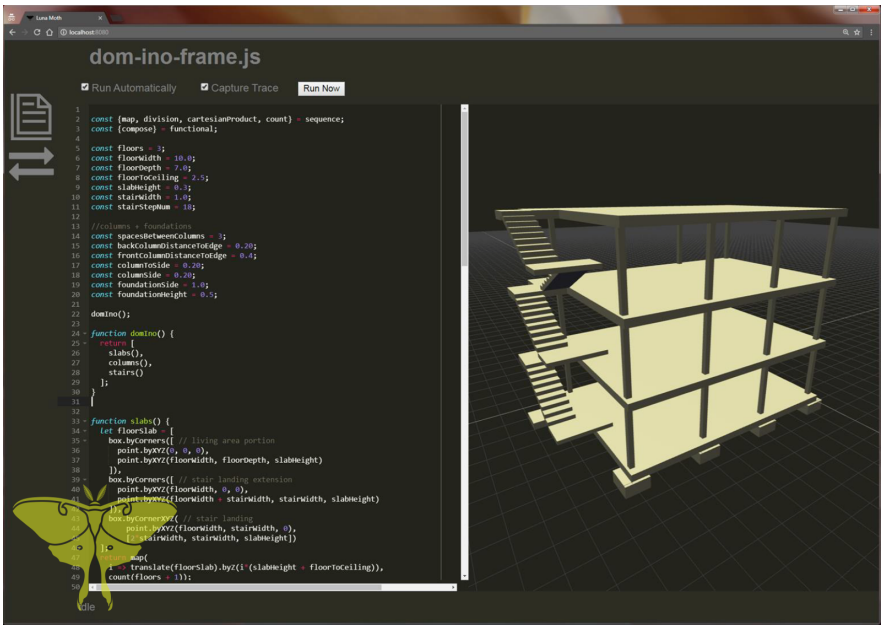


Figure 2.2: A screenshot of the Luna Moth IDE running in a browser. The left side contains a text editor with the AD program description while the right side contains the 3D visualizer implemented using three.js.

2.1.3 Twinmotion

Twinmotion[8] is an Unreal Engine[9] real-time visualizer capable of rendering CAD and BIM models in high quality. The Unreal Engine, despite being a Game Engine (GE), i.e., a tool for game development, in Twinmotion's case, it was repurposed to create a performant visualizer tailored for the architectural context.

[8] Twinmotion: https://www.unrealengine.com/en-US/twinmotion
[9] Unreal Engine: https://www.unrealengine.com

Twinmotion features a plethora of Physically Based Rendering (PBR) materials, real-time radiosity, and a library containing not only static assets, like furniture and rocks, but also animated assets, such as animated trees with leaves blowing with the wind, to provide realistic renders. The user can choose to either visualize the design model only partially, by toggling a button to hide model elements, or in various different static views, namely an overview or a side view of the model. This is presented in a simple and clear interface, where most operations can be performed by a drag and drop operation.

With respect to navigation, it features two modes: a walking mode, where the user is grounded by gravity and cannot go through solid objects, and a free-fly mode. During navigation, the user can control the navigation speed and the weather of the scene, resulting in different luminosity levels. Additionally, if the user intends, navigating on the design model in VR is also possible.

In order to visualize a CAD or BIM model, the user must first export their model from the design tool and later import it in Twinmotion. This solution is not fit for our purpose since we aim at visualizing the geometry quickly as it is being encoded by the user. Additionally, it only features a limited control over the quality level of the visualizer, such as material detail level, meaning it may not scale properly with large and complex projects.

2.1.4 Lumion

Similarly to Twinmotion, Lumion[10] is a real-time rendering engine capable of generating high quality results from CAD and BIM models. Other similarities include an interactive interface, a weather system, and a resourceful library of PBR materials and assets, along with the possibility to import custom ones. Lumion can only provide static VR capabilities, i.e., it can only pre-render a still panorama view of a design, as opposed to a dynamic view with navigation. Regarding the navigation, it only features a free-fly mode, outside its VR mode. It also allows for a better quality control on the scene, such as lighting control providing three different options: (1) accuracy, a realistic but computationally expensive way to compute the lighting, (2) speed, a faster and less realistic lighting and shadowing, and (3) memory, which is an hybrid of the former two that uses additional memory to increase the lighting quality without excessively affecting the performance by using pre-calculated lightmaps.[11] Other post-processing effects include depth of field, lens flare, bloom, chromatic aberration, etc.

Lumion also uses an export-import mechanism to communicate with CAD and BIM applications, deeming it inadequate for an AD workflow, as this kind of mechanism is slow to process. Additionally, the usage of Lumion may not scale well with large projects since it focuses primarily on high quality renders.

[10] Lumion: https://lumion.com/

[11] Pre-calculated lightmaps refer to textures on which the lighting was statically calculated before runtime and embedded into the texture.

2.1.5 VIM AEC

VIM AEC[12] is a real-time interactive visualizer based on the Unity[13] GE for Revit BIM models. It presents a clear interface only capable of basic operations, such as: the ability to toggle the visibility of certain objects in the scene, the ability to add metadata to objects, and the possibility for VR. With respect to navigation, only a free-fly mode is made available.

This application, however, is still very limited in terms of features, provides little to no extensibility, and, above all, lacks various important features that were present in previously studied visualizers, such as an asset library, realistic lighting, weather control, etc. Nonetheless, it introduces us to the concept of Virtual Information Modeling (VIM). In contrast with BIM, VIM aims to join the best of both worlds by integrating the interactive capabilities of GEs along with the model-enriching feature of BIM's metadata. Thus, the result is an improved BIM concept capable of fast visualization and VR, allowing for an enhanced communication with the stakeholders about the design.

2.1.6 Unity Reflect

Using the same base concept as the previously studied visualizer, Unity Reflect [14] is a novel real time visualizer developed by the Unity team itself in collabration with Autodesk, the creators of AutoCAD, a CAD application. This collaboration brings the Architecture, Engineering and Construction (AEC) community and the gaming industry even closer together.

Unity Reflect features a one-click connection with Revit to visualize an architectural design model in a GE based visualizer. As such, aside from the possibility to visualize a BIM model in real time, it enables: (1) real time analysis, for instance acoustic analysis, (2) portable experiences, either in a desktop or mobile device, and even allows for (3) VR and Augmented Reality (AR) integration. The latter brings a novelty to AEC. As an example, one could project a design model, stored in a smartphone or tablet, to the yet unbuilt designated construction site using AR to get a more accurate grasp on the looks of the building, as if it was already constructed there. This scenario can be seen in Figure 2.3.

Differently from the previously studied solution, Unity Reflect is meant to be extensible and customizable, even allowing a connection to Unity's editor to further modify the model in various ways, such as the use of assets, props, and materials, to adorn it. Additionally, as mentioned in the previous visualizer, this is all meant to leverage BIM's construction information, among other types of information, with the power of a GE. Therefore, this application aims to be used by the various stakeholders of an architectural project to improve communication. As for architects, since this application supports real time changes, an architect only needs to work on the model directly in Revit and see the changes being applied within

[12] VIM AEC: https://www.vimaec.com/
[13] Unity: https://unity.com/
[14] Unity Reflect: https://unity.com/products/reflect

Unity Reflect directly, needing little to no former experience to use this application. However, as an interactive extension of Revit, this still lacks the appeals needed for a tool to be used for AD. Nonetheless, at this point, we are sure that GE sets a good foundation for the development of a powerful visualizer.

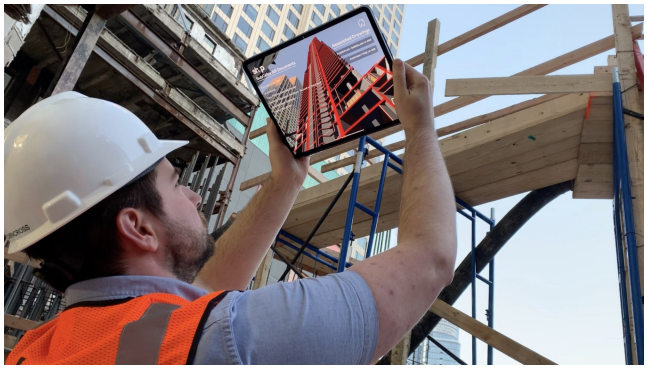


Figure 2.3: Unity Reflect being used in AR mode to visualize a building in contruction. Source: https://unity.com/products/reflect

2.1.7 Game Engines

GEs' main purpose is not for architectural visualization. Instead, they are a intended for game development. However, as seen previously, and in accordance to many other authors [Indraprastha and Shinozaki, 2009, Shiratuddin and Thabet, 2011, Ratcliffe and Simons, 2017], there is an enormous potential in taking advantage of GEs' power for visualization in various AEC activities.

Nowadays, given GEs' presence, their popularity, and market value, they are well maintained and studied tools, packed with all sorts of acceleration algorithms. Furthermore, most of them are very well documented and programmable, meaning GEs can be easily adapted [Fritsch et al., 2004]. In our case, we want to adapt them for the architectural context, in particular, taking the shape of an interactive and fast visualizer for AD.

The most popular GEs nowadays are Unity and Unreal Engine. These engines carry all the qualities that make up a good visualizer, such as: (1) portability, since they are capable of running on most platforms and can create program builds compatible even with gaming consoles; (2) they have an active community, composed of developers and users that constantly improve the tool; (3) they are updated on a regular basis, to augment the GE tool with the latest algorithms and compatibility with the state-of-the-art visualization technology, such as VR; (4) they have high quality real-time visualizers, which support PBR materials, lighting, shadowing, and many other effects; (5) there is a multitude of assets, present either on each respective asset store or user-imported; (6) they include a physics engine to allow interactions to optionally obey the laws of physics; (7) they are highly configurable and are ready to be adapted to the user's needs, including quality performance control; (8) they are programmable,

either through scripting or GPU shaders.

In regards to navigation, one can implement any kind of navigation on account of how programmable GEs are. The physics engine allows us to add different kinds of scene navigation, such as wheel-chair navigation for accessibility simulation [Boeykens, 2011] and other interesting experiences with object interaction, in an architectural context.

This means that GEs' capabilities satisfy our needs, and this is also proven by the fact that they are at the base of most of the real-time tools we discussed previously. Therefore, they are a good candidate for our solution.

2.2 Acceleration Algorithms

The acceleration work to achieve real-time and interactive visualization has been an on-going effort for many years. We can describe at least four performance goals for a visualizer: (1) large number of Frames per Second (FPS), (2) high resolution, (3) realistic materials and lighting, and (4) high geometry complexity. The first three goals, i.e., frame rate, resolution, and shading, can always be more demanding, but, past certain optimal values, there is a sense of diminishing returns to increasing any of these [Akenine-Möller et al., 2018], i.e., even though a higher frame rate is better, there is no reason for increasing it any further than the monitor refresh rate. However, for the last goal, it is important to note that there is no real upper limit to a scene's complexity, especially with the usual scale of architectural projects, hence the necessity for acceleration algorithms.

Before discussing the acceleration algorithms, it is important to decide what kind of visualizer we will be developing. As such, in the next subsection we will discuss rasterization and ray-tracing, along with their advantages and disadvantages. Afterwards, we will choose one of them as our rendering algorithm of choice for the visualizer.

2.2.1 Rasterization or Ray-tracing

There are two kinds of rendering techniques: rasterization and ray-tracing. Rasterization is described as the rendering through geometry iteration, mapping to screen space, and shading it using usually local lighting techniques. It is a very efficient technique capable of producing sufficiently good results at low costs.

Ray-tracing is a physics based technique described as the rendering through the use of rays, to simulate rays of light, that intersect and bounce around the objects in the scene. These rays are used to calculate the color of the screen pixels using both local and global lighting techniques. This technique produces photorealistic results, with reflections and refractions, but at a very high computational cost.

Although architecture praises high quality results, our main focus is speed. Rasterization has a rendering time that is typically linear with the number of geometric elements. Currently, modern GPUs are specialized for this kind of work. Despite that, ray-tracing has recently been gaining some traction. Some GPUs already support real-time ray-tracing, but this is still far from ideal. Thus, until the technology matures, and since we are looking for speedy results, we will focus on rasterization.

In the following subsections, we will explore some acceleration techniques for real-time visualization. The primary focus of acceleration lies in two main points: (1) Visibility Culling and (2) Level of Detail (LOD) techniques, both of which depend on Spatial Data Structures. Spatial Data Structures are sorted arrangements of the geometry in an n-dimensional space. Their organization is usually hierarchical, in the form of a tree, for a more efficient traversal search. Some examples of these hierarchical structures include: bounding volumes hierarchies, binary space partitioning trees, and octrees [Akenine-Möller et al., 2018]. These serve as a foundation to many acceleration algorithms as they accelerate the queries of the geometry, i.e., queries for culling algorithms, collision detection, ray-tracing, etc. [Clark, 1976].

2.2.2 Visibility Culling

Visibility determination has been a persistent problem in computer graphics. Algorithms for determining visible portions of a scene's primitives have been developed ever since 1970, when they were coined as hidden surface removal algorithms. Many implementations exist nowadays, but the most used one for interactive applications is the Z-buffer algorithm [Cohen-Or et al., 2003], which is hardware-supported. However, this is a brute force method that solves the visibility problem at a computational cost. On complex scenes, Z-buffer often suffers from overdraw problems, that is, when it draws several occluding objects, depending on the drawing order, it can draw, in the worst case scenario, every object without actually rejecting any of them, as illustrated in Figure 2.4. Ideally, we would like to perform a rejection of invisible geometry before the actual hidden surface removal algorithm, in order to reduce the geometry load. To this end, Visibility Culling algorithms should be applied beforehand.

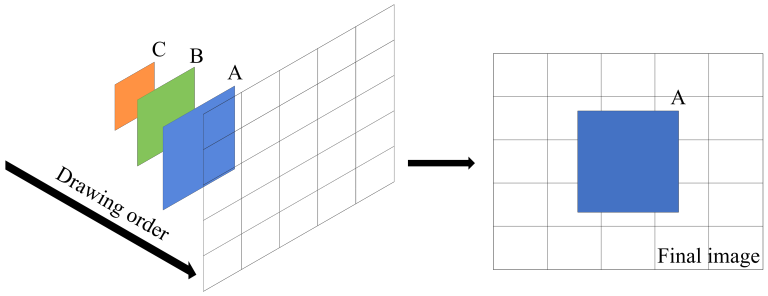


Figure 2.4: Z-buffer overdraw problem. Although the final image will only contain object A, as it occludes all others, they are all still drawn if the drawing order is: C, B and A.

Visibility Culling is responsible for the removal of portions of the scene that do not contribute to the final image, leading to less processing, since we no longer need to fetch, transform, rasterize, or shade invisible objects. If applied correctly, we can gain great performance benefits at no visual cost, even on large detailed scenes. This can be categorized mainly in three distinct types, illustrated in Figure 2.5: (1) Back-Face Culling, which eliminates surfaces facing away from the camera, (2) View-Frustum Culling, which eliminates geometry outside the camera's view frustum, and (3) Occlusion Culling, which eliminates objects fully obstructed by other objects.

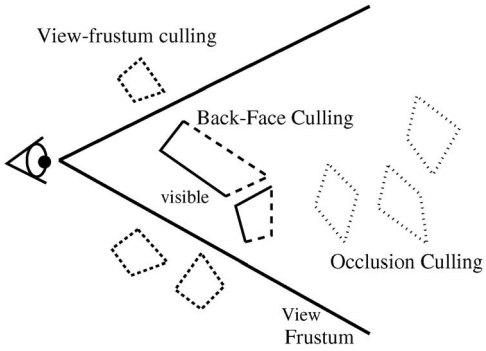


Figure 2.5: Three types of visibility culling techniques: 1) Back-Face Culling, 2) View-Frustum Culling, and 3) Occlusion Culling. Taken from [Cohen-Or et al., 2003]

Back-Face Culling and View-Frustum Culling have fairly simple solutions. Firstly, Back-Face Culling is solved by testing if the surface normal, given by the vertex winding order, is facing the camera. As for the View-Frustum Culling, it entails going through a structured scene hierarchy [Clark, 1976], and testing if the bounding volume of the object is inside the view frustum [Assarsson and Moller, 2000]. Architectural designs will greatly benefit from this, given that in indoor scenarios, only a minor part of the scene will be inside the viewer's view frustum. On the other hand, Occlusion Culling is a far more complex technique, in comparison with the previously described techniques, since it is a global technique that involves interrelationship among objects. It is also the one that provides the most performance benefits, specially in architectural designs, as they normally comprise of multiple large connected opaque elements, such as walls, which will occlude a great part of the scene [Teller and Séquin, 1991].

Much of the recent work on Occlusion Culling techniques are based on the work of Teller and Séquin [Teller and Séquin, 1991] and Aiery et al. [Airey, 1990, Airey et al., 1990]. An extensive survey work on Occlusion Culling techniques has been performed by Cohen-Or et al. [Cohen-Or et al., 2003].

2.2.3 Level of Detail

Typically, architectural projects are products of great detail, although creating large detailed scenes further hinders interactivity. For this order of detail to be mostly kept and to allow a real-time rendering, we can apply LOD techniques.

The core idea of LOD, introduced by Clark [Clark, 1976], is to use simpler versions of an object depending on how far it is from the viewer, as details will be less visible as distance grows. LOD will boost performance by reducing the amount of vertices to process, also including less pixel shading processing, on distant objects. This technique is best applied after culling, in order to reduce the amount of processing. LOD algorithms consist of three major steps: (1) generation, (2) selection, and (3) switching [Akenine-Möller et al., 2018].

LOD generation is the step characterized by the generation of different representations of an object with varying degrees of detail. These degrees of detail can range from changes in the model itself, due to the application of simplification algorithms or shaping a low detailed version of the object by hand, to changes in the resolution of textures and shading. There can also be changes in how the objects are represented, either by simplifying details using bump mapping,[15] or even replacing the entire object with a billboard[16] at longer distances. Some primitive objects, such as spheres, can be simplified by their geometrical descriptions [Hoppe, 1996].

LOD selection step is where we choose an appropriate LOD for an object based on a metric. The most common metrics used are *ranged based* or *projected area based*. On *ranged based* metrics, we associate the LOD based on the distance between the object and the camera. This is intuitive, since as an object gets farther from the camera, it consequently gets smaller, thus less visible. On the other hand, for the *projected area based* metrics, we use the bounding volume's projected area, or an estimation of it, as a metric to select the appropriate LOD. This way, even if an object is not that far from the camera, when its size in relation to it is very small, we will apply this technique.

LOD switching is the step where we change the LOD of an object to another. However, this switching will cause noticeably abrupt changes, an effect called popping. To alleviate this effect, there are several techniques, such as blend LOD, where, at the point of switching, a linear blend is done between the two LODs, which consists in adding a transparency on both LODs that changes inversely as one LOD switches to another.

Another advantage of LOD is that it can be adjusted to the computer specifications or the current frame rate. This means we can apply a more aggressive LOD policy, such as a closer selection range, in weaker workstations or when the visualizer frame rate falls below a certain minimum threshold [Funkhouser and Séquin, 1993].

[15] Bump maps are texture maps containing surface normals.
[16] Billboards are 2D sprites always facing the camera.

3

Proposed Solution

Contents

3.1 Unity Backend 23

3.2 Standard Features 24

3.3 Advanced Features 35

Given the results of our study on existing visualizers, we conclude that by using GEs, such as Unity and Unreal Engine, we can achieve both high visual quality and interactivity in real-time. GEs are highly optimized for providing realism even in large scale projects and are thus a good candidate for our needs.

Another alternative would be to develop a visualizer using low-level graphical libraries such as OpenGL or Vulkan. However, given how well developed GEs are and the fact that they can provide most of the acceleration features discussed in section 2.2, we find this alternative less appealing.

For our implementation, we choose Unity, not only because of our previous experience with it but also because, as it is free to use, it is one of the most popular GEs. With its large and active community we can expect Unity to be constantly supported and improved throughout the years and we can rely on its extensive documentation to guide us during the development. Furthermore, its simple interface makes the learning task easier for new practitioners, such as architects.

In regards to how we will integrate with the AD methodology, we will couple the proposed Unity-based visualizer with an existing AD tool, Khepri. Khepri is a novel AD tool capable of providing: (1) good performance, (2) a smooth learning curve for architects, (3) traceability between the AD model and the AD program, (4) backend portability, integrating several visualization and analysis backends [Leitão and Lopes, 2011, Feist et al., 2016, Aguiar et al., 2017], among other features.

In the context of Khepri, the term backend refers to the mediating software between Khepri and its supported visualization or analysis tools. Khepri users who intend to visualize and explore AD models already have various visualization backends at their disposal. However, those backends are based on CAD or BIM applications and, as mentioned in section 2.1.1, these do not fare well in performance with the complex models that the AD methodology is able to generate. As illustrated in Figure 3.1, the proposed solution intends to play this missing role as another visualization backend for Khepri.

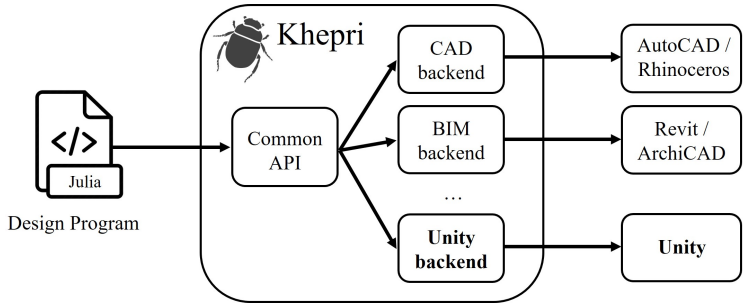


Figure 3.1: A simplified overview of Khepri's architecture, its supported backends and related tools.

3.1 Unity Backend

Inspired by the visualizers studied previously, we have organized a set of features that we deemed important in order to build our own visualization backend for Khepri.

Unity already provides a good foundation for achieving visualization performance and we will complement that with two other sets of features. The Standard Features (1), explained in section 3.2, are the ones that must be present in order to develop a working backend for Khepri. These features include, for instance, support for construction primitives and other Khepri operations, a high quality assets library, means for navigation throughout the design model, and a simple UI for the user to interact and customize the backend's behaviour.

Although these features alone can already provide a satisfying and enhanced visualization experience compared to the existing backends that Khepri supports, we can go even further by leveraging the power of this GE. The Advanced Features (2), described in section 3.3, seek to provide functionalities to improve and even innovate the workflow of Khepri users. Throughout our backend's development, is was already used by architects who wished to quickly visualize their Khepri AD models. This synergy helped identify potential shortcomings and improvements beforehand. As such, many of these advanced features were incrementally developed to accommodate new use cases. Such features include but are not limited to: support for bidirectional traceability between the design model and its design program, to enhance the program comprehension; layers, to help architects organize the generated design; standalone build, to build such design into a standalone runnable program for other project stakeholders to use without having to install Unity; integration with VR, to bring a new approach for AD architects to visualize and interact with their designs.

3.2 Standard Features

In this section, we will discuss the core features that must be present in our solution to develop a visualization backend that is compatible with Khepri. We will start by explaining the solution's structure and how the communication is done with Khepri.

The Julia programming language[1] is the core language used for Khepri's implementation. On the other end, Unity uses C# as its scripting language, so an extra step is required in order to make the communication between the two interoperable. Using RPC, Khepri is able to marshal its own data and send it for the Unity Backend to unmarshal it into the data structures it uses. As seen on Figure 3.2, our Unity backend represents the server, receiving the user commands from a Julia client, Khepri. On this side, there is a Khepri Plugin component containing the unmarshalling and the remote method invocation logic to call the appropriate methods on the server. On Khepri's side, there is a Unity Plugin component that declares all the available method stubs for the user to call and their data marshalling logic. These method stubs correspond to the methods defined inside the Primitives module, which contains the operations available for Khepri users to model their designs. On Figure 3.3, we can see an

[1] Julia: https://julialang.org/

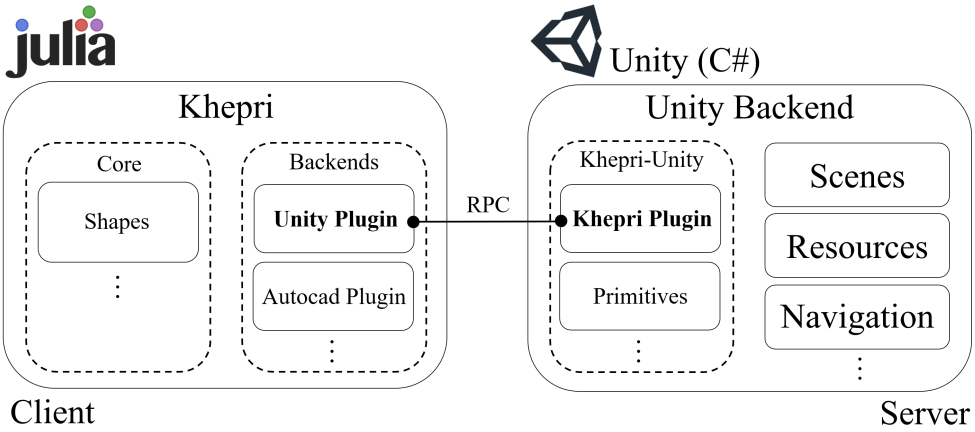


Figure 3.2: Overview of our solution's project structure. Khepri, running as a client under a Julia environment, and our Unity Backend, running as a server under a C# environment, communicate with each other via RPC.

example of Khepri operations running on a Khepri client, using the Atom IDE,[2] and the respective visual results on our Unity Backend server. In the next section we will discuss these operations in detail.

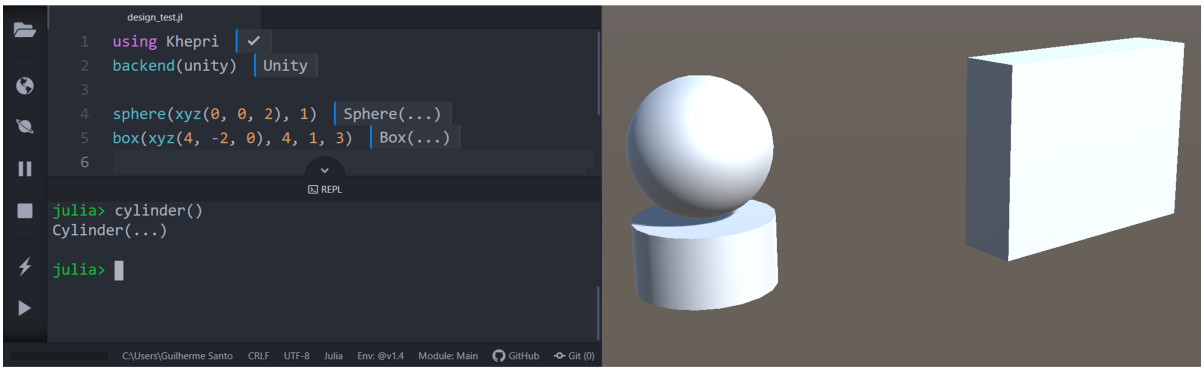


Figure 3.3: Interaction example between a Khepri client, on the left, and a Unity Backend server, on the right. Note that the Julia programming language is interactive and can receive operation at any moment either from a program file or a read-eval-print loop prompt.

3.2.1 Operations

With an established communication channel between our backend and Khepri, we must use this channel to process operations from the client, which will be responsible for the generation of the user's coded design. However, to be able to process these operations, which imply translating the user's code into an actual three dimensional representation of it, we need an understanding of Unity's own data structures.

In Unity, a Game Object represents the most elementary entity, which can be placed in a virtual world, called Scene. These Game Objects, in our context, can be used to represent a construction element of

[2] Atom IDE: https://atom.io/

a design, for instance, a wall. Game Objects can be acted upon through its composing Components. These Components define the behaviour of their corresponding Game Object, either by using pre-built Components or user-made scripts. Game Objects can also contain other Game Objects, creating a hierarchy which composes the scene graph[3]. As such, we need means to create these Game Objects in a Scene, representing the user's generated design model, and we need to be able to modify them according to the user's will using remote Khepri operations.

As Khepri supports a large variety of operations, we will only focus on the implementation of those that are most commonly used. The supported operations can be split into four different categories: (1) construction primitives operations, (2) basic geometric operations, (3) boolean operations, and (4) camera operations.

The construction primitives operations (1) represent all the operations that can create or delete a Game Object, 2D or 3D, in a Scene. These range from the creation of simple objects such as spheres, cuboids, cylinders, pyramids, to the creation of more complex objects with semantics, such as the BIM operations. The latter are related to the BIM families, described in chapter 2.1.1, and represent the operations that build common construction elements like windows, walls, slabs, panels, surfaces, beams, etc. These specialized operations for architecture are used by BIM applications to construct buildings, each family containing several variations of shapes and materials.

For simple objects we mostly use Unity's predefined primitives, transformed according to the parameters given by the user, e.g., scaling Unity's unit sphere to match the radius requested by the user. As for the complex objects, given a sequence of points that compose the shape of the object, an external helper library, Poly2Mesh[4] was used to generate the polygon meshes that compose these objects. All these objects are generated as static objects, which means that internally Unity will pre-compute some expensive calculations, like transformation matrices, improving the overall performance. This also allows Unity to perform static batching, which reduces the amount of draw calls[5] processed by the GPU for objects that share the same material and do not move. The downside is that, these objects can no longer be moved. In order to change their location they must be deleted and re-generated.

The basic geometric operations (2) comprise those that position Game Objects in a Scene, such as: scale, translate, and rotate. These are natively supported by Unity so the implementation of these methods is straightforward.

The boolean operations (3) represent those described by Constructive Solid Geometry (CSG) to create complex objects out of simple primitive objects, such as cubes or spheres, using three operations: union, subtraction, and intersection.

The union operation, as the name suggests, merges two primitive objects into a composite object.

[3] A scene graph is a data structure organized in a hierarchical fashion used in Computer Graphics to represent the relations of the vector transformations applied to a set of objects.

[4] Poly2Mesh: http://luminaryapps.com/blog/triangulating3d-polygonsin-unity/

[5] A draw call is a command call to the graphics API to draw an object.

The subtraction takes one primitive object and subtracts the overlapping portion of another primitive object from it. Lastly, the intersection results from the overlapping portion between two primitive objects. The result of applying these boolean operations to two primitives, a cube and a sphere, can be seen in Figure 3.4.

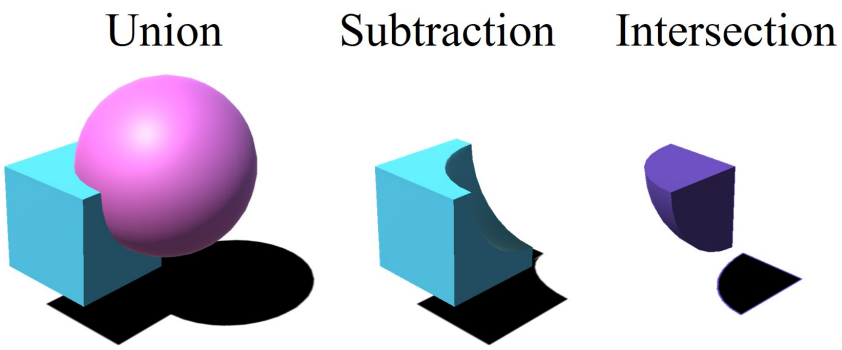


Figure 3.4: Given two primitive objects, a cube and an overlapping sphere, the result of applying a union can be seen on the left, in the middle the result of a subtraction of the sphere from the cube, and on the right the result of an intersection.

Typically used architectural tools, such as CAD and BIM, support CSG, which can be utilized to perform common tasks, such as opening a hole in a wall to insert a door or a window. Thus, the implementation of these operations on our backend is crucial. As Unity does not natively support them, for their implementation, an external open source library, named pb_CSG, [6] was used.

Finally, the camera operations (4) are those that programatically control the view of the Scene, save to disk a frame, and modify the camera properties, such as the lens' size. These operations are commonly used to create frame sequences using coded routes throughout the design model on the user's Khepri program. At the end, a video can be composed using all the frame sequences. For the evaluation of our solution, in section 4, these operations will be used to create fixed routes to perform benchmarks.

So far, we have explained the most important operations. Other utility-based operations were also implemented. They will be better explained throughout the following sections.

3.2.2 Assets

Assets are the elements responsible for giving the design model a degree of realism, thus, representing an important feature for a visualizer. These comprise: (1) materials, which adorn the generated Game Objects to give the user a better idea of its physical composition, and (2) 3D models, such as tables,

[6] pb_CSG: https://github.com/karl/pb_CSG

chairs, and trees, used to populate the design model to give a better sense of scaling and aesthetics. On Figure 3.5 we can see the visual impact assets have in a design model. The image shows the same view of a design, without and with the use of assets.

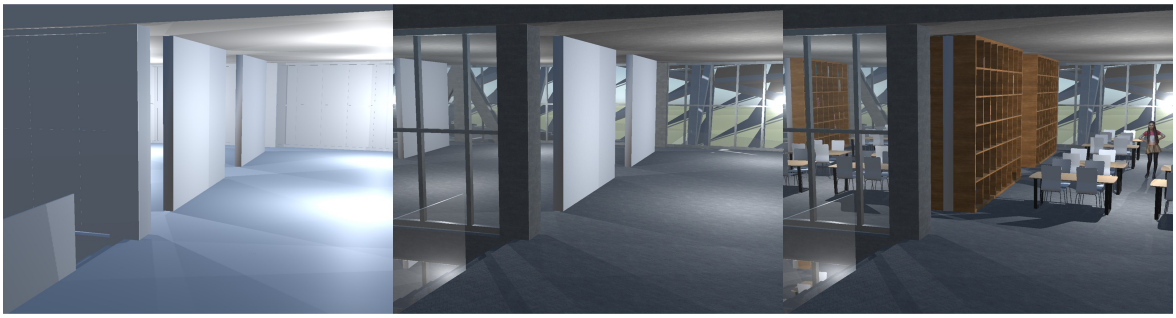


Figure 3.5: Interior view of a design model generated on our backend. On the left, no assets were used, on the middle, only materials on the construction elements were used, and, on the right, materials and 3D models were used.

For materials, Unity supports scriptable shaders and PBR materials. With those two features it is possible to create a high-quality material library.

As most of the construction operations are able to create a great variety of complex objects, including free-form objects,[7] such as surfaces or panels, it is important to ensure proper material application on such objects. However, the task of applying materials correctly in the general case is not easy. We identified three major challenges, illustrated in Figure 3.6. First, if an object was scaled in a non-uniform manner, the applied material would look deformed as its UV values[8] would no longer produce a visually correct result. Second, with free-form objects or objects that resulted from boolean operations, it is inherently hard to calculate correct UV values due to their arbitrarily complex shape. Lastly, when the same material is applied to objects next to each other, users often expect the material to be applied in a seamless manner. The result would be more visually pleasing if patterns in the texture would align, masking, for instance, the fact that a composite object is actually made of various other smaller objects.

We tackled these challenges using a combination of three methods: (1) use of seamless textures, (2) world-space texture mapping, and (3) triplanar mapping. Our goal was to be able to apply materials to any kind of generated object, complex or not.

Seamless textures (1) are meant for repeating patterns and allow us to disregard how the tiling would look like for any kind of objects of any size.

World-space texture mapping, or planar mapping, (2) is a technique that allows us to, instead of using the object's UVs, use the object's world coordinates to map a texture onto it. Using this technique along with seamless textures, we can map textures to complex objects without UVs. Additionally, since different objects share the same texture space, the world coordinates, if applied with the same material,

[7] Objects programatically generated by the user given, for instance, a sequence of vertices that form a shape.
[8] UV refers to the information about the object's texture space, used to apply a 2D texture over a 3D object.

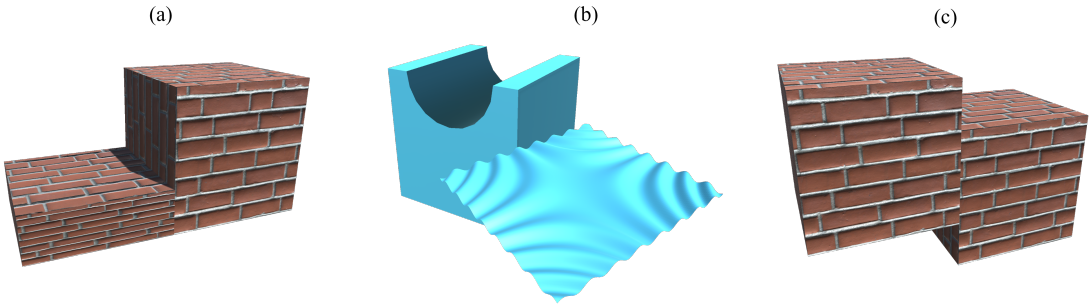


Figure 3.6: Material application challenges: (a), the left cuboid was scaled down in a non-uniform manner, hence its material looks compressed; (b) it is hard to calculate the UV values for these two objects, a cube with a cylinder subtracted and a wavy surface, thus, it is not possible to directly apply a material correctly; (c) two cuboids of the same size are placed next to each other with a slight vertical offset. As can be seen, even though the same material was used, the texture's seams do not align with one another.

patterns will align and connect seamlessly. However, this technique presented two drawbacks. Since the texture relied on the world coordinates of the object, if the object were to move, its material's pattern would change depending on the object position in the world. Nevertheless, this is a relatively innocuous problem in the case of AD, as it mainly deals with static objects. The second drawback is the fact that we are mapping a 2D space, the texture space, onto a 3D space, the world space, meaning that an axis would be discarded in the process. While the two considered axes, which form the plane where the texture is being projected, would no longer suffer from the previously mentioned material deformation problem, the discarded axis will. This happens because points on the same discarded axis would map to the same texture coordinate. In order to solve the deformation problem once and for all, we applied a technique called triplanar mapping.

With triplanar mapping (3), instead of mapping a texture once to an object, we apply it three times with different orientations according to the three different axes. Afterwards, we just blend the result, using the normal vectors of the object as a guidance for which oriented texture to choose from.

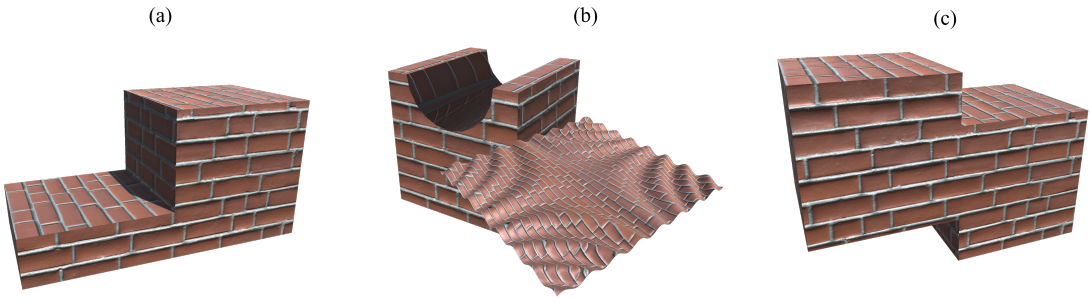


Figure 3.7: Material application resolved: (a) the scaled down cuboid's material is no longer deformed and the pattern aligns with the right cuboid's pattern; (b) we are able to apply a material to objects without UV values; (c) even with slight position offsets, the material of both objects align perfectly.

As for the implementation, unfortunately, Unity does not natively support these techniques. Nonethe-

less, these can be implemented thanks to Unity's support for scriptable shaders. Writing a shader that supports both triplanar mapping and world space mapping is not a hard task, however, we still desire to have all the PBR capabilities of Unity's built-in shaders so we can use high quality PBR materials for our designs. These PBR capabilities typically include: normal mapping, height mapping, occlusion mapping, material metallic and smoothness level, and many others. Combining these capabilities correctly with triplanar mapping is no easy feat. Fortunately, we can take advantage of Unity's Asset Store, which contains many third-party shaders that perfectly serve our purpose.

So far, along with the PBR triplanar shader and seamless textures, we already have a good foundation to build a library with all kinds of PBR materials for architects to use on their designs. Since Unity supports all sorts of external formats for textures to be imported, ultimately, it is up to the architects to create a material library that fits their needs, as there are various existing high quality and seamless PBR texture libraries out in the internet and even in the Unity's Asset Store. Nonetheless, our backend provides commonly used materials, which are required by BIM families, such as concrete, wood, steel, aluminum, glass, and plaster. These families, introduced in chapter 2.1.1, represent important component groups commonly used to aid architects in the construction of designs, such as walls, beams, slabs, panels, etc. Architects may choose to construct a design model using these BIM elements, as opposed to using the primitives directly, such as cubes and spheres. The BIM operations offers a semantic layer to the AD program which both increases its abstraction level and facilitates the modeling tasks for architects by automating the most typical modeling sequences, for instance, the *add_window* operation performs both the subtraction of a cuboid from the wall, and the addition of another cuboid with the proper window dimensions automatically. Each one of these families have pre-defined default materials and these are the ones that our backend should provide. Along with that, we provide Khepri operations to change the materials of objects and the default materials of BIM families, given its file path in the Unity project folder.

For optimization reasons, objects that use the same type of material will share the same instance of that said material. This reduces memory use and allows Unity to batch together and process objects that use the same type of material. The only trade-off here is that, if the material property of one object were to be modified, like its color, all the objects that share the same material would change. Since this use case is rare, this could be overcome by just instancing another material to apply that material variation.

As for the other type of assets, the 3D models, they are used to decorate design models in various ways, such as furniture, people, vegetation, etc. Since Unity supports a variety of popular model format, such as .obj, .fbx, and many others, an architect can easily import 3D models either from Unity's Asset Store or from other external sources. However, in order for these to be supported by our backend and to be generated through Khepri operations, a Unity Prefab containing the model must be created.

Prefabs are just Game Objects that were manually created previously, modified, and then saved on disk. In the current context, our Prefabs would carry Game Objects with Components that are capable of rendering 3D models. These Prefabs can also be transformed beforehand by the user, e.g., scaled to sizes that match other Prefabs, to better match the current architectural project requirements. With this mechanism, we can instantiate any kind of complex Game Object into a Scene using Khepri operations. Similarly to the material library, we provide a default library of model Prefabs, containing simple and static BIM families objects, like chairs, tables, and people 3D models.

As mentioned previously, we already provide some basic default assets, although this list is not extensive. Since the requirements of architectural designs can vary in many different ways, if desired, users can import custom assets that fit their needs. As long as they set their newly imported materials to use our PBR shader and create Prefabs containing the imported 3D models, these can be used right away in their code. Users should also mind the size and quality of their imported assets so as to avoid performance problems.

3.2.3 Navigation

In regard to navigation, our solution supports the following types of navigation: (1) free-fly mode, (2) walk mode, and (3) static overview mode.

Starting with the (1) free-fly mode, the user can fly at high speeds through the design model and pass-through any object. We can find this mode available in most of the visualizers studied in section 2.1. This mode is best used during the early stages of the architectural design process, as it makes it easy to quickly monitor the current output of the design program as it is being encoded by the user.

The second navigation mode is the (2) walk mode, in which the user can explore the design model in a more realistic manner. Grounded by gravity, the camera moves throughout the Scene at walking speed and will collide with the various composing objects. Collision detection is done by Unity using Components, included in all generated Game Objects, called colliders. The control bindings for this mode and the previous one are similar to those commonly found in first-person shooter games that use the keyboard as the input device, i.e., mouse position to control the camera orientation; ”W” key to walk forward; ”S” key to walk backwards; ”A” and ”D” keys to walk sideways, left and right side, respectively; hold ”SHIFT” key to move faster; ”SPACEBAR” key to jump or fly up, if on free-fly mode; and ”ALT” to fly down, if also on free-fly mode. A user can also cycle between this mode and free-fly mode using the keyboard ”M” key. The use of this mode is recommended at later stages of the architectural design process, when the design is mostly completed. Initial design stages frequently include the manipulation of isolated geometrical elements in space, which means users have no ground to walk on. Using this modes gives the user a better sense of the design scale and may help, for example, during interior decoration with assets.

On static overview mode (3), a user can define specific viewpoints of the design model and easily switch between them. This can either be done programatically or through special keyboard hotkeys during navigation. In the first case, the user navigates through the design model to find a good view of it, then, resorts to Khepri operations to save the current camera position as a fixed viewpoint in the code. Afterwards, another operation over those saved positions can be used to switch between them. This can also be done using keyboard hotkeys to save the current camera position using a combination of keys from: ”SHIFT” + ”0” to ”SHIFT” + ”9”, then switch between viewpoints using the same number keys. This mode works best during the last stage of the architectural design process, the render creation stage. One possible use case involves users saving several camera positions in their program, then, getting a render image for each one of those positions. At the end, an image sequence can be created, resulting in a video.

3.2.4 UI

All visualizers require a way for the user to interact with it and control its behaviour. The last standard feature presented is the backend's UI.

Since Unity supports interface scripts, we can easily integrate a custom interface into the Unity Editor, as seen in Figure 3.8, in which we allow the user to control the backend.

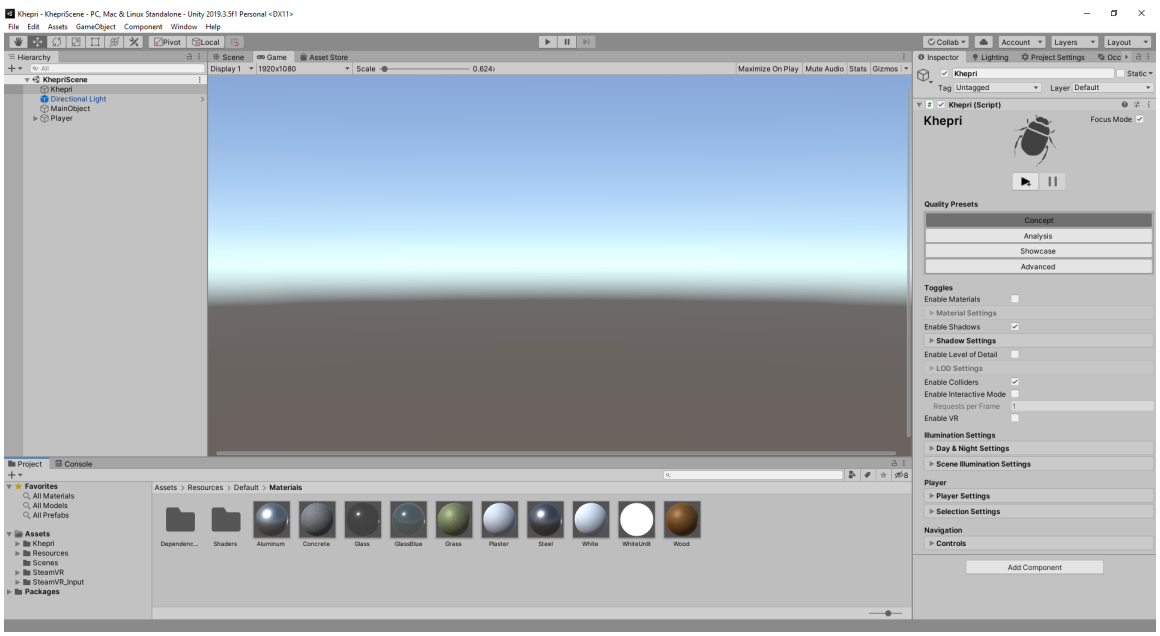


Figure 3.8: The implemented custom UI located on the right. On the bottom, our custom materials library is displayed and, in the middle, the Scene visualizer. Users can also freely rearrange all these elements to match their preferences.

The reason why we chose to develop our backend to be part of the Unity Editor and not as a stan-

dalone Unity application is because the Unity Editor provides useful and powerful features that would be difficult to replicate, such as the ability to import and create custom assets into the project, thus avoiding being limited to the existing default assets, and performance acceleration features, such as Occlusion Culling, mentioned in section 2.2.2.

Our interface can be used for two main purposes: (1) to initialize the communication with a Khepri client and start the navigation on the design model; (2) to allow users to configure the GE itself in many aspects.

For the first main purpose (1), our UI provides the user a way to start a server on our backend that waits for a Khepri client to connect to it. On the Khepri client side, the user must declare in the design program the backend of choice. In this case, if Unity was declared to be used as the visualization backend for the design program, a communication channel between the client and server gets established. Since the design program runs on Julia, an interactive language, users can run their code as it is being written and visualize the result right away. Additionally, the user can switch between navigating on the design or interacting with the UI using the right mouse button. This is necessary because during the navigation mode the mouse cursor gets hidden and locked in place to allow the users to utilize the mouse and keyboard to navigate throughout the design model. However, when users wish to configure the UI, they must switch out from the navigation mode to recover the mouse cursor. Figure 3.9 shows the portion of our backend's UI responsible for this functionality. This top portion of our custom interface shows at least two buttons. The left button starts a server on our backend and enables navigation. After establishing a connection with a client, the right button becomes active. This right button is used to pause this communication between the client and the server. This is important because the communication might impose a very slight overhead during the visualization of the design model. Hence, users may choose to temporarily pause this communication when they no longer need to run any new code.

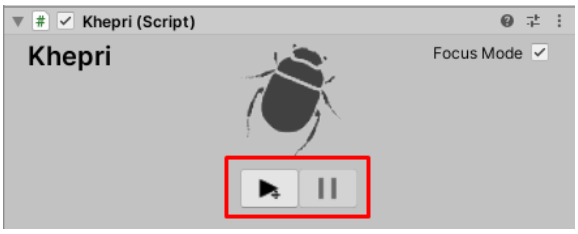


Figure 3.9: Top portion of the UI responsible for the Khepri communication and Scene navigation.

The topic of (2) configurability, on the other hand, requires more discussion. In section 2.2 we referred to four ideal goals: (a) large number of FPS, (b) high resolution, (c) realistic materials and lighting, and (d) high geometry complexity. In general, these are conflicting goals. Fortunately, Unity was designed to allow an easy configuration for trade-offs between these conflicting objectives. Depending on the project stage, the AD methodology has different requirements, which we can accommodate by allowing

a dynamic configuration of these trade-offs.

The first stage (1), called the geometric experimentation stage, is where an architect following the AD approach starts writing the initial coding for the building. This is also the stage where rapid development happens, thus high quality might not be the main focus. Instead, we would like to have more performance to be able to visualize all these changes as they occur. This development stage can last up until the architect is satisfied with the looks of the building.

The next stage covers (2) analysis and optimization studies, where the architect wants to test the generated solution according to certain design criteria, like thermal, acoustic, and structural. By taking advantage of the parameterization of the building solution, the architect starts to test different variations of the original design to find a fit solution, i.e., one that is in accordance with both the defined analysis criteria and the architect's aesthetics judgment. This stage is where we get closer to the final design, thus we require good visual fidelity but, nonetheless, still require the backend to be able to quickly show various design variations in quick succession.

At the end, the last stage, (3) project showcase, the architect must present and communicate the created solution to the various stakeholders involved in the project. Usually, this is done through a showcase of high quality renders of the outcome of the design program, thus we require maximum visual quality. Figure 3.10 shows a visual representation of these different visualization levels, where we adjust Unity's configurations as to allow either for a better performance or quality.

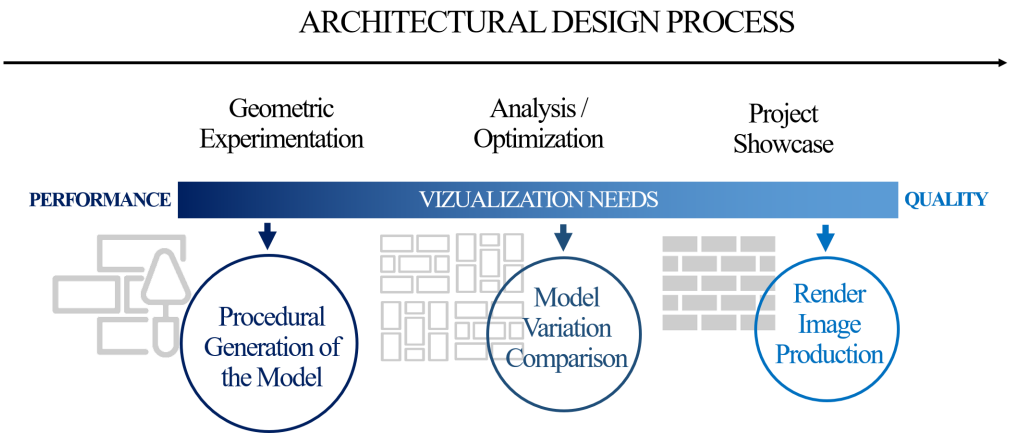


Figure 3.10: Quality and performance trade-off throughout the various stages of the architectural design process.

Unity provides their users with many ways to configure its settings to control performance and quality. However, learning how to use all of these settings can be a daunting task, especially for architects using Unity for the first time or not familiarized with computer graphics' terminology. To overcome that problem, as shown in Figure 3.11, we have implemented on our UI, in an easy to use manner, all the different levels of visualization with pre-defined configurations. These pre-defined configurations are tailored to the distinct use cases of each stage of the design process. For instance, if the Concept

preset is selected, generated objects will not have materials applied to them. Other configurations and even additional advanced features, which will be discussed in the following section, apply when different presets are selected.

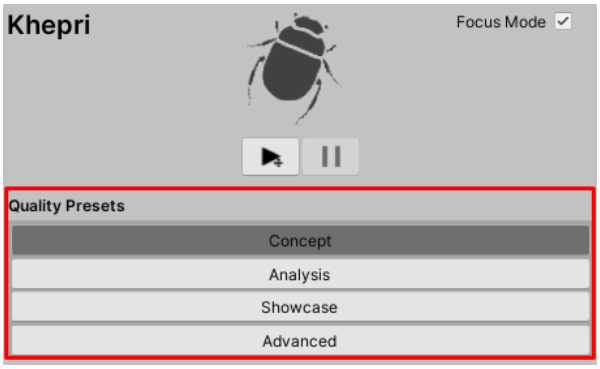


Figure 3.11: UI quality presets. Each button represent a configuration preset which change the quality settings of the GE to accommodate the needs of the represented design process stage. The last one allows architects to manually configure the GE to their preference, if none of the other presets satisfy their needs.

Architect should be able to easily switch between these levels of visualization using this interface. Additionally, every option in our UI provides a tooltip description when the user hovers the mouse cursor over it. This helps users learn unfamiliar terminologies and even the options' effects and consequences.

Since users' computers may greatly differ in specifications, it is hard to create pre-defined configurations that fit everyone's needs. So, it is important that we allow manual custom configuration by the user, in a more user-friendly manner, in order to make the visualizer fit their requirements, as shown in Figure 3.12.

So far, we have explained the features that make a functional visualization backend for Khepri. In the following section, we will discuss some additional features meant to complement the workflow of an AD user.

3.3 Advanced Features

In this section, we will discuss a set of additional features that were identified as relevant to achieve a better backend performance and improve the designing workflow either by decreasing the required effort to code AD programs or by introducing better tools for architects to inspect their designs.

3.3.1 Day and Night System and Scene Illumination

Architects often need to visualize their designs during different times of the day to ponder over a room's natural illumination. To support natural illumination studies, we decided to take advantage of Unity's

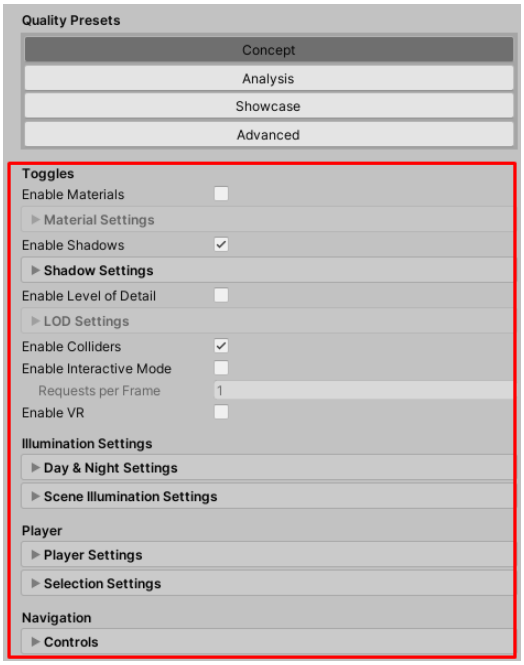


Figure 3.12: A portion of our interface that enables the user to fine-tune the GE. Depending on which quality preset is selected, this interface may differ as different presets support additional features, explained in the following section.

built-in dynamic skybox[9] capabilities. This skybox is able to change in color and aspect according to the sun's position, representing daytime or nighttime in a Scene. To represent the sun in our design Scene, a Game Object with an attached white directional light Component is used for the lighting, in which the built-in dynamic skybox will match the orientation of the directional light to create visual representation of the sun in the sky.

We allow the user to control the position of the sun in the sky in two different ways, either manually, or using Khepri operations. For manual operation, the user can set the sun position in a simplified manner through our UI using two sliders representing two angles. These two angles are enough to describe any position in the sky. Alternatively, the user can position the sun through the use of a single slider that controls the time of the day. This last method simply interpolates the given time in a rough and inaccurate manner to position the sun. However, this was implemented in such way to simplify this feature for the user. As seen in Figure 3.13, under the Illumination Settings dropdown we can find the interface that controls the sun's position.

Users can also control the sun's position in a more accurate way through the use of Khepri operations. This way, if required, the time of the day information can also be part of the design's code. Khepri provides operations that can set the sun's position either in a simple manner through two angles or by accurately calculating its analemma. The analemma is a diagram that represents accurate positions of

[9] Skybox refers to the visuals of the enclosing area that surrounds a Unity Scene. In our case, it is Unity's default clear blue sky with a gray horizon.

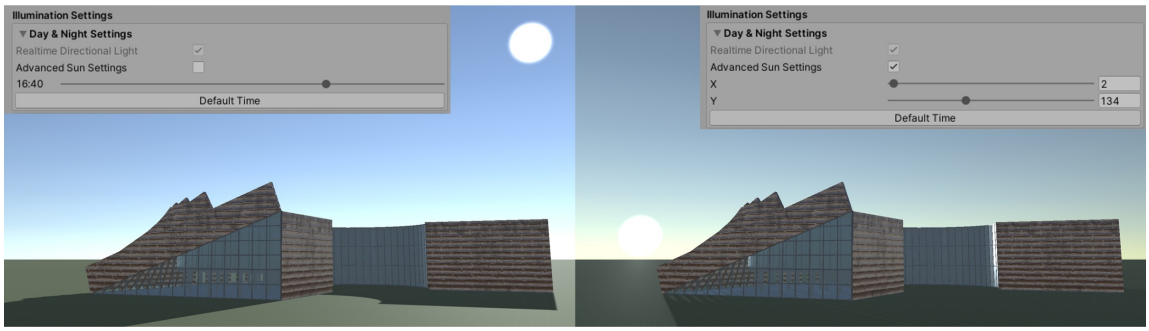


Figure 3.13: These two images with a generated design illustrate the different ways to manually control the sun position on our backend. Note the different sun position and sky color based on the UI configuration.

the sun in the sky, by taking in consideration the position of the viewer on Earth, and the date and time.

In regards to other types of illumination aside from the sun, we provide the user with the control over both indirect or direct illumination on Scene. On the topic of indirect illumination, Unity uses global illumination to cheaply simulate this effect. Using our interface, users can adjust this illumination to their preferences, either color of the light or intensity. This functionality can also be used on unrealistic scales to create concept arts of the design models, as illustrated in Figure 3.14.

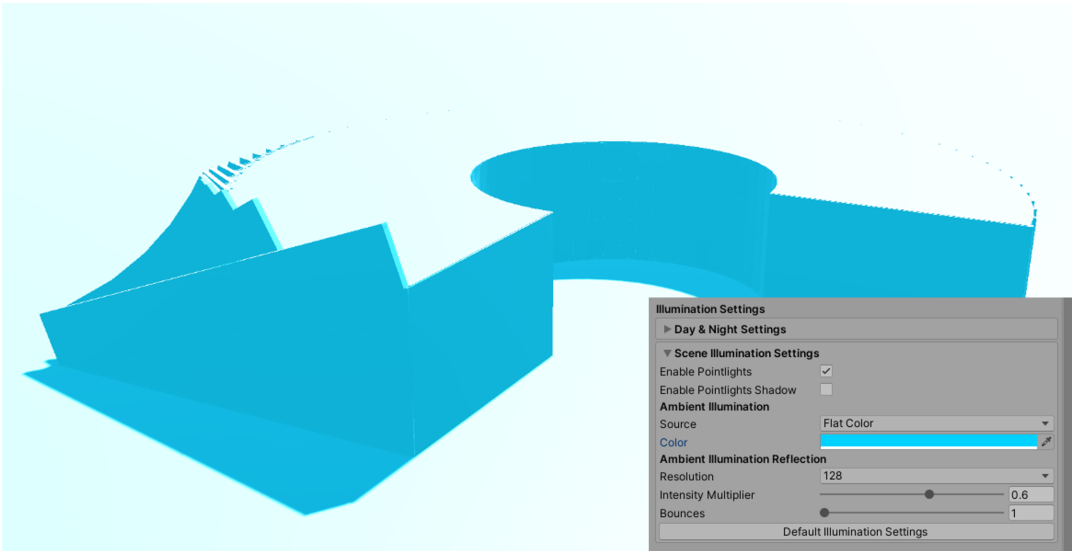


Figure 3.14: Inside the Scene Illumination Settings interface, the user can set the global illumination's light to illuminate the whole generated design model with, for instance, a blue color, to create a concept image.

For direct illumination, users can use Khepri operations to create pointlights in their design models. However, depending on the user's machine, these, when used in abundance, can create performance struggles. As such, we allow users to turn these off or even configure if these lights should cast shadows or not.

As for shadows in general, we also provide multiple configurations to control these. As shadows are

one of the most performance-expensive features of a visualizer, our interface provides various configuration controls over them, such as, quality level, and distance limit at which shadows no longer get calculated. All these configurations are meant to keep the visualizer running at acceptable frames while still providing the user with a desired visual result as the project grows.

3.3.2 Visibility Culling and Level of Detail

In this section we will explain the implementation of two acceleration techniques mentioned in section 2.2: visibility culling and LOD.

Starting off with visibility culling, as mentioned in section 2.2.2, this increases performance by avoiding processing objects outside the user's view, i.e., hidden objects that do not contribute to the final image of the scene. This can be applied using three different techniques: (1) Back-Face Culling, (2) View-Frustum Culling, and (3) Occlusion Culling.

For Back-Face Culling (1), most of Unity's shaders already perform this, including the one we use. In Unity, the front facing polygons are determined using a clockwise vertex winding order. Knowing the direction of this winding order is important in order to correctly create operations that generate free-form objects, using the vertices provided by the Khepri client to define such object. Likewise, Unity already performs a Frustum Culling (2) by default on every camera Component in the Scene.

Lastly, Occlusion Culling (3), which is the most complex of all the techniques, despite not being enabled by default, the Unity Editor already supports it. This is one of the reasons why we decided to integrate our visualizer inside the Unity Editor, as oppose to creating a standalone application. For this technique to be used, Unity needs to calculate Occlusion Culling data beforehand, such as dividing a Scene into cells containing the objects, and calculating the visibility between adjacent cells. Despite the performance benefits, one slight drawback of this technique is that these calculations may take several minutes depending on the complexity of the generated design. Additionally, as this is a technique that involves the interrelationship among all objects, this can only be performed when the Scene is already generated and in a final state, as design can no longer be modified for the calculated Occlusion Culling data to work. This means that this acceleration technique is only recommended at the later stages of the architectural design process and must be done manually by the users when they desire this performance increase, in exchange for the additional calculation's waiting time and increased memory usage. For users to calculate the Occlusion Culling data for their current design, they need to select the configuration preset button for Showcase for the related menu to appear, and then, through a single button press, the user can start generating this data, as seen in Figure 3.15. This menu is only present in this preset to avoid cluttering the UI with unnecessary features that we only deemed useful at later stages of the architectural design process.

In regards to LOD, as described in section 2.2.3, this technique improves performance by simplifying

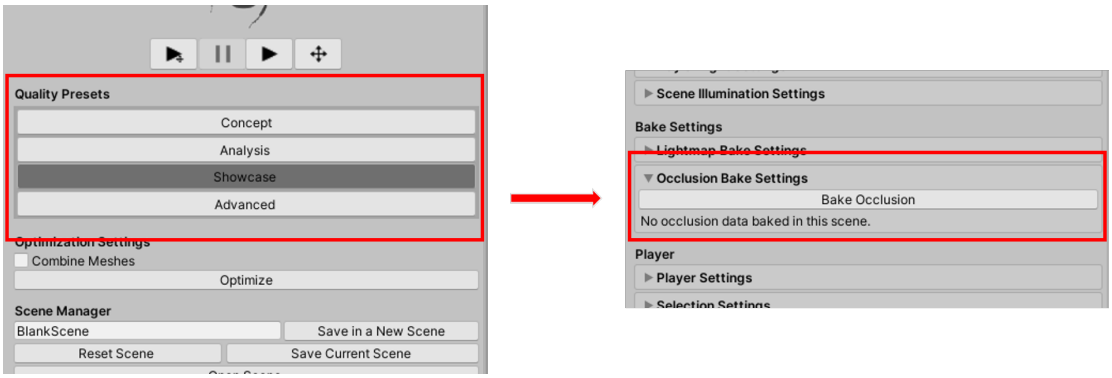


Figure 3.15: After setting the Quality Preset to Showcase or Advanced, the Occlusion Culling Settings interface will appear.

the details of distant objects, which might be barely visible because of the distance. We described three major steps to execute this technique: (1) generation, (2) selection, and (3) switching. Unity already provides an implementation for the last two steps, selection and switching, through the form of a Component for Game Objects. However, including such Component in our generated objects is not sufficient to achieve a LOD implementation. Not only we still need to ponder about how we will execute the selection step, i.e., how many different levels of detail shall we include for each object and which selection metric shall we use to switch between them, but also, we are still missing the implementation of a crucial step, the generation. The generation step is characterized by the creation of additional representations of an object with different degrees of detail. In the context of our system, we need a way to create those different representations for all our generated objects. However, it is not viable to manually create those representations for each object we support, especially if we plan to include many different levels of detail for each object. Although it might be an easy task for simple solids like spheres and cylinders, we have to remember that our system supports more complex objects, such as free-form objects, whose shape is not know *a priori*. In fact, the more complex an object is, the better are the performance gains by applying the LOD technique on it, so we must place our efforts in that group of objects. As complex objects may take any shape, manual approaches are out of question. Instead, we must resort to automatic approaches to generate a LOD representation.

For the implementation of such approach we must target the object's polygon mesh for simplification. A polygon mesh is a set of vertices, faces, and normals that describe the shape on an object. We describe the complexity of a polygon mesh by the number of vertices or triangles that compose it. The higher the number of vertices, the more computationally expensive it is to process. The solution is to take advantage of mesh simplification algorithms that, given an arbitrarily complex mesh and a quality level value, can compute a simplified version of such mesh. This simplified mesh would have its number of vertices cut in accordance to the given quality level value but while still retaining, as much as possible, the general shape of the original mesh. However, since we only have information about the shape of

a complex object after its creation, we can only apply this technique at this moment. Also, since we might have to apply this technique several times for the same object to create different levels of detail for it, we have to make sure the algorithm is fast enough to not excessively increase generation time. For the concretization of this technique, an external open source library, UnityMeshSimplifier[10] was used as it fits our needs of being fast and producing good quality results. It is important to note that this acceleration technique highly depends on the design model composition, i.e., the higher the number and complexity of the composing objects of the scene, the better is the performance improvement. However, the additional performance benefits come at the cost of a slight increase in generation time and memory usage, so we ultimately leave up to the user to decide if its usage is desired. Through the use of our UI, a user can easily enable this feature, as shown in Figure 3.16.

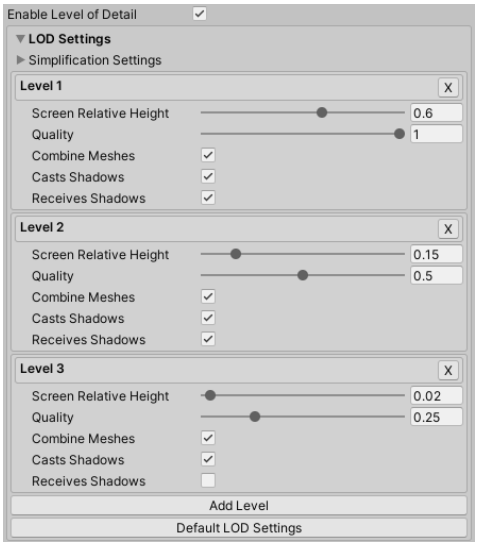


Figure 3.16: Representation of the level of detail user interface. When the toggle button on the top of the image is enabled, the menu below is shown. This menu allows a manual configuration of the LOD options.

As this feature increases the generation time and is only effective on designs which have a large number of complex objects, we only recommend its usage at later stages of the architectural design process. As such, this is disabled by default, when the Concept preset is selected, since during a design experimentation phase a faster generation is preferred.

The remaining matter left to tackle about LOD is the selection step. The selection metric determines when to switch between different levels of detail. Our selection metric of choice is object's relative size to the user, i.e., how big the object looks in the screen of the user. Using this metric, large objects that even at longer distances have good visibility, such as buildings, are not a target to apply LOD techniques, whereas other smaller objects, although closer, are chosen for it. The optimal configuration for these settings, however, is hard to determine as it varies from one user's computer to another, so we provide a

[10] UnityMeshSimplifier: https://github.com/Whinarn/UnityMeshSimplifier

default configuration as shown in Figure 3.16, with the possibility of manual configuration, like modifying, adding or removing levels of detail. This default configuration entails three different levels of detail. On the first level, the objects are processed in full quality up until the object's screen relative size is bellow 60%. On the second level, the polygon mesh composing such object is cut by half of the complexity up until its screen relative size is bellow 15%. On the last level, the quality of the polygon mesh is cut to a quarter of the original quality, while retaining the general shape of the object, and when the screen relative size of such object is bellow 2% the object is no longer processed. Additionally, at the last level, the object no longer receives shadows from other objects, further improving performance. All these levels can be observed in Figure 3.17, where we can observe that the quality difference between the levels is not noticeable.

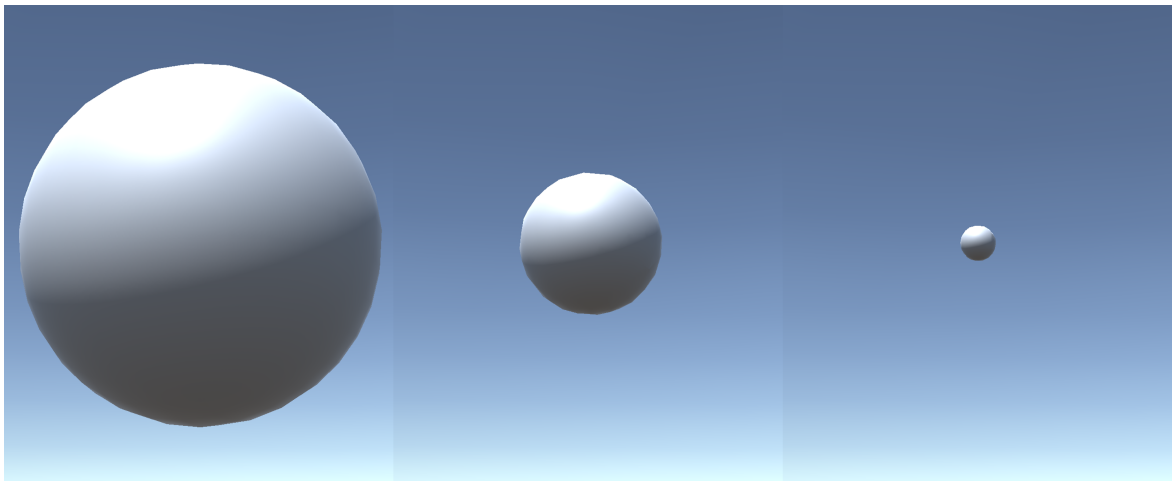


Figure 3.17: Sphere generated using Khepri with different automatic generated levels of detail using the default configuration. The sphere on the left represents the first level of detail, at full quality and at exactly when the object's relative size would occupy 60% of the screen, close to transitioning to the next level of detail. The second sphere, in the middle, represents the second level of detail, at half the quality and at exactly 15% of the screen relative size. Lastly, the sphere on the right represents a third level of detail, at a quarter of the quality and at exactly 2% of the screen relative size. Note that this image was cropped and scaled to fit this document, so the object might seem bigger than when observed on a full screen.

3.3.3 Additional Performance Acceleration

Throughout the development of this visualizer, we were constantly investigating performance struggles using Unity's built-in profiler. By testing our solution, even midway development, with real scenarios, using complex AD programs coded by users of Khepri, we were able to identify several possible performance improvements.

One of these performance improvements was to declare all generated objects as static objects, since most design models are static, so we can take advantage of the Unity's static batching, as described

in section 3.2.1. Another improvement was to reduce the performance weight of our communication channel. One flaw that Unity gets often criticized is how its architecture mostly only uses a single core of the CPU. In particular, as most Unity operations can only be executed in the main thread, which compose our object creation operations, our communication channel would need to process these Khepri requests from the client on such thread, creating a frame overhead during navigation when processing multiple requests at once. We then optimized our communication channel's introducing a way to control how many operations can be processed at each frame. This mechanism will be better explained in detail in section 3.3.9. As the communication channel on its own would slightly hamper performance, we also included a way for users to manually pause the communication channel when their design is already fully generated. Unfortunately, this cannot be done automatically as only the user knows when the design is completed and no more operations will be sent to our visualizer.

Still, the problem with the biggest impact on performance was inherent in the AD workflow itself. One downfall of the AD workflow, in terms of performance, is that a complex design model might require the use of several small and simple objects. This is encouraged by one of the AD's core features, the parameterization of the design model. For AD architects to parameterize their design models, they must decompose them into several logical portions. This decomposition not only helps architects make their program more comprehensible to the human eye, but also benefit at latter stages of the architectural design process, when design variations must be made. As an architect makes a design more parameterizable, its decomposition gets finer and so the number of small objects increase. This, however, creates a huge strain on the CPU, as the project grows, since it needs to process, batch, and send a larger number of draw calls containing small objects to the GPU to compute. While, on one side, the GPU is able to quickly complete its task, since the objects are simple, on the other side, the CPU is getting overloaded by the sheer number of objects. This work imbalance, imposed by the AD workflow, creates a CPU bottleneck which worsens as a project grows.

One way to solve this bottleneck is to reduce the amount of draw calls required. A way to reduce the number of draw calls without removing objects from the designs is to merge the objects' polygon mesh together, resulting in fewer but more computationally expensive draw calls. Unity itself has support for polygon mesh merging operations, which we took advantage to create a feature called Design Merge. However, it still remains the question of how we should merge a design. Unfortunately, the task of logically dividing and merging a design in a fair way is not simple. This division has to be done in such way as to not only balance, fairly for every user, the work between the CPU and the GPU, but also guaranteeing that we will not disrupt the efficacy of other features such as Occlusion Culling and Unity's built-in shadow calculation algorithm, both requiring a degree of element division to work properly and performantly. Currently, our implementation only supports the merge of a design as a whole. We will discuss the performance benefits and drawbacks of such implementation in section 4.1.3. Design Merge

can only be manually enabled by the user, through our UI, if the Showcase quality preset is selected, since it requires a completed design and involves a waiting time.

3.3.4 Traceability

The establishment of the AD approach imposed a completely novel way of designing buildings in architecture. However, introducing programming into the workflow of an architect requires a great learning effort from them. We have previously mentioned the importance of increasing program comprehension to make designing a more natural task. In this section, we introduce a feature aiming at improving the program comprehension.

Nowadays, commonly used IDEs provide traceability features to improve programming productivity, either in form of code debuggers, to help programmers identify issues with their code, or in the form of code navigation features, to help the programmer, for example, jump to method declarations, or identify the callers of a certain method. This same concept could be applied in our backend, specifically, to aid the coding architects by showing the relation between the code and the generated result. In other words, we would like to be able to trace which function generated a certain design element and vice-versa.

This bidirectional traceability brings a plethora of benefits to the workflow of an AD architect. Since AD programs of a building are meant to be shared with many other participants in an architectural project, one might not have full knowledge about the code's structure, leading to difficulties in understanding it. Additionally, in the event of a design flaw being spotted while visualizing a design model, it is necessary to find the fraction of code that is responsible for the flaw.

To support this feature, Khepri is able to keep track of all the construction elements that were requested to be generated, as well as the respective lines of code responsible for such request. To make this feature compatible with our backend, a connection needs to be created between the data present in the Khepri client and the data present in the Unity server. Furthermore, a way to select and highlight objects during navigation needs to be developed.

Regarding the data connection, on the backend side we keep track of each generated object by labeling them with a unique identifier. When a new object generation request arrives to our backend, we communicate its respective identifier to Khepri for it to associate and store along with its own representation of the object, as illustrated in Figure 3.18. Note that that the backend will never exchange the Game Objects directly with Khepri, and Khepri will never exchange its Shapes with our backend, as these are data structures only known by each one of them. Instead, when referring to them in the communication, only the attributed identifiers are used. If users wish to select an object on our backend to know which lines of code in the client are responsible for its generation, they must first run a Khepri operation to start the selection process on our backend. Then, we allow users, while navigating, to simply utilize their mouse cursor and press on the pretended object with the left mouse button to select it. Alternatively, if

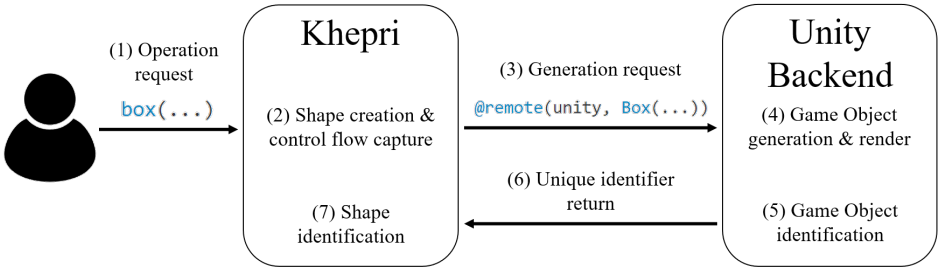


Figure 3.18: Traceability feature flowchart. The starting point is when a user sends a construction operation to Khepri (1), through their client.

required, users can also select multiple objects at the same time.

Regarding the implementation of this selection process, when a user presses the mouse button we fire a raycast[11] through the user's view frustum. We make sure to calculate the correct ray that goes through the object that the user's mouse cursor was on during selection by converting the position of the selected point on the screen from screen space[12] to the correspondent coordinate in world space,[13] both in the near plane and in the far plane. Then, we can create a ray that goes through these points, ensuring that it will collide with the object the user wanted to select. Finally, our backend only needs to map this Game Object to the corresponding identifier and query the client for the code location that generated such object.

After the selection process, Khepri can then highlight both the code in the client and the object in the Unity server. Regarding the implementation of the object highlight feature in our backend, we can use outline shaders to accomplish this task. Unity features a built-in outline shader, used by the Unity Editor to mark selected Game Objects. However, as this shader was not meant to be used outside the Unity Editor, it works incorrectly when used along with VR. To render a view in VR, it is necessary to render it twice with slightly different perspectives to create a stereoscopic view of the scene. Since this shader uses the screen space to draw the outline and there are two different screen spaces in VR, it would compute the wrong outline. As we require the ability to select and highlight objects, even when navigating in VR, this kind of shaders is unfit for our purpose. We instead choose an implentation using world space outline shaders. The problem with these type of shaders is that, as they are built upon the world space, they are dependent on the complexity of the object they are outlining and often do not work along with the LOD feature.

For our backend, we require a performant outline shader that can be applied to any kind of generated object. We were able to achieve this by using a third-party outline plugin called QuickOutline.[14] This plugin allows us to apply a shader to any object that quickly computes an outline, even for multiple

[11] Raycasting is a feature present in many Game Engines used to check for object collisions, using a ray with a defined origin point and direction vector to test if it intersects with any object.
[12] Screen space refers to a 2D coordinate space where a 3D rendering is projected upon when displayed on a computer screen.
[13] World space refers to a 3D coordinate space of the world where a 3D rendering is built upon.
[14] QuickOutline: https://github.com/chrisnolet/QuickOutline

objects at once, which is necessary when using the LOD feature. As Unity supports multi-texturing, we are able apply this shader along with an object's own existing material. By default, this shader will compute an orange outline. However, since its visibility is dependant on both the applied object and the surrounding's material, a customization menu in our UI was implemented, as shown in Figure 3.19, where users can modify the outline's line color and size. In the figure's case, as the highlighted portion of the pagoda is made of wood, an orange outline would not be ideal, so instead, its color was modified into a more contrasting one.

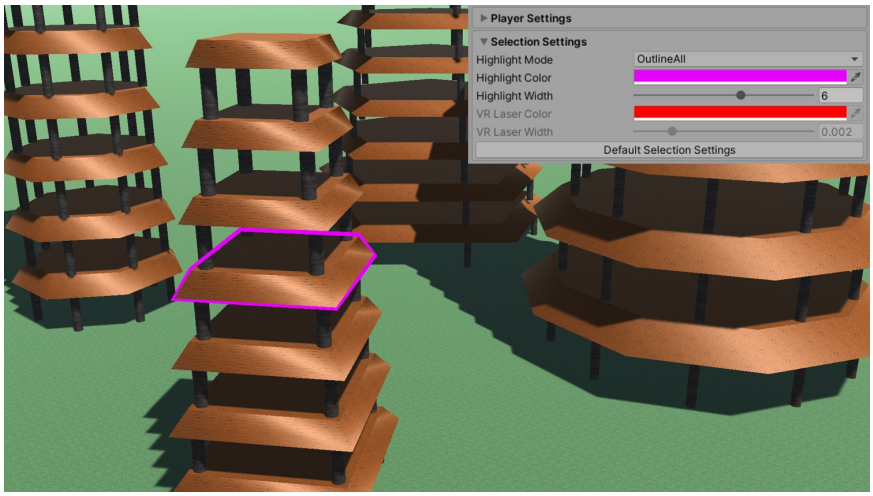


Figure 3.19: This figure illustrates a pagoda city generated in our backend. One of the levels of the pagoda was selected and outlined. On the top right corner of the figure we can see the portion of the interface responsible for customizing the outline. The last two settings are only available when navigating using VR.

This plugin also includes five different types of outline, as described in Figure 3.20. These modes were included for their usefulness during the designing process, for instance, when architects need to check if the generated objects are placed on the intended positions, or overlapping with other objects.

3.3.5 Layers

In this section we define the concept of layers, commonly used by CAD and BIM programs, and describe its importance in the digital era of architecture, use cases, and its implementation on our backend.

Layers are sets of elements which compose a digital design model, used to logically divide it according to the needs of the architect. This division of the design model's elements helps the architect organize the design in such a way as to aid the completion of certain tasks such as, but not limited to: showing and hiding certain layers to reduce the visual complexity of a design when evaluating a portion of it, coloring different layers with different colors to highlight the results of an execution of analysis programs over the design, grouping elements with similar physical characteristics in the same layer to easily change their material at the same time, etc. More specificaly, architects can create layers for a finer or

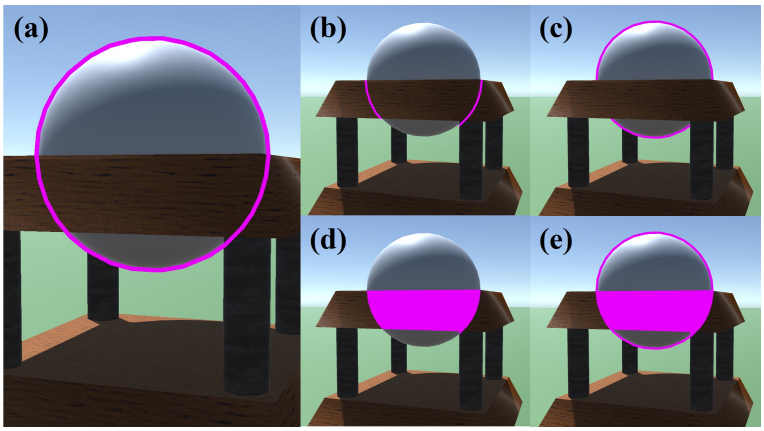


Figure 3.20: Five different types of outlines: (a) default option which outlines the whole object, (b) outline only the hidden portions of the object, (c) outline only the visible portions, (d) silhouette of the hidden portion, and (e) outline of the visible portion and a silhouette of the hidden part.

a coarser division of their design, like: dividing the building's floors into distinct layers, separating the building layout from the interior decorations, aggregating all door elements in the same layer, etc. The organizational value of layers is of big relevance for digital designs, hence Khepri supports operations to create layers and manipulate them. Naturally, these operations must be supported by our backend as well.

Its implementation is pretty straightforward as this feature can be replicated by adding operations to manipulate the scene graph of our design. Users are able to programatically create and name custom Game Objects which will serve as layers. Generated objects will then be created as children of their respective layer. Changing the visibility a layer can either be done programatically, by using Khepri operations in the code, or manually, in the scene graph section of Unity Editor's UI. When creating a layer, users can also declare the color for the layer's objects in their code. Figure 3.21 illustrates a building whose floors and facade are divided using layers. Additionally, the last floor was declared to be colored in cyan by the user's code. Regarding the implementation of the layer coloring feature, we developed a custom Component that allows us to color a whole layer when added to it. This is done in such way to detect when new objects are added to the layer containing this Component, coloring them accordingly, i.e. if a layer was targeted for coloring, all further generated objects on it will also get colored. This Component also works correctly when added to an existing uncolored layer that already contained generated objects. In such case, this Component would recursively search all the children objects and color them. To color an object, the Component creates a new material with the respective color, to be shared with all the colored objects, that will be applied above an object's current material without replacing. This way, we can revert a colored layer by removing this material from the colored objects, preserving their inital state.

By taking advantage of the benefits brought by the use of GEs, we can go much further to make

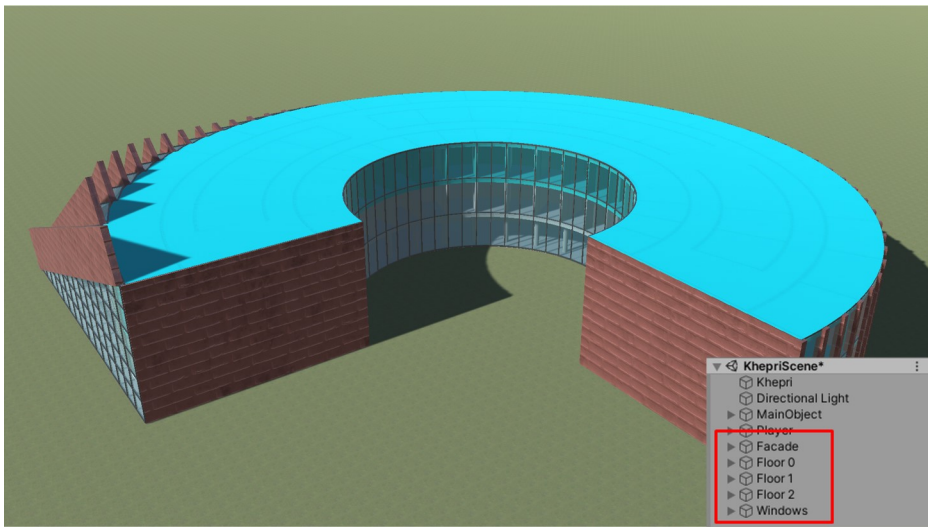


Figure 3.21: Render of a design divided by layers where its last floor was colored in cyan. On the bottom right side, the scene graph UI, displaying the existing layers of the design.

this feature better fit the AD approach and augment it for other purposes. During the earlier stages of the architectural design process, architects want to produce several design variations. As our backend allows architects to easily create and visualize design variations right away, we can use layers to store all the produced variations, each in a different layer representing a moment of the generated design model. This way an architect can easily roll back a change if required, or quickly cycle between all the variations without interruptions in the visualization, by hiding or showing layers through Khepri operations, to provide an improved judgment of the differences.

3.3.6 Scene Manager and Standalone Build

Previously we have discussed about various features aiming at optimizing the performance of our backend. Some of those features could bring us performance benefits albeit having some sort of processing waiting time. For instance, Occlusion Culling and Design Merge, discussed in section 3.3.2 and section 3.3.3 respectively, are examples of features that require prior calculations to be used, so we only recommended their use at later stages of the architectural design process, on a completed design. By using these features we were able to provide users with a fast visualizer capable of good interactivity even on complex and large designs. Unfortunately, the addition of these features brought a new unwanted task to the workflow of AD architects. Having to optimize the performance of an AD model each time they want to visualize and navigate on it can be a time-consuming task if repeated regularly.

In the AD methodology, a design is a computer program. This way of representing a design, as opposed to having directly the corresponding digital 3D model, is exactly its strong point and has proven to be very advantageous throughout the architectural design process, as we have previously discussed

on section 1.1. However, such indirection always incurs some sort of generation time when we need to visualize a design. The end product of the architectural design process is a design to be visualized by the project stakeholders, including clients. Currently, to optimally showcase a complex design in our backend to a client, we have to go through both the generation and the performance optimization process.

The AD methodology benefits us most when we need to apply modifications to a design to create variations. However, if a design is indeed completed, we no longer have the need to modify it. Therefore, at this stage, we can release ourselves from this methodology and figure out a way to shorten the waiting time to visualize a design model in our backend. This section defines a solution to tackle this problem, split into two features, called Scene Manager and Standalone Build.

To break free from both Khepri and the AD methodology, we have to be able to convert an AD model from its program description into a compatible data structure used by Unity and save it to disk for further accesses when we need to visualize the design once more. Using the Scene Manager allows us to convert such program into a Unity Scene and save it to disk.

During regular usage of our backend, when we establish a connection with Khepri and start generating a design, we already do this conversion. However, at this point we are unable to save such generated design Scene to the disk because the Unity Editor is currently on play mode. The Unity Editor, at any point, is in one of two operation modes: edit mode and play mode. When we open a project in the Unity Editor, it always starts in edit mode, where a user can use editor-specific features to create, modify, and debug the current Scene, but most importantly, save such Scene to the disk. However, during this mode, scripts and Game Object's Components are disabled, meaning that the Scene is currently in a paused state in which we are unable to act upon. On the other hand, during play mode, scripts and Components are active, thus the state of the Scene is able to change according to them. Hence it is in this mode where games made in Unity run, and, in our context, where we are able to navigate in designs. When we use our UI to establish a connection with Khepri we change the Unity Editor from edit mode to play mode to be able to run our navigation scripts. In play mode, although in a limited way, we are also able to modify the current Scene, as we already do when we generate a design during this mode, however, any modifications made to it are temporary, thus will not affect the Scene present in edit mode or be able to be saved. This happens because, when switching to play mode, Unity loads the Scene in a separate temporary instance, which is deleted when we we switch back to edit mode to preserve the initial state of a Unity project. Therefore, to be able to save our Scene, we must generate it in edit mode first. For this effect, we have implemented an alternative way to establish a connection with Khepri outside play mode. The only trade-off is that during this edit mode connection with Khepri we are unable to navigate on our design, as the navigation scripts are disabled during edit mode. However, when the design generation is finished, we can shut down the communication with Khepri, perform any

necessary performance optimization, and switch back to play mode whenever we need to navigate on it. Figure 3.22 illustrates the buttons on our UI responsible for this behaviour. Finally, by having the design

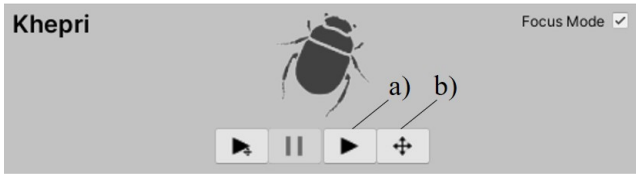


Figure 3.22: When the Showcase preset or the Advanced preset is selected, these new buttons on the right become available on our UI. Button a) starts a connection with a Khepri client on edit mode and button b) switches to play mode to start navigating on the current Scene without a Khepri connection.

generated in edit mode we are now able to save it to disk using our interface, as shown in Figure 3.23. Users can also load previously saved Scenes using the ”Open Scene” button or even clear up a saved Scene using the ”Reset Scene” button.

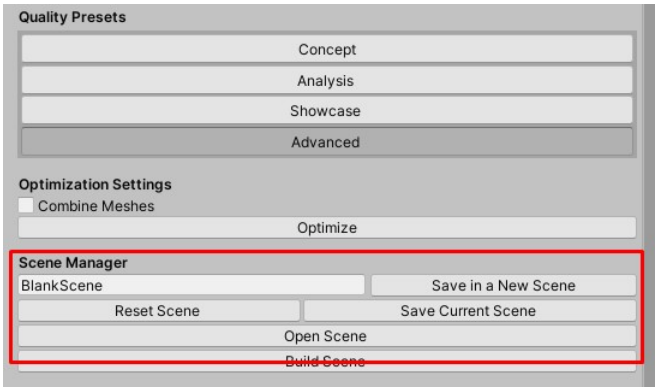


Figure 3.23: Highlighted in red, the Scene Manager interface menu, only available when the Showcase or Advanced preset are selected.

By being able to convert an AD program into a Scene and save it, we have successfully removed any dependencies with Khepri and we can start visualizing and navigating on it at any time without further generation or performance optimization waiting time.

However, if the sole purpose of detaching the backend from Khepri was for visualization and client presentation, we can go much further and detach from the Unity Editor itself. This is where the second feature, Standalone Build, comes into play. By taking advantage of Unity's power, we can build a self-contained executable file of the Scene to freely navigate on it anytime and anywhere, without the need to install the Unity Editor or any other dependency. Our executable file will also have a better performance as it is no longer attached to the Unity Editor, which frees us from all the debug information provided by it. Unity has support to create executable files for a plethora of platforms such as, for example, Linux, PlayStation 4, Nintendo Switch, etc. Despite that, for simplicity purposes, our interface, illustrated in Figure 3.24, only creates an executable file for Windows 64bits. Additionally, if the VR option is enabled,

users can even create executable files of the Scene to be navigated using VR hardware.

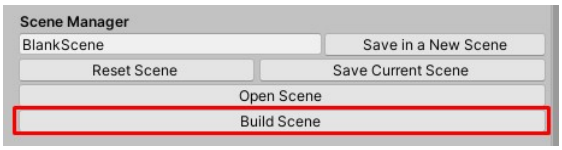


Figure 3.24: The last button of the Scene Manager menu is used to easily create a standalone executable of the currently loaded Scene.

3.3.7 Per Project Assets

One of the reasons why we chose to build our backend on top of the Unity Editor was to take advantage of its ability to import and manage the assets of a Unity project. On section 3.2.2, we have discussed the importance of these assets in an architectural project. Aside from providing users with a default assets library, we also offer them the possibility to expand their library with custom assets, as requirements between projects may vary. Nonetheless, from our experience often the default library does not suffice, requiring the regular use of custom assets on every other project. Since these custom assets will persist in the backend of a user, after a good amount of architectural projects, the backend can become quite bloated with numerous assets that were probably only used once or twice, affecting the startup time of the backend. To counter this issue, in this section, we introduce a feature which gives users better organizational control over their assets. At the time of development, Unity did not have any feature that allowed us to smartly organize our assets.

If the custom assets are dependent upon a design model, a possible solution would be to keep them stored next to that said design model's description, outside Unity, and only import them to Unity when we plan to generate such model. However, to import an asset to Unity we must move them to our project folder, which completely negates our effort of keeping the assets separated and organized. To solve that, we propose a new workflow with the use of symbolic links.[15] To support the creation of symbolic links inside our backend, we used a third party extension called unity-symlink-utility.[16] The operation of symbolic links is completely transparent to Unity so we can take advantage of that and use them to completely separate a project's assets from the backend while still retaining the possibility to import them later on. The new proposed workflow, as a mere guideline to support a better organization in the long term, entails the following: (1) when architects wish to start coding a new AD project they must create a folder, separate from the backend, to contain their code of the project; (2) inside that project's folder they must create two folders called "Materials" and "Prefabs" which will contain their custom assets; (3) inside the backend they must create a symbolic link inside the assets' folder pointing at their project's

[15] Symbolic links are a feature present in nowadays' operating systems which consist of files or folders with references to another file or folder.

[16] unity-symlink-utility: https://github.com/karl-/unity-symlink-utility

folder; (4) then, they can start importing their custom assets inside the backend and use them; (5) when they no longer need to visualize this project, they can remove the symbolic link, freeing up the backend from such assets. This way, we are able to keep our backend more organized and whenever we want to visualize a previously created project we just need to add back the symbolic link and all the custom assets related will be imported back in. Following this workflow is not mandatory and will not affect the architectural design process in any way, however, this feature was added to provide architects with the right tools to organize their projects and allow our backend to scale throughout multiple projects.

The next section will be dedicated to VR, where we discuss the impact of this new technology in the architectural design process. Despite not being a requirement for the current AD process, VR comes as a perk of the GE implementation. This technology is capable of greatly expanding the experience of architectural visualization, thus the opportunity to couple it into the architectural context cannot be overlooked.

3.3.8 Virtual Reality

Besides the enumerated advantages GEs bring to the AD workflow, they also allow for integration with the current state-of-the-art methods of visualization such as VR.

As many other fields currently exploring the potentialities of VR, architecture is an area that can benefit a lot from these technologies for visualization. Whyte [Whyte, 2003] identified three main strategies for the application of VR in the AEC industry: (1) VR as part of the design process, where designing, prototyping and simulation of the construction process occurs in VR. By immersing the architects in their creations, we evoke a better sense of occupancy, which allows for a better understanding of the design and error detection. We can also augment the experience with auditive information and the ability to interact with the surroundings, improving the immersion further. While inside the design, one can monitor the implications of different designs. Next, (2) VR can be used as a customer interface, for client interactions, storytelling and project selling. Communicating a design to a client has always been a challenging step in the architectural design process, where architects invest a lot of effort. Clients are rarely experts in architectural design's typical representations mechanisms, plans and sections, and thus prefer more realistic representations to which they can relate. VR can take client experience with the projects to a whole other level, hence helping architects sell their ideas. Lastly, (3) VR opens up more possibilities for communication, which include, for instance, new methods for remote collaboration in the design process, where peers, located at different parts of the world, can join together in the same virtual environment and work in a design as if they were together in reality.

VR is a technology growing rapidly in the current days, and we can achieve all these benefits in an inexpensive way. However, this comes at a cost. VR requires a lot more computational resources in order to maintain an high frame rate. If this cannot be provided, the user might experience motion sickness,

thus breaking the immersion. As such, for a smooth experience with VR, users have to take advantage of the performance optimizations explained previously and tune the settings of our UI accordingly.

Currently, the VR industry is in a state of rapid development, meaning that new VR hardware devices get released often, and the industry has not yet reached a common standard either in form of software or hardware.

To combat the lack of interoperability, Steam has being allocating efforts into developing a plugin called SteamVR. Steam is a popular digital game distribution service which developed this plugin to provide game developers with a common API to integrate any kind of VR hardware, present and future, with their games. SteamVR currently supports popular hardware devices such as Oculus Rift, HTC Vive, Valve Index, and many others. Since it is continuously being developed and supported, it will also allow integration with yet to come VR hardware. To take advantages of these benefits, we decided to integrate our backend with SteamVR.

Using SteamVR, architectural designs can be further explored in VR. If users have access to VR hardware and a workstation that supports it, they can simply enable the use of VR during navigation using our UI, as illustrated in Figure 3.25, and the backend will load the SteamVR plugin.

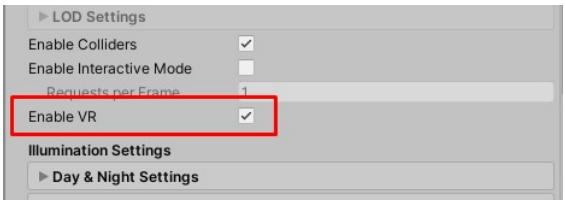


Figure 3.25: Portion of our interface where users can enable VR, through a toggle button.

We allow architects to navigate in VR using all the navigation modes previously described in section 3.2.3. As for input mapping, since there are still no fixed standards in this industry, each VR hardware has its own type of controllers. Nonetheless, SteamVR provides an easy to use input mapping system, where we just need to declare our possible commands for our application and map them to all kind of controllers using a visual interface, as illustrated in Figure 3.26. Using this interface we can setup in each controller how each button should behave, using our previously declared movement commands. For example, if the user is using an Oculus Rift, the navigation controls are the following: the user's head orientation to control the camera's orientation, left joystick to control the camera's horizontal orientation, right joystick to control the movement, "A" button to change navigation modes, hold the right trigger button to move faster, press on the left joystick to move downwards if on free-fly mode, and press of right joystick to jump or move upwards if on walk or free-fly mode respectively. We note that fast movements in VR, especially in free-fly mode where movement is not natural, can provoke nausea to users new to this technology. Additionally, we also allow users to select objects in VR, as described in section 3.3.4. However, as we no longer have a mouse cursor to point at objects, laser pointers are used instead by

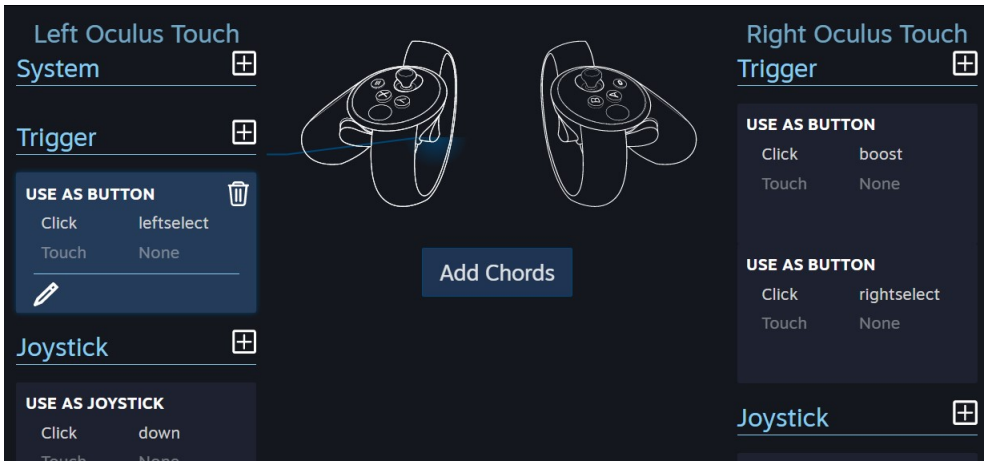


Figure 3.26: Screenshot of the SteamVR's input mapping interface for Oculus Rift's controller, Oculus Touch.

taking the position of the controllers, as illustrated in Figure 3.27. As seen in the figure, the outline shader does work properly even in VR. To enable the laser pointers, users just need to hold the left or the right grip button of the controller. The grip button of both controllers can be accessed using the middle finger of each hand when holding the Oculus Rift controllers. Then they can select the objects using both trigger buttons of the controller. Similar controls can be found on a HTC Vive controller,

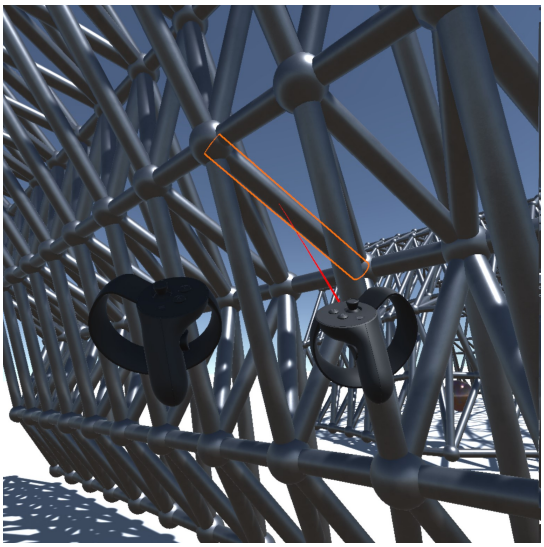


Figure 3.27: Object selection in VR with the use of laser pointers, on a generated design in our backend.

however, as this specific controller does not have joysticks and instead has touchpads, all the movement and camera controls are mapped there instead.

The practicality of this technology during the architectural design process benefits most at later stages, when a design is mostly completed. Giving users a better sense of occupancy allows them to evaluate a design in ways they could not do before. By immersing into a design, we get a perspective

that allows us to, for instance, evaluate the natural illumination of a room, examine the safety of railings, ponder upon the scale of a division, etc.

In regards to the other stages of the architectural design process, since this technology requires users to wear a headset, it is impractical, at first glance, to program a design while using such wearable, either because it hinders the use of a physical keyboard during its use or because it is bothersome to repeatedly remove the headset when we need to program and put it back on when we need to visualize the results of the program. We have, however, explored other possibilities for allowing the task of programming inside this new medium, which opens doors for new opportunities to further augment the AD methodology with the power of VR [Castelo-Branco et al., 2019]. We propose a new workflow named Live Coding in Virtual Reality (LCVR), where VR complements the AD workflow to transform the designing task into a more interactive one, which may improve the architect's ideation process. In this new workflow, architects, while immersed in VR, work on their designs with a visual representation of it and its respective code side by side. This way, users are able to promptly apply changes to their design's description and witness the materialization of those modifications around them, boosting their creativity and judgment. Nonetheless, this new workflow still faces some challenges.

The resolution of nowadays VR headsets is still very limited, which affects the readability of text in VR. This may improve with newer models, but, until then the immediate solution would be to increase the text size. Next, this workflow is faced with the aforementioned text input problem. This is one serious problem, as typing is one of the most important requirements of the AD workflow. However, our access to a physical keyboard gets limited with the use of a VR headset. To overcome this issue and attain the benefits of LCVR, we suggest the use of many other alternatives at the cost of some typing efficacy, such as, a virtual keyboard, voice input, handwriting, or resort to visual approaches to program.

We did not further pursue the exploration and implementation of these alternative typing mechanisms. Nonetheless, we can already replicate LCVR with our implementation so far. In our case, both the Oculus Rift and the HTC Vive VR headsets are accompanied with software that is able to provide the backend's two missing requirements to achieve LCVR, desktop mirroring and a virtual keyboard. Figure 3.28 illustrates our attempt of replicating the LCVR workflow in our backend with the use of the software provided by the Oculus Rift. In this example, we blindly used a physical keyboard as an input mechanism, which may not be ideal for users accustomed to typing without looking at the keyboard.

This might be an imperfect solution as we are not only relying on a piece of software provided by these two headsets, and others might not have these features, but also, we are not proving a good enough variety of input mechanisms to aid the execution of this workflow. However, this gives us the opportunity to at least provide architects with the possibility of exploring the potentiality of LCVR during the production of their designs, or even, to show one of the many potentialities of VR itself when applied to the field of architecture.

Figure 3.28: Design generated on our backend using LCVR. On right side of the image there is a window on top of the design with the its code.

3.3.9 Interactive Mode

In this section we define one last feature, name Interactive Mode, which originated as a solution to a problem that the use of LCVR brought upon. When users of our backend desire to generate a design on it they just have to run their design program on their Khepri client. Consequently, the Khepri client will send a multitude of operations to our backend, responsible for the generation of the design. To cut down on generation time as much as possible, we simply allowed our backend to receive and process all these operations without limitation, even if they would cause our backend to be in overload, which greatly affected the navigation's interactivity while the backend is still processing the sheer number of operations. Up until now, this was an acceptable behaviour as users of Khepri would also expect this kind of unresponsiveness during generation from other visualizers. So far, with our backend, we allowed Khepri users to experience their designs in VR for the first time and consequently try out the LCVR workflow. However, we then found out that this generation unresponsiveness, previously left unchecked, would cause a problem when coupled with VR. When users desired to visualize their design being generated from scratch during LCVR, as this issue would severely decrease the frame rate during navigation, this often caused motion sickness.

A solution could pass by limiting the amount of operations our backend should process during each frame while on VR. However, this solution would consequently increase the generation time each time we desire to generate something in VR, even when there are other use cases outside LCVR, where we are not immersed during generation and simply just want the fastest generation time to later navigate on the design in VR. We then decided to adopt a different strategy to tackle this issue.

A feature called Interactive Mode was implemented, where limiting the processing of operations only occurs when this mode is enabled. Interactive Mode can be enabled in two different ways, through our UI or the design code. The latter was deemed important to implement to satisfy a scenario where users of Khepri, although require the fastest generation possible for the design itself, implement methods to be called during LCVR. These methods may apply large modifications to their designs, requiring the use of Interactive Mode only during their execution. An example of this usage was described in section 4.2.

As computer specifications may vary from user to user, we also allow them to modify how many operation requests can the Interactive Mode handle in each frame through our interface, as illustrated in Figure 3.29. The higher the number of requests per frame, the faster will be the generation time, at the cost of a less smooth frame rate.

Figure 3.29: Portion of the UI relative to the Interactive Mode toggle.

4

Evaluation

Contents

4.1 Performance Benchmarks 59

4.2 Practicability Analysis 69

In this chapter, case studies will be used to demonstrate our fast and interactive visualizer for Khepri. More precisely, we will perform (1) an evaluation of our backend's performance and the incremental impact of the implemented performance-increasing features, and (2) an analysis over the practicality of the utility features, such as Layers, VR, and Interactive Mode.

We will only perform a formal evaluation for performance as this metric enables us to easily quantify the main goals of our solution. We initially proposed to provide AD architects with a visualizer capable of achieving better performance than the visualizers they used. Beyond that, we took advantage of the powerful framework that Unity provides to extend our work and improve the architects' AD workflow with many utility features. However, it is not a simple task to quantify the benefits of these features. Some of these utility features were not present in past visualizers used in the AD workflow, such as the ability to create a standalone program containing a design, discussed in section 3.23, giving us no ground to effectively perform comparisons. Also, we find that the benefits of those features are too subjective to be formally evaluated and would require an extensive qualitative study of how much our backend improved the overall AD workflow when compared to the use of other backends, i.e., did the task of programming a design become easier, did it improve creativity, did it make the architectural design process faster. Instead, we would rather focus our efforts into obtaining objective results for performance and only perform a subjective analysis over the utility features.

4.1 Performance Benchmarks

Our benchmarks will consist of: (1) measuring the performance of our backend under the load of a complex case study design, the Astana National Library (ANL), over different scenarios, and (2) performing an analysis on the impact of the various performance-increasing features, such as LOD, Occlusion Culling, and Design Merge. In order to effectively measure performance we need to register, not only, the current frame rate, in FPS, but also, the current view's complexity. We need to mind about the latter because of the innate work of View-Frustum Culling, meaning that the current frame rate of our backend is highly influenced by the complexity of the design's portion that is currently being visualized, and not the whole design itself. For instance, if the user's view is pointing at the opposite direction of the design, where nothing can be seen, no matter how complex the currently generated design is, we will always register good performance values. We measure a view's complexity by the number of triangles which compose the visible objects of the design.

The chosen case study for our performance benchmarks, the ANL project, was designed by the Bjarke Ingels Group. This design projected a large building that would occupy an estimated thirty three thousand square meters and would house a library and various multifunctional spaces in its multiple floors. Figure 4.1 shows a render of the exterior and the interior of this project. As seen in the figure, its

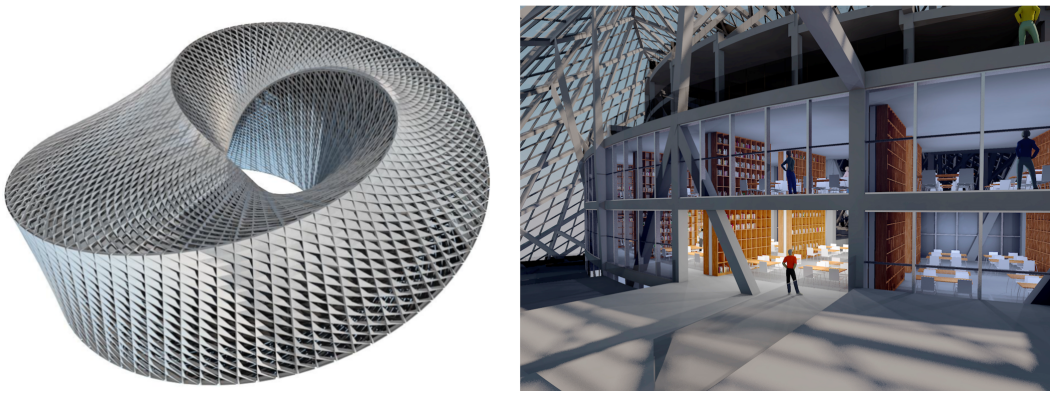


Figure 4.1: On the left, a render of the exterior facade of the ANL and, on the right, a render of the interior library.

shape imitates a möbius strip, which makes it, not only, aesthetically unique, but also, a perfect candidate to apply the AD methodology. As such, this project was redesigned and made parametric using AD by Castelo Branco [Castelo Branco and Leitão, 2017].

The AD version of ANL is described by more than two thousand lines of code, which generate more than forty thousand construction elements, including: (1) the facade, composed by the facade glass, steel frame, and the photovoltaic panels; (2) the interior structure, composed by floor slabs, walls, glass curtain walls, columns, beams, staircases and railings; (3) the interior assets, composed by bookshelves filled with books, tables, chairs, elevators and lights. The complexity of this design model along with the fact that it is a real architectural project makes it a solid case study to benchmark our backend.

To ensure the reliability of the registered performance values, all the evaluation tests will be performed in reasonable navigation scenarios, by following pre-defined routes on the generated design. These different routes will allow us to obtain reliable sets of data which will differ according to the part of design that we are navigating through. Additionally, we can cover the performance of our backend over the whole design in a fair way as some parts of it can be more complex than others, for instance, as we already expect from the View-Frustum culling work, the performance of the exterior of the design can be much worse than of the interior, where we only have a limited view of the scene. For that effect we defined four routes to navigate through the generated ANL that cover most navigation use cases. Figure 4.2 shows a rough illustration of these four routes in which we will perform our evaluations. For representational simplification reasons, all of these routes are illustrated as full circuits but some will not perform a full circuit. For the exterior routes, route a) will encompass an aerial tracking of the whole building, in which we can already predict that it will be the one to perform worse in frame rate as every object of the design will be inside the viewer's frustum. On route b), although the camera is still outside the building, it will only visualize a slice of the model at each moment. As for the interior routes, route c) will perform a walkthrough on an inner corridor of one of ANL's floors. This walkthrough will have a partial view of the libraries, which are the parts of this design with the most assets, in particular, the

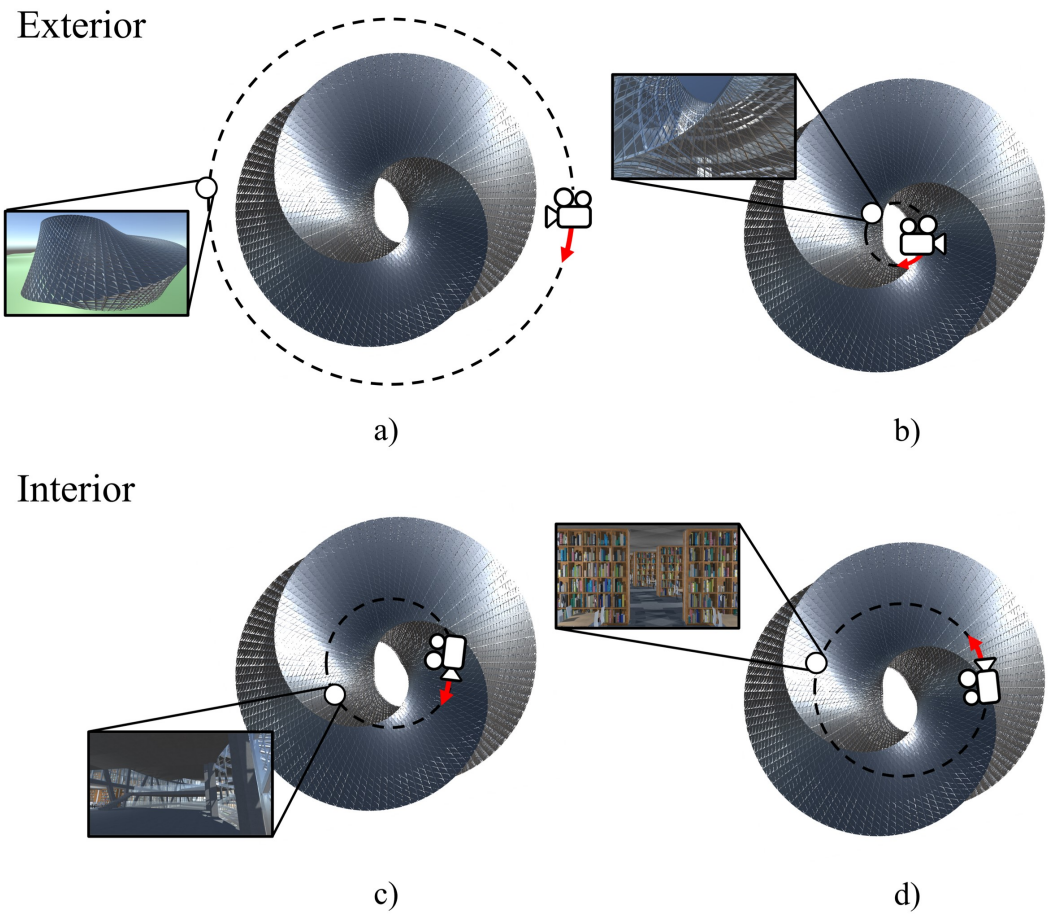


Figure 4.2: Four different routes, two in the exterior of ANL and two in the interior. The dotted circle on each representation of ANL represents the path of each route, the red arrow represents the direction of the navigation, and the camera icon represents the viewing direction. For the exterior routes, the camera points perpendicularly to the direction of movement, whereas, in the interior routes, the camera points at the direction of the movement, simulating a walkthrough.

bookshelves, which are filled with several books using different materials. Finally, route d) will perform a walkthrough inside the libraries of a floor. All these routes will be discretized into several points, in which we will perform the data collection of the frame rate and the complexity of the view. At the end of a route we will perform the average of those values.

Previously, to create render images of this AD description, the architect that created this building used ArchiCAD's render engine, CineRender. To serve as a performance comparison to our backend, we performed an evaluation with CineRender over the ANL AD model, to know how much time it would take to complete each of the mentioned pre-defined routes. Table 4.1 illustrates the results of this evaluation, conducted on a workstation with the following specifications: dual Intel Xeon CPU E5-2670 @ 2.60GHz with 64GB RAM, and a NVIDIA Quadro K5000. Since the CineRender visualizer was not meant to be used for navigation purposes, but only for obtaining static renders of a design, in the table, we included

Table 4.1: Benchmark of ANL on CineRender, following the four pre-defined routes.

ROUTE	NUMBER OF FRAMES	TOTAL RENDER TIME [HH:MM:SS]	TIME PER FRAME [HH:MM:SS]
a)	371	470:15:00	01:16:03
b)	83	64:28:00	00:46:36
c)	53	30:41:00	00:34:44
d)	37	66:50:00	01:48:22

the number of frames that we rendered for each route and the amount of time it took to conclude the rendering of all those frames. As we can observe, just to render a single frame this visualizer took on average an hour, which is why we cannot apply the FPS metric. Just to create a render sequence, to compose a video, architects waited several days for this visualizer to complete the job.

For comparison, we performed the same evaluation on our backend. All the evaluations of our backend were performed at a resolution of 1920x1080 and on a workstation with the following specifications: Intel i7-7700HQ @ 2.80GHz with 16GB RAM, and a NVIDIA GTX 1060. Although the test are being conducted in a, as much as possible, controlled environment, minor fluctuations in the values may be observed due to the presence of other applications causing an overhead on the workstation. As we can observe on Table 4.2, on our visualizer, using the default settings of the Showcase quality preset, we were able to obtain results orders of magnitude better than CineRender, with each frame only taking milliseconds to generate on all routes in contrast to the hours CineRender took. As predicted, route

Table 4.2: The table on the left presents the results of the ANL's benchmark on our backend using the default settings, without enabling any performance acceleration feature. On the right are the results of the same evaluation but without the presence of the libraries' pointlights, which was used as control for the following evaluations.

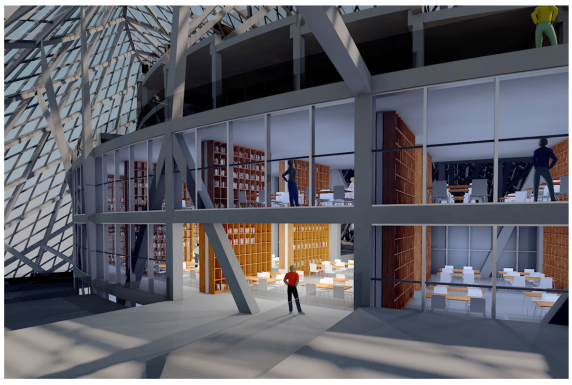
	DEFAULT SETTINGS			CONTROL: NO POINTLIGHTS		
ROUTE	AVERAGE NUMBER OF TRIANGLES	AVERAGE FPS	TIME PER FRAME [ms]	AVERAGE NUMBER OF TRIANGLES	AVERAGE FPS	TIME PER FRAME [ms]
a)	179 650 848	1,86	538	13 914 088	16,98	59
b)	46 635 909	8,70	115	749 445	142,07	7
c)	109 746 991	4,59	218	3 916 203	102,25	10
d)	119 154 519	7,13	140	5 113 570	116,70	9

a) performed worse as it encompassed a view of the whole design, hence has the highest average number of triangles count and would be the most computationally expensive route to follow. Route b) performed the best as, although being an exterior route, its view only covered a small sector of ANL each time. Route c) is second to perform worse as its view from the corridor would have a visibility on multiple libraries at once. We do note, however, that the obtained frame rate with this evaluation is still not ideal for good interactivity, as a frame rate bellow 30 FPS will still be regarded as unresponsive. The reason behind this low frame rate is mainly due to the sheer amount of pointlights which illuminate the libraries' section of the design. The table on the right illustrates the results without pointlights, showing a significant increase in FPS. We can verify the impact of the pointlights on this table just by looking at the great reduction of the average number of triangles on each route. This happens because, for Unity to calculate the shadows produced by a pointlight, it has to project the affected objects several times for each pointlight and in this design we have several small objects under several pointlights, leading to performance struggles.

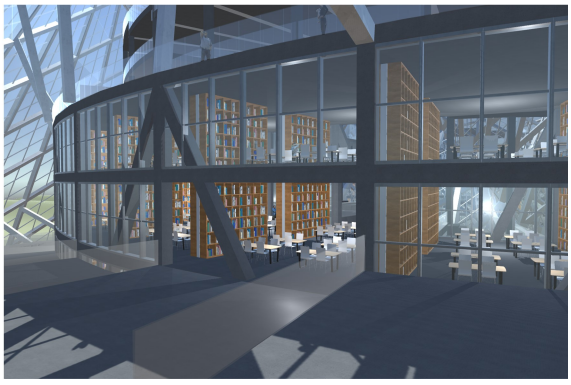
One of the main differences that set these two visualizers apart is the produced image quality, as illustrated in Figure 4.3. However, we do not have a good method to objectively measure the quality of an image, so we did not conduct a formal evaluation in this regard. Nonetheless, we can observe that

Figure 4.3: On the left, a render of ANL on CineRender and, on the right, a render on our Unity backend.

CineRender



Unity



one of the most notable differences between the two images is the way each visualizer calculates the lighting. Note that these renders do not present the same conditions, such as the presence of people inside the library on CineRender's design, and the smaller windows, but in higher number, on Unity's curtain walls. Additionally, the lighting conditions and the materiality are also slightly different, i.e., the sun's position, the intensity of the library's lamp, and the concrete material on the walls. Despite that, we can still observe that CineRender is able to produce softer shadows. Even with these image quality differences, which we deem minor in comparison to the performance gains, we conclude that we have successfully developed a faster visualizer for the architects to use with the AD methodology.

4.1.1 Occlusion Culling Benchmark

To better highlight the impact of the benchmarks using the performance acceleration techniques, we performed all these tests with the pointlights disabled and use as a test control Table 4.2, particulary, the one on the right. Additionally, we registered the speedups by comparing the obtained average FPS in the new evaluations with the aforementioned control table's values.

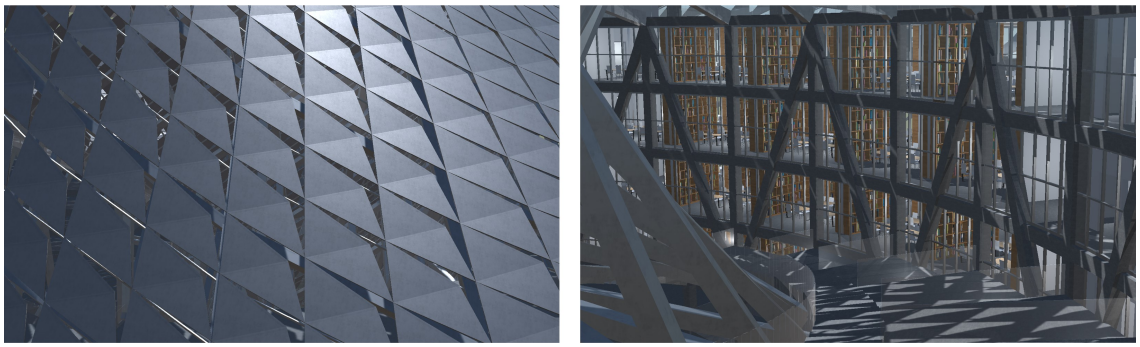
Table 4.3 shows the results of our evaluation with Occlusion Culling (OC) enabled, which incurred an additional 6 minutes of optimization time. As noted, only the last route, d), which navigates inside the

Table 4.3: Results of the Occlusion Culling benchmarks and the respective speedups in comparison to the control table, Table 4.2.

	NO POINTLIGHTS + OC		
ROUTE	**AVERAGE NUMBER OF TRIANGLES**	**AVERAGE FPS**	**SPEEDUP**
a)	14 041 062	12,05	0,71
b)	693 632	142,39	1,00
c)	3 638 207	97,37	0,95
d)	2 759 620	210,27	1,80

library, performed better with almost a double speedup in comparison the rest. Although the libraries section is fairly open, the view of route d) was oriented in such was as to only encompass the library itself, hence, nothing else can be seen as the rest was obstructed by the few short walls dividing each library. As for the lack of improvements on the other routes, they are due to the transparent nature of this design, composed by a transparent facade made of several small objects and also a very open interior with few walls to obstruct the view of the libraries, as illustrated in Figure 4.4. On the left image we can

Figure 4.4: On the left, an outside render of ANL facade, composed of photovoltaic panels, and, on the right, an interior hallway render.



observe that, although the surrounding metallic facade may seem opaque, it is actually composed by several photovoltaic panels that were shaped this way as a result of a lighting analysis. This analysis was performed to control the amount of incident natural illumination on the inside, while using the rest of the sun's light to generate solar energy. On the right image, we can verify that, as all libraries are covered with a glass curtain, we have good visibility of their interior even from the hallway. Furthermore, on the floor of this figure, we can observe the amount sunlight that is able to get past the facade's photovoltaic panels. We conclude that, as expected, the effectiveness of this feature greatly favors navigations in the interior of a design.

4.1.2 LOD Benchmark

Table 4.4 presents the results of applying the LOD technique over the ANL model and comparing the obtained frame rate from all routes with the control table. The application of this technique incurred an additional 3 seconds of generation time, at a total of 50 seconds to generate the whole design with LOD. We can observe only a slight decrease in the average number of triangles and a minor increase in the

Table 4.4: Results of the LOD benchmarks and the respective speedups in comparison to the control table, Table 4.2.

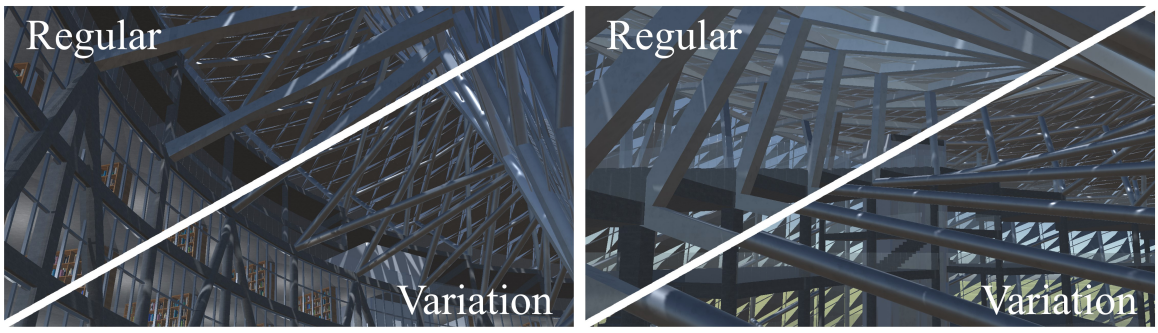
	NO POINTLIGHTS + LOD		
ROUTE	AVERAGE NUMBER OF TRIANGLES	AVERAGE FPS	SPEEDUP
a)	13 872 067	17,11	1,01
b)	746 744	140,19	0,99
c)	3 914 389	108,60	1,06
d)	5 128 195	127,71	1,09

frame rate on all routes. At first inspection, we suspected that the lack of significant improvements was due to the simplicity of the objects that comprised ANL, such as the rectangular beams that composed the metallic facade. As LOD is only most effective on complex geometries with several polygons, we determined not to apply it on simple objects as, otherwise, its overhead would be higher than its benefits. However, as ANL had only a handful of complex objects, for example, the circular slabs that compose each floor, applying LOD solely on them barely have an impact.

To prove our last point and to further test the effectiveness of this feature, we instead created a variation of ANL in which all rectangular beams, which composed the facade structure, the connecting beams with the interior structure, and the supporting beams for each floor, were replaced with high

polygon cylinders, as illustrated in Figure 4.5. With this change, each of the seven thousand three

Figure 4.5: Two composite renders for comparison of regular ANL and its variation.



hundred and eighty six beams was modified from a simple twelve triangles rectangular beam into a eight hundred forty four triangles circular beam. According to our default LOD configuration, detailed in section 3.3.2, the application of LOD on the circular beams cuts its triangle count by half for the first level, and by a quarter on the second level.

Table 4.5 displays, on the left, the results of our benchmark over the ANL's variation on default settings to serve as a new control table for the results on the right, where we applied LOD to the variation. The application of this technique to this variation incurred an additional 111 seconds of generation time, at a total of 157 seconds to generate the whole variation model with the LOD technique. On the left of

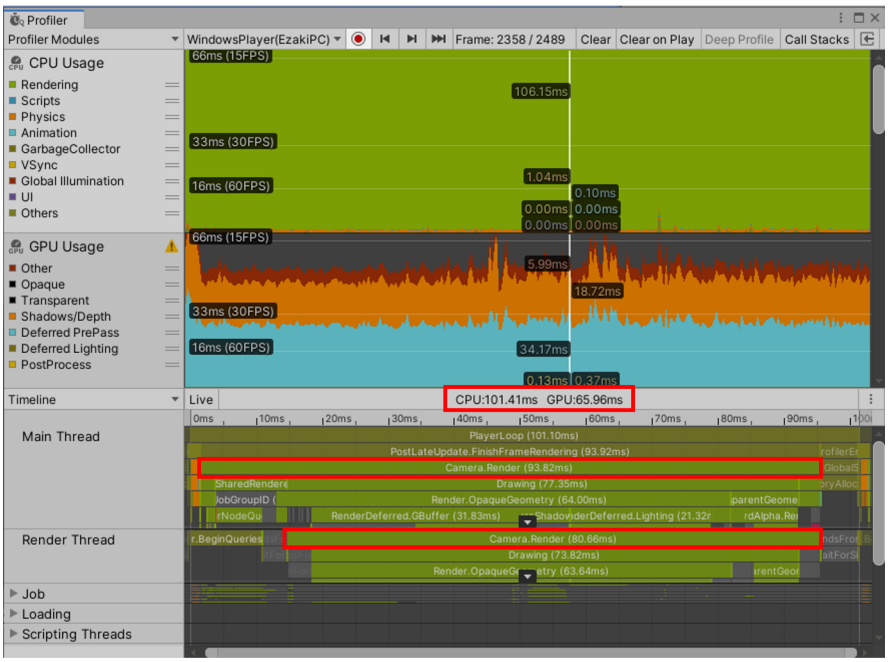
Table 4.5: On the left, the results of ANL's variation benchmark on default settings and no pointlights, which served as a control table. On the right, the results of applying LOD to that variation and the respective speedups against the new control table.

	CYLINDER: NO POINTLIGHTS		CYLINDER: NO POINTLIGHTS + LOD		
ROUTE	AVERAGE NUMBER OF TRIANGLES	AVERAGE FPS	AVERAGE NUMBER OF TRIANGLES	AVERAGE FPS	SPEEDUP
a)	22 944 040	16,89	17 809 347	17,32	1,03
b)	2 256 284	144,63	2 124 848	129,23	0,89
c)	4 905 387	113,43	4 799 388	97,26	0,86
d)	5 845 511	122,41	5 790 433	114,64	0,94

Table 4.5, we can observe that this variation caused an increase of over 50% on the average triangle count for each route in comparison to our original design. On the right results, despite verifying a significant cut in the average number of triangles, specially on route a), the speedups, however, still did not display positive results. On further inspection the main reason behind these results was something we already discussed previously in section 3.3.3.

If we take a closer look at the frame rate of both the variation model on default settings and the frame rate of our original model on default settings, we can already verify that, although we significantly increased the number of triangles in our new case, the frame rate barely changed. This implies that we might be in the presence of a CPU bottleneck, as the current frame rate is not being dictated by the complexity of the scene. Instead, it is being dictated by the amount of draw calls, which in this case remained unchanged because we did not add additional objects to the variation, just changed the complexity of existing ones, hence why the frame rate also remained without significant changes. We can better observe the cause of this issue by using Unity's built-in profiler, presented in Figure 4.6. In

Figure 4.6: Unity profiler results of the time taken by our backend on default settings to render a single frame containing a overview of regular ANL without pointlights. Remarked by a red box are the times taken by the CPU and GPU, and the methods that took the most time on the CPU.



this image we note that, to process a single frame, the CPU requires more time than the GPU. In Unity, the CPU's rendering work is divided in two core threads, main and render threads. The main thread processes all the methods required to render a frame, including user scripts and physics calculation. Whereas the render thread specializes in sending drawing commands to the GPU. On the bottom portion of the image, we can observe that the Camera.Render function, which is responsible for the geometry processing and draw calls batching for the GPU, makes up for over 90% of the main thread and render thread work.

Regarding the negative results of the application of this technique to the variation, although it greatly helped to reduce the impact of a high number of triangles of a view, which would otherwise affect the GPU's performance, its benefits were largely overshadowed by the aggravation of the already existing

bottleneck with the CPU performance overhead incurred by this technique. We conclude that this feature might have its effectiveness greatly limited when used along the AD methodology, which expects the creation of multiple simple objects to compose complex ones to favor the parameterization of a design.

4.1.3 Design Merge Benchmark

Using our Design Merge feature, we are able to cut the number of draw calls in exchange for fewer but more expensive draw calls. We predict that this will shift some of the work from the CPU to the GPU. The results of this evaluation, shown in Table 4.6, further highlights our findings. The use of this feature incurred an additional 85 seconds of optimization time. By merging the design, even in a nonoptimal way, as discussed in section 3.3.3, we were able to greatly improve performance. Route a), which contained

Table 4.6: Benchmark results of the Design Merge application on the regular ANL model, compared to the control table, Table 4.2.

	NO POINTLIGHTS + MERGE		
ROUTE	AVERAGE NUMBER OF TRIANGLES	AVERAGE FPS	SPEEDUP
a)	15 840 248	110,53	6,51
b)	3 316 736	409,26	2,88
c)	6 496 171	229,06	2,24
d)	7 724 148	232,77	1,99

views of the whole design, was the one that obtained the highest speedup.

With this feature, the whole design was compressed down to a single polygon mesh. However, this comes at a cost. Since the generated design is no longer decomposable, it can no longer be affected by features that rely on a certain degree of decomposition to efficiently work, such as both View-Frustum Culling and Occlusion Culling, and LOD, hence we cannot perform a benchmark of the combined techniques. Another hindered feature is the pointlights. By having no decomposition, the performance of calculating the pointlights' shadows greatly increases since, to project a shadow of a single pointlight, out of the two hundred and eighty six pointlights present in the ANL model, Unity has to render the whole design mesh six additional times, as it is the one and only object affected by every pointlight. As shown in Table 4.7, this results in a great increase in the number of triangles to render, comparing to the left results of Table 4.2, impairing the performance more than the Design Merge performance benefits.

Ideally, if we merged logical divisions of the design, instead of the whole, we would be able to get

Table 4.7: Benchmark results of the Design Merge application on the regular ANL model with the pointlights enabled. The obtained speedups also result in comparing the obtained FPS to the ones obtained on the left results of Table 4.2.

DEFAULT SETTINGS + MERGE		
AVERAGE NUMBER OF TRIANGLES	AVERAGE FPS	SPEEDUP
657 823 472	3,69	1,98
319 155 063	6,50	0,75
469 552 193	3,73	0,81
423 558 229	4,34	0,61

further performance boosts. With a correct division, not only it would solve the pointlight shadow performance, but more importantly, the Culling techniques and LOD would be way more effective. Occlusion Culling performs best when we have bigger objects which can act as occluders to hide a larger portion of the design. Additionally, with bigger and more complex objects, the more effective is the LOD technique, which would be able to cut more triangles over a smaller number of objects than previously, impacting less on the CPU.

In conclusion, the performance impact caused by the fine nature of AD models can be solved through merging its composing objects. However, to efficiently use this feature, a certain division must be accomplished. Currently, we can already obtain a significant performance boost without a division, at the cost of some features. However, this boost allows us to obtain an acceptable frame rate, even on complex designs like this case study, making it sufficiently navigable in VR which prevents motion sickness.

4.2 Practicability Analysis

To describe the practicability of the utility features, such as Layers and VR, we will present another case study where these features were used in the AD workflow. Before our backend was developed, an architectural project was conducted in Instituto Superior Técnico (IST) university to improve the acoustics and the visuals of a certain classroom [Martinho et al., 2020]. This classroom is composed by four large flat walls with only a small set of windows on one of the walls. With little to no decorations, this classroom had a severe echoing problem which hindered lectures on it. The goal of this project was to reduce the echo produced in this classroom in an aesthetic way to improve the teaching and learning conditions. To that end, first, this classroom was modeled using Khepri to create a digital prototype of the solution. The

determined solution was to apply a rough absorbent acoustic treatment in the ceiling's surface, to reduce the amount of echo produced, and create a wooden structure composed by several curved panels to be suspended on the classroom's ceiling, hiding the absorbent material and improving the classroom's aesthetics. Lastly, to confirm the effectiveness of the proposed solution, both visually and acoustically, analyses over the digital design were conducted.

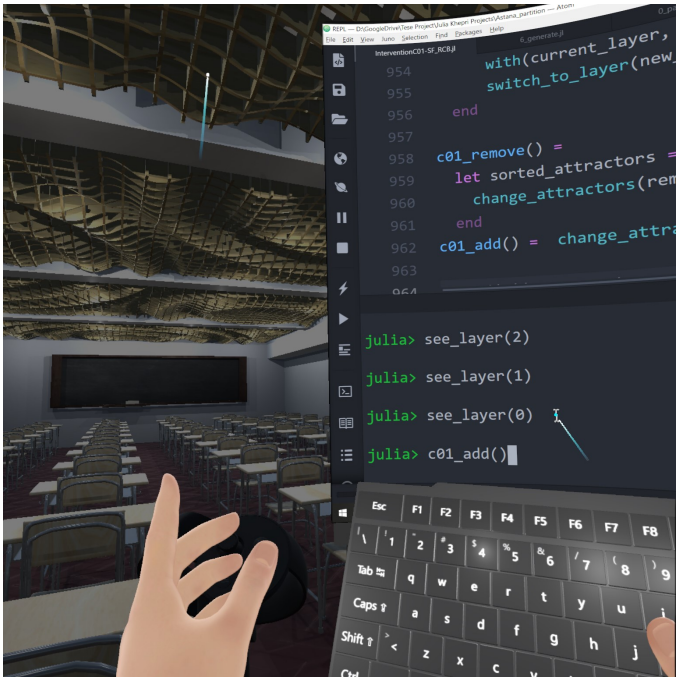
To test the effectiveness of the absorbent material, an analysis tool paired with Khepri was used to run an acoustical simulation over this digital model. During the development of our Unity backend, it was proposed to test the visuals of the wooden structure with it. This wooden structure, also coded using Khepri, was composed by a grid of panels interlaced perpendicularly, with each panel curved in such way to represent a ripple effect. This effect was mathematically modeled with Khepri and its shape would be determined depending on the position of attractor points. By coding the position and the attractor strength of these points on the design program of the classroom, architects could create various design variations of the structure. On Figure 4.7 we can observe the digital model of the classroom generated on our backend with two variations of the wooden structure.

Figure 4.7: Wooden structure variations generated on our backend by changing the attractor points' strength. On the left image the attractor points' strength is weak, only generating a subtle ripple effect. On the right image, the attractor points' strength is stronger so the ripple effect is more predominant.

To evaluate the aesthetics of this structure, the judgement of an architect is required. The best way to do so with a digital prototype is to immerse an architect in a virtual representation of the design model, by taking advantage of the VR capabilities of our backend, discussed in section 3.3.8. While immersed, the architect could directly observe the wooden structure in the digital classroom in scale, as if it was already constructed in the real classroom. Since the shape of ripple effect was dependent on the attractor points, architects could add and remove those points to observe the different shapes it would create. This was done while inside VR, using the LCVR workflow, to avoid having to remove the headset each time a modification was required, making this process faster. In addition to that, layers were used to help the architect's judgment. Each time an attractor point was to be modified, architects used the Khepri client directly in VR, using Oculus Rift virtual keyboard, to run a method that would take their current position in the virtual design to create or delete an attractor point. In turn, this created a

new variation of the design, generated in a separate layer without deleting the original design. When the generation of the new variation was completed, the original design was hidden to then display the new variation. This way, the architect could instantly swap between layers to view the differences caused by the applied changes. The architect can then continue to apply changes and proceed to repeat the process to compare the differences, as illustrated in Figure 4.8. In the end, the layer containing the variation that better fits the architect and even the client's criteria could be chosen.

Figure 4.8: Example use of LCVR to switch between three layers using the *see_layer* operation.



Regarding the method that would cause the modification of an attractor point, causing the upper wooden structure to change in shape, it was necessary to have the Interactive Mode enabled only during its execution. The reason behind this is because, although the design itself was generated outside VR, hence we required the fastest generation possible, this method would be called during LCVR, causing the design to modify while the architect is still immersed. To prevent our backend of becoming unresponsive during this change, while the architect is in VR, this method was declared in the user's code to run in Interactive Mode, making its execution smoother as not to break the user's immersion. When a desired classroom configuration were to be decided, architects could create a demonstration program, using the Standalone Build functionality described in section 3.3.6, for later showcases.

5

Conclusion

Contents

5.1 Limitations and Future Work 76
5.2 Contributions 77

Designing complex buildings requires the architect to use several tools in order to accomplish various tasks, such as 3D modeling, analysis, and rendering. The usage of all these tools leads to a tiresome and error-prone process. Algorithmic Design presents itself as a solution by automating this process. However, it requires the architects to code their design, which is not an easy task for most practitioners. This further allows them to easily build repetitive geometry and to generate the design in any visualization tool. The problem, nevertheless, lies in the fact that currently used visualizers hardly handle the amount of geometry generated by Algorithmic Design programs. This is particularly concerning, since it is extremely difficult to infer the design result by simply observing a computer program. We proposed a fast visualizer which reduces the program-design disconnection, offering richer program comprehension mechanisms to the architectural design process. This solutions allows the architect to receive immediate visual feedback on the changes applied to the program, hence understanding the impact of said changes and accessing errors right away.

Given the advanced state of the Game Engines industry, more specifically the ability to provide high quality results in near real-time with little effort, we proposed to implement a Unity-based visualizer for Algorithmic Design. We chose to integrate our solution with Khepri, a performant Algorithmic Design tool which already supports a variety of both visualization and analysis tools, named backends. The developed Unity visualization backend, explained in section 3, is divided into two sets of features: standard and advanced features. The standard features represent those that are crucial to develop a compatible backend for Khepri. These features include: (1) Khepri operations, composed by construction, geometric manipulation, boolean and camera operations; (2) Assets, comprised by materials and 3D models to adorn a design; (3) Navigation system, allowing an user to explore a design; (4) User Interface, used to connect our backend with a Khepri client and to configure our backend's settings. Khepri, acting as a client, would send operations to our backend, acting as a server, to process and generate the results for the user to visualize. The advanced features include the implementation of performance increasing functionalities and other utility features, that aim at improving the overall Algorithmic Design workflow of an architect. The performance increasing features include: Occlusion Culling, Level of Detail, and Design Merge. The last increases performance by compressing a generated design, composed by multiple small objects, into a single, but larger, object. The utility features include: a Day and Night System, Traceability, Layers, Scene Manager and Standalone Build, Per Project Assets, and lastly, Virtual Reality. The last enhances the architect's workflow by allowing a tangible experience with the designs, motivating creativity and increasing the capacity for error detection in a design.

To evaluate our solution, in section 4, we conducted both a performance benchmark and a practicality analysis for the utility features. The first was performed on an Algorithmic Design representation of the Astana National Library, a complex project with the shape of a möbius strip, which represents a fit algorithmic case study. The latter was performed during a requalification project of a classroom.

An Algorithmic Design representation of this classroom was developed to evaluate the aesthetics of a wooden structure that would be constructed on its ceiling. The use of Virtual Reality and other utility features were used to perform such evaluation.

5.1 Limitations and Future Work

The introducing of Game Engines to the field of architecture opened up doors for many novelties in which we took advantage to completely innovate the workflow of an architect. However, this extensive potentiality reinforced a more horizontal development course for our solution rather than a vertical one. More specifically, we ended up pursuing the implementation of multiple useful functionalities rather than strengthening the fundamental ones. The possible improvements include: (1) a better custom User Interface with visual information of the generated objects, namely their overall complexity, and integration with the Layers feature, instead of relying on the Unity Editor Scene hierarchy; (2) an advanced weather control, instead of just configuring the sun, that enables the addition of weather adversities, for instance, rain, snow, wind; (3) a more stable Constructive Solid Geometry library, since the current one cannot deal with boolean operations over two complex geometries, calculating an incorrect output; (4) the implementation of more advanced Khepri operations, as we only implemented the most used ones; (5) a more extensive default assets library, populated with dynamic assets, such as water shaders and moving vegetation; (6) support for the state-of-the-art real-time ray-tracing technologies, introducing an alternative mode for our visualizer to create photo-realistic renders, if the user has the right hardware; (7) the creation of interactable objects, such as dynamic objects that can be pushed when the player moves against them, or functional objects, such as openable doors and light switches, that a player can use inside or outside Virtual Reality; (8) a better Design Merge division to obtain better performance improvements; (9) integration with lightmapping, since current versions of Unity do not correctly generate lightmaps for our objects. Lightmapping allows us to pre-compute the lighting of a static scene into the objects' materials, hence presenting itself as another major candidate for a performance-increasing feature. Since we deal with static designs, this would give us a considerable performance boost as we no longer needed to calculate lighting during navigation, which is one of the most expensive calculations of a render.

Additionally, as Unity is still being incrementally developed, many new features, that could have been useful for our backend, were announced during the development of this thesis. However, because of the volatility of these features, many in beta stage, and their lack of documentation, we decided to not take advantage of them. These included: (a) Unity's new render pipeline named HDRP, which features a better render quality, more powerful shaders, and even a better overall performance in comparison to the previous render pipeline that we used; (b) integration with the new programming paradigm in

Unity, named DOTS, which, instead of programming using the usual object-oriented paradigm, Unity implemented an alternative way to code games using functional programming from the bottom-up. This new paradigm made the task of creating high-performance multi-threading applications easier, even when dealing with scenes composed by millions of dynamic objects; (c) Unity has started tackling Virtual Reality's lack of interoperability by implementing their own version of SteamVR. Integrating our solution with this new feature, instead of relying on a third party plugin like SteamVR, would give our solution future official support from Unity; (d) Better built-in assets management, which included ways to import assets into a Unity project in a lazy evaluation fashion. Using this, Unity would only effectively load an asset when necessary. As of now, our backend will load all the included assets in the project unless the architect has explicitly excluded them out using our symbolic link solution.

5.2 Contributions

During the development of this thesis, we published two scientific papers regarding our Unity backend:

• Leitão, A., Castelo-Branco, R., & Santos, G. (2019). Game of Renders: The Use of Game Engines for Architectural Visualization. In M. H. Haeusler, M. A. Schnabel, & T. Fukuda (Eds.), Intelligent & Informed: Proceedings of the 24th International Conference of the Association for Computer-Aided Architectural Design Research in Asia (CAADRIA) (Vol. 1, pp. 655–664). Victoria University of Wellington, Wellington, New Zealand [Leitão et al., 2019].

• Castelo-Branco, R., Leitão, A., & Santos, G. (2019). Immersive Algorithmic Design: Live Coding in Virtual Reality. In J. P. Sousa, Henriques, Gonçalo Castro, & J. P. Xavier (Eds.), Architecture in the Age of the 4th Industrial Revolution: Proceedings of the 37th Education and research in Computer Aided Architectural Design in Europe (eCAADe) Conference (Vol. 2, pp. 455–464). University of Porto, Porto, Portugal [Castelo-Branco et al., 2019].

The first scientific paper focussed on the applicability of Game Engines in the field of architecture and the benefits of the use of our backend along with the Algorithmic Design methodology.

The second scientific paper detailed the use of our backend with Virtual Reality to create a new workflow named Live Coding in Virtual Reality. In this workflow, architects can work on their designs while immersed in a virtual representation of them. During the development of both these papers, an intermediate version of our backend was used for the papers' evaluation.

Bibliography

[Aguiar et al., 2017] Aguiar, R., Cardoso, C., and Leitão, A. (2017). Algorithmic design and analysis fusing disciplines. In *ACADIA 2017: DISCIPLINES & DISRUPTION [Proceedings of the 37th Annual Conference of the Association for Computer Aided Design in Architecture (ACADIA)*, ACADIA, pages 28–37.

[Airey, 1990] Airey, J. M. (1990). Increasing update rates in the building walkthrough system with automatic model-space subdivision and potentially visible set calculations. Technical report, North Carolina Univ at Chapel Hill, Dept. of Computer Science.

[Airey et al., 1990] Airey, J. M., Rohlf, J. H., and Brooks Jr, F. P. (1990). Towards image realism with interactive update rates in complex virtual building environments. *ACM SIGGRAPH computer graphics*, 24(2):41–50.

[Akenine-Möller et al., 2018] Akenine-Möller, T., Haines, E., and Hoffman, N. (2018). *Real-Time Rendering, Fourth Edition*. CRC Press.

[Alfaiate et al., 2017] Alfaiate, P., Caetano, I., and Leitão, A. (2017). Luna Moth: Supporting creativity in the cloud. In *Proceedings of the 37th ACADIA Conference*, pages 72 – 81.

[Alfaiate and Leitão, 2017] Alfaiate, P. and Leitão, A. (2017). Luna Moth: A web-based programming environment for generative design. In *Proceedings of the 35th eCAADe Conference*, volume 2, pages 511 – 518.

[Ashour and Kolarevic, 2015] Ashour, Y. and Kolarevic, B. (2015). Optimizing creatively in multi-objective optimization. In *Proceedings of the Symposium on Simulation for Architecture & Urban Design*, SimAUD '15, pages 128–135, San Diego, CA, USA. Society for Computer Simulation International.

[Assarsson and Moller, 2000] Assarsson, U. and Moller, T. (2000). Optimized view frustum culling algorithms for bounding boxes. *Journal of graphics tools*, 5(1):9–22.

[Boeykens, 2011] Boeykens, S. (2011). Using 3d design software, bim and game engines for architectural historical reconstruction. pages 493–509.

[Castelo Branco and Leitão, 2017] Castelo Branco, R. and Leitão, A. (2017). Integrated algorithmic design - a single-script approach for multiple design tasks. In *Proceedings of the 35th eCAADe Conference*, volume 1, pages 729–738.

[Castelo-Branco et al., 2019] Castelo-Branco, R., Leitão, A., and Santos, G. (2019). Immersive Algorithmic Design: Live Coding in Virtual Reality. In *Architecture in the Age of the 4th Industrial Revolution: Proceedings of the 37th eCAADe Conference*, volume 2, pages 455 – 464.

[Clark, 1976] Clark, J. H. (1976). Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19(10):547–554.

[Cohen-Or et al., 2003] Cohen-Or, D., Chrysanthou, Y., and Silva, C. (2003). A survey of visibility for walkthrough applications. pages 412–431.

[Feist et al., 2016] Feist, S., Barreto, G., Ferreira, B., and Leitão, A. (2016). Portable generative design for building information modelling. In *CLiving Systems and Micro-Utopias: Towards Continuous Designing, Proceedings of the 21st CAADRIA Conference*, pages 147–156.

[Fritsch et al., 2004] Fritsch, D., Kada, M., et al. (2004). Visualisation using game engines. *Archiwum ISPRS*, 35:B5.

[Funkhouser and Séquin, 1993] Funkhouser, T. A. and Séquin, C. H. (1993). Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *Proceedings of the 20th annual conference on Computer Graphics and Interactive Techniques*, pages 247–254. ACM.

[Hensel and Nilsson, 2016] Hensel, M. and Nilsson, F. (2016). *The Changing Shape of Practice - Integrating Research and Design in Architecture*.

[Hoppe, 1996] Hoppe, H. (1996). Progressive meshes. In *Proceedings of the 23rd annual conference on Computer Graphics and Interactive Techniques*, pages 99–108. ACM.

[Indraprastha and Shinozaki, 2009] Indraprastha, A. and Shinozaki, M. (2009). The investigation on using unity3d game engine in urban design study. *Journal of ICT Research and Applications*, 3(1):1–18.

[Johansson and Roupé, 2012] Johansson, M. and Roupé, M. (2012). Real-time rendering of large building information models: Current state vs. state-of-the-art.

[Kensek and Noble, 2014] Kensek, K. and Noble, D. (2014). *Building Information Modeling: BIM in Current and Future Practice*.

[Leitão et al., 2019] Leitão, A., Castelo-Branco, R., and Santos, G. (2019). Game of renders - the use of game engines for architectural visualization. In *Proceedings of the 24th CAADRIA Conference*, volume 1, pages 655–664.

[Leitão et al., 2013] Leitão, A., Fernandes, R., and Santos, L. (2013). Pushing the Envelope: Stretching the Limits of Generative Design. In *Proceedings of the 17th SIGraDi*, pages 235 – 238.

[Leitão and Lopes, 2011] Leitão, A. and Lopes, J. (2011). Portable generative design for cad applications. In *Proceedings of the 31st ACADIA Conference*, pages 196–203.

[Martinho et al., 2020] Martinho, H., Pereira, I., Feist, S., and Leitão, A. (2020). Integrated Algorithmic Design in Practice - A Renovation Case Study. In *Proceedings of the 38th eCAADe Conference*.

[Moloney and Harvey, 2004] Moloney, J. and Harvey, L. (2004). Visualization and 'auralization' of architectural design in a game engine based collaborative virtual environment. pages 827– 832.

[Ratcliffe and Simons, 2017] Ratcliffe, J. and Simons, A. (2017). How can 3d game engines create photo-realistic interactive architectural visualizations? In *International Conference on Technologies for E-Learning and Digital Entertainment*, pages 164–172. Springer.

[Rugaber, 1997] Rugaber, S. (1997). Program comprehension.

[Shiratuddin and Thabet, 2011] Shiratuddin, M. F. and Thabet, W. (2011). Utilizing a 3d game engine to develop a virtual design review system. *Journal of Information Technology in Construction–ITcon*, 16:39–68.

[Teller and Séquin, 1991] Teller, S. J. and Séquin, C. H. (1991). Visibility preprocessing for interactive walkthroughs. In *Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '91, pages 61–70, New York, NY, USA. ACM.

[Whyte, 2003] Whyte, J. (2003). Industrial applications of virtual reality in architecture and construction. 8.