Fast Interactive Visualization for Algorithmic Design

Guilherme Jorge dos Santos
guilherme.j.santos@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

October 2020

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

1. Introduction

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 [1].

Currently, the creation of digital designs depends on the execution of several tasks, such as 3D modeling, analysis, and rendering, which require different tools [2]. 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. However, 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.

1.1. Algorithmic Design

Algorithmic Design (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. 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 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, illustrated in Figure 1, 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 [2].

Pursuing the AD approach for designing 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 be-

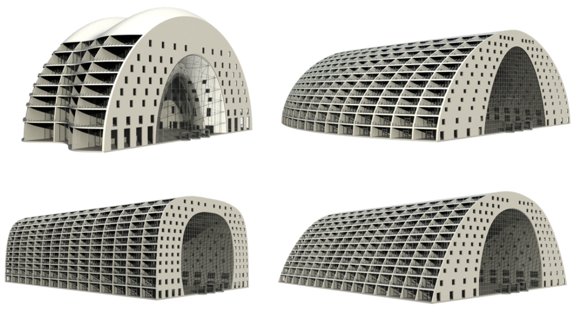


Figure 1: The Market Hall, in Rotterdam, designed using the AD approach, in a parametric fashion. Depicted on the left, is the original design, and, on the right, two design variations, generated by applying simple changes to the parameters' values. Source: [3]

tween 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 a subjective aesthetics evaluation. Additionally, injecting algorithmic logic can hinder the creativity of an architect when designing a building [2].

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 an AD program. 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. The immediate visual 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 [4]. This reduces the distance between the architect's idealized design and the design program's results, increasing program comprehension [5], and thus making programming a less daunting task. To this end, the development of a fast and interactive visualizer is required. Additionally, this visualizer needs to have good visual quality to compete against CAD and BIM tools, while still providing better performance.

2. Related 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 process 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.

2.1. CAD and BIM

The two main paradigms for production and visualization of 3D models in architecture, CAD and BIM, both consist in the creation and modification of a design by means of a computer.

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 and ArchiCAD, 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.

CAD, on the other hand, relinquishes semantics, 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, Rhinoceros, and Sketchup, is a contrastingly easier and faster process, hence architects typically tend to opt for said applications in the early design stages.

On both CAD and BIM application's interface, the user is presented with multiple views of the design model along with all the tools to modify it. Additionally, both kinds of applications are capable of offering visually appealing rendered results, but, despite serving their purpose by modernizing the current architectural design process, they have their performance limitations when paired with the AD process. 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 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, 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. A visualizer for AD should provide good navigation capabilities alongside a good visual representation of the model.

2.2. Luna Moth

Luna Moth is a web-based Integrated Development Environment (IDE) for AD [6, 7] that aimed 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 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 [5], 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.

2.3. Twinmotion

Twinmotion is an Unreal Engine 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.

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 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 a walking mode, where the user is grounded by gravity, and a free-fly mode. Additionally, if the user intends, navigating on the design model in Virtual Reality (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.

2.4. Lumion

Similarly to Twinmotion, Lumion 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.

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.

2.5. Unity Reflect

Unity Reflect is a novel real time visualizer created by the developers of Unity, another popular GE, in collabration with Autodesk, the creators of AutoCAD, a CAD application. This collaboration brings the architecture community and the gaming industry even closer together.

Unity Reflect features a one-click connection with Revit, a BIM application, 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. 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.

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. 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.

3. Acceleration Algorithms

In this section, we explore some acceleration algorithms used for real-time rendering. 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 [8], 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.

3.1. 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 [9], 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. 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.

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: (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.

Back-Face Culling and View-Frustum Culling have fairly simple solutions. 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 [10].

3.2. 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 Level of Detail (LOD) techniques.

The core idea of LOD, introduced by Clark [11], 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 [8].

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, to changes in the resolution of textures and shading.

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. 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.

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.

4. Solution Implementation

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.

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.

In regard to the integration 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 [12], 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, these do not fare well in performance with the complex models that the AD methodology is able to generate. As illustrated in Figure 2, the proposed solution intends to play this missing role as another visualization backend for Khepri.

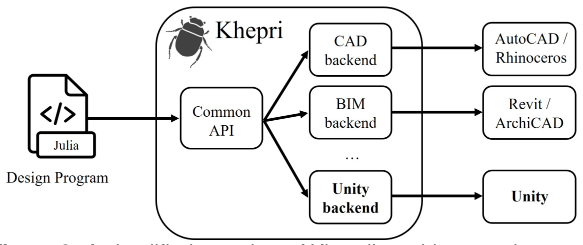


Figure 2: A simplified overview of Khepri's architecture, its supported backends and related tools.

Unity already provides a good foundation for achieving visualization performance. Additionally, we will complement that with two other sets of features to develop our own Unity backend, namely standard and advanced features.

4.1. Standard Features

The standard features include the core functionalities that must be present in our solution to develop a visualization backend that is compatible with Khepri. These include: a communication channel, support for operations to build a design, assets to adorn a design, a navigation system, and a user interface.

Regarding the communication between Khepri and our backend, Khepri uses a client-server architecture with RPC to communicate with its backends. A backend, acting as a server, receives operations to generate a design from a Khepri client, used by architects. The Julia programming language 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.

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. 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 building elements in a design. These range from the creation of simple objects such as spheres, cuboids, cylinders, pyramids, to the creation of more complex objects with semantics, like windows, walls, slabs, panels, surfaces, beams, etc.

The basic geometric operations (2) comprise those that position and modify an object in space, such as: scale, translate, and rotate. 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. 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 pre-determined routes throughout the design model.

Regarding the assets, these are the elements responsible for giving the design model a degree of realism. These comprise: materials, which adorn the generated objects to give the user a better idea of its physical composition, and 3D models, such as tables, chairs, and trees, used to populate the design model to give a better sense of scaling and aesthetics. For materials, Unity supports scriptable shaders and PBR materials. With those two features it is possible to create a high-quality material library. As 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. Nonetheless, our backend provides commonly used materials, such as concrete, wood, steel, aluminum, glass, and plaster. 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.

In regard to navigation, our solution supports the following types of navigation: (1) free-fly mode, (2) walk mode, and (3) static overview mode. On (1) free-fly mode, the user can fly at high speeds through the design model and pass-through any object. The second navigation mode is the (2) walk mode, in which the user can explore the design model in a more realistic manner. On static overview mode (3), a user can define specific viewpoints of the design model and easily switch between them.

Lastly, our backend's User Interface (UI) is used for two main purposes: (1) to initialize the communication with a Khepri client and start navigation of the design model; (2) to allow users to configure the rendering quality and the optimization features of our backend. The latter is important to adapt our backend's performance to the user's workstation.

4.2. Advanced Features

The advanced features include a set of additional functionalities to provide 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. More concretely, the performance increasing features encompass those mentioned in section 3: (1) visibility culling and (2) LOD, and also (3) Design Merge. The utility features encompass: (4) day and night system, (5) traceability, (6) layers, (7) scene manager, (8) standalone build, (9) per project assets, and (10) VR.

For (1) visibility culling, Unity natively supports all three culling algorithms, back-face, view-frustum and occlusion culling. However, the latter requires pre-processing to function since it is a technique that involves the interrelationship among all objects in a design. Users can use our UI to compute the occlusion culling data only when their design is completed.

Regarding the (2) LOD, this feature can also be enabled in the UI. The selection metric used to calculate which object representation should be chosen is the screen relative size of such object. Using this metric, instead of a distance 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 generation step involves mesh simplification algorithms to automatically create a low polygon representation of any generated objects, from simple to complex free-form objects. Since this algorithm is performant, we can apply it to the same object several times, creating several levels of detail. The use of this technique, however, impacts the generation time for a design on our backend, which makes its use more suitable for later stages where navigation is more important that immediate visual feedback on program changes.

The (3) Design Merge is a feature that takes into consideration a performance flaw in designs made by AD. These designs are mostly composed by small objects that, in great number, make up more complex structures. This is due to the parameterization nature of AD, leading to a finer division of a design. However, in complex designs, this may cause a work imbalance between the CPU and the GPU. To solve this issue, this functionality aims at merging all these objects together to reduce the amount drawcalls to process, thus improving performance. This can also be enabled through the UI.

The other type of features aim at improving the AD workflow. The (4) day and night system allows architects to visualize their designs during different times of the day to inspect the interior's natural illumination or create render images. Using our UI they can also configure the global illumination of the design and direct illumination elements, such as pointlights. The (5) traceability feature allows users to query objects present on our backend for the respective function responsible for its generation, and vice-versa. This improves their program comprehension, making the task of programming designs easier. If a user queries a construction function on their code, the generated object will be highlight on the backend using outline shaders.

Layers (6) allow users to organize their generated designs by separating the composing elements into different logical groups, according to their needs. These layers can then be used to change the color of sets of objects, toggle their visibility to evaluate a portion of a design or compare differences between generated design variations. A layer can be created through the user's code to, for instance, separate the objects related to different floors of a building.

Scene manager (7) enable the users to save to disk their generated designs, if they no longer need to modify them using the AD approach, to save on generation time. Additionally, by using (8) standalone build, they can create self-contained programs which carry the generated design that can be used by project clients, without any prior installation of any software, including Unity itself. The (9) per project assets feature can be used to organize an architects personal assets on our backend, to prevent it from getting bloated when those assets are no longer being used.

Lastly (10), as many other fields currently exploring the potentialities of VR, architecture can also benefit from these technologies for visualization. By using the SteamVR plugin, users can connect any kind of VR hardware to our backend to navigate and visualize their designs first-hand. To integrate this novel technology with the current context, 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 [13].

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. To accommodate the use of LCVR, we suggest the use of a virtual keyboard in VR to program a design while immersed. However, during the application of changes to a design, our backend may become unresponsive which may cause nausea to the immersed user. To solve this issue, we have implemented a feature, named Interactive Mode, that limits the processing of operations per frame during LCVR. This reduces the generation speed of a design but makes the process smoother, allowing it to be experienced in VR.

5. Evaluation

To test our solution we have performed an evaluation of our backend's performance, and an analysis over the practicality of the utility features, such as layers, and VR.

5.1. Performance Benchmarks

Our benchmarks will consist of measuring the performance of our backend under the load of a complex case study design, an adaptation of the Astana National Library (ANL) modeled using AD [2], over different scenarios. Figure 3 shows a render of the exterior of this design. The AD version

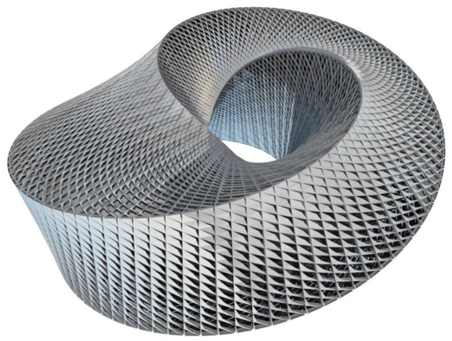


Figure 3: A render of the exterior facade of the ANL.

of ANL is composed by 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 exterior and interior routes on the generated design. For the exterior routes, route a) will encompass an aerial tracking of the building. 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. Finally, route d) will perform a walkthrough inside the libraries of a floor.

Previously, to create render images of this AD description, the architects that created this often 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 predefined routes. Table 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.

Table 2 illustrates the results of the same evaluation over our backend, without any performance acceleration feature enabled, 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. In this ta-

Table 1: CineRender benchmark of ANL over the four routes.

	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

Table 2: Unity backend benchmark of ANL over the four routes.

	AVERAGE # TRIANGLES	AVERAGE FPS	TIME PER FRAME ms
a)	179,650,848	1.86	538
b)	46,635,909	8.70	115
c)	109,746,991	4.59	218
d)	119,154,519	7.13	140

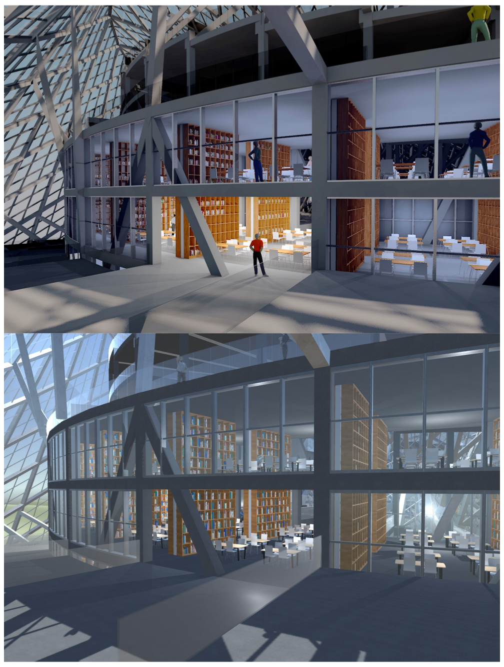
ble, the average number of triangles is displayed, as this is one of the main factors contributing to the backend's performance. We can observe that our backend has obtained results in several orders of magnitude better than CineRender [14]. We do note that the obtained frame rate with this evaluation is still not ideal for good interactivity, as a frame rate bellow 30 FPS will still be regarded as unresponsive. A major factor that deteriorated the performance in this study is the abundant use of pointlights on this design. This ANL representation contains two hundred and eighty six pointlights, illuminating all libraries of each floor. However, by using the performance acceleration features, defined in section 4.2, such as Design Merge, we can obtain up to six times the speed up in frame rate, considerably improving the interactiveness.

One of the main differences that set these two visualizers apart is the produced image quality, as illustrated in Figure 4. 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. 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.

5.2. Practicability Analysis

To describe the practicability of the utility features, such as Layers and VR, we will present another case study, but 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 [15]. 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 decora-

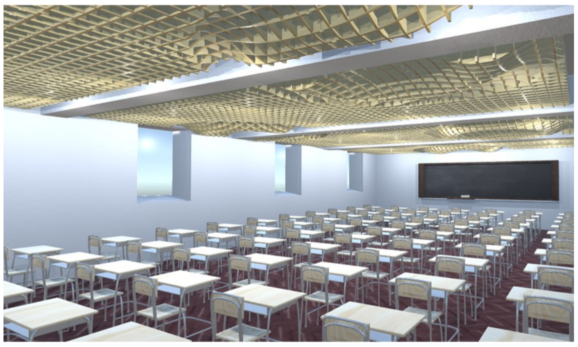
Figure 4: On the top, a render of ANL on CineRender and, on the bottom, a render on our Unity backend.



tions, 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, as illustrated in Figure 5.

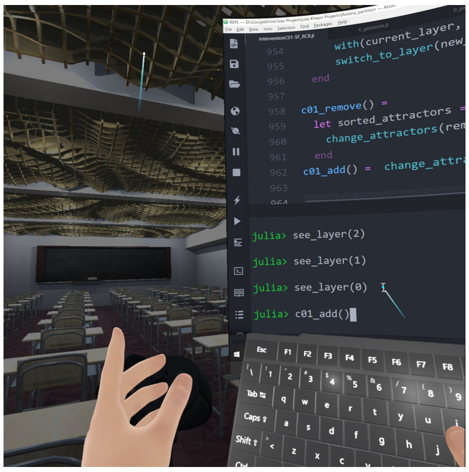
Figure 5: Our backends render of the case study's classroom.



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. 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 6. In the end, the layer containing the variation that better fits the architect's criteria could be chosen.

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

Figure 6: Example use of LCVR to switch between three layers using the see_layer operation.



```
        with(current_layer,
        switch_to_layer(new_
    end

c01_remove() =
    let sorted_attractors =
      change_attractors(rem
    end
c01_add() =  change_attra
```

```
julia> see_layer(2)

julia> see_layer(1)

julia> see_layer(0)

julia> c01_add()
```

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.

6. Conclusions

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. Thus, having a fast visualizer would reduce the existing 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 he applies to his 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 implemented a Unity-based visualizer for Algorithmic Design. This allowed us to attain various performance benefits along with features capable of improving the overall workflow of an Algorithmic Design architect. Furthermore, with the integration of Virtual Reality into the Algorithmic Design methodology, we introduced a novel workflow for programming while immersed in a virtual representation of the design model, called Live Coding in Virtual Reality. This new workflow aims at improving the architects' creativity and error detection by allowing them to experience their designs at first hand and directly observe the materialization of code changes.

References

[1] Michael Hensel and Fredrik Nilsson. *The Changing Shape of Practice - Integrating Research and Design in Architecture*. 03 2016.

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

[3] António Leitão, Rita Fernandes, and Luís Santos. Pushing the Envelope: Stretching the Limits of Generative Design. In *Proceedings of the 17th SIGraDi*, pages 235 – 238, 2013.

[4] Jules Moloney and Lawrence Harvey. Visualization and 'auralization' of architectural design in a game engine based collaborative virtual environment. pages 827– 832, 08 2004.

[5] Spencer Rugaber. Program comprehension. 08 1997.

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

[7] Pedro Alfaiate, Inês Caetano, and António Leitão. Luna Moth: Supporting creativity in the cloud. In *Proceedings of the 37th ACADIA Conference*, pages 72 – 81, 2017.

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

[9] Daniel Cohen-Or, Yiorgos Chrysanthou, and Cláudio Silva. A survey of visibility for walkthrough applications. pages 412–431, 07 2003.

[10] Seth J. Teller and Carlo H. Séquin. 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, 1991. ACM.

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

[12] António Leitão and José Lopes. Portable generative design for cad applications. In *Proceedings of the 31st ACADIA Conference*, pages 196–203, 2011.

[13] Renata Castelo-Branco, António Leitão, and Guilherme Santos. 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, 2019.

[14] António Leitão, Renata Castelo-Branco, and Guilherme Santos. Game of renders - the use of game engines for architectural visualization. In *Proceedings of the 24th CAADRIA Conference*, volume 1, pages 655–664, apr 2019.

[15] Helena Martinho, Inês Pereira, Sofia Feist, and António Leitão. Integrated Algorithmic Design in Practice - A Renovation Case Study. In *Proceedings of the 38th eCAADe Conference*, 2020.