Fast Visualization of Large Architectural Models

Artur Alkaim
INESC-ID, Instituto Superior Técnico
Universidade de Lisboa
Rua Alves Redol 9
Lisboa, Portugal
arturalkaim@tecnico.ulisboa.pt

ABSTRACT
Architects and designers often use Computer Aided Design (CAD) tools to create their models. These tools are very powerful but they are mostly intended for manual use. Unfortunately, the manual production of large amounts of geometry is very time consuming.

Generative Design is one approach which considerably speeds up their modeling process. This approach consists in an algorithmic construction of forms and allows the quick creation of massive amounts of geometry. Most CAD tools recognize the usefulness of generative design and provide programming environments for that purpose. However, CAD tools were not originally designed for this type of usage, and favor instead manual use. As a result, they do not have the necessary performance for the effective exploration of this approach.

This work proposes a solution to this performance problem. Through the use of different techniques that speedup the production and visualization of large volumes of geometry, we propose a library designed to support procedural generation. Together with that, we propose a Wrapper API to simplify the library usage that is developed in using the Racket Programming Language. We use it to evaluate the proposal and explore the use of Racket for the generation of large volumes of geometry through Generative Design.

Keywords
Racket; 3D modeling; Performance

1. INTRODUCTION
Architects and designers are increasingly using large and complex shapes. Figure 1 is an example. Creating such complex models using the traditional manual modeling approach, where the artist has to place each object manually within the model, is very time consuming and error prone. A solution to this problem is the use of a new methodology known as Generative Design (GD). With GD, users describe their designs algorithmically using a programming language, that then generates the models. With this method, the artists have can create very complex models with a modicum of programming effort. Moreover, when properly parameterized, the program can also be used to try different variations of the intended model.

When applicable, GD affords considerable time and effort savings, particularly when there is a need for the exploration of several variations of the same idea. Depending on the particular model being generated, the time needed to develop the GD program plus the time needed to execute the GD program can be much smaller than the time needed to build the model using the interactive means available in the CAD tools. Even in the case when the GD program was difficult to implement, there are gains to expect from the repeated use of that program: a slight variation in one model that can take a few minutes to (re-)generate by a GD program might take the typical architect a few days or weeks to reproduce by manual means.

Being able to quickly regenerate a model opens possibilities that were completely impossible before, including (1) understanding the impact of a particular parameter in the overall design of the generated model, simply by sampling that parameter and visualizing the different models that are produced, (2) fine-tuning the generated model, e.g., by using a user interface containing sliders that are directly related to those parameters, and allowing the user to move the sliders and visualize their effect in the model, (3) complementing the design process with optimization phases that, based on the generation of large numbers of different models, select those that better satisfy some optimization criteria.

Unfortunately, traditional CAD tools, such as AutoCAD or Rhinoceros 3D, were not originally developed for GD use. On the contrary, they are mainly geared towards manual use. As a result, despite the huge savings allowed by generating a model with GD, complex models can still take minutes or hours to produce, making the previous possibilities difficult to explore for those cases. The solution to this problem is, then, to speed up the generation process, possibly using different, faster, CAD tools. Obviously, this is a problem that we will never be able to solve completely, as it is always
possible to create a model that is so complex that it will make even the fastest CAD tool incapable of showing it in a reasonable amount of time. However, as we will show in this work, it is possible to go a long way in this speed up process and to support the quick generation of large models containing considerable amounts of geometry.

2. OBJECTIVES

With the use of Generative Design (GD), architects an designers are able to create very complex models with the composition of large amounts of simple objects, Figure 1 is a good example of that. This is a very complex model that is generated from a relatively simple description in Racket. Due to the large amount of geometry, this example is not easy to render with regular GD tools and it takes a lot of time to see the final its rendering.

The overall goal of our work is to build a GD tool that is able to handle the kind of projects mentioned before. Thus, it should fulfill the following requirements:

- **Performance** - GLFast should be able to quickly generate large amounts of geometry. It should also be able to support the implementation of Immediate Feedback mechanisms to allow the users to quickly see the results of the changes they make, even with large models.

- **Portability** - We want GLFast to be portable and run in the most used systems. It must be implemented so that various Programming Languages are able to connect to its APIs.

- **Completeness** - This work should support a broad set of geometric primitives to provide high creativity freedom to the users.

This work is being developed in the context of the Rosetta Project, that is also a GD tool that helps architects and designers to develop their work using GD. Rosetta is an extensible IDE based on DrRacket and built in Racket. To allow the connection with Rosetta, the development of a wrapper API in Racket is a complementary objective for this work. In this context the module will act as a fast preview mode that allows the users to rapidly see the changes they make on their model during their creative process.

3. RELATED WORK

Several modeling tools were analyzed to guide our development. Our focus was on modeling tools related to architecture or generative design.

3.1 Computer Aided Tools

CAD tools are very powerful tools for design and architecture. With these tools, architects and designers are able to create, interact, and visualize models. CAD Tools are developed to be interactively used. Although the users create the models in a manner very close to drawing on paper, these tools provide a growing set of commands for them to automatize their work. They are very good to analyze and visualize models, Figure 2.

Most of the CAD tools also support GD providing APIs for several Programming Languages, such as AutoLisp, C#, and more recently Ruby. But, since these tools were designed for manual use, they are not prepared to deal with the speed that GD is usually able to create large amounts of geometry. This causes significant losses in performance for these tools.

Another problem is that the functionality that each tool provides is different, and also the APIs are different even for the same functionality. This raises portability issues since GD programs written for one CAD tool will not work with other one.

3.2 Building Information Modeling

Building Information Modeling (BIM) is a recent development in the fields of architecture, engineering and construction. This technology creates an highly accurate virtual model of a building. This model, known as building information model, contains much more than geometric information, that is provided by as CAD tools. It contains objects with semantics that includes information such as geometry, spatial relationships, parametric rules and properties such as materials, finishes, manufacturer and costs. For example, it allows the users to choose a specific door from one provider with all its data, rather than manually drawing a door with all its components.

This technology brings functionalities such as visualization, overall cost estimation, collision detection or even building management.

BIM tools also provide programming interfaces that enable the use of GD. However it implies a very different way to describe the models using this BIM paradigm. For instance, the model of a wall with a door, in a CAD, can be described in any order since there is no semantic inherent to these objects, however, with BIM the user have to define a wall first and, only then, a door can be created with the previously created wall as a parameter.

3.3 Grasshopper

Grasshopper is a Generative Design tool that explore the visual programming paradigm. It have been highly accepted within the designers since is very easy start modeling without much prior coding knowledge and create simple programs. It happens that, as most visual languages, when the user starts to create complex models, it gets very confusing and difficult to understand the programs. Figure 3 is an example of a Grasshopper program that is difficult to understand.

\[\text{http://www.grasshopper3d.com/}\]
3.4 Rosetta

Rosetta is a Tool for generative design that was developed mainly for architects and designers. This tool is written in Racket and supports several programming languages as front-ends, and several CAD tools as back-ends.

It allows the architects to describe their models with programming languages such as Racket, Python or Processing. These languages that are either fit for beginners, such as Racket and Python, since are used in introductory programming courses all over the world, or are already known to them as is Processing, since it has a very large user base among architects and designers.

On the other end, Rosetta allow the users to seemingly see their models with the most popular visualization tools such as AutoCAD and Rhinoceros 3D taking advantage of the pros and cons of each tool. This can be done without any change in the code.

The major problem with these tools is performance. Since the CAD tools were not developed concerning generative design, they have a limited support to it. This is a problem because these tools act as just IDEs and they depend on the CAD tools to visualize the resulting models. Since these models are generated by, usually, succinct programs, they run very fast and are able to generate large amounts of geometry also very fast. CAD tools are simply not prepared to handle these amounts of geometry, which results in long waiting times to see the results. This is an important problem for one main reason: since the users are not experienced programmers they follow an iterative approach, where they write some code and need to see immediately the changes they made in the resulting model.

3.5 Pict3D

Pict3D is a tool written in Typed Racket that provides a purely functional interface to rendering hardware. This interface provides a set of geometric primitives that allows the user to produce a large set of very different models. This tool is based on pict, one of the standard Racket functional picture libraries.

This tool also provides different visualization mechanisms. It takes advantage of the DrRacket IDE and can show the models in its REPL along with the usual windowed visualization.

The big issue with this tool is performance. DrRacket have known performance issues, because it instruments the code for debug by default and also the garbage collection system is sometimes slow. This tool also have the issues inherent to interprocess communication.

Pict3D provides a large set of geometric primitives such as, spheres, boxes, etc., and also various operations to manage illumination. In the example in Figure 6, there are two spheres and an light object that illuminates the scene.

4. GLFast

This Chapter will present our system, GLFast, which provides a 3D modeling alternative that emphasizes performance. Our goals are to provide a solution that is able to quickly regenerate models, thus allowing the users to experiment different variations of their models, mainly during the creative phase, when changes are frequent.

\[\text{http://www.autodesk.com.br/products/autocad/overview}\]

\[\text{https://www.rhino3d.com/}\]
5. Architecture

The system follows the architecture represented in Figure 7. This Chapter will present our system, GLFast, which provides a 3D modeling alternative that emphasizes performance. Our goals are to provide a solution that is able to quickly regenerate models, thus allowing the users to experiment different variations of their models, mainly during the creative phase, when changes are frequent.

5.1 Core

The core of GLFast was written in C++ using modern OpenGL. We use this platform with two main reasons: (1) OpenGL is a cross-platform API for graphics, which is important since we want our system to be cross-platform, and (2) our previous experience with the platform.

As presented in Figure 7, the Core has two independent modules, the Manager and the Primitive Generator. The Manager written in C++ has several purposes. It creates and manages the OpenGL context and windows, manages the data to be sent to the shaders and implements the rendering loop. The Manager runs on the CPU. The Primitive Generator contains the shaders, written in GLSL. It is within this module that the primitives are generated, using geometry shaders. The Primitive Generator runs on the GPU, taking advantage of its multi-core architecture to achieve better performance.

5.1.1 Primitive Generator

The most important modules of the Core are the Shaders since it is there where the primitives are generated. It has simple/pass-through Vertex and Fragment Shaders. The Geometry Shaders are much more complex.

The Geometry Shaders implement the generation of the objects through simple 3D modeling functions. However, since the output type must be specified within the code, as points, lines or triangles, we could only have one output type for a shader. Our solution is to have different Geometry shaders that deal with different output types. This means that we have a shader that generates solid objects from triangles and lines/wireframes with another shader. This fact allows us to optimize the code to generate solids or wireframes.

5.1.2 Manager

The Manager implemented in C++ have several purposes. Its most important role is to create and manage the OpenGL context and windows. It also implements part of the system logic, by receiving and processing all the primitive generating calls and creating the data structures to be sent to the Primitive Generator.

5.2 Wrapper API

Since the focus users for this system are architects and designers, which do not have large programming experience, several design decisions have been made taking this into account. Because the Core API is very low level, we propose a Wrapper API to make easier for the users to use our tool through an simpler Programming Language.

The programming language we chose was Racket. Racket is a descendant of Scheme, a language that is well-known for its use in introductory programming courses. Racket comes with DrRacket, a pedagogic IDE used in many schools around the world, as it provides a simple and straightforward interface aimed at inexperienced programmers. As mentioned before, this language also allows the inclusion of GLFast with Rosetta.

The Wrapper API creates a thin layer that implements concepts that are closer to those our users are used to. Concepts, such as $xyz$ (a point in 3D), are not available within the Core. Currently, it is within the Wrapper API that is implemented the creation of more complex shapes that are composed by simpler ones. Although this is completely transparent to the user, it makes more complex to create other Wrapper API with other PL.

6. Implementation Details

This section explains GLFast. It presents some implementation issues that we have faced and the respective solutions that we developed to solve them. As expected, the main problem GLFast aims to solve is about performance. Current tools for Generative Design are simply too slow. We attack this problem in a number of fronts, form diminishing the amount of data that is transferred between layers, to the management of the size of the model that is processed.
6.1 Encoding Primitives Data

We define primitives as any model that can be modeled by one encoding description.

One problem that traditional approaches face is the amount of data that is transferred from the GD tools to the module where it is processed and visualized, usually CAD tools, which is a bottleneck that has impact on the performance of the GD tools.

In order to reduce the amount of data that is transferred between modules, our solution is to transfer the minimum amount of data possible between modules of the system. To achieve this, instead of sending the whole model description, with every point information, such as position, color, etc., a minimum set of descriptors is used to describe each primitive and generate the model as late as possible within the pipeline.

To create any regular primitive, information about three concepts is necessary: position, scale and rotation. Assuming that the objects are created with unitary size, at the origin and oriented along the z axis, it has to be translated, scaled and rotated. Our solutions to store this information are presented below, having as a goal to minimize the data needed.

6.1.1 Minimum Data

One solution is to implement the transformation using the minimum possible data.

Translations and scales can be trivially implemented using three floats each, six in total. By centering the object at a desired point, and, since the objects are created with unitary size, by multiplying its size values by the scale.

For the rotation a solution is the axis-angle representation \( \phi \) and \( \theta \). This way, only four floats are used, three for the axis of rotation \( \phi \), and one for the angle \( \theta \).

However, this solution is not cumulative, which means that users could not create an object and after that, apply any other transformation without conversions between formats.

<table>
<thead>
<tr>
<th>Position</th>
<th>Scale</th>
<th>Axis-Angle Rotation</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
<td>z</td>
</tr>
<tr>
<td>sx</td>
<td>sy</td>
<td>sz</td>
</tr>
<tr>
<td>( \phi_x )</td>
<td>( \phi_y )</td>
<td>( \phi_z )</td>
</tr>
<tr>
<td>( \theta )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 8: Axis-Angle Data Layout

6.1.2 Transformation Matrix

As an alternative, to solve the mentioned issue, there is another approach which compromises on the used data. The previous solution used a total of ten floats to represent the transformations. By using a \( 4 \times 4 \) matrix six more floats are used, but it solves both issues. This way we can support cumulative transformations.

For each supported object there is a set of attributes that has to be created, as described in Figure 9. This attributes are an object type, a transformation matrix, the color of the object and a value for the scale. Since this approach uses a fixed amount of data that is sent between layers, it might be not optimal for very simple primitives since, some times, they could fit in less data with the traditional approach. But it is easily compensated with more complex primitives. For instance, the data for all the points to model a sphere would be much higher than the data needed using our approach.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Color</th>
<th>Type</th>
<th>Scale</th>
</tr>
</thead>
<tbody>
<tr>
<td>(16 floats)</td>
<td>(3 floats)</td>
<td>(3 floats)</td>
<td>(1 float)</td>
</tr>
</tbody>
</table>

Figure 9: Data Layout

6.2 Irregular Objects

Irregular objects can not be described with the same amount of information that is needed to describe a regular solid. For instance, is not easy to describe an irregular polygon with our data structure. However, it is easy to describe it with a list of its vertices. GLFast uses lists of points to describe irregular objects, which removes the need for a transformation matrix. Without the transformation matrix, there is room for 3 points within the fixed data layout. With these 3 points, it is possible to increase the amount of objects that GLFast supports, adding several regular objects such as lines and polygons.

To create irregular polygons with more than 3 vertices, we implemented a triangulation function that, within the CPU, creates the individual primitives that are fed to our shaders.

6.3 Primitive Generation

Since we use a description of the models instead of the whole information, the primitives need to be generated at some point. This is done within the GPU with shaders. More specifically using geometry shaders. To generate the primitives in the GPU according to the descriptions, simple modeling parametric algorithms to generate the primitives where implemented. Therefore, the primitives are generated as late as possible, and the generated model is never moved since it is already in the GPU. This fact helps us improve the performance of our system.

GLFast supports a limited set of primitives that we believe are able to produce a meaningful set of different models. Primitives such as Prisms, Pyramids and Spheres are available in the 3D space. However, since all primitives are parametrized, a large set of different models can be generated from these ones. Note that the prisms include Box’s, Cylinders and anything in between. GLFast also supports lines and polygons, which can be used as building blocks to create other, more complex, objects.

6.4 Model Processing

We aim to support very large models with our system. This means that sometimes, part of the model might be not visible form the current camera position. The GPU are able to cut out this parts of the models when they are processing the faces. This requires that the entire model is generated before it is cut out. For very large models, this process happens too late. For instance, if we have a very large model of a city, but only a small portion of it is visible, due to the current camera position. The whole model is generated and then part of it is discarded, and this is obviously not good for the performance of the system.

To work on a better solution we take advantage of the timing in which the primitives are generated and apply the Occlusion Culling technique. When each description of an object is received by the GPU, we can know if that object is currently visible, taking into account the position of the camera. With this information we can stop the generation of the object at the beginning, thus generating only the objects
that are visible.

This approach allows another optimization. When an object is very far or is very small, its details are not noticeable. It is usually a waste of resources to have fully detailed objects very far from the camera, when they occupy a very small part of the screen. We implemented a Level of Detail (LOD) technique to manage the level of detail that each object is generated with. When we generate each primitive we can know how far it is from the camera and the scale it has, and with that decide the level of detail in which it is generated. This means that if an object is very far from the camera, we can generate it with less detail without any visible change, thus, taking less time to generate that object and reducing the amount of points to be rendered.

6.5 Adding Color

The assignment of color to the objects was a desired feature, to make GLFast more similar to the other systems. One problem about the common approach, which is to simply transport 3 more values, is the growth of the total transported data. A solution to avoid that, is to use the last line of the transformation matrix to transport the color values, since this values are fixed. Thus, adding a totally new feature without adding any more data to transport, just by better using the available space, as shown in Figure 11. This trick makes the data usage gap shorter between the option of having the transformation matrix versus the minimum data option, finally resulting in the data layout in Figure 12.

7. FUNCTIONALITY

The following sections present the functionality currently available within GLFast through the Wrapper API. First we present the primitives that are supported, which is followed by other operations.

7.1 Objects

Our tool supports a broad set of primitives, which include polygons and geometric solids. The primitives have an additional argument that defines its color and has white as a default value. The last four are an exception, since they are defined by a type that have to be previously specified and that define all style settings to this objects. Some of these objects are created by composing others, for instance, the polygon is created with triangle mesh.

GLFast also support objects such as beam, column, slab and roof, that were included as a proof of concept for BIM support.

7.2 Operations

GLFast also implements some complementary operation. These operations are: View sets the camera position and to where it looks at, mirror creates mirrored copies of the objects according to a point and an axis, rotate and transform apply this transformations to the primitives, init and start sets up the library and starts the rendering respectively.

7.3 Interface

To use the system, we propose the users to use the Racket’s IDE, DrRacket to develop the code. Within the visualization window we implement some functions to improve the

---

**Figure 10:** Data Flow Diagram

**Figure 11:** Transport of the transformation and color data

**Figure 12:** Final Data Layout
user’s interaction with the models. GLFast has *camera control* with the mouse, using a camera system similar to AutoCAD’s.

Our tool also has some shortcuts to change the view of the model. We allow, for instance, to user to change the rendering model between solid, wireframe, points, and solid with edges.

8. **EVALUATION**

Within the following sections we present the evaluation for GLFast.

First we present a functional evaluation that demonstrate the capabilities of GLFast with some examples and, after that, we present a performance evaluation, which was done with several benchmarks and compared with results from related systems.

8.1 **Examples**

In this section we present some examples of models that were generated using GLFast tool and other tools for comparison.

8.1.1 **Truss**

Figure 14 shows a truss model generated by GLFast. This is an architectural example that is very hard to model manually and very heavy for the CAD tools to generate due to the large amount of geometry.

8.1.2 **Cities**

Figure 15 presents a model of a city, to show that this tool is able to cope with very large models. In this example, the city contains approximately 160000 buildings with different sizes, shapes and colors. Which represents \( \sim 7 \text{ million polygons} \). This example was implemented using C++ calling the Core functionality directly.

8.1.3 **Moroccan Pattern**

Figures 16 and 17 show a model that was used to test the different tools, generate with GLFast and with Rhinoceros 3D. This model is a real life example created by an architect for project he worked on. Our example is richer since we support color within the API.

8.2 **Performance Evaluation**

The following sections will present the results of the performance tests made with GLFast and other related tools. Since the goal is to provide an alternative to the use of CAD tools with Rosetta, several examples were generated using the other backends supported by Rosetta, namely AutoCAD\(^7\) and Rhinoceros 3D\(^8\). Different tests were made to evaluate different aspects regarding performance. Since the main focus is to show a final result of a model without much interactivity, the usual metrics for graphics performance, namely FPS and Frame Time, are not the best. The most important metric is the time that a tool takes to generate the first image. However, we used FPS in one test to evaluate the generation of the geometry in the CPU versus GPU.

These benchmarks, and other examples presented throughout this document, were done using a machine with Windows 10, 64-bit, an Intel Core i7-4900MQ CPU @ 2.80GHz,

\(^7\)http://www.autodesk.com.br/products/autocad/overview
\(^8\)https://www.rhino3d.com/
32.0 GB Ram, and an NVIDIA Quadro K3000M 2048MB GPU.

### 8.2.1 Moroccan Pattern Test

The Figure 18 shows the results that the tested tools had with different configurations of the Moroccan Pattern, Section 8.1.3. Although the model produced by GLFast is richer, since it has color and the ones produced by AutoCAD and Rhinoceros 3D are just wire-frames, it has large performance gains as the chart shows.

### 8.2.2 Building Test

A different test was made to include the other backends supported by Rosetta, the Figure 19 shows the resulting model within all the backends, starting with GLFast at the top left corner, followed, per line, with Rhinoceros 3D, ArchiCAD, SketchUp, AutoCAD and Revit. The Figure 20 presents the times that each backend spend to show the final generated model. As we can see, there are major differences between each one of them, and GLFast is much faster than the others.

### 8.2.3 GPU vs. CPU Test

The Figure 21 represents the CPU usage of GLFast generating the Sinusoidal Truss model with different sizes, but is compared with a simple OpenGL implementation without our optimizing techniques or the use of shaders. As mentioned, with small and simple models GLFast can have lower performances but the gains are significant with larger models. The Figure 22 also represents the same test but measured in frames per second, in which GLFast has consistently better results with the same configuration.

### 9. CONCLUSIONS

Architects and designers increasingly use programming as a tool. This powerful tool enables them to work faster and with greater creative freedom. With the development of their programming capabilities they begin to create larger, more complex models. This, unfortunately, creates a performance problem because the CAD systems being used were not built for this kind of use. They were developed for a manual, slow usage, because it was not thought at the time that a user could generate massive amounts of geometry in seconds. As a result generative design users have to wait for

---

large periods of time before they can see the results of their programs.

This is a relevant problem which this thesis attempts to solve. To become a valid alternative to the currently used tools, our solution must have good performance and support the most used functions that the users needs.

The defined requirements led to a simple architecture, that has proved to be a right choice, as it provides good performance and, at the same time, allows the extensibility of our tool to work with different programming languages. This resulted in a shorter pipeline than the commonly used.

Regarding implementation, the use of shaders was a risky choice that turned out to be a good one. Also, the other applied techniques, such as Level of Detail Management and Culling, had significant impact to improve performance.

We proposed goals regarding 3 different aspects for our tool and we analyze each one in the following way:

- **Performance** - GLFast ’s current performance met the goals, since it largely reduces the visualization times in comparison to the other tools, achieving very good performance levels, as shown in Chapter S.2

- **Portability**
  - Since GLFast ’s Core was developed in $C/C++$, which is a language known to run in almost any
system, it has high portability. Racket also runs on the most used Operating Systems. This means that our tool can support Windows, MAC OS X and GNU/Linux. It happens that the current thread implementation of Racket for MAC OS X and GNU/Linux is deprecated, which result in an unstable coordination between the Wrapper API and the Core.

- We demonstrated that it is very simple to use our Core with other Programming Languages such as C++ and Python.

- Completeness - Although our goal was not to replace a CAD tool completely, we still managed to implement some of its functionality, as presented in Section 7. The set of examples presented throughout this documents shows that GLFast is complete enough to create a lot of different models.

We believe we have been able to achieve the objectives defined in these three aspects. From that we conclude that GLFast is a large benefit to this user community, due to the fact that they have now a different tool to use throughout their creative process.

10. FUTURE WORK

There are many features that would benefit this project but due to the time frame for this thesis, are left to future work, namely:

- Although the overall performance of the tool is good, as previously shown, some primitives are naively implemented and would benefit from a better implementation. Some primitives are composed by a set of other primitives which implies the increase of the amount of data that is transported. These improvements can be achieved by profiling the system to find out where are the points that are having the largest impacts on performance;

- Investigate if the implementation of independent shaders for each primitive type would improve performance, i.e., whether the elimination of points of decision within the GPU, by choosing which code to run before, within the CPU, and then running it without further tests in the GPU;

- This system would be improved if the set of primitives that supports were expanded, thus increasing the creative freedom of users, including more high level objects such as building or city;

- Include illumination and shadows to the system, which would improve the rendering quality of the models;

- The implementation of other wrapper API's with different Programming Languages to allow more users to implement their models directly using our library;

- Solve issues regarding the operation in non-Windows Operating Systems, such as the sockets issue presented before;

- Explore different APIs for graphic content such as DirectX 12, Vulkan and Metal to see if there are significant improvements in performance.

We believe that our tool and, therefore our users, would benefit from these improvements.

11. REFERENCES


https://developer.apple.com/metal