House generation using procedural modeling with rules

TFM

Author: Ávila Parra, Rafael

Defense date: July 1st, 2021

Course: 2020-2021/Q2

Speciality: Computer Graphics and Virtual Reality

Master In Innovation And Research In Informatics

Director: Andújar Gran, Carlos

Department: Computer Science Department

Facultat d'Informàtica de Barcelona (FIB)

Universitat Politècnica de Catalunya (UPC) – BarcelonaTech

Table of contents

Abstract

The field of procedural modeling has been increasing in popularity during the recent years, and a lot of different applications appear constantly. CGA shape grammars are one of the many applications that exist and that are designed to create 3D models of houses and buildings based on a set of rules defined by the user. At the same time, there is another field that is also evolving in some creative ways, and that is the design of houses. It is more common to find homes with construction patterns which are very distinctive and different from the traditional designs seen up to this point. It will be the main focus of this project to merge both of these fields together. Create an application based on CGA shape grammars that will be able to easily generate 3D house models with all their particularities taken into account.

The most important part of this project will consist of developing the grammar and all the operations that will be allowed to use. Every scene will start with a simple basic 3D shape. Then, a combination of operations like extrusions, subdivisions or prefab additions will increase the complexity of the whole model to make it look like an actual house. This part was conducted with some basic tests in order to assure a correct performance of the algorithm, since this is the core of the whole application.

The next part will be in charge of improving the overall quality of the results. Since a combination of 3D shapes is not enough to generate house models with enough quality, it is necessary to do some further steps. For instance, texturing is an essential aspect to achieve this goal, together with some extra environmental features.

Finally, the application was tested with some real examples of modern house designs. Not only did it deal quite similar results to the reality, but it also proposed some alternatives due to the not deterministic nature of most CGA shape grammars. All of these models were generated with a very slim set of rules, which made the task of the user really simple for the result they could obtain.

1. Introduction

A house defines one of the most basic necessities of a human being. It's a place that you can call your own and as everything else during history it has evolved. From the most basic and brute edifications you could find thousands of years ago, to some impressive edifications when the most prominent architects started showing their best ideas and concepts. House design has evolved so much through the years, but there is one style that has been appearing specially on the most wealthy side of the spectrum when it comes to properties. Modern houses show a very distinctive set of unique traits that make them stand out from any other designs found in history.

It is not completely out of place to think that all those new designs should be taken into account in other fields. It is easy to assume, knowing the title of this project, that the field we will talk about is digital modeling. There is an incredibly large range of applications where it is necessary to model a different set of buildings in order to generate a sort of 3d urban environment. Those scenarios could be found in movies that involve CGI or any video game that includes urban landscapes or levels.

Of course, modeling an entire scene, which can be as big as an entire city, can be a difficult task. There would be two possibilities in this situation, either model a few sample buildings and repeat them all over the scene or model each unique building for more diversity. None of these options is really viable since they would deal very obviously poor results or would take an insane amount of work behind all the modeling.

This is the reason why during the last years, there have been a set of techniques that have been developed that aim to solve this problem. The name of the most relevant one and where we will focus on this project is procedural modeling. The core of this kind of modeling is to generate 3D models based on a number of rules and constraints described by the user. Even though they can be used in a wider range of applications, this project will only analyze and apply this technique exclusively for house modeling. The name these particular methods recieve is CGA shape grammars.

All in all, this project will try to convey all of this knowledge into one. As we will see, modern houses are very unique and offer a variety of features not common in more traditional designs. For this reason, we will try to come up with different methods and ways to apply and understand CGA shape grammars in order to fit this problem with the best quality possible.

2. Related Work

2.1. Procedural modeling

The concept of procedural modeling has been around for a long time, but not necessarily related to the generation of buildings.

The first well known occurrence of procedural generation are fractals [Bri]. Firstly introduced by the mathematician Felix Hausdorff in 1918, but not actually expressed as "fractals" until 1975 by Benoît Mandelbrot. These particular geometric shapes were the results of a series of rules that were intended to represent a series of events related to several fields, such as physics, economy, psychology or fluid mechanics.

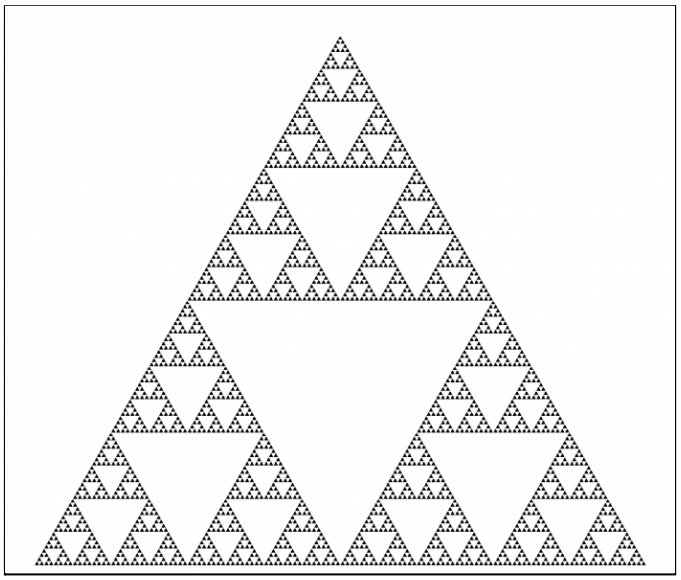

Figure 1: An example of a simple fractal. Source: [https://www.e-education.psu.edu/geogvr/sites/www.e-education.psu.edu.geogvr/files/Lesson_02/Images/SierpinskiTriangle_0.PNG]

As the example shown above, a very simple shape like a triangle can generate a complex geometry by a simple subdivision into 4 smaller triangles, hence, using a single rule.

Another relevant field in procedural modeling is what is known as an L-System. A type of fractals that is generated with a set of rules in order to generate plants. They make use of variables and axioms that, together with the rules already explained, generate a string of characters, based on the elements defined.

After that, an interpreter is used in order to graphically represent the result previously obtained. A simple *turtle* that varies its position and angle will be enough to represent a large variety of plants and trees.

Here is an example of possible plants procedurally generated with an small set of variables and rules:

Figure 2: Example of plants generated through L-Systems. Source: [https://jsantell.com/l-systems/lsystem-header.png]

There has been some work through the years regarding this subject. For instance [Fil94], which makes a description of methods that allow the user to procedurally generate terrain shapes and trees using fractal techniques and L-systems. Another important member of the research community is Przemyslaw Prusinkiewicz [Prz99] who presents a general overview on the field of L-systems and their development and usage on different applications. He would later continue on another one of his papers with some visualizations of his work [PHHM01].

2.2. CGA Grammars

However, the focus of this project is located on a different technique of procedural modeling, CGA grammars. Originally proposed in 2006 in the paper called “Procedural modeling of buildings” [Mül et al. 06], the authors defined a completely new technique of procedural modeling that would be able to generate a large variety of buildings with a simple and reduced input.

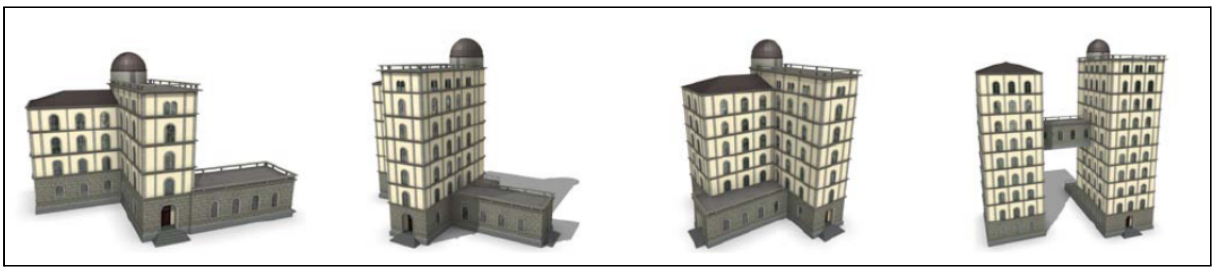

Figure 3: Building examples using CGA. Source: [Mül et al. 06]

The first important concept that was introduced in this paper is the idea to use the combination of basic shapes to generate more complex ones. To monitor each one of them and being able to apply different transformations they defined the following term: scope.

This is an oriented bounding box that contains a simple shape storing information like its position, size and its coordinate system.

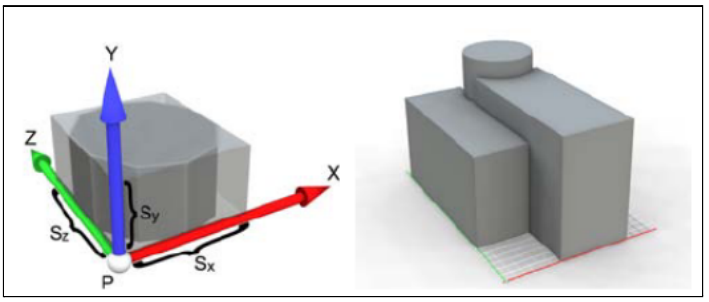

Figure 4: Definition of a scope together with a union of basic shapes. Source: [Mül et al. 06]

After defining the basics, since this is a technique based on procedural modeling, they proposed a production process. Similarly to L-systems, we will have a predecessor, which can be any shape with its corresponding scope together with a successor after passing through a condition. The successor will be the result of any of the following operations: a basic split, a repetition or a component split. Each of them allow the user to increase the level of detail in any shape following some basic rules.

This was, of course, the first approach towards this kind of building modeling, and it had its drawbacks, some of them explained also by the authors. One of the main issues is how to deal with complex surfaces after the application of several production rules over some simple shapes. Occlusion is a dangerous problem that can lay undesired results, as the visibility of a face or part of one is not taken into account when generating the building model. Hence, it was addressed in the paper by tagging a tile, or part of the facade of the building, by being not-occluded, partially or fully occluded and treated consequently as so.

The results they obtained from their work are good looking and their grammar is easy to work with. However, the type of building seems fairly limited and the procedurally generated buildings resemble traditional houses or office buildings with a very simple shape. Even so, they are able to generate a massive amount of individual buildings to obtain a very convincing urban landscape.

```
1: facade ⇝
   Subdiv(Y,5.4,1r,3.9,0.6){ floor1 | Repeat(Y,4){floor2} | floor3 | top }
2: floor1 ⇝ Subdiv(X,5.3,1r){ tile1 | Repeat(X,3.1){ tile2 } }
3: floor2 ⇝ Subdiv(X,5.3,1r){ tile3 | Repeat(X,3.1){ tile4 } }
⋮
```

Figure 5: Portion of a ruleset of a CGA shape grammar. Source: [MZWV07]

There has been a lot of work done on the topic after the work developed on [Mül et al. 06]. For instance, [IG19] created a software that uses a ruleset as input and the user adds some general transformations to obtain a modified ruleset. Another interesting addition can be

found on [MZWV07] where they base their work on a combination of cga rules together with some high quality image analysis to derive an accurate facade subdivision. [LWW08] describes the usage of visual methods to generate rulesets instead of the most common approach, which is text file editing. Finally, a very recent and interesting work shown in [Wil et al. 21] shows the development of a new Procedural Shape Modeling Language (PSML) which uses a volumetric approach to model shapes procedurally. They will describe each shape with a semantic meaning and use position, size and orientation to generate them, instead of building them from the ground as most of the methods seen up until now do.

2.3. Extensions

It has been 15 years already since the paper “Procedural modeling of buildings” was published. During this time, there have been several other works that have used the same technique the original authors described with some improved methodology in order to try and obtain better results.

The first interesting one presents the concept of *CGA++* in the paper “Advanced Procedural Modeling of Architecture'' [SM15]. This paper increased the possibilities in potential results as well as a closer attention to detail.

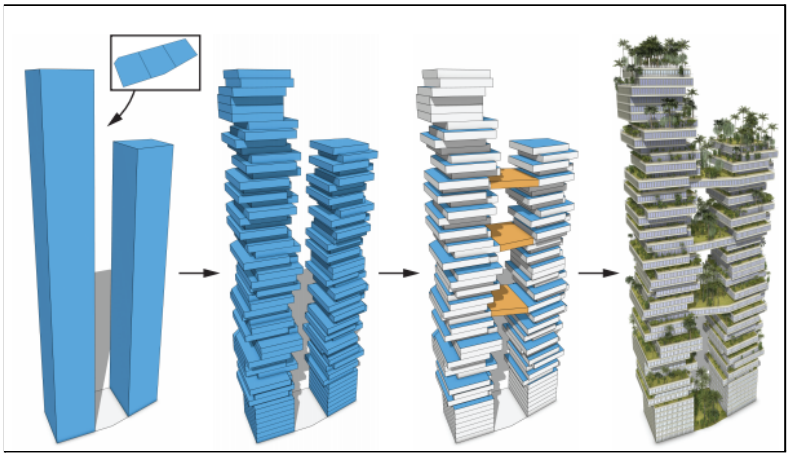

Figure 6: Example of a building using CGA++. Source: [SM15]

They explain how they introduce two main features to the CGA language. The first one is related to the shapes used as a basis for the procedurally generated buildings. As opposed to the original work, they allow the shapes to be included in the grammar and they can be uniquely identified and the possibility to be stored in a tree data structure to keep the relations between one another. This allows the option to add extra operations that involve the shape to be passed as arguments. A very notorious example are boolean operations. This allows to generate a new shape by performing a difference between two of them, or even more.

The second main feature they introduced is what they called a dynamic grouping mechanism. What this means is that they can influence the order of the derivation process to

apply the rules written by the user in a consistent and correct manner. This will be done by taking into consideration the shapes of a set of shapes, as depicted in the following figure.

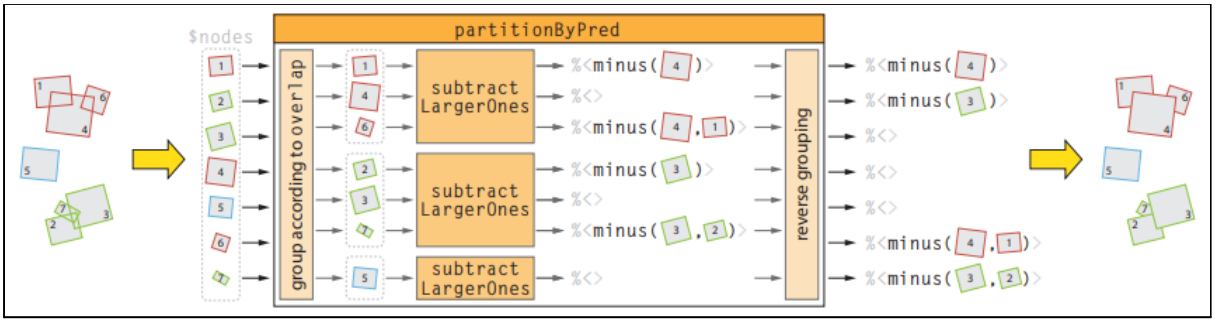

Figure 7: Dynamic grouping mechanism of different shapes using CGA++. Source: [SM15]

A second paper was also released that brought up a very interesting concept regarding CGA grammars. This time it will focus more on the power of the grammar, allowing it to split the generation of the house or building into different parts. The name of the paper is “Layered Shape Grammars for Procedural Modelling of Buildings” [JCS16].

Since CGA grammars always work from a low to high level of detail following the production rules, there are some shapes or house designs that might be a challenge to generate. This is why the authors added the possibility to overlap multiple layers.

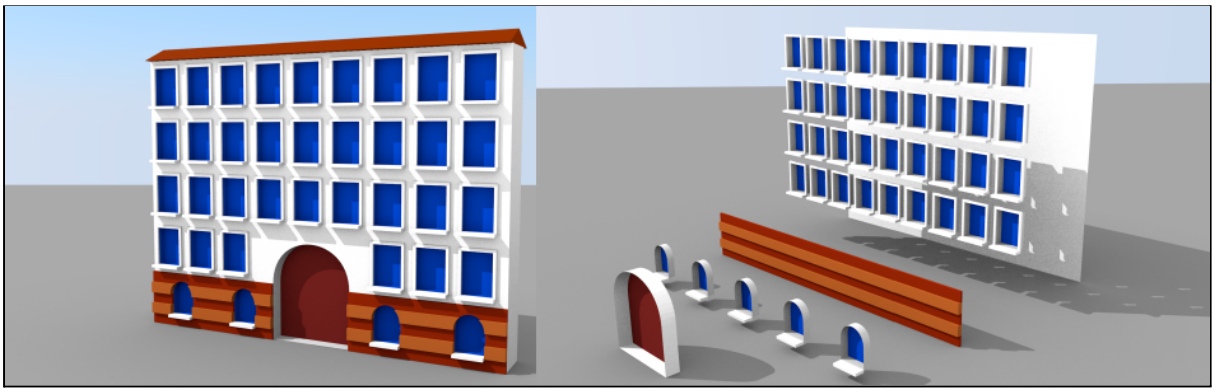

Figure 8: Different layers used to generate a facade. Source: [JCS16]

As seen in the example shown above, the facade obtained on the left is a result of four different layers. Generating it with a traditional approach, like the ones explained up to this point, would be a challenge. This results in a more flexible technique that is easier to use and understand.

```
FloorsSplit ⇝ segment(y,2,~1)
  { GroundFloor | TopFloors };
TopFloors ⇝ segment(y,2)
  { Floor }*;
Floor ⇝ segment(x,1.5)
  { TileSegment }*;
TileSegment ⇝ segment(x,'0.1,~1,'0.1)
  { ϵ | TileV | ϵ };
TileV ⇝ segment(y,'0.1,~1,'0.1)
  { ϵ | WindowTile | ϵ };
```

Figure 9: Some more rules used on a CGA shape grammar. Source: [JCS16]

The greater challenge faced and explained in the paper is how to seamlessly blend several layers with proper results. The authors explained how they will do so by taking into account the relative depth together with other tests to avoid elements that might get partially or totally occluded.

```
1  function occlude(shape, M):
2    if(shape.placeholder):
3      return ∅
4    else if(shape ∩ M = ∅):
5      return shape.geometry
6    else if(shape.fixed_geometry):
7      return ∅
8    else:
9      return shape.geometry \ M
```

Figure 10: Algorithm that describes the occlusion handler. Source: [JCS16]

The figure above describes how their algorithm will classify a 3D shape as occluded on the scene given their intersection with a mask shape named M. This process will be repeated several times when the layer merging is taking place.

Finally, many other extensions can be found around similar topics, even on some subjects that are actually very different to the ones seen up until now. For instance, [STBB14] presents a usage of procedural modeling to develop 3D environment settings. Following a similar scenario, [HM10] provides a review on the evolution of the state of the art of city modeling. This is also related to the ideas presented on the paper "Interactive procedural street modeling" [Che et al. 08], where they develop a software that can create street networks using high quality procedurally modeled urban geometry. Another interesting approach, this time focusing on the generation of house models is [MSK10], where they present a generation of house models following real floor plans and realistic results using real world data. Finally, a last interesting approach on this topic is presented in the paper "Procedural Modeling for Cultural Heritage" [CSN20]. It presents an overview of the application of methods like L-systems and shape grammars used to generate 3D models of cultural heritage.

2.4. Applications

A simple generation of a large number of buildings following the desires of the user can be used to easily generate urban landscapes. Since the release of the original CGA grammar paper, there have been some applications and tools that took advantage of those techniques. The most important one, and the one with the most impressive results, is "ArcGis City Engine" [Esr].

The application is able to generate 3D visualizations of urban environments generated based on the user's preferences. The next figure shows an example together with additional elements such as trees.

Figure 11: A city landscape generated with City Engine. Source: [Esr]

However, its arguably most impressive feature is the ability to procedurally generate buildings based on a real urban map. It has the ability to analyze spatial patterns in maps and to logically detect where to place buildings and generate them accordingly. An example of the power of such features is shown below. More examples are also displayed on their web page linked to the references of this paper.

Figure 12: Recreation of real city buildings using City Engine. Source: [Esr]

This tool is a great example of the properties of CGA grammars, showing how the user can generate a very convincing 3D urban landscape with a few directions to affect the final results to match his preferences.

Another example of an application using CGA grammars, this time more even more related to the topic explained on this project, is BCGA [Eli]. This is another tool created to generate different kinds of buildings using rules. It is a rather simplistic approach, nothing close to the results achieved by using City Engine for example.

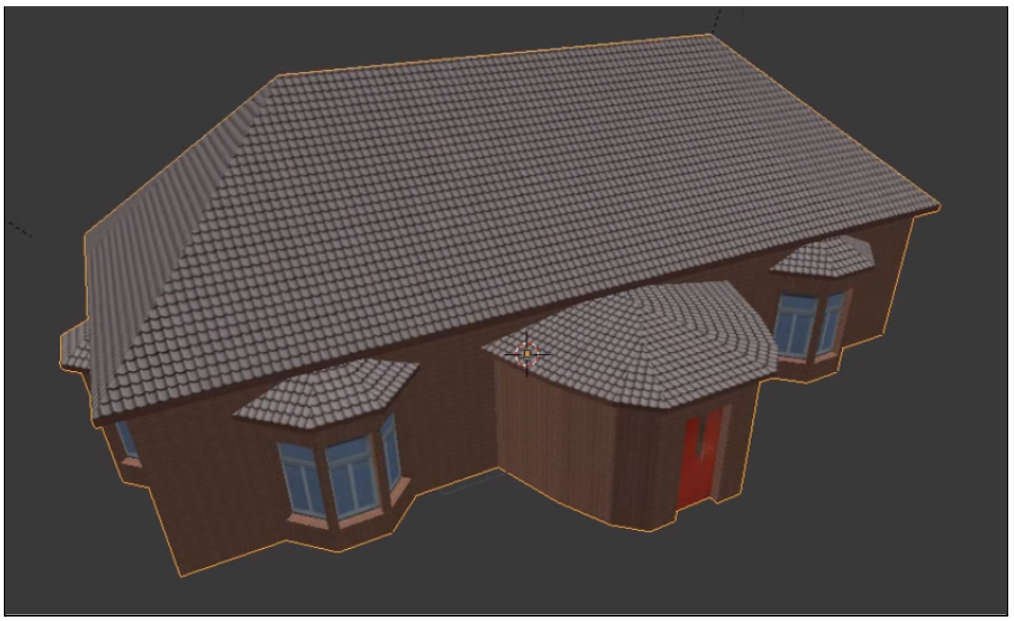

Figure 13: Example of house model generated with BCGA. Source: [Eli]

The reason for its close relationship with the project at hand is they used a tool named Blender to develop it. Blender is a free and open source software used mainly for the creation of 3D models. However, it has many other possibilities that make it interesting for the development of CGA grammars. For instance, it has a Python interpreter embedded that is always available for the user as an extra option in order to make 3D models.

BCGA uses these tools in order to create houses and buildings like the one shown in the figure above. By using simple operations such as extruding, subdivisions and other common operations on CGA grammars, they obtain clean and effective results. As Blender is a tool designed for 3D modeling, some of the operations are already partially implemented by the software itself, which simplifies the task a lot. We will also take advantage of this fact on the creation of modern houses.

Finally, the third final example of application we will talk about is similar to BCGA in some aspects. This is the asset package called blender-osm or OpenStreetMap [Pro]. Even though this is not an open source or free extension as BCGA, it provides more complex results and a wider range of possibilities. In the previous image, we could see that the result only included the model of a house made with a set of rules, but nothing else. Even though one might consider that this is the one result you want to obtain with this kind of software, adding extra elements radically improves the quality of the result. This is the case of OpenStreetMap, which allows the user to use real terrain data where they can place their buildings. Also, the user can place some other type of decorations, such as trees, plants and even roads and pathways.

Figure 14: Buildings modeled using OpenStreetMap. Source: [Pro]

The results above are an example provided by the creator of the application. Even though they are visibly more simple, the results are more similar to the ones obtained on City Engine rather than BCGA looking at the whole landscape. However, by using tiled textures

on the facades, we can see how the results of the buildings themselves will be simpler than the ones seen on City Engine. Nevertheless, for office buildings or other kinds of tall city edifications, this Blender add-on deals very good results.

3. Objectives

The current state of the art related to CGA Grammars is already quite advanced. As we have seen in the previous section, there has been a lot of work done by many people towards the improvement of the procedural modeling of 3D buildings. However, there's one factor all of them lack and that will be the main focus of this project, the generation of modern houses.

Even though it might not seem very different from all the work done up to this point, the design of modern houses, mansions or even buildings is incredibly different. Up to this point, buildings were designed to be as efficient and strong as possible, which resulted in simple shapes, like a box would look. In more contemporary works, architects would use their imagination to create different and more artistic shapes to their creations.


Figures 15 and 16: Designs of modern houses. Sources:
[http://cdn.home-designing.com/wp-content/uploads/2017/05/wood-white-and-charcoal-modern-exterior-paint-themes.jpg]
[https://st.hzcdn.com/simgs/pictures/exteriors/display-home-sentosa-52-metricon-img~3de1a26407620c76_4-2428-1-a74c4ee.jpg]

As we can see on both examples shown on figures _ and _, the general shape of the building shows already clear differences to traditional designs. One of the most common traits is the clear differentiation between the elements of the house. One could say that the overall result is the merging of different elements that were separated originally. This will be one of the basis of our work that will separate itself from other more traditional approaches.

Another element that will help towards the development of this new tool is the fact that the house can also be divided into different layers. As strange as it may seem at first sight, this idea will be very essential in order to obtain greater results with less effort. Due to the rather complex design of the house models, dividing the generation between different layers seems almost mandatory. Fortunately, this technique has been already worked on in the past, as seen on the Related Work section earlier on. This will be another big objective to accomplish during the development of this application.

However, as the houses we are trying to model are complex, the objective of this project is not to develop the most realistic and accurate results. The techniques that will be used will

also be rather complex in its core, and have different options in order to polish final results, that is not the main idea of this work. Since the main innovation comes from the type of building that will be modeled and there is already a very important increase on the complexity of the problem to solve, it is reasonable that there will be other topics that won't receive as much attention.

Another objective, this time more related to the CGA grammars themselves, is to allow the user to easily interact with the final result. We can't expect that everyone would be able to interact with the code itself or the interface to directly modify the result. So, another objective will be to introduce a parser in order to allow the user to write down the rules to generate building models and then use them in our application. Even though this would increase the number of potential users of the application, this will be considered as a feature to be implemented after we obtain good enough results, since it is not mandatory for a good functioning of the application.


Figures 17 and 18: Example of modern house and its representation with our software. Source: [https://hips.hearstapps.com/hmg-prod.s3.amazonaws.com/images/modern-house-2-1538579843.jpg]

In the end, the main objective is to be able to easily recreate houses as the one seen above with a very simple set of inputs. The result obtained with our application shows that you can model an approximation of a real house and obtain some good results, even with low lighting settings. Of course, the amount of detail is not, and will not likely be, the same as a real modern house, since that is not the main goal of the project. In the end, we will see how simple it was to make this simple recreation and how we can obtain lots of different models with the same set of inputs.

4. Planning

4.1. Project development

This project was initially formulated on the 22nd of January of 2021, and accepted five days later, on the 27th as a TFM. The work on the topic has been started since then and will be finished around the 21st of June of 2021. The expected work dedication to this project will be around 800 and 900 hours, which is the equivalent to the 30 ECTS required for this kind of final work.

The workload has been scattered around this time period and more details will be explained on the following subsections. To control the progress of the project and make sure everything is in place, a meeting with the project's director will be scheduled every 2 or 3 weeks.

4.2. Minimum work

After defining the time period where this project will be developed and the corresponding workload, we have to define the minimum work that is expected to come during these months. The technical details on all the tasks to complete will be further explained on the development and implementation sections later on.

First of all, we have to explain the initial set-up for the project to understand our starting point. All the software created for all tasks will be coded in Python, and more specifically, on the interpreter embedded on Blender. For this reason, some of the initial tasks that would be required if using other alternatives such as OpenGL as a 3D viewer are not necessary to implement. For instance, all the rendering processes of the building models will not be implemented on this project. So, we can skip to other tasks more related to the project in hand. This is a list of the features the baseline of this project will have:

- Basic functions to model buildings from, such as extrusion, subdivision or adding prefabs
- The usage of a CGA grammar without the necessity of implementing a parser
- Allow the use of multiple layers on the grammar
- Implement basic texturing
- Model some basic prefabs like windows, doors or roofs

The first one on the list is probably the most important from a visual point of view. These basic functions will be needed to modify the 3D models into more complex ones. By clever use of all of them, you can generate a large variety of buildings and houses, which is a very important element in procedural modeling. The functions chosen are extrusion, which will be used to modify an already existing object, subdivision, used to divide an object into several objects to increase the complexity and the addition of prefabs. This last function is directly connected to the last task, but it's an easy way of increasing the realm of possibilities

in the modeling of the house. As we will see later on, this will be a very useful tool given the complex shapes and elements modern houses often display.

The second task is the core of the project, but will need some functions to actually work, which is the reason it is considered the second task to complete. The ability to write rules is what will make the procedural modeling come to life, which can take the complexity of the results to any point, which will be marked by the user that will actually write them. Although, for this point of the project, it will not be user-friendly yet. The main objective of this task is to execute the rules and obtain a result that will be visible through the viewer. So, up to this point, the rules can be explicitly written on the code. Of course, after we are done with the minimum work of the project, these will be one of the main priorities, in order to make this application more friendly to any user.

The third task regarding multi-layered CGA grammars would always be considered an extra feature on this sort of application, since it's not mandatory to obtain good results. However, as it was already mentioned in previous sections, due to the shape of modern houses, using multiple layers becomes almost a necessity to obtain competent results. By taking a closer look into any of the house designs shown in the previous section, it is fairly easy to see how the facades can be divided into different, very distinctive, elements. This is a very large contrast to other more basic shapes that are present in office buildings or traditional houses, which is what is expected from a regular CGA grammar, as seen on this next figure.

Figure 19: Building created with a single-layered CGA grammar. Source: [https://www.cs.upc.edu/~virtual/SGI/docs/1.%20Theory/Unit%2011.%20Procedural%20modeling/CGA%20shape%20grammar.pdf]

Modern houses could also be created in a single layer, but the grammar would be cluttered and complex, and the results may not be the best. For this reason, this third task is added to the minimum work expected for the completion of this project.

The fourth task is related to texturing the resulting models. Up to this point, we have been working exclusively on the shape and its modeling and generation. However, this will mean the results will be plain and lack any meaning. So, in order to transform a combination of shapes into a proper building, it was decided to implement texturing as a basic task. Blender

adds the possibility to create materials, which could also be very helpful to visually lay better results. But, it is easy to see that using custom textures to represent different surfaces will significantly increase the quality of the results. For that reason, it is decided that custom textures will be added to the models through the code and allow the user to add them to their models with our grammar.

Finally, it was decided that creating some basic prefabs would be necessary to obtain a good baseline for the project. As it was explained before, we will allow the user to add custom prefabs to the house model through the grammar. So, it is important to provide some of them in order to be able to represent basic elements that are common in all houses. Some windows, doors or roof types will be the main objectives for this particular task. It is at this point where we can take advantage, once again, of using Blender for the development of this project. Since the primary functionality of this software is to create 3D models, we will be able to easily model these simple prefabs in the same environment they will be imported later on.

After finishing with this list of essential features, the minimal work will be completed. Then, it will be the moment to move on to some extra features to increase the potential of our software and obtain better and more appealing results.

4.3. Extra features

This is a list of features that were considered not as vital for the project, but will still be considered to improve the general quality of the application:

- Implement a parser to read files with rules and use them in our code
- Add more complex prefabs to the grammar
- Complex texturing methods

The most important extra feature and that will be prioritised as the minimum work is completed is the addition of a parser. Since the code of the project revolves around the CGA grammar, it is quite intuitive to think that there should be a parser to be able to read said grammar at some point. At this point, what we have in this subject is a set of rules that are hard coded into the software that will procedurally generate a house model based on them. This situation is not very user-friendly, which is considered to be important in this project, so this will require some changes. So, the objective with this task is to be able to read files in a simple format (.txt, for instance) and read directly from it. It is not expected to add messages with regard to syntax errors, which means that the user will be responsible for writing rules our software can read. Although, this may change in the future.

Another easy way to increase the potential of the application in general without implementing complex mechanics is to add extra prefabs. As explained in the previous section, creating prefabs with Blender is simple and offers a great realm of possibilities that will give the final model way better looks. Up to this point, the prefab pool will consist of simple elements that are common to most of the modern houses and even more traditional ones (doors, windows and other elements). In this case, it would be wise to look up real

examples of modern house designs and replicate some of the elements that would repeat in some of them. That would allow the user to generate more personal and uncommon house models.

One of the mandatory features that were explained earlier is related to model texturing. A rather simplistic approach is most likely enough for the expected results on this project, but it is a possibility that there might be some improvements. There are several techniques that would generally produce more accurate results. One of them consists of generating the texture coordinates as the dot product of the vertex object space coordinates and a couple of planes. This method, often used when working with OpenGL, can be called by using the glTexGen function with GL_OBJECT_LINEAR parameter. An adaptation of this technique should improve the quality of the texturing, especially when working with tileable (or seamless) textures.

5. Development

5.1. Starting point

The first step when starting any big project like the one we will develop during these months is to set up a starting point. Even though this might change in the future, it is very important to know what is the most basic element we can have in our environment.

Since the core of the project is modeling, we should start with a shape. The most simple 3D shape anyone would think of is a cube. In fact, since the tool used for this project is Blender it feels appropriate, since the very first shape you see in a default scene is a cube.

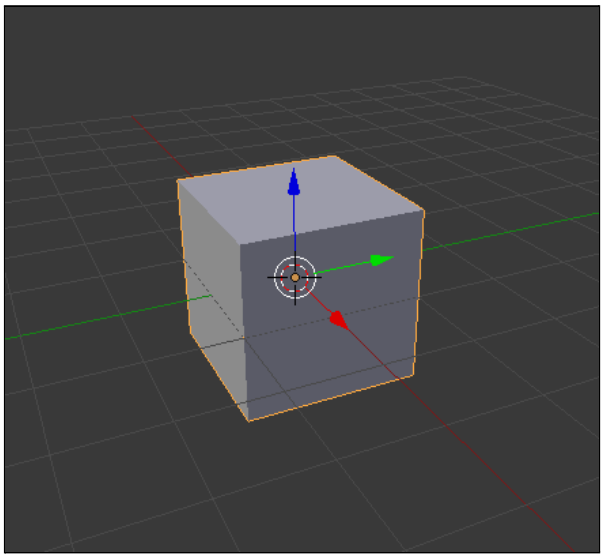

Figure 20: Default cube generated in Blender

Now, it is time to design different ways in which we will be able to modify and work with this cube. Of course, these modifications need to follow certain guidelines, meaning not everything is valid in the context of this project. We will refer to these modifications as operations from now on, and these are the properties they should include:

- Simple and effective: At first, we will design some basic operations that will have the most noticeable impact on the result and that are easy to use and imagine the possible result you will obtain.
- House related: Since this is a project related with the modeling of houses, these operations should help us reach such shapes.

5.2. Basic operations

These are the first basic operations that will be implemented. Of course, they follow the guidelines explained in the previous section.

5.2.1. Extrusion

The first operation implemented in our application will be an extrude function. This shape modification consists of selecting a face of a 3D object and dragging it along its normal. Doing so, you generate a number of extra faces that will depend on the number of edges the selected face has. These new faces will connect the dragged out face with the rest of the 3D shape that will remain unchanged.

This operation can easily be performed on the Blender interface in several different ways, and the result after applying it on the cube we set as the starting point can be seen on the images below.

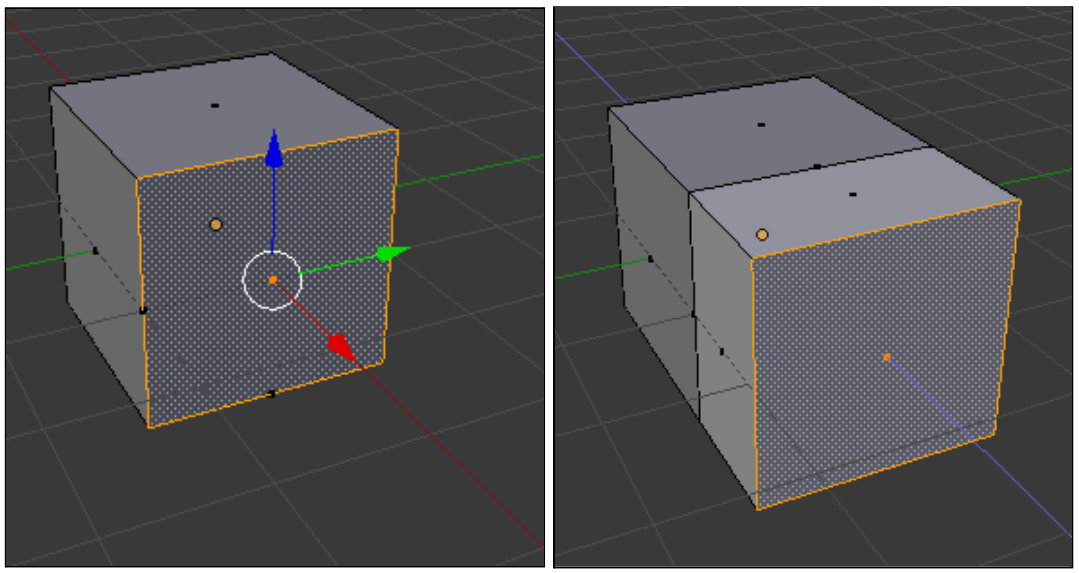

Figure 21: The extrude operations applied to a cube's face

However, as simple as it might be to use with the Blender interface, this is not the objective of this project. Since the main idea is to procedurally generate house models, it would be pointless if the user had to manually extrude or manually manipulate the objects at any point. So, this operation will be implemented with a function in Python using the interface that is already embedded on Blender.

The implementation of this operation was originally based on an already existing code designed to extrude a face from a mesh [Ble]. However, it was modified in order to work on a specific object of the user choice and with other restrictions. As we previously stated, the object we will be working on is a cube, and it has some obvious properties that we can take advantage of. For instance, we know that each of the faces' normal will be facing a different

axis and direction. Knowing this, we can easily know which face to select and subsequently extrude using a parameter to know the distance of extrusion.

Even though it wasn't originally conceived as a problem, there were some issues when applying the extrusion operation. One of the key aspects of this whole process was that we would be working with cubic shapes all the time, to keep the results of such operations consistent. However, when applying an extrusion to one of the faces, in most of the cases we will obtain shapes with more than one face with the same normal. We will see later on, that if we find a situation where there are coplanar faces, the algorithm will produce undesirable results.

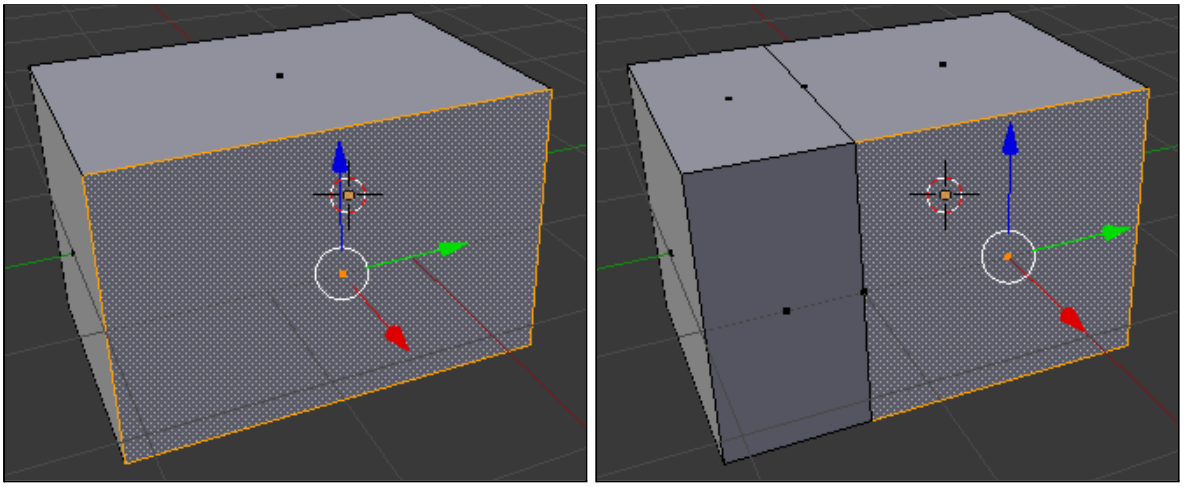

Figure 22: Similar cubic shape obtained with both extrusion techniques

To solve it, we will divide this operator into two different possible outputs, a regular extrusion and a pseudo-extrusion. The first one will be the method already explained up to this point. This will only deal with appropriate results if one of the dimensions of the cubic shape is equal to zero, in other words, it's a plane. In this particular situation, generating four extra faces will still be under the conditions where our application can work properly. This will become useful later on, as the models we will generate will start from a "lot" on the ground, which will basically be a plane.

On the other hand, if we extrude a cubic shape with all three dimensions not equal to zero or having six total faces, we will perform this pseudo-extrusion. In this case, we will just translate all the vertices present in the selected face in the direction of the normal of the face. The result will be the exact same as with a regular extrusion, but we will keep all the shapes within the constraint of only one face per direction. We can observe the results on the figure above.

5.2.2. Subdivision

The previous operation will allow the user to modify an existing cube by modifying its dimensions with an extrusion over a single face. However, we still have a big limitation in terms of complexity of the model, since we are still constrained with a box-shaped shape. This can be solved by implementing a subdivision operation.

In the context of our project, we will call the subdivision of a cube the result of generating several cubic shapes by dividing the original shape along a certain axis. Even though the result is not exactly the same as the one we intend, Blender has a built in operation called "Loop cut and slide" which is shown in the image below.

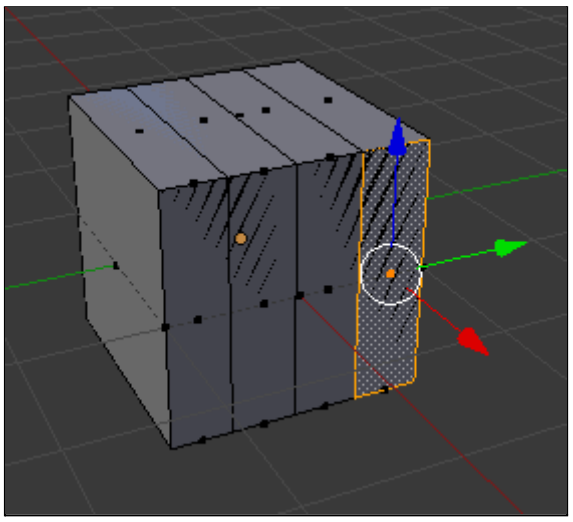

Figure 23: The subdivision operation applied on a cube along the y-axis

As much as it performs a subdivision and we could actually use an operation of the sort for our program, it will conflict with the first operation implemented. As we stated, we took advantage of the simple shape of a cube by extruding a selected shape using the axis on which its normal lies and it's direction. If we use the example from the figure above, we can see that there are several faces with the same normal, breaking our application. As such, this is the solution proposed.

Each subdivision will generate a completely new object. Hence, the result of subdividing a shape x times will generate a total of x+1 objects on the scene. To avoid future issues with overlapping shapes, the original cube that was subdivided will be hidden, but not deleted. As we will see later, it might still be needed once again.

5.3. Prefab addition

With both operations we have implemented up to this point we can obtain almost any object, since we can improve the level of detail of the whole model and generate new volumes by extruding certain faces. However, the intention of this project is to build houses, and reaching a certain level of detail might result in a complex series of both basic operations. It's easy to understand that this is not a desired situation for either the user or the developer, since the goal is to achieve great results with a simple set of inputs. This will get even easier to understand when we start developing the CGA grammar later on.

To solve this problem, we will introduce the idea of prefabs and how they will be added into our own house models. The concept of a prefab comes from the word prefabricated, meaning something that has been previously done or crafted and is ready to be used. In the context of this project, a prefab will be known as a previously own-made 3D model that we will be able to instantiate as much as we want on the scene. Below we show as an example one of the first prefabs designed, a chimney.

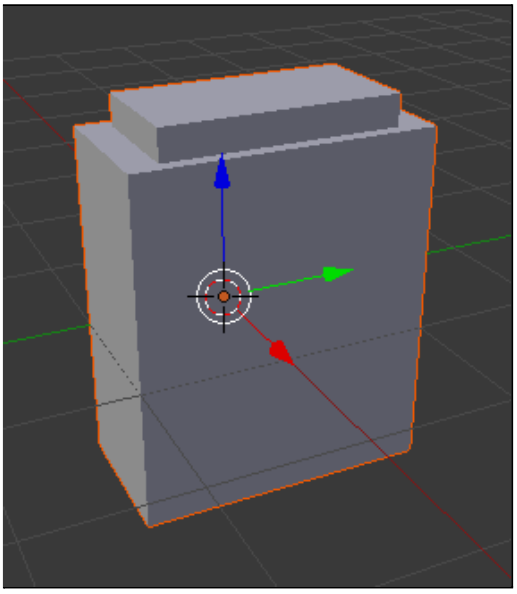

Figure 24: A simple model representing a chimney prefab

Another interesting aspect of the prefabs we will use in this application is that they don't need to be complex. It's quite easy to see on the example how simple the prefab is. However, this does not mean the final result will be as it looks right now. The interesting part comes after the creation of the prefab, the instantiation. Our application will allow the user to place the prefab in the position of any face of any object that is already present on the scene. Moreover, it will adjust it's dimensions to adapt to that face by performing a scaling operation already embedded on Blender. The result could be something similar as seen on the next figure.

Figure 25: The chimney prefab after scaling and texture application

Even though we haven't talked about texturing objects, this proves the point presented before. The power of implementing prefabs easily improves the quality of the results obtained.

However, we still need to explain how we can create such prefabs. This is a very simple task, and the best part is that the user can do it too in order to apply them to their models to obtain more personal results. Since we are working on Blender, creating models and exporting them is one of the most basic tasks you can perform on such a tool. However there are some constraints that need to be respected in order for the models to be properly loaded and placed on the scene. The first one is the size, for the model to match the size of the face it will be placed on, it needs to be centered on the origin of coordinates and have a size of 2 units in both y and z axis. It also needs to be facing the positive x-axis direction in order to be properly oriented.

5.4. Texture application

Up to this point, we coded several functionalities that would help us generate the model from the basic starting point. Adding complexity increases the realm of possibilities and will allow us to obtain models like the one shown on the figure below.

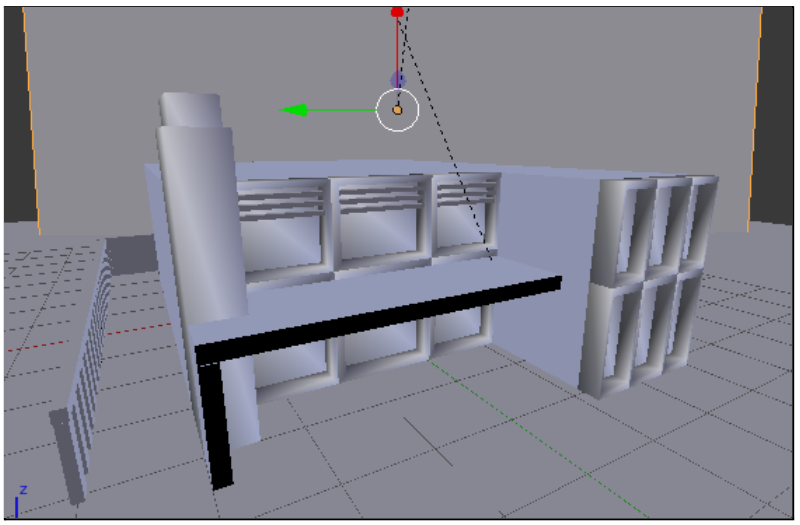

Figure 26: House model without any texture

However, it does not matter how complex or realistic a model is if it looks plain and unrealistic. A combination of 3d shapes is not enough to obtain visually appealing results. This is why we decided to add textures to the modeling steps. To have a clear representation of the increase in quality, we show in the following image the same house model but with some simple texturing and other environmental changes such as lights and shading.

Figure 27: House model with textures

Since we are working with Blender, adding textures is not a difficult task thanks to the existence of materials. Every object that is located in a scene will have material slots, which will define the color of said object. It can be created in very different ways, from the most simple one which is a fixed RGB color decided by the user to materials based on textures. This second option is the one that we will use for this section.

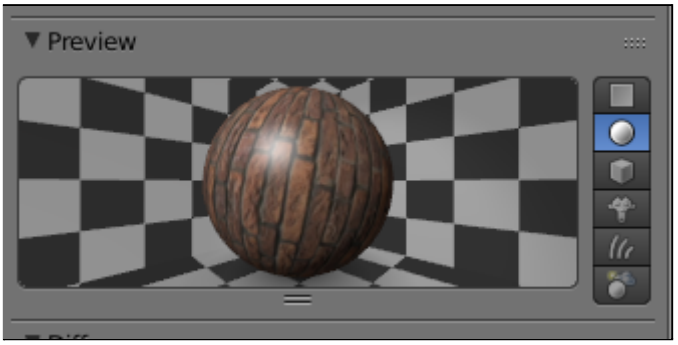

Figure 28: The material edition window in Blender 2.79

Everytime a new material is needed for our model, it will be created from scratch and the texture used will be added as the “active_texture” of the material. Then, the material will be added into the material list of the desired object.

It will also be possible to unwrap any given texture using an angle based method already implemented by default in Blender. The user will also be able to change the scaling of the texture to allow differences in the tiling of the texture to achieve greater results in some cases. It will also be possible to change the offset of the texture. This feature will be of great use later on when designing prefabs with interior views of the house. Of course, to obtain visually appealing results, the texture used must be seamless, which means it can be tileable.

5.5. Block structure and hierarchy

Before we start explaining the details of the grammar, we have to focus on the elements that will be on the scene and will be the basis of all the models created. As we have seen in the previous sections, all operations were made starting from a cube. From now on, we will refer to it as a block.

Of course, we will need some more information about the cubes when we apply certain rules over them to increase the complexity of the scene. For that reason, a block will be a class that stores the following attributes:

- ID: The first and most important attribute is related to the name of the object this block is related to. This name will be given at the time of creation of the block and will depend on several factors. The first one is the parent block, the second one is the layer and the last one is a unique identifier from the rest of the parent's children. On the figure below you can see some examples of block IDs. Their meaning will be further explained in the next section.
- Tag: Meaningless up to this point, but key element when developing the CGA grammar. The tag of a block will determine the next rule to be applied on itself, if there's any.
- Layer: Attribute used to determine on which layer this block is generated. More information when multiple layered CGA grammars are explained.
- Hidden: Determines whether this block will be visible when the model is completed.

Here is a simple example of a combination of blocks obtained after a series of extrusions and subdivisions.

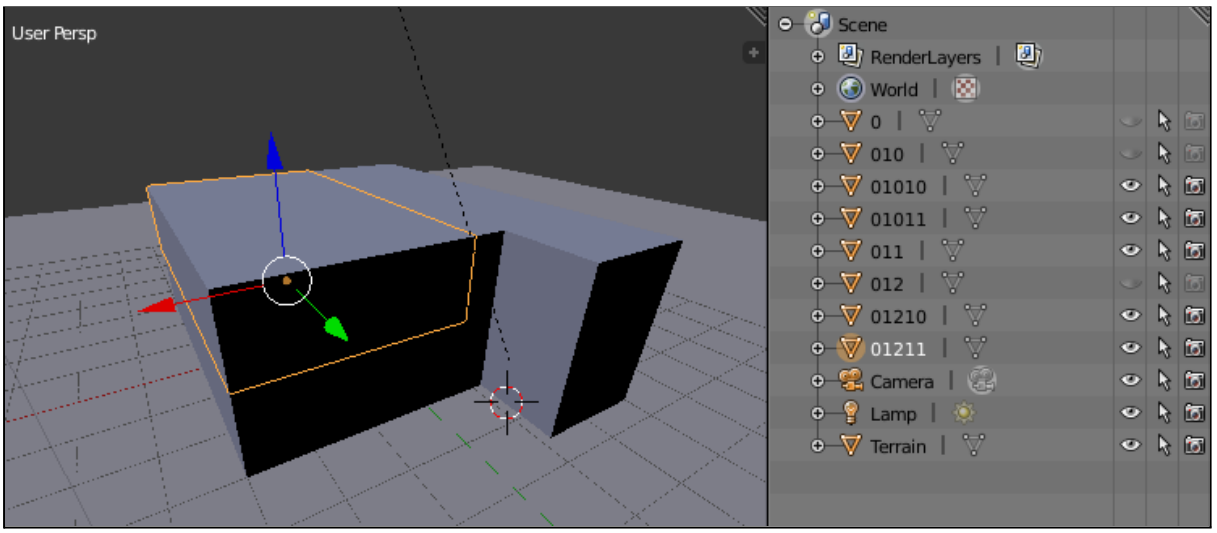

Figure 29: Combination of blocks and their hierarchy

Since it's quite difficult to distinguish where the blocks are and how many are there on the scene, this next figure represents the same model but with spreaded out blocks.

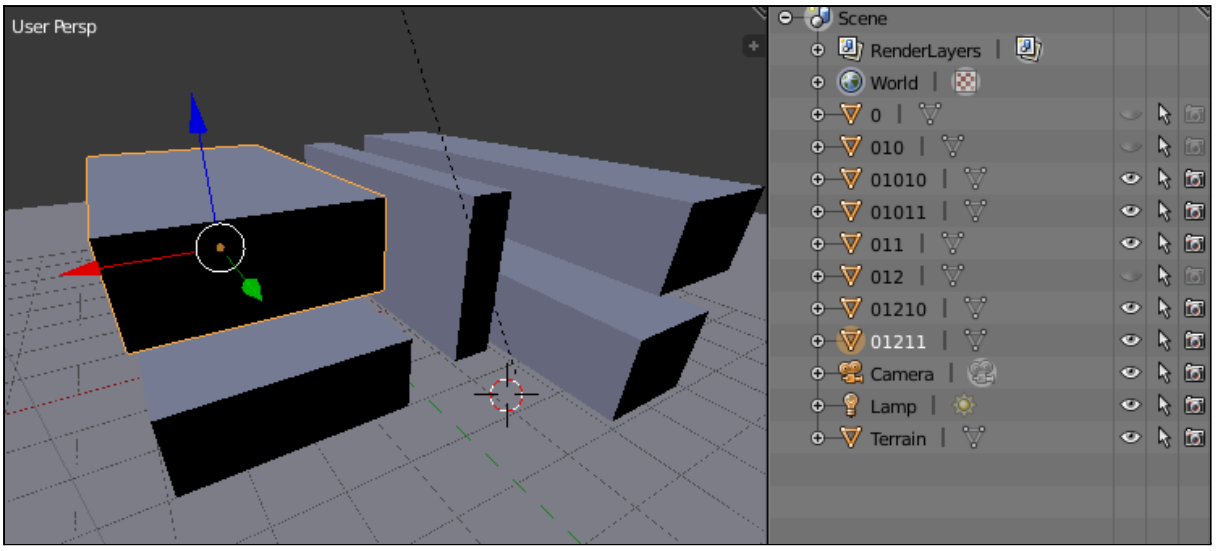

Figure 30: Combination of spread out blocks and their hierarchy

Each block has a unique identifier and together they can form a very large set of models with the set of operations that were explained earlier on. This concept of block will also apply to prefabs that are imported to the scene, where their attributes will work similarly except the IDs, where the prefab name will be included.

5.6. CGA grammar

5.6.1. Introduction to rules

The core of this project is the development of the CGA grammar as it is the functionality that groups all the aspects of the work done up to this point into one. However, there are a few topics that need to be discussed before explaining the details.

As it was seen along the related work section, most CGA grammars use a completely different approach to the block concept presented here. There are two big reasons to apply for this completely different and unique approach.

The first reason, and perhaps the most important one, is the topic of the project. As mentioned on several occasions up to this point, modern houses are unique, hence a traditional approach may not be ideal. Since their shape is actually more similar to a set of blocks than a textured box with windows, like an office building, it was the best solution to this particular situation. This will be more visible in the final results.

The second reason is convenience and simplicity. When working with blocks, you will always apply operations to cubic shapes that, as stated before, are very simple and easy to manipulate. Hence, it was very simple to adapt the grammar to this environment rather than dividing faces or keeping track of different texture coordinates. This will also make the addition of extra layers very simple as we will see in the following section.

Now, let's take a closer look at how the rules are coded in the application and how the user can call and combine them with each other.

The general structure of any rule is the following:

```
if tag == 'TAG':
        return rule
```

It starts with a condition that checks for the tag of the current block that is being worked on. It was explained in the previous section that every block will store a tag. This is the point where it will be accessed and the user can filter any block using this attribute. After that, the second line starts with a return statement, as the rule will always return at least one new block which will be added to the queue of blocks that will go through this process.

Any block that has a tag that is not present in any rule condition will be considered terminal and will not suffer any further modifications or expansions. Another alternative to consider a block terminal is to remove the return statement, which means that the resulting block will not be added into the queue.

5.6.2. Rules and operations

After learning the basics about the CGA rules as they will be used on this project, it is time to look closely at each of the rules and their particular grammar.

The first one is extrusion. This is how the grammar has been set up:

`extrude_face ( object_name, direction, value, tag )`

- object_name: the object that will be extruded, fixed to obj
- direction: the direction where the extrusion will be performed. Since we are working on a cube, there are six possible directions along the three different axes: 0 for positive x, 1 for negative, 2 for positive y, 3 for negative, 4 for positive z and 5 for negative.
- value: the amount of units the selected face will be extruded. This value can either be a fixed value or a range. Both options have to be explicit between "[]".
- tag: the tag that will be assigned to the block created after performing the operation. In this case, it will be changed to the already existing block, since its id will remain unchanged.

The figure below shows an example of the extrusion rule

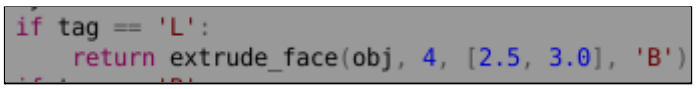

Figure 31: Rule including an extrusion

The next rule to be explained is the subdivision, some of its syntax is similar to the extrusion. However, it includes some new tokens and a different meaning to some already existing ones. Here is a template of this rule:

`subdivide ( object_name, direction, values, tags, material )`

- object_name: the object that will be extruded, fixed to obj
- direction: the direction where the subdivision will be performed. In this case, we will only have three different values available, since the orientation does not matter in this situation, only the axis in which the subdivision will be performed. The values are 0 for the x-axis, 1 for the y-axis and 2 for the z-axis.
- value: the amount of units each subdivided block will measure in the specified direction. This value can either be a fixed value or a range, as before. Each element added in this field will mean an extra subdivision. In this kind of rule, there is an extra possibility for this field, adding a constant. Defined by the name 'R', it will calculate the size of the current block depending on the total size of the original block and the other blocks that will be generated from this subdivision. This constant

can also have a value attached to it which is a multiplier that will grant larger or smaller sizes to the current block.
- tag: the tag that will be assigned to the blocks created after performing the operation. In this case, each specified tag will correspond to each of the blocks generated by the subdivision. The number of tags must be the same as the number of blocks created in the operation.
- material: this rule also allows a material to use for the blocks generated. To load a texture, simply specify the name of a .jpg image that will be used as a texture. This image has to be inside the "textures" folder.

Once again, here's an example of a subdivision rule taken from the Python interpreter in Blender.

```
if tag == 'F':
    return subdivide(obj, 2, [[1.2, 1.5], ['R', 1]], ['FL', 'FL'], "marble")
```

Figure 32: Rule including a subdivision

Finally, we can switch to the last functionality that has been explained. This is the syntax for the prefab addition:

add_prefab (object_name, prefab_name, scale, direction, tag, material)

- object_name: the object that will be extruded, fixed to obj
- prefab_name: the name of the prefab that will be instantiated on the scene. It must consist of a .obj model created using certain guidelines that will be explained in the "Prefab design" section later on.
- scale: scaling of the last component of the prefab. As it was explained before, when adding prefabs into the scene, the size of the instance will be matched to the face of the block where it was called from on the grammar. However, the face it is attached to is a square, meaning only two components of the scaling vector will be automatically calculated. Then, this last degree of freedom will be for the user to decide depending on their preferences.
- direction: the direction where the intantation will be done. In this case, we face a similar situation as the extrude rule. We will have six different faces where the prefab can be attached to. The values for each face are the same too.
- tag: the tag that will be assigned to the block created after performing the operation. Similar outcome to the extrusion operation here as well.
- material: this rule also allows a material to use for the block generated. The usability is the exact same as the one explained on the subdivision operation.

Once again, here are some examples of prefab addition in the ruleset.

```
if tag == 'W':
    return add_prefab(obj, "window", 0.5, 2, 'N', "wood")
if tag == 'W2':
    return add_prefab(obj, "window2", 0.5, 2, 'N', "wood")
```

Figure 33: Rules including prefab additions

This is every aspect of the rule syntax regarding the basic operations that were explained up to this point. However, there are many ways to improve the complexity of the ruleset and get even more interesting results without the need of implementing more functionalities to the application.

The most important and easy to use is randomness. Here is an example on how it can be applied to the ruleset.

```
if tag == 'B':
    rng = random.uniform(0.0, 1.0)
    if rng < 0.3:
        return subdivide(obj, 0, [['R', 2], [0.5, 0.7], ['R', 1]], ['F', 'C', 'EF'], "marble")
    elif rng < 0.6:
        return subdivide(obj, 0, [['R', 1], ['R', 2], [0.5, 0.7]], ['EF', 'F', 'C'], "marble")
    else:
        return subdivide(obj, 0, [['R', 1], [0.5, 0.7], ['R', 2]], ['EF', 'C', 'F'], "marble")
```

Figure 34: Rule including 3 possible subdivision chosen at random

The figure above describes three possible outcomes for any block with a tag “B”. Each of them is a different kind of subdivision that has different possibilities of happening. 30% for the first 2 outcomes and 40% for the third one.

This is a very important feature in CGA grammars in general. Since the main goal of any application like this one is to easily generate house models, the more the better, it’s important that the results are not 100% deterministic. It is true that as we will see in the results and discussions, that due to the complexity of modern houses, this randomness can lead to some undesirable results, or maybe unrealistic. However, the point of having a functionality like this one is to allow the user to obtain more than one house model for the same set of rules, allowing him to make multiple instances at the same time without being exact copies of each other.

All the types of rules that have been explained up to this point are very powerful and simple to use. Here is an example of a modern house model generated from only 8 different rules and some of the blocks that are visible on the hierarchy on the right side of the image.

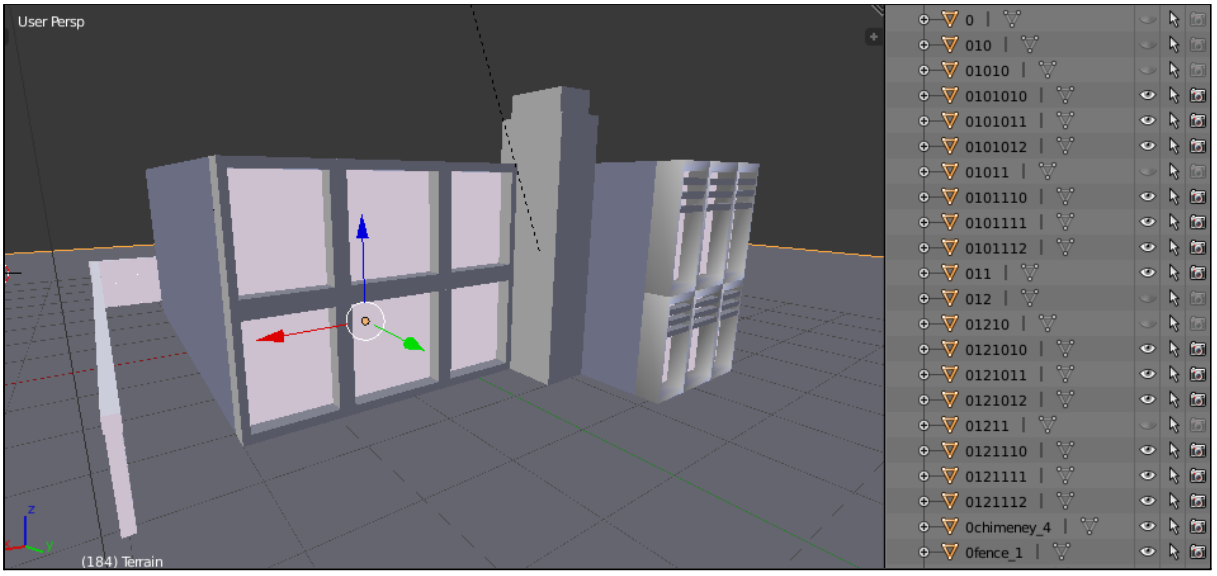

Figure 35: Example of model obtained with a set of rules

However, as much as there are an infinity of possibilities with the resources discussed up to this point, there are simple ways to add more elements to the model that would require extra rules that might make the grammar way more complex than it should be.

5.7. Multi-layered CGA grammar

The concept of multi-layered CGA grammars is not new, as we saw on the Related Work section. However, it will be implemented in this project to further increase the potential of our grammar due to its rather simple implementation. Working with blocks instead of texture coordinates as in previous works will help us avoid some issues regarding occlusion between layers which are the main problems on these types of applications.

However, we still need to discuss the reason why this is a needed feature in the application. Once again, the biggest motive is that the inclusion of such a technique is ideal for the type of models we are designing. We can use the example model presented in the previous section as a starting point. Below, we can see the real house where the model was inspired from, but there is a section missing, the one highlighted in red.

Figure 36: Modern house example with possible second layer highlighted

With the current ressources implemented up to this point, it is possible to recreate this missing part of the house structure. But, it will take further operations to achieve that and it may result in more deterministic results overall. That is not the case if we add that section of the house afterwards, with a second layer of blocks.

The usage of different layers on our application is fairly simple. This is the code template to divide the rule set into different layers:

```
if layer == number:
        rules for this layer
```

And the next figure shows an example with a few rules in it.

```
if layer == 1:
    if tag == 'L':
        return extrude_face(obj, 4, [2.5, 3.0], 'B')
    if tag == 'B':
        return extrude_face(obj, 2, [1.5, 2.0], 'F')

if layer == 2:
    if tag == 'B':
        return extrude_face(obj, 3, [2.5], 'F')
```

Figure 37: Ruleset including 2 layers

By using this, we are allowed to expand the potential of our CGA grammar in a very simple manner. For instance, these are the 3 rules that were added to the original 8, which were included into the first layer, that allow to complete the house model from the example.

```
if layer == 2:
    if tag == 'B':
        return subdivide(obj, 2, [['R', 1], [0.15, 0.2], ['R', 1]], ['CB', 'CO', 'N'], "marble")
    if tag == 'CB':
        return subdivide(obj, 0, [[0.15, 0.2], ['R', 1], [0.15, 0.2]], ['CO', 'N', 'CO'], "marble")
    if tag == 'CO':
        return extrude_face(obj, 2, [1.5, 1.6], 'N')
```

Figure 38: A set of rules added to the previous model

And this is the result obtained.

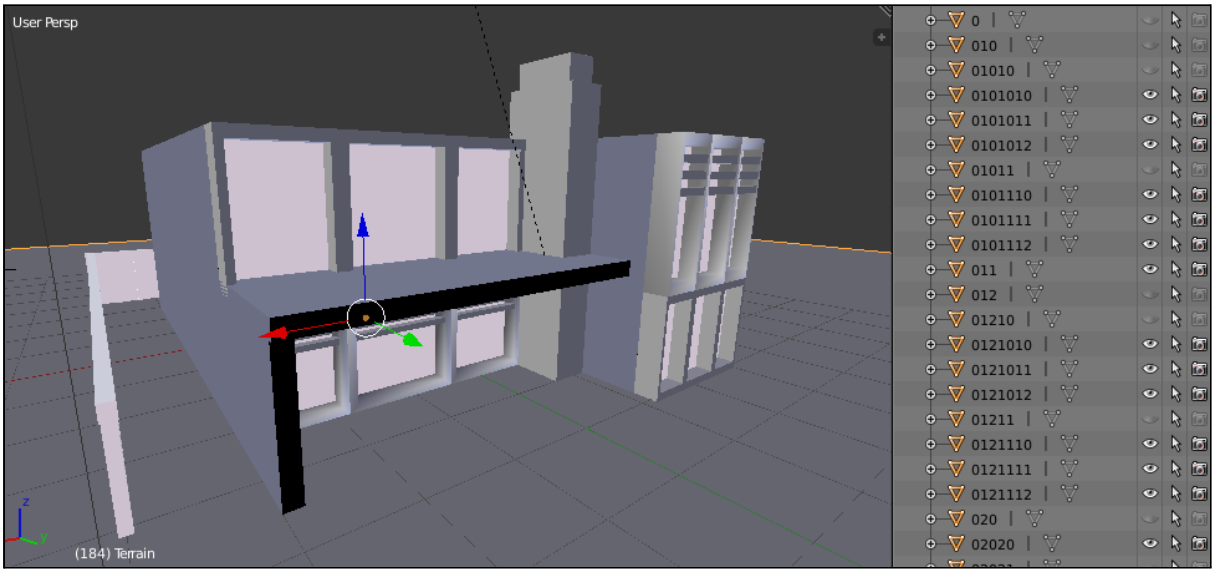

Figure 39: Example of model obtained with a set of rules in 2 layers

The best part of this addition is that it allows for several variations of the model and still obtain great results. The shape obtained from the second layer is, in fact, symmetrical to the other end of the house. That means that if another instance of the house generates with the extruded part that is located currently on the right side on the left side instead, the second

layer will still be visible and blended in perfectly. It is also important to notice how in the hierarchy, we can differentiate the blocks and in what layer they are located. All of them start with the number 0 on their name, since the first block and parent of all the other blocks is named 0. Then, we have an identifier for their layer, which will be 1 or 2, as we only have 2 layers on that model.

However, this presents an issue. Even though we discussed previously that occlusion can happen when dealing with multiple layers on other more traditional approaches, we will be dealing with another issue, Z-fighting. This is a particular situation when two primitives are very close to each other or directly overlapping with one another. When it happens, since both faces will be at the same distance from the camera, the viewer will try to render both of them at the same time, leading to undesirable outcomes. This was not an issue up to this point in our application because of the management of duplicate blocks. Whenever a subdivision was performed, the parent block was hidden to avoid having coplanar faces all over the scene. But, adding extra layers of blocks will be an issue, since there is no control of their situation with respect to the other shapes.

Even though the problem is important, the solution used is rather simple compared to the other problems presented on other multi-layered CGA grammars. First of all, we will join all blocks that are from the same layer. A rather simple task as their ID shows that information as it was explained earlier. Then, all layers will be iteratively joined to the first one applying a small scale offset to the second one. This is a very simple trick that has proven to deal great results without significantly altering the final model and avoiding any Z-fighting both on the viewer and on the renders.

5.8. Prefab design

At this point of the project, we have implemented the CGA grammar, which is arguably the core of the application, but we are missing some details on the visual part of the results. We presented the idea of adding prefabs to the model, and how with a very small amount of work, you can achieve great results without adding way too many rules to the ruleset. So, it is very important to know how to model those prefabs and the constraints that you have to keep in mind when modeling them so that they can be used in the grammar.

First of all, we will talk about how to position them, and then we will discuss the different types of prefabs that are likely to be added at some point. Below, we have a figure that has, what we will call, the template cube. However, we will only be interested in the face that is marked and which normal is pointing closer to the camera. It is very important to understand that this cube is a representation of the block where this prefab will be attached to later on, and it will look and be positioned in the same position afterwards.

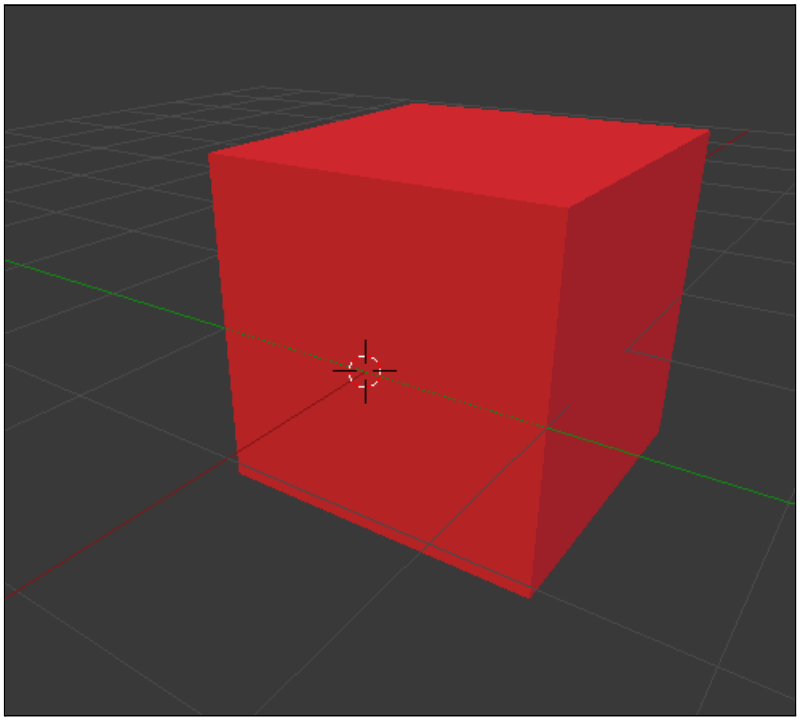

Figure 40: Cube template to design prefab models

Obviously, not all the blocks that will be created with the application will be a cube, let alone have the same dimensions. As it was hinted at in the prefab addition section of this paper, the prefab will be scaled. It is a necessary step so that the instantiated model has the same relative size to the block as when it was modeled. This means that the user is only required to create the prefab once, without needing to save several of them with different scaling or rotation.

5.8.1. Simple prefabs

Following these guidelines, we present an example on how a prefab that will not have any further properties or particularities should be created.

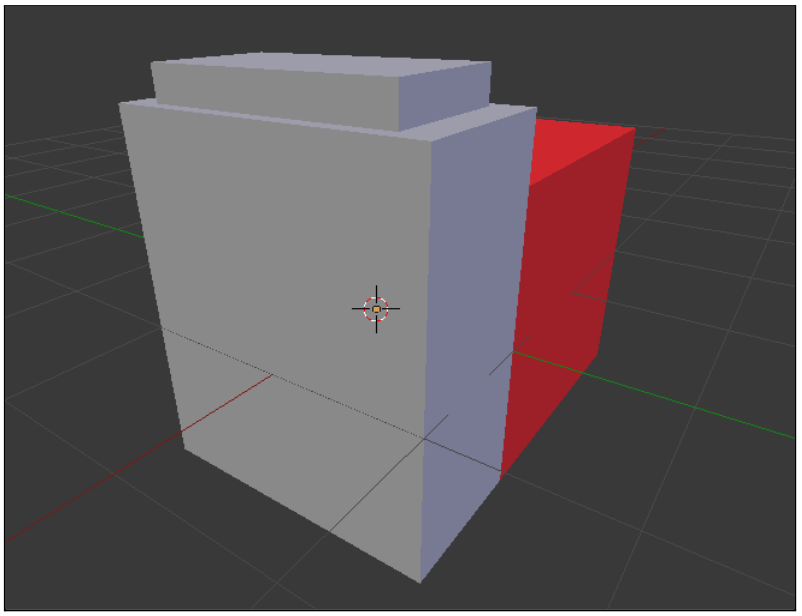

Figure 41: Chimney prefab model with the template

Even though it doesn't look like it, the prefab displayed above is a very simplistic chimney. In section 6.3 we could see the potential of such a prefab and how it can have different sizes and shapes, even before adding its texture. There is no need to add any extra specifications when using simple prefabs. However, that will not be the case for all of them.

There is a variation of what we define as simple prefabs. Those are the roof prefabs. For simplicity in both the coding and prefab creation, we decided that any prefab that will be used as a roof, or the superior face of a block, has to be marked as such. There will be a setting section on the script, which we will discuss in more detail afterwards.

5.8.2. Prefabs with interior view

The last type and probably the more complex kind of prefab are prefabs with an interior view of the house. Since modern houses are commonly more open to the exterior, with big windows and similar elements, we decided it was important to get such an impression. The most important unique aspect of these prefabs is that it will consist of two different models instead of one. The first one will be anything the user wants it to be, a door, a window or any other design. The second one will be the interior view. Below we show a very simple example of a prefab with an interior view.

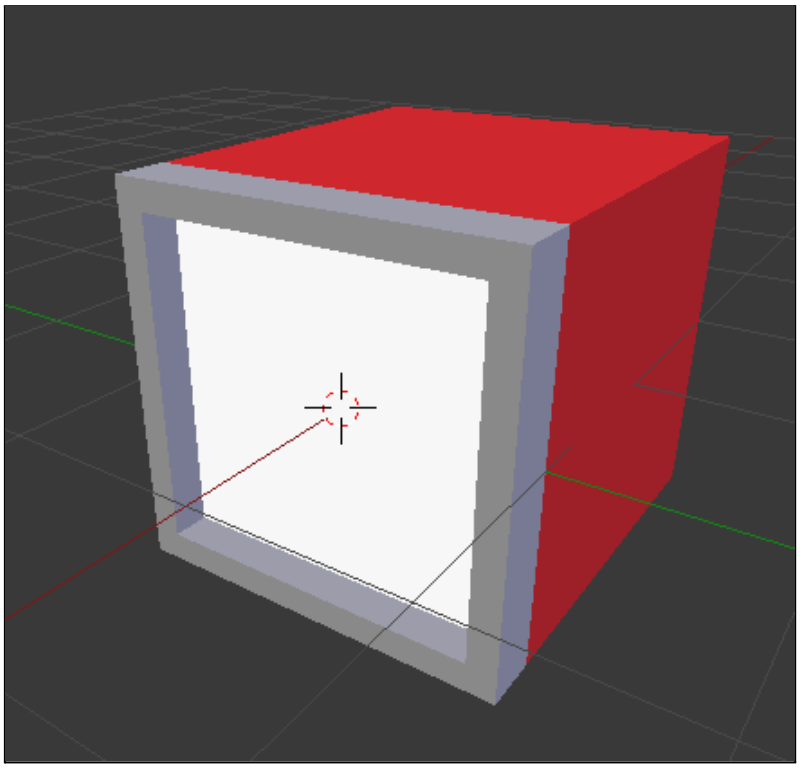

Figure 42: Window plus interior prefab with the template model

This prefab represents a large window that will be as big as the block's face where it will be instantiated. A very important constraint for the design of these models is that the interior view model should be named "Interior" or at least start with such a name. This was done in order to differentiate which one of the two models is the interior view and which one is the other more generic part.

Following the example on the figure, the frame of the window will be treated as a simple prefab, where the user will be able to assign any texture to it. However, the interior model will have one of the four interior textures that are included by default on the application. These textures are based on wide-angle photographs of some room interiors that you can find in a modern house.

Figure 43: Wide-angle photograph of a house interior

The reason why such images were selected is to give the user an illusion of depth. At first, some regular interior images were used for this feature, but it didn't matter where the camera was placed that the view would remain completely still and would lack some realism. For that reason, wide-angle pictures were used so we could select only a portion of it. The offset of such a portion would vary in relation to the z-angle of the camera's orientation in the scene. In addition to that, a slight vertical crop was done to reduce the distortion on both ends of the texture due to the nature of wide-angle photographs.

5.9. Environment

This final part of the development was never planned as such when deciding the objectives of the project or the minimum features it should have. However, it was decided afterwards that a good appearance besides the house model would be a huge attraction for the user to enjoy the application. With some extra elements added to the scene, the renders obtained will be much more realistic besides the house model, which might give us a bonus towards other applications we reviewed earlier.

The list of environment features that were added to the application are:

- Terrain and paths
- Background
- Day and night modes
- Extra light sources on the scene

First of all, the easiest way to add the feeling to the user that they are building a house model, is to place the resulting model on top of a terrain. By adding a plane, which dimensions can be adjusted to the tastes of the user and adding it a texture it can significantly improve the quality of the renders. By default, a grass texture is used on the terrain. Of course, considering that the terrain will be quite large, the texture used is seamless. This allows the use of the texturing features explained earlier to tile it and grant a higher level of detail.

Figure 44: House model with a terrain and roads

To add some extra elements to the scene, it was reasonable to add some paths too, also dependent on parameters that define its position and size. Once again, there is a default texture used, which in this case is an asphalt one, to represent a road. We can see a possible result in the figure above.

The second improvement in quality is the background. Up to this point, by using terrain and some extra elements, we added some content to the lower part of the scene. However, the upper part is still plain and empty, but the solution is rather simple.

Using another textured plane, this time with an image depicting an urban scene with some vegetation for better blending with the terrain, we can generate the illusion of a background behind the house model. We present below another example of the scene, this time with the hand picked picture as background.

Figure 45: House model with an added background

Although adding a background plane would be enough to produce some renders, there are still some issues to be solved. Moving the camera or placing the house model in a different position will ruin the illusion of background behind it. To solve it, the background plane will be converted into a child element of the Camera in the scene hierarchy. By performing this simple modification, the background will move along with the camera keeping the same offset towards it. Of course, the position of the background with respect to the camera is defined by the user's input and can be changed at any time.

The third feature included is a simple, but effective one. As we have seen on some of the examples of modern houses up to this point, not all images were in a setting with perfect lighting or even with daylight. For that reason, we decided to add the possibility of keeping the scene as it is right now in a daylight setting or change it into night mode.

The only two necessary tweaks to obtain the impression that the whole scene is at nighttime where to hide the main light source, in other words, the sun, and lowering the environmental lighting. It was decided after some testing that it will be lowered to a 20% of the value that is used at daytime.

Even though this is a welcomed alternative to achieve some extra appealing renders, pictures obtained with night mode will be underwhelming due to the low visibility. For that reason, we decided to add the fourth and last environmental feature, point lights.

After some discussion on what was the best option to apply some local lights on the scene, it was decided that the best solution would be to instantiate them along with some prefabs. Of course, not all of them would be eligible since it would not make much sense. Luckily, we already defined some different types of prefabs in the previous section, and one of them is perfect for this task, prefabs with interior view.

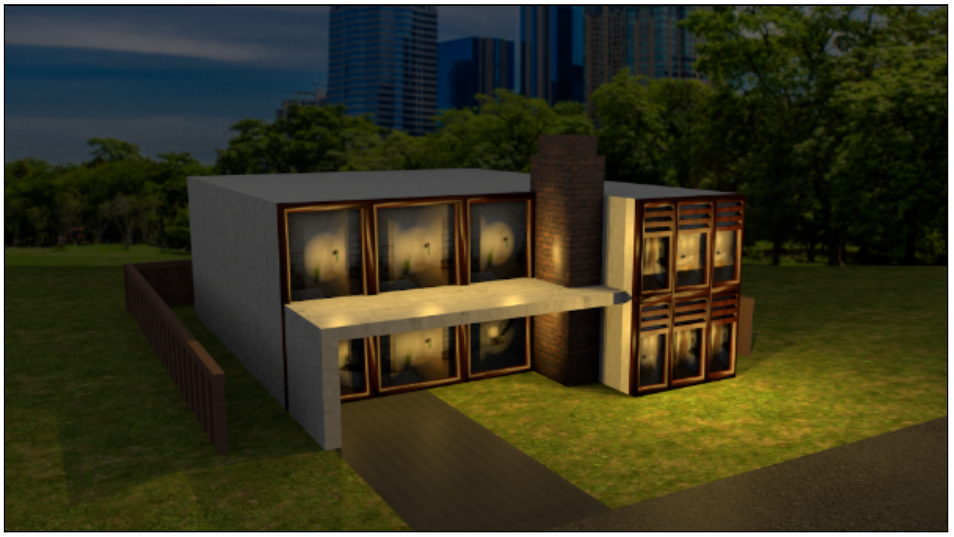

Figure 46: Scene on night mode with point lights added

For each prefab that has an interior view of the house, a point light will be instantiated with a small offset in the direction of the face the model is placed on. With this, we generate the illusion that the light is coming from the interior of the house, as can be seen in the example above. To obtain better details, the light color was slightly changed to a more yellowish variant and the light intensity was reduced to keep some reasonable light levels on the scene.

As a bonus feature, it was decided that the materials used to represent the inside of the house had a positive value of light emission. What that means is that besides the point lights that were added to the screen, the texture itself will also emit light, giving the illusion to the user that the light source is inside the house, rather than outside, as it is seen on the figure above. The value designated after some testing is 0.7, which gave some visually appealing results. The changes obtained with this modification will be seen in the results section.

6. Implementation

6.1. Hardware and software

This project was designed, implemented and tested on a rather low-end machine. An ASUS A55V with Intel Core i7-3610QM CPU and 4GB of RAM. The software used is Blender, particularly, in the version 2.79b released for Windows OS. The whole application was developed inside Blender with the use of the embedded Python interpreter it possesses by default.

The whole code is included in a single Python script that is attached to a Blender file for simplicity for both coding and running. The script contains the following sections:

- Configuration: Some global variables that need to be specified by the user, like the path to the file on their computer or some details to the house generation.
- Input: The place where the user defines the ruleset.
- Globals: Some global variables needed for the correct behaviour of the application.
- Classes: the Block class, representing each object of the scene that goes through the grammar at some point.
- Selectors: some functions used to select faces of a block based on the direction of their normal.
- Extrusion: all functions needed for the extrusion operation.
- Subdivision: all functions needed for the subdivision operation.
- Texturing: all the functions needed for the creation of materials and loading of textures from images the user will apply into the model.
- Prefab addition: all functions needed for the prefab addition operation and loading of models created by the user.
- Object manipulation: A set of functions that are in charge of performing some functionalities that are mostly embedded on Blender. They are divided between object selection, merging and hiding.
- Production of rules: functions that will read the rules specified by the user and apply them in order of reception by using a queue to process them and add blocks to the scene.
- Environment elements: several functions that add some environmental objects or effects to the scene
- Extra functions: some extra functions needed to ensure the whole application works properly
- Main: the main function

6.2. Feature overview

After finishing with the implementation of the project, we can do a brief overview over what has been implemented and what was not. To do it, we will go over everything that was planned at the objectives and the planning back at sections 3 and 4.

Fortunately, all the intended features that were decided to be implemented as part of the minimum work were successfully added to the code. The basic functions were implemented, which in the end, where the ones expected to be added. The CGA shape grammar is working as intended and it allows the use of multiple layers. All the objects in the scene are textured and allow tiling to increase their quality. Finally, there are a set of prefabs that were used to replicate some real house designs that will be seen on the results later on.

However, not all the extra features were added into the final version of the application. The most relevant one might be the absence of a parser to read the grammar from an external file. Even though this was originally thought to be a rather important aspect of this project, it was decided after some discussion that it was not essential. The priority was set towards a general improvement on the quality of the results rather than on the usability from now on. So, the application will remain restricted to those users that are somewhat familiar with Python and Blender. The more complex prefabs that were also referred to as the extra features would also have been a nice addition. However, this is also in the hands of the user, that will decide the level of complexity his models should have. If the user wants some more complex prefabs, he can create them at any time. Finally, in terms of texturing, it was decided that the method used produced good results in most cases, and did not requiere further dedication.

Of course, all of these features can be added in the future, since it does not mean that the application can't be improved any further.

It is also worth mentioning that there are some features that were added that were not even on the original plan. As a reason for the realization that good results that were appealing to the user were more important than its usability, there were some extra functionalities added to the code. This is the case for all the environmental elements added that were explained back in the Environment section.

7. Results

After all this detailed explanation on the concepts that were involved in the project, their application to the project and finally the implementation, it is time to see some results. There are a lot of different ways to test the application since we intend it to be useful and unique in many different aspects.

7.1. Renders

For this first section, we will present three real house designs and we will try to generate a 3D model that resembles their overall appearance as much as possible. All of them have been already exposed in previous sections of the paper as they represent most of the unique elements that you can find on the houses that relate to the topic of the paper. Moreover, their constitution fits the frame of the application and it is very likely that we can obtain good results when trying to replicate them. From now on, we will refer to them as houses 1 through 3 when displaying their virtually generated replicas with renders.

Figure 47: From left to right, house 1, 2 and 3

It is worth noticing that for all three examples, only one side of the house is visible on the images. Hence, all the models that will be generated in this part of the project will only be detailed in a single side too. Even though our program can work on different sides at once, we will save it for some posterior examples. Since the most amount of detail and interesting elements of the facade are usually located on the front side of the house, it doesn't mean these examples will be oversimplistic.

Let's start with house 1. You may have noticed that this was the model that was followed during the development of the application, since most of the pictures taken from Blender showed a similar model. This time we will present the final result of this model that we have been working on.

Each model will try to take advantage of different features that have been implemented during the development of the project. For instance, house 1 will be rendered at nighttime, since the original picture was taken with a low amount of environmental light and the interior of the house is shown with the lights on. To obtain more realistic results, ambient

occlusion was activated with the value given by default on Blender, 1. These are the results obtained:

Figure 48: Render of house 1 (variation number 1)

Figure 49: Render of house 1 (variation number 2)

Figure 50: Render of house 1 (variation number 3)

The first point that should be explained is the fact that there are three different realistic interpretations of the house. Even though the result that is closest to the reference is the one in Figure 48, we have two extra variations. One of the most important aspects of any CGA shape grammar is the fact that the results don't have to be deterministic. We took advantage of that and designed a simple set of rules that can be seen on the Annex as the code used to generate house 1. The most noticeable changes between all three models are the position of the chimney and the extrusion of a part of the house towards the camera. But, if you pay close attention, there are other details that are randomized, such as the window type. Two different window prefabs were added to the rules and they will be used randomly. Also, the dimension of the house is slightly different, since most of these values are also given by a range.

The reason why all of these combinations are possible and deal valid results is the fact that there are two independent layers on the scene that merge perfectly. As it was explained in the development section, the inverted u-shape extrusion that divides both floors of the house is generated by the second layer of the grammar. Since we don't have a problem with it going inside the house in some sections, we can obtain good results without knowing what the resulting shapes of the first layer will be. This is one of the most important reasons why using multiple layers is a good option for these kinds of models and how it simplifies the grammar by a large margin.

Another remarkable aspect of all the renders shown is the fence that is around the house. It may seem as if it was an external element added after the house was generated, but it is not. It was never said that a prefab had to be attached to the model. The fence prefab was designed to be one unit apart from the face it would be instanced on, which in this case are

the sides of the house except the front one. After a proper scaling given the dimensions of the house, the result is a simple fence that borders the house. This is one of the many uses the addition of prefabs can have.

This house model is also very important to test one of the features that were implemented, prefabs with interior. In this particular model, all the variations will have a total of 12 instances of such prefabs. Moreover, each of them has a very large frame which allows the interior texture to be highly visible. Even though the scene was set to night mode, the visibility of the interior is great. This is thanks to both the illumination obtained from the point lights that were instanced close to each prefab and the light emission from the material itself.

As a whole, the results have a good quality level, considering that the amount of rules used to create this model was particularly low and that the amount of variability is quite noticeable.

Now it is time to see the results obtained for house number 2. On this occasion, trying to follow the real design shown in the picture as much as possible, the scene will be set to daylight. Aside from that, the rest of the settings will remain the same as in the previous renders. These are three different instances that were obtained with the rules that can be found on the Annex as code for house 2.

Figure 51: Render of house 2 (variation number 1)

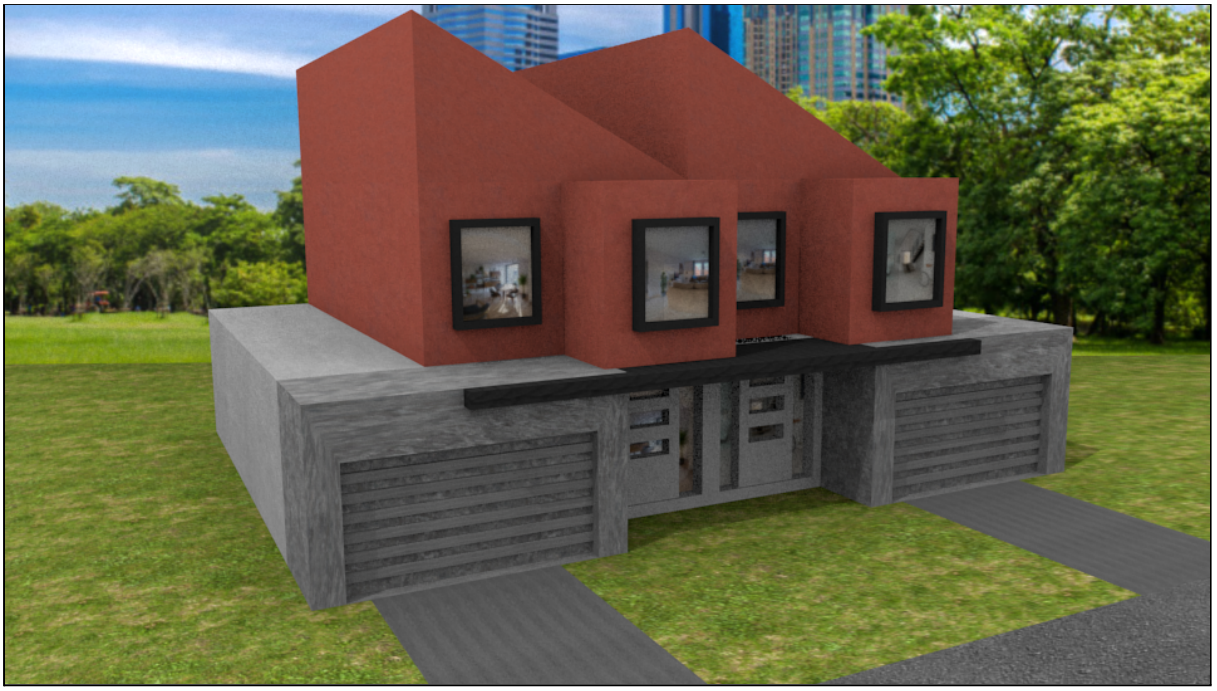

Figure 52: Render of house 2 (variation number 2)

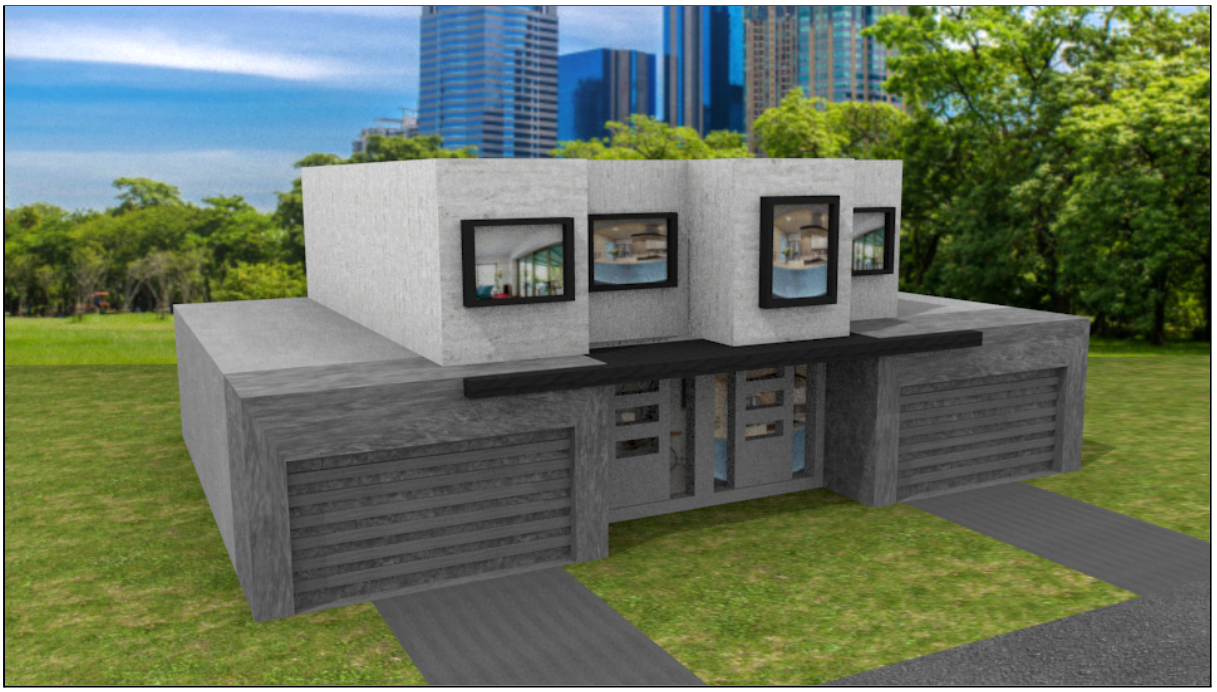

Figure 53: Render of house 2 (variation number 3)

The first impression on all those three models is the fact that the variability between all of them is even larger than the one seen on house 1, specially on the second floor of the house. Once again, the first result, the one seen on Figure 51 is the closest to the one observable on the original picture. Once again, the objective of this project is not to replicate to the closest

detail already existing designs. Instead, we present three more models of modern houses that were created in a short amount of time with a similar style.

The first difference we can spot on these designs are roof prefabs. Some of the models include one or more instances of one of the simple roof designs that are included with the base application. As you can see, the amount of detail on that roof is minimal, but it matches the overall design of the house after the texture application and completely changes the outcome. This is one of those situations where it is clearly shown how a small time investment on creating prefabs will heavily contribute to the quality of the final result.

These designs also present some new regular prefabs, such as the garages and the doors. Both of them have an actual higher level of detail than the other prefabs seen up to this point. However, they are still rather simple and very easy to model.

Finally, even if it is less noticeable than in the models of house 1, there are two layers of blocks present on the scene. This time, the second layer only consists of the black division between both floors. It is a simple but effective addition, which would take much more effort to include if there was only the possibility of creating one layer of elements.

Now, it is time to see the final of the three sample models. This will be a representation of house 3. The code for its representation is also included in the Annex, where you will be able to read all the rules used to create such models. First of all, we will use the same settings as on house 1 to use a nighttime environment.

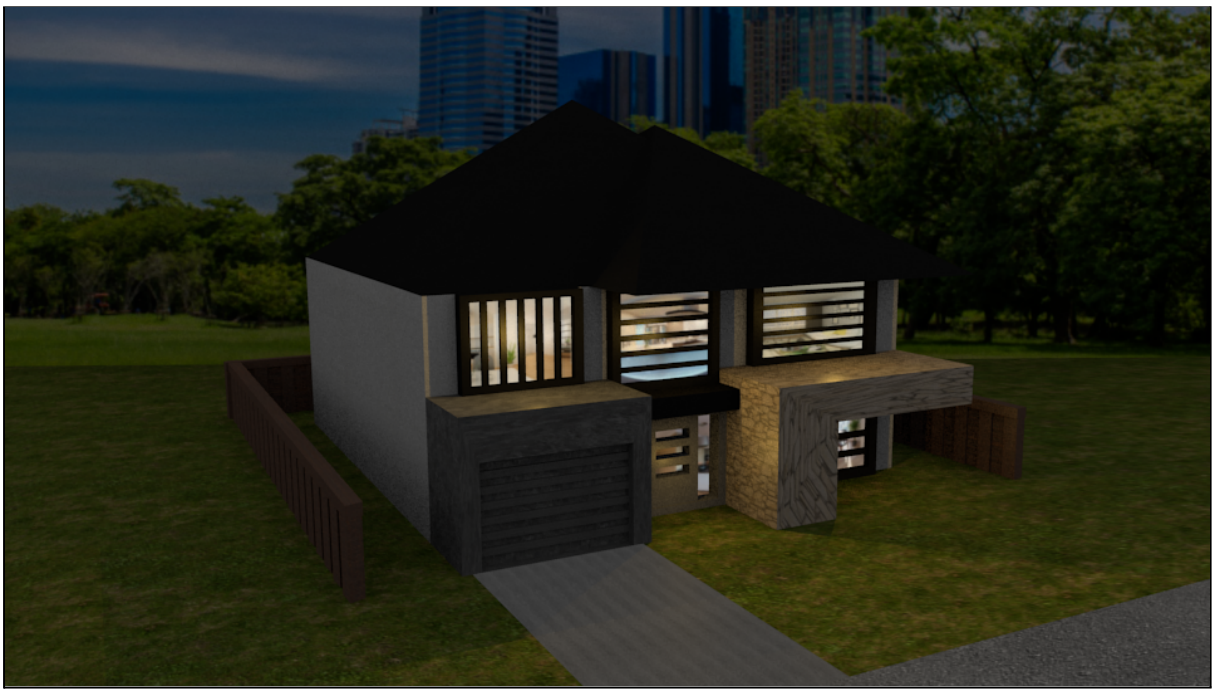

Figure 54: Render of house 3 (variation number 1)

And now, we will render a similar model but in daytime. The render settings are also identical to the ones used on house 2.

Figure 55: Render of house 3 (variation number 2)

The reason why both models are so similar is that this time the ruleset used is more deterministic. To provide another way of using the application, it was interesting to see how the results will produce themselves when restricting the randomness of the results. In this case, we can both see how both models are very similar to the original house design presented on Figure 47. The only slight differences that may be hard to spot are the house dimensions and proportions between the blocks, which are slightly randomized. The window prefabs are also chosen at random between two options: horizontal or vertical bars traversing the frame.

Even though it was not necessary, since we have the possibility to use it, we decided to divide the block generation into two layers once again. The two prefabs on the lower right corner of the front facade were added from different layers.

Talking about prefabs, there is one particular flaw that is visible on one of them. The inverted L-shape model on the lower right corner has some issues with texturing. As it was explained, the texturing method used is rather simple, and this is one of those situations where its flaws are visible. This problem could be solved by implementing a more complex texturing technique, like the one that was proposed back then on Section 4.3. Besides that, the results are on the same line as the previous models.

After seeing all three designs being represented with our application, we can say that the results obtained are, at least, visually pleasant and similar enough considering the simplicity of the ruleset used.

However, there are still some other features that need to be tested. First of all, there is no limit in the amount of house models you can represent at once. For instance, we present a render that shows three different models generated at once of house 1.

Figure 56: Render of 3 variations of house 1

The only needed step to obtain a scene like this is to generate more lots, placing them on a proper spot on the scene, and adding them to the queue in order to apply the rules. Of course, some extra details were added, such as the extra asphalt paths that lead to each of the house models, but those are entirely optional and up to the user to decide whether to add them or not.

Up to this point, all the house models had one thing in common, all the details were concentrated in one of the sides of the model. Even though the results were quite satisfying, our grammar is not limited to such models. For that reason, we decided to create an additional model in order to display it here. We will name it “house 4”, and this time it will not be based on any real design, it will be a free creation from scratch which we will try to represent using all of the features of our software.

The main objective is to add some detail in all the facades of the house, instead of only the front one. We will also use the same set of prefabs that were used on the previous house models, to show the modeling potential with no necessity of creating more of them. Of course, we will try to benefit ourselves from the multi-layered nature of our CGA grammar

when incorporating more details to the model. The ruleset can be seen, once again, on the Annex of this document. This is how the resulting model looked at the end.

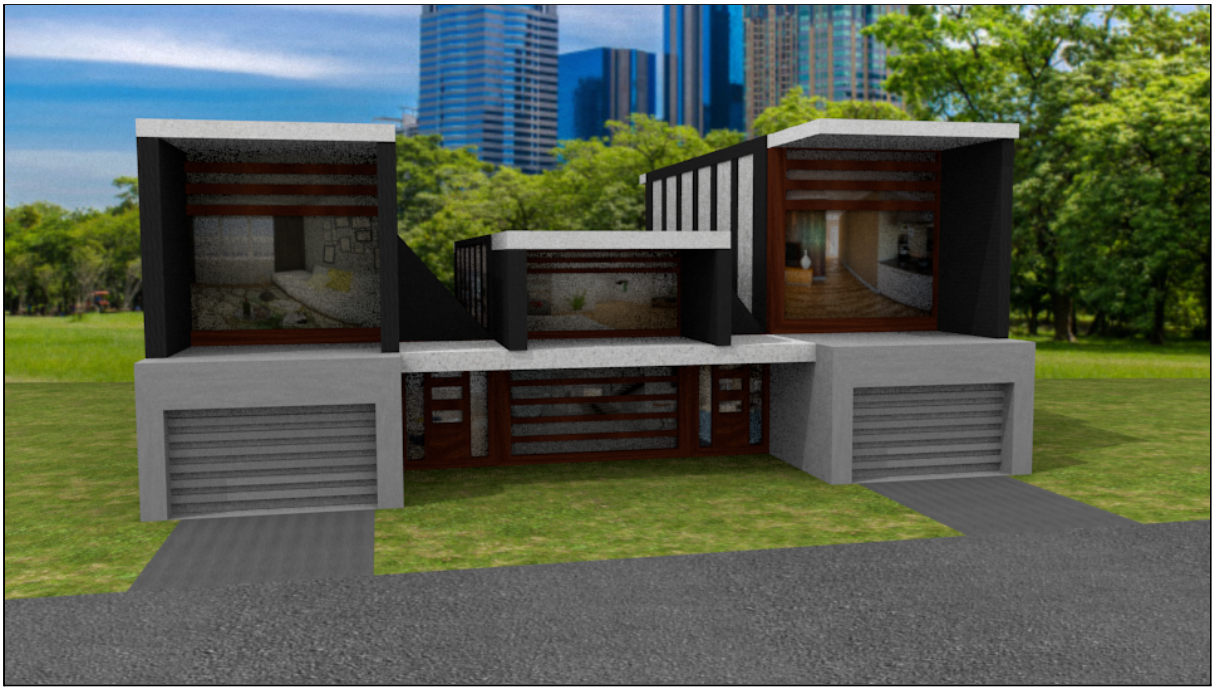

Figure 57: Render of house 4 (front view at daytime)

Figure 58: Render of house 4 (back view at daytime)

And now, another model obtained from the same set of rules. This time at nighttime to obtain a different point of view of the result.

Figure 59: Render of house 4 (back view at nighttime)

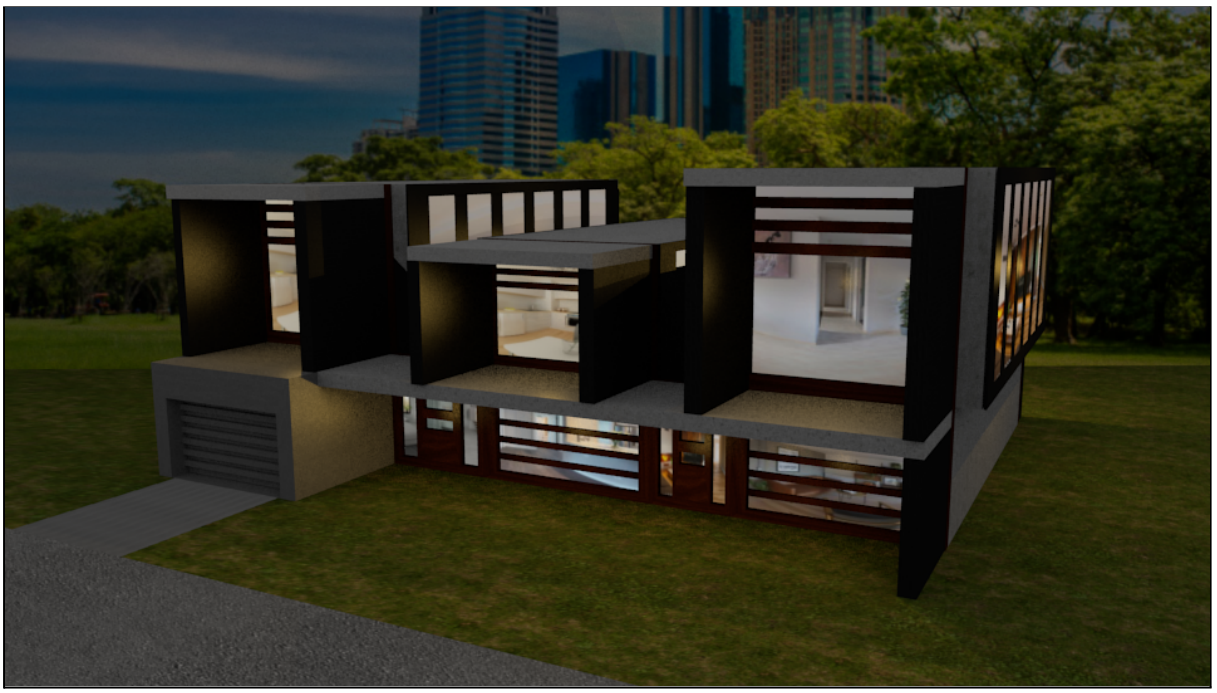

Figure 60: Render of house 4 (front view at nighttime)

Even though the design of the house itself might not be great, it served its purpose. This model, which is a whole level more complex than the previous three, shows more of the

potential of the application. Prefabs and other operations can be added and performed into any direction, following different combinations doing so. All of the elements are positioned correctly and rotated as they are supposed to be.

All in all, it has been shown in this section how different models can be designed when using our application. It is just a matter of how the user decides to write the rules and the assets he is willing to use or create.

7.2. Application Performance

After witnessing the results that can be obtained when rendering a diversity of models obtained through our algorithm, it is time to analyze its performance. To do it, we divided the whole execution of the script into different parts, naming them to acknowledge the partial times obtained on each of them. These are the different parts:

- Preparation: Some adjustments on the editor plus removing all the objects on the scene from previous executions.
- Lot generation: creation of the lots and their addition to the queue.
- Rules: the core of the application, the production of rules for all the lots added on the scene.
- Layer merging: if the house model has more than one layer of blocks, they will be merged properly here.
- Environment: addition of elements on the scene plus the lighting adjustments depending on whether day or night mode is activated.
- Total: the total amount of time needed to execute the whole script.

And this is an example of a prompt in the console obtained after generating three models of house 3.

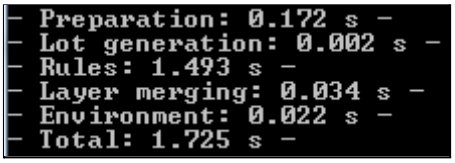

Figure 61: Example of application performance output on the console

However, for simplicity, the tests that will be run for this section will only include one model per script execution.

Before starting with the actual execution time testing, first of all, we should take notes on the amount of geometry we will be working with. It will be very important to understand the results obtained later on. The results provided on the table below are a result of an average of 5 repetitions. As it was stated clearly, most models are not deterministic, which means the amount of geometry will vary in each execution. These are the results.

	Vertexs	Faces	Triangles
House 1	1,196	884	1,896
House 2	518	380	864
House 3	1,142	850	1,764
House 4	1,562	1,132	2,590

Now, we can do the timing test. However, there are two small details that we need to take into consideration. The first one is regarding the preparation. Since that step involves the deletion of every element on the scene, that split may deliver different results if the previous model on the scene is simple or complex. To avoid those inconsistencies, we will do six repetitions instead of five, and we will ignore the results on the first one. This way, we can consistently remove the same amount of geometry each time. The second detail is about the environment. We believe that the differences between generating the model using daytime or nighttime might affect the results. For that reason, we will perform two different tests, one on each mode.

These are the results obtained on daytime:

Daytime	Prep.	Lots	Rules	Merging	Env.	Total
House 1	0.008 s	0.002 s	0.586 s	0.020 s	0.030 s	0.637 s
House 2	0.007 s	0.001 s	0.359 s	0.014 s	0.033 s	0.416 s
House 3	0.012 s	0.002 s	0.455 s	0.015 s	0.035 s	0.520 s
House 4	0.014 s	0.002 s	1.238 s	0.018 s	0.031 s	1.305 s

And these are the ones obtained on nighttime:

Nighttime	Prep.	Lots	Rules	Merging	Env.	Total
House 1	0.106 s	0.002 s	0.589 s	0.020 s	0.019 s	0.737 s
House 2	0.095 s	0.001 s	0.364 s	0.015 s	0.023 s	0.499 s
House 3	0.110 s	0.002 s	0.464 s	0.013 s	0.021 s	0.611 s
House 4	0.141 s	0.001 s	1.240 s	0.023 s	0.024 s	1.430 s

Even though it is not present in any of the tables, we also tested the rendering time when obtaining results as the ones seen in the previous section. The images, with a size of 960x540 pixels, took between 4 and 5 seconds to generate.

Returning to the results in the tables, there are some already visible pieces of interesting information in them. However, it would be better to display them in a different way to appreciate visually some of the larger differences that are present on these results.

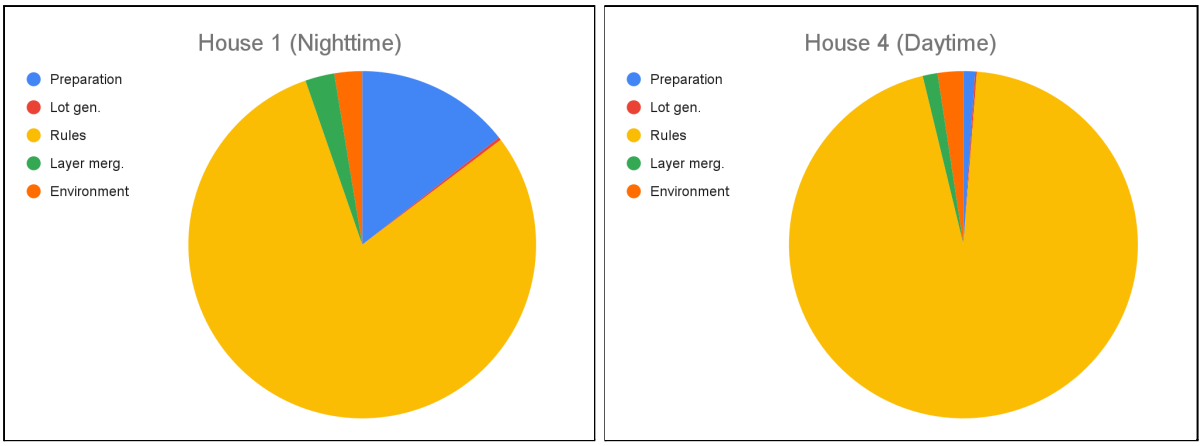

Figure 62: Pie-charts representing the execution time of the creation of houses 1 and 4

To begin with, we can tell by using the graphs on Figure 62 how the production of rules takes the largest amount of time in comparison to the other splits. In both of the examples chosen, the percentages of time on the Rules part are both 79.9% and 94.8% respectively. However, there is another detail that can also be shown by placing together some other results.

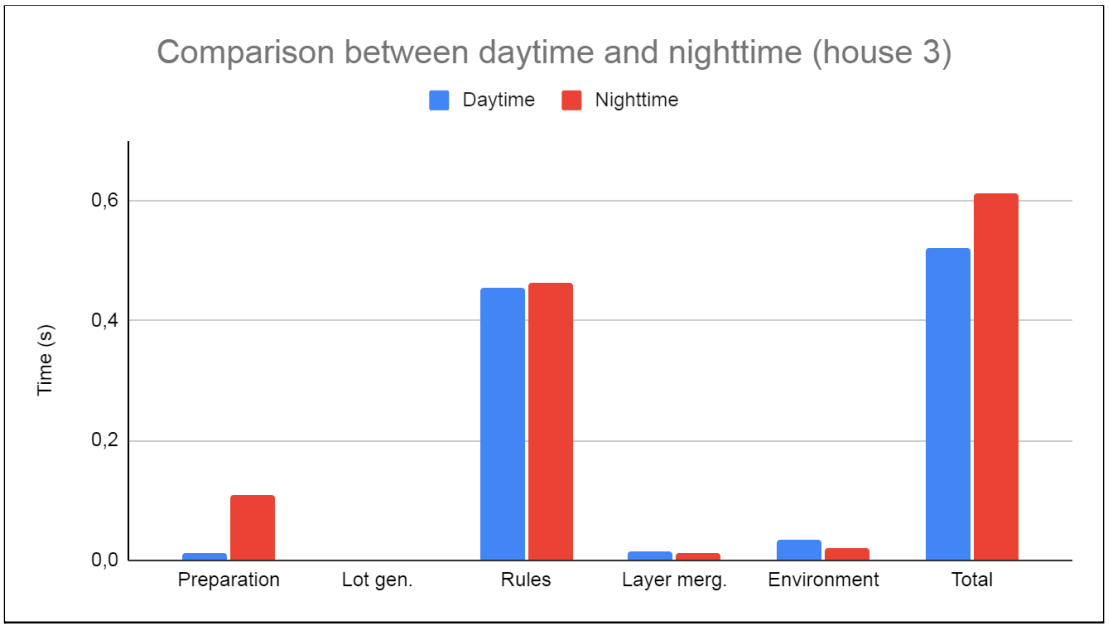

Figure 63: Bar-chart comparing execution time between daytime and nighttime modes using house 3

In the figure above we can see how, in most of the sections, there is no effect when switching from day to night mode. However, there is one that suffers major changes, the preparation. This is most likely due to the fact that when using night mode, it is necessary to remove all the point lights from the scene that were added from previous executions. Which adds more complexity to the task.

7.3. User Performance

The performance of the application is a very important aspect to know how well it performs and what parts of the algorithm take more time to complete. However, these kinds of applications require some extra performance testing to actually tell how useful they can truly be. In this section, we will analyze the performance of the user when taking advantage of the software we developed versus not doing so.

The test will be fairly simple. The goal is to generate a house model, no need to follow any real life design, and see how fast we can obtain a decent model. This was one of the reasons why we presented house 4 on the Renders section, as it was the result of this particular test when generating a model with our application.

Before jumping into the results, let's do a quick review of the steps needed to create a house model with our application and without it. First of all, when using our application, the first step is to gather all the needed materials. These range from textures to prefabs. Of course, one possibility is that the user decides to benefit from already existing prefabs and materials which either he found or modeled beforehand or that were already present on the default application. After that, the only step needed was the definition of the rules on the Python script provided. However, if the user decides not to use our application and model the house directly through the Blender interface, he will have to do the following steps. First of all, he will have to manually transform or add one or several 3D shapes into the scene to generate the basic shape of the house. After he is done with that part, he will have to apply all of the texturing with UV editing, which might be tedious if the house model is rather complex.

For this test, I will personally be in charge of performing both tests. I feel like I am a good test subject for it since I have a balanced knowledge for both my rule system and the Blender interface. Even though I designed and implemented the software of the application, I have only modeled 3 houses with it, so I believe I will work with it at a reasonable pace. Same situation for the Blender interface, I have worked with it in the past, but I am not a big expert with it, so I feel like I have a similar level of skills for both tests. It would have been great to have more test subjects for this experiment, but unfortunately I was unable to find more people with both Python and Blender skills after the application was finished.

And now, for the results. The design of what the target model should look like, took about 6 minutes, but that time is not relevant for the testing. The most interesting part is that the whole process of writing the ruleset and adjusting some of the parameters to obtain greater results took only about 8 minutes. The part of the modeling where I was supposed to look for textures and prefabs was omitted since I already had everything I needed, which is one of the strong points in favor of the application. Once you have a prefab or material, you don't have to create it again.

After that, I decided to perform the second test, with the default Blender interface. It was not long until I realized that I would not come close to even getting somewhat close to the results obtained on the application. I decided it would be a good indicator to show the

results of the two models that I obtained after 8 minutes of work with the same target house design. These are the results.

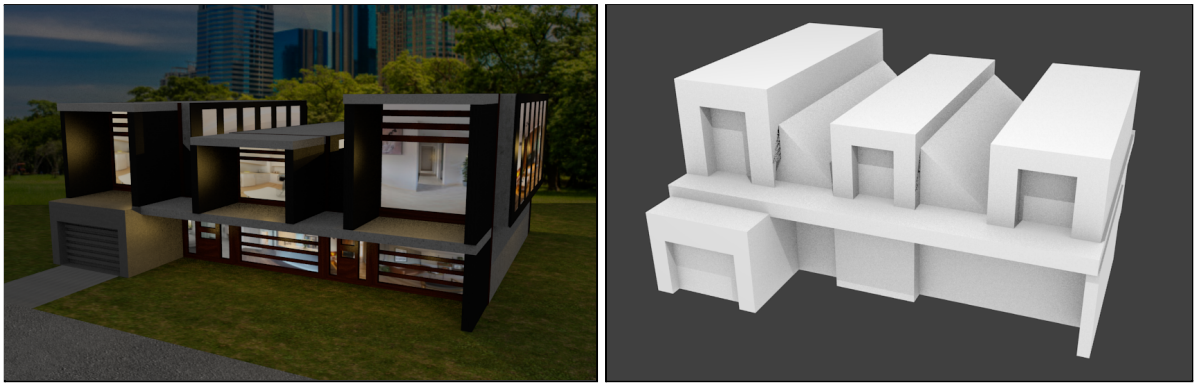

Figure 64: Similar house model design created with our application (left) and manually on the Blender interface (right) after 8 minutes of work

It can be seen how I was not even finished with the general shape, let alone the smaller details and texturing. Also, some small visual issues are present as a reason for the fast paced work I was doing in order to keep up with the other method. It should be pretty obvious which method is faster and deals better results.

There are some points that need to be addressed though. One of the main reasons why our application was so much faster is the fact that all prefabs and materials were already created. But, as much as it would slow down the first test, it would only have been by a few minutes, since all the prefabs used on the model of the picture are very simplistic and can be created in 1 or 2 minutes. I believe that the amount of skill and experience needed to create the model on the left of Figure 64 on the default Blender interface, in 8 minutes, is too high for an average user to obtain. It is also worth noticing once again that, as seen on section 7.1, the application can return similar models in a short amount of time. Manually modifying the model would take the user even longer, which gives another upside to using our software.

7.4. Comparison against similar applications

After analyzing the performance of the application, it is time to compare its results with other similar softwares that are available in the market.

The first one, and perhaps the most similar one to the application developed on this project is BCGA. It provides a rather simple approach to the modeling of houses compared to the other options, but it allows for some good level of detail on its creations.

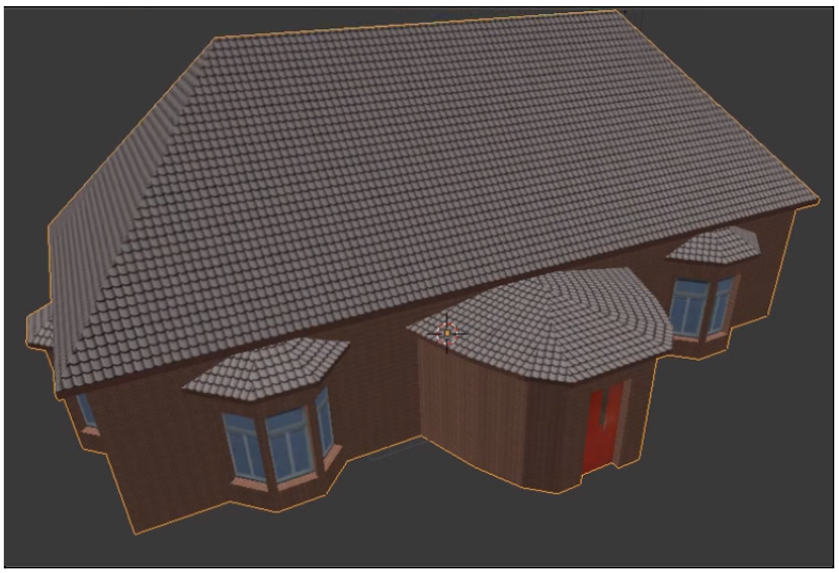

Figure 65: House model created with BCGA in Blender. Source: [Eli]

When comparing purely the result obtained, as the one on the figure above, there are some elements that show that our application deals better results aesthetically speaking. Perhaps, the most important difference is the lack of environment features, which our application uses to accompany the resulting model to create more complete renders. Of course that is not necessarily a downside, since all of those details can be added afterwards, but it leaves the responsibility to the user to handle it. Overall, the house shown above could be easily replicated with our software provided that we design the necessary prefabs beforehand.

If we focus on the methodology instead of the final results, it is the point where our application loses some ground with BCGA. The lack of a parser makes it impossible for us to use text files in order to write the rules to be executed. Even though this isn't really a downside, more of an accessibility issue, there is an important feature BCGA possesses, the ability to manually edit the result on the scene. They are able to transform each of the elements, as well as the general dimensions of the model by using the interface, instead of rewriting the ruleset. This is a feature that we should consider to include in the future.

Now, it is time to jump into another two more professional tools that we will unite for this section due to their similarity in results. These are some examples obtained from CityEngine and Blender-osm.

Figure 66: City representation made with Esri's CityEngine. Source: [http://www.andrewweyrich.com/images/paris2.jpg]

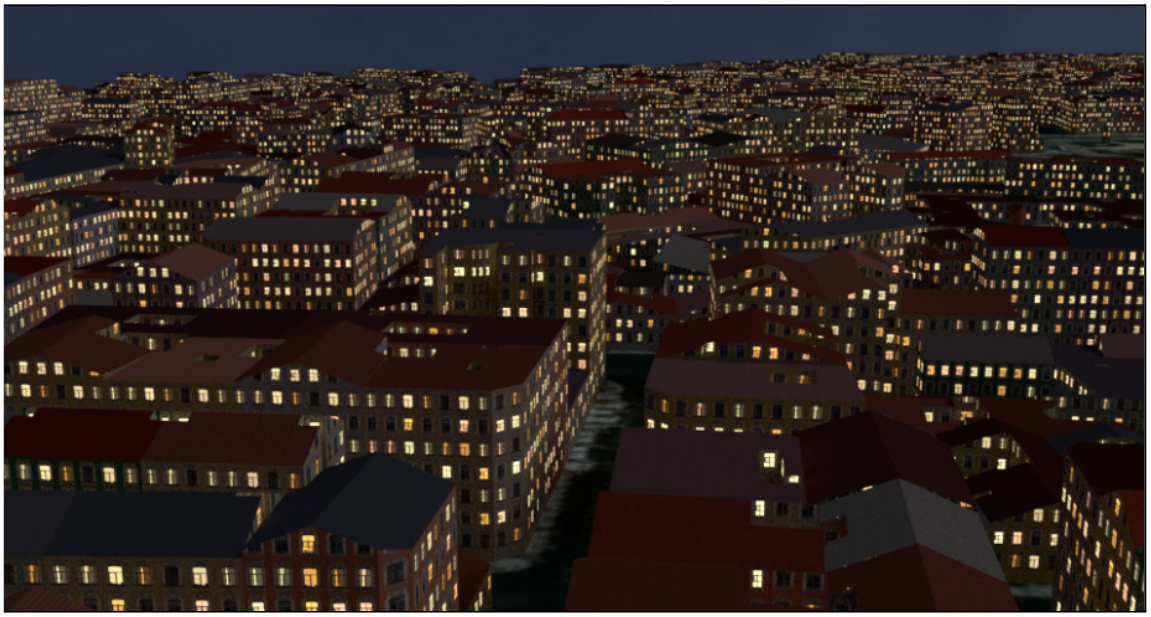

Figure 67: City representation using Blender-osm extension. Source: [Pro]

Both of the render results obtained from these applications have an incredibly high quality and really give the illusion of witnessing a real life city. There are however a few details that, in my opinion, make them less powerful when designing modern houses as our application and possibly BCGA.

Both of these applications are more oriented towards generating a large number of buildings that are common in urban areas. It is easy to assume that they are more commonly used towards the creation of office buildings or multiple layered apartments that are often found in highly populated areas. Our application, however, is more centered towards more complex models, where even though you need a higher number of rules due to the complexity and design of modern houses, they provide more accurate results. CityEngine and Blender-osm use texturing as a way to subdivide a facade and use tiling to display windows or other details.

Another important topic to discuss, specially related to CityEngine, is the learning curve the user has to go through in order to use it. The big complexity of the tool makes it really hard to learn and show its full potential, which is a big downside for a new or inexperienced user. On the other hand, the requirements to use our application are basic knowledge on how to handle Blender's interface and some basics on Python. These could be acquired by the user in a matter of minutes and they could start using our software freely.

A different kind of wall exists when comparing both these applications with ours. The costs of acquiring these applications are really high, which could be inaccessible for some of the potential users. Even though Blender-osm offers a free base version, most interesting features are included on the premium version.

In conclusion, even though the results obtained from both these applications will generally be better and with great quality, I believe that our application will deal slightly better results when designing modern houses.

8. Conclusions

The first conclusion, and most important one, is the fact that the overall results were satisfactory. The application as a whole offers a big realm of possibilities and a good variability in the results if the user wants to. One important detail is that the results obtained can be high quality renders without any extra work required. This means that not only the results are aesthetically pleasing but also fast and easy to obtain.

Another important topic to talk about is the fact that all the minimum work that was stated on the objectives and the planification has been completed successfully. All the features regarding the CGA shape grammar work correctly as it has been proven by all the testing done in the previous sections. Moreover, some extra features were added, even some that were not even considered to be applied from the start. For example, the idea of adding detail to the environment and creating different lighting modes like day and night. All of them were not very complex to implement but helped to improve the final result by a large margin.

Moving on to the performance of the application, it is quite safe to affirm that the results are quite positive. For most of the results, the design of the houses took a few minutes and their generation was performed in less than one second. Given the low-end hardware that was used to conduct such tests, we could say that the algorithm is rather clean and effective.

Related to the performance subject, we can also say that the user performance is greatly increased when making use of the application. It's very simple to use overall and it only requires some basic knowledge of Python and how to navigate the Blender interface. By doing so, the user doesn't have to go through a series of steps to even manually create the house model, let alone prepare the whole environment for any potential renders.

Finally, when we compare the results obtained with our work developed during this project, we can say that we achieved a good solution to the problem proposed. By doing so, we manage to achieve results that compete with other similar applications on the market, arguably obtaining better results when the house to model has a modern design, as the ones we have seen during this paper.

9. Future Work

Even though the results are considered to be good enough in most cases, there are still some flaws or rather weak points that could be improved in the future. For instance, the grammar currently allows a great realm of possibilities, but it would work even better with perhaps some extra operations or a more complex syntax. For example, the addition of boolean operators to the grammar could allow for some interesting shapes by making use of the multi-layered nature of our CGA grammar.

Another topic that should be addressed is the fact that there are some extra features mentioned on the planification section that, in the end, were not implemented. Even though their addition was not fully necessary for the success of the project, some of them would be interesting to have. For example, a parser would allow the addition of text files as an option to write the rules. This would make it way simpler for users without programming skills, which means they would not require a previous knowledge of Python. Another feature that would have been interesting to implement are some extra texturing methods, to increase a bit more the quality of the results.

As it was seen in section 7.4, other applications contain interesting features that could be applied in the future to our own. The most interesting example is the manual edition over an already generated house model. This would allow the user to make some quick updates on the model without needing to modify the ruleset and rerunning the script.

Finally, one last interesting idea would be to create a nicer interface and polish the usability of the application. This would allow us to maybe publish this tool and make it possible so that anyone with Blender can download it and use it to create their own models. This would be the last step on the list after applying the mentioned extra features and upgrades. This will end up making it a strong and easy to use software to model modern houses.

10. Annex

Code used for defining the rules of house 1

```
# House 1
if layer == 1:
    if tag == 'L':
        return extrude_face(obj, 4, [2.5, 3.0], 'B')
    if tag == 'B':
        add_prefab(obj, "fence", 1, 0, 'N', "bricks")
        add_prefab(obj, "fence", 1, 1, 'N', "bricks")
        add_prefab(obj, "fence", 1, 3, 'N', "bricks")
        rng = random.uniform(0.0, 1.0)
        if rng < 0.3:
            return subdivide(obj, 0, [['R', 2], [0.5, 0.7], ['R', 1]], ['F', 'C', 'EF'], "marble")
        elif rng < 0.6:
            return subdivide(obj, 0, [['R', 1], ['R', 2], [0.5, 0.7]], ['EF', 'F', 'C'], "marble")
        else:
            return subdivide(obj, 0, [['R', 1], [0.5, 0.7], ['R', 2]], ['EF', 'C', 'F'], "marble")
    if tag == 'EF':
        return extrude_face(obj, 2, [1.5, 2.0], 'F')
    if tag == 'C':
        return add_prefab(obj, "chimeney", 1, 2, 'N', "bricks")
    if tag == 'F':
        return subdivide(obj, 2, [[1.2, 1.5], ['R', 1]], ['FL', 'FL'], "marble")
    if tag == 'FL':
        rng = random.uniform(0.0, 1.0)
        if rng < 0.5:
            return subdivide(obj, 0, [['R', 0.5], ['R', 0.6], ['R', 0.5]], ['W','W','W'], "marble")
        else:
            return subdivide(obj, 0, [['R', 0.5], ['R', 0.6], ['R', 0.5]], ['W2','W2','W2'], "marble")
    if tag == 'W':
        return add_prefab(obj, "window", 0.5, 2, 'N', "wood")
    if tag == 'W2':
        return add_prefab(obj, "window2", 0.5, 2, 'N', "wood")

if layer == 2:
    if tag == 'B':
        return subdivide(obj, 2, [['R', 1], [0.15, 0.2], ['R', 1]], ['CB', 'CO', 'N'], "marble")
    if tag == 'CB':
        return subdivide(obj, 0, [[0.15, 0.2], ['R', 1], [0.15, 0.2]], ['CO', 'N', 'CO'], "marble")
    if tag == 'CO':
        return extrude_face(obj, 2, [1.5, 1.6], 'N')
```

Code used for defining the rules of house 2

```
# House 2
if layer == 1:
    if tag == 'L':
        return extrude_face(obj, 4, [3.5, 4.0], 'B')
    if tag == 'B':
        return subdivide(obj, 2, [['R', 1], ['R', 1]], ['FL1', 'FL2'], "concrete")
    if tag == 'FL1':
        extrude_face(obj, 0, [0.5, 2.0], 'FL3')
        return extrude_face(obj, 1, [0.5, 2.0], 'FL3')
    if tag == 'FL2':
        return subdivide(obj, 0, [['R', 1], ['R', 1]], ['MID', 'MID'], "concrete")
    if tag == 'FL3':
        return subdivide(obj, 0, [['R', 1], ['R', 1]], ['LEFT', 'RIGHT'], "concrete")
    if tag == 'LEFT':
        return subdivide(obj, 0, [['R', 2], ['R', 1]], ['G', 'D'], "concrete")
    if tag == 'RIGHT':
        return subdivide(obj, 0, [['R', 1], ['R', 2]], ['D', 'G'], "concrete")
    if tag == 'MID':
        rng = random.uniform(0.0, 1.0)
        if rng < 0.5:
            add_prefab(obj, "roof3", 1, 4, 'N', "redconcrete")
            return subdivide(obj, 0, [['R', 1], ['R', 1]], ['EW', 'W'], "redconcrete")
        else:
            return subdivide(obj, 0, [['R', 1], ['R', 1]], ['W', 'EW'], "marble")
    if tag == 'G':
        return add_prefab(obj, "garage", 0.6, 2, 'N', "metal")
    if tag == 'D':
        return add_prefab(obj, "door", 1, 2, 'N', "concrete")
    if tag == 'W':
        rng = random.uniform(0.0, 1.0)
        if rng < 0.5:
            return add_prefab(obj, "window4", 0.5, 2, 'N', "asphalt")
        else:
            return add_prefab(obj, "window3", 0.5, 2, 'N', "asphalt")
    if tag == 'EW':
        return extrude_face(obj, 2, [0.5, 1.0], 'W')

if layer == 2:
    if tag == 'B':
        return subdivide(obj, 2, [['R', 1], [0.16], ['R', 1.1]], ['N', 'M', 'N'], "asphalt")
    if tag == 'M':
        return extrude_face(obj, 2, [1.0, 1.5], 'N')
```

Code used for defining the rules of house 3

```
# House 3
if layer == 1:
    if tag == 'L':
        return extrude_face(obj, 4, [2.4, 2.8], 'B')
    if tag == 'B':
        add_prefab(obj, "fence", 1, 0, 'N', "bricks")
        add_prefab(obj, "fence", 1, 1, 'N', "bricks")
        add_prefab(obj, "fence", 1, 3, 'N', "bricks")
        add_prefab(obj, "roof4", 1, 4, 'N', "asphalt")
        rng = random.uniform(0.0, 1.0)
        return subdivide(obj, 0, [['R', 2], ['R', 1]], ['RIGHT', 'LEFT'], "quartz")
    if tag == 'LEFT':
        return subdivide(obj, 2, [['R', 1.5], ['R', 1]], ['G', 'W3'], "quartz")
    if tag == 'RIGHT':
        add_prefab(obj, "roof2", 0.9, 4, 'N', "asphalt")
        return subdivide(obj, 0, [['R', 2], ['R', 1]], ['R2', 'M'], "quartz")
    if tag == 'R2':
        return subdivide(obj, 2, [['R', 1.5], ['R', 1]], ['LS', 'W3'], "quartz")
    if tag == 'M':
        return subdivide(obj, 2, [['R', 1], ['R', 0.2], ['R', 1]], ['D', 'EF', 'W'], "asphalt")
    if tag == 'EF':
        return extrude_face(obj, 2, [0.5, 1.0], 'N')
    if tag == 'G':
        return add_prefab(obj, "garage", 0.5, 2, 'N', "metal")
    if tag == 'D':
        return add_prefab(obj, "door", 1, 2, 'N', "concrete")
    if tag == 'W':
        return add_prefab(obj, "window5", 0.5, 2, 'N', "asphalt")
    if tag == 'W2':
        return add_prefab(obj, "window6", 0.5, 2, 'N', "asphalt")
    if tag == 'W3':
        rng = random.uniform(0.0, 1.0)
        if rng < 0.5:
            return subdivide(obj, 0, [['R', 1], ['R', 4], ['R', 1]], ['N', 'W', 'N'], "quartz")
        else:
            return subdivide(obj, 0, [['R', 1], ['R', 4], ['R', 1]], ['N', 'W2', 'N'], "quartz")
    if tag == 'LS':
        return subdivide(obj, 0, [['R', 1], ['R', 4], ['R', 2]], ['N', 'W', 'N'], "quartz")

if layer == 2:
    if tag == 'LS':
        add_prefab(obj, "lshape", 0.8, 2, 'W3', "stone")
```

Code used for defining the rules of house 4

```
# House 4
if layer == 1:
    if tag == 'L':
        return extrude_face(obj, 4, [1.5], 'B')
    if tag == 'B':
        extrude_face(obj, 0, [1.5, 2.0], 'N')
        return extrude_face(obj, 1, [1.5, 2.0], 'BB')
    if tag == 'BB':
        return subdivide(obj, 0, [['R', 1], [1.0], ['R', 1], [1.0, 1.5], ['R', 1]], ['S', 'M', 'P', 'M', 'S'], "quartz")
    if tag == 'S':
        return extrude_face(obj, 4, [2.0, 2.5], '2M')
    if tag == 'P':
        return extrude_face(obj, 4, [1.2, 1.8], '2M')
    if tag == '2M':
        return subdivide(obj, 2, [[1.5], ['R', 1]], ['F1', 'F2'], "marble")
    if tag == 'M':
        add_prefab(obj, "roof3", 0.8, 4, 'N', "blackstone")
        add_prefab(obj, "door", 0.5, 3, 'N', "wood")
        return add_prefab(obj, "door", 0.5, 2, 'N', "wood")
    if tag == 'F1':
        add_prefab(obj, "window5", 0.5, 3, 'N', "wood")
        rng = random.uniform(0.0, 1.0)
        if rng < 0.5:
            return add_prefab(obj, "garage", 1, 2, 'N', "concrete")
        else:
            return add_prefab(obj, "window5", 0.5, 2, 'N', "wood")
    if tag == 'F2':
        add_prefab(obj, "window2", 0.5, 3, 'N', "wood")
        return add_prefab(obj, "window2", 0.5, 2, 'N', "wood")
    if tag == 'W':
        add_prefab(obj, "window6", 0.1, 0, 'N', "blackstone")
        return add_prefab(obj, "window6", 0.1, 1, 'N', "blackstone")
```

```
if layer == 2:
    if tag == 'BB':
        return subdivide(obj, 2, [['R', 1], [0.15, 0.2]], ['CB', 'CO'], "quartz")
    if tag == 'CB':
        rng = random.uniform(0.0, 1.0)
        if rng < 0.5:
            return subdivide(obj, 0, [[0.15, 0.2], ['R', 1], [0.15, 0.2]], ['CO', 'N', 'CO'], "blackstone")
        else:
            return subdivide(obj, 0, [[0.15, 0.2], ['R', 1]], ['CO', 'N'], "blackstone")
    if tag == 'CO':
        extrude_face(obj, 3, [0.5, 0.6], 'N')
        return extrude_face(obj, 2, [1.5, 1.6], 'N')
    if tag == 'F2':
        return subdivide(obj, 2, [['R', 1], [0.15, 0.2]], ['CB', 'CO'], "quartz")

if layer == 3:
    if tag == 'F2':
        rng = random.uniform(0.0, 1.0)
        if rng < 0.5:
            return extrude_face(obj, 0, [0.1, 0.3], 'W')
        else:
            return extrude_face(obj, 1, [0.1, 0.3], 'W')
```

Bibliography

[Ble] Blender stackexchange. “Extrude in python”. URL: https://blender.stackexchange.com/questions/115397/extrude-in-python/115417 (accessed: 13/05/2021)

[Bri] The Editors of Encyclopaedia Britannica. “Fractal”. URL: https://www.britannica.com/science/fractal (accessed: 22/03/2021)

[Che et al. 08] Chen, G., Esch, G., Wonka, P., Müller, P., & Zhang, E. "Interactive procedural street modeling." ACM SIGGRAPH 2008 papers. 2008. 1-10.

[CSN20] Coelho, António, Augusto Sousa, and Fernando Nunes Ferreira. "Procedural Modeling for Cultural Heritage." Visual Computing for Cultural Heritage. Springer, Cham, 2020. 63-81.

[Eli] Elistratov, Vladimir (vvoovv on github.com). “BCGA”. URL: https://github.com/vvoovv/bcga (accessed: 27/04/2021)

[Esr] Esri. “ArcGIS CityEngine”, URL: https://www.esri.com/en-us/arcgis/products/arcgis-cityengine/overview (accessed: 23/03/2021)

[Fil94] Filip, Norbert. "Fractal based procedural modelling." (1994).

[HM10] Haala, Norbert, and Martin Kada. "An update on automatic 3D building reconstruction." ISPRS Journal of Photogrammetry and Remote Sensing 65.6 (2010): 570-580.

[IG19] Ignacio Martín; Gustavo Patow "Ruleset-rewriting for procedural modeling of buildings." Computers and Graphics 84 (2019): 93-102.

[JCS16] Jesus, D., Coelho, A. & Sousa, A.A. “Layered shape grammars for procedural modelling of buildings.” Vis Comput 32, 933–943 (2016).

[LWW08] Lipp, Markus, Peter Wonka, and Michael Wimmer. "Interactive visual editing of grammars for procedural architecture." ACM SIGGRAPH 2008 papers. 2008. 1-10.

[MSK10] Merrell, Paul, Eric Schkufza, and Vladlen Koltun. "Computer-generated residential building layouts." ACM SIGGRAPH Asia 2010 papers. 2010. 1-12.

[Mül et al. 06] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. 2006. “Procedural modeling of buildings.” ACM Trans. Graph. 25, 3 (July 2006), 614–623.

[MZWV07] Müller, P., Zeng, G., Wonka, P., & Van Gool, L. "Image-based procedural modeling of facades." ACM Trans. Graph. 26.3 (2007): 85.

[PHHM96] Prusinkiewicz, P., Hammel, M., Hanan, J., & Mech, R. "L-systems: from the theory to visual models of plants." Proceedings of the 2nd CSIRO Symposium on Computational Challenges in Life Sciences. Vol. 3. Citeseer, 1996.

[Pro] Prochitecture. “blender-osm (premium)”. URL: https://gumroad.com/l/blosm (accessed: 20/05/2021)

[Prz99] Przemyslaw Prusinkiewicz. “A look at the visual modeling of plants using L-systems.” Agronomie, EDP Sciences, 1999, 19 (3-4), pp.211-224.

[SM15] Schwarz, M., Müller, P. 2015. “Advanced procedural modeling of architecture.” ACM Transactions on Graphics, 34, 4 (Proceedings of SIGGRAPH 2015), Article 107

[STBB14] Smelik, R. M., Tutenel, T., Bidarra, R., & Benes, B. "A survey on procedural modelling for virtual worlds." Computer Graphics Forum. Vol. 33. No. 6. 2014.

[Wil et al. 21] Willis, A. R., Ganesh, P., Volle, K., Zhang, J., & Brink, K. "Volumetric procedural models for shape representation." Graphics and Visual Computing 4 (2021): 200018.