Bachelor Thesis

Procedural Aging Of Buildings

Christos Othonos

University of Cyprus

Department of Computer Science

May 2017

Bachelor Thesis

Procedural Aging Of Buildings

Christos Othonos

University of Cyprus

Department of Computer Science

May 2017

University of Cyprus

Department of Computer Science

Procedural Aging Of Buildings

Christos Othonos

Advisor

Professor Yiorgos Chrysanthou

Diploma project has been submitted for partial fulfillment of the requierements of Informatics Degree acquisition from the University of Cyprus

May 2017

Acknowledgement

I would like to express my sincere gratitude to a few people that have helped me with the completion of this project. Firstly I would like to thank my advisor Professor Yiorgo Chrysanthou, for his guidance and support throughout the completion of this project, his patience, motivation, and immense knowledge. I would also like to thank Professor Gustavo Patow from the University of Girona, for providing us with his valuable expertise in procedural building algorithms and aging. Finally I would like to thank Gregorie Nieto for letting us study and learn from his work on the same subject.

Abstract

Due to the increasing need for more graphical content, many professionals resort to incorporating procedural generation tools to their work-flows. One such category of graphical content is building generation and already there are specialized tools that make use of a set of rules and minimal user input to generate an astounding amount of buildings. Such technology has been put into great use, providing city models for open world games and movies. We argue however, that the procedural building generators have limited capacity for realism due to buildings always being in pristine condition. To increase realism one must add layers of degradation to make it look like it was affected by real world phenomena. This manual work is feasible for small projects, but not for an entire city scape. We propose a system that can be integrated in any procedural building generator and can add degradation details to buildings automatically as they are constructed. Our contributions include a proof of concept pipeline extending the popular CGA Shape Grammar, the storing of aging and degradation related attributes on the building using a grid model, several simulation techniques to detect areas affected by humidity and weakness of structure, and several techniques that visualize degradation using the aforementioned attributes.

Contents

List of Figures

Chapter 1

Introduction

Contents

1.1 Motivation

Computer graphics have revolutionized the way we work and entertain ourselves. The applications of computer graphics are endless and varied, ranging from medical visualizations to movie productions. In addition, with the constant evolution of graphics hardware we see a steady improvement in realistic real-time 3d rendering and image processing.

Some of the most impressive uses of computer graphics is in the entertainment industry. Movie studios make extensive use of CGI graphics, to cut down the costs of practical effects and rentals. Things like helicopters, creatures, vegetation or even entire cities are often drawn by 3d modelers and rendered at a very high fidelity. If done correctly, these graphics look so realistic that it is hard to discern them from their real counterparts. Video games by default make use of computer graphics, because of their need to generate a picture on the fly based on a dynamic game state. To immerse the player, game studios usually focus on a photorealistic aesthetic and invest a big part of the budget on good looking 3d models.

Since most movie productions and electronic games have a high reliance on 3d graphics, it is imperative to have the manpower to generate all of the 3d content. For that reason triple-A studios with big budgets hire dozens of artists to model what is needed. However with the ever increasing need for better and more graphics, especially in video games, even big companies find it hard to keep up. At some point, the costs of graphics will be so prohibitive that companies will find ways to cut corners. One such shortcut is procedural content generation.

Procedural generation algorithms are able to generate a specific type of content with a pre-defined set of rules, a typically small user input and randomization. Procedural content can vary from text, to texture, to audio or even 3d models. Even as back as the early days of computers such algorithms existed, but it is only in recent years that they saw a rise in popularity.

Typically procedural generation offers two great advantages. Firstly it allows for the effective compression of extremely large data sets. This is because stochastic procedural methods can generate potentially infinite amounts of the same type of content, with only a few rules and minimal user input. This feature was exploited a few years back by computer games such as Elder Scrolls:Daggerfall [29], where a large map the size of Great Britain was compressed to fit in a single CD-ROM. More recent examples include the game “Minecraft” being able to generate entire worlds from a single seed value. Another great advantage is that it helps with the creation of large amounts of content. Esri’s CityEngine [7] is a program that allows for the creation of entire cities using a floor plan and a set of procedural rules.

Ofcourse procedural algorithms are not without their shortcomings. A common complaint by artists is that procedural generating solutions, with the exception of programs like Houdini FX [28], severely limit artistic freedom. One reason is the fact that most procedural methods make use of random elements to add variety; this in turn makes it difficult for the artist to predict or even control the output of such generators. Artistic direction is a very important resource in the creation of media, and it goes without saying that severely limiting the control of the artist is a waste of ‘creative’ resources.

The second pitfall of procedural methods is the lack of creativity in the generated content. Due to the algorithmic nature of these systems; even with the inclusion of stochastic elements, the output will most likely look the same to the average observer. To better understand this problem imagine a procedural generator that creates a set of clothes for a virtual character to wear. It works by simply changing the colour of existing clothing items to a random colour and then dressing the avatar. This system would be able to generate a seemingly endless amount of varied content; but the human observer would quickly notice a pattern in all of the combinations since all of the clothes have the same shape and style as the previous avatar. So a procedural generator such as the one in this example fails in adding variety. An improved version of the generator would simply include more clothing items and randomly switch between them. This would certainly create a greater variety, but this still requires an artist to draw all of the base clothing items. The amount of work required by artists for the creation of sufficient procedural generators is debatable, but that doesn't change the fact that the generator does not innovate by itself; it simply follows a set of discrete rules.

It becomes clear that the strength of procedural systems does not lie in creativity but to facilitate large and monotonous tasks. In that regard artists can make use of such systems to get rid of the tedious bulk of content, and then add their own creative touches afterwards. One notable example is the dialogue systems of games such as "The Witcher 3" [30], where all of the dialogue cinematics were procedurally generated by algorithms, and then refined by hand. Therefore even if procedural algorithms themselves fail to be creative, they are essential in projects with massive amounts of content.

One such type of task is the generation of buildings and entire cities, and specialized software such as Esri's cityengine [7] are put into effect. The fact that the majority of man made structures are comprised of simple shapes and patterns, favors procedural algorithms. Additionally the importance of procedural modeling of buildings and structures, attracted a lot of research effort; and because of that we now have a plethora of tools and techniques to generate buildings.

The entirety of conventional procedural building methods, always create buildings in perfect shape. This is a huge disadvantage for projects that strive for a photo realistic aesthetic. Most people in this world live in cities with old buildings full of imperfections due to long-term weathering; so it is not common for them to see structures in pristine condition. Even in newly built buildings, the most subtle of aging effects will appear. Also humans are very good at pattern matching tasks, and can easily pick up minor details or lack thereof. Therefore a city setting with perfect buildings would strike many people as unrealistic or even unsettling. Especially outside the western world quite a lot of people live or work in buildings made of earth or other natural materials that degrade quickly if not maintained.

Manually adding weathering details to a building is feasible, especially with the evolution of 3d modeling software capabilities. Sculpting tools can be used to add dents, bumps and cracks on a surface. Textures can be used to add moss, discolorations and layer degradation. Additionally an artist can modify the entire structure of the building, collapse roofs, break windows etc. Not only can an artist have full control of the process, but also use intuition to create an aesthetically pleasing output. Unfortunately this is not viable for large tasks, such as with an entire city.

Alternatively a simulation based method may be used. Simulation methods take into consideration various environmental and internal factors, and approximate reality in a more algorithmic way. Depending on the granularity and resolution of the simulation, varying amounts of computing power and computation time will be required. Seeing as how animation studios have no problem in running a rendering farm for hours to get the perfect picture, a slow physics simulation is not out of the question. What is a concern however is the lack of simulation tools that specialize in building degradation.

When mentioning the words physics and building degradation, physics engines might come to mind. State of the art physics engines such as NVidia's PhysX [22] and Havok [10] are real time and have very convincing results. One could procedurally create an entire building out of different components, and then use a physics engine to simulate a sequence of events that will collapse and degrade the structure. Physics engines among other things can simulate the structural collapse of a building and the 'peeling' of layers such as plaster. Again since it is the results that we are interested in, the cost of the simulation is not our major concern. However, no matter how appealing this method is, we believe that it is very inefficient for this task. The focus of a physics engine is to simulate real-time events, so that every independent body or particle moves in a believable way. In other words physics engines are focused on real-time and short-term simulations. Our goal however is to perform a long-term simulation, where each simulation step can be as much as an entire month. There is no need to know how each wall fragment moves every second; only where it has ended up.

1.2 Objective

Our objective is to develop a system that procedurally adds aging imperfections to buildings. Such a system would help professional artists make their workflow more efficient when constructing ruined buildings. This would also help companies in the entertainment industry cut costs when tackling this problem, so that they can focus their resources on other areas. This system must have as much automation as possible to avoid unessesary manual refinements.

1.3 Contributions

We propose an extension to the popular CGA shape grammar pipeline, that allows for the simulation and application of long-term degradation effects on buildings. We implemented a proof of concept on the Houdini software by Side FX, which requires a slight modification of CGA shape grammar operations and the addition of a “simulation component” and an “aging application component”. As this is one of the first projects to tackle procedural building and degradation, we have placed a number of restrictions to keep the scope at a reasonable minimum. First of all we will not be focusing on the accuracy of the simulation as compared to the real world. Instead we want to demonstrate that our ideas and techniques are flexible enough for any simulation algorithm to work. Secondly we will not be concerned with specific cases of materials and building structures. We believe that by tweaking the attributes, algorithms and techniques that we present; the believable degradation of most materials can be achieved.

1.4 Thesis Structure

In chapter 2 we will present related previous work that this thesis is based on. In chapter 3 we will mention a few things regarding the degradation and aging of buildings in the real world. Next we will explain our Simulation pipeline in chapter 4. Then we will go into more depth in the Implementation specifics of our work in chapter 5, followed by chapter 6 in which we show some simulation results in various conditions. Finally we will give some concluding remarks and mention what we plan to improve in the future in chapter 7.

Chapter 2

Previous Work

Contents

2.1 Procedural Algorithms

Procedural algorithms are a set of rules and steps, that result in the creation of a specific type of content. There are currently many procedural generators and they can be used generate any type of content: from audio, to textures and 3d models. Procedural generation as we have mentioned in the previous chapter, have the ability to compress large data sets, as well as facilitating monotonous content creating workloads.

Music can be generated procedurally. Even as back as 1963, a software known as Musicomp was used to compose musical scores such as the "Computer Cantata" [11]. Additionally audio effects can also be generated by a computer. This is useful in interactive forms of media such as video games, where pre-recorded sound effects cannot match all possible interactions [8].

Texture synthesis involves the generation of textures with various methods. The generated textures can have various properties, but should be able to fit together seamlessly and be varied enough. Substance [3] is a notable example of a software package that synthesizes infinite and seamless textures. Instead however of the typical 2D texture, the target program will use procedural rules to re-create the texture where it is needed.

Procedural modeling is a term referring to the semi-automatic generation of 3d models. These models can represent anything visible in the virtual world, that is typically made of polygons. One type of model that can be procedurally generated is terrain: a model that is used to express features of the earth such as hills, valleys and mountains. Artists can use traditional modeling tools to sculpt such models, but a more cost-effective solution is rather to use height maps. Heightmaps not only offer a cheap way to store terrain models as 2D textures, but can easily be generated procedurally using texture generation methods [25]. Terrains can also be represented by voxel maps, which solve the cave and overhang problem of heightmaps [26] but may leave a huge memory footprint for large models. For that reason voxel terrains are generated on the fly using 3d fractals and procedural rulesets. One such example is the successful computer game Minecraft [19].

Although nature is important to replicate in the graphics industry, man-made structures are equally as important. Most of these structures follow predictable and simple patterns, which makes them easier to work with by artists. Unfortunately most projects require graphics of massive city landscapes or other manmade structures, that are tedious to draw by hand. Because of the aforementioned reason, the generation of buildings and other manmade structures, has made this category of procedural generation gain a lot of research interest.

An interesting structure to procedurally construct are roads. Although this is a trivial problem in cases where the city layout is provided, roads outside of major cities can be quite complex. Not only that but road construction takes into account the cost of building roads a certain way; factors such as the inclination of slopes and natural obstructions may change the trajectory and shape of the road. This can be solved by using a cost function to design the trajectories of the road, and then procedurally construct structures such as roads, tunnels and bridges [9].

The procedural generation of buildings is a topic that received a great deal of research effort. An L-system based procedural building algorithm was developed by Parish et al. Given a city layout with population density and water boundaries, their proposed system can generate road layouts, buildings, and parking lots [23]. Although their method was sufficient for generating large city scapes, the generated buildings had simplistic mass models and their facade relied on textures and shaders to depict features such windows and doors. A few years later P. Wonka et Al. present a method that uses split grammars instead of L-systems to generate buildings. Their focus however was on speeding up the procedural algorithms of urban building generation [31]. The P. Wonka et al. method produced high quality buildings, but only worked well with simple mass models with axis aligned orientations. The shortcomings of both algorithms were addressed with the CGA shape grammar. The CGA shape grammar uses ideas from [23], as well as the split grammars from [31] and extends upon them. As of now this method is one of the most popular in the procedural generation of cities and towns [20].

The majority of procedural building generation algorithms, including the ones mentioned in the previous paragraph, suffer from a few problems. Firstly they only generate building shells and completely neglect the interiors. This can be a problem for open world sandbox games, that are advertised for the amount of freedom they give to the player. Game developers can create a work-arounds such as the adding transitions that teleport the avatar to the room, but such methods might be undesired for professional game studios. Alternatively it is possible to generate interiors procedurally [4]. The other problem procedural generation algorithms may face is the lack of building functionality. This is a given since most procedural algorithms do not start with customer specifications, the same way an architect would start designing a building. Both of these problems can be addressed, but for the scope of this project we will not consider solving them.

2.2 CGA Shape Grammar

CGA shape (Computer Generated Architecture) is a shape grammar that efficiently generates high quality building shells. The authors of the paper have build a virtual model of Pompeii to demonstrate the capability of their grammar [20]. The Esri Cityengine is a specialized software which uses a rule-based modeling system based on CGA shape to generate cities [7]. Due to the flexibility and efficiency of this procedural algorithm we have decided to use it as a base to present our aging simulation pipeline.

2.2.1 Notation

In this subsection we will present various terms and notation associated with the original CGA shape grammar. It is important to do so, because we will need these terms to describe both the how CGA shape works, and then how our system works.

Shapes

According to the original paper, a shape is a collection containing a symbol, a geometry and numeric attributes. Symbols are integral in any grammar, being either a terminal symbol or a non-terminal symbol. In the case of CGA Shape grammar, every shape is represented and identified by its symbol. The geometry is enclosed in a bounding box referred to as a scope. For example we may have a bounding box like in Figure 2.1 that contains a cylinder mesh.

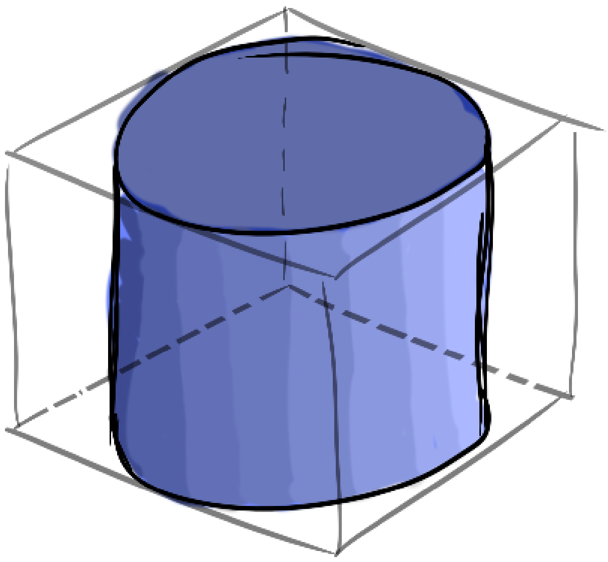


Figure 2.1: Example of a scope.

Facades

Facade is a word from the French language meaning “face” of a building (see Figure 2.2). In procedural building jargon facades have the exact same meaning. In CGA Shape facades are initially the geometric faces of the building geometry. Note that in CGA Shape Grammar facades are not considered shapes, but they can be represented by a symbol.

Production Rules

Like any formal grammar, CGA shape uses a set of production rules. The symbols in these rules represent a shape. The form of the rules is shown in Equation 2.1. On the left hand side we have a numerical id for the rule, and the predecessor symbol. Once a condition is met, the successor shape is generated in place of the predecessor. The value of prob is simply a probability to

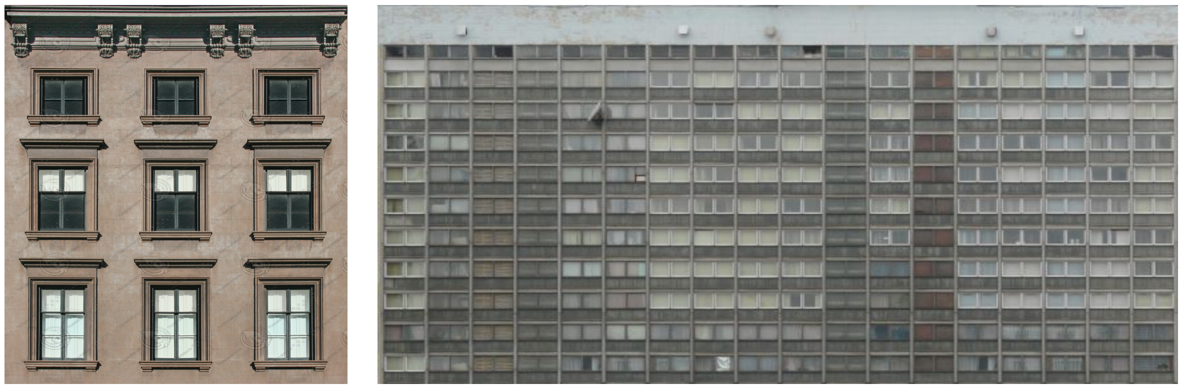


Figure 2.2: Examples of building facades. Left image from [5] and right image from [6]

add stochastic behavior and randomness in the building process. For example we may specify multiple types of buildings that can be placed in an empty place, along with a probabillity for their appearance. This way we can add variety in an otherwise deterministic system.

$$id : predecessor : cond \to successor : prob \tag{2.1}$$

Scope Rules

Scope rules are operations that modify shapes. For example we can translate, rotate or scale a shape. We can also add a geometric shape that fits inside the current scope. Additionally there is an operation called Roof that is used along with the more conventional scope rules. The Roof operation adds a roof on top of a shape, given a preset roof type and roof inclination.

Split Rules

An important part of CGA shape is its ability to split shapes and surfaces to refine the geometry. The split rules are more or less identical to the ones presented in the book ”Instant Architecture” [31].

First we have the basic split rule. We can split our shape or facade in one of three axes (or a combination of them) and in various locations. An example of such a rule is given in the original paper and in Figure 2.2. In this example we split a facade in the Y axis into 5 sections. We determine the size of each segment as well as the output symbol (in the curly brackets). We can also split a 3d scope, if we want for example to split our building into multiple stories. The results of the production rule can be seen in Figure 2.3.

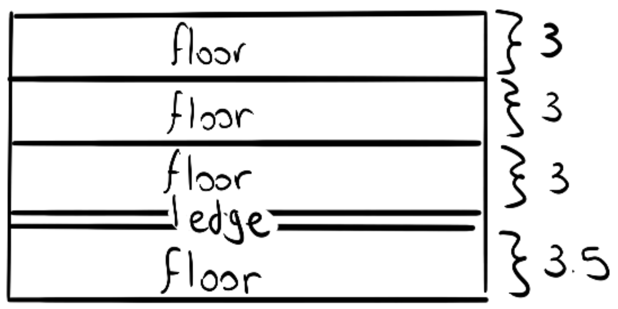


Figure 2.3: Subdivision of a building into multiple floors.

$$1 : fac \rightarrow Subdiv(\text{"}Y\text{"}, 3.5, 0.3, 3, 3, 3)\{floor|ledge|floor|floor|floor\} \qquad (2.2)$$

Using the exact size to determine the segment may not always work in our favour. For example, if we applied the same split rule on a building with a different height we will most likely get unexpected results. For that reasons “scaling” is used. Using the letter r along with a multiplier we can be sure that a segment’s size is relative to the size of the shape. We can see this from another example given in the original paper, shown here in Equation 2.3. Just like before we take a predecessor symbol (“floor” in this case) and subdivide it into 4 sections. Note that the two middle sections have relative sizing as opposed to sections A. No matter what the size of the floor is we will get two sections of size 2, and two middle sections that take the rest of the length. Figure 2.4 illustrates the use the production rule making use of scaling, and comparing its use on two different building sizes.

Another split rule is “Repeat”, which performs a predetermined split repeatedly until the shape is exhausted. The example from the ”Procedural Modeling Of Buildings” paper [20] (and Figure 2.5) splits the floor along the X-axis into N subsections of size 2. This is especially useful when we want to separate an arbitrarily long surface into equally sized chunks.

Finally we have the Component split which splits a shape into shapes of lesser dimensions. Using this split we can extract the facades from the 3d shape of the building.

Occlusion

One of the problems presented in the CGA shape paper was the danger of placing windows and doors in areas that are partially or completely occluded. To solve this an occlusion checking rule

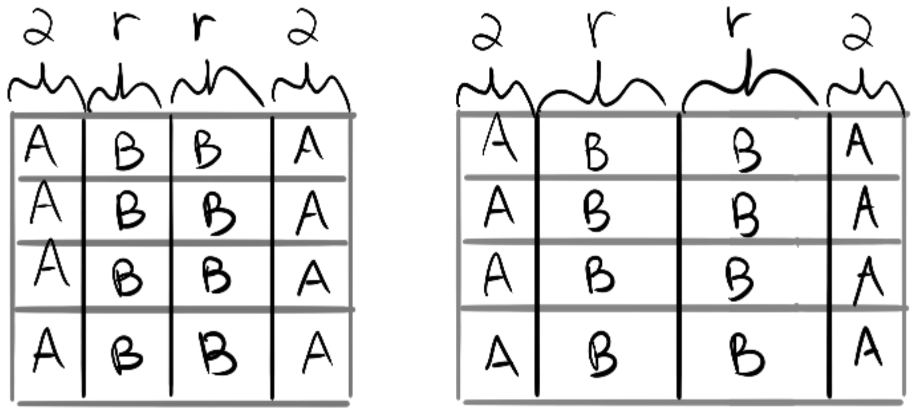


Figure 2.4: Using scaling. Regardless of the size of the building, sections with relative sizing take up the remaining space. Note that the example buildings are a continuation of 2.3

$$2 : floor \rightarrow Subdiv(”X”, 2, 1r, 1r, 2)\{A|B|B|A\} \tag{2.3}$$

is used such as the example in Equation 2.5. In the example we make sure that before placing the shape door over the shape tile, that there is no other shape that occludes it. The same rule can also check against a specific group of shapes instead of “all” and it can check for partial or full occlusion instead of “none”

$$1 : tile : Shape.occ(”all”) == ”none” \rightarrow door \tag{2.5}$$

This operation is very important when procedurally creating new buildings (e.x by using probability rules) because the occlusions between the shapes can vary significantly. However we have decided against implementing this functionality as it was not necessary to demonstrate our pipeline.

2.2.2 Pipeline

We will now present the workflow of a typical CGA Shape pipeline. Some inferences had to be made since some details were not explicitly specified. The process starts with the Mass Modeling stage, assembling solids or extruding footprints to create the basic shape of the building. Then we split the building into lesser dimensions in the Component Splitting stage. For the

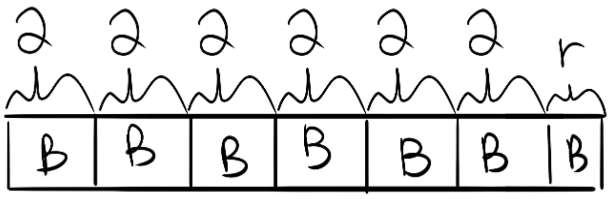


Figure 2.5: Using the repeat operation to cut the shape into equally sized chunks. In cases where the length of the shape isn't divisible by the given size, the last chunk will simply take up the remaining space.

$$2 : floor \rightarrow Repeat(\text{"}X\text{"}, 2)\{B\} \tag{2.4}$$

sake of simplicity we only consider facades as a lower dimension primitive. Next we use splits to form the seams for the placement of detailed models in Facade Operations. Finally we place detail geometry in the Apply Geometry stage. A flow diagram of the pipeline is show in Figure 2.6.

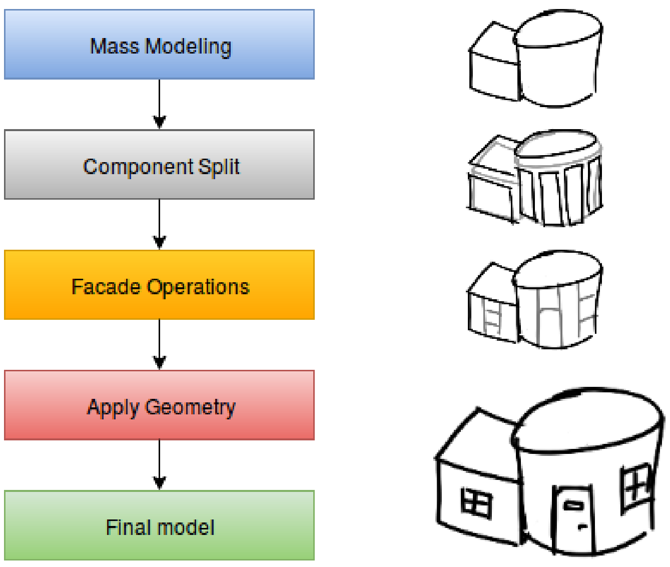


Figure 2.6: The CGA Shape grammar pipeline.

Mass Modeling

The Mass Modeling stage involves the assembling of solids to form the basic shape of the building. This can be achieved with a combination of scope operations, split rules and inserting shapes in scopes. Alternatively we can follow the Cityengine approach and extrude the building shape from a building floor plan, followed by regular production rules [7]. Figure 2.7 shows two examples of mass models.

Another feature of the base shape is the roof. The original paper of CGA Shape grammar suggests placing basic roof shapes that match the base solids. Depending on the shape of the base model, we can perform a simple extrusion with an inset and collapse several edges to get a basic roof shape. If our building was extruded from an arbitrary footprint we can use a roof generating algorithm such as [15]. Figure 2.8 shows an example of a generated roof model.

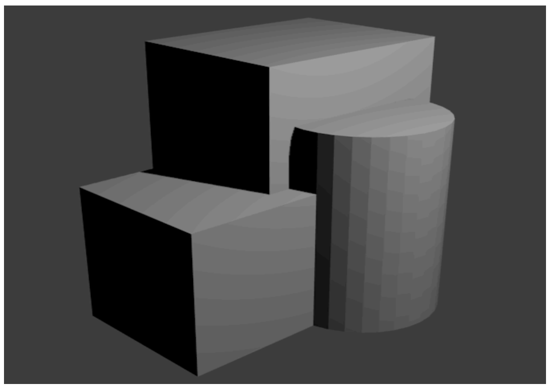
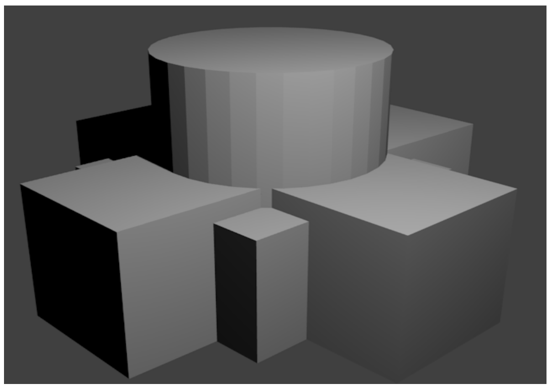



Figure 2.7: Examples of combinations possible with mass modeling.

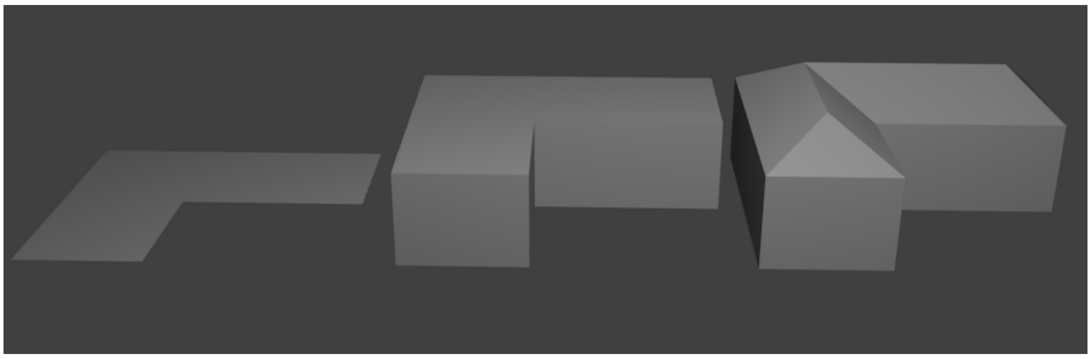


Figure 2.8: Extruding a footprint and generating a roof from the shape.

Facade Operations

In this stage of the pipeline we work on a per-facade basis. Using facade symbols and the appropriate generation rules we can add details such as windows and doors. What to keep in mind here is that we are working on two dimensional faces and not 3d shapes. For that reason all the split operations can be oriented based on the orientation of the facade, thus requiring only two splitting axes. This is inferred from the claims that the CGA shape grammar can work on shapes of arbitrary orientation including sloped roof surfaces.

Another thing that we should point out is that our method depends on the fact that no 3D operations (or scope operations) are performed in Facade Operations. We work under the assumption that upon the completion of the Mass Modeling stage, the algorithm has created the best possible coarse representation of the final model. So the purpose of this stage of the pipeline is to refine the shape of the model while keeping the basic shape the same.

2.2.3 Example

Here we will show a typical example using the standard CGA shape pipeline that we have shown in the previous section. We will begin with the mass modeling stage.

Assembling Solids

We assume that initially we have a unit sized scope with a box shape that we call “footprint”. Then we use a rule that takes the “footprint” shape as a predecessor and scales it 2 times in the Y-axis. To add more components we can use the “footprint” again as many times as we want as it was not deleted with the previous rule. In this example we need another box and a cylinder. We use another rule which scales the “footprint” shape 3 times in the X axis, 1.25 times in the Z-axis and then moves it 2 units in the Y-axis and 1.25 in the Z-axis (upwards). Finally we will add the scope for the cylinder, by using the “footprint” shape once again, scaling by 2 units in all directions, and then moving it 2.3 units in the Y-axis and 2 units in the Z-axis. Additionally we use the insert scope rule to place a cylinder in place of the scope. We will give all building parts the same symbol name “building” for the sake of simplicity. See figure 2.9 for a visual representation.

Then we separate the entire building into three stories using a basic split in the Y-axis. Note that the different colors for each subsection in figure 2.10 is only for demonstration purposes.

Finally we add roofs to each building using the Roof operation. The roofs will be given the symbol name “roof” to differentiate them from the other side-facades.

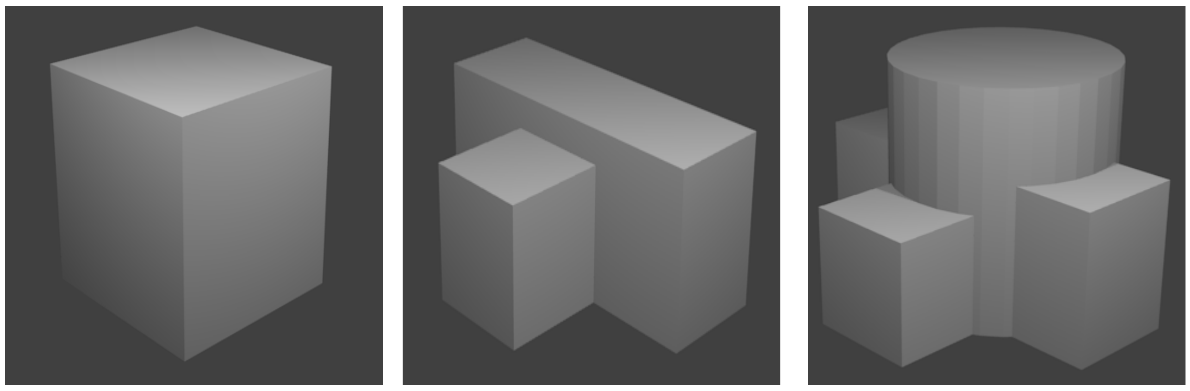


Figure 2.9: Visual representation of assembling the solids in the example.

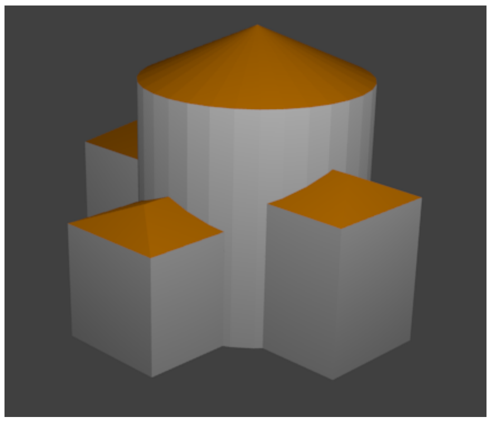


Figure 2.10: Placing roof models on our example. The roof surfaces are tinted orange for contrast.

Component Split

Now that we have our three dimensional model ready it is time to move to a two-dimensional workspace. To do that we need to extract the facades from the base model. We use the Comp operation on the symbol “building” and extract all side-facades which we name with the symbol “wall”. We use Comp also on the roofs to extract the roof surfaces.

Facade Operations

We will finally move to the Facade Operations stage. Remember: we will not touch any of the symbols corresponding to 3d shapes, only facades. First we will select all walls that are on the ground floor and we will subdivide them according to different patterns: 1) Window | Door |

Window 2) Window | Window | Window 3) Wall | Window | Wall. We select each pattern with an equal probability to add some variation. For the other floors we simply place patterns 2 and 3. As for the roof surfaces we will use a Repeat operation twice, once horizontally and once vertically to form tiles. Using the occlusion check we place windows and door models on the appropriate primitives. Finally we replace every tile of the roof surfaces with a roof-tile model. Thus we have finished our simple example. See figure 2.11 for a visual representation.

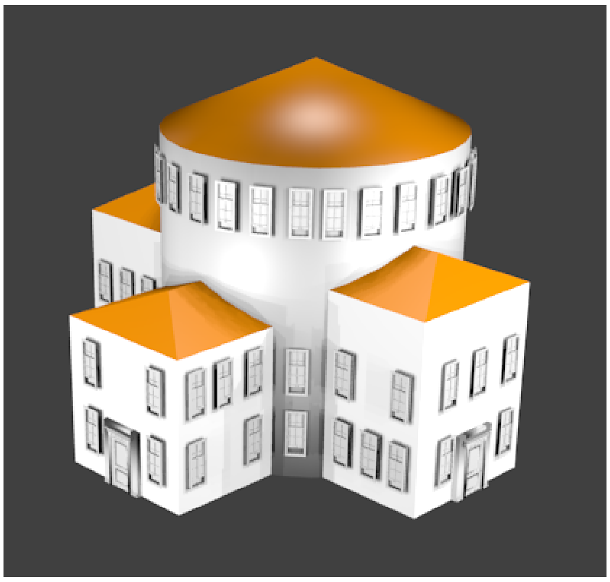


Figure 2.11: The finished procedural model from the example.

2.3 Aging

A survey presented in 2008 by Mrillou and Ghazanfarpour [18] presents numerous techniques for creating weathering and aging effects. The survey classifies aging phenomena for different materials such as rocks, metals and organic matter. These aging techniques for the materials can be used in the context of building weathering.

One form of aging effect in building which is important is the appearance of wall cracks. Several methods for generating cracks and fractures on models have been developed succesfully by many researchers over the past few decades [12–14].

Another thing we would like to mention is that this thesis is a continuation of Gregoire Nieto's work "Simulation of Aging for Procedural Buildings" [21]. His work sought to solve the issue of aging by introducing procedural aging mechanisms on top of the very popular CGA Shape Grammar. His approach involves using textures that are mapped on the building surfaces to store the aging attributes, and by using texture space algorithms it simulates the effects of aging. The texture would then be applied on the desired surfaces mostly by using the Deepening method which sculpts the surface depending on the amount of degradation. Although his idea of using textures has great advantages, it does have a few shortcomings: Firstly each texel of the aging texture is disassociated with its 3D counterpart. This removes the potential of using simulation algorithms that require a point's spatial relevance to the environment. Fixing this with a texture space approach would require storing position and normal vectors along with the aging values. Another problem concerns the connectivity of the surfaces. Each texture usually corresponds to a single facade, meaning that a connected facade would be mapped to a different texture. This creates issues when trying to use simulation algorithms that require to propagate information from point to point. Such propagation algorithms would have to either stop at the last texel, or use a complicated method to figure out which texel of another texture the current one is connected to. To fix these problems we have decided to store aging related information on the model itself rather than external images.

Chapter 3

Real World Aging

Contents

3.1 Causes of degradation

In the real world buildings suffer from the effects of degradation. Not even recently constructed buildings are immune to chemical and physical changes, and that is why most buildings that people observe in their lifetime will have some flaw.

One of the leading causes of degradation is the accumulation of humidity. As we all know water is fluid and has the ability to flow even in the tiniest of spaces. In some cases water is absorbed by materials making them softer or even causing some chemical reactions that effectively weakening them with time. One such example is when the metal reinforcement of concrete rusts and expands, causing the concrete to crack.

Another destructive effect that humidity is partly responsible for is expansion and shrinking of underground masses. When dirt absorbs water it expands taking more volume. That doesn't mean that the accumulated humidity stays in the same place: Tree roots absorb water, extreme heat causes evaporation and pressures force the water to spread out. This means that if not taken into consideration the soil under a building can expand and contract. If this happens it causes deformities on that building such as the walls tilting inwards our outwards creating noticeable cracks [1]. Figure 3.1 shows a graphical example of this effect.

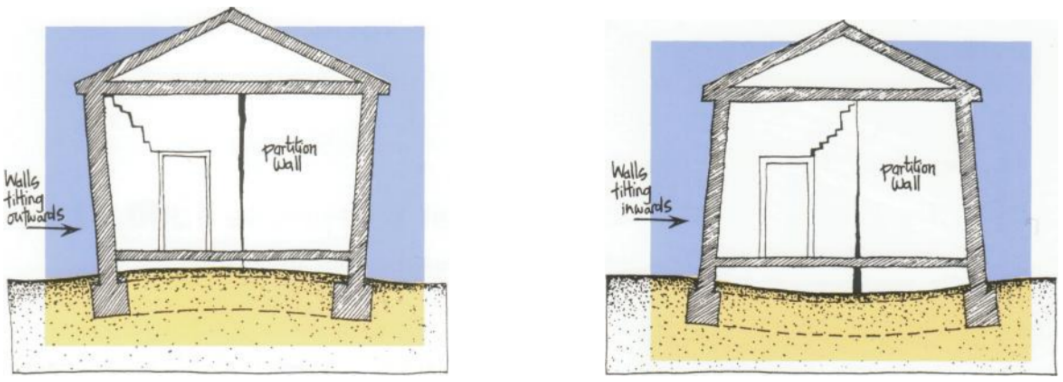


Figure 3.1: The tilting of the walls due to the expansion/contraction of the soil under the house. Source: [1]

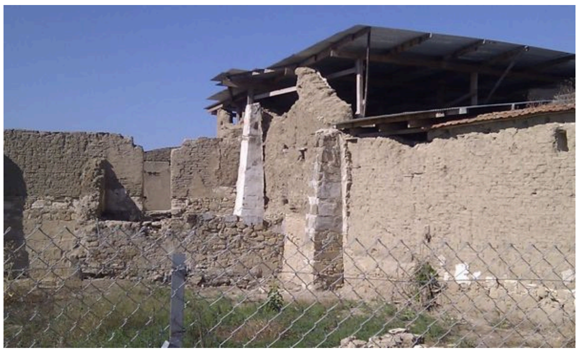
There are of course more extreme causes of building degradation. These range from calamities and destructive forces of nature, to the intentional demolition and destruction of a building by man. The thesis will not focus on such extreme causes of degradation, as a physics engine could prove more productive in this case.

3.2 Natural Building Materials

We decided to focus on the weathering of old houses made of natural materials such as stone and mudbrick. One of the reasons we made this decision is because the structure of these typed of buildings are relatively simple, compared to modern structures. Secondly there are a lot of old abandoned houses and buildings around the world, giving us the oportunity to study patterns of degradation in the real world. In addition, unmaintained mudbrick houses age quickly, so we can observe the changes that occur with time.

Mudbrick is a very old building material that was used as back as 7000 BC in Mehrgarh [24]. The manufacturing method and materials are varied between cultures but it typically involves mixing mud with a binding material such as straw or rice. The mudbricks would then be left to dry out in the sun to harden. When constructing the house, the builders would use mud as a binding agent between the bricks. Figure 3.2 illustrates a mudbrick house. Figure 3.2b show part of a mudbrick wall that was weathered away revealing the straws used as a binding material of the brick.

Stone on the other hand is much more resistant to corrosion than mudbrick. Luckily there are many old structure and buildings made of stone including castles, walls, fountains and roads. Figure 3.2 shows a medieval stone wall in France.


(a) Mudbrick house.

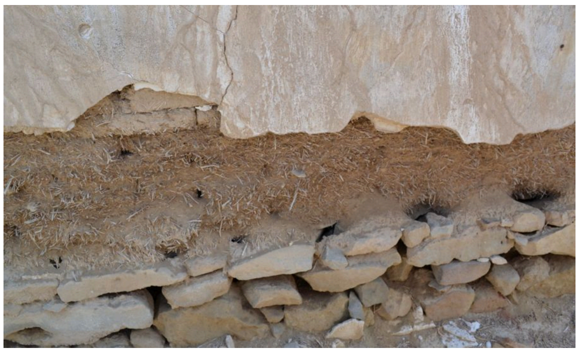

(b) Exposed mudbrick.

Figure 3.2: Examples mudbrick houses.

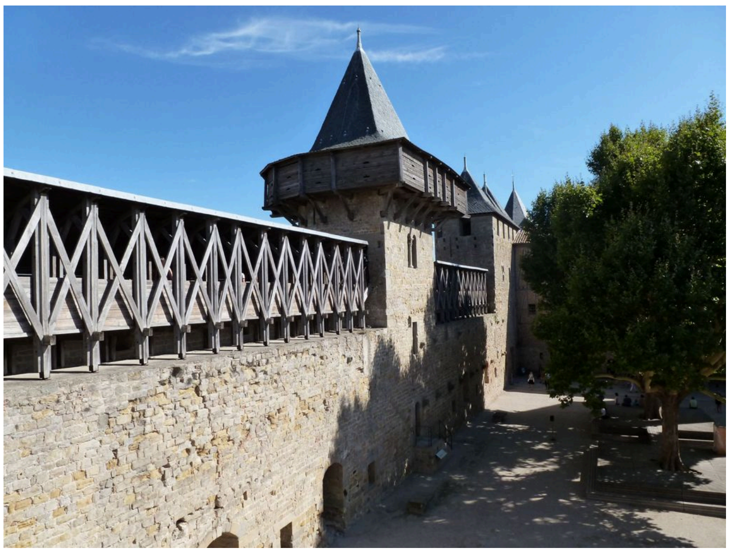


Figure 3.3: Picture of an old stone wall.

3.3 Weathering Effects of Natural Materials

Corrosion

One of the most interesting degradation effects on natural building materials is corrosion. Due to various factors including humidity, water flow, and chemical reactions portions of the material are 'eaten' away. In the case of mudbrick houses rain is the worst culprit for corrosion. In dry areas of the world adobe houses last longer due to the lack of rain and the general lack of humidity. Otherwise houses would need to be painted with a protective plaster layer, and since that paint layer washes away with time, the owner was responsible for re-painting it. Corrosion of mudbrick can also start from the top of the wall and progress downwards. This happens when the protective 'plague' placed between the roof surface and the wall is misplaced allowing for rain to flow on top of the wall. Figure 3.4 shows examples of corrosion weathering effects on mudbrick houses.

We have mentioned previously stone is more resistant to weathering. As opposed to mudbrick water is not the largest factor for stone degradation, chemical reactions are. See Figure 3.5 for such examples.

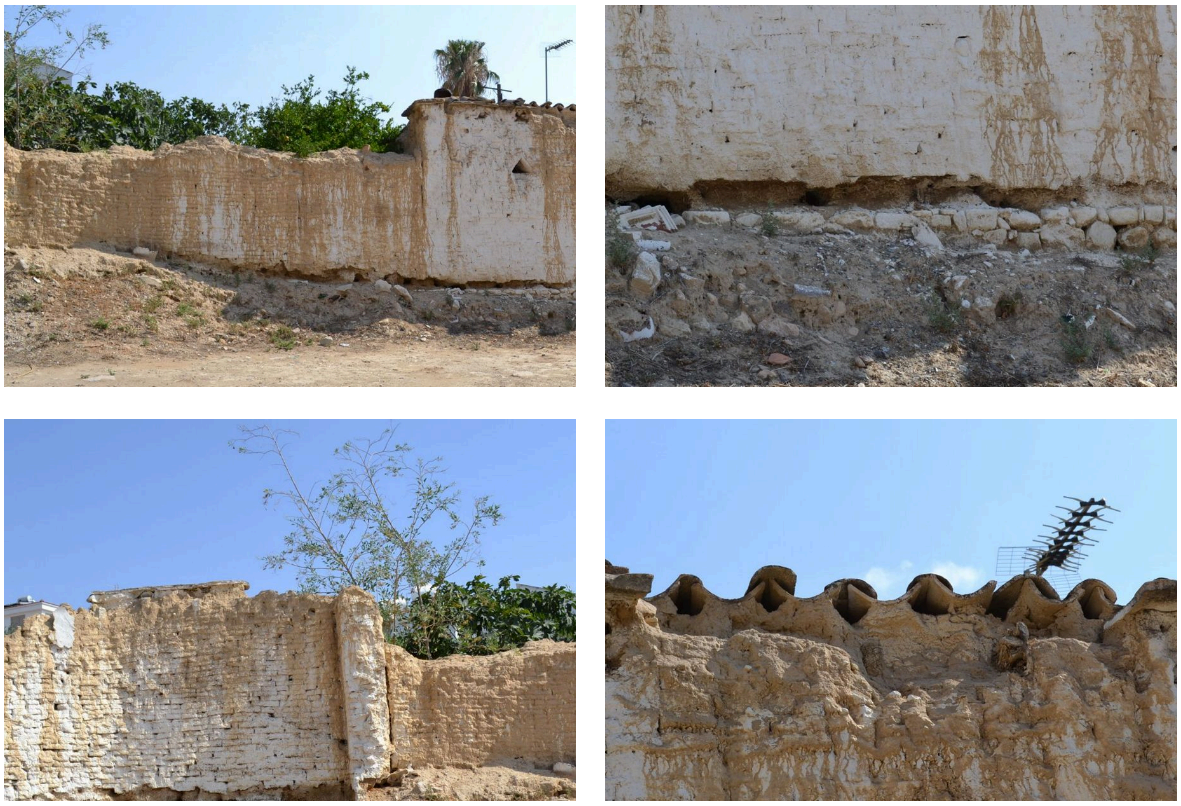


Figure 3.4: Photographs of mudbrick houses subject to corrosion.

Cracks

Cracks form in all rigid materials both in nature and in manmade structures. We often see cracks in concrete and brick walls. This even happens with mudbrick houses. Since the material is rigid, seismic activity and other stresses can tear mudbrick walls into pieces. With stone structures the appearance of cracks is less frequent. This is because each stone is independent of other stones (no binding material is used). Figure 3.6 shows cracks on mudbrick walls.

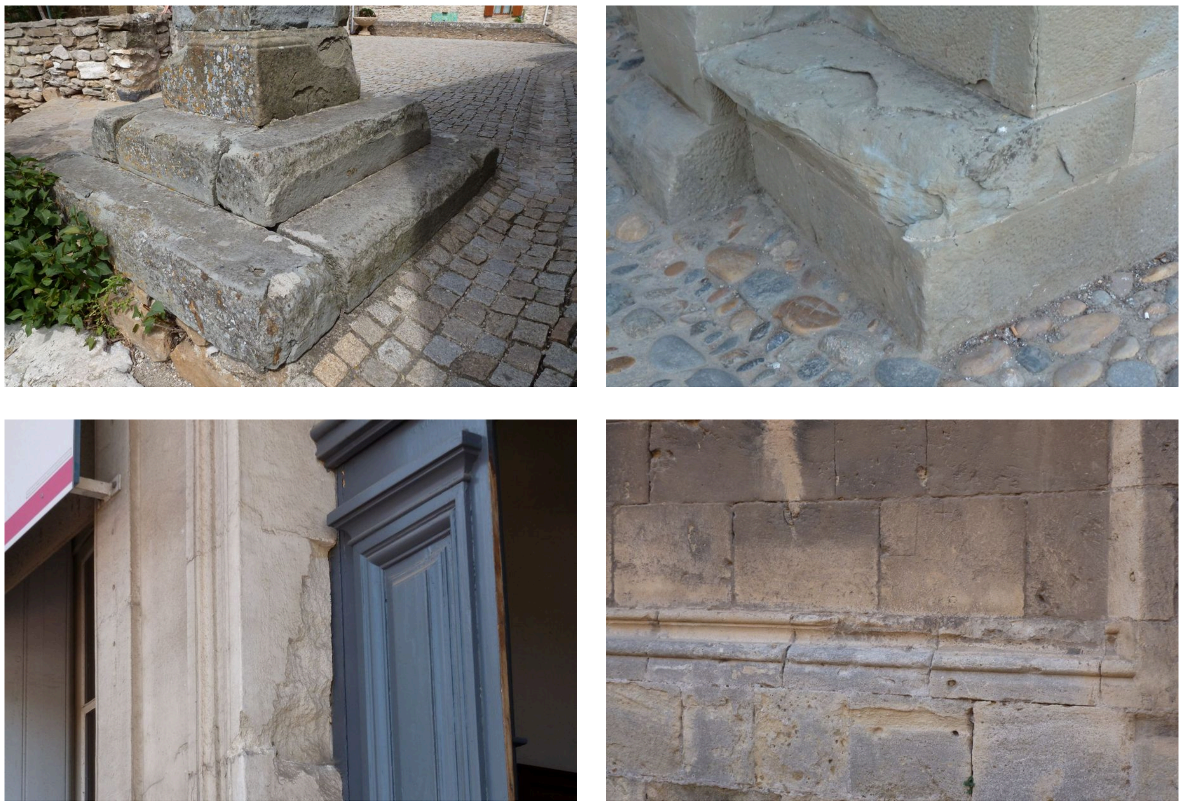


Figure 3.5: Stone structures affected by corossion.

Peeling

In the case of mudbrick houses a plaster coating is painted over the mudbrick to protect the material from corrosion. Ofcourse that layer doesn't last forever: it can be peeled off and reveal the mudbrick hiding behind it. Figure 3.7 has a few pictures showing this effect on mudbrick houses.

Roof Collapse

Roof slopes are also vulnerable to humidity. Glazed ceramic tiles are placed on those slopes to shield materials that can be easily corroded with water. This increases the life time of a house significantly, as roof tiles are effective at keeping water out and letting it flow to the ground where it will be eventually drained. However if a set of tiles was to be removed the patch of uncovered roof material would gradually corrode with time. Since in most cases tiles are supported by other tiles, the fallen tiles would slowly drag other tiles along with them as the damage spreads along the roof. Such an effect is noticeable in old houses, where after many years of being neglected, holes form on the roofs. Figure 3.8 and 3.9 show examples of houses with collapsing rooftops.

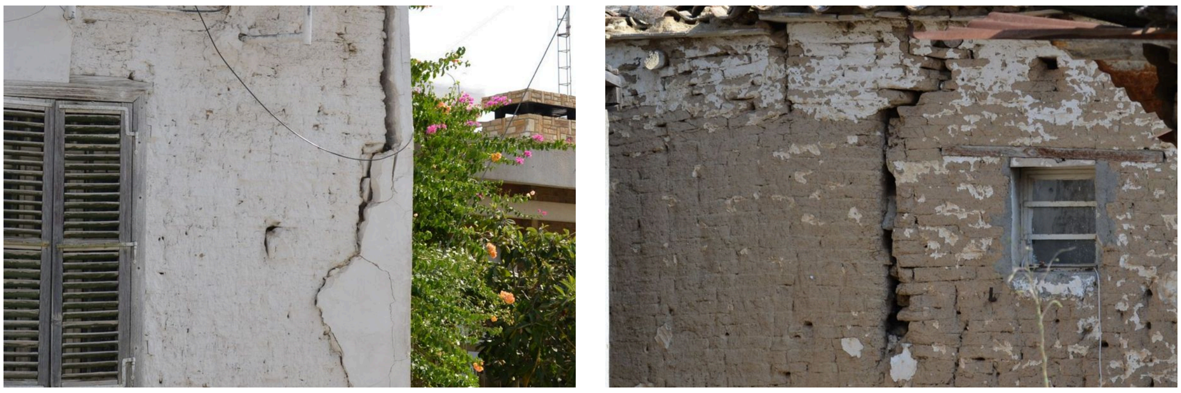


Figure 3.6: Cracks on the walls of mudbrick houses.

Tilting Walls

In many cases the earth below a building is not stable. The absorbsion of humidity causes soil to expand, while evaporation or displacement of that humidity causes the soil to contract. Buildings that are build on top of such soils, and without any treatment (removing humidity) are subject to structural cracks. More specifically, the walls of the building may be pulled inwards or outwards forming those cracks in the process. From the outside these walls may appear tilted, as is in the examples in figure 3.10.

Rubble

The accumulation of rubble is a great indication of where and how much a building was damaged. Rubble piles form under collapsed rooftops, next to broken walls and in ruins. Figure 3.11 shows the rubble of a several abandoned mudbrick houses.

Ruins

After the longterm exposure to humidity and earthquakes, abandoned structures become ruins. In the case of houses made of natural materials, we have the total collapse of roofs and walls, the accumulation of rubble and the uncontrollable spreading of vegetation. Figure 3.12 shows mudbrick house ruins, while figure 3.13 shows ruins of stone houses.

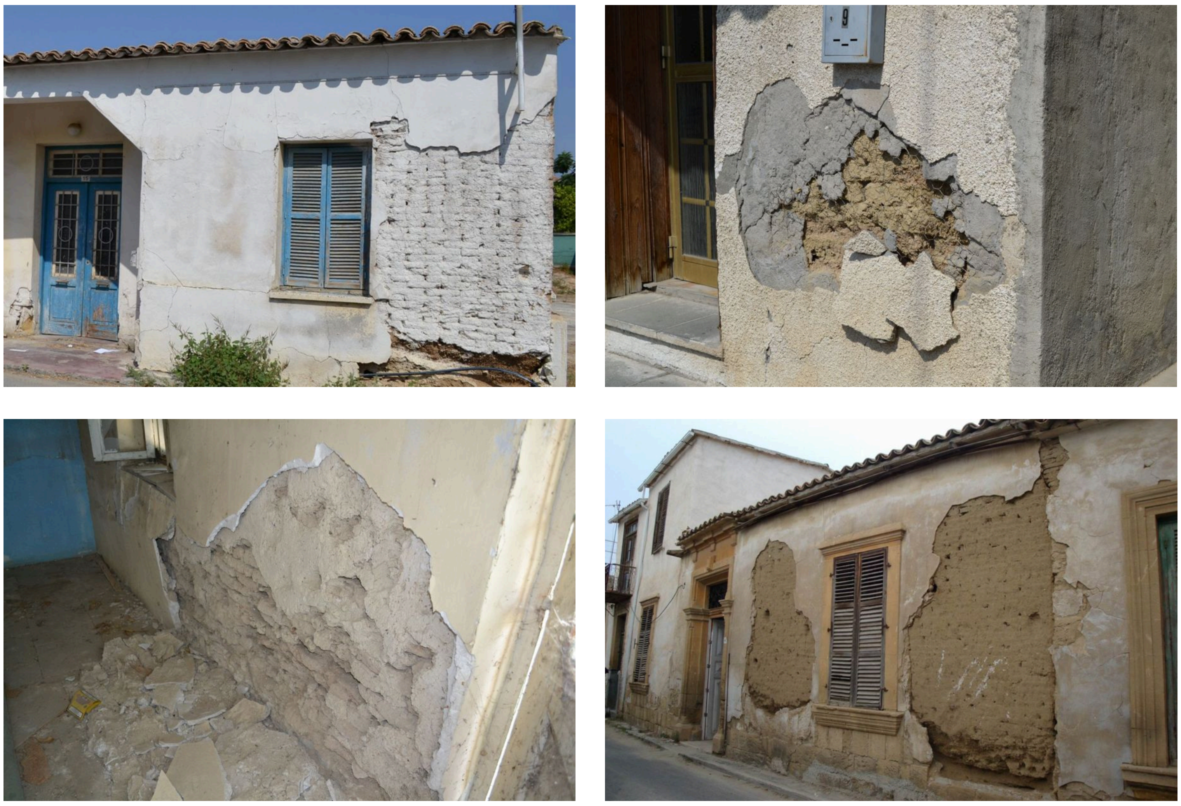


Figure 3.7: Paint and plaster layer is peeled off, revealing the mudbricks.

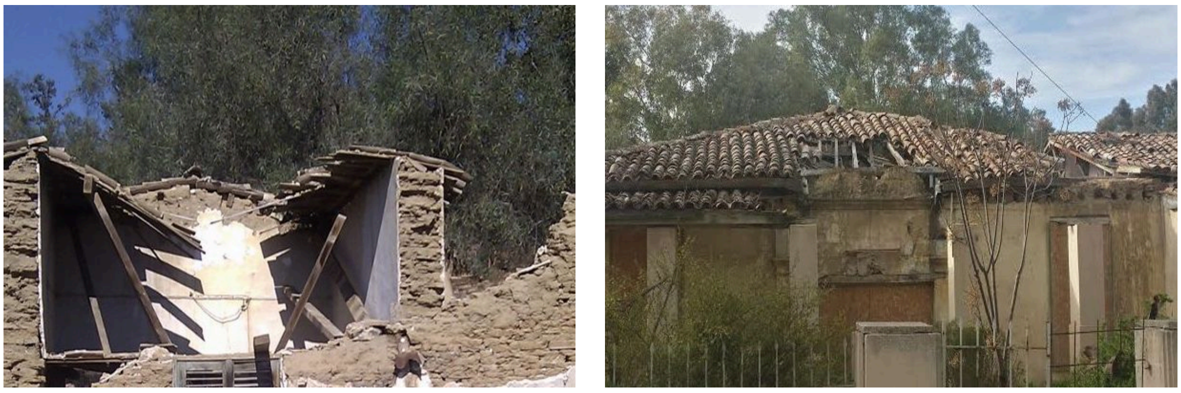


Figure 3.8: Mudbrick house with collapsing rooftops.

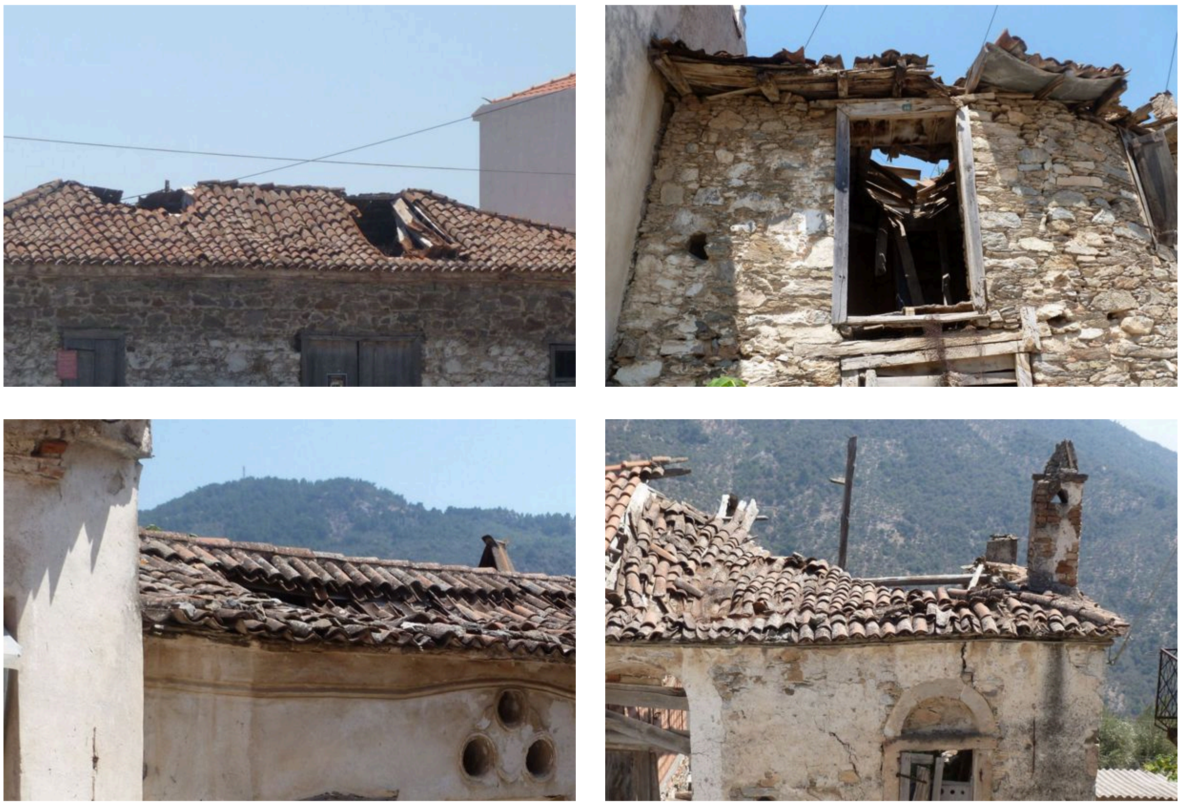


Figure 3.9: Stone houses with collapsing rooftops.

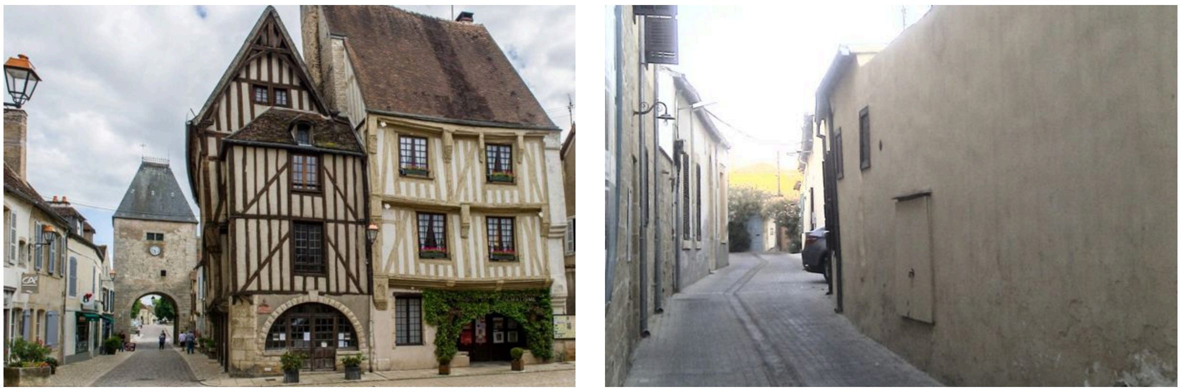


Figure 3.10: The walls are tilted outwards.

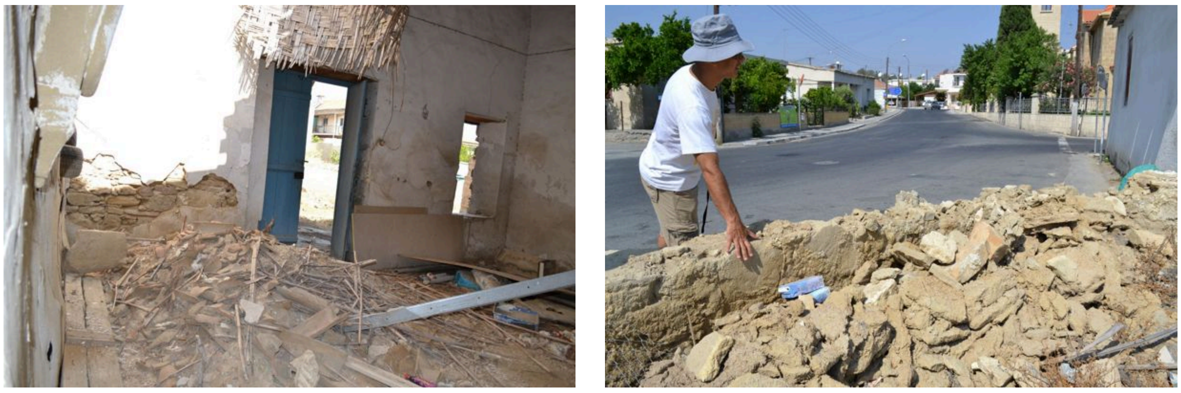


Figure 3.11: Rubble accumulation from the ruination of mudbrick houses.

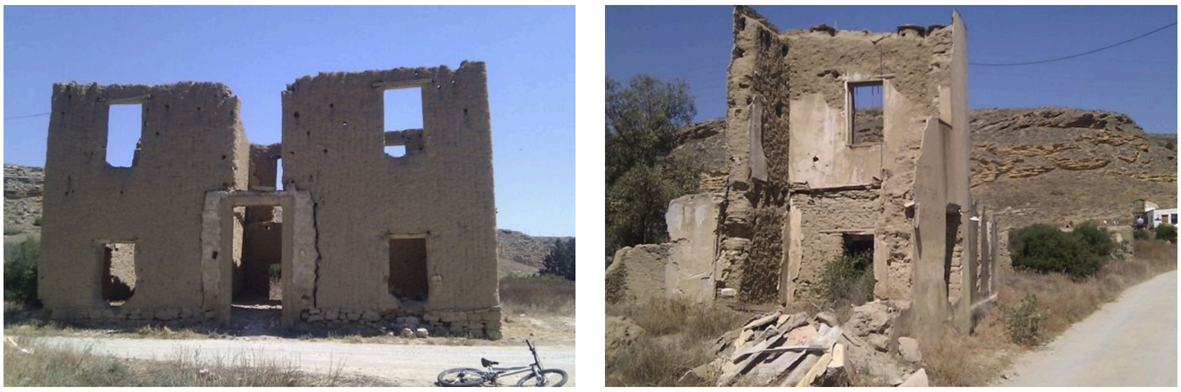


Figure 3.12: Mudbrick house ruins.

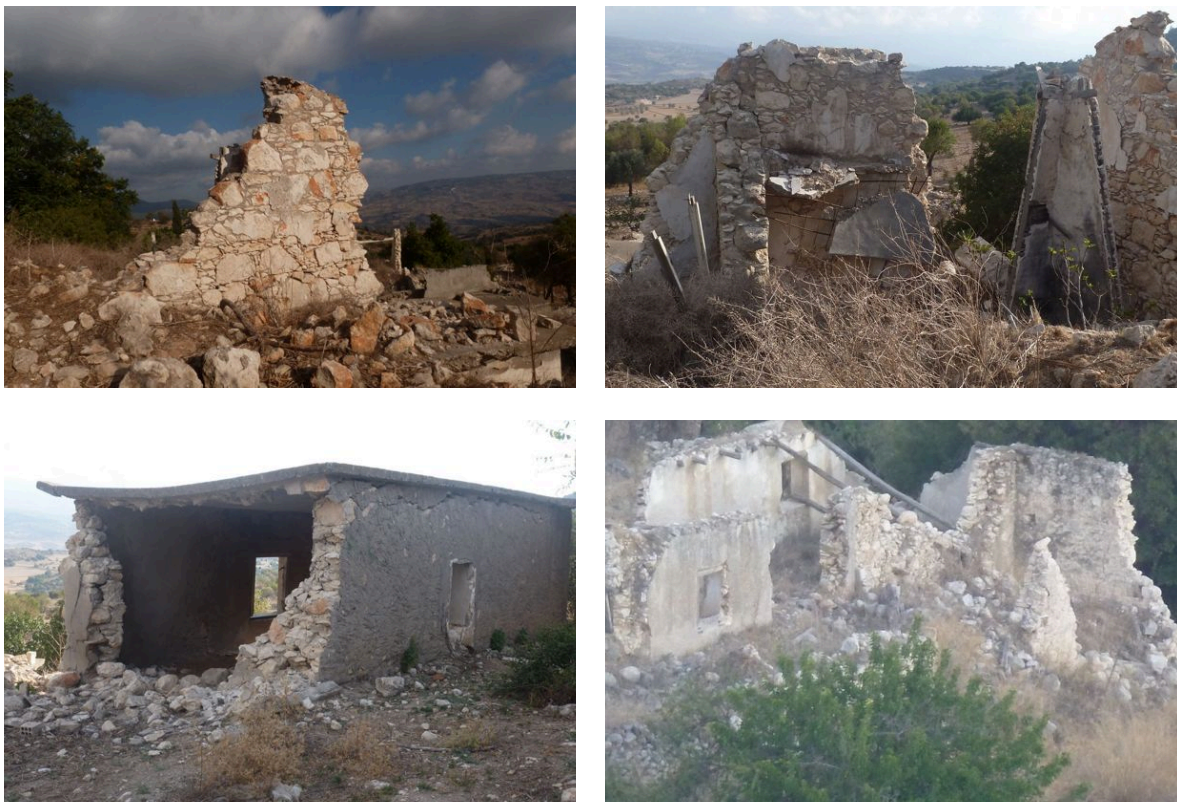


Figure 3.13: Ruins of old stone houses.

Chapter 4

Aging Simulation Pipeline

Contents

4.1 Overview

In Subsection 2.2.2 the CGA Shape Grammar pipeline was illustrated and described. Our aging simulation pipeline extends upon the CGA Shape Grammar by adding several new components. The goal of this new pipeline is to perform simulation to calculate where the weathering should occur and then apply the degradation using various techniques. Please use Figure 4.1 as a visual guide while we explain each component of the new pipeline.

Mass Modeling

First we form the mass model, either by assembling solids or extruding footprints. This component remains the same for our simulation pipeline.

Component Split

We extract the facades of the volumetric representation of our building. Although this component remains the same on an abstract level, but depending on the implementation it might require an extra step. In our implementation it is required to give each face a unique identifier.

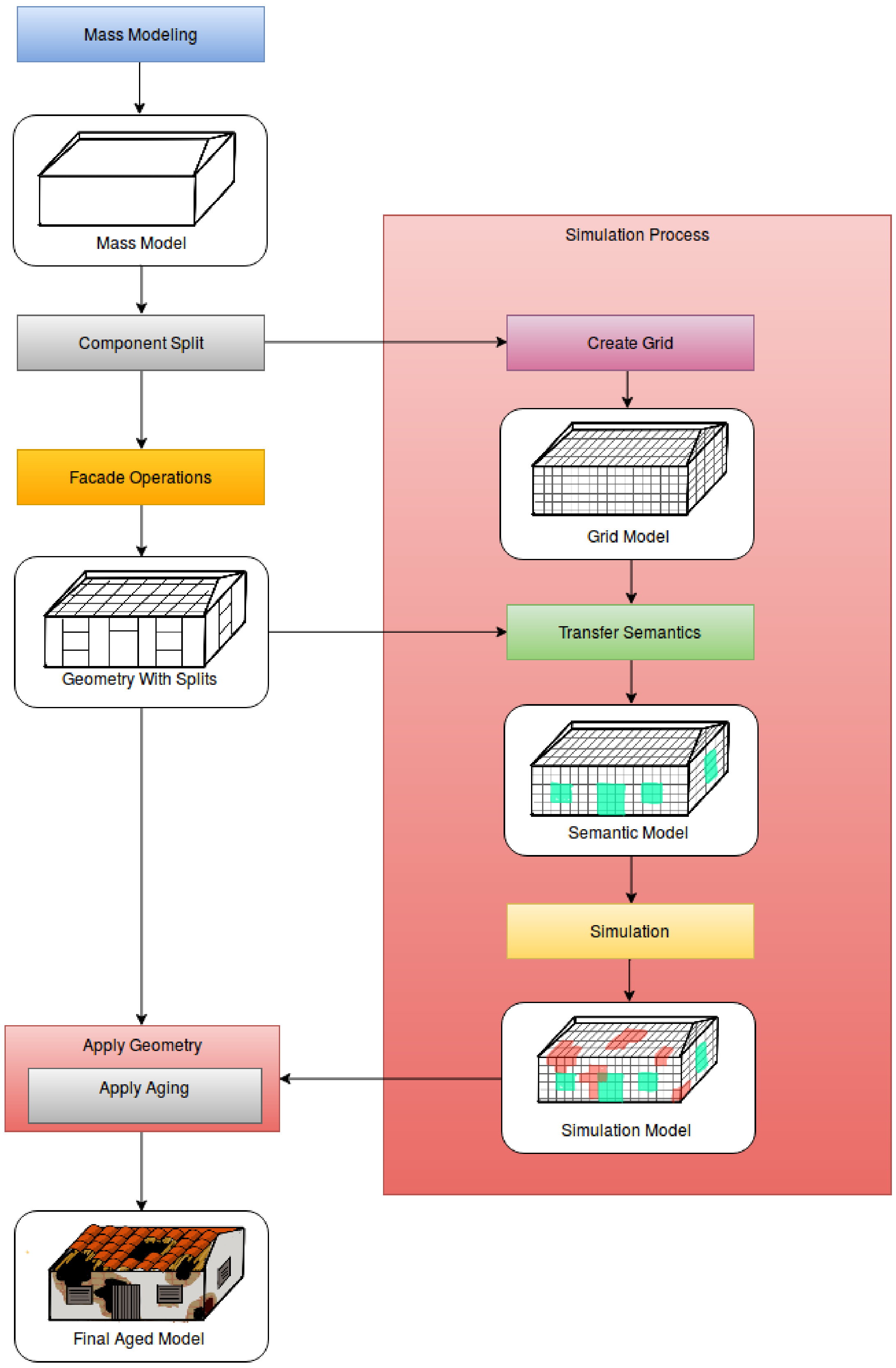


Figure 4.1: Flow diagram of the aging simulation pipeline.

Grid Model

We form a uniform grid on the surface of the model that will store simulation related attributes. The smaller the cells of the grid the higher the simulation resolution.

Transfer Semantics

We transfer information regarding different materials and objects on the grid model. To get this information we need to reach the Facade Operations stage in the CGA Shape Grammar pipeline. The transfered semantics contributes to the accuracy of the simulation.

Simulation

This component uses multiple simulation techniques, set by the user, in order to calculate the points of degradation on our building.

Facade Operations

We will then split our grid model the same way we would in the CGA Shape Grammar pipeline. Depending on the implementation approach, the split components may need to be modified. In our implementation we created an alternate version of the CGA Shape Grammar split operations, to split each facade while ignoring the existence of the grid model cells.

Apply Geometry

In the CGA Shape Grammar pipeline we place detailed models on our models, such as doors and windows. The difference now is that we will concurrently apply degradation using various methods. In some cases the application of aging effects at a certain point may interrupt the placement of a model, such as the cases where roof tiles fell through a hole on the roof.

4.2 Simulation

In Section 4.1 we have presented our aging simulation pipeline which extends the CGA Shape Grammar pipeline. In this section we will explain in more detail the simulation process. Please note that for our implementation we decided to only take into consideration the weathering effects caused by humidity from rain. For that reason the extension of the simulation process to add more weathering effects is encouraged. Figure 5.75 illustrates our simulation process in more detail.

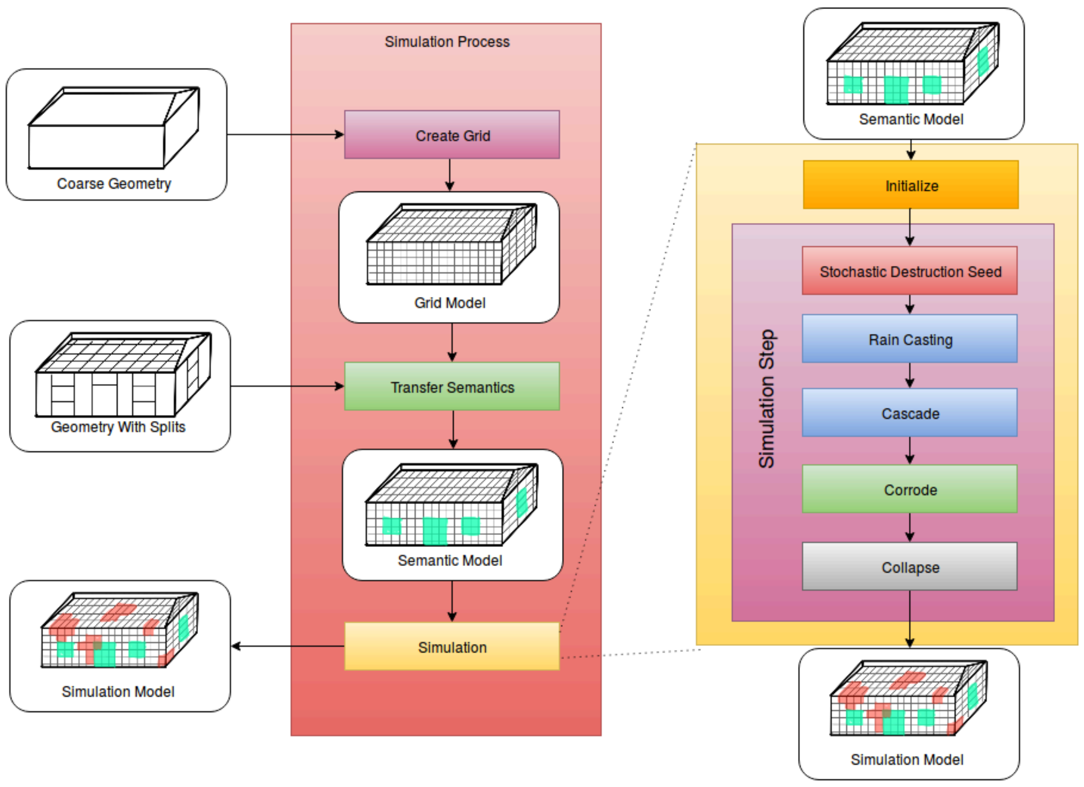


Figure 4.2: Flow diagram of the simulation process.

Attributes

In our simulation we use the term attributes to refer to values representing a point on the building's surface. The simulation uses those attributes to understand the status of a point and to decide how it should be affected by damaging effects. Figure 4.3 illustrates the attributes used in the simulation.

The current version of our work uses 4 layers of attributes for every single point. The first is called rain and it is used to represent the amount of humidity accumulated by rainfall, directly or indirectly (from downpour). It is important to differentiate points of the building that receive very little rain and points that receive a lot of rain over-all, as this affects the speed of corrosion.

The next attribute has the name Shield and its purpose is to act as a shield to the underlying structure mainly from heavy weather conditions. Think of a roof tile, or the plaster and paint layer over a wall. The 'amount' of Shield does not represent any realistic quantity but it is used simply to adjust the robustness of certain parts of the building. For example the artist may want to add a tougher material on a certain part of a wall.

Next we have the Support attribute which acts as a brake to the aging process. A real life example would be the horizontal support beams that support parts of the roof. The Support attribute also makes sure that a point does not collapse as easily as its neighbors: this can be useful in cases where entire parts of a roof collapse, and we want certain parts to keep holding on, due to underlying supporting structures.

Finally we have the Composite attribute which measures the damage inflicted on the layer behind the shielding. This Composite layer can be thought of as composite materials that roof tiles sit on, or the bricks behind the plaster layer. During the Aging Application process we take into consideration the value of the Composite layer of an area to decide on the severity of the weathering effect.

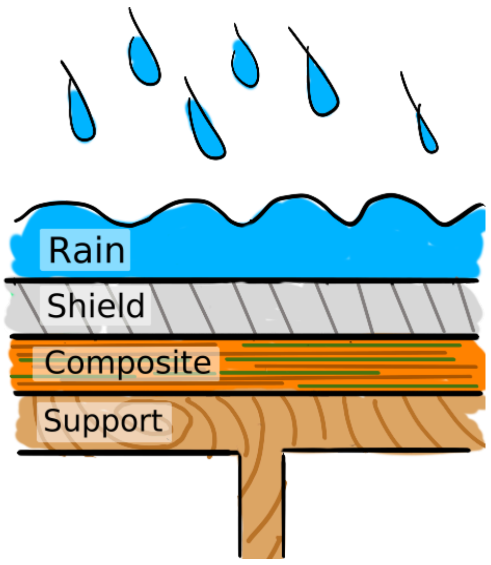


Figure 4.3: An illustration of the four layers of attributes on a surface.

Grid Model

To store our attributes we use a “Grid Model” of our building. A grid model is a model that has the same shape as the building model, but it’s surface is subdivided into a uniform mesh allowing for more points to be stored on a surface. As we have mentioned previously in the Previous Work chapter, we strayed away from the texture space approach because we wanted each point to have spatial information as well as access to its neighbors and environment in 3D space. We acknowledge that this approach does introduce aliasing problems as well as memory and performance issues. A user can change the resolution of the grid model to either increase the quality, or decrease resource usage.

Semantics Transfer

The transfer of Semantics contributes to the accuracy of the simulation. With only the grid model, we lack any information that alludes to the existence of different materials and objects on our model. Such information can only be attained towards the end of the procedural pipeline, more specifically right after the Facade Operations have finished. For that reason we use the procedural model after the facade operations to transfer that semantic information on our grid model in the form of attributes. The user has the freedom to transfer any value to any one of the 4 attribute layers we have mentioned previously. For example points that belong to a window should have more humidity resistance than its surrounding mudbricks, or the points belonging to supporting beams of the roof should have a much higher 'Support' value than surrounding points.

Simulation Component

First I'll explain was "Stochastic Destruction Seed" is and why it is useful: it is a process that randomly strips away parts of the shield layer. Without such a tool, the shield layer would remain intact indefinetely (at least until user intervention), thus keeping the simulation static and unchanging. Although this tool may seem unsophisticated it does has a basis in reality. In mudbrick houses there are various forces that misplace roof tiles or remove the plaster of the wall. This forces the owners of the mudbrick houses to regularly maintain the house to avoid its degradation.

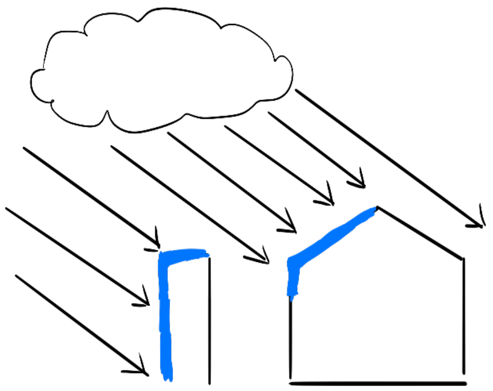
The simulation component can contain any number of simulation methods making use simulation attributes, geometry attributes, and the environment to calculate the amount of degradation on each point. In our case we have implemented four subcomponents: Rain Casting, Rain Cascade, Corrosion and Collapse. Illustrations of each sub component are presented in Figure 4.4 and Figure 4.5.

Rain Casting is a component that performs ray casts from each point of the building towards the sky. The ray vector matches the angle at which rain falls (or at least an average estimate) and the raycast checks whether rain is blocked by another structure.

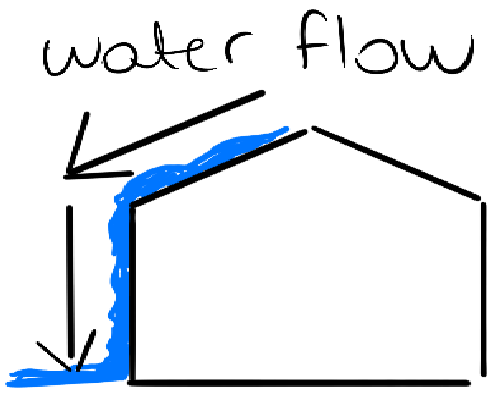
Another useful component in Rain Cascade which finds the points that rain water reaches indirectly. Rain can fall on any point of the building, but it can propagate nonetheless to other lower parts of the building.

Then there is Corrode which is a component that calculates the amount of degradation on each point depending on the amount of humidity, whether the point is shielded or not, and whether there is underlying support or not.

Finally there is the Collapse component which complements the previous component. Its purpose is to collapse points that have no logical reason to stay intact. For example an entire segment of a roof slope collapsed: since the lower tiles were mostly supported by the upper tiles, they will tend to fall regardless of the roof-tile shielding.

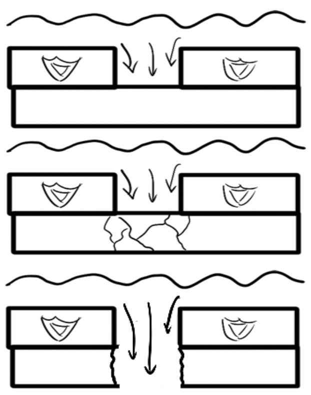


(a) Rain casting: simulates rainfall

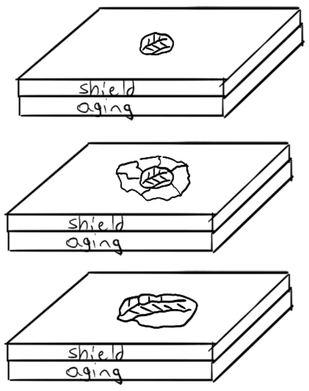


(b) Cascade: simulates flow of water

Figure 4.4: An illustration of he different simulation components



(a) Corrosion: destroys unprotected surfaces exposed to humidity



(b) Collapse: collapses points with no neighbouring support

Figure 4.5: An illustration of he different simulation components

4.3 Aging Application

Aging application is the process that applies weathering effects on our procedural model using the simulated grid model as a reference. At this point artists can use their imagination and skills to manipulate the geometry, apply special shaders or add grunge textures. We have developed three aging applications tools for our proof of concept. In Figure 4.6 we illustrate the process of aging application.

Select Geometry

The three tools we have mentioned work in different ways and produce different results. For that reason we need to be carerful which tool we use with each part of the mesh. Thus we separate our mesh into 3 types of geometry, each type corresponding to a different aging application tool. For example type A geometry is the part of the geometry that will be processed by the Deepening tool, type B will be processed by Peeling and type C will be processed by NoInsert.

Deepening

The first tool is called Deepening and it involves moving points along their normal based on the damage inflicted on the Composite layer attribute. If a quad primitive's points are all completely destroyed, then the quad is removed, forming holes in areas of severe degradation. Wall surfaces can make use of this technique but require subdividing to add enough resolution first. Roof surfaces are more suitable for this tool. Firstly the surface is already subdivided for the tile seams. In addition to that, when using deepending the quad primitives change their orientation depending on the surrounding destruction. This in turn creates an interesting effect of tiles leaning towards the roof hole, seemingly ready to fall in next.

Peeling

The second tool we use for aging application is called Peeling. It works in a more discrete fashion than Deepening in that it assumes that our surface is made our of multiple layers. The user defines the different layers along with a range of Composite attribute values that it will make it visible. Depending on the amount of degradation on an area different layers will dissapear revealing the layers behind them. For example we may have a plaster layer then two layers of bricks. With minimal damage the paint layer will be peeled off revealing the bricks

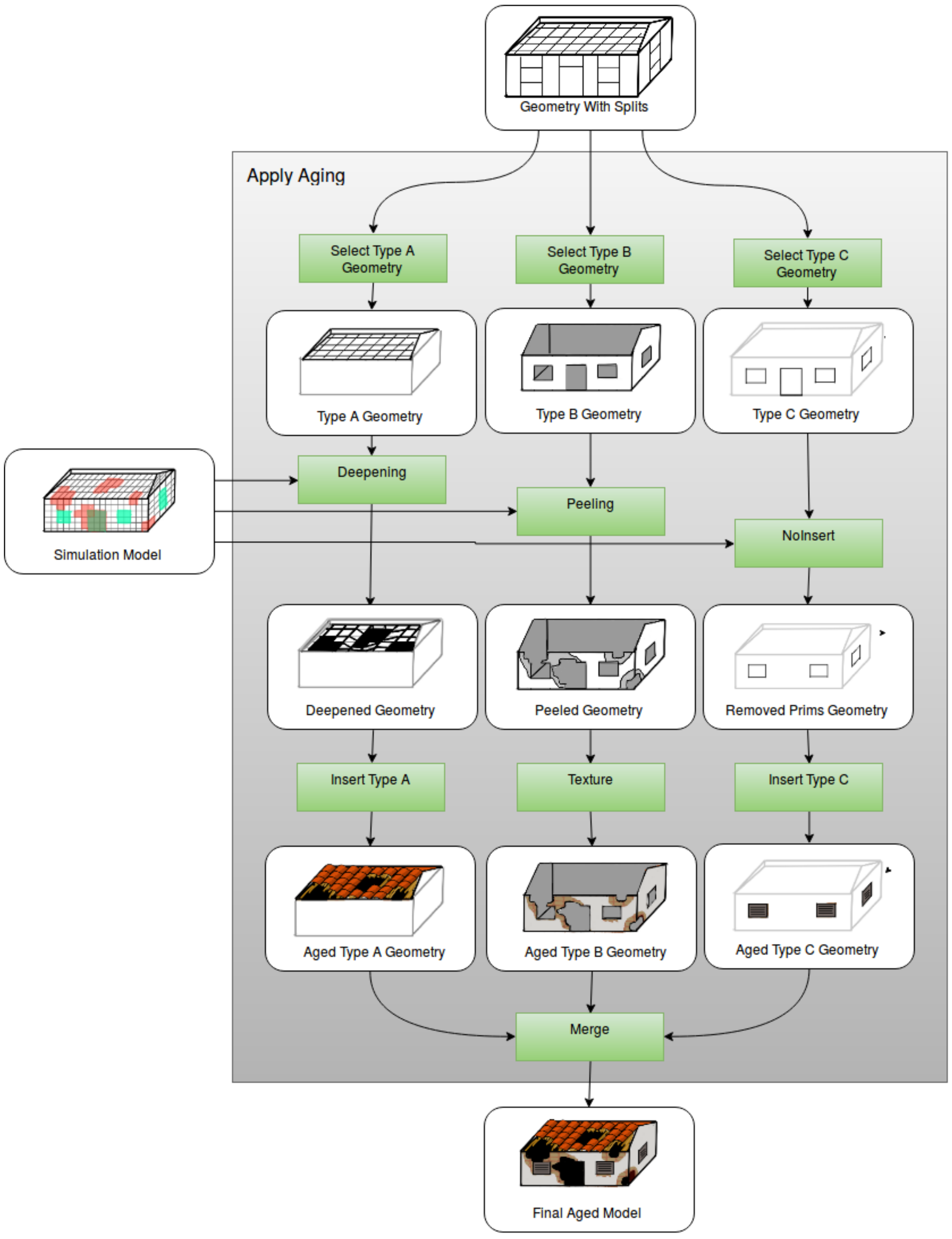


Figure 4.6: Flow diagram of the simulation process.

underneath. With enough damage the front layer of bricks will dissapear revealing the second layer of bricks. Then if the area is completely ruined all layers will be invisible revealing the interior of the house.

NoInsert

Finally we use the NoInsert tool to prevent the insertion of detailed models in ruined areas. Similar to how Deepening removes quads to simulate the fall of the roof tiles, the NoInsert removes primitives to simulate the fall of wall features such as windows and doors. As opposed to Deepening however we do not perform any displacement on the vertices of the primitive.

Chapter 5

Implementation

Contents

This chapter is separated into three sections. First we will present the program we chose to implement our pipeline, Houdini FX, along with all of the in-built tools and features we have used from it. Then we will describe how we used Houdini to implement the basic CGA Shape Grammar toolset. And finally we will describe how Houdini was used to implement the simulation pipeline extension.

5.1 Houdini

Houdini is a set of tools that procedurally create content for movies and games. It handles almost all tasks of CGI pipeline including Animation, Simulations, Shading and Graphics. We have chose to use Houdini for two reasons: A) It was designed in a way that it facilitates procedural design. In more traditional modeling and animation packages you work on a single model, modifying the geometry, the shading etc. With Houdini you work on a pipeline that processes models: you can create a pipeline of operations that perform a series of complicated operations and modify any part of that pipeline that you wish. B) It is extremely flexible and customizable. Not only does it have a complete toolset of geometry manipulation operations, it allows you to write scripts to do specialized processing. Although CityEngine has a working CGA Shape pipeline it does not offer the same level of flexibility as Houdini.

In the next subsections we will explain some basic terminology of the Houdini Software. We will not go into great detail about each subject and there is no need to present every feature that this software package contains, only the tools that we use in our simulation pipeline. Note: We have used Houdini Apprendice version 16.0 to implement the CGA Shape Grammar toolset and the simulation pipeline.

5.1.1 Notation

Networks

A typical Houdini workload is done inside a network of nodes (See Figure 5.1). Nodes represent objects, operations or subnetworks and can be interconnected in various ways to form a directional graph. The processing of a geometry usually starts from a node that creates a mesh. That mesh is then directed to the next node, where it will be processed or used in some way. A network can also be encapsulated in a Subnetwork node, allowing for abstraction and reusabillity. On top of that a Houdini user can create Digital Assets: custom subnetworks that are stored and can be reused in multiple projects. This also gave the oportunity for an online economy of buying and selling Digital Assets online.

Operators

Nodes in a Houdini network are also known as Operators. The most common type of Operator is the Surface Operator (or SOP), which specializes in creating or manipulating geometry. There

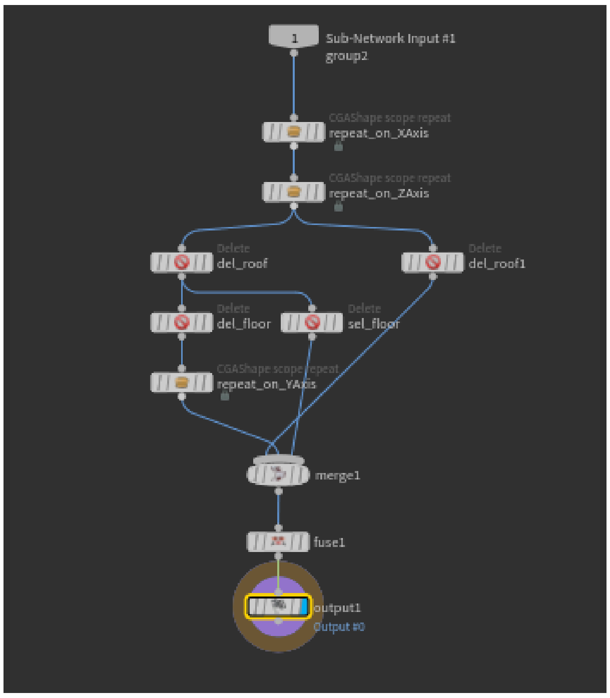


Figure 5.1: An example of a Houdini network.

are also Vector Operators (VOPs) for handling volumetric objects, Shader Operators (SHOPs) for applying shaders, Composition Operators (COPs) for image composition and many more. Our simulation pipeline makes use of SOPs most of the time.

Geometry

Geometries in houdini are not unlike standard geometry definitions. The geometry itself contains a list of points and a list of primitives. Primitives are polygons of arbitrary shape (usually triangles or quads) that are made out of vertices. A Vertex is different than a Point: vertices are unique to each primitive, but points are shared by connected primitives. See figure 5.2 for a visual presentation.

Attributes

Think of attributes as values. They can represent a single number, or a vector (color, normals, position etc). Houdini stores attributes at different levels of the Geometry. First it has attributes at the “detail” level which encapsulate the entire mesh. Then it stores attributes per primitive, per point and per vertex. These attributes can be used for a variety of tasks including the manipulation of the geometry.

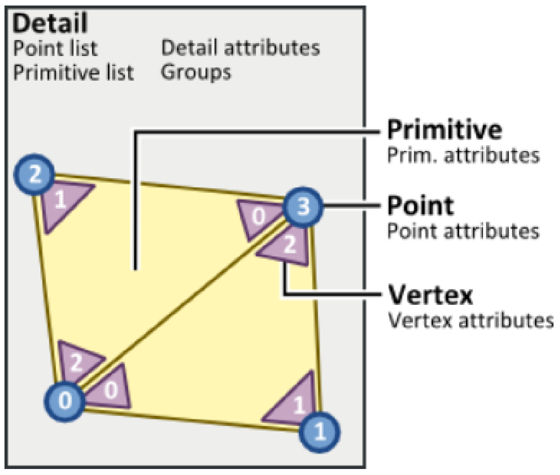


Figure 5.2: An illustration for the various components of a mesh in houdini. Source: [27].

Houdini has a number of in-built Operators that manipulate or create new attributes. One such example is the Attribute Create geometry node, which creates a new type of attribute on any level of the geometry. You can set attributes using surface operators such as Vertex, Point and Primitive.

Parameters

Operators in Houdini have several input parameters. These parameters can influence the way the operator works; for example we can set the translation vector for a Transform node. Parameters can be of any type: integers, floating point values, strings, vectors and more. They can also be customized by the user.

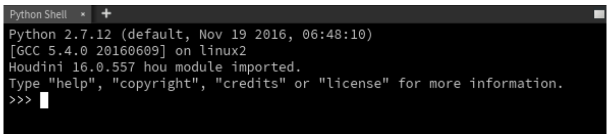
One of the strengths of Houdini however is to use expressions in parameters. Expressions are pieces of code that have a single output and are evaluated when Houdini "cooks" the node. By parametrizing operators, an artist is able to create an entirely procedural system.

Groups

Points and primitives in Houdini can belong to one or more groups. Group assignment can be based on almost anything the user can imagine: from the value of an attribute, to the position of the entity in space. This feature is very helpful in selecting and filtering vertices based on a shared feature of multiple points/primitives. An important thing to note is that the points/primitives themselves see the group as another attribute, where if the value is 1 it means that they belong to that group. This means that by adding more groups, a new attribute is added to all the points/primitive, adding some extra cost to the storage of the model in physical memory.

Python

The Houdini Software supports the use of python scripts to perform various calculations and interact with the Houdini environment. A python script can process geometry the same way a built-in Surface Operator can. By using the Houdini Object Model, a python library, a user can manipulate attributes and geometric entities with relative ease. It is possible to simulate the function of any in-built node with the use of Python. In addition to that a python script can also manipulate the Houdini Network and work environment: a user can create new nodes, move them around and connect them in various ways. A use for this functionallity would be the creation of macros that create a complicated subnetwork from scratch. Python code can be executed via a terminal-like interface in Houdini (See Figure 5.3), via the Python Operator, or by using scripts in the operator properties.



```
Python 2.7.12 (default, Nov 19 2016, 06:48:10)
[GCC 5.4.0 20160609] on linux2
Houdini 16.0.557 hou module imported.
Type "help", "copyright", "credits" or "license" for more information.
>>> 
```

Figure 5.3: Screenshot of the Python Shell in Houdini.

5.1.2 Relative Nodes

In this section we will mention the Houdini in-built nodes that we use in our implementation of the CGA Shape toolset and the simulation pipeline.

Group Node

Houdini gives users the ability to place points or primitives in a group with the Group Operator. This can be useful in cases where we want to manipulate certain parts of the geometry depending on whether they belong to a group or not. Many in-built Surface Operators use a group name as input, allowing users to filter and process only one part of the geometry. Figure 5.4 shows a group node as it appears in the Houdini Software environment.

The Group Operator gives a user many options to select points/primitives to assign to a group. Figure 5.5 shows the interface of the group node, containing all the relative options and parameters that a user can modify. First of all the user decides on the name of the group, which acts as a label and an identifier. Then an entity type between Primitives, Points and Edges is selected.

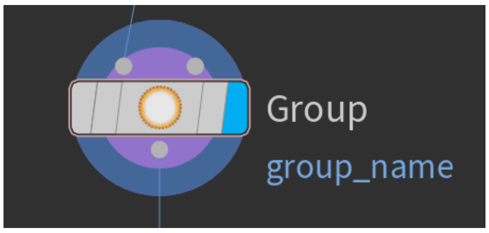
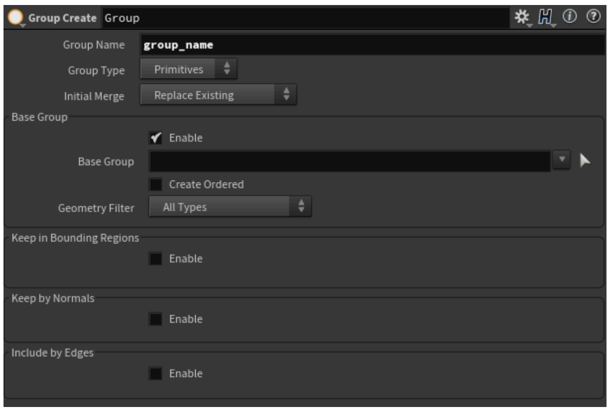


Figure 5.4: The Group Operator in the Houdini Software.



Group Create Group

Group Name: group_name
Group Type: Primitives
Initial Merge: Replace Existing

Base Group
- [✓] Enable
- Base Group:
- [ ] Create Ordered
- Geometry Filter: All Types

Keep in Bounding Regions
- [ ] Enable

Keep by Normals
- [ ] Enable

Include by Edges
- [ ] Enable

Figure 5.5: The Group Operator parameter interface.

In the case that a group with the same name was created previously, the merging mode option will decide how to handle merging with entities belonging to the group. The selected entities can be added to a group based on the known set operations (union,subtract,intersect) or completely remove those entities from that group.

Subsequently the user can choose one or more ways of selection. The first way uses an expression which can be based on attribute values of the selected entity or group names. The second way involves a bounding volume check, so that entities within the defined bounding box/sphere will be assigned to the group. Then we can select entities based on their normal. More specifically a vector and angle tolerance is defined, and it selects entities that have normals within the angle tolerance of the defined vector. Finally we can select edges based on various characteristics such as length, angle and depth.

Delete Operator

The Delete operator is an elementary tool allowing for deletion of filtered primitives or points of the input geometry. Figure 5.6 shows a delete node in Houdini. In a similar fashion to the Group Operator, the Delete Node can select entities using varrious methods (attribute values, group names, normal direction, bounding volume etc.) and either delete them or keep them.

This proves useful when the user wants to filter part of the mesh to process it separately. See Figure 5.7 for a screenshot of the delete node parameter interface.

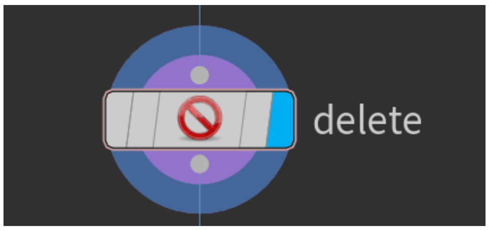
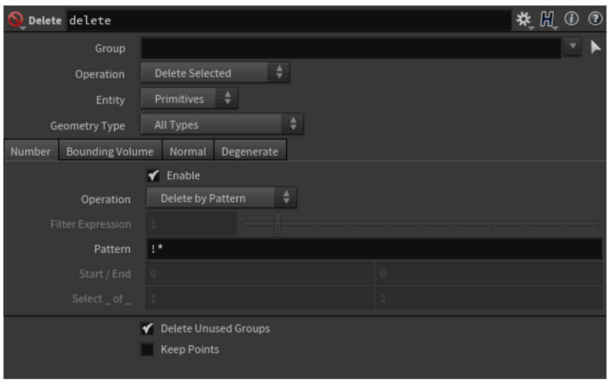


Figure 5.6: The Delete Operator in the Houdini Software.



Delete delete

Group

Operation: Delete Selected

Entity: Primitives

Geometry Type: All Types

Number | Bounding Volume | Normal | Degenerate

✓ Enable

Operation: Delete by Pattern

Filter Expression: 1

Pattern: !*

Start / End: 0 0

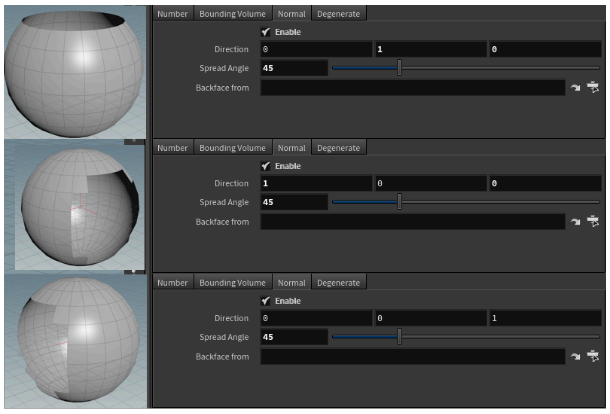
Select _ of _: 1 2

✓ Delete Unused Groups

Keep Points

Figure 5.7: The Delete Operator parameter interface.

In Figure 5.8 we show an example of usage. We use deletion based on normal, by using 3 different vectors as input with an angle tolerance. The left side of the picture shows the effects when applied to a spherical mesh, and the right side shows the parameters used.



Number | Bounding Volume | Normal | Degenerate

✓ Enable

Direction: 0 1 0

Spread Angle: 45

Backface from

Number | Bounding Volume | Normal | Degenerate

✓ Enable

Direction: 1 0 0

Spread Angle: 45

Backface from

Number | Bounding Volume | Normal | Degenerate

✓ Enable

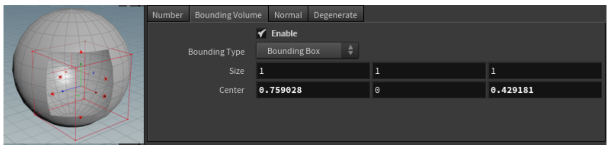
Direction: 0 0 1

Spread Angle: 45

Backface from

Figure 5.8: Deleting primitives based on normal.

Another example can be seen in Figure 5.9 where we delete part of the sphere using a bounding box.



✓ Enable
Bounding Type: Bounding Box
Size: 1 1 1
Center: 0.759028 0 0.429181

Figure 5.9: Deleting primitives based on a bounding volume.

Transform Node

Another fundamental operator is the Transform node, which as its name suggests, allows the user to translate, rotate and scale a geometric object. It also includes a filter box that allows for the transformation of only the selected entities. In addition to that every parameter, including the ones for transformation values, can be based on an expression and from values outside of the current node. This level of parameterization allows for some truly powerful procedural designs. Figures 5.10 and 5.11 show the Transform Operator in Houdini.

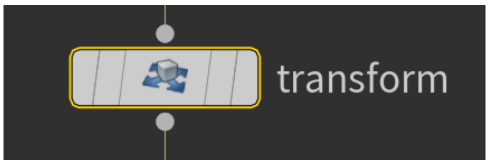
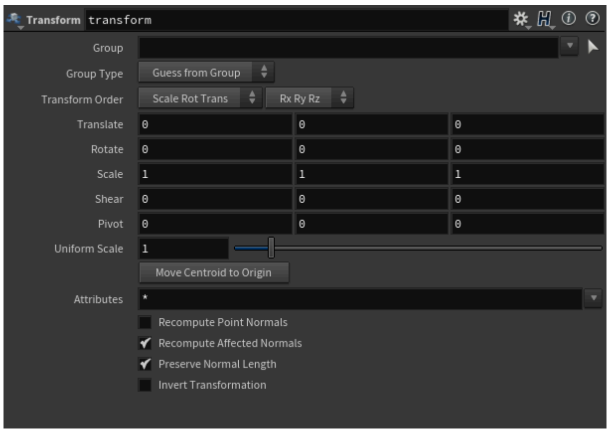


Figure 5.10: The Transform Operator in the Houdini Software.



Transform transform
Group:
Group Type: Guess from Group
Transform Order: Scale Rot Trans, Rx Ry Rz
Translate: 0 0 0
Rotate: 0 0 0
Scale: 1 1 1
Shear: 0 0 0
Pivot: 0 0 0
Uniform Scale: 1
Move Centroid to Origin
Attributes: *
☐ Recompute Point Normals
✓ Recompute Affected Normals
✓ Preserve Normal Length
☐ Invert Transformation

Figure 5.11: The Transform Operator parameter interface.

Clip Node

In cases where we need to cull part of the geometry using a plane, we can use the Clip node (see Figure 5.12).

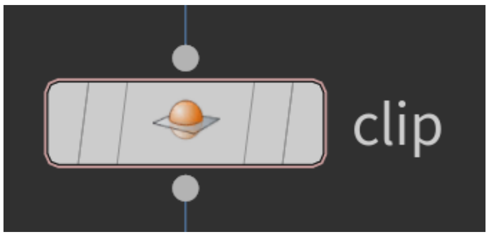
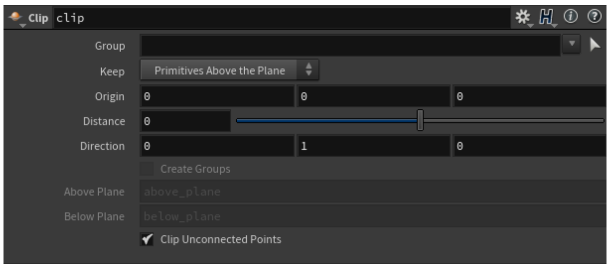


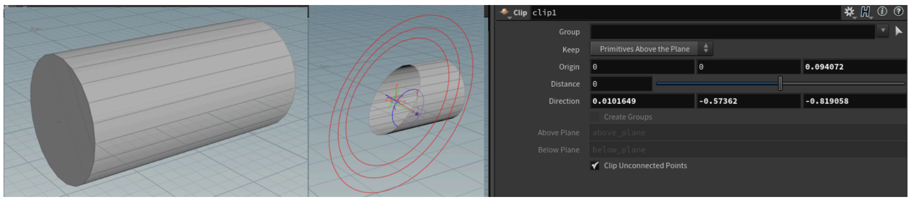
Figure 5.12: The Clip Node in the Houdini Software.



```
Clip clip
        Group
         Keep  Primitives Above the Plane
       Origin  0          0          0
     Distance  0
    Direction  0          1          0
               Create Groups
  Above Plane  above_plane
  Below Plane  below_plane
               ✓ Clip Unconnected Points
```

Figure 5.13: The Clip Node parameter interface.

The user sets the normal of the clipping plane as well as its origin and distance. The origin of the plane is used as a starting point, and the distance parameter is used to move the plane itself forwards or backwards, depending on the plane normal. Additionally the user has the option to clip geometry behind the plane, or in front of the plane without needing to change the plane normals. We use the clip node to select subsections of a facade: we do this by using two clipping planes that have the same origin and normal, but are distanced apart by a predetermined distance D, and the first plane clips geometry behind it while the second clips geometry ahead of it. Figure 5.13 shows the parameters of the Clip Node.



```
Clip clip1
        Group
         Keep  Primitives Above the Plane
       Origin  0          0          0.094072
     Distance  0
    Direction  0.0101649  -0.57362   -0.819058
               Create Groups
  Above Plane  above_plane
  Below Plane  below_plane
               ✓ Clip Unconnected Points
```

Figure 5.14: Usage example of the Clip Node.

Figure 5.14 shows the clip node being used to clip a cylindrical mesh. On the left side of the figure you can see the original mesh. In the middle you can see the clipped mesh, along with the clipping plane. And the chosen parameters are shown on the right side of the figure.

Poly Extrude

The Poly Extrude node extrudes the filtered entities (see Figure 5.15). Despite its simple premise, it is a quite intricate tool with many features. We will briefly explain most of its features by using Figure 5.16 as a visual guide.

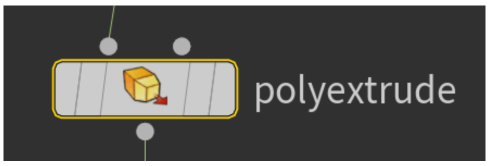
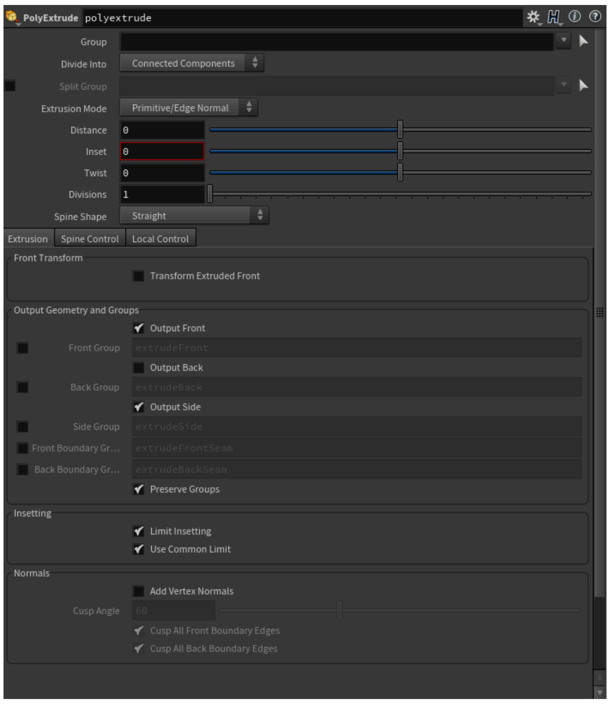


Figure 5.15: The Poly Extrude Node in the Houdini Software.



PolyExtrude polyextrude

Group	
Divide Into	Connected Components
Split Group	
Extrusion Mode	Primitive/Edge Normal
Distance	0
Inset	0
Twist	0
Divisions	1
Spine Shape	Straight

Extrusion | Spine Control | Local Control

Front Transform

- [ ] Transform Extruded Front

Output Geometry and Groups

- [✓] Output Front
- Front Group: extrudeFront
- [ ] Output Back
- Back Group: extrudeBack
- [✓] Output Side
- Side Group: extrudeSide
- Front Boundary Gr...: extrudeFrontSeam
- Back Boundary Gr...: extrudeBackSeam
- [✓] Preserve Groups

Insetting

- [✓] Limit Insetting
- [✓] Use Common Limit

Normals

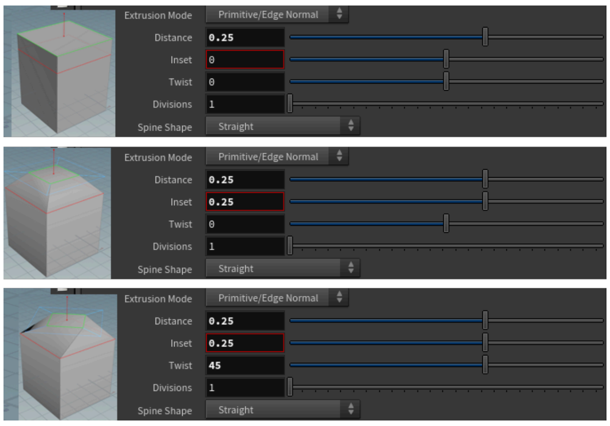
- [ ] Add Vertex Normals
- Cusp Angle: 60
- [✓] Cusp All Front Boundary Edges
- [✓] Cusp All Back Boundary Edges

Figure 5.16: The Poly Extrude Node parameter interface.

Firstly the tool can filter entities using a group name or by evaluating an expression. The extrusion can be based on the primitive or point normal. Next the distance, inset and twist parameters can be set. The distance determines the extrusion length, the inset determines the scale of the extruded surface relative to its progenitor and twist determines the relative rotation of the extruded surface. Figure 5.17 shows examples of manipulating these settings.

Subsequently we have options that can add a curvature to our extrusion. The Divisions parameter sets the amount of subdivisions to perform for the curvature, the Spine Shape parameter enable/disable curvature extrusion, and the Spine Control options fine tune the shape of the curve. An example of this can be seen in Figure 5.18.

Another useful feature is the Output Geometry and Groups options, that allow for the assignment of various parts of the extrusion to be assigned to a group. This can be useful when

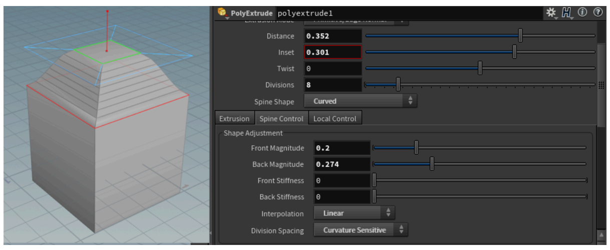


```
Extrusion Mode  Primitive/Edge Normal
Distance        0.25
Inset           0
Twist           0
Divisions       1
Spine Shape     Straight

Extrusion Mode  Primitive/Edge Normal
Distance        0.25
Inset           0.25
Twist           0
Divisions       1
Spine Shape     Straight

Extrusion Mode  Primitive/Edge Normal
Distance        0.25
Inset           0.25
Twist           45
Divisions       1
Spine Shape     Straight
```

Figure 5.17: Usage examples of the Poly Extrude Operator.



```
PolyExtrude polyextrude1
Distance        0.352
Inset           0.301
Twist           0
Divisions       8
Spine Shape     Curved
Extrusion | Spine Control | Local Control
Shape Adjustment
Front Magnitude   0.2
Back Magnitude    0.274
Front Stiffness   0
Back Stiffness    0
Interpolation     Linear
Division Spacing  Curvature Sensitive
```

Figure 5.18: Manipulating the curvature of the extrusion.

the user wants to later select the sides of the extrusion.

Edge Collapse

The edge collapse can merge a group of edges into a single point. The user can either specify each edge manually by using the point indices, or specify an edge group that contains those edges. Figure 5.19 and 5.20 show the Edge Collapse node in Houdini.

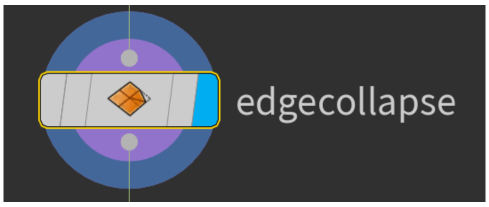


Figure 5.19: The Edge Collapse Node in the Houdini Software.

A usage example can be seen in Figure 5.21, where two edges of a box were merged to form a triangular prism.

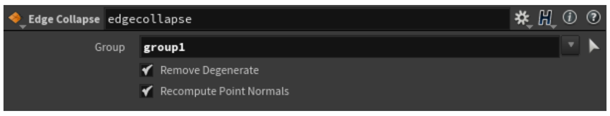

Edge Collapse edgecollapse
Group group1
✓ Remove Degenerate
✓ Recompute Point Normals

Figure 5.20: The Edge Collapse Node parameter interface.

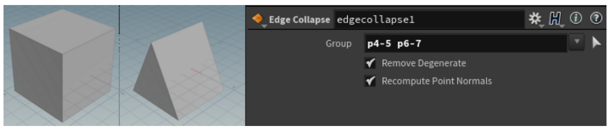

Edge Collapse edgecollapse1
Group p4-5 p6-7
✓ Remove Degenerate
✓ Recompute Point Normals

Figure 5.21: Usage example of Edge Collapse.

Partition Node

The Partition operator places entities into groups following a user-defined rule. This rule can make use of attribute values of the processed entity, or even houdini related parameters such as the current animation frame. Figure 5.22 show the Partition node in Houdini.

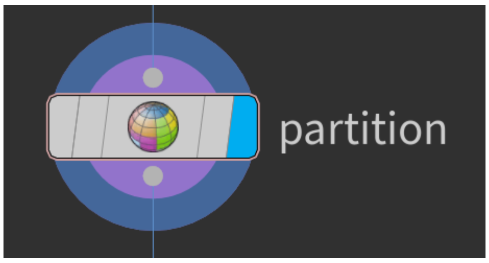
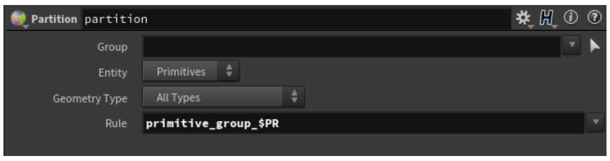


Figure 5.22: The Partition Node in the Houdini Software.

Figure 5.23 shows a screenshot of the node's parameters. As you can see the rule parameter in the example has the string ”primitive_group_$PR”. What this rule does it that it puts every primitive in a group named primitive_group with its primitive number at the end. This means that every primitive will be put in a unique group which can act as an identifier for that primitive. Note that even when the primitive gets subdivided into smaller pieces, all those pieces will still belong to the group of the original primitive. Figure 5.24 shows the groups created by executing that rule. There are 6 groups each corresponding to each primitive of the geometry which is a cube.

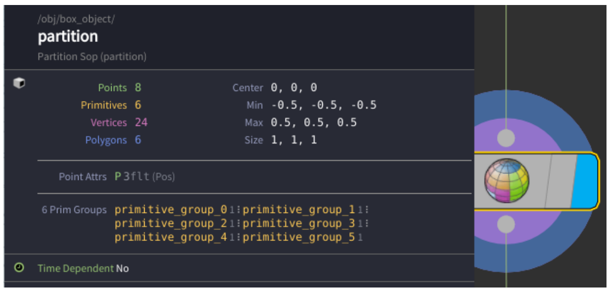
For Loop Subnetwork

An essential node in a typical Houdini workflow is the For Loop Subnetwork seen in Figure 5.25. Although there is a for loop block, we prefer the subnetwork for its simplicity and abstraction of the loop function.



```
Partition partition
Group
Entity           Primitives
Geometry Type    All Types
Rule             primitive_group_$PR
```

Figure 5.23: The Partition Node parameter interface.



```
/obj/box_object/
partition
Partition Sop (partition)

     Points  8            Center  0, 0, 0
 Primitives  6               Min  -0.5, -0.5, -0.5
   Vertices  24              Max  0.5, 0.5, 0.5
   Polygons  6              Size  1, 1, 1

Point Attrs  P 3flt (Pos)

6 Prim Groups  primitive_group_0 1 primitive_group_1 1
               primitive_group_2 1 primitive_group_3 1
               primitive_group_4 1 primitive_group_5 1

Time Dependent No
```

Figure 5.24: Geometry info of the Partition Node example.

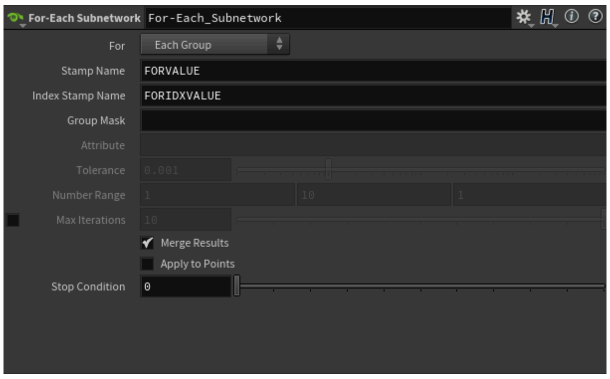
The for loop performs the functions of its subnetwork for each iteration. As seen in Figure 5.26 the user can iterate over each group of the input geometry (from the first input). Additionally the user may iterate over each primitive/point of the geometry or use a number index. If the for loop iterates over each group, the subnetwork will work on the primitives that are contained over the current group. Similarly if the for loop iterates over each primitive, the subnetwork of the loop will only be able to see one primitive at a time. However by using a number iterator, the entire model will be available for processesing. Note that the subnetwork can reference the current loop index (regardless of the selected scheme) by using what is called a stamp value.

Raycasting Node

Houdini offers an in-built solution for ray casting checks throught the Ray Operator (Figure 5.27).

It works differently than a conventional "check if ray hits" function. Instead the Ray tool projects the geometry's points on another geometry and if there is no geometry to project to (raycasting check fails) then the point stays on place. An example of this can be seen in Figure 5.28.In the first image we show how we have set our example scene: A fine grid mesh is positioned in front of an example mesh (a shader ball). In the second image we use the Ray node with a given ray direction to project the grid onto the target object. The results speak for themselves

Figure 5.25: Screenshot of the For-Each Subnetwork Node in Houdini.



For-Each Subnetwork For-Each_Subnetwork

For	Each Group
Stamp Name	FORVALUE
Index Stamp Name	FORIDXVALUE
Group Mask	
Attribute	
Tolerance	0.001
Number Range	1 10 1
Max Iterations	10
✓	Merge Results
	Apply to Points
Stop Condition	0

Figure 5.26: For-Each Subnetwork parameter interface.

Attribute Transfer

The Attribute Transfer node is used for transfering attributes from one mesh to another (Figure 5.29).

The first input of the node is the geometry we wish to transfer attributes to, and the second input is for the source geometry. The transfer method is based on spatial locallity. A point of the source geometry transfers its value to points of the destination mesh that are within a certain distance. We use this method to transfer information from the simulation model to our procedural model when we want to apply aging.

Python Operator

As we have mentioned previously, Houdini has the abillity to execute python code, and one way to do so is to use the Python Node (Figure 5.30).

The python node only contains a single parameter: a box where the user can type the python code as seen in Figure 5.31. In addition to using the Houdini Object Model libraries, the python script can import external scripts (from a special directory) and use their functions. Writing scripts externally with python IDEs is easier, but requires the user to restart the Houdini workspace for their compilation into ".pyc" files.

Figure 5.27: The Ray Node in Houdini.

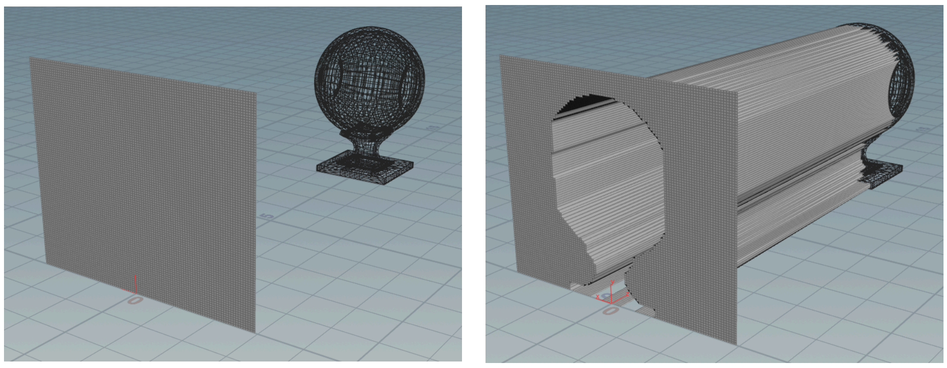


Figure 5.28: Projecting geometry with the Ray Node.

5.2 CGA Shape Grammar Toolset

In this section we will describe our implementation of several elementary CGA Shape Grammar tools in the Houdini Software. We have decided to create these tools ourselves, in order to have full control of their function and to facillitate their modification in the simulation pipeline.

5.2.1 Subdivide Scope

Subdivide Scope is a digital asset we created to simulate the CGA Shape Grammar splitting rule for scopes (Figure 5.32).

As you can see in Figure 5.33, the tool has all the necessary parameters to describe a basic split rule (aside from probabillistic selection). First the Input Filter is used to select the predecessor shape, the Axis determines the splitting axis, and for each subdivision the sizing, length and output symbol can be defined.

Figure 5.34a shows the subnetwork inside the Subdivide Scope Operator. First we use a Delete SOP to select the predecessor shape. We then perform some calculations for the subsection lengths and the splitting plane inside a Null node. Next we use a foreach subnetwork to iterate over each subdivision. At the end of the foreach node we receive the subdivided sections

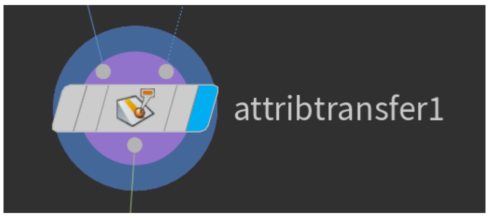


Figure 5.29: Screenshot of the Attribute Transfer node.

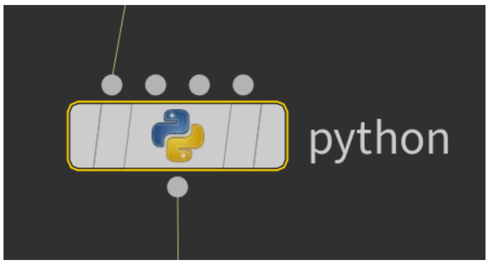


Figure 5.30: Screenshot of the Python Operator in Houdini.

which we merge with the geometry that did not belong to the predecessor symbol group.

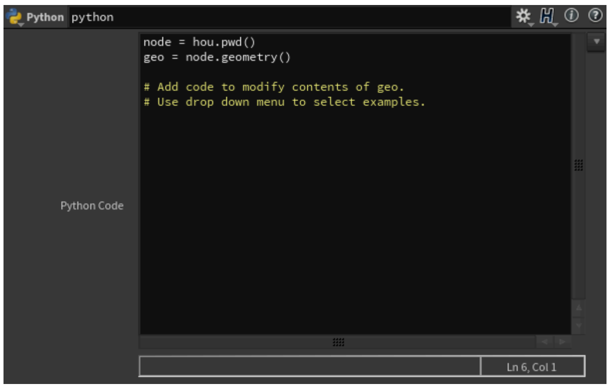
The foreach loop subnetwork is illustrated in Figure 5.34b. We start with another Null node making calculations relevant to the current subdivision. The clip section subnetwork defines two clipping planes that will cut our current subdivision, and finally we give our subsection its new symbol name and make sure it forgets its old symbol name (if needed).

5.2.2 Subdivide Facade

Subdivide Facade works similar to the Subdivide Scope tool, but works on facades. Whereas the previous splitting operation used axis aligned splitting planes to do the job, now the task of calculating the size of split, the splitting plane normal and splitting plane origin have become much harder due to the arbitrary orientation of the facade. The input parameters of Subdivide Facade remain exactly the same as the other tool, with the exception of having only two selectable axes: X and Y. Since we are working with facades and the splitting planes will follow the orientation of the facade, splitting on the Z-axis is entirely pointless.

Subdivide Facade is an implementation of the same split rule as Subdivide Scope, but is designed to work on arbitrary orientations of facades (Figure 5.35).

The parameter interface of Subdivide Facade is similar to Subdivide Scope. The only difference is that the user can only select between two splitting axes. Since facades are two dimensional entities, depth has no meaning. Figure 5.36 shows an example of use and the parameters used. In the example we only split a single face, and we do so vertically.



```
node = hou.pwd()
geo = node.geometry()

# Add code to modify contents of geo.
# Use drop down menu to select examples.
```

Figure 5.31: Python Operator parameter interface.

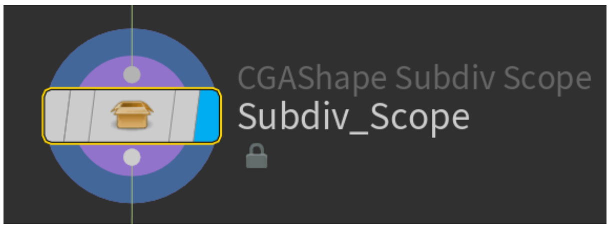
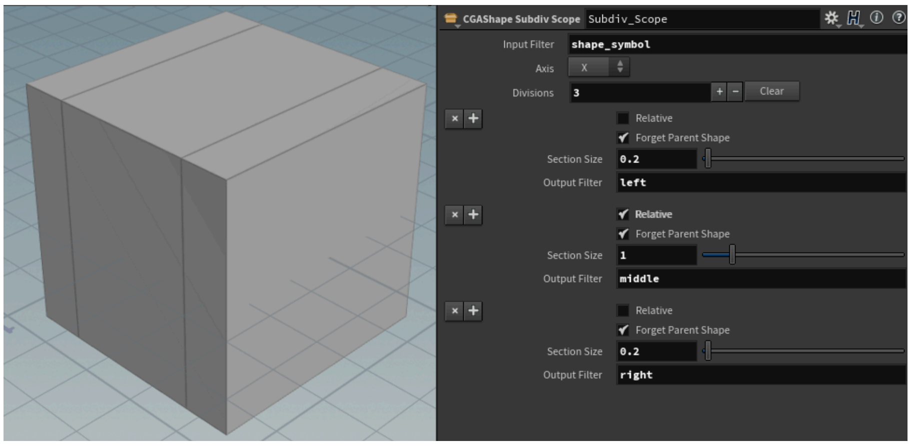


Figure 5.32: The Subdivide Scope Node.

Inside the node (Figure 5.37a) we select the primitives that belong to the predecessor symbol group. For each of the selected primitives, one by one we subdivide them inside the foreach_primitive subnetwork. Finally we merge the subdivided primitives with the rest of the geometry and output the results.

Inside the foreach_primtive subnetwork (Figure 5.37b) we do the following: First we move the primitive to the center of axes with move_to_origin (we use the centroid). Then we rotate the primitive using the vertical axes, so that it aligns with the normal of a reference primitive facing in the positive Z direction. Using another reference primitive pointing upwards we rotate the primitive so that its normal faces upwards. At this point the primitive is laying flat on the horizontal plane, with its center at (0,0,0). Next we use a Null node to perform calculations for the subdivisions, and perform the subdivisions in a foreach loop subnetwork. Lastly we use the previous versions of the primitive as references to take the primitive back to its original position and orientation.

The foreach_subdivision node is identical to the one in Subdivide Scope with a minor difference (Figure 5.37c). Due to implementation specific reasons we revert the vertical rotation of the primitive inside this subnetwork, once for each subdivision section.



CGAShape Subdiv Scope Subdiv_Scope

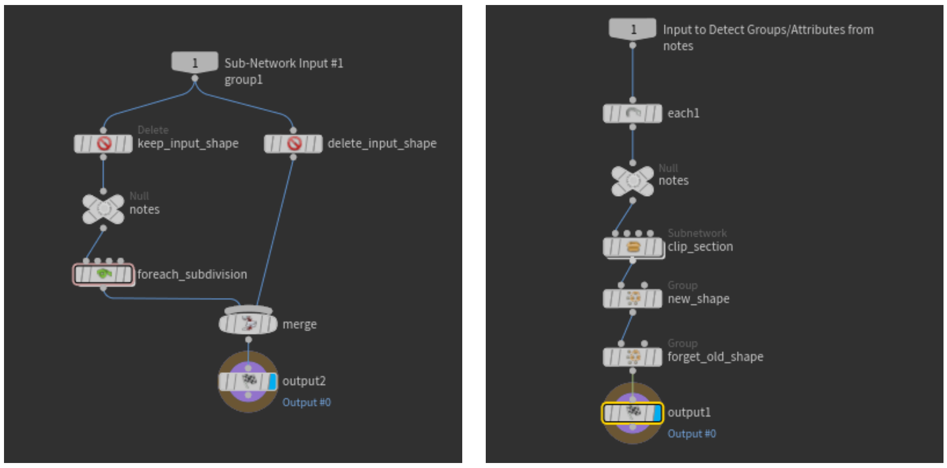
Input Filter shape_symbol
Axis X
Divisions 3

Relative
✓ Forget Parent Shape
Section Size 0.2
Output Filter left

✓ Relative
✓ Forget Parent Shape
Section Size 1
Output Filter middle

Relative
✓ Forget Parent Shape
Section Size 0.2
Output Filter right

Figure 5.33: Parameter Interface of Subdivide Scope and usage example.



(a) Subnetwork (b) ForLoop Subnetwork

Figure 5.34: The networks inside the Subdivide Scope Node.

5.2.3 Repeat

The Repeat Node is a custom made digital asset that simulates the Repeat rule of CGA Shape Grammar (Figure 5.38).

Just like in the production rule, the parameters let the user define the number of subdivisions (5.39a). The user may also define the size of each subdivision section (5.39b).

The implementation of the Repeat Operator is almost identical to Subdivide Facade, so we will focus on the differences. The first difference is that the group assignment happens at the highest possible level (Figure 5.40a). This is because all subsections will be assigned the same symbol name, so there is no need to do it individually. The second major difference is the way the calculations are being performed.

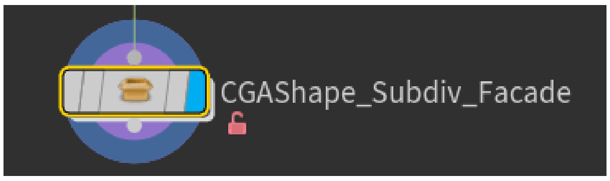
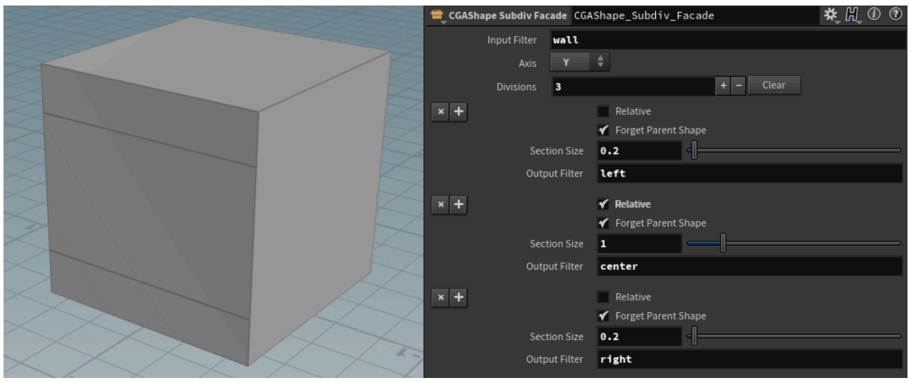


Figure 5.35: The Subdivide Facade Node.


CGAShape Subdiv Facade CGAShape_Subdiv_Facade

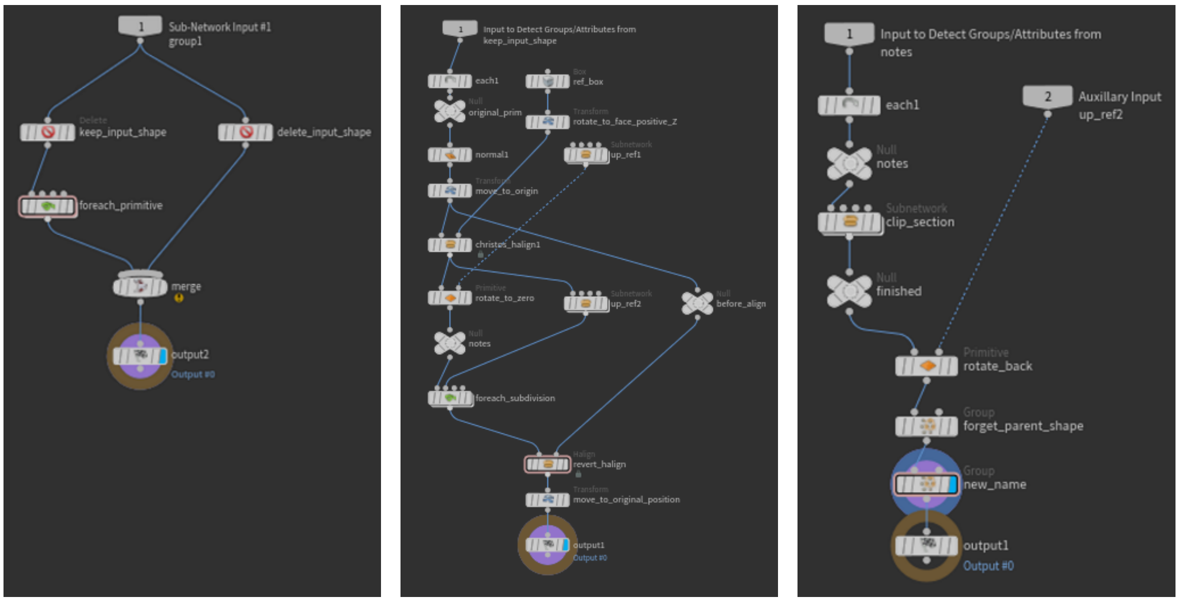
Input Filter wall
Axis Y
Divisions 3 Clear

[ ] Relative
[✓] Forget Parent Shape
Section Size 0.2
Output Filter left

[✓] Relative
[✓] Forget Parent Shape
Section Size 1
Output Filter center

[ ] Relative
[✓] Forget Parent Shape
Section Size 0.2
Output Filter right

Figure 5.36: Parameter Interface of Subdivide Facade and usage example.



(a) Subnetwork (b) ForEach Primitive (c) ForEach Subdivision

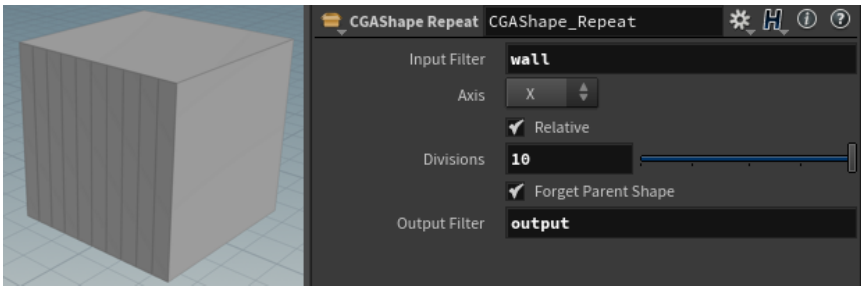
Figure 5.37: The networks inside the Subdivide Scope Node.

5.2.4 Insert

Finally we present our implementation of the Insertion rule. As you can see in Figure 5.41, the node accepts two inputs: the first input is the procedural geometry, and the second input is the insertion model.

Examples of usage can be seen in Figure 5.42. In the first example the roof of the house

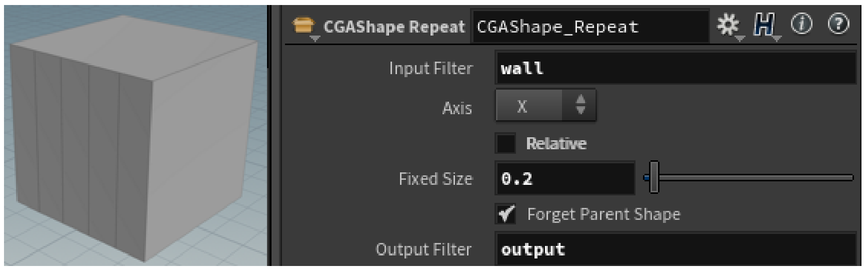
Figure 5.38: The Repeat Operator.



CGAShape Repeat CGAShape_Repeat

Input Filter: wall
Axis: X
✓ Relative
Divisions: 10
✓ Forget Parent Shape
Output Filter: output

(a) Defining number of subdivisions



CGAShape Repeat CGAShape_Repeat

Input Filter: wall
Axis: X
Relative
Fixed Size: 0.2
✓ Forget Parent Shape
Output Filter: output

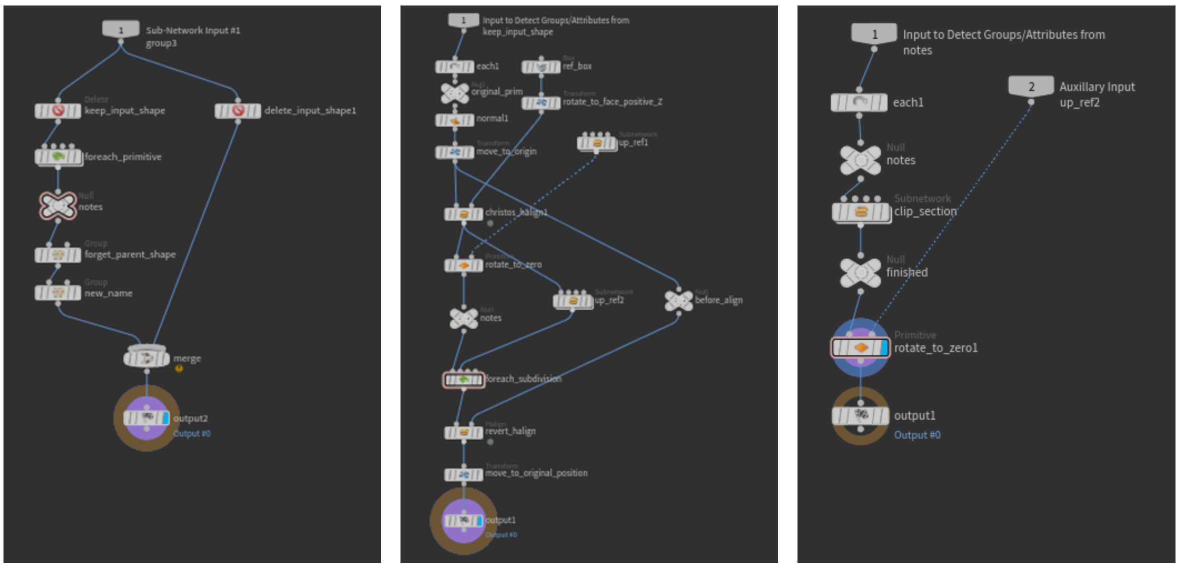
(b) Defining size of subdivision sections

Figure 5.39: Example of Repeat Operation.

was subdivided with Repeat operations to form tiles. By using the Insert operation we added roof tile models to create the results in the second picture. In the second example we insert a window model to a primitive that belongs to the group ”window”. The parameters are minimal; besides the first input which is the filter for the predecessor symbol, the other two are used for fine tuning the location and size of the input model.

Figure 5.43a illustrates the subnetwork of the Insert node. First we select the primitives that belong to the input filter, and for each one we insert the model from the second node input.

Inside the for loop node (Figure 5.43b) we move the primitive to the axes origin, and rotate it so that it lays flat on the horizontal plane. Then we place the insertion model at the same point and try to match its size. Since the primitive is flat we need to extrude it to form a bounding box. The length of the extrusion depends on a user defined parameter and it will affect the depth of the insertion model. Next we merge the primitive with the model and revert all rotations and translations using the original primitive as a reference. Since the model is merged with the



(a) Subnetwork (b) ForEach Primitive (c) ForEach Subdivision

Figure 5.40: The networks inside the Repeat Node.

Figure 5.41: The Repeat Operator.

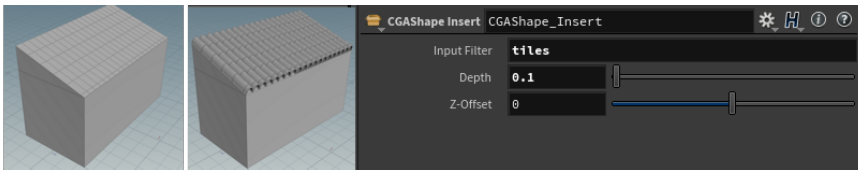
primitive it will move along with it. The last step is to remove the primitive since we don't need it anymore.

5.2.5 Pipeline

We will now demonstrate the use of our tools in a standard CGA Shape pipeline. For each stage of the procedural pipeline we will use a simple house as an example. The house in question will look like the one in figure 5.44.

Mass Modeling

We begin by creating our base scope using a Box node (Figure 5.45). Then we extrude the ceiling surface with a small inset using Poly Extrude and then Edge Collapse the right set of edges to get a pointy roof as seen in Figure 5.46.

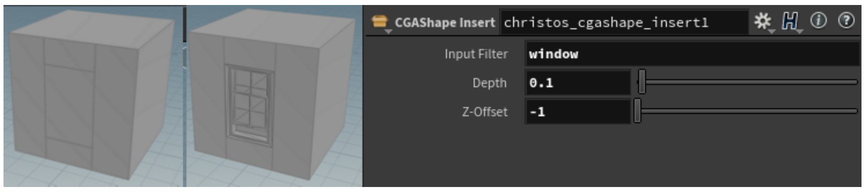

CGAShape Insert CGAShape_Insert

Input Filter tiles

Depth 0.1

Z-Offset 0

(a) Inserting roof tiles


CGAShape Insert christos_cgashape_insert1

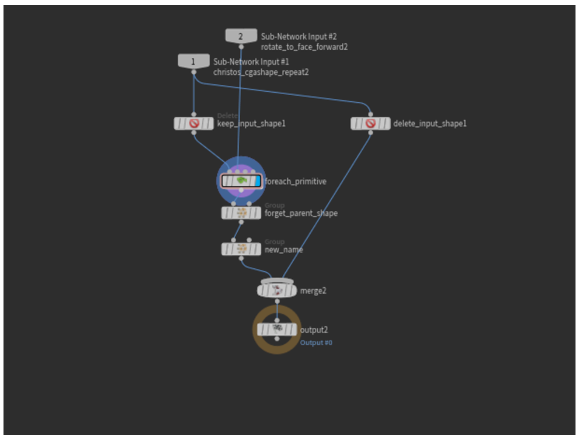
Input Filter window

Depth 0.1

Z-Offset -1

(b) Inserting a window

Figure 5.42: Example of Insert Operation.



(a) Subnetwork

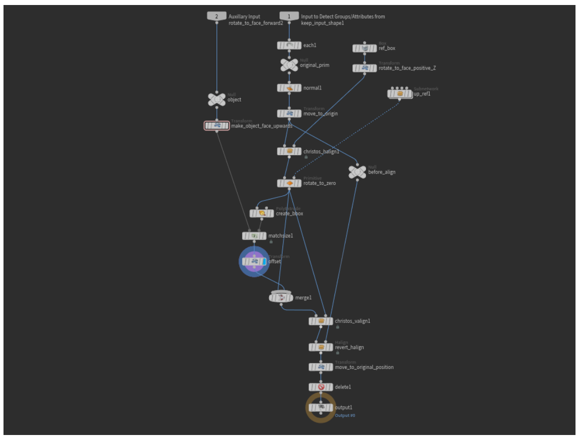

(b) ForEach Primitive

Figure 5.43: The networks inside the Insert Node.

Component Split

As opposed to the original paper we have not implemented a Comp tool to extract facades. This is because our splitting tools extract facades by themselves with the ForLoop Subnetwork. What we will do however in this stage is give appropriate names to each facade. By using the Group node we select primitives according to their normal and name them. For the wall facing the positive Z direction we named it “north wall”, while the wall facing the positive X direction we named “west wall”. We also grouped the primitives facing upwards into the “roof” group. We then grouped the west, east, and south walls into the “window wall” group. Similarly we grouped the north wall on the “door wall” group. These names will help us decide later on where to place windows and where to place doors. Figure 5.47 illustrates the different groups assigned to the building.

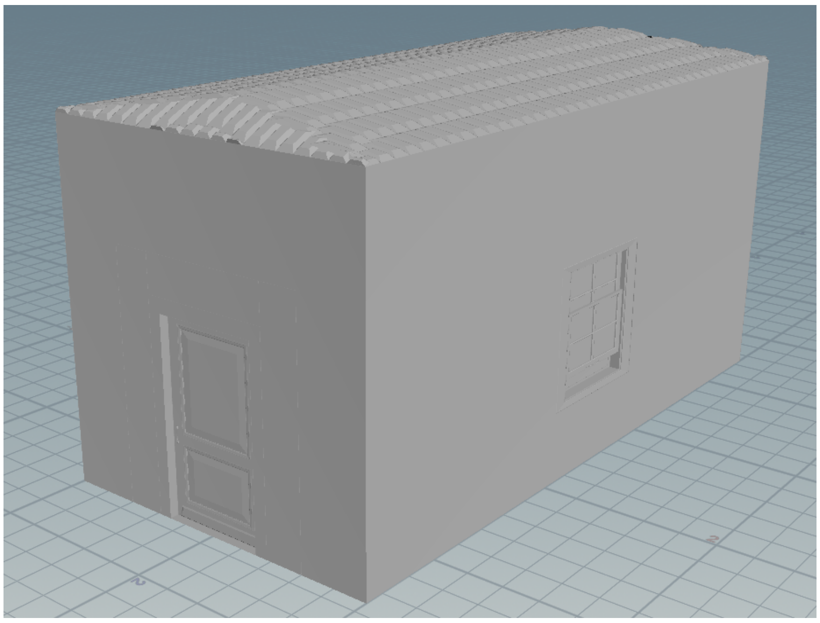
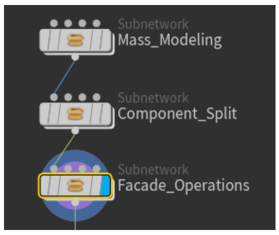



Figure 5.44: The final procedural model on the left, and the houdini network on the right.

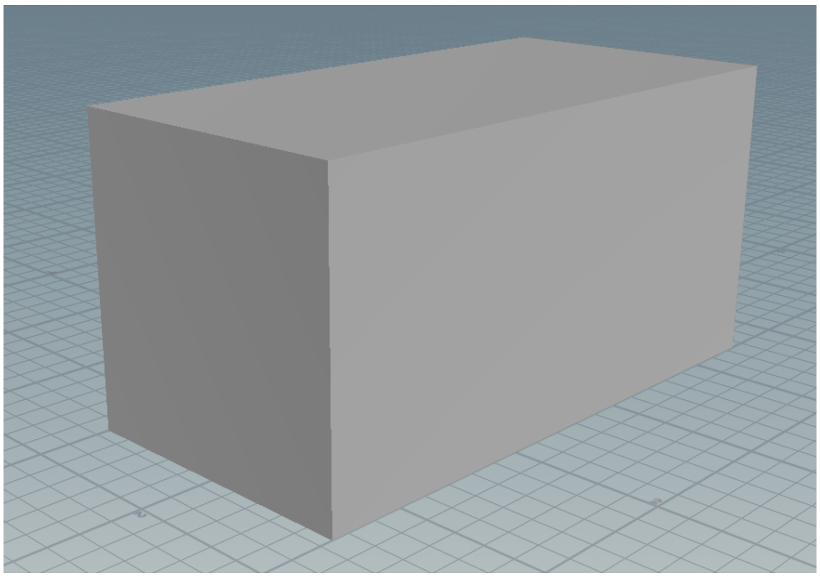
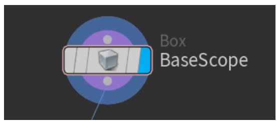



Figure 5.45: The initial box shape.

Facade Operations

Now it is time to work on the facades. First we subdivide the walls that belong to the “window wall” group. We use two Subdivide Facade nodes: the first one splits with the X splitting axis and creates 3 subsections; two subsections have relative sizing and the middle one has a fixed size of 0.8. We name the middle section “window wall” again and forget the original shape. The second subdivision is done on the new “window wall” and on the Y splitting axis. Like before we subdivide the wall into 3 subsections, the middle section having fixed size 0.8. To

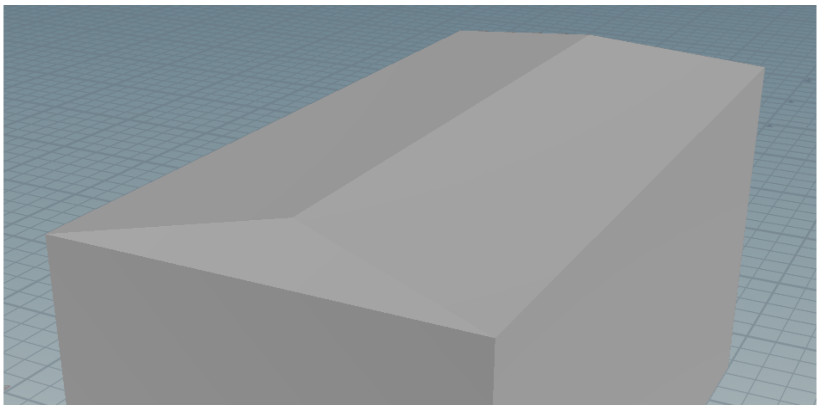
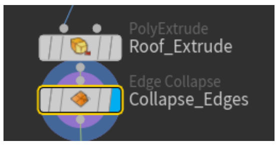
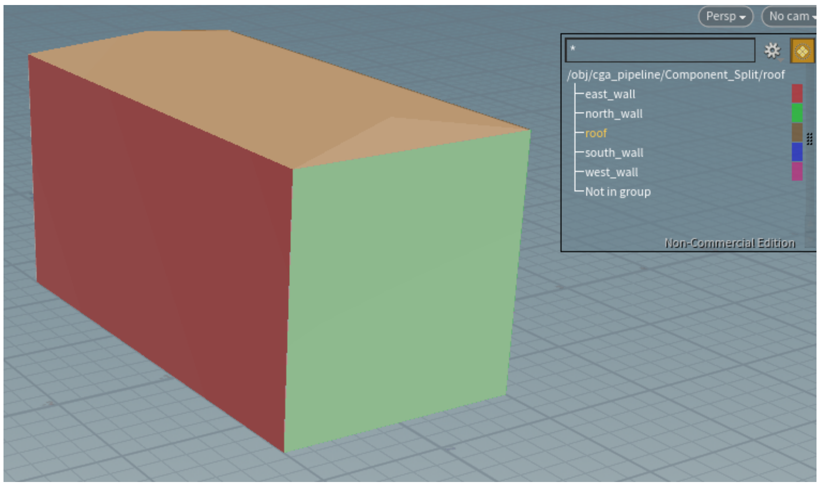



Figure 5.46: Extruding the roof model.



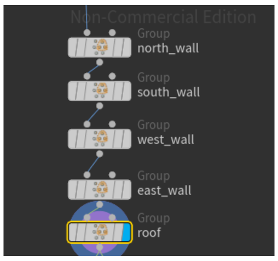


Figure 5.47: Grouping each facade of the model appropriately

lower the windows I have given the upper relative subsection a multiplier of two. The results of the above description are shown in Figure 5.48.

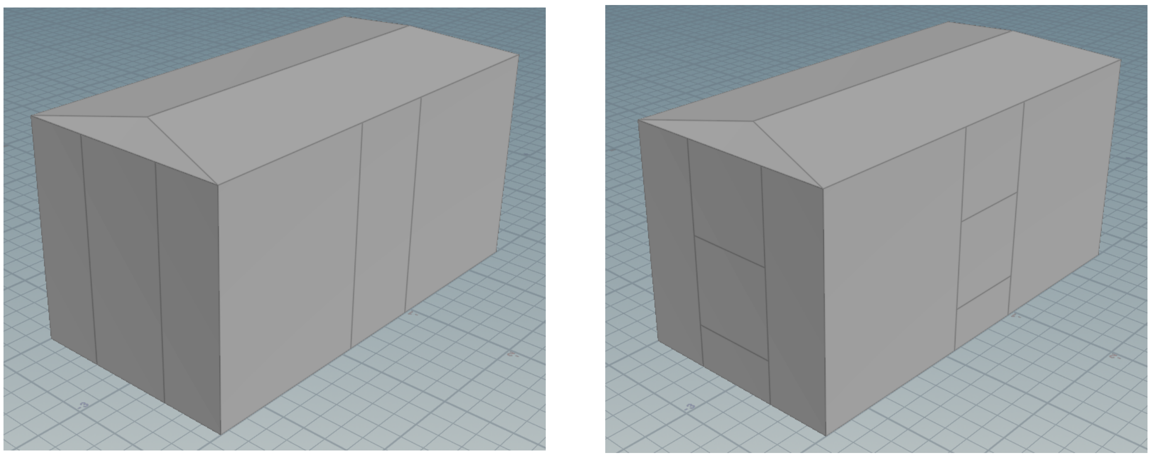


Figure 5.48: Splitting the facades to form square primitives for the windows

Next we will subdivide the “door wall”. We will once again use two Split Facade operations in a similar manner as before. We will split 3 sections in the X-axis with the middle section having size 1.2, and then split in the Y-axis with 2 sections the fixed section having size 1.4. See Figure 5.49.

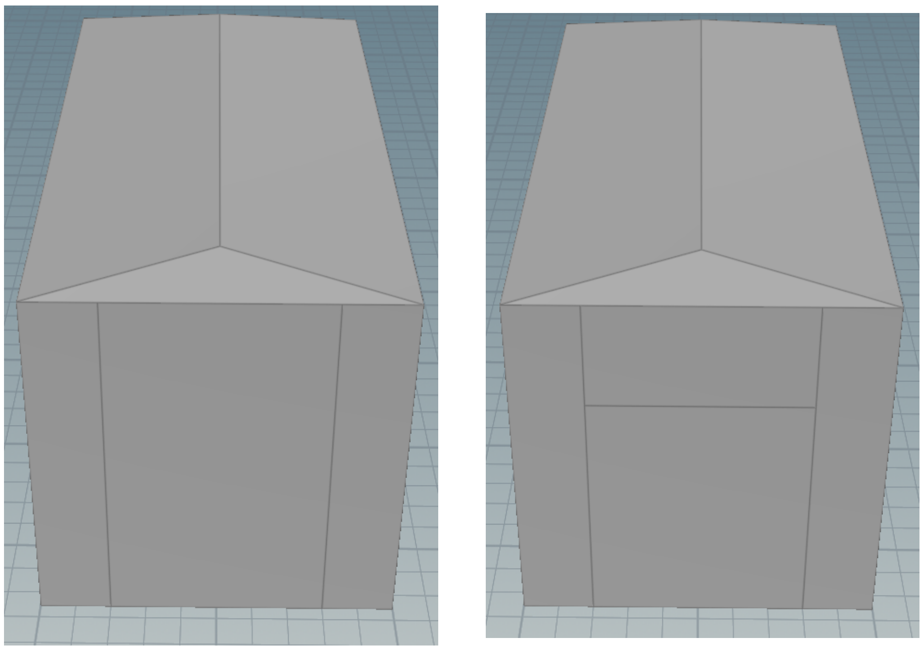


Figure 5.49: Splitting the door wall so that the a door seam is formed.

Then we will need to cut the roof into tiles. We will do this with two Repeat operations: the first will cut each roof surface into vertical strips of size 0.1. The second Repeat will cut each vertical strip horizontally into tiles of length 0.26. See Figure 5.50.

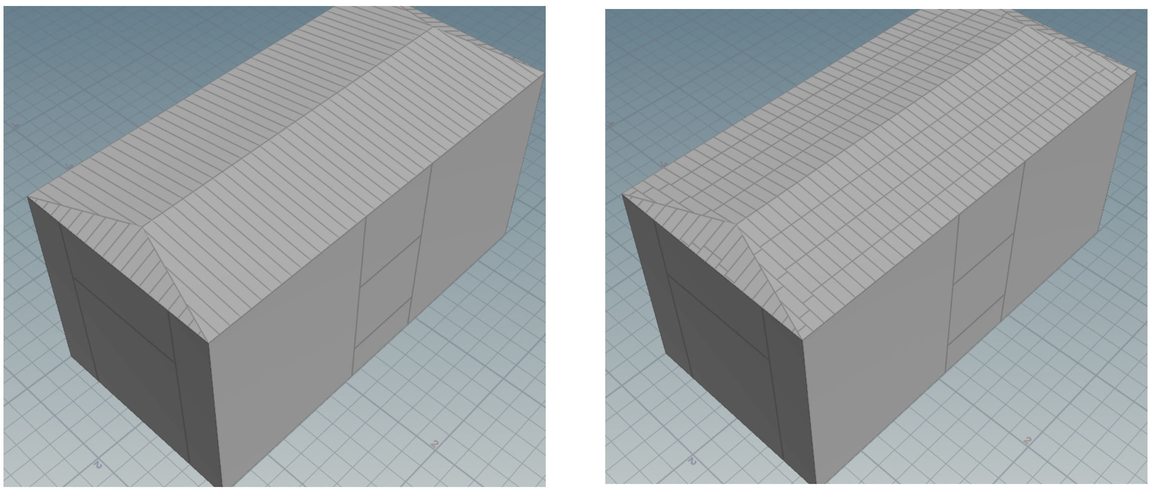


Figure 5.50: Using the repeat split to form the seams for roof tile placement

Apply Geometry

Finally it is time to use the Insert node to place the windows, doors and tiles. We import the geometries of those models from file and then insert them to the primitives that belong to the group's "door", "wall", and "tile" respectively. You can see the results in Figure 5.51.

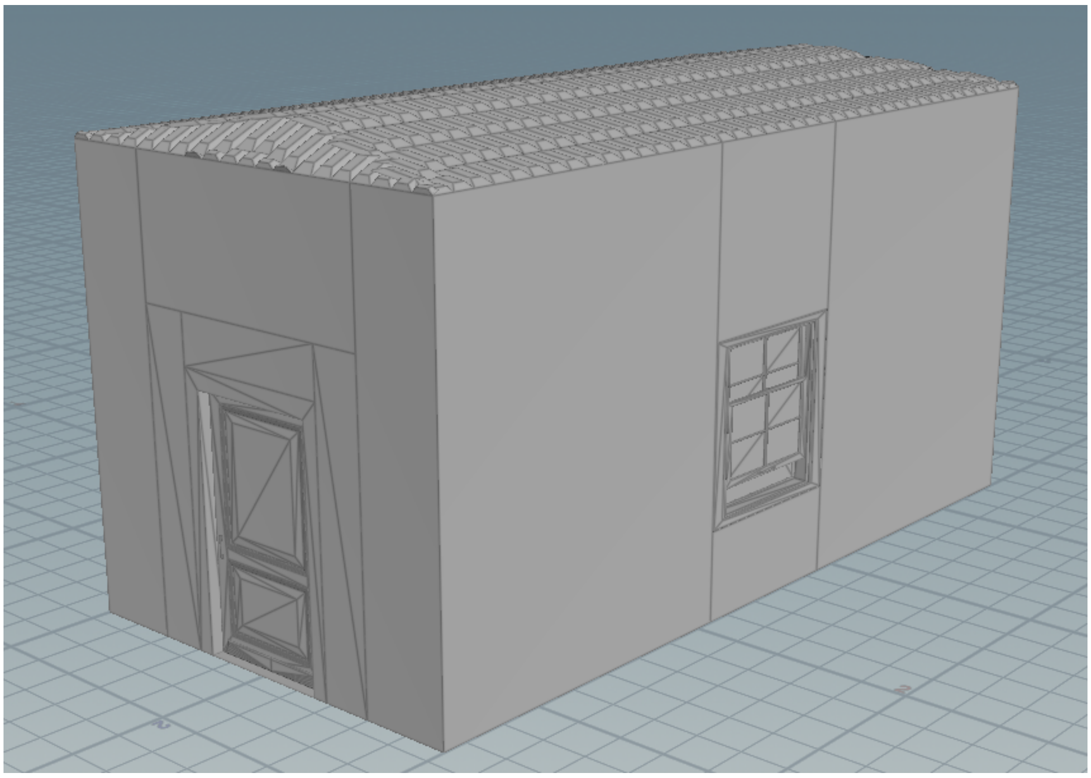


Figure 5.51: Finally we insert the door, window and roof tile models.

5.3 Aging Simulation Tools

In this section we will present the basic functionality of all the tools we use for the simulation pipeline.

5.3.1 Simulation

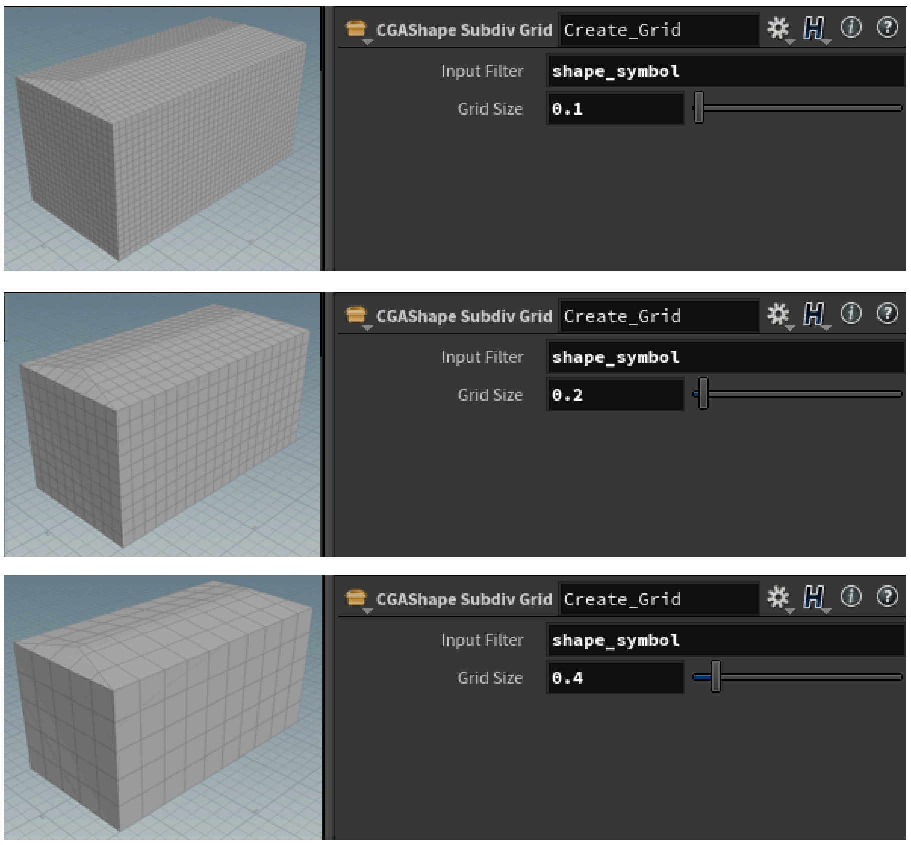
Subdivide Grid

In order to store the attributes on arbitrary points of the building, we subdivide it into a uniform grid using an implementation of Create Grid we named Subdivide Grid (5.52).

The tools takes only two parameters. First it needs an input filter, to only process the predecessor shape, and it also needs the size of the uniform grid. Examples with different

Figure 5.52: The Subivide Grid Node.

grid sizes can be seen in Figure 5.57.



Create_Grid
Input Filter shape_symbol
Grid Size 0.1

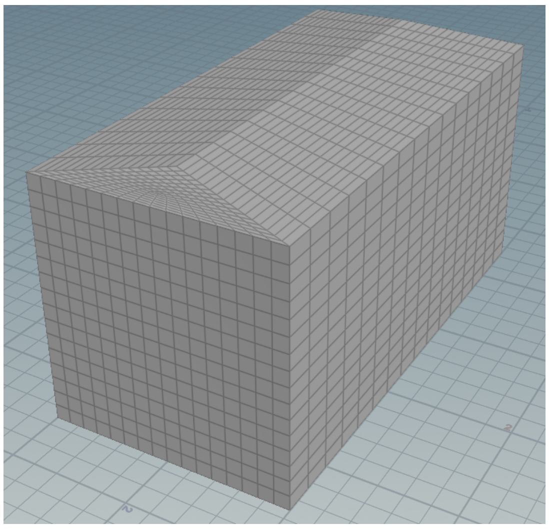
Create_Grid
Input Filter shape_symbol
Grid Size 0.2

Create_Grid
Input Filter shape_symbol
Grid Size 0.4

Figure 5.53: Example of using Subdivide Grid.

We have refrained from using the in-built subdivision method (with Houdini's Subdivide node) because it does not subdivide the mesh into uniform quads. Our tool on the other hand ensures that all "grid cells" are uniform and connected. See Figure 5.54 below for a visual comparison.

As for the implementation specifics, Create Grid makes use of a modified Repeat operation called Repeat Scope. This modified split operation works like the Repeat operation but the subdivision axes are orthogonal (X,Y and Z) and works on the entire 3D model. First the user defines the grid size as a parameter to Create Grid. Then we perform 3 consecutive Repeat


(a) Using in-built Subdivision node

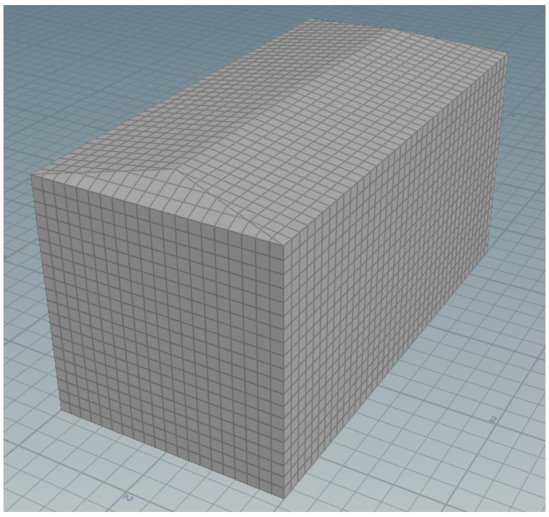

(b) Using our custom tool

Figure 5.54: Using the repeat split to form the seams for roof tile placement

Scope, one for each axis. First we repeat splits along the X-axis with the given spacing. Next we split along the Z-axis. However the two repeats will only form vertical strips on vertically aligned walls, so we need to repeat split on the Y-axis. Before we do that we must temporarily remove sloped and horizontally aligned surfaces (roof surface, floors, ceilings etc.) to avoid creating "double splits". Now the model has multiple points on its surface that can store all of the simulation related attributes. The Houdini network of this tool can be seen in Figure 5.55.

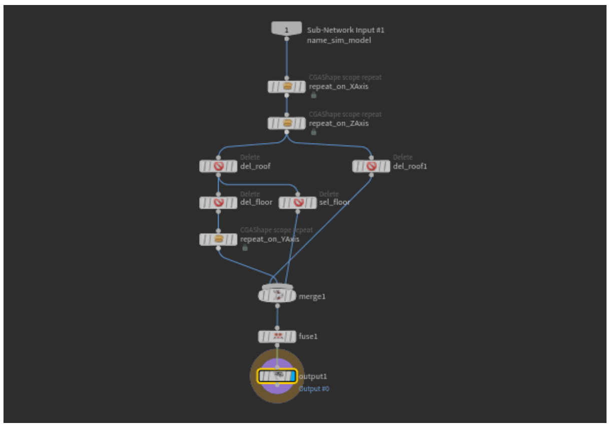


Figure 5.55: The networks inside the Subdivide Grid Node.

Transfer Support

As we have mentioned in the Simulation Chapter, we need to transfer semantic information from the procedurally generated building to the "grid surface" model. To do that we have created the

Transfer Support tool, which transfers shielding and support attribute values. (Figure 5.56).

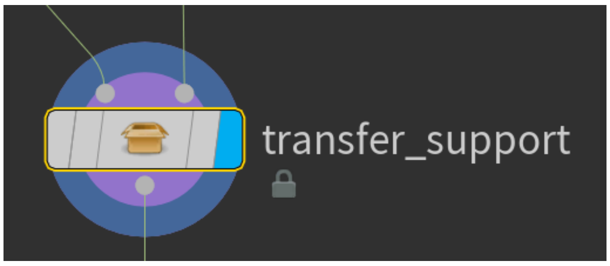
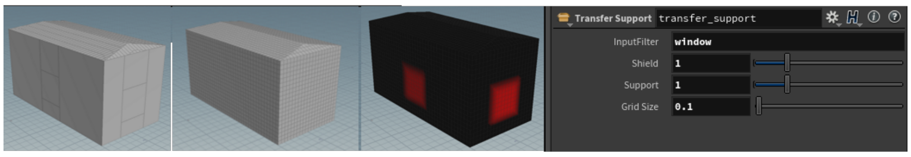


Figure 5.56: The Transfer Support Node.

In example Figure 5.57 you can see the tool in action. The node takes two inputs: It takes the grid model we wish to transfer to, and the procedural model. In the same figure you can see the parameters in use. As with any other tool so far we need an input filter, but it is used slightly different in this case. Instead of selecting the primitives that we will process and refine, we instead pick the primitives from the procedural model that we will transfer. In the example we want to transfer window information to the grid model. The next two parameters control the value of the shield and support attributes at the areas of transfer, and finally the grid size attribute is used to properly match the transfer bounding boxes with the grid of the grid model. The first image shows the procedural model, the second shows the grid model and the third picture shows the transfer taking place (we visualize the shield value on the model).

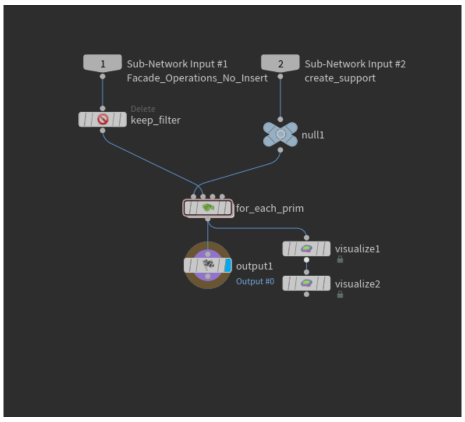

Transfer Support transfer_support

InputFilter	window
Shield	1
Support	1
Grid Size	0.1

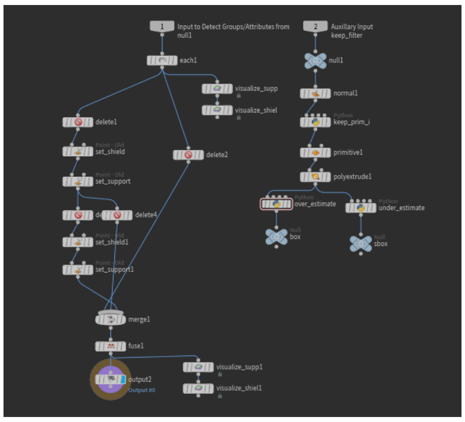
Figure 5.57: Example of using Trasfer Support.

The implementation of the tool is simple in principle. The idea is to pick all the primitives from the procedural model, form a bounding box around them and use it to the select points from the grid model, which we will then set their attribute values corresponding to the parameters. To do this we start with a simple deletion filter to select the right procedural model primitives (See Figure 5.58a). Then we use a for loop network that iterates on each of the primitives, the transfering operation taking place inside. If you are careful you might notice that something is off with the for loop node. That is because the first input of the node is the grid model and not the selected procedural model primitives. That contradicts the notion that we iterate on each of the procedural model primitives. While we don't actually iterate on geometry, what we do is that we iterate as many times as the number of filtered procedural primitives using a number index. We select the appropriate primitive inside the for loop by using the iteration index (or stamp value). This was done due to implementation specific reasons, mainly because other iteration schemes failed to keep the attribute transfers on the grid model from previous loop iterations.

Inside the for loop we have two pathways (See Figure 5.58b). The left path way takes the grid model and by using the calculated bounding box named ”box” we select the contained points of the grid model (using a delete node) and set their attribute value. Subsequently we use a under-estimate bounding box named ”sbox” and repeat the same process. The under-estimate transfers the user given value, while the ’over-estimate’ box transfers a fraction of the user given value. The right path way uses the for loop index to select a procedural model primitive and through some python code we create the two bounding boxes ”box” and ”sbox”. We round up the size of the bounding box by using the given grid size.



(a) Subnetwork



(b) ForLoop Subnetwork

Figure 5.58: The networks inside the Transfer Support Node.

Ruin Shield

During each step of the simulation we destroy random parts of the shielding using the Ruin Shield tool (Figure 5.59). This tool plays a major role in the process ”Stochastic Destruction Seed” in the Simulation Component Diagram (Figure 5.75).

The tool has two parameters (please ignore the input filter): the probabillity of shield degradation striking a point, and the amount of damage that will be dealt. Figure 5.60 illustrates two examples of use, where the model is subject to shield degradation for each frame. We took 3 snapshots at time frames 30,60 and 90 for both cases. In the upper case the probabillity of

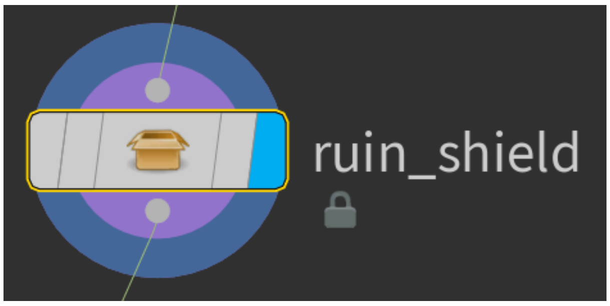
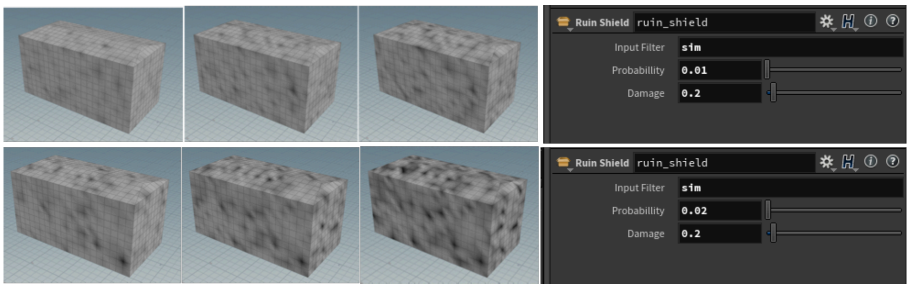


Figure 5.59: The Ruin Shield Node.

degradation is lower and as you can see this affects the speed at which the shield layer gets ruined.

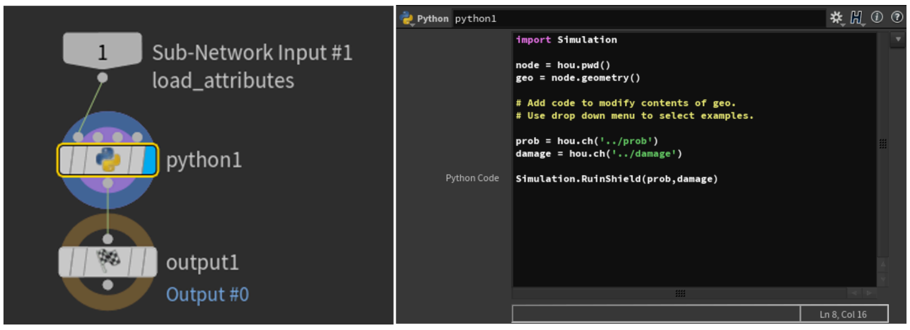


Ruin Shield ruin_shield
Input Filter sim
Probability 0.01
Damage 0.2

Ruin Shield ruin_shield
Input Filter sim
Probability 0.02
Damage 0.2

Figure 5.60: Example of using Ruin Shield.

Implementation wise, the tool is very simple. The node itself comprises of a single python SOP as seen in Figure 5.61. The code makes a call to an external function ”RuinShield” in our Simulation script. The function basically iterates through each point of the geometry, and by using a pseudo random number generator it decides whether or not to inflict damage.



```
import Simulation

node = hou.pwd()
geo = node.geometry()

# Add code to modify contents of geo.
# Use drop down menu to select examples.

prob = hou.ch('../prob')
damage = hou.ch('../damage')

Simulation.RuinShield(prob,damage)
```

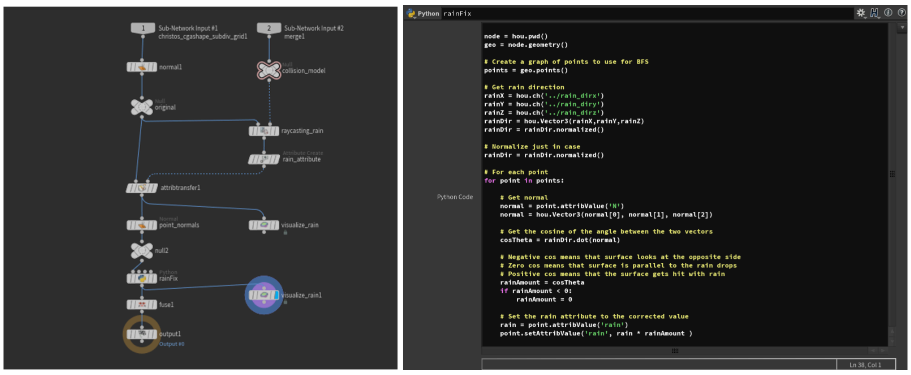
Figure 5.61: Network and python code of Ruin Shield.

Rain Casting

Rain Casting is the implementation of the rain simulation component in the Simulation Chapter (5.62).

Figure 5.62: The Rain Casting Node.

The basic idea is that we project our grid model to a collision model of the world using Houdini's in-built Ray node. Then by using a copy of the grid model, but with all of its rain attibute values equal to 1, we transfer its attributes to the ray cast model. As a result: the points of the grid model that failed the ray cast will remain in place and receive rain value of 1. In the collision model of the world we include the building itself to take self-occlusions into consideration, but it must be shrinked to avoid floating point precision problems. See Figure 5.63 for an illustration of the subnetwork.



```python
node = hou.pwd()
geo = node.geometry()

# Create a graph of points to use for BFS
points = geo.points()

# Get rain direction
rainX = hou.ch('../rain_dirx')
rainY = hou.ch('../rain_diry')
rainZ = hou.ch('../rain_dirz')
rainDir = hou.Vector3(rainX,rainY,rainZ)
rainDir = rainDir.normalized()

# Normalize just in case
rainDir = rainDir.normalized()

# For each point
for point in points:

    # Get normal
    normal = point.attribValue('N')
    normal = hou.Vector3(normal[0], normal[1], normal[2])

    # Get the cosine of the angle between the two vectors
    cosTheta = rainDir.dot(normal)

    # Negative cos means that surface looks at the opposite side
    # Zero cos means that surface is parallel to the rain drops
    # Positive cos means that the surface gets hit with rain
    rainAmount = cosTheta
    if rainAmount < 0:
        rainAmount = 0

    # Set the rain attribute to the corrected value
    rain = point.attribValue('rain')
    point.setAttribValue('rain', rain * rainAmount )
```

Figure 5.63: The subnetwork and python code for Rain Cast.

Another problem that this tool has to face is to correct false positives. Such false positives may include ceiling surfaces that should not receive any rain but do so anyway due to projecting to a nearby surface, thus receiving rain attribute of value one. To fix this we run a simple Python script that checks if the dot product of the rain direction and the normal is greater than 0 and if that is the case we know that that point cannot receive any rain. We can also calculate the amount of rainfall that should fall depending on the slope angle relative to the rain direction. The code of the Rain Fix script can be seen on the right image of Figure 5.63. Furthermore there is a visual example in Figure 5.64.

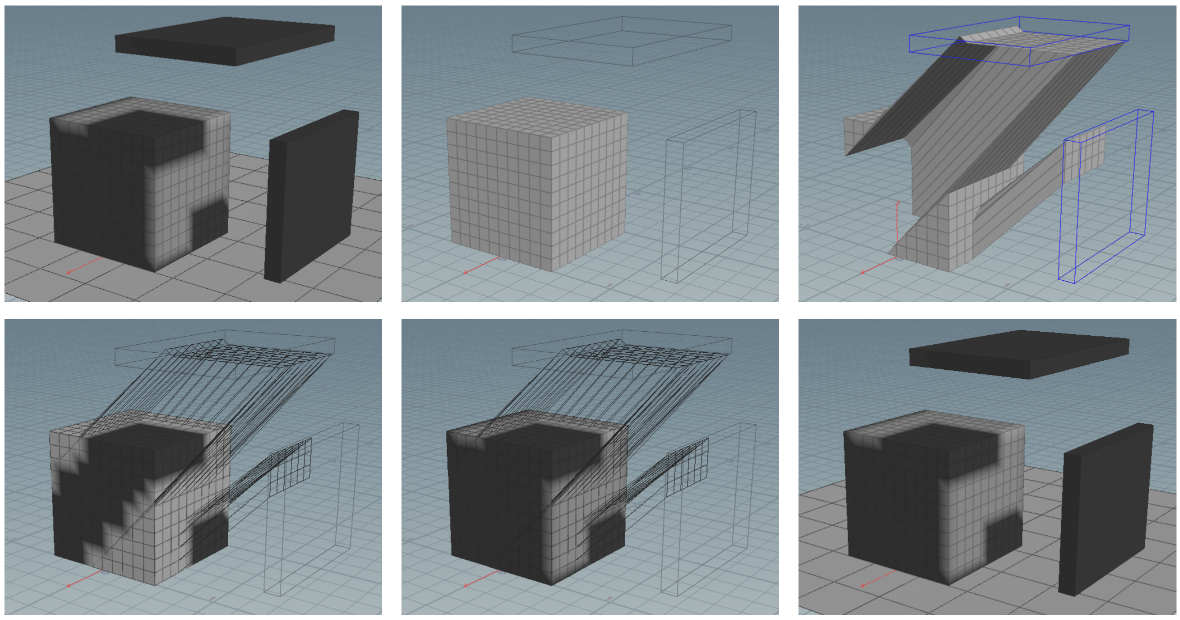


Figure 5.64: An illustration of how ray casting is used to calculate ’rain occlusion’.

Cascade

The entirety of the Rain Cascade simulation component (Figure 5.65) was implemented in Python scripts. Since the script is very long, we will not include it in the thesis. We will however explain the basic idea and the algorithms that we use to achieve the desired results.

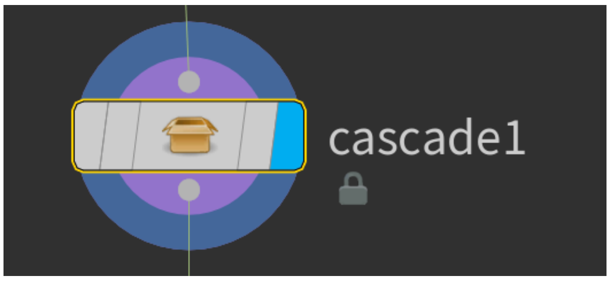


Figure 5.65: The Cascade Node.

We start by creating a vertex connectivity graph from the mesh of the “grid surface” model. The reason we need to construct this data structure is because Houdini does not store the neighbours of each point. Instead one needs to find the edges that the point belongs to and from there discover its neighbor. The connectivity graph is an array of neighbor lists, where we can find the neighbors of each point quickly by using its id. Then we create an ordered set of all the points of our “grid surface” model and iterate on each point until the set is exhausted. For each iteration of the loop we find the highest point of our mesh that exists in the point set and has rain falling on it. If no such point exists, then we end our loop abruptly. For optimization reasons we remove points from point set that we come across that have no rainfall, or have no shielding as lack of shielding means absorption of water.

Once the highest point has been picked we use it as an origin point to propagate the rain water to lower points. Specifically we find all the direct and indirect neighbors that are on the same height level as our origin point and add them as additional origin points. All origin points are added to a queue, that holds the points along with the humidity that they transfer and start a BFS like algorithm to propagate the humidity downwards. We use queue aggregation to minimize the overhead of multipath propagation. The aggregation method involves finding points in the queue that have the same destination and merge their humidity quantities. For every point that we propagate we add the humidity from the source with the rainfall of the current point and then move on. The algorithm stops of course when the queue is exhausted. Additionally we remove all the points from the point set that we visited during this phase of the algorithm.

After the BFS propagation is done we return to our first while loop which iterates until the point set is empty. Then we find the next highest point and repeat the entire process again and again.

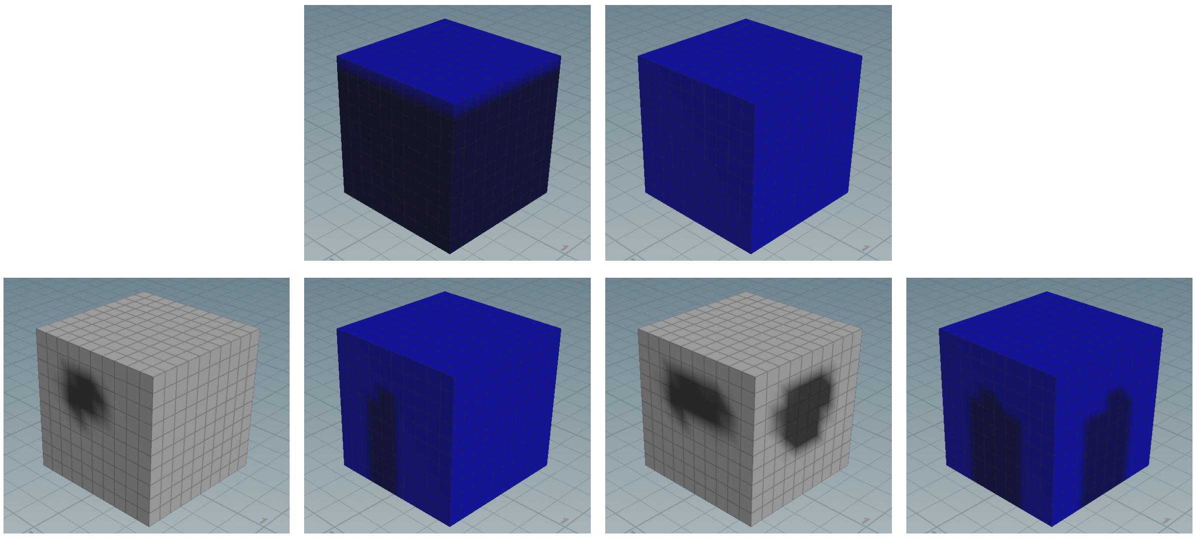


Figure 5.66: Here is the Cascade node in action.

An example can be seen in Figure 5.66, where we show how the existence of water with a bright blue color. In the first row of the Figure, we show how the rain water that falls on top of a object (first image) can flow down the sides (second image). The second row illustrates how holes, illustrated as black spots in the gray scale images, effect the flow of water. As you can see the existence of holes interrupts the flow of water. This feature was mainly introduced to account for the dynamic nature of the building: during the aging simulation parts of the building get ruined, creating holes that cut off the flow of water.

Corrode

Corrode works with a simple python script that iterates over each point of the "grid surface" model. For each point calculate the amount of damage caused by humidity using a formula. This formula basically factors in the damage protection from the shield and support layers as well as the amount of humidity. Depending on the resolution of the "grid surface" model we use a multiplier to increase the damage, as a larger grid cell represents more points of the surface thus gets damaged less.

Collapse

When a point of the building collapses, it gradually pulls all of the neighboring points along with it. By using Collapse we create an area of influence that affects the surrounding points of a destroyed point. To do this we use a specialized python script that finds the points all of the destroyed points, and for each one a circular BFS is used to influence direct and indirect neighbors.

5.3.2 Aging Application

We will now present the implementations of our Aging Application components.

Select Geometry

The separation of the 'types' of geometry is done with a simple delete node (Figure 5.67). We enter the symbol names for all the groups that should be aged a particular way.

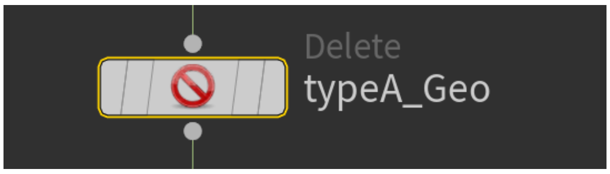


Figure 5.67: Selecting the type of geometry with a Delete node.

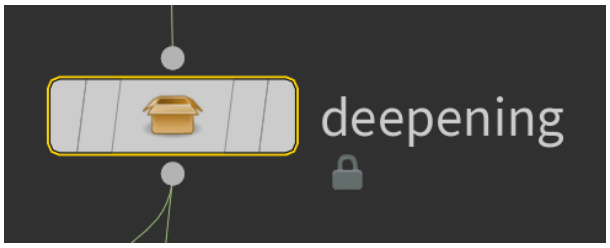
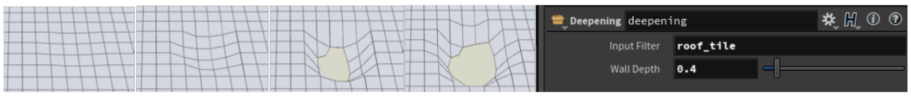


Figure 5.68: The Deepening node.

Deepening

Deepening is our implementation (Figure 5.68) of the first aging technique we have mentioned in the Simulation Chapter.

An example of this tool can be seen in Figure 5.69, where a mesh gets deeper every simulation frame until a hole opens.



Deepening deepening
Input Filter roof_tile
Wall Depth 0.4

Figure 5.69: Usage example of the Deepening node.

The subnetwork hiding under the node is show in Figure 5.70. First we select the input filter primitives. After we calculate the point normals we use a python script to perform deepening. This involves moving the points along their normal, but in the opposite direction, multiplied by the amount of aging and the wall depth. Then we use another python script to remove primitives that have all of their points completely destroyed. Next we use an anti-aliasing script, which moves every point by the average of its ”alive neighbor” vectors. These vectors are the difference between the point in question and its neighbors that are not yet destroyed. Points that have no destroyed neighbors will stay in place (assuming that our primitives are quads) while the rest will tend to move away from the hole.

Peeling

We now move to the implementation of the Peeling component (Figure 5.71).

The parameters of the nodes include an input filter, tolerance values for the dissapearance of the outer and inner layer and the depth of the inner layer. An example can be seen in Figure 5.72. The first image shows a visualization of the Composite attribute value (the amount of aging), and the second image shows the effects of peeling. We should point out that the peeling tool itself does not texture the layers.

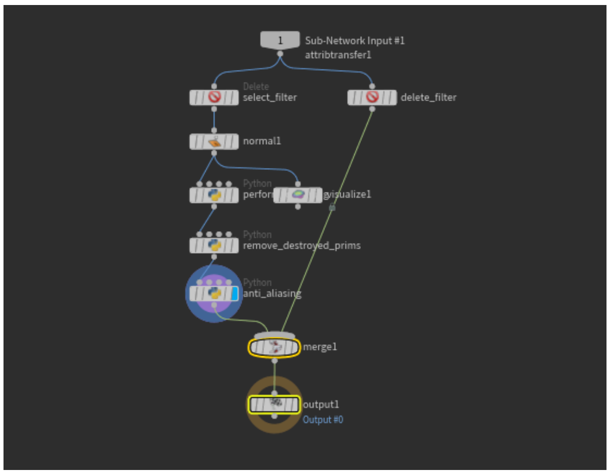


Figure 5.70: Network of Deepening.

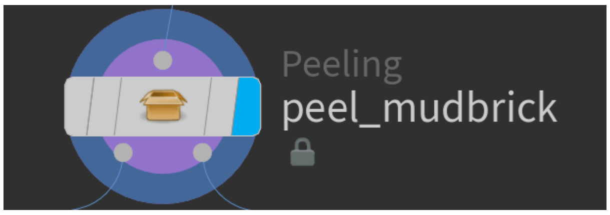


Figure 5.71: The Peeling node.

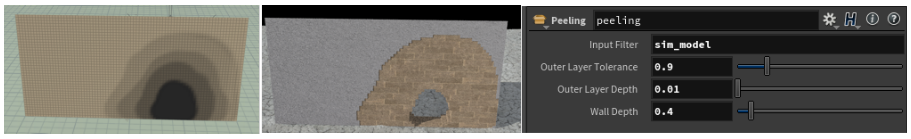
Implementation wise, we first filter the geometry to get the right primitives. Then we create a new attribute for primitives named 'aging'. The value of the attributes are equal to the average value of the aging attribute of each point. This is done in a python script. Subsequently we use a delete node to select the primitives that belong to the inner layer, based on their aging value. We extrude those primitives by the depth of the outer layer. We repeat this process to open a hole. See Figure 5.73

NoInsert

We have two implementations of this node. The first version deletes the primitive if even one of its points are fully destroyed. The second version deletes the primitive only if all of its points are fully destroyed. We have found that the first implementation suits objects like support beams, while the other suits objects like windows and doors.

5.3.3 Pipeline

Now that we have presented all of the tools and techniques used in the simulation and aging process, we will briefly present an implementation of our simulation pipeline. First take a


Peeling peeling

Input Filter sim_model

Outer Layer Tolerance 0.9

Outer Layer Depth 0.01

Wall Depth 0.4

Figure 5.72: Usage example of the Peeling node.

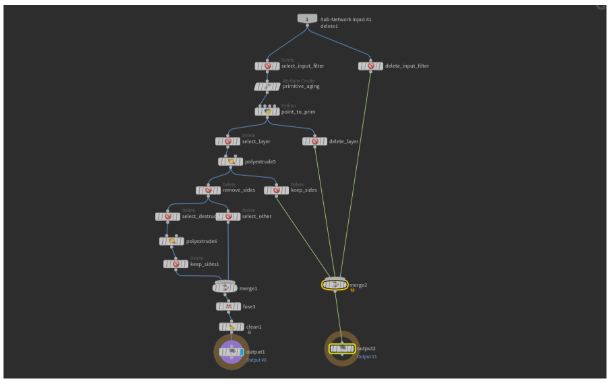


Figure 5.73: Subnetwork in Peeling node.

look at Figure 5.74. The network illustrates an implementation of the Simulation Pipeline we presented in the Simulation Chapter. First we create a mass model, then we split its components and then split its facades. Next we use the procedural model as input to the Simulation process. Finally we apply weathering effects on the procedural model by using the simulation model as a reference.

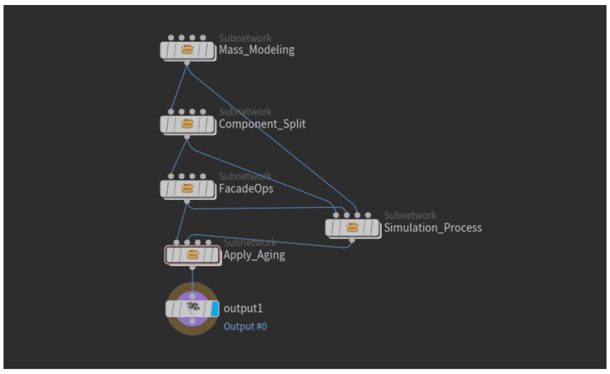


Figure 5.74: Overview of all pipeline stages.

The inside of the simulation process can be seen in Figure 5.75. First we subdivide our original model to form a grid model, and create the simulation attributes. We then transfer semantics from the procedural model and finally run the simulation. The second input of the simulation component is a model that will be used for ray cast collisions.

Figure 5.76 shows an example of what goes on in the Transfer Semantics Component. As expected we make use of the Transfer Support node to transfer semantics from the procedural

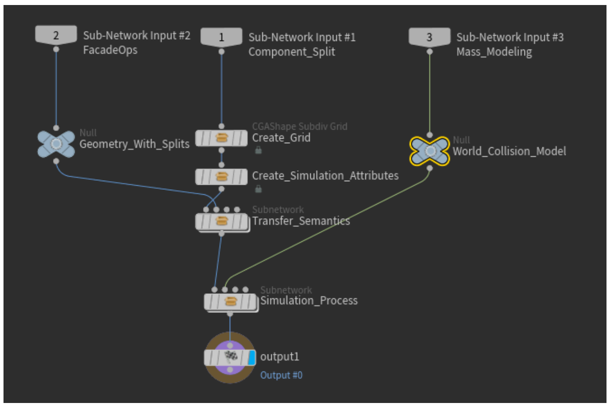


Figure 5.75: Simulation Process subnetwork.

model to our grid model. We use one node for each different component: windows, doors, gates, beams, etc.

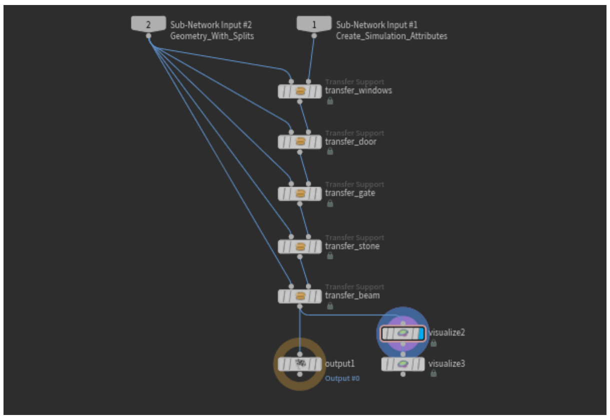


Figure 5.76: Inside the Transfer Semantics component.

Next we will dwelve inside the simulation component. In Figure 5.77 we illustrate the subnetwork inside the simulation component. The first node is used to initialize the simulation model that will be used to store attributes from the previous simulation frames. The next node contains all of the simulation related nodes, such as Rain Casting, Cascade and Collapse (See Figure 5.78).

The Apply Aging component contains three subnetworks, each dedicated to a different type of weathering effect (Figure 5.79). The type A is for surfaces that need to be deepened, type B is for peeling, and type C is for the no-insert effect.

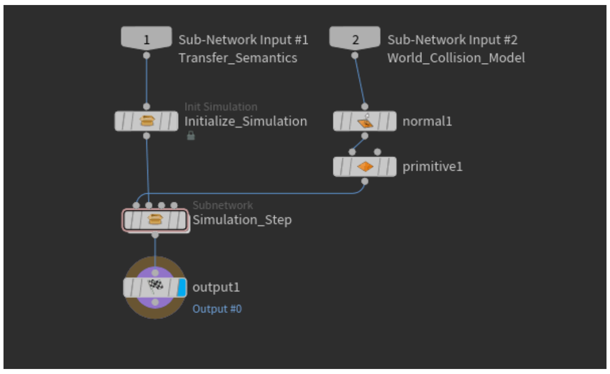


Figure 5.77: The simulation component.

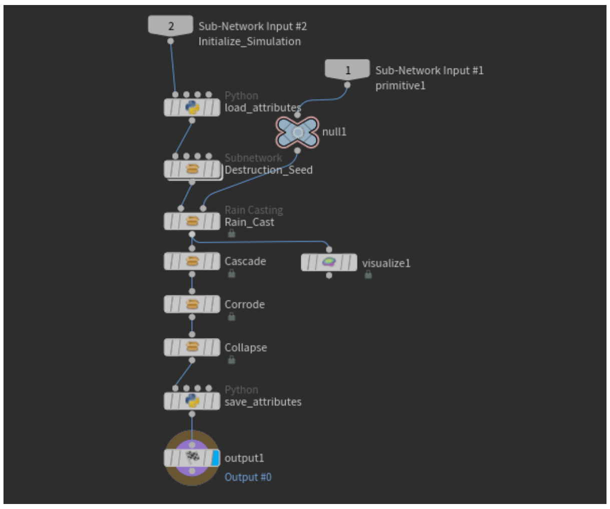


Figure 5.78: Subnetwork of the simulation step.

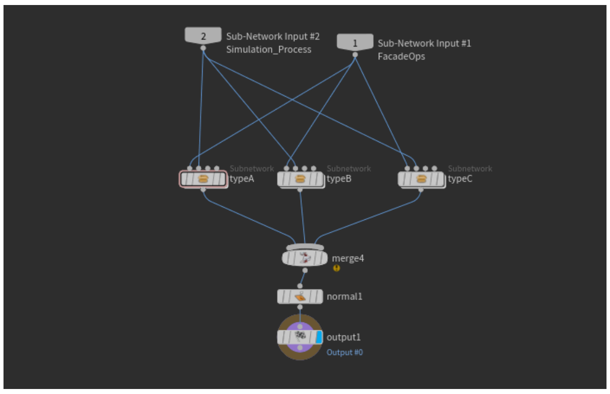


Figure 5.79: Inside the Apply Aging component.

Chapter 6

Results

Contents

6.1 Procedural House

Using the CGA Shape grammar tools that we implemented we have created a traditional Cypriot mudbrick house (Figure 6.1).

The images of Figure 6.2 show the different stages of the development of the building, starting from the Mass Modeling stage, then the Facade Split Operations (the shader is used to show the seams) and finally the Application of Geometry.

6.2 General Degradation

Here we will demonstrate general degradation of the mudbrick house we built. To speed up the simulation we have created several points of weakness as seen in the grayscale image of Figure 6.3. We run the simulation for 5 frames before we show the next render, so that the changes are clearer. Notice how the degradation spreads to neighbouring points. All three aging application techniques can be seen. Deepening is used for the roof tiles, NoInsert is used

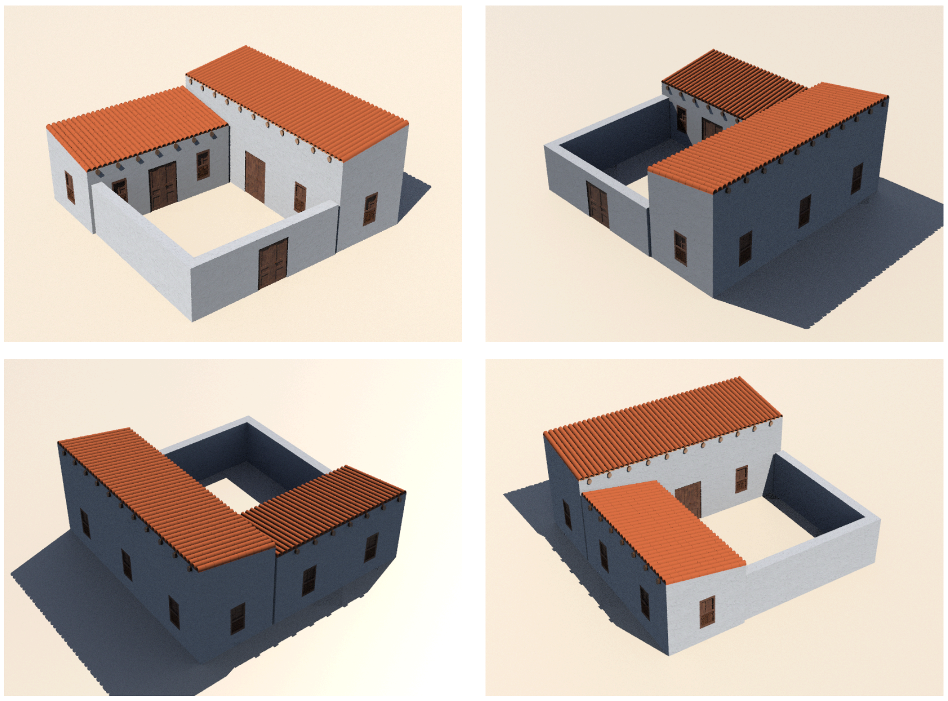


Figure 6.1: Renders of the procedural house.

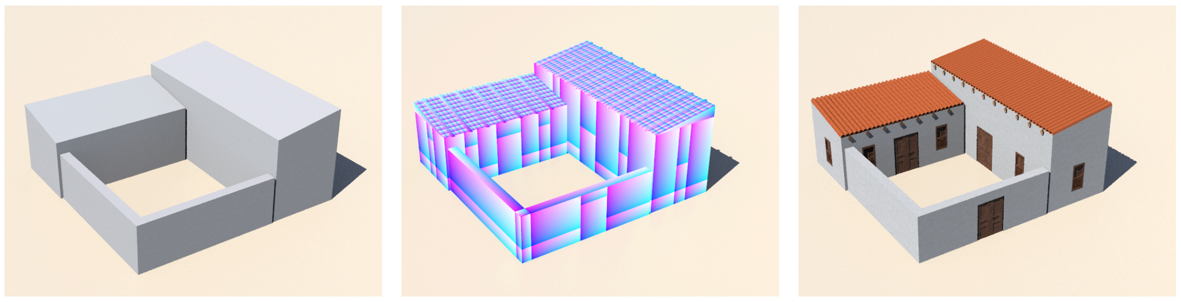


Figure 6.2: CGA Shape Grammar procedure.

to remove the wooden beams when there is no support, and Peeling is used to peel layers of the mudbrick wall and stone wall.

6.3 Humidity Destruction

Now we will demonstrate the effects of humidity from rain on the rate of deterioration of a surface. For this example we have constructed a simple box shaped building with a single slope roof and stripped away part of the protective layer of the roof to allow for rain to flood into the sublayer. We created two cases: one where the building is completely unprotected, and the second one has a tall brick wall protecting it from the rain. For reference the rain is falling from the right side of the building. Observe how in the bottom row pictures (no wall protection) the

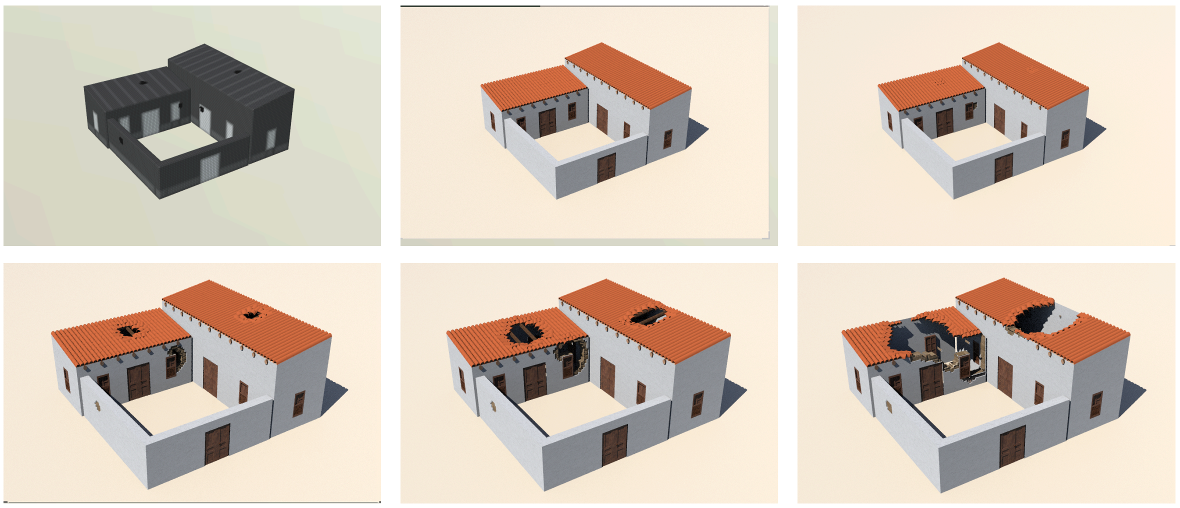


Figure 6.3: Degradation of the mudbrick house.

roof collapses in a few simulation frames. See figure 6.4 for a visual comparison.

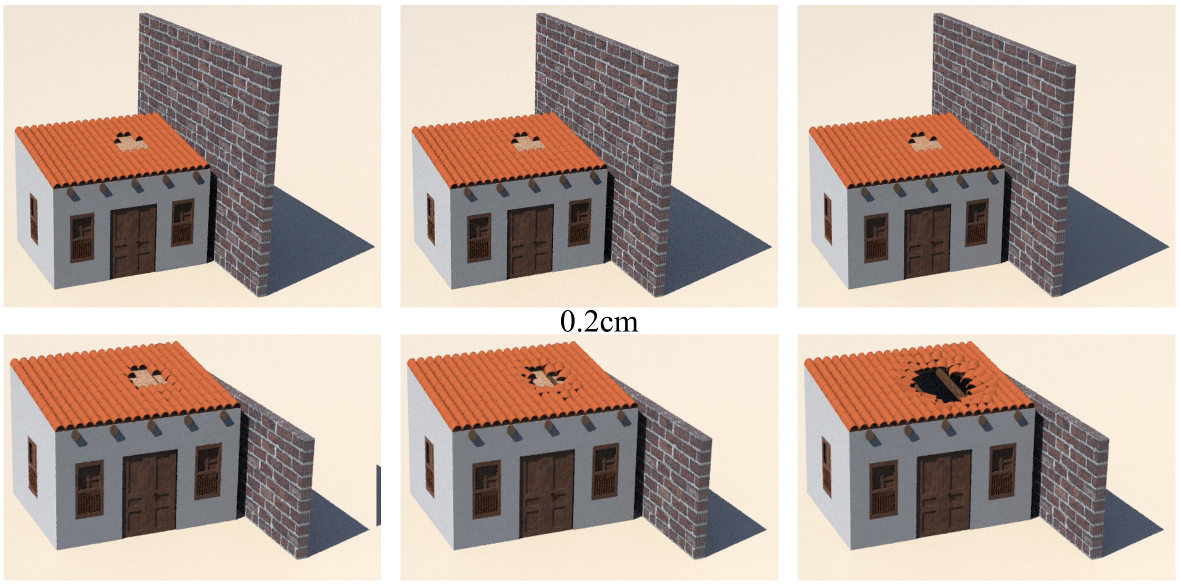


Figure 6.4: Effects of rain on building degradation.

6.4 Support Resistance

In the next set of images we will demonstrate how a roof collapses with and without the existence of underlying support structures. Figure 6.5 shows the gradual degradation of a roof without any supporting beams. As you can see, there seems to be no limit on how much the hole spreads. The frames were simulated at intervals of 10 frames.

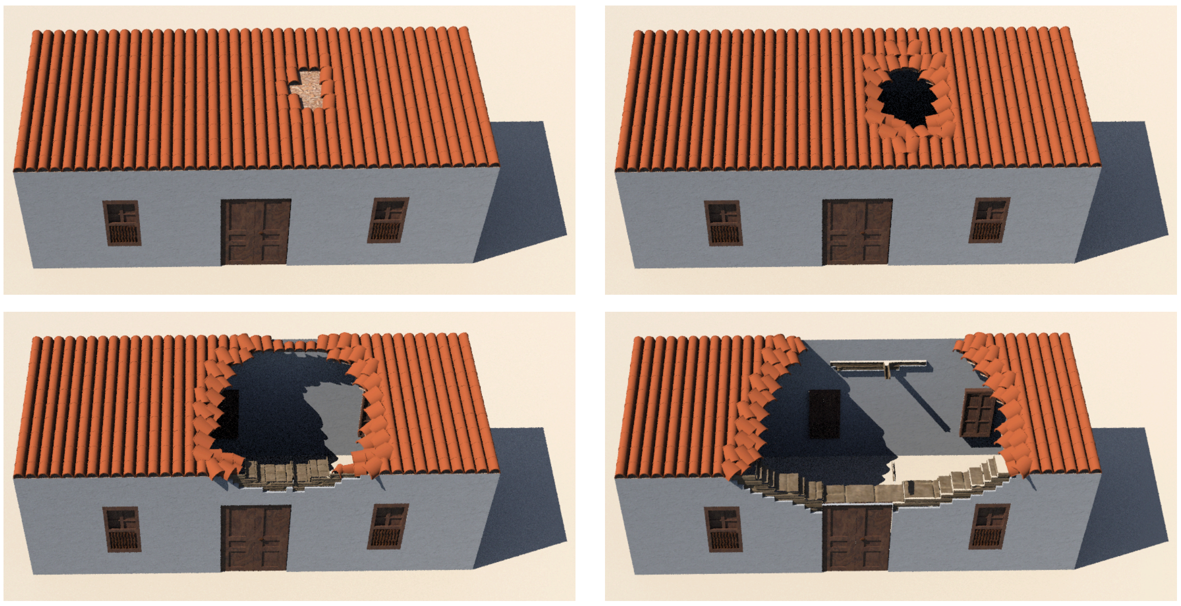


Figure 6.5: Degradation of roof without supporting beams.

In Figure 6.6 we illustrate the same test but with a more sturdy roof. It is apparent that with the existence of supporting structures, the degradation rate decellerates.

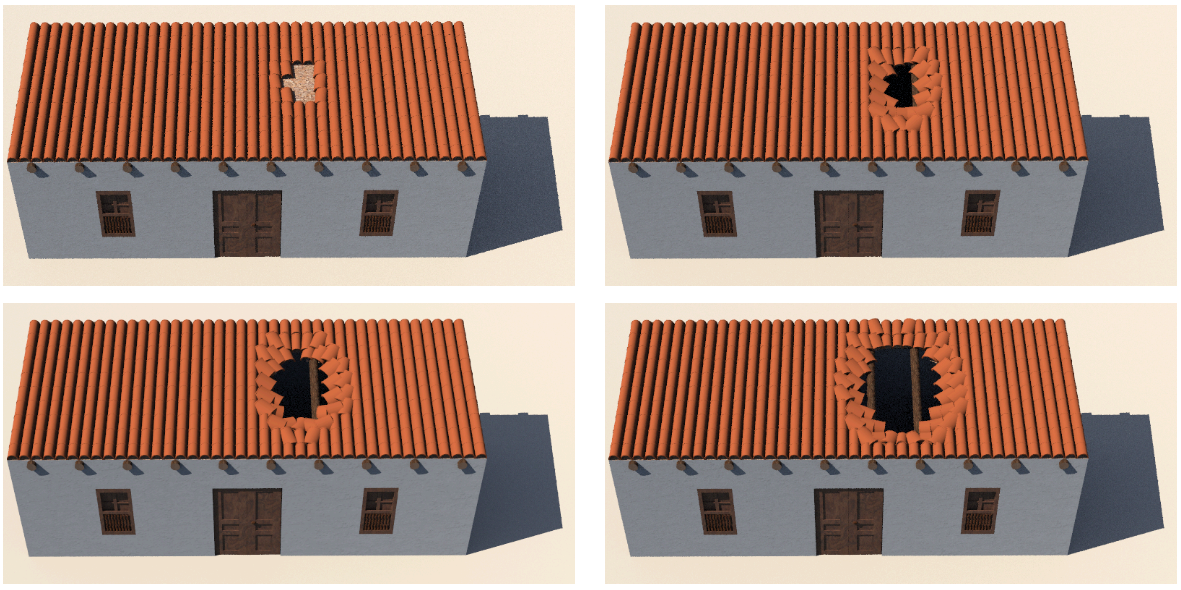


Figure 6.6: Degradation of roof with supporting beams.

Chapter 7

Conclusive Remarks

Contents

7.1 Conclusion

We have shown you a aging simulation pipeline as a proof of concept. The simulation extension as we have shown can be integrated in procedural algorithms such as CGA Shape. The grid-model approach of representing aging attributes has potential for realistic simulation, as each point of the building has a sense of space and interaction with neighboring points. Ray Tracing can be used to simulate rainfall, graph propagation algorithms can be used to simulate the flow of fluids and mathematical formulas can be used to affect points over time.

We have included means to store the simulation attributes of the building so that they can be used for later frames, which allows us to follow an iterative method of progressive degradation rather than a time-based formula method. The iterative approach makes sure that the simulation flows from one state to the next in a logical order while allowing stochastic behavior; whereas time-based functions have to be deterministic in order for this to be the case.

We have also introduced simple methods for presenting degradation in buildings. Artists can ofcourse use their skills and creativity to create more intricate aging tools.

7.2 Future Work

We believe that procedurally based aging has a lot of potential in benefiting the graphics industry. However our work has a lot of room for improvement before reaching the goal of mainstream attention. For that reason we have listed a number of issues that our team will try and solve in the future.

Interior Spaces

Since our aging tools allows for the opening of holes through the surface of a building an interesting problem arises. The procedural algorithm that we have chosen focuses on constructing the exterior surface of a building without giving any thought for the interior spaces. That of course can be solved by incorporating procedural algorithms that given a volumetric space attempt to create an interior space. We have mentioned one example in our Previous Work section (Chapter 2).

Debris

Debris and rubble give the viewer pieces of information regarding the way that the building collapsed. The lack of this detail can easily arouse the suspicion of human observers and break the realism of the scene. Of course this is not the case if the disappearance of that mass has an in-game explanation. One possible solution is to detect areas of collapsed surfaces and the use of ray casting to place a debris model on a solid surface below the that surface. Additionally artists could use procedural algorithms to generate a random pile of rubble if variety is of concern.

Structural Damage

Our simulation pipeline as of now is concerned mostly with surface level degradation. Extreme forms of structural damage caused by catastrophes such as earthquakes, tornadoes and floods require knowledge of the building's underlying supporting structure. This is admittedly the hardest problem to solve, especially without the use of manual configuration. There are without a doubt ways to procedurally create a building in way similar to how builders create houses, but we haven't explored that possibility yet.

Performance

Although in-built Houdini SOPs are highly optimized, our custom python scripts do not have that advantage. In cases where expensive graph algorithms need to run on our grid model a significant drop in performance will be noticed. One way to deal with this problem is to simplify the code as much as possible at the expense of accuracy. Another way to increase performance is to lower the resolution of the grid model. Alternatively the code can be written with parallel programming in mind. As of now our scripts make use of a single thread, thus parallelizing them will greatly improve performance.

Usability

Our priority was to make the simulation pipeline work and not to make industry grade tools. Despite that our CGA procedural building tools themselves were designed to be straightforward and easy to use. One only needs to drop the OTL on the current workspace, connect the inputs correctly and get the results. On the other hand our simulation tools are a bit more clunky to use.

An important usability issue is the fact that our simulation only moves forward frame by frame. If an artist needs to jump ten frames, he will need to wait for the simulation to run frame by frame until that point. On top of that, if the artist needs to compare a simulation result from a previous point he will need to restart the simulation and recompute all the frames until that point. This is undesirable for two reasons: A) It takes too long for the artist to go to the wanted simulation frame and B) restarting the simulation and recomputing the frames might get a completely different result due to the stochastic elements in the simulation. Thus we have thought of storing all of the versions of the simulation on disk once they have been calculated. All of these states can be accessed freely simply by moving the animation slider to the desired frame. The artist will also be able to restart the simulation with different parameters with the press of a button.

Bibliography

[1] Cracking in brickwork - architecture technical information sheet. http://www.residentialreports.com.au/wp-content/uploads/2015/03/Cracking-in-Brickwork.pdf, 2011.

[2] Classification of cracks: Nature-wise, width-wise shape-wise. http://gharpedia.com/classification-cracks-nature-wise-width-wise-shape-wise/, 2016.

[3] Allegorithmic. Substance. https://www.allegorithmic.com/products/substance-painter, 2017.

[4] Benjamin Bradley. Towards the procedural generation of urban building interiors. *University of Hull*, 2005.

[5] ARCHITEXTURA by Persephone. Brownstone facade. https://www.turbosquid.com/FullPreview/Index.cfm/ID/482792.

[6] Dmitriy Chugai. Dull concrete building façade. http://texturelib.com/texture/?path=/Textures/buildings/buildings/buildings_buildings_0076.

[7] ESRI. Cityengine. http://www.esri.com/software/cityengine/, 2017.

[8] Andy Farnell. An introduction to procedural audio and its application in computer games (2007), 2013.

[9] Eric Galin, Adrien Peytavie, Nicolas Maréchal, and Eric Guérin. Procedural generation of roads. In *Computer Graphics Forum*, volume 29, pages 429–438. Wiley Online Library, 2010.

[10] Havok. Physics. https://www.havok.com/physics/, 2017.

[11] Lejaren A Hiller and Robert A Baker. Computer cantata: A study in compositional method. *Perspectives of New Music*, pages 62–90, 1964.

[12] Koichi Hirota, Yasuyuki Tanoue, and Toyohisa Kaneko. Generation of crack patterns with a physical model. *The Visual Computer*, 14(3):126–137, 1998.

[13] Koichi Hirota, Yasuyuki Tanoue, and Toyohisa Kaneko. Simulation of three-dimensional cracks. *The Visual Computer*, 16(7):371–378, 2000.

[14] Hayley N Iben and James F O'brien. Generating surface crack patterns. *Graphical Models*, 71(6):198–208, 2009.

[15] Robert G Laycock and AM Day. Automatically generating roof models from building footprints. 2003.

[16] Aristid Lindenmayer. Mathematical models for cellular interactions in development i. filaments with one-sided inputs. *Journal of theoretical biology*, 18(3):280–299, 1968.

[17] Jess Martin. Algorithmic beauty of buildings methods for procedural building generation. *Computer Science Honors Theses*, page 4, 2005.

[18] Stéphane Mérillou and Djamchid Ghazanfarpour. A survey of aging and weathering phenomena in computer graphics. *Computers & Graphics*, 32(2):159–174, 2008.

[19] Mojang. Minecraft. `https://mojang.com/games/`, 2011.

[20] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural modeling of buildings. In *Acm Transactions On Graphics (Tog)*, volume 25, pages 614–623. ACM, 2006.

[21] Gregoire Nieto. Simulation of Aging for Procedural Buildings. Master's thesis, University College London, 2014.

[22] NVIDIA. Physx. `http://www.geforce.com/hardware/technology/physx`, 2017.

[23] Yoav IH Parish and Pascal Müller. Procedural modeling of cities. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 301–308. ACM, 2001.

[24] GL Possehl. Mehrgarh in oxford companion to archaeology, 1996.

[25] Ruben M Smelik, Klaas Jan De Kraker, Tim Tutenel, Rafael Bidarra, and Saskia A Groenewegen. A survey of procedural methods for terrain modelling. In *Proceedings of the CASA Workshop on 3D Advanced Media In Gaming And Simulation (3AMIGAS)*, pages 25–34, 2009.

[26] Ruben M Smelik, Tim Tutenel, Rafael Bidarra, and Bedrich Benes. A survey on procedural modelling for virtual worlds. In *Computer Graphics Forum*, volume 33, pages 31–50. Wiley Online Library, 2014.

[27] Side Effects Software. Geometry attributes. http://www.sidefx.com/docs/houdini/model/attributes.

[28] Side Effects Software. Houdini fx. http://www.sidefx.com, 2017.

[29] Bethesda Softworks. The elder scrolls ii: Daggerfall. https://elderscrolls.bethesda.net/daggerfall/, 1996.

[30] Piotr Tomsinksi. Behind the scenes of the witcher 3: Wild hunt – cinematic dialogue system, 2016.

[31] Peter Wonka, Michael Wimmer, François Sillion, and William Ribarsky. *Instant architecture*, volume 22. ACM, 2003.