Honours Project Report

Generating Hidden Structure for Procedural Scenes

Richard Baxter
Supervisor: James Gain
Co-Supervisor: Rudy Neeser

	Category	**Min**	**Max**	**Chosen**
1	Software Engineering/System Analysis	0	15	0
2	Theoretical Analysis	0	25	0
3	Experiment Design and Execution	0	20	15
4	System Development and Implementation	0	15	15
5	Results, Findings and Conclusion	10	20	10
6	Aim Formulation and Background Work	10		10
7	Quality of Report Writing and Presentation	10		10
8	Adherence to Project Proposal and Quality of Deliverables	10		10
9	Overall General Project Evaluation	0	10	10
Total marks		**80**		**80**

Department of Computer Science
University of Cape Town
2009.

Abstract

This report explores a method of augmenting an existing building generation method (more specifically a wall generation method) in such a way that internal structure of that building can be extracted. This is used in the generated building model to give it a physical internal structure that, when hit, will collapse and break apart in a realistic manner. The system designed is able to produce video of simulated buildings falling and collapsing.

In order to validate the method, user experiments were conducted to determine the level of realism and Heuristic evaluations were performed to extract noticeable features. These results showed that users felt the simulations highly realistic, even though they could notice a number of distracting unrealistic features. The heuristic evaluation extracted valuable information about the level of detail of internal structure that is needed for such physics simulations, as the current method, whilst adequate and very believable, was not completely sufficient.

As this method would be of particular interest to the games and movies industry, performance tests were run to determine if the method was fast enough to allow real-time changing of building parameters (> 30 FPS), or failing that, sufficient speed as to be interactively used by a designer ($10 - 30$ FPS). Performance testing was also used to determine if the generated 3D geometric objects that represent the building, with all internal structure, could be simulated in real-time. The method generates buildings at a high enough speed that parameter changing is comfortabley real-time. However, the simulations were found to be too slow for even interactive use excepting for only very small shed like buildings.

The increased number of geometric objects and interactions, as a result of increased structure, caused massive slowdown, however with GPGPU technology and different physics-engine considerations, this could be alleviated. This, combined with the high level realism noted in user experiments, suggests that this is a topic that merits further investigation.

Acknowledgments

I would like to thank my supervisor, Dr. James Gain for his guidance during this project and for the incredibly detailed feedback and effort put in on numerous occasions. I would also like to thank my co-supervisor Rudy Neeser for his assistance and guidance, especially for the more low-level issues that arose during the project. Also, my project partner, Zacharia Crumley, whom I was able to lean on and rely on during the more intensely busy periods of the project, and with whom I was able to be pedantic with about small issues that normal people would find weird to even consider. I would like to thank all the users that took part in our tests and gave us some very useful and insightful ideas and comments. Finally, thanks to all my class-mates, with which many late nights in the labs were spent.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

It seems almost inevitable that computing power will increase as time goes on. Innovations in CPU power, reduction of bottlenecks and GPGPUs mean faster processing and faster rendering of computer graphics. This means more expansive and detailed environments become feasible and even runnable in real-time. The gaming and movie industry has taken full advantage of these sorts of technologies, allowing for the creation of sandbox games such as GTA4 [6] and MorrowWind [1] that are set in massive game worlds and movies such as Wolverine, The Incredibles and The Matrix Trilogy. However, as the world sizes and details increase, so does production time and cost to create these worlds. Artists and modelers have to be hired to create the scenes, teams of 'riggers' have to take these models and break them up in such a way that they can be realistically broken apart if impacted, and the overall task of managing large teams increases exponentially.

1.1 Procedural Generation

To alleviating the amount of man hours needed to produce a large scene, and hence reduce time, effort and number of people needed, procedural generation has been used to partially automate the process. Generating models of buildings via an automated process allows artists to concentrate on what they want the final outcome to look at and not to deal, as much, with repetitive tasks.

Procedural generation of buildings has been explored in the scientific community and is used in industry. However, scenes including objects that are destroyed and broken apart in a realistic manner have become more popular. In order to do this, modeled buildings cannot simply be four huge slabs for walls and a roof. They need to be breakable so that, when impacted, they do so in a realistic and believable manner, with debris being scattered and the building walls crumpling and destroyed. Current procedural generation methods do not create buildings with these breakable sections (or break-points) and are normally added manually by a team of riggers. This project intended to automate that process.

1.2 Research Topic

We wish still to use procedurally generated building models, as not doing so would be against the ethos of automation. It is a impossible task to create a process that calculates breakpoints given an arbitrary building model without any information about the building's characteristics. One method

could be to add this building structure information after the building is generated. However, it is more intuitive and natural to gain this building structure information during the generation process. Often structural information is inherent in the generation process and extraction. Use of that inherent information in combination with information provided by the building designer, can be used to generate building breakpoints.

We wish to explore whether there is a viable method that can be created to do such a task, whether it produces realistically believable results, as well as explore how efficient such a method would be in terms of various performance metrics

1.3 Overview

This report outlines the building generation section of this project. The other section, done by Zacharia Crumley, involves adding internal structure to trees and subjecting them to simulated wind. Any reference to 'tree section' or 'tree generation' that appears in the report refers to Zaceria's section of this project.

Chapter 2 will outline current research and concepts in the field of procedural building generation. Chapter 3 outlines the design and implementation of the algorithm and the final testing system, Chapter 4 outlines the testing procedure to validate the system and Chapter 5 outlines the results of that testing. Ethical and licencing issues are discussed in Chapter 6. Future Works and Conclusions are in the final Chapter, 7.

Chapter 2

Background

2.1 Overview of Research

There is surprisingly little research solely devoted to the topic of procedural generation of buildings/architecture. However, many papers on procedural generation of cities have sections on building geometry (e.g., Parish et al. [13]). "Instant architecture" [17], "Procedural modeling of buildings" [12], "Procedural modeling of cities" [13] and "Wall Grammar For Building Generation" [9] are the main papers around which most up-to-date research has been conducted. Papers such as "Citygen: An interactive system for procedural city generation"[8] and "Real-time procedural generation of 'pseudo infinite' cities" [7] have built from this research.

However, most follow-up papers are largely about city generation and their use of the more advanced building generation techniques are somewhat lacking. The research tends to use simpler techniques and then suggests that they should be replaced by more advanced methods. Most, if not all, research on procedural generation (of anything) refer to "The Algorithmic Beauty of Plants" [14] and most papers that address procedural generation of buildings and/or cities reference "The logic of architecture: Design, computation, and cognition" [11] and "Pictorial and formal aspects of shape and shape grammars" [15] as theoretical background from the world of architecture.

2.2 Definitions and Concepts

2.2.1 L-Systems

An L-system (Lindenmeyer system) consists of a string of characters from an alphabet α (initially set to some non-empty string) and a set of production rules. A production rule is a function from A to B where A and B are ordered tuples of characters from α. The L-systems iterates in steps by parsing the L-system's string, either character by character or all in parallel, and replacing that character with other characters based on a production rule.

2.2.2 General Grammars

An L-system is, in fact, a specific type of something called a grammar (or a set grammar). The definition in Wonka et al. [17] is that a grammar consists of 4 sets: a set of terminal objects, a set of non-terminal objects, a set of replacement rules and an initial object. This general definition can be used to further describe string grammars (such as an L-system) and other grammars (such as shape grammars). The grammars operate by use of a set of production rules are used to 'rewrite' objects of the grammar (i.e. the same way characters are rewritten in L-systems). An end state can be reached when all the objects are terminal objects (though this is not necessary in all grammars).

2.2.3 Shape Grammars

Shape grammar can be defined so that one can do procedural generation on 3D or 2D shapes, meaning that production rules operate on the shapes themselves, turning one shape into one or more other shapes. Potentially, this can use architectural concepts more accurately. For this reason, and whilst still achievable with an L-System, it is argued by Wonka et al. [17] that a shape grammar is a more natural method than trying to create an L-system to do the same type of generation. A rewrite in a shape grammar involves replacing one shape with one or more other shapes.

2.2.4 Split Grammars

A more useful specialisation of the shape grammar, the split grammar (Wonka et al. [17], Müller et al. [12] & Larive et al. [9]) includes operations to break up (or split) shapes into meaningful chunks (e.g. splitting a block/building into multiple stories). In the case of Wonka et al. and Müller et al., the splits are performed on 3D geometry. In Larive et al. the splits are done on 2D façades.

2.3 Procedural Generation of Buildings

In the literature around the topic of procedural generation of buildings there is a clear thread of work from Parish's L-systems [13] to Wonka et al.'s Split grammar [17] to Müller et al.'s extended split grammar [12] to Larive et al's Wall Grammar [9] as each paper built on the lessons learned from the previous one. Unfortunately, besides these 4 major advancements there is little else on the subject that extends this work. A potentially interesting paper was Lipp et al.'s 'Interactive visual editing of grammars for procedural architecture' paper [10], however this was more focused on making procedurally generated building directly editable instead of indirectly editable.

The following sections are split into geometry and facades. This distinction is made since in all procedural generation techniques, it was found that these two sections are dealt with separately (either one after the other, or as separate grammars used for each section, or some other distinction is made). Geometry deals with the 3D space that the building occupies, whereas the façades deal with what details appear on the surfaces of those geometries (usually windows, doors, etc..).

2.4 Geometry

2.4.1 Categorisation

There are two main areas that seem to have formed. The first is use of geometry growth systems (e.g. [13] ,[7]), characterised by starting with a simple shape or polygon and progressively extruding and cutting to create building-like geometry. This involves L-Systems or simple iterative algorithms that are performed on a given ground plan (the plot on which the building generator is asked to work).

The other techniques is the use of complicated geometry division and decomposition whereby architectural concepts of the building, such as levels, roofs, doorways, etc., are used to generate the building ([17], [12], [9]). The main concepts used to tackle this are shape and split grammars. These techniques involve simple geometry creation using extrusion or geometry insertion, but then a context is added to the geometry (e.g., "this cube is part of the main building body", and "this block is one of the levels", or "this 2D plane is a window area") so that it can be split into logical divisions by use of rules (e.g., there is a rule to divide a rectangular block, such as a main building block into a set of shorter blocks stacked on top of each other. Thus the main building can be split into levels).

2.4.2 Characterisation

The simple grammar systems produce very fast, but very uniform designs. The grammar does not take into account architectural considerations since it builds simple polygon extrusions that are tweaked to look building-like. Although architecturally the designs are lacking, high levels of visual complexity can still be achieved.

The more complicated shape and split grammars can produce a much wider range of building types and rules that take advantage of architectural knowledge can be more naturally created. Stylised buildings with high visual detail can easily be made and remade (by adjusting parameters). The complicated grammars also have the property of potentially being more 'realistic', due to their abstraction of rules that are created with architectural concepts. 'Realism' has obvious visual implications, but also implications if one wishes to make these buildings destructible.

2.4.3 Major Advancements

2.4.4 L-System

In Parish & Müller [13], building geometry is built from a simple L-System, and totally automatically (no user interaction). The L-System consists of transformation, extrusion, branch and termination rules which are all performed (initially) on the ground plan that was given to the building geometry generator. In this way, fairly convincing building geometries can be made that have 'high visual complexity' [13], although the building are limited to very 'blocky', orthogonal shapes due to the nature of the grammar. However, these geometries do not, in any way, represent the function of the buildings, they are simply transformed geometry with no reference to floor plans, number of stories, structural references, etc. Also the only way to create variation is to rewrite the production system. In Parish & Müller [13] this was done three times, once each for skyscrapers, commercial buildings and residential houses.

2.4.5 Split Grammar

Wonka et al. [17], concentrates exclusively on building generation and a more advanced method of geometry generation is introduced. It is argued that simple L-systems are not sufficiently powerful to create the varied building geometry that is required. It is also argued that an L-system models growth patterns, but since architecture does not have such growth patterns, L-systems are not a natural type of grammar for building generation. Shape grammars are considered to be the better method as there has been research in the field of architecture (namely construction and architectural analysis) motivating the use of shape grammars. In section 2 of "Instant Architecture", Wonka et al.[17] cites architectural works as motivation for their use of shape grammars, as well as Stiny [15], who developed a shape grammar for building generation.

The basic idea is to create a large number of grammar rules that can be used to create building geometry (as well as façades). Each of these rules is associated with certain attributes. The user does not specifically pick which rules the generator uses but rather chooses attributes and, then, via a stochastic statistical selection process the 'correct' rules are chosen at each step (by 'correct' rules it is meant that the rules reflect the attributes chosen by the user). A more general split grammar is favoured over the L-System grammar. Wonka et al. offers a concise mathematical definition of this in section 4 of their paper[17]. It involves the generator starting with a ground plan, building some basic geometry, and then splitting it into, say, levels (how exactly would depend on what attributes the user chooses). It then, say, splits the levels up into places for the windows and doors and so on (again, how this is specifically done depends on what rules where chosen, which depends on the user input). This system allows for extremely varied building geometries that are created with architectural concepts behind them (assuming the production rules have been created with informed architectural insight and knowledge). However, there are still problems,h in that geometries within a building can intersect and occlude each other, potentially causing unnatural placement of objects (e.g., a window placed on an occluded wall). No solution was found to this in Wonka et al.'s paper [17].

2.4.6 Extended Split Grammar

Müller et al. [12] introduce a far more complicated and powerful system based on his and Wonka et al.'s previous work [17]. The process was revised and upgraded. The grammar rules used in Wonka et al. [17] are enhanced to include scoping of operations, occlusion detection, truth conditions and other more complicated operations, such as more complicated splitting, repeating and decomposition. This forms a scripting language that one can use and edit (and share, it was suggested in Müller et al. [12]) with little knowledge of the underlying systems. The extended power allows for an even more vast range of buildings. The extended split grammar encompasses both geometry and façade generation.

Actual creation of geometry occurs via "mass modeling". First a simple 3D geometry is created, after which the 2D façades are extracted. This allows for the façades as well as the geometry to be used in production rules. An octree is used for occlusion queries and snapping is implemented. Again, these are tools one can use in a production rule. The system holds many technical challenges for implementation due to its complexity, but to date is the most varied in potential building types. Although far easier for a user to use than previous work, the productions rules are still at a relatively high technical level. It is undetermined whether this is an efficient or intuitive method for artists and other non-technical people. This is addressed to some degree in Lipp et al.'s paper [10].

2.4.7 Wall Grammar

Larive et al. [9] follows a similar route to Wonka et al.'s [17] split grammar by creating a form of split grammar called a Wall Grammar. According to Larive et al.'s "Wall Grammar For Building Generation" [9] this can be considered a type of split grammar except that it operates on 2D walls rather than 3D shapes. Multiple walls are then used to generate a building. Unlike Wonka and Müller et al.'s work [17] [12], the wall grammar only consists of 5 symbols as opposed to hundreds (or even thousands) of symbols which are selected via user attributes. This allows for more direct control of the outcome of the building. Roofs are generated after the walls have been created. This method is not as complex as the other shape grammars mentioned. However, they have created an, arguably, simpler system to use. They motivate this claim by the decreased number of rules which allow artists and graphic designers to easily learn the system. Although it was claimed that complex buildings can be created with the wall grammar, the building variation is not as great as in Wonka and Müller et al [17] [12]. The system is, however, far simpler to implement.

2.4.8 Other Techniques employed

In Greuter et al. [7] a pseudo-random number generator was used to procedurally generate a city. The buildings are also generated with the same pseudo-random number generator. The technique used can probably be best described as polygon extrusion. They start from the top of the building and extrude a polygon to form a building section. The polygon is then randomly changed or added to and then it is extruded down again. This is repeated a number of times to create a 'city building'. Kelly and McCabe [8], use an even simpler approach, one main extrusion from the ground plan and smaller extrusions to generate the windows. Needless to say, this technique can only produce one type of building and the paper states that it should be replaced with a more sophisticated building generation method.

2.5 Façades and Textures

The distinction between complex and simple carries forward to façade generation, but with an additional consideration. There is a clear split between after-geometry generation and during-geometry generation of façades. The simplest methods only generate the façades after the geometry has been generated (after-geometry), as a separate stage of generation. The more complicated methods generate facades during the same stage in which the geometry is created. To be clear, the actual textures are usually synthesised separately from the geometry, but the distinction here is when the façade details are generated. The complex methods build the façade generation into the rule system and thus it is created as part of that system and not in a separate stage. During-geometry façade generation relies, in some sense, on the geometry[9] generation stage.

2.5.1 Major Advancements

The following sections will outline the various important papers on the subject in an attempt to tease out more clearly the distinction between after-geometry and during-geometry façade generation.

2.5.2 Layered Grid: After-geometry

Parish & Müller [13] use a simple geometry generation technique and the façade details are generated after geometry. They employ what they call a 'Layered Grid' technique. This involves creating a number of grids using interval groups, which are, put simply, patterned grids that can be layered (i.e., may have a patterned grid within a grid cell). These layers are then logically AND'ed together to create areas on the geometry where the windows (and doors) are supposed to appear. This method, like the geometry stage, is not parametrised, and thus the user has no control over its style and final appearance.

2.5.3 Split Grammar: During-geometry

In Wonka et al. [17] split grammars are used to create geometry. After specific sections of geometry are generated (i.e., the shapes are terminal objects) the '2D' polygons that represent the sides of the shape are extracted and potentially split further. Façades are created on those polygons. Which façades are then applied depends on what attributes the polygons have. Thus, the façcades are generated 'during' the geometry phase. As with the geometry, the attributes are propagated via a Control grammar, which is a simple context-free grammar. The control grammar operations run parallel to the split grammar operations in that 'rewrite' operations are performed in the content grammar when ever the split grammar 'rewrites' are performed. Attributes are propagated down from the geometry and when the geometry has been split sufficiently (i.e. has reached a terminal symbol) the façades can be applied. Like the geometry, a number of rules can be chosen depending on what attributes and styles the user wants. Again, which rule is used is chosen via a stochastic method.

2.5.4 Extended Split Grammar: During-geometry

In the paper written by Müller et al. [12] the façades are created with the same grammar system as the geometry. They are simply extra production rules that are added to the whole grammar system. The additional functionality to check for occlusion that the paper contributes means that the façades can be created so that wall objects (such as windows) would not be created behind other objects.

2.5.5 Wall Grammar: During-geometry

Larive et al.'s [9] wall grammar deals almost exclusively with façade generation. Thus it can be considered a separate process to the geometry generation. However, the method involves building the basic geometry of the building first to get the 2D façade sizes, either by floor plan extrusion or some other method. The actual façades are generated by the wall grammar, including all extrusions and the final 3D geometry descriptions (in reality 2.5D since there is just a 2D plane and depth). The use of a reduced rule-set allows an artist to only create a single set of rules for a single wall description. This wall description will then be used to create the different façades based on the basic geometry. This has the additional advantage of allowing many different styles of wall to be applied to the same basic geometry.

2.6 Synthesis

There is not a vast amount of research in the area of procedural generation of buildings but sufficient such that there are a number of techniques that allow one to choose between a highly technical and accurate technique or a less technical, simpler technique. Whether split grammars are the way to go in the future

is undecided in the literature, they do allow for very complex, 'realistic' and detailed buildings, but are very difficult to use if you are not technically inclined. That said, other techniques that allow artists and graphic designers to more easily generate buildings tend to be unable to produce as much variation and 'realism'.

Realism may not be necessary for a purely artistic creation. However, if one is considering making these building interact in some realistic manner (e.g., creating interior floor plans or destroying them), realistic building models would most likely lend themselves to more natural interaction. For example, a building created in levels can be broken up more naturally then a building model that is simply an extruded polygon.

For the purpose of this project, the split grammar of Wonka et al. [17] and Müller et al. [12] is ideal. The generation process inherently contains much structural information that could and would be used in a destructible model of a building. However, it is far too complicated to implement in the scope of an Honours Project. Simpler methods like the polygon extrusion methods used in Greuter et al. [7] and Parish & Müller [13] are far too simple and uniform. More importantly, they do not lend themselves naturally to the creation of break-points in the models as they do not generate models with such structural concepts in mind.

The wall grammar of Larive et al.'s [9] sacrifices some architectural accuracy for ease of generation. Whilst not offering an internal structure, as the 3D shape grammars might do, one can attach such structure using the information in the wall generation process. This offers a good compromise between scope and how well the underlying method lends itself to generating break-points. Another advantage is that the wall grammar is more easily used by non-technical people such as artists than the more complicated shape grammars.

Chapter 3

Design and Implementation

3.1 Building Generation Method

For this project it was decided to use the Wall Grammar [9]. When considering the method to pursue, the three main considerations were: implementation complexity of the method, potential for variation in buildings and how 'realistic' the generation method is (i.e. how much it can naturally incorporate architectural concepts). The 3D split grammar of Müller et al. [12] and Wonka et al. [17] is too complicated to fit in the scope of this project but might produce more realistic results since it deals directly with 3D geometry. The simpler polygon extrusion algorithms do not offer sufficient variation in building styles and are under scoped for a project this size. Wall Grammars offer sufficient variation and an acceptable degree of potential architectural 'realism'. It is also well scoped for a project of this size and offer many potential extensions.

In this chapter, section 3.2 will explain the Wall generation algorithm, as interpreted from Larive et al [9]. Section 3.3 explains the various modifications to the algorithm that allow the building to be split up into destructible parts.

3.2 Façade Generation Method

A wall grammar generates a single 2.5 dimension façade of a building. After a number of façades are created, they are attached together to form a while building. For the remainder of this paper the words wall and façade are used interchangeably. A single façade is described by a number of *GeometryDescriptions*, each containing a rectangle and context data. The concept of a GeometryDescription is used to collectively refer to a rectangular geometry with some associated context data about that geometry. Exactly what the context data is depends on various factors, but it usually includes the geometry's texture and depth (more detailed description of context data is described in Section 3.2.3 and Appendix B).

3.2.1 Geometry Description

A single 2.5D wall is created by taking a single large GeometryDescription and splitting it into a number of smaller GeometryDescriptions recursively. As they are split, additional data is added to the child GeometryDescriptions. This is achieved via a number of production rules. All walls have a depth value,

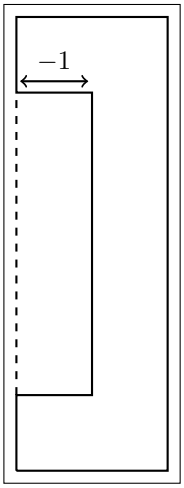

Figure 3.1: Cross-Section of an Extruded Wall (depth = -1)

initially set to 0. Depth represents how much it sinks into or extrudes out-of the wall (see Figure 3.1). They also have a minimum, maximum and preferred size (explained more fully in section 3.2.4). Preferred sizes are set by the user and may be set to 'NULL' for no preferred size [1].

3.2.2 Symbols and Production Rules

The actual symbols and their parameters are set by the user. There are 4 different types of non-terminal symbols and 1 type of terminal symbol. Each symbol represents a section of a wall (a GeometryDescription) and how that GeometryDescription is to be split further.

Each production rule is based on its symbol's type. Symbol types and associated production rules are as follows:

(Note: For all following rules, W can be a symbol of any type)

- Wall Panel: (Terminal Symbol)
  This is the most basic type of geometry and simply adds a texture variable to the GeometryDescription WP as well as any pointers to pre-generated models (such as models of window panes or doors). This is the only terminal symbol. Since this is the terminal symbol, it has no associated production rule.
- Bordered Wall: $BW \rightarrow W$
  This transforms the GeometryDescription BW to GeometryDescription W by making it smaller and adding borders (up to 4 border rectangles). Borders are similar to GeometryDescriptions, they contain a rectangle, texture and depth, however, they are not considered symbols inside the Wall Grammar (hence they are not transformed any further by any production rule). See Figure 3.2.
- Extruded Wall: $EW \rightarrow W$
  This simply transforms the GeometryDescription EW into W by altering the depth value of EW, allowing one to create walls sections with depth. NOTE: This was not fully implemented as the extrusions required additional geometry to cover the inner or outer extrusions. Implementing this

[1] One can set the preferred size to NULL in one or both dimensions, thus a wall can have a preferred width but no preferred height, or vice-versa

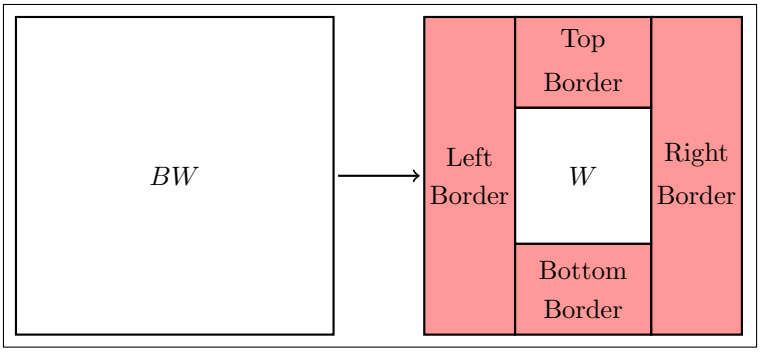

Figure 3.2: Example of Bordered Wall Production Rule $BW \rightarrow W$

in a 3D simulator as well as allowing it to break into finer-detail (see section 3.3.4) was not possible due to time constraints

- Wall Grid: $WG \rightarrow W$
  This transforms the GeometryDescription WG into multiple walls of type W, either tiled vertically, horizontally or both. How many are tiled in each direction depends on W's preferred, minimum and maximum size. The right hand side of the production rule contains only one symbol and not multiple symbols as the transformed GeometryDescriptions are just identical copies of W and because the number of copies depends on size information calculated inside the generation process. See Figure 3.3.
- Wall List: $WL \rightarrow W_1, W_2, ..., W_n$
  This transforms the GeometryDescription WL into a number of child walls ($W_i \forall i \in [1, n]$), either tiled vertically (bottom to top) or horizontally (left to right). See Figure wallListPic for example.

For each non-terminal symbol the user creates, they can create a single corresponding rule. In each case, the user sets the rule's right hand side to some other symbol to complete that rule. The Wall List is an exception, as it can contain a list of child symbols as its right hand side.

A set of rules that cause a cycle is invalid. For instance, rules $\{A \rightarrow B, B \rightarrow C, C \rightarrow A\}$, will cause an infinite loop in the generation process (see 3.2.4).

3.2.3 Implementation of the Wall Grammar

The low level implementation of the symbols are represented by instances of sub-classes of the AbstractWall class. Each subclass (BorderedWall, WallGrid etc.) has additional variables that are used to describe its associated symbol type. A list of all the symbols' associated classes follows: (a more detailed, low-level version can be found in Appendix B)

- AbstractWall
  Contains a unique identifier (string or integer ID), a rectangle representing its final size, a depth variable, the preferred size set by the user and a child wall.

(NOTE: All following classes are subclasses of AbstractWall.)

- WallPanel
  Contains a texture variable that will be used to texture the geometry in the final generation as well as user set minimum and maximum sizes.

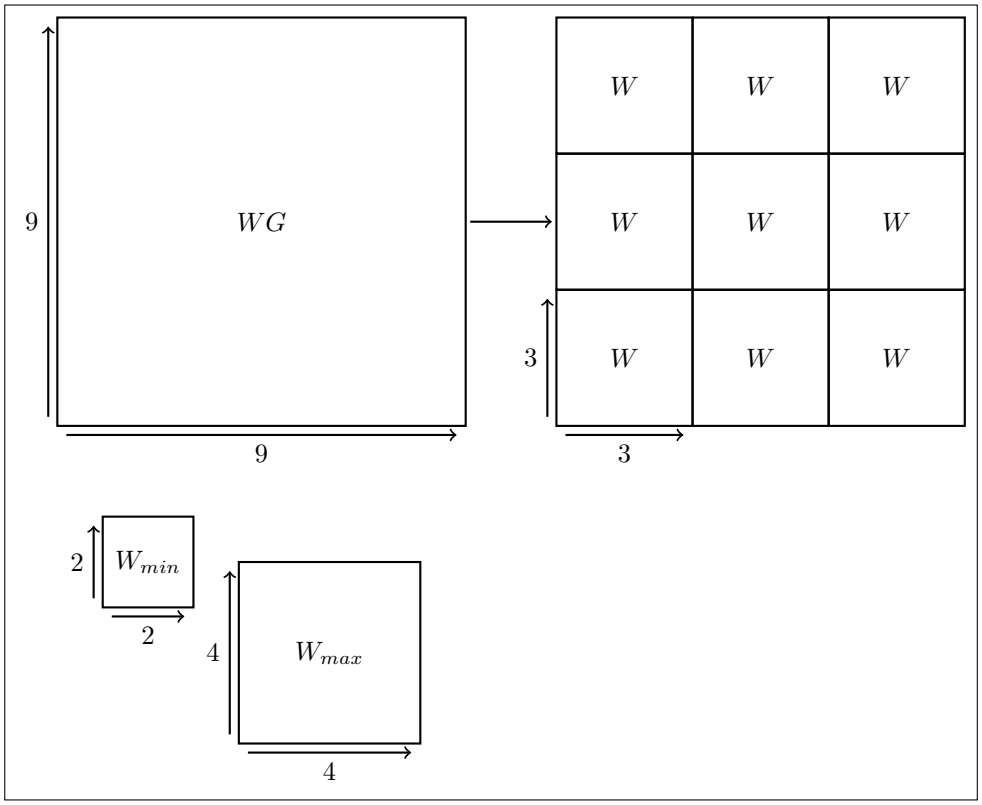

Figure 3.3: Example of Wall Grid Production Rule $WG \rightarrow W$: tiled in Vertical and Horizontal

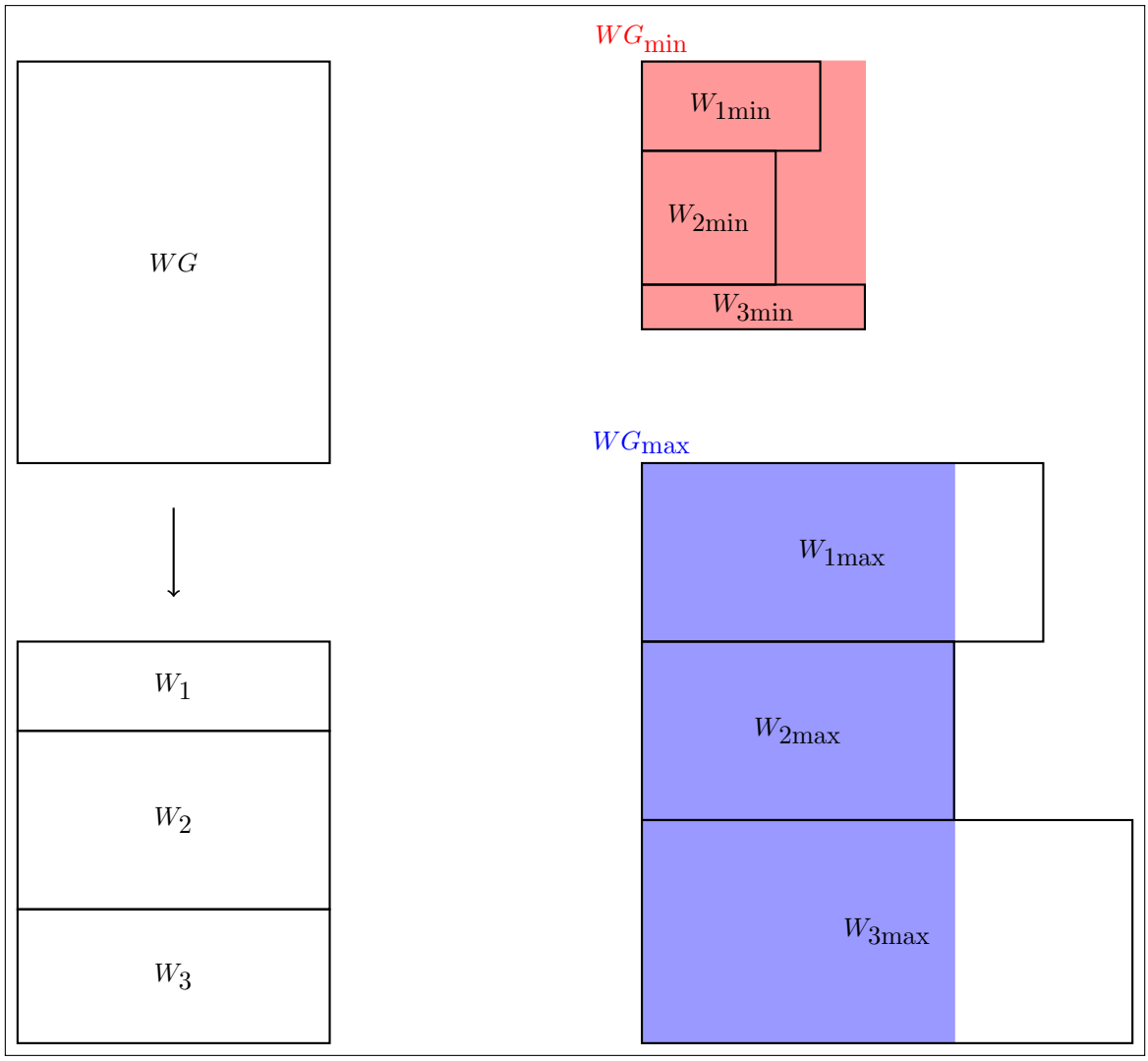

Figure 3.4: Example of Wall List Production Rule $WG \rightarrow W_1, W_2, W_3$: tiled Vertically

- BorderedWall
  Contains a texture variable that will be used to texture the border geometry in the final generation as well as margin variables for the left, right, top and bottom borders (detailed description of the Margin variable can be found in Appendix B).
- ExtrudedWall
  Contains the depth that describes how much to extrude (or alter the depth) when the associated Extruded Wall production rule is performed on it.
- WallGrid
  Contains an orientation variable that can be set to 'Vertical', 'Horizontal' or 'Both'
- WallList
  Contains an orientation variable that can be set to 'Vertical' or 'Horizontal' (but not Both) and a list of child walls. The child wall stored in AbstractWall, is hence ignored.

As stated in the beginning of the chapter, a single wall is described by a number of GeometryDescriptions. These are created by a number of symbols and a number of production rules, as well as one initial production rule (w) that creates the first large GeometryDescription object. See Table 3.1 for an example list of symbols. Table 3.2 describes the production rules. Figures 3.5 and 3.6 show each step of the generation process.

3.2.4 Generating the Building from the Production Rules

In order to build up the geometry of the building, 4 steps are needed:

1. Build a tree of symbols that represent how an initial large GeometryDescription is to be split up.
2. Calculate the minimum and maximum sizes of each node of the tree.
3. Using these sizes, calculate the actual sizes of the geometry.
4. Extract the actual 3D geometries and place in simulator or game environment.

Step 1: Building the Symbol Tree

The symbol tree that is being built has, as its root node, the right hand side of the initial w rule. This root adds children to itself based on the right hand side of its associated production rule. This is then repeated for each child. Thus, the tree is built up recursively.

This tree represents how the final building will be split up from a single initial GeometryDescription. An example set of symbols and rules are displayed in Tables 3.1 and 3.2. Figure 3.7 shows the associated tree generated.

Step 2: Calculating the Minimum and Maximum Sizes

In order to be able to calculate the sizes of the final geometries, the minimum and maximum sizes for each node on the tree must first be calculated. Each node's minimum and maximum size is determined by a function of their childrens' minimum and maximum sizes (or set by the user, as in the case of `WallPanel`).

Name	Type	Preferred Size	Type Specific Variables	
Window	Panel	(-1, -1)	Texture	Window
			Min Size	(15, 15)
			Max Size	(300, 300)
Door	Panel	(20, -1)	Texture	Door
			Min Size	(10, 10)
			Max Size	(90, 300)
Window Border	Bordered	(-1, -1)	Texture	Brick
			Left Border	Minimum 5
			Right Border	Minimum 5
			Top Border	Minimum 5
			Bottom Border	Minimum 10
Door Border	Bordered	(-1, -1)	Texture	Brick
			Left Border	Minimum 5
			Right Border	Minimum 5
			Top Border	Minimum 10
			Bottom Border	Minimum 0
Window Grid	Grid	(-1, -1)	Orientation	Horizontal
First Floor	List	(-1, 30)	Orientation	Horizontal
Main	List	(-1, -1)	Orientation	Vertical
Middle Floors	Grid	(-1, -1)	Orientation	Vertical
First Floor Windows	Grid	(100, -1)	Orientation	Horizontal

Table 3.1: Example Building Symbols

1.	w	→ Main
2.	Main	→ First Floor | Middle Floors
3.	First Floor	→ First Floor Windows | Door Border | First Floor Windows
4.	Middle Floors	→ Window Grid
5.	First Floor Windows	→ Window Border
6.	Window Grid	→ Window Border
7.	Window Border	→ Window
8.	Door Border	→ Door

Table 3.2: Example Building Production Rules

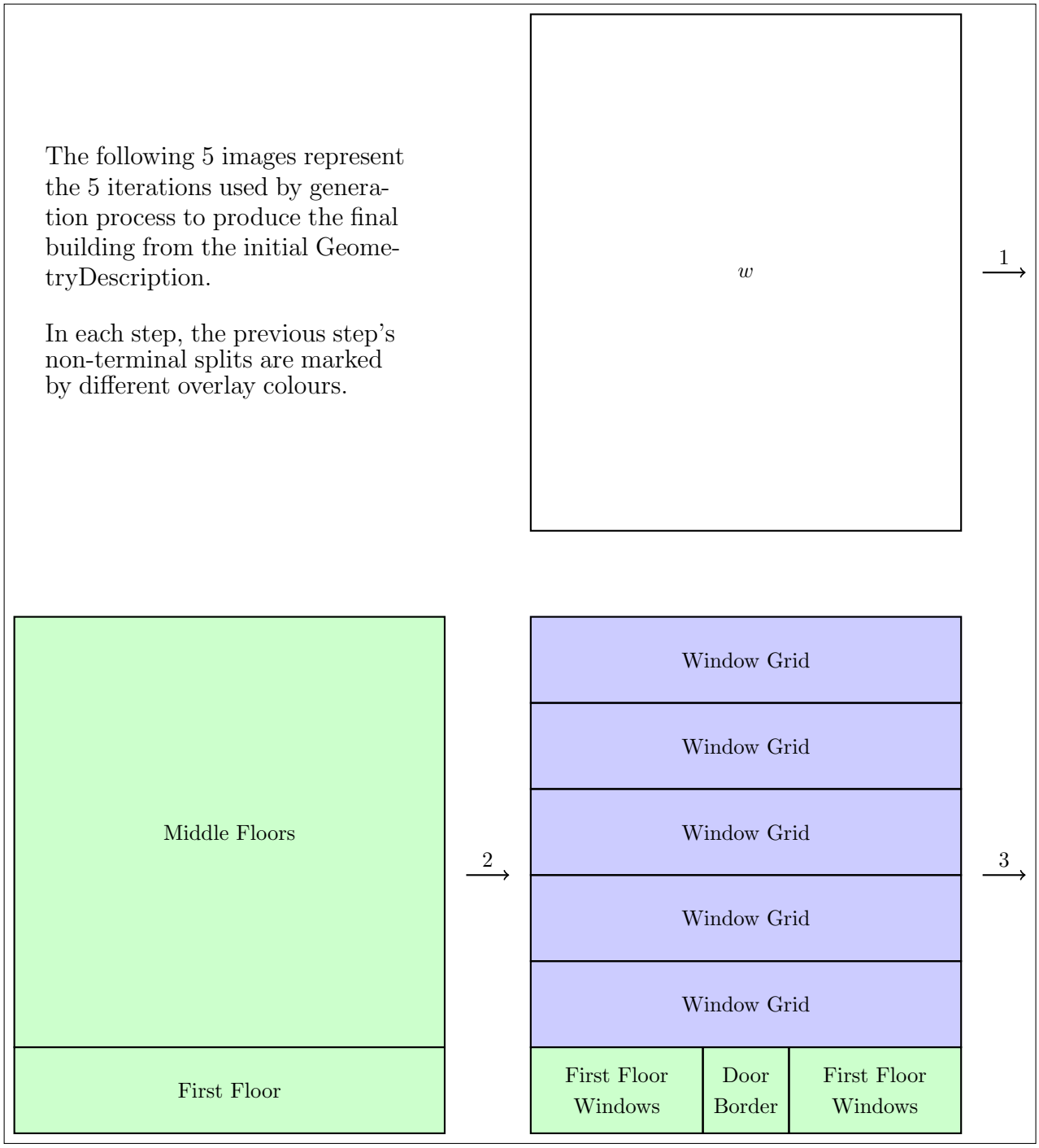

Figure 3.5: Example Building Generation (Part 1/2)

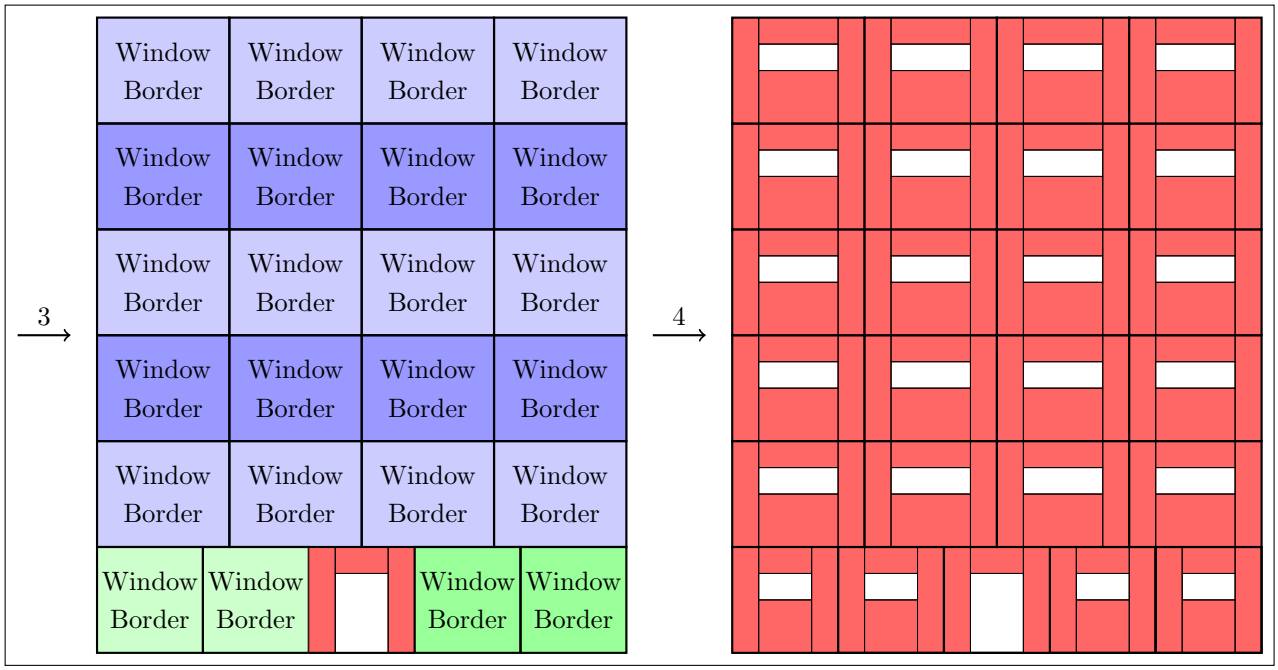

Figure 3.6: Example Building Generation (Part 2/2)

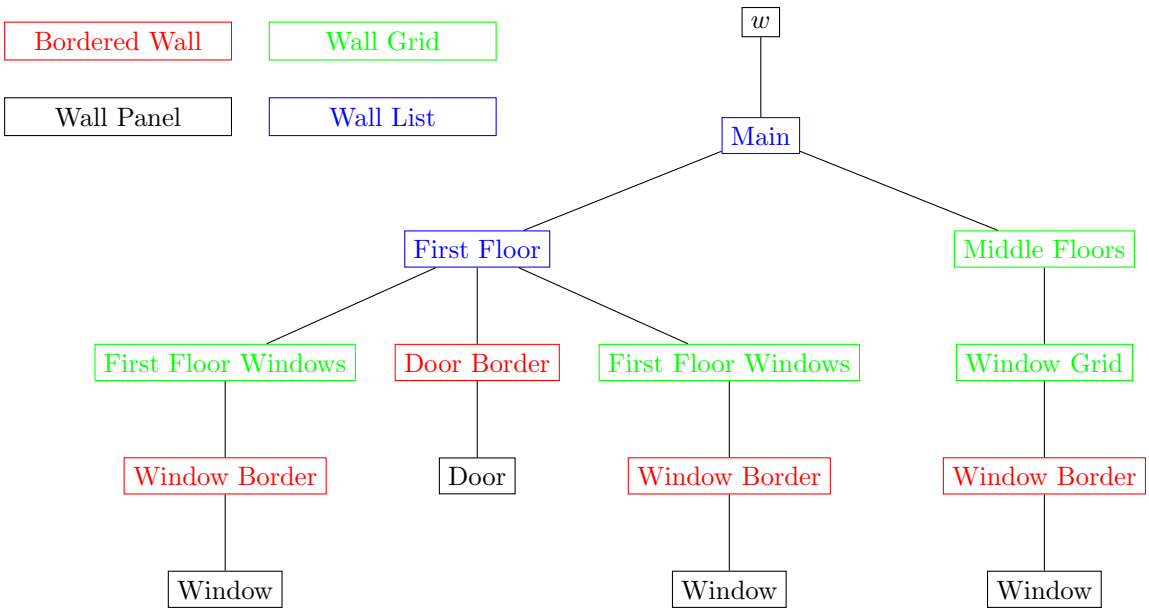

Figure 3.7: Example Building Symbol Tree from Step 1 of Generation Process

Calculations are done recursively via an overloaded `calcMinMax()` function in each `AbstractWall` subclass.

For each symbol type, min and max sizes are calculated as follows:

- Wall Panel: WP
  Minimum and maximum sizes are set by the user.
- Border Wall: $BW \rightarrow W$
  Minimum and maximum sizes are copied from parent and then shrunk to ensure the borders fit in. Depending on the resize policy, the minimum or maximum may remain unchanged. See Figure 3.2.
- Extruded Wall: $EW \rightarrow W$
  Minimum and maximum sizes are copied from the child with no change.
- Wall Grid: $WG \rightarrow W$
  Depending on the orientation, maximum height and width can be set to infinite.
  For WG the parent and W the child:
  - `Vertical`
    $WG_{min} = W_{min}$
    $WG_{max} = (W_{max}.\text{width},\ \infty)$
  - `Horizontal`
    $WG_{min} = W_{min}$
    $WG_{max} = (\infty,\ W_{max}.\text{height})$
  - `Both`
    $WG_{min} = W_{min}$
    $WG_{max} = (\infty,\ \infty)$
- Wall List: $WL \rightarrow W_1, W_2, \ldots, W_n$
  For WL the parent and W_i the children:
  - `Vertical`
    $WL_{min} = (\max_{i \in 1,2,\ldots,n}(W_{i\ min}.\text{width}),\ \sum_i(W_{i\ min}.\text{height}))$
    $WL_{max} = (\min_{i \in 1,2,\ldots,n}(W_{i\ max}.\text{width}),\ \sum_i(W_{i\ max}.\text{height}))$
  - `Horizontal`
    $WL_{min} = (\sum_i(W_{i\ min}.\text{width}),\ \max_{i \in 1,2,\ldots,n}(W_{i\ min}.\text{height}))$
    $WL_{max} = (\sum_i(W_{i\ max}.\text{width}),\ \min_{i \in 1,2,\ldots,n}(W_{i\ max}.\text{height}))$

  See Figure 3.4 for example

Step 3: Calculating the Geometry Sizes of the Wall

Every symbol in the tree needs to calculate its actual size, and is achieved by calling the `resize()` function recursively. The symbol is given a target size it must fit into, it first calculates if this is a valid size (throwing an error if not), then calculates the target size(s) for its child(ren). If valid, the actual size of the symbol is set to the target size. After this, the resize function is called on its child(ren). The root node is given its target size from the user and represents the final size of the wall.

How this is done for each symbol type is as follows (associated `resize()` psudo-code can be found in Appendix D [2]):

[2] There are many cases for different parameters, the code shown is a subset of total functionality.

- Wall Panel: WP
  Actual size given to it by parent node. No children to resize.
- Border Wall: $BW \rightarrow W$
  Child wall's target size is set its target size, shrunk to adjust for the borders
- Extruded Wall: $EW \rightarrow W$
  Child wall's target size is set to its actual size.
- Wall Grid: $WG \rightarrow W$ Child wall's target size is set fraction of the parent wall's actual size, depending on how many times it can be tiled.
- Wall List: $WL \rightarrow W_1, W_2, ..., W_n$
  For a vertical orientation, children are allocated all the same width and allocated height based on their minimum, maximum and preferred heights. Vice-versa for a horizontal orientation.

Step 4: Extracting the 3D Geometries of the wall

With the sizes calculated, all that remains is to calculate the actual 2.5D GeometryDescriptions that will make up the final wall. This is done in by recursively calling the `getGeometries()` function. `getGeometries()` returns a list of geometries in the following manner, depending on the symbol type:

- Wall Panel: WP
  Returns a 2.5D textured rectangle (a GeometryDescription object) based on its actual size and totalDepth. Or possibly a 3D pregenerated model (this was not implemented in this system, but was done so in the original research [9]).
- Border Wall: $BW \rightarrow W$
  Returns a number of GeometryDescriptions representing the borders, as well as its child's list of geometries.
- Extruded Wall: $EW \rightarrow W$
  Returns its child's list of geometries, except each child's totalDepth variable is adjusted by EW's depth variable.
- Wall Grid: $WG \rightarrow W$
  Takes its child's list of geometries and depending on how many times the child is tiled in the Vertical and Horizontal direction, a list containing a number of offset copies of the child's list is returned.
- Wall List: $WL \rightarrow W_1, W_2, ..., W_n$
  Returns a list of all its children's geometries.

Figure 3.8 shows a building generated using the same set of production rules (but with slightly different parameters).

3.3 Extensions to the Method

As seen in the previous section, the Wall Grammar method produces a number of GeometryDescriptions to build up a final 2.5D façade. If run through a physics simulator this produces a number of disjointed 'coarse-grained' or 'coarse-detail' blocks. In order to make a realistically destructible building we need to: 1) split these coarsely detailed blocks up into smaller 'fine-grained' or 'fine-detail' blocks, and then

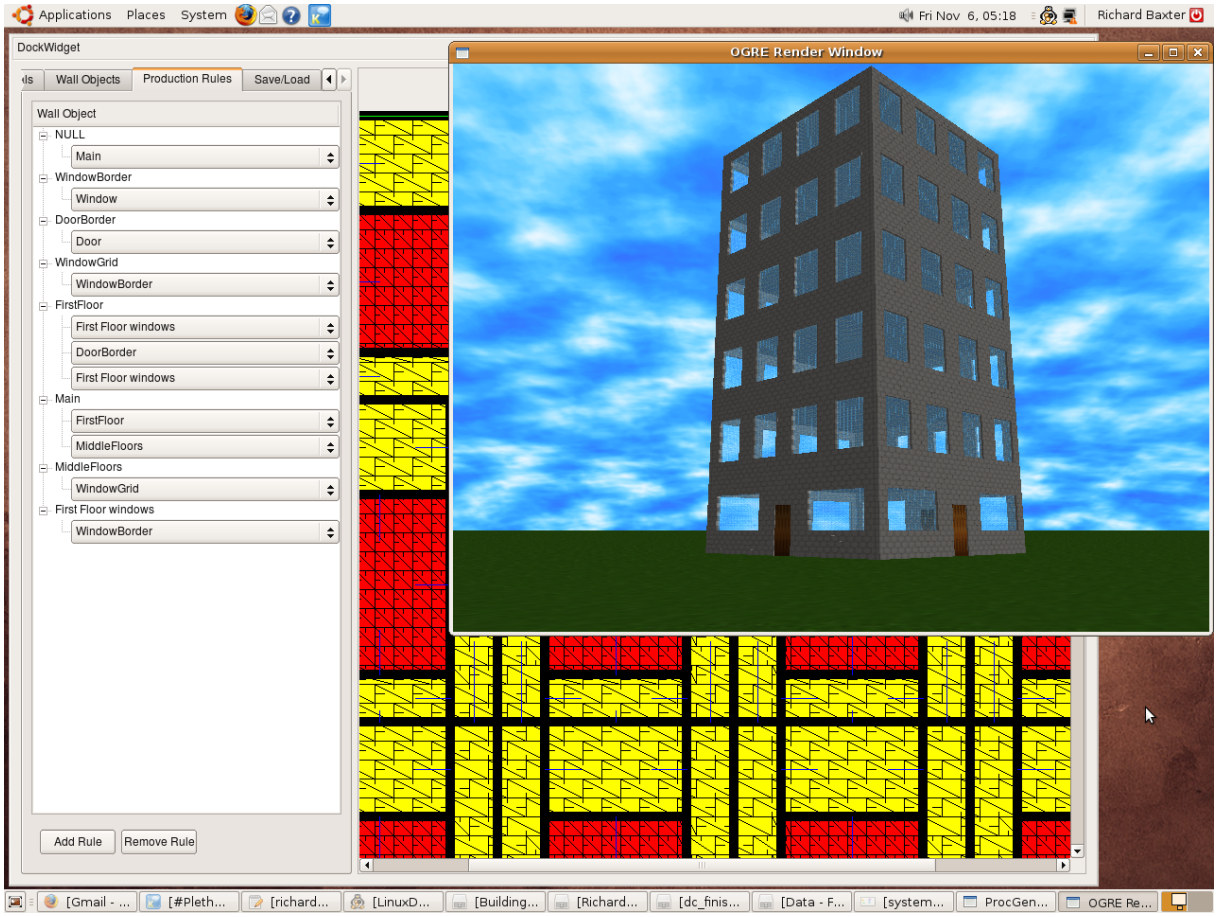

Figure 3.8: Example Building Generated using the System

2) join these fine blocks together in a reasonable manner so that the final building does not simply fall apart when put into a physics simulator or game environment. Splitting into finer detail blocks mimics the complexity of an actual building that is build up from many small bricks or components. Joining them together acts as the mortar between these bricks that keep the wall stable.

3.3.1 Additions and Modifications to the Wall Grammar

As a result of these extra requirements, additional variables are needed in the grammar's symbols. The Texture variable, as present in some of the symbols, is replaced with a Material variable. A Material holds wall thickness, a splitting rule (e.g. split into staggered bricks, orthogonally aligned windows, long strips, see section 3.3.5) for splitting the GeometryDescriptions into fine-detail GeometryDescriptions and the Texture of the fine-detail blocks.

fine-detail blocks are essentially the same as coarse-detail GeometryDescriptions: they contain a size, depth, Texture as well as a list of its neighbours [3].

In order to represent the different heights at which the levels of the building should be generated, the WallList and WallGrid symbols also contain a Boolean variable stating whether their GeometryDescriptions represent a section of the façade that are levels or stories of the building. This is done only in the WallList and WallGrid symbols as they are the only symbols that produce a number of child wall sections Vertically. BorderWall, ExtrudedWall and WallPanel can only produce a part of a story in any reasonably constructed wall.

3.3.2 Overview of Algorithm

At its most basic and unoptimised, the extended algorithm is as follows:

1. Basic Generation:
   Produce the 'coarse-detail' GeometryDescription, as outlined in the section 3.2.4.
2. Floor Height Extraction:
   Produce a list of heights representing the heights of the different stories by traversing the symbol tree and accumulating a list of heights from the appropriate WallList and WallGrid symbols.
3. Fine Detail Tiling:
   For each coarse-detail GeometryDescription, split it into smaller 'fine-detail' block. How this is done depends on the Material of the coarse-detail GeometryDescription (see section 3.3.3).
4. Fine Detail Joining:
   For each fine-detail block, calculate what other fine-detail blocks are touching/neighbouring it to the left, right, top, and bottom and the extent to which they overlap.
5. Façade Generation:
   When generating the actual building in 3D simply create the fine-detail blocks as generated above, join them with a certain breaking force [4] depending on who their neighbours are and how much they overlap (i.e. a brick that only overlaps over part of its edge should only receive partial breaking force between the two).

[3] More accurately: a list of neighbours each with an associated overlap distance. This is explained in more detail in section 3.3.2.

[4] 'breaking force' is the force that the joint can withstand before breaking. This is implemented with the PhysX *FixedJoint* in the system. See section 3.4.1.

6. Façade Stitching:
   The previous step is done for each façcade. These façcades are then joined together using a similar method to that described in 3.3.3. This creates the building's exterior.

7. Floor Generation and Stitching:
   Generate the internal floors of the various levels and join them to the appropriate 3D geometries in the generated building exterior. This can either be one single slab as a floor, or a grid of smaller blocks joined together and then to the building itself. [5]

A problem with this naïve approach is that to calculate neighbouring blocks (in step 4) is $O(N^2)$. There is also the possibility of floating point inaccuracies, leading to missing joints. In order to speed up the process, information on each block's neighbours is calculated for each fine-detail block inside the generation process, instead of afterwards.

It is useful to calculate the neighbours for the coarse-detail GeometryDescriptions first before splitting them up into smaller blocks. Only after this will the fine-detail blocks' neighbours list be calculated.

Each GeometryDescription/block is extended with a list of `JointDescription`'s, one `JointDescription` contains a pointer to another GeometryDescription/block that neighbours it, which direction it lies (Up, Down, Left or Right), as well as how much they overlap. It also contains a variable representing what sides of the GeometryDescription/block is 'open' or has no neighbours (e.g., a block can have its top and left sides open/unjointed).

With these modifications, the extended algorithm is as follows:

1. Basic Generation:
   Produce the 'coarse-detail' GeometryDescriptions, as described in section 3.2.4. In addition, calculate each coarse-detail GeometryDescription's neighbours and open sides (see section 3.3.6 for details).

2. Floor Height Extraction:
   Produce a list of heights representing the heights of the different levels by traversing the symbol tree and accumulating a list of heights from the appropriate WallList and WallGrid symbols.

3. Fine Detail Tiling and Internal Joining:
   For each coarse-detail GeometryDescription, split that into smaller 'fine-detail' blocks. For each one of these fine-detail blocks, calculate its neighbours, as well as what sides of it are open (this can be done as the fine-detail block are tiled (see section 3.3.5.This will join together the sets of fine-detail GeometryDescriptions within each coarse-detail GeometryDescription. For each coarse-detail block, store lists of fine-detail GeometryDescriptions that are open to the left, right, top and bottom (to be used in next step).

4. Fine Detail External Joining:
   What is stored now is a number of sets of fine-detail blocks. Each set representing the fine-detail of the coarse-detail GeometryDescriptions. By using the coarse-detail GeometryDescriptions neighbour information, the fine-detail blocks of two neighbouring coarse-detail GeometryDescriptions can be joined/stitched together (e.g. if GeometryDescription A is joined to GeometryDescription B to the right, we would join the fine-detail blocks in A that are open on the right to the fine-detail blocks in B that are open to the left [6] - see section 3.3.3 for more details). See Figure 3.10 for example.

[5] in the code, each floor was a 6x6 grid of slabs, all joined to all neighbouring slabs in the up, down, left right direction, this was to mimic long solid structural reinforcement spanning the length or width of the building.

[6] This can be done quickly with the lists of open blocks to the left, right, top and bottom, generated in the previous step. One can optionally forgo the storage of the lists and rather use an exhaustive search. This is obviously slower, but uses less memory.

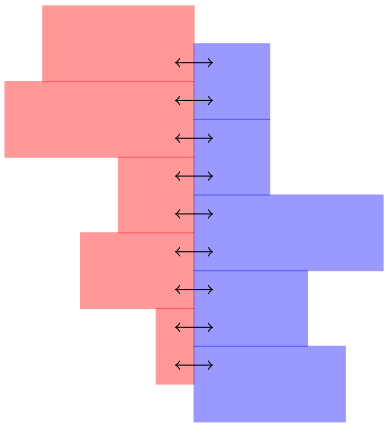

Figure 3.9: Stitching Example: red and blue rectangles are from separate sets of GeometryDescriptions. Arrows represent joints.

5. Façade Generation:
   When generating the actual building in 3D, create the fine-detail blocks as generated above, join them with a certain breaking force depending on who their neighbours are and how much they overlap [7].

6. Façade Stitching:
   The previous step is done for each wall, joining the sides of the each wall to its neighbour wall using a similar method to that described in 3.3.3. Since all fine-detail blocks will retain their openness, when joining two walls together one can, for instance, take the right-open fine-detail blocks on the one wall and join them to the left-open fine-detail blocks on its adjacent wall. One can quickly find all right-open fine-detail blocks by looking only in coarse-detail GeometryDescriptions that are right open and taking their fine-detail right-open blocks.

7. Floor Generation and Stitching:
   Generate the internal floors of the various levels and join them to the appropriate 3D geometries in the generated building. This can either be one single slab as a floor, or a grid of smaller blocks joined together and then to the building itself.

The above optimisations can be seen a space-time trade-off that stress information inherent in the generation process.

3.3.3 Stitching two sets of GeometryDescriptions Together

It is often the case that 2 sets of GeometryDescriptions, that neighbour each other on one axis (vertical or horizontal) need to be joined together. However, GeometryDescriptions within these sets might not be touching, so simply joining all elements in the one set to all elements in the other is incorrect. The way in which to do this is to either use an $O(N^2)$ algorithm to calculate which elements from one set overlap which elements from the other, or to order the sets by their x or y coordinates (depending on whether the stitch is Up | Down or Left | Right) and do a single pass of the one set, joining the elements as it goes along (see Figure 3.9 for explanation). Since nearly all of the sets of GeometryDescriptions can be generated in order anyway, the single pass method is preferred.

[7] These are joined with a PhysX *FixedJoint*. See section 3.4.1.

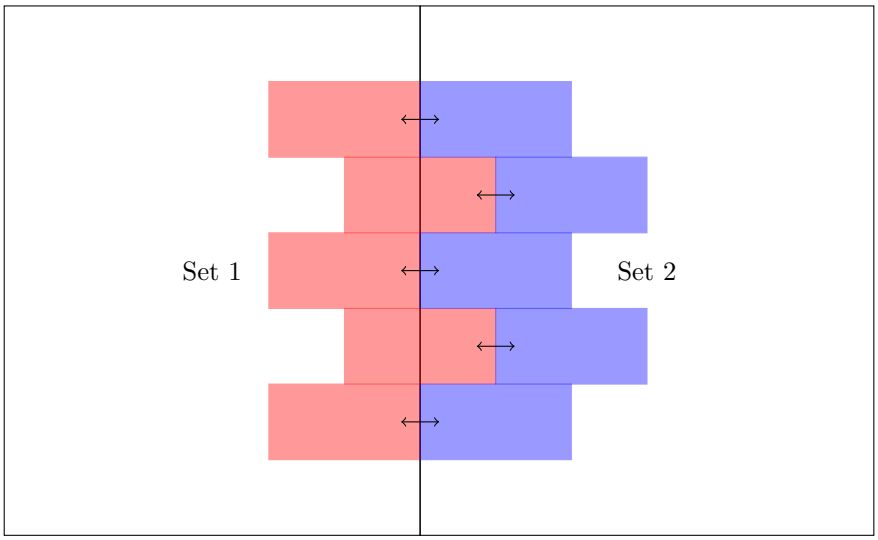

Figure 3.10: Joining Fine-Detail blocks from 2 Separate Sets. Arrows represent joints.

3.3.4 Generating/Tiling of Fine-Detail Blocks

When the GeometryDescriptions are split into fine-detail blocks, there are many different ways in which one might want to do this. For instance, to get a brick-like pattern, you would need to stagger the bricks from row to row. For a window pane you might just want a square tiling aligned in both dimensions. Also one might want long vertical or horizontal blocks to represent wooden planks. See 3.3.5 for examples.

We need a method that will tile the current geometry, and give enough context information to the neighbouring geometry such that is can continue the tiling there. This context information in addition to the tiling method must also ensure that no file detail blocks overlap.

The tiling simply layers fine-detail blocks, left to right, bottom to top, starting at the bottom left most coarse-detail GeometryDescription. As the fine-detail blocks are layered left to right, it will eventually get to the end of the coarse-detail GeometryDescription and overlap with on of its neighbours. Similarly, it might overlap over the top once the tiling gets too high. When an overlap occurs, one of the following things will happen:

1. If the neighbour has the same fine-detail tiling scheme as the originating GeometryDescription, and has not been given a starting coordinate (or has only been given a starting coordinate from rule 3), then it will be given the coordinates of the next block to be laid as its starting coordinate as well as any relevant context information (see 3.3.5 for some examples of context information). The neighbouring GeometryDescription will be added to a list of coarse-detail geometries to be tiled. This will cause the two GeometryDescriptions' sets of fine-details to mesh.
2. If the neighbour has the same fine-detail tiling scheme as the originating GeometryDescription, and has been given a starting coordinate by rule 1 already, nothing happens.
3. If the neighbour has a different fine-detail tiling scheme as the originating GeometryDescription, then it is given its bottom left most coordinate at the starting point and set to start tiling from there as if new. This starting coordinate can be overridden by rule 1 above, so that a neighbouring GeometryDescription of the same type can mesh with it. The neighbouring GeometryDescription is added to a separate list of GeometryDescriptions, that will only start tiling once all the GeometryDescriptions of the previous type tile first.

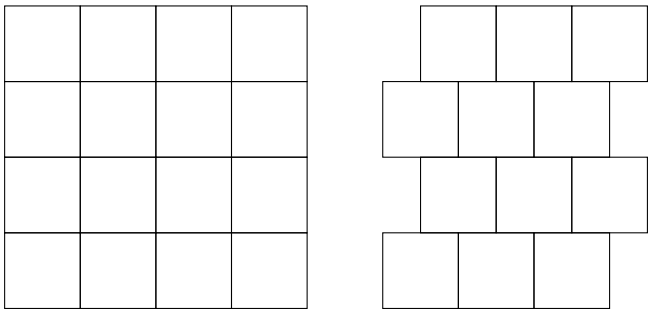

Figure 3.11: Plain Tiling Scheme (left) versus Brick Tiling Scheme (right)

One could also create a fine-detail tiling method that can break up a GeometryDescription into small psudo-random triangles to create a shattered glass effect. More generally, to create a valid tiling scheme, for each GeometryDescription, one needs a start point (where its tiling starts) and enough context information to continue the tiling without overlapping onto other coarse-detail GeometryDescriptions' fine-detail blocks.

3.3.5 Examples of Different Types of fine-detail Tiling Schemes

Plain

This a rectangular block that is layered left to right until the end of the geometry, then layered on top of that layer in the same way. There is no context information needed in order to continue the tiling.

This method can use small square blocks to create a window pane like effect or long vertical blocks to create long wooden planks (e.g. for a wooden door). See Figure 3.11 for example.

To calculate neighbouring fine-detail blocks, join the bricks left-to-right as they are tiles one after the other, and top-to-bottom with the row below.

Brick

This is a rectangular block (usually longer than higher) that is layered left to right until the end of the geometry, then layered on top of that layer, but with a half width offset. After that row is done, it is layered again like the first row, and so on. The context information needed to continue this is simply whether the row is staggered or not. See Figure 3.11 for example.

To calculate neighbouring fine-detail blocks, join the bricks left-to-right as they are tiles one after the other, and top-to-bottom with the row below. Since they pattern is staggered, there will be two neighbours to the top and bottom of each brick.

Window Shards

This was not implemented in the code, but is implementable. The method would involve laying down triangular geometries in a similar fashion to Plain or Brick, except that the edges will be psudo-randomly perturbed to give it a random look. The psudo-random number generator will actually be a function from $(x, y) \in N^2 \rightarrow (q, r \in R^2)$ to give a a unique protuberance to each triangle vertex).

Context information would simply be what 2D index of triangle it is generating at the starting point. Thereby the new GeometryDescription can start tiling triangles and perturb them in the same manner as the previous GeometryDescription had.

3.3.6 Calculating coarse-detail GeometryDescription's Neighbour Lists

During the stage of generation when the geometry is extracted (Step 4 in 3.2.4) the neighbours list can be calculated for each block inside the `getGeometries()` function. Take, for instance, a Vertical WallList that contains 2 children. Each child contains a set of GeometryDescriptions. The set of the top child must be connected downwards to the set of the bottom child. This can be done by joining all the children in the top set that are open-bottom to the children in the bottom set that are open-top. How these are joined is explained in section 3.3.3.

So, this means that for the different symbol types, for each set of GeometryDescriptions generated by the `getGeometries()` function, that symbol's GeometryDescriptions must contain what sides of theirs is open, so that the `getGeometries()` function called by the parents can stitch them together.

How the openness is calculated for each type is as follows:

- Wall Panel: WP
  Since only one GeometryDescription is generated, and there are no child symbols, the GeometryDescription is open on all sides.
- Border Wall: $BW \rightarrow W$
  This generates up to 4 borders. There are a number of cases of openness, two of these are shown in Figure 3.12. It is possible that the child's set of GeometryDescriptions are on one or more of the edges already, in which case they just retain their openness in those directions.
- Extruded Wall: $EW \rightarrow W$
  Since the wall does not add any GeometryDescriptions, the child set of GeometryDescriptions retain all their openesses.
- Wall Grid: $WG \rightarrow W$
  The child symbol's set of GeometryDescriptions are copied and tiled. All the copies that are on the edges, retain their openness in those directions. The internal copies loose all openness when they are joined to their neighboring copies. See Figure neighCalcGridExample.
- Wall List: $WL \rightarrow W_1, W_2, ..., W_n$
  Works in exactly the same way that a Wall Grid would work if it were tile Vertically or Horizontally. If the Wall List is Vertical, for instance, the first and last child walls will retain their top and bottom openesses (respectively). All child walls will retain their left and right openesses. Similarly for a Horizontal list, except the first and last children keep their left and right openesses and all children keep their top and bottom openesses.

How the child sets of GeometryDescriptions are stitched together for each type is as follows:

- Wall Panel: WP
  No child sets to stitch together.
- Border Wall: $BW \rightarrow W$
  The child's set of GeometryDescriptions are joined to the appropriate border. So, for instance, the

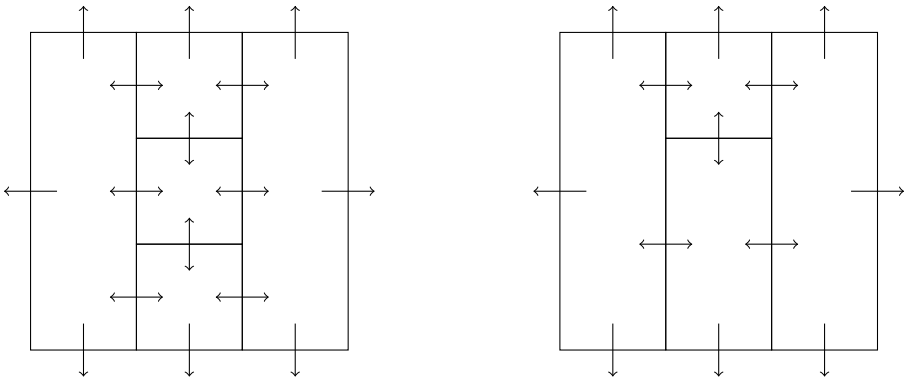

Figure 3.12: Examples of Calculations of Neighbours List for Border Wall Type: Double sided arrows represent regions that are to be stitched. Single sided arrows represent open sides

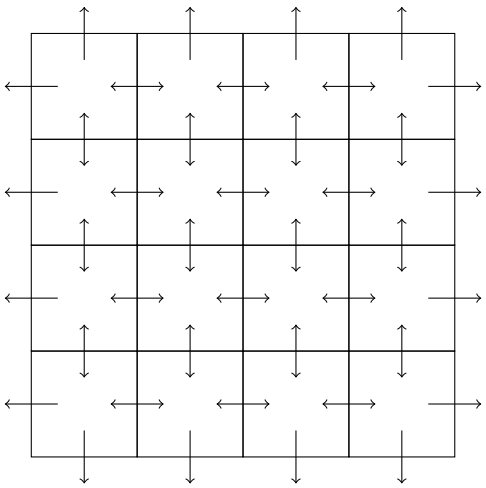

Figure 3.13: Examples of Calculations of Neighbours List for Wall Grid Types: Double sided arrows represent regions that are to be stitched. Single sided arrows represent open sides

top-open GeometryDescriptions of the child's set of GeometryDescriptions will be joined to the top border. Similarly for other directions. If a border does not exist, the child's set of GeometryDescriptions retain their openness in that direction. See Figure 3.12.

- Extruded Wall: $EW \rightarrow W$
  There is only one child and no other GeometryDescription, no stitching occurs.

- Wall Grid: $WG \rightarrow W$
  The child symbol's set of GeometryDescriptions are copied and tiled. If two sets are next to each other horizontally, the right-open GeometryDescriptions in the left set are stitched together with the left-open GeometryDescriptions in the right set. Similarly for set tiled next to each other vertically. See Figure 3.13.

- Wall List: $WL \rightarrow W_1, W_2, ..., W_n$
  Similar to the Wall Grid, the children are stitched together horizontally or vertically depending on the orientation.

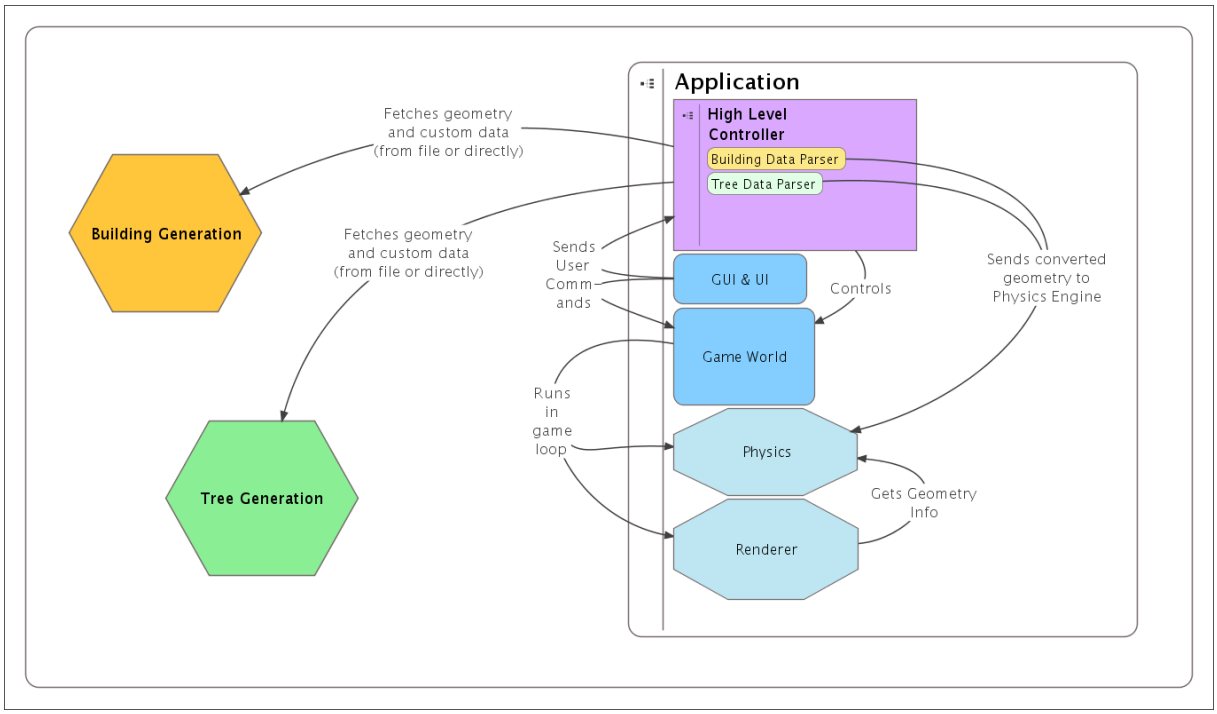

Figure 3.14: System Diagram

3.4 System

Figure 3.14 shows all the components of the system.

3.4.1 Physics: PhysX

PhysX is nVidia's proprietary physics simulation software designed for gaming environments [4]. All building geometry objects were created as PhysX boxes and all simulations were rigid-body only, this means there was no fluid simulations present and all geometric objects were simulated with Newtonian physics only. Since this is a gaming physics engine, extremely high levels of physical accuracy is substituted for greater speed, as is normal for physics engines intended for gaming.

FixedJoints

These boxes were joined together using PhysX *FixJoint*s, these offer a way in which to tightly 'connect' two PhysX objects together so that they essentially become one. A FixJoint can be assigned a certain breaking value, so that if the joint experiences a force on it greater than its breaking value, it will be immediately destroyed.

It was thought that a FixedJoint would offer sufficient rigidity, however, it was found to have a very slight propensity to flex. This multiplied over many layers of bricks, caused wobbling artifacts (discussed in more detail in Section 5.1.3). It is still to be determined the exact reason for this as the source code for PhysX is not freely available.

However, it is suspected that the slight wobble is caused by the way in which most physics simulators maintain geometries between joints, which is by adding an opposing force to reposition it. This could causes a slight flex. Further investigation would be required.

3.4.2 Rendering: OGRE

The Open-source Graphics Rendering Engine or OGRE [3] , was the engine used to do all game world rendering. For the buildings, all geometric objects were represented as 3D rectangular blocks. For each PhysX object that represented part of a building (or tree), there was a corresponding ORGE object that was used to render it graphically. Since each GeometryDescription created in the building generation process lead to the creation of a PhysX object, it also had a corresponding OGRE object. The texture in the GeometryDescription's Material variable was used to texture it. Other rendering objects included a textured sky-box and ground plane, as well as wire-frame boxes representing difference wind forces (used solely in the trees section of this project).

3.4.3 Game Engine and UI: Qt

The game engine, game UI, and general GUI were all done using the Qt toolkit [5].

The game engine started the OGRE rendering contexts, initiated PhysX and ran the main logic of the game world. It ran on a separate thread to the GUI and endlessly ran the game-loop until the program exited.

The GUI consisted of a viewing area and 3 tabs for creating symbols, creating the associated production rules, and saving and loading buildings. The viewing area would automatically generate and display the building the production rules described when a parameter was changed. Figure 3.15 shows some screen-shots of the interface.

The building generation methods were called by the GUI thread when generating the display image in the interface. When the game-logic required the building it was generated in the game-loop thread.

3.4.4 Building Generation

This module consisted of one main function `generateBuilding()` which took as arguments a list of symbols and production rules and returned a list of GeometryDescriptions using the method outlined above.

3.4.5 Tree Generation

This is the module that generates procedurally generated trees and is the other module of this project, not dealt with in this report.

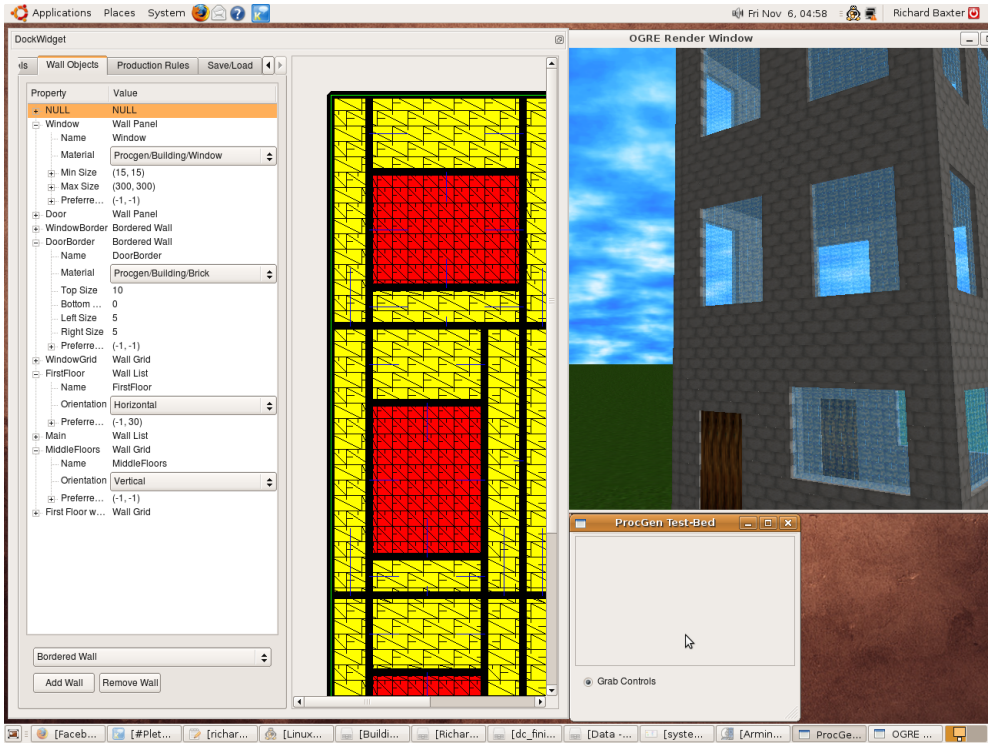

Figure 3.15: System Interface for Building Generation

Chapter 4

Testing and Experimentation

4.1 User Testing

4.1.1 Motivation

In order to ensure the system answers the research question, tests must be set up to validate that the building generation method is non-trivial and can produce results that hold to at least a basic level of 'realism'. Since realism is a subjective user experience, user testing in mandatory. A common way to quantify user perceptions is via a questionnaire, conducted over a sufficiently large and diverse experimental group.

4.1.2 Hypothesis and Characteristics

Hypothesis Do the simulated buildings that break and fall down, do so in a realistic manner? With realistic being defined in terms of "how well it passes for real in a game environment" as well as (in a separate question) "how well it passes for real in the real world".

The experiment was designed so users viewed the videos of what the system produces. They had no prior experience with the system nor had they seen the videos before. Each experiment was held in the Honours Lab, on the same computer under similar lighting and noise conditions. Twenty users were tested to allow a sufficiently large sample size for statistical purposes.

The users' experience with games and gaming environments were determined as this might have effect on quantitative results. Getting a diverse range of users in terms of game experience was also preferable, as was an even split in gender. However, a significant majority were male computer scientists with fairly substantial video game experience.

Interaction with the experimenter was kept to a minimum during the test in order to reduce the affect of extraneous variables on the quantitative results. After testing was complete, discussion of the system was occurred in most cases.

Users were asked to view 3 different sets of videos, each containing different sizes of buildings: small 2 story shed-like building, medium-sized 3 story buildings and large wide 6 story buildings. (Figures 4.1 and 4.2)

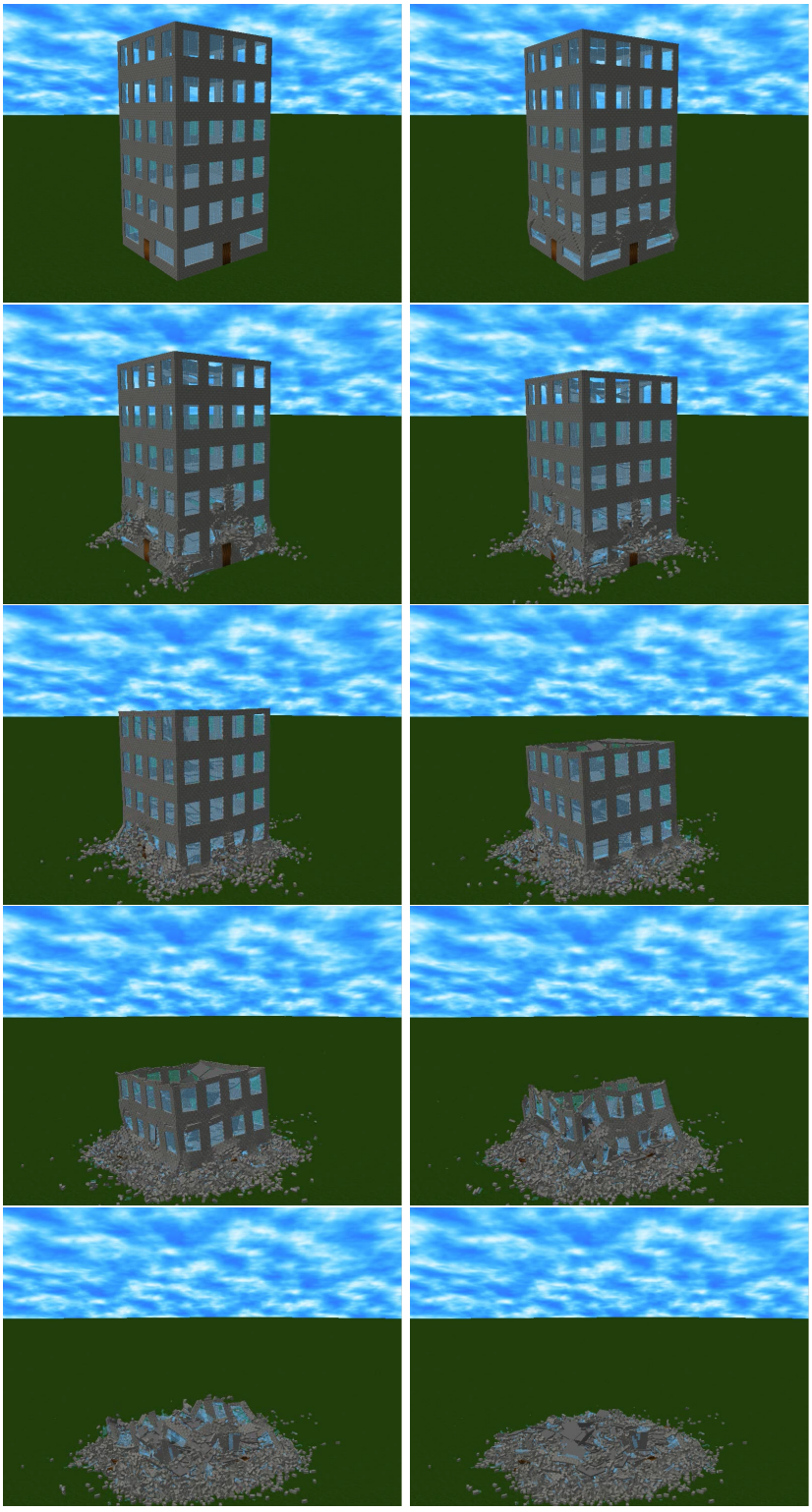

Figure 4.1: Large Building Collapse Example

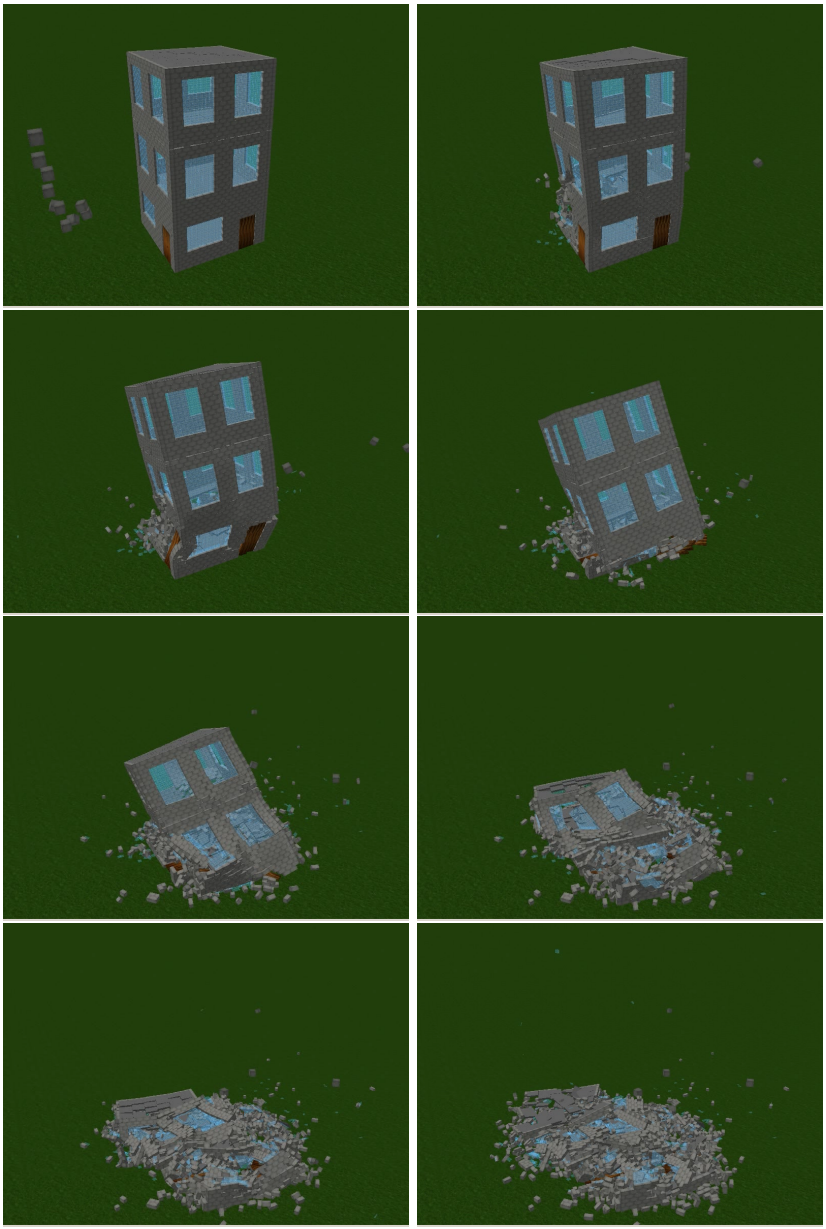

Figure 4.2: Medium-Sized Building Collapse Example

4.1.3 General Overview of a Single Experiment

Documentation

The experiment was split into two sections, one for the trees subsystem, and one for the buildings subsystem. All test sheets are included in Appendix A. The user received each of these pages at some stage of the test (explained in next section). Each set of pages is as follows:

- Preparation/Intro Document:
  This document contains general information about the experiment to follow and initial instructions.
- User Information:
  This document contains questions asking for general information about the user and their gaming experience.
- Scenario Descriptions:
  A reference document with single sentence descriptions of the video sets
- Building Experiment Documents:
  - Buildings Section Instructions:
    These are the specific step-by-step instructions for how the user should carry out the buildings section of the experiments
  - Buildings Section Answer Sheets:
    3 identical pages, each with questions referring to a specific set of videos (2 quantitative questions and 2 qualitative questions and 1 question asking the user for any general comments they may have); followed by 2 pages containing 4 questions about all the sets of videos.
- Tree Experiment Documents:
  - Trees Section Instructions:
    These are the specific step-by-step instructions for how the user should carry out the trees section of the experiment
  - Trees Section Answer Sheets:
    7 identical pages, each with questions referring to a specific set of videos (2 quantitative questions and 2 qualitative questions and 1 question asking the user for any comments they may have). Followed by 2 pages containing 4 questions about all the sets of videos.

Experiment Procedure

Users completed the questionnaires in two parts, one part being the buildings section, the other being the trees section. Ten users did the building section first and ten did the trees section first. The experiment ran as follows:

1. A user receives the 'Preparation/Intro Document' and the 'User Information' and is asked to read the Intro Document. The introduction document asks the user to fill out the 'User Information' document.

2. After reading this the user is given either the 'Building Experiment Documents' or 'Tree Experiment Documents' depending on experiment order, as well as the 'Scenario Descriptions'. The instructions tell the user exactly what to do.
3. There are 3 scenarios for the buildings and 7 for the trees, and the order in which these were completed was random for each test.
4. After the answer sheets are filled out, the user gives the answer sheets back and receives the second set of documents (Trees or Buildings).
5. The user is thanked for their time and input. Time permitting, and if the user is willing, they aer allowed to interact with the program and give additional commentary.
6. The user is given R20 for their time and asked to fill out a receipt.

4.1.4 Heuristic Evaluation

The ideal validation of the system would be a full physical evaluation of the building collapse with use of mechanical engineering properties. However, such information is not easily available and of too technical a level to be used in this project given the time constraints. Instead we performed a Heuristic Evaluation of the collapsing process. This was achieved by analysing multiple videos of buildings collapsing, and taking note of what features are found in such videos.

These features fall into two categories: physical and non-physical. A physical feature is something that is present in the real-life videos, and hopefully the simulation videos as well (although it might not be). A non-physical feature is one that is not present in the real-life videos, and hopefully not present in the simulation videos (though it might be). A sufficiently high number of physical features and a corresponding low number of non-physical features in the simulation videos could be grounds for positive overall results.

Initially, during development, features were extracted from real life videos in order to guide the physics simulation and generation processes. A list of features will be validated and expanded upon in qualitative user testing (see section 5.1.3).

4.2 Performance Testing

4.2.1 Testing Equipment and Software

All performance testing was undertaken on a machine with the following specifications:

- Nehalem Bloomfield Core i7-950, quad-core hyper-threaded at 3.067GHz
- 4GB DDR3-1066Mhz RAM
- Ubuntu 9.04 32-bit
- Compilation with gcc 4.3.3 using cmake (`-O2` optimisation and no debugging flags used)
- For timing, functionality in the `sys/time.h` class is used, all results are measured in nanoseconds or frames-per-second (FPS).

4.2.2 PhysX

GPGPU vs. CPU

Currently, PhysX (v2.8.1) only has GPGPU support in Windows and not Linux. To test PhysX, and hence the system, on a GPU, one needs to run the program in a Windows OS. Although all software used to develop this system is cross platform, there were too many issues involved in compiling in a Windows environment. Thus, testing was executed using the CPU alone.

Collision Detection and Partial Physics Simulation

Testing was done using PhysX's naïve collision detection. No independent collision detection system was implemented as is was out of scope of a project of this size.

The option to 'sleep' the buildings [1], thereby reducing the strain on the physics simulation was considered and implemented. Eventually, sleeping objects were taken out as it offered little real advantage.

Preliminary testing revealed that because all building geometries were so tightly packed, it was a non-trivial task to cause a section of the building to undergo simulation without 'waking' the rest of the building. In its current form, the only way in which to perform physics simulation on only part of the building is to manually set what sections are breakable and put the rest permanently to sleep. This is deferred to future work, since it does not influence the main research questions of the project.

4.2.3 Performance Metrics

Frame Rate

In order for the system to be viable for games, the physics simulation needs to execute at a sufficiently high frame rate. However, when considering collapsible buildings for use in movies or pre-rendered media, the frame rate need not be as high. Still, if the frame rate is too low and it takes too long to render a collapsing building, then it is less useful as it would take too long for a user to notice and edit the building collapse to be practicable.

Frame rates are split into these categories:

- > 30 FPS: Real time Frame Rate - can be used without any noticeable slowdown.
- $10 - 30$ FPS: Interactive Frame Rate - can be used for editing purposes, but may be jerky or slow.
- < 10 FPS: Non Interactive Frame Rate - can only be pre-rendered as it is too slow to even use for editing.

There are 3 main scenarios that will be considered:

1. With Physics before building is impacted
   This is with full physics on but with no other interaction.

[1] 'sleeping' an object in PhysX causes it not to move or undergo extra collision checks until such time as another object contacts it. This can reduce total simulation time if done properly.

2. With Physics after building is impacted
   This is with full physics on after the building has been hit by external objects (approximately 5-10 objects). The objects impacting the building will usually cause it to start collapsing.
   As a result of lack of internal structure for larger buildings, in some cases the building will collapse without necessarily being hit (see Section 5.1.3)

3. Without Physics
   This condition applies while the building is being created using the production rules and view the building without any physics and is more dependant on rendering time of the scene. This allows a designer to navigate around the scene and make changes.

Simulation times

Directly related to the frame rate above, the simulation time considers the time taken for a single simulation step. This eliminates any system overhead that might distort the results and removes rendering time as a factor. Results are measured in nanoseconds and frame rates are calculated from the simulation time (frame rate = 1sec/simulation time). Due to the lack of system overhead and rendering time, results will most likely be slightly higher than when tested in the full environment.

Results will be of the form of time vs. number of fine-detail geometric entities.

Generation and Placement times

Consideration is given to how long it takes to generate a single building. This does not include time to create the geometries in the physics simulator, rather it is the time taken to convert the production rules into a list of GeometryDescriptions.

Anything under a few seconds is acceptable. Faster times will allow functionality such as real time modification of parameters by the user, instead of changing parameters then regenerating. In a similar fashion to the Frame Rate section above > 30, $10 - 30$ and < 10 FPS will be the categories.

The system is only designed as a prototype to test the method and not as an editor, so real time parameter changing was not directly implemented. As a result, frame rates are estimated from generation times (frame rate = 1 sec/generation time). The overhead of the system is not considered and results will most likely be slightly higher than if tested in a proper editing environment.

Results will be of the form of time vs. sizes of buildings and time vs. number of fine-detail geometries generated.

Results for how long it takes to place those geometries in the simulator and join them are also included. There are 4 walls per building and a number of floors. We expect the time to place and join all these should be near $O(N)$ for N geometries generated. This is because the only other factors are number of levels and number of joints. Geometries, on average, do not increase the number of joints they have as total number of geometries increase. Similarly for the number of levels there is usually approximately the same number of geometries per level. The number of geometries on the bottom and top floors might be slightly different than in a middle floor, but these differences are amortized as the number of middle floors grows larger.

Chapter 5

Results and Analysis

5.1 User Testing

5.1.1 Quantitative Results

Results fall into 3 subcategories for each of the 3 scenarios (large buildings, medium-sized buildings and small buildings). For each, the user was asked to rate how realistic they felt the videos were, compared to real life and how realistic they felt the videos were considered in the context of a video game (see Section 4.1.3 for experiment set up and Appendix A for all relevant questions).

As seen in Figure 5.1 (also Table C) the results for the various types of buildings fall largely in the 5-6/7 range. In the questionnaires, 4 was "Equal amount of realistic and unrealistic elements" with 5,6,7 being varying degrees of realism above this 'middle mark'. This shows that, for the most part, users felt the simulation were more realistic than unrealistic in the way they collapsed.

Users showed a higher rating for video games than real life. Since the questions were side-by-side, this was to be expected.

The question about overall impressions was given a high score (Figure 5.2), with no one providing scores lower than 5/10 (5 being "somewhat realistic but had noticeably unrealistic elements"). Similarly, the question on whether the simulations were an improvement on what is seen in existing games, were never scored less than 5 (5 being "no improvement") and a score of 5 only occurred twice out of the 20 participants. This reflects that most users felt that there was some improvement if these simulations were to appear in a gaming environment.

5.1.2 Qualitative Results

It is also interesting to note that whilst the impressions seemed positive quantitatively, qualitative results show that users mainly pointed out what they felt was wrong with the simulations and hardly ever made note of what they felt was realistic. They often mentioned that they found the simulation realistic, but never qualifying what exactly it was they found realistic.

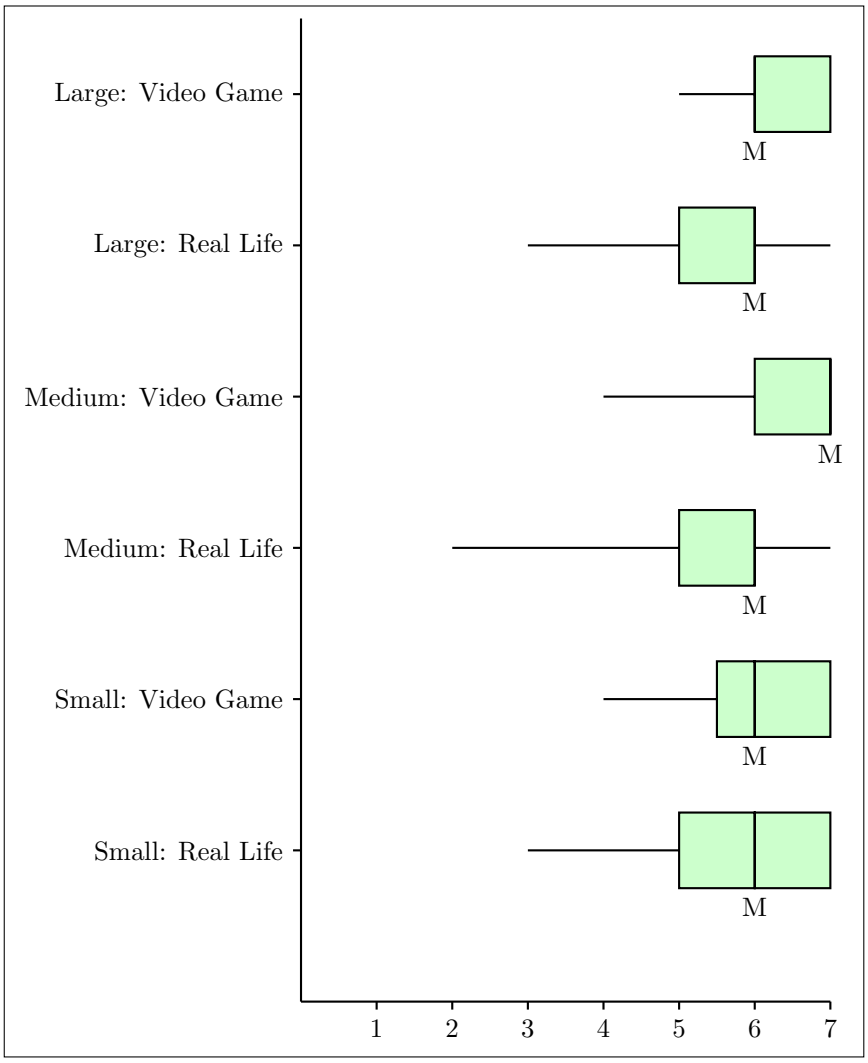

Figure 5.1: Box and Whiskers Plot of Scenario Specific Results

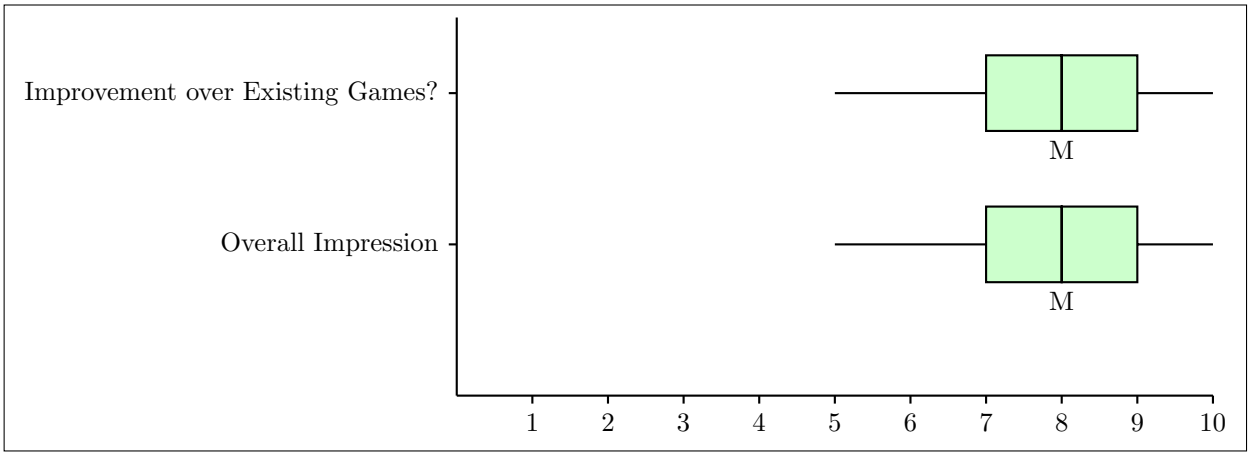

Figure 5.2: Box and Whiskers Plot of Scenario Specific Results

	Number of times stated in:				
Feature Noticed	Small Buildings	Medium Buildings	Large Buildings	General Commentary	Suggested Improvements
Lack of Dust	8	8	10	3	10
Rubber-like walls	12	3	3	4	2
Rubble not settling after collapse	0	4	1	2	0
Floating and Spinning Rubble	8	7	11	2	2
Buildings Auto-collapsing	0	1	5	0	0
Rooves Auto-collapsing	0	0	6	0	0
Unrealistic glass	1	2	8	0	6
Bricks not breaking up into smaller pieces	1	2	2	2	4

Table 5.1: Heuristic Evaluation Validation from User Testing: Count of number of times features were stated. Each feature is explained and analysed in more detail in Section 5.1.3 and 5.1.4.

5.1.3 Heuristic Evaluation Validation

As mentioned in the qualitative results section above, users pointed out vastly more features they felt were unrealistic than features they thought were realistic. As a result of this, not many of the physical features in the heuristic evaluation section (4.1.4) could be validated. However, many non-physical features could be validated that reveal flaws with the system. Many of them, unfortunately, are artifacts of PhysX and how it was employed. Table 5.1 shows how many times and in what sections each feature was stated.

Main list of Features Noticed by Users

- Lack of Dust
  In the building videos, when buildings collapsed, dust and smoke billowed from the crash zone. This was not present in the simulations. Many users noted that the dust obscured much of the fine-detail in the real life videos. Some also noted that the ability to see fine detail felt unrealistic. Dust and smoke effects fall more naturally into a graphics effect rather than a physics effect, and so was not considered in this research.

- Rubber-like walls
  In order to join the bricks together in a realistic fashion, the PhysX *FixedJoint* is used. This is sufficient to keep the structure stable in most cases but, unfortunately, can never make it rigid enough. Increasing the breaking force of the joint makes the joints more rigid and realistic, but causes the wall to be relatively unbreakable. This is part of a larger problem of structural realism, discussed in more detail under Section 5.1.4. A balance has to be struck between rigidity sufficient to ensure a stable structure and rigidity low enough, such that it will still break apart. At this point, the building still suffers from a slight wobble, which gave it a rubbery or wobbling appearance. This was thought to be slight, but user testing has revealed it to be very distracting.

- Rubble not settling after collapse
  Due to the large weight of the buildings and the way PhysX does its simulations, when the building collapses on itself, it causes a massive downward force on the bricks. In real life the bricks would compress or break into smaller fragments, in a physics simulation they do not compress and have a slight bounce. This is normally not noticeable for a limited number of geometries, but due to the vast number present in the building, a noticeable spring action occurs as the bricks all bounce back

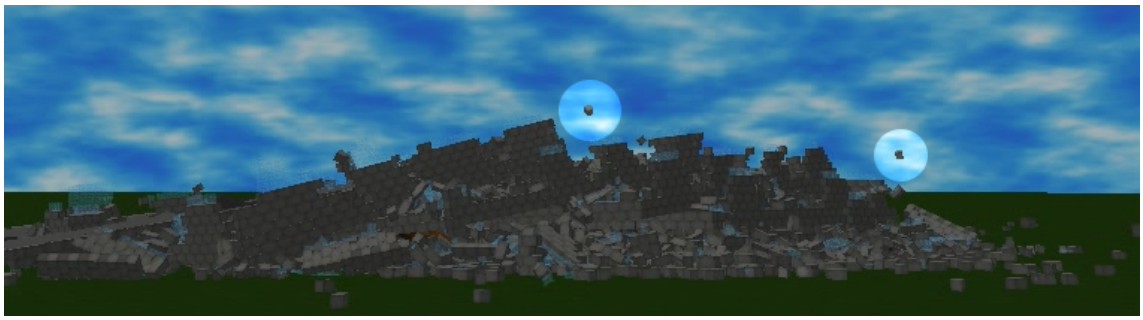

Figure 5.3: Rubble floating in air after building has collapsed completely

up. One user notes that "...the rubble behaves more like a liquid than a solid (it's very 'flowy')", another stating that "Attention to be paid AFTER collapse - in terms of individual stray bricks ... and 'thud' of building floors - too much reverberation"

- Floating and Spinning Rubble (Figure 5.3)
  This artifact was only picked up on during testing and could not be fixed in time. Some of the bricks would not break their joints when hit, since some of the bricks were joined to other bricks and floor tiles but not actually touching each other, it caused some of the geometry to flutter in the air even after the building had collapsed. This only occurs a few times in the collapsing buildings but was nonetheless very distracting to nearly all of the users.

- Buildings Auto-collapsing
  This is a significant effect of one of the major drawbacks of simulating a building, namely that with greater physical realism of the simulation, greater realism in the building structure is required. This is discussed more in Section 5.1.4. For smaller buildings, the breaking force of the brick was sufficient for the bricks near the bottom of the building to support the weight of the entire building. However, as the building grow, the breaking force of the bricks becomes insufficient, causing the building to begin collapsing immediately. It would usually buckle outwards at the base of the building, causing the rest of the building to start falling, ultimately destroying the building. Users found the collapse itself realistic, but noticed that the collapse was not due to any external forces.

- Rooves Auto-collapsing (Figure 5.4)
  As with the Auto-collapsing of the entire building, some floors (noticeably the roof) would collapse in on themselves with little or no additional force. The floors on larger buildings are larger in surface area, and with no additional internal structure to support larger floors, as would be found in real life, they sometimes collapsed immediately.

- Unrealistic glass
  As with the rubber like walls, glass panes suffered from a certain rubber like effect, whereby they would sway after collapsing. Users stated that they expected a more shattered like effect. This was due to glass panes being attached to each other with PhysX FixJoints and similar balancing issues, similar to the rubber-like walls case, between rigidity and rubber like effects were present. Also, it was noted that many users discerned the square pattern of the window pieces, noting that they thought a more random effect would be more realistic.

- Bricks not breaking up into smaller pieces
  Some users noted that the bricks remained whole, even after they had fallen a great distance. Users expected the bricks to break into smaller fragments upon collapse, finding that the whole brick surviving intact seemed unrealistic.

Figure 5.4: Roof collapsing in on itself before it has been impacted

Other Features Noticed

This is a list of features that were noticed only once or twice in the user tests.

- Buildings Collapsing Sideways
  One of the users noted that some of the buildings collapsed straight down and not sideways. The user stated that "...it [the sideways movement] was quite significant in the real videos...". This particular user noted this for the medium-sized buildings. Another user noted that the sideways movement was noticeable for the large buildings, but that this was "counterintuituve" for a building undergoing a demolition like collapse. The user did state, however, that the building falling over sideways as a result of foreign objects impacting it was acceptable.

- Building falling too fast
  One user noted that the buildings seemed to fall too fast compared to the real life videos. Slowing down the simulation rate slightly or up-scaling the buildings might alleviate this, but this remains to be tested as a solution.

- Partial collapse of top floors (Figure 5.5)
  Some of the videos, particularly the smaller to medium sizes buildings, showed a partial collapse, during which the top one or two stories of the building were impacted and collapsed but the rest of the building remained upright. There were one or two very positive comments on the realism of these situations. Unfortunately, due to the lack of internal structures and resulting building instability at larger scales, this could only be tested on smaller buildings.

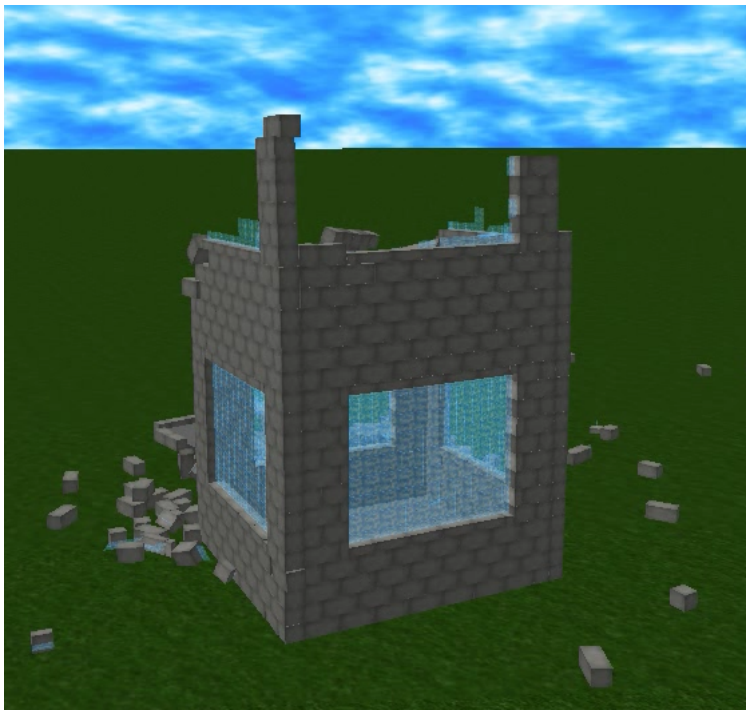

Figure 5.5: Building only partially collapsed after being impacted

5.1.4 Analysis

Lack of Internal Structure in Building

The main result that can be drawn from the user tests, is that with more accurate physics simulation come the need for more accurate building structure. Notably, for buildings, the lack of internal structure causes many of the unrealistic features noticed by the users.

Since the Wall Grammar only generates façades and no building structure as such, internal structure has to be added after the generation process. The only internal structure that was inherent in the generation process was the height of the various levels, and this was not sufficient to produce the level stability required for such a physics simulation. A 3D split grammar might be able to more naturally include floor plans and internal structure in the generation process.

Realistic 'Enough'

Despite all of the unrealistic non-physical features noticed, users still found the simulations fairly convincing. Many stated that they would find it totally realistic in a game environment (31.36% of questions answered were given 7 whilst 71.19% of questions answered were given 6 or greater). One user stated that, although there were many small distracting details that can be noticed whilst analysing it in a video, when immersed in a fast-paced game environment, they would probably not notice such details.

Additional Visual Effects

One of the main differences noticed from the real life videos was the lack of dust, smoke and very fine particles. User testing revealed that most users seem to find the video realistic even without these effects. The addition of dust would most likely increase this perception, because of user expectation but also, as was noted by one of the users, because the dust and smoke covers a significant fraction of the building collapse neat ground level, thus obscuring the user from noticing small, but distracting artifacts in the collision.

Potential Solutions to Non-Physical Features

- Lack of Dust
  A lot of research has gone into the visualisation and creation of dust and particle effects. For a simulated movie of a collapsing building, where the camera angles are know beforehand, dust clouds from existing real life videos can be overlayed onto the simulation video. For a 3D gaming environment, more advanced techniques such will be required that simulate dust positions and densities.

- Rubber-like walls
  This effect could be reduced with more internal structure, and less force on the actual brick themselves. The lower force of the bricks will reduce the wobble as the internal structure would provide stability instead.

- Rubble not settling after collapse and Bricks not breaking up into smaller pieces
  This is very difficult to alleviate in the current setup, one possible solution is to allow individual bricks to breakup when impacted with a sufficiently high force. This would add an extra layer of complexity to the algorithm as well as adding extra computation time to the physics simulation. However, for move generation, if done properly one might be able to make this an optional feature, only turning it on for the final render once the rough, non breakable bricks, version looks sufficiently good to the designer.

- Buildings Auto-collapsing and Rooves Auto-collapsing
  These problem would be alleviated with the introduction of sufficient internal building structure.

- Floating and Spinning Rubble
  More internal structure would ensure that no two geometries are joined whilst not directly touching, which was that sole reason for this effect.

- Unrealistic glass
  Different settings in PhysX might alleviate this problem. Also, making the glass out of 'random' triangle geometries instead of rectangular grid-like geometries would also probably have a positive effect on the perception of realism.

5.2 Performance

For each section, 3 buildings of multiple sizes were tested (small, medium and large, as with the user tests). All rates are averaged out over the last 10 frames. As stated in section 4.2.3, Simulation, Generation and Placement rates are estimated.

As mentioned before in section 4.2.3, three parts of each simulation run will be considered: with physics before building is impacted, with physics after building is impacted and without physics. In all cases, the change in frame rates when the building was impacted was negligible. For this reason the two 'with physics' cases are collectively considered. Each run starts with no building and an empty game world, a building in then generated and brought into view whilst impactors are created, after this physics simulations are turned on and the run is stopped when the building has reached a stable collapsed state. The range of frames in which each section lies for each of these cases are as follows:

- Large Building: 15196 bricks
  - Without Physics: 350-800
  - With Physics: 800-1600
- Medium Building: 3744 bricks
  - Without Physics: 250-600
  - With Physics: 600-1600
- Small Building: 1560 bricks
  - Without Physics: 300-800
  - With Physics: 800-1600

5.2.1 Frame Rate

The graphs in Figures 5.6, 5.7 and 5.8, display the frame rates achieved with various buildings. When the building generated, without physics, only the small building case was able to maintain a real-time frame rate, the medium building was able to keep an average of just under 30 frames per second. The large building however, barely reached interactive levels (approximatly 7-9 frames per second). When physics was turned on, only the small building was able to maintain an interactive frame rate of approximately 10-20 frames per second. The medium building and large buildings were unable to achieve interactive rates.

In terms of real time environments, such as games, methods for speeding up this process need to be explored, even for a relatively small building. For editing purposes, anything beyond the size of the medium sized building (3744 bricks) becomes infeasible.

5.2.2 Simulation times

Results are found in Figure 5.9, 5.10 and 5.11. In each of the cases, the simulation rate dropped instantaneously, as to be expect, but the rate slowly climbed to reach a steady state as the building had reached a fully collapsed state. This was due to the joints breaking as the building collapsed and broke, meaning the simulator only had to consider a smaller and smaller fraction of joints as time progressed. Only for the small building, was the frame rate at an interactive level (around 15 FPS, tending towards approximately 25 FPS).

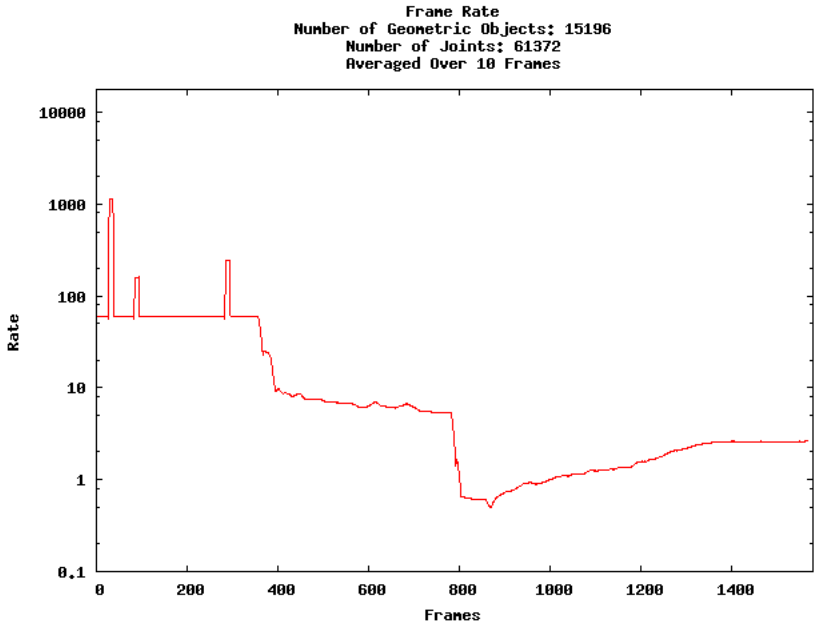

Figure 5.6: Frame Rate Results - Large Building

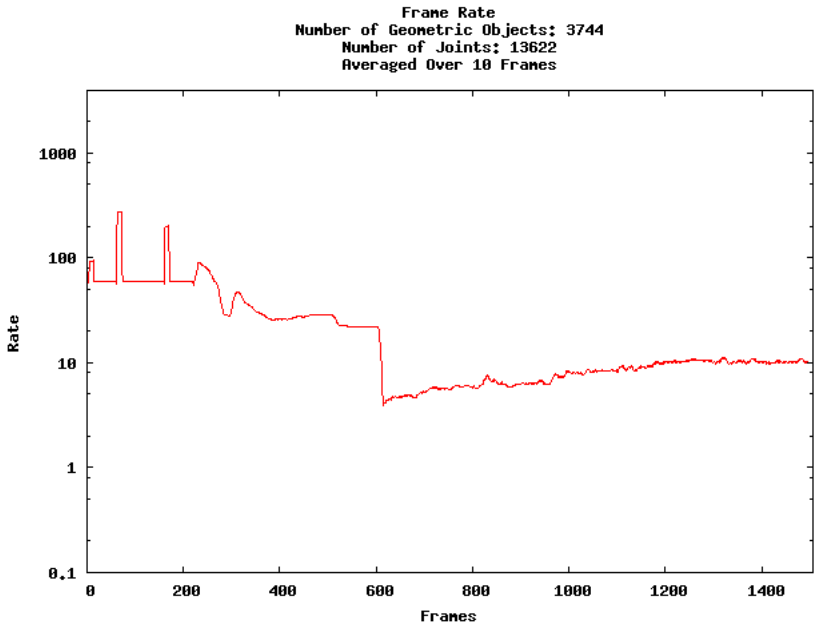

Figure 5.7: Frame Rate Results - Medium Building

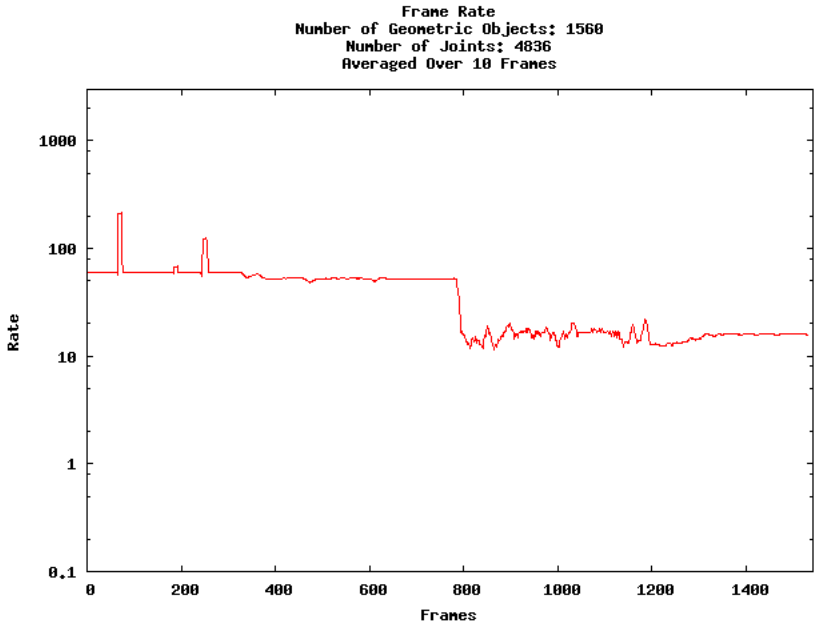

Figure 5.8: Frame Rate Results - Small Building

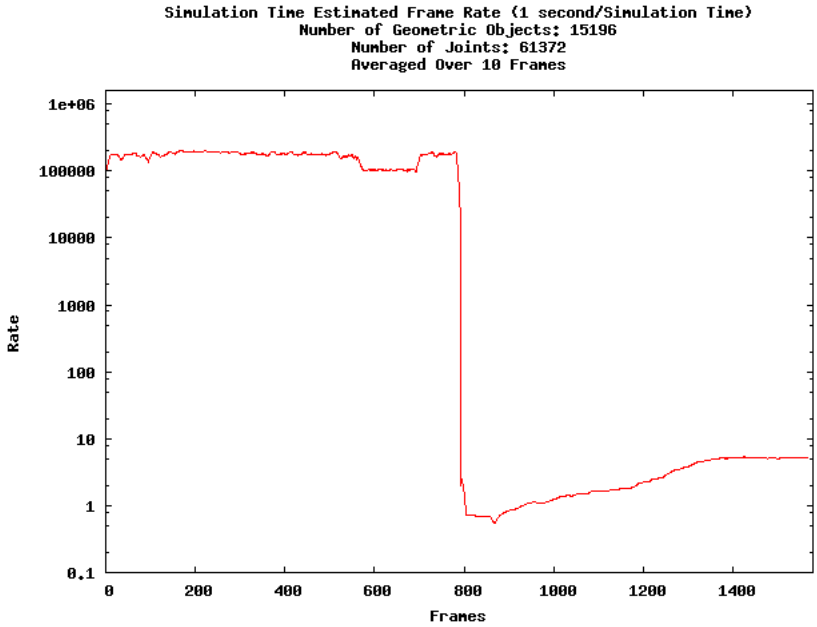

Figure 5.9: Estimated Simulation Rate Results - Large Building

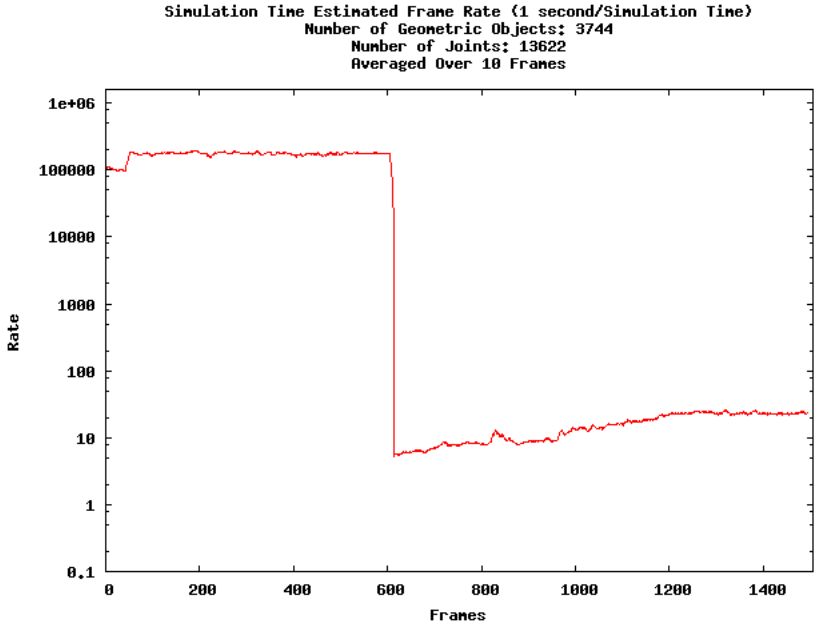

Figure 5.10: Estimated Simulation Rate Results - Medium Building

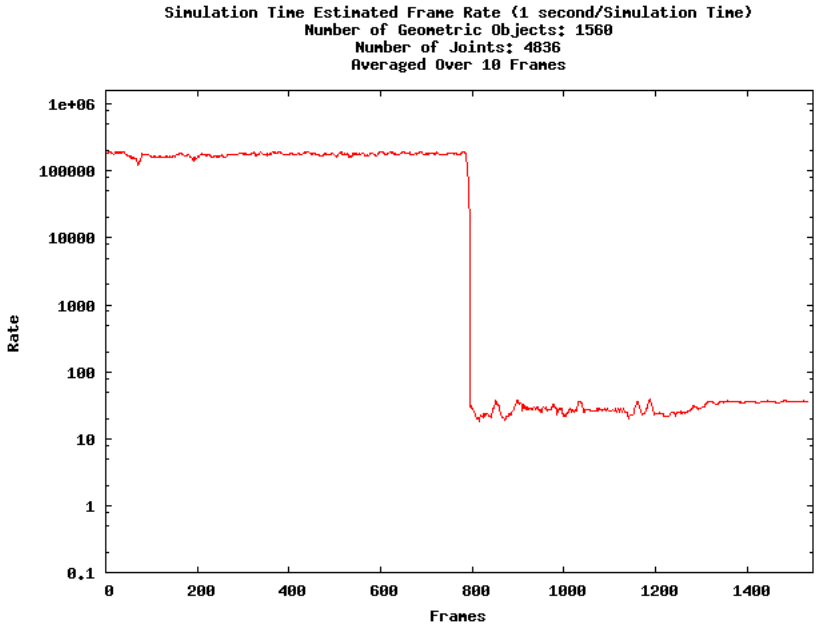

Figure 5.11: Estimated Simulation Rate Results - Small Building

	Large Building	Medium Building	Small Building
Number of Bricks	15196	3744	1560
Number of Joints	61372	13622	4836
Generation Time (10^{-3} sec)	10.359	3.668	2.104
Placement Time (10^{-3} sec)	552.667	147.248	66.576
Generation Rate (est. FPS)	96.53	272.63	475.29
Placement Rate (est. FPS)	1.81	6.79	15.02

Table 5.2: Generation and Placement Times

5.2.3 Generation and Placement Times

As seen in Table 5.2, the rates needed to dynamically edit the buildings are very well in the real-time frame rate range of > 30 FPS, even for the largest building. In order to place the items in a physics simulator would be too slow for interactive use, but might not be an issue, as frame rates for use without physics simulation (see section 5.2.1) are correspondingly low.

Chapter 6

Ethical and Legal Issues

6.1 Licencing

Due to the different libraries used in the project, licencing became a potential problem. OGRE and Qt are under the LGPL, so there was no issues there. The main issue with licencing for this project was in PhysX, which has it's own custom licence. This licence for PhysX is not compatible LGPL for public use and thus the code could not be made public. So, although the methods and algorithms employed are still the intellectual property of the authors, the system which was created to test the software had to be kept non open-source due to PhysX's licencing incompatibilities.

Initially, the project was hosted on GitHub [2], a free, public version control system. This had to be moved to a local server as GitHub does not support free private projects. All other licences used in the system, such as Qt, OGRE and flex were, fortunately, compatible with PhysX's licence (and each other) for private use, as long as we did not make an income from the project.

6.2 User Testing

User testing involved a 45 - 60 minute test, users were informed of this before they signed up, and were compensated for their time and effort after the test was completed. If the user did not complete the test (as was the case for one of the tests that was interrupted by a power outage) the qualitative results of that test was invalidated (qualitative commentary was still used however), and those users were ineligible to restart the test, as they had seen the videos already. Experiments were conducted with a standard PC with no unusual equipment and no material shown was surprizing to what the user had been told or disturbing in any way. Thus, gaining ethics clearance was not necessary.

The experimenter was simply there to hand out the instructions to the user. Otherwise, their involvement was kept minimal, only interacting with the user when answering questions or making clarifications. The experimenter was always away from the testing computer and user, so as to not intimidate or otherwise influence the results. Instructions were straightforward and the environment was kept as controlled as reasonably possible (same environment, location, test set up and lighting conditions).

Chapter 7

Future Work and Conclusion

7.1 Future Work

7.1.1 Extension to 3D Procedural Generation Process

A more complicated 3D split grammar could allow for more complex and accurate internal structures. The split grammar developed by Wonka et al. [16] could be extended in a similar fashion as the Wall Grammar done here. Exception being that the whole building is generated, not just the walls which are stitched together only after.

7.1.2 More Context Data

Besides the various materials of the wall, what other context data can used to build internal structure? For instance, in this system, breaking strength of the joints was not a parameter of the geometry object and was globally set, this breaking force could be included in the context data.

7.1.3 Pregenerated Models

In the original research [9] it is stated that, instead of just plain textured geometry (such as a window), one can rather include a 3D model. This can be applied in the extended method presented in this report, but the geometry would have to be breakable as well. A drawback to this would be that any additional context data that is needed by the geometry objects will have to be present in the model as well. This will either have to be present in the model or set by the user in some way.

7.1.4 Realism Testing with Visual Effects

It is entirely possible that with the addition of certain visual effects, such as dust and smoke, the very distracting non-physical features present in the system would not be noticed as much or at all. Additional surrounding scenery might also have an effect on a user's perception of realism.

7.1.5 GPGPU and Multi-Threaded Simulation

GPGPU programming is whereby heavy processing tasks are offloaded to a graphics card. PhysX in particular is able to use certain types of GPUs to perform a lot of its heavy calculations, as well as make use of multiple threads to speedup. As noted in section 4.2.2, the system could not be run on the GPU.

7.1.6 Partial simulation of building

As mentioned in Section 4.2.2, it was a non-trivial task to break up the simulation into spacial sections. It would be much faster if only the part of the building that was being hit actually underwent physics simulation, leaving the rest uncalculated. Different methods of doing this can be explored.

7.1.7 On the Fly Generation and Inherent Level-of-Detail

Instead of generating all the fine-detail geometries in the beginning, one might be able to generate just the coarse-detail GeometryDescriptions, and then only create the fine detail as it is impacted. Also, for rendering purposes, the course-detail geometry could be generated when the camera is far away from the building, only generating the finer details as it gets closer.

7.1.8 Additional Tiling Schemes

As said in Section 3.3.5, additional tiling schemes might be useful for increasing levels of realism. For example, in the heuristic evaluation results section (5.1.3) it was found that the square shape of the windows was seen as unrealistic. Pseudo-random tiled triangular window section could create a realistic shattering effect. Other tiling schemes could be explored that cause the individual bricks to be breakable themselves, another negative feature stated by many users.

7.2 Summary and Conclusion

In this report, we explored an avenue to add internal structure to the Wall Grammar method of building generation. The original method was extended such that the geometric objects of the building were augmented with extra context data about the internal structure (such as what other geometric objects are attached to it, what material it is and whether it marks a new story of the building). This context data could be extracted from inherent structure created in the generation process.

Results found that, whilst a high level of realism was achieved, there are many noticeable artifacts that are clearly not realistic. Unfortunately, some of these are a result of settings used in the physics simulation, but a major issue was that, whilst stability of the buildings are present, they are not sufficient. The use of a fully fledged games physics simulation on the buildings revealed that they simply did not have enough internal structure maintain a realistic stability. This was, in part, due to the availability of only a limited amount of context data inherent in the generation process. The wall grammar that operates on façades that are then stitched together to form a building, a grammar that operated on 3D geometry might include more internal structure information and contain this information more naturally.

As it stands the system can run at near interactive rates for smaller sized building and can be used to pre-generate collapsing buildings. With GPGPU enabled physics simulation and a more efficient collision

detection algorithm, this rates might get high enough to become interactive. But these are results for future work. Currently, it can be said that it is fast enough that it can conceivably become real-time with further research and improvements in technology. With the high scores of realism given in the user tests, is certainly an avenue of research that should be explored further.

Bibliography

[1] The elder scrolls. `http://www.elderscrolls.com/games/morrowind_overview.htm`. Last Accessed: November 6, 2009.

[2] Github. `https://github.com/`. Last Accessed: November 6, 2009.

[3] Ogre - open source 3d graphics engine. `http://www.ogre3d.org/`. Last Accessed: November 6, 2009.

[4] Physx. `http://www.nvidia.com/object/physx_new.html`. Last Accessed: November 6, 2009.

[5] Qt - a cross-platform application and ui framework. `http://qt.nokia.com/`. Last Accessed: November 6, 2009.

[6] Rockstar games: Grand theft auto iv. `http://www.rockstargames.com/IV/`. Last Accessed: November 6, 2009.

[7] S. Greuter, J. Parker, N. Stewart, and G. Leach. Real-time procedural generation of 'pseudo infinite' cities. In *GRAPHITE '03: Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, pages 87–ff, New York, NY, USA, 2003. ACM.

[8] G. Kelly and H. McCabe. Citygen: An interactive system for procedural city generation. In *Fifth International Conference on Game Design and Technology*, pages 8–16, 2007.

[9] M. Larive and V. Gaildrat. Wall grammar for building generation. In *GRAPHITE '06: Proceedings of the 4th international conference on Computer graphics and interactive techniques in Australasia and Southeast Asia*, pages 429–437, New York, NY, USA, 2006. ACM.

[10] M. Lipp, P. Wonka, and M. Wimmer. Interactive visual editing of grammars for procedural architecture. In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, pages 1–10, New York, NY, USA, 2008. ACM.

[11] W. Mitchell. *The logic of architecture: Design, computation, and cognition.* MIT Press Cambridge, MA, USA, 1990.

[12] P. Müller, P. Wonka, S. Haegler, A. Ulmer, and L. Van Gool. Procedural modeling of buildings. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, pages 614–623, New York, NY, USA, 2006. ACM.

[13] Y. I. H. Parish and P. Müller. Procedural modeling of cities. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 301–308, New York, NY, USA, 2001. ACM.

[14] P. Prusinkiewicz and A. Lindenmayer. *The algorithmic beauty of plants.* Springer-Verlag New York, Inc, New York, NY, USA, 1990.

[15] G. Stiny. Pictorial and formal aspects of shape and shape grammars and aesthetic systems. 1975.

[16] P. Wonka. Procedural modeling of architecture. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Courses*, pages 17–83, New York, NY, USA, 2006. ACM.

[17] P. Wonka, M. Wimmer, F. Sillion, and W. Ribarsky. Instant architecture. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pages 669–677, New York, NY, USA, 2003. ACM.

Appendix A

User Experiment Documents

What follows is the list of documents that were used in user testing. All documents pertaining to the building section of the user tests are included. For pragmatic reasons, the tree specific documents are omitted. All documents are A4 sheets of paper. They are shown at 0.75 scale.

Physically Realistic Procedural Generation - User Study

Thank you for taking part in this user study. During the course of this experiment, you'll be shown a number of videos of computer simulated situations and be asked to answer questions based on how realistic you find the videos to be, and whether you think they would be realistic enough to pass in a video game/simulation. There are two sets of scenarios, one is simulated trees under a variety of forces, and the other is simulated building that are being demolished/destroyed.

For each of the two sets of scenarios, the following will proceed:

1. You will be shown a series of videos of real life examples of the scenario (involving either trees or buildings). You may watch these over as many times as you wish to in order to get a good idea of the movements/behaviour.

2. Then, you will be shown videos of a computer simulation of a similar scenario.

3. After you are finished watching you will be asked to answer a number of questions about how realistic you felt the simulated videos were in comparison to the real-life videos. You may refer to the simulated videos while answering.

4. Steps 1 – will be repeated a number of times, with different entities and situations each time. You will receive more detailed instructions for this.

5. After all of the sets of videos have been shown and the questions answered, you will be asked some general questions about all the videos.

6. You will then move onto buildings, if you have just done trees, or vice-versa.

After doing both the tree and building questions, you may, if you want to, interact with the simulation system and give additional commentary and discussion.

The simulated videos are rendered using computer graphics and thus won't have the detailed textures and colours that real life objects would. **The focus of this study is on the realism of how the trees and buildings move and interact with forces and other entities, not their visual appearance**. We ask that you bear this in mind when answering the questions.

Before the experiment begins, you will need to provide some basic information about you. This information will be kept completely confidential and only used in this study. No personally identifiable information will be included in our results.
If you have any questions about the experiment, please feel free to ask them now.

User Information

• Age: _____

• Gender: [F] [M] (circle one)

• What field(s) of study or work do you consider yourself to be in?
(e.g. Computer Science, Engineering, English, Commerce, etc)

Answer : _____

•During work and/or term time, how many hours a day on average do you spend playing video games? (video games include those on computers, consoles and hand-held devices such as the Nintendo DS)

Answer : _____

•If you did not have work and/or studies (e.g. were on holiday), how many hours a day on average do you think you would spend playing video games?

Answer : _____

•How many years/months have you been playing video games for?

Answer : _____

•Do you have any experience with video game programming, computer simulations and/or physics simulation? If so how much, and under what capacity?

Answer : _____

_____

_____

Descriptions of the different tree simulation scenarios

1. Trees with no external forces acting on it except for gravity. This means that the only forces affecting the tree are the tension and springiness of its branches and trunk, and its weight under gravity.

2. Trees with winds of different strengths blowing at them from a single direction.

3. Large heavy boxes being thrown at trees.

4. Trees that have been cut at a point very low to the ground and are falling over as a result

5. A powerful explosion occuring in the middle of a group of trees.

6. A very strong tornado in the middle of a group of trees. (There are not any real-life videos of this, so please use your imagination to decide what this would look like in the real world)

7. A very strong attractive force (effectively a weak black hole) created in the middle of a group of trees. (There are not any real-life videos of this, so please use your imagination to decide what this would look like in the real world)

Descriptions of the different building simulation scenarios

1. Large 6 story tower being collapsed

2. Medium 3 story building being collapsed and hit with heavy projectiles

3. Small 2 story shed-like building being being collapsed and hit with heavy projectiles

Instructions for user for the buildings section of the experiment

1. In the folder that you are in on the computer there will be 4 files named 'Small.m3u', 'Medium.m3u', 'Large.m3u' and 'Real Life.m3u'.

2. Double click on 'Real Life.m3u'. This will open a movie player program with several movies queued up in the playlist. Watch these movies. Feel free to watch them more than once if you wish.

3. There are 3 pages titled 'Building Video Questionnaire (x / 3)'. These pages will have an 'Small', 'Medium' or 'Large' handwritten in the top right corner.

4. Open the file corresponding to the handwritten number on the first sheet (e.g. if page 1/3 has a 'Medium' handwritten in the top right corner, then double click the file labelled 'Medium.m3u').

5. This will open a movie player program with several movies queued up in the playlist. Watch these movies. Feel free to watch them more than once if you wish.

6. Fill in the questions on the current page of the questionnaire. You may refer to the videos in the playlist as many times as you wish whilst answering.

7. Move onto the next page of the questionnaire. It will have a different handwritten word in the top right hand corner. Double click the file whose name corresponds to the new word (e.g. if page 2/3 has 'Small' written in the top right corner, then double click the file labelled named 'Small.m3u'). Repeat the process of watching the playlist in that folder and answering the questionnaire as per steps 4 – 6.

8. Continue this process until you've gone through all 3 of the pages and watched all 3 sets of movies ('Large', 'Medium' and 'Small').

9. Finally, at the end of the questionnaire, there will be two pages of general questions. Please answer these questions with respect to all of the videos that you saw in all the playlists.

10. You're done. Call one of us and let us know that you're finished.

Building Video Questionnaire (1/3)

• On a scale of 1 to 7, how realistic did the events in the simulation video seem to you, compared to the events in the real life videos? (tick/cross one)

1.Totally unrealistic	[ ]
2.Unrealistic, with some elements of realism	[ ]
3.Unrealistic, but with a noticeable amount of realistic elements	[ ]
4.Equal amount of realistic and unrealistic elements	[ ]
5.Realistic, but with a noticeable amount of unrealistic elements	[ ]
6.Realistic, with some elements that were unrealistic	[ ]
7.Totally realistic	[ ]

• On a scale of 1 to 7, how realistic would you consider the video in terms of what you would expect in a video game? (tick/cross one)

1.Totally unrealistic	[ ]
2.Unrealistic, with some elements of realism	[ ]
3.Unrealistic, but with a noticeable amount of realistic elements	[ ]
4.Equal amount of realistic and unrealistic elements	[ ]
5.Realistic, but with a noticeable amount of unrealistic elements	[ ]
6.Realistic, with some elements that were unrealistic	[ ]
7.Totally realistic	[ ]

• a) Where there any features or elements that you noticed in the real-life videos that were not in the simulation videos?

Answer : _____

_____

_____

• b) Where there any features or elements that you noticed in the simulation videos videos that were not in the real-life videos?

Answer : _____

_____

_____

• Are there any other comments or observations you wish to give? (Comments on the realism of the simulation, unrealistic moments, etc.) **Please answer on back of page.**

Building Videos General Questions (1/2)

• What was your overall impression of the realism of the simulations shown in the videos (based on all the videos shown)?
Answer on a scale of 1 to 10 where:

- •1 means totally unrealistic,
- •5 means somewhat realistic but had noticeably unrealistic elements
- •10 means completely realistic

Answer: _____

Comments: _____

_____

_____

_____

• Do you feel the building simulation videos you were shown are an improvement over existing building realism in current video games?
Answer on a scale of 1 to 10 where:

- •1 means negative improvement (i.e. current games are more realistic),
- •5 means no improvement, and
- •10 means a massive improvement

Answer: _____

Comments: _____

_____

_____

_____

Building Videos General Questions (2/2)

•Are there any suggestions that you could give that you think would make the simulations appear more realistic?

Answer: _____

_____

_____

_____

_____

_____

• Any other comments or observations?

_____

_____

_____

_____

_____

_____

_____

Instructions for user for the trees section of the experiment

1. There will be 7 pages titled 'Tree Video Questionnaire (x / 7)'.

2. These pages will have a number between 1 and 7 (inclusive) handwritten in the top right corner.

3. In the folder that you are in on the computer that you are testing on there will be 7 sub-folders named '1' to '7'.

4. Open the folder corresponding to the handwritten number on the first sheet (e.g. if page 1/7 has a '3' handwritten in the top right corner, then open the folder labelled '3'). You will have a reference sheet which explains what the videos in each folder are meant to show. You may wish to refer to this to get an idea of what the videos are meant to show.

5. Within this folder there will be two sub-folders and two .m3u files.

6. Double click on real.m3u. This will open a movie player program with several movies queued up in the playlist. Watch these movies. Feel free to watch them more than once if you wish.

7. After you have watched all of those movies, double click on simulation.m3u. This will open a different set of videos. Watch these videos. Feel free to watch them as many times as you wish to.

8. Fill in the questions on the current page of the questionnaire (e.g. page 1/7 to continue the example from before).

9. Move onto the next page of the questionnaire. It will have a different number in the top right hand corner. Move up a level in the file browser and go into the folder whose name corresponds to the new number (e.g. If page 2/7 has '6' written in the top right corner, open the folder named '6'). Repeat the process of watching the 2 playlists in that folder and answering the questionnaire as per steps 5 – 8.

10. Continue this process until you've gone through all 7 of the pages and watched all 7 sets of movies. IMPORTANT NOTE: There are no real videos in folders number 6 and 7. These cover things that we could not find real-life simulations of. Please use your imagination to determine what you believe the situations described on the reference sheet would look like.

11. Finally, at the end of the questionnaire, there will be two pages of general questions. Please answer these questions with respect to all of the videos that you saw in the folders numbered 1 to 7.

12. You're done. Call one of us and let us know that you're finished.

Appendix B

Wall Grammar Symbol Types: Low Level Details

- AbstractWall
  - `string name;`
    The name of this wall (e.g. "First Floor", "Bordered Window") and also serves as a unique identifier (identification could also be achieved with an integer ID).
  - `Rectangle actualSize;`
    The size of the rectangle that will generated in step 3 & 4 of the generation process (see section 3.2.4).
  - `double totalDepth;`
    The totalDepth of this AbstractWall, this value is modified when the ExtrudedWall production rule is applied to it.
  - `AbstractWall * childWall;`
    A pointer to the child AbstractWall that will be used in the generation process (section 3.2.4), this is the code representation of the right hand side of the production rules (except for the Wall List, see below).
  - `Size preferredSize;`
    The preferred final size of the rectangle, this is not a strict requirement but the algorithm will try get to as close to it as possible (see step 3 of the generation process, section 3.2.4). The x or y value can be set to -1 to denote no preference. E.g. $(-1, 80)$ means no preference in width, 80 units preference in height.

(NOTE: All following classes are subclasses of AbstractWall.)

- WallPanel
  - `Texture texture;` [1]
    The texture object that will be used to texture the geometry in the final generation. [2]
  - `Size minSize;`
    The user set minimum size that this panel can be. For all other symbols, minimum size is derived during step 2 of the generation process (section 3.2.4)

[1] In the actual code, the Texture was stored as a string and contained in the AbstractWall class, not in the subclasses.
[2] One can also have a pointer to pre-generated models (e.g. windows or doors). This implementation did not include such functionality.

  - `Size maxSize;`
    The user set maximum size that this panel can be. For all other symbols, maximum size is derived during step 2 of the generation process (section 3.2.4)

- BorderedWall
  - `Texture texture;`
    The texture object that will be used to texture the border geometry in the final generation.
  - `Margin left, right, top, bottom;`
    Each margin describes one border of the Bordered Wall. A Margin object contains a distance variable and a resize policy. The resize policy is set to either Minimum (border size is never smaller than the specified distance), Maximum (border size is never greater than the specified distance), or Fixed (border size is strictly set to specified distance) [3].

- ExtrudedWall
  - `double extrudeDepth;`
    The extrudeDepth variable simply describes how much to extrude (or alter the depth) when the associated Extruded Wall production rule is performed on it.

- WallGrid
  - `Orientation orientation;`
    The orientation is set to either Vertical, Horizontal or Both, depending on how it is to be tiled.

- WallList
  - `Orientation orientation;`
    Set to either Vertical or Horizontal, depending on how it is to be tiled [4].
  - `list<AbstractWall *> childWalls;`
    Stores pointers to the child AbstractWalls that will be used in the generation process (section 3.2.4) [5], this is the code representation of the right hand side of the Wall List production rule.

[3] In the code, only the Minimum resize policy was implemented
[4] In the code it can be set to Both, but this is an invalid option and will throw an error.
[5] As a result, `AbstractWall * childWall`, inherited from the AbstractWall class, is ignored.

Appendix C

Additional Statistics

Table C.1: Quantitative Results from User Testing

Scenario	Compared To	Mean Score(/7)	Std. Deviation
Small Building	Real life	5.58	1.26
	Video game	6.00	1.11
Medium Building	Real life	5.47	1.39
	Video game	6.32	0.89
Large Building	Real life	5.63	1.01
	Video game	6.26	0.65
	Question	Average Score(/10)	Std. Deviation
General Questions	Overall impression	7.82	1.22
	Improvement[1]	7.85	1.43

Appendix D

Resize function psudocode:

- Wall Panel: WP
  Actual size given to it by parent node. No children to resize.
- Border Wall: $BW \rightarrow W$
  Child wall's target size is set its target size, shrunk to adjust for the borders
  Note: psudocode assumes a FixedSize resize policy on each border

```
innerSize.width  = actualSize.width - leftBorder.size - rightBorder.size
innerSize.height = actualSize.height - topBorder.size - bottomBorder.size

IF (childWall.minSize.width <= innerSize.width  <= childWall.maxSize.width AND
    childWall.minSize.height() <= innerSize.height <= childWall.maxSize.height )
  childWall.targetSize = innerSize
ELSE
  invalid fit, throw error
```

- Extruded Wall: $EW \rightarrow W$
  Child wall's target size is set to its actual size.

```
childWall.targetSize = actualSize
```

- Wall Grid: $WG \rightarrow W$ Child wall's target size is set fraction of the parent wall's actual size, depending on how many times it can be tiled.

```
IF (orientation == Vertical OR orientation == Both)
  maxNumberOfVerticalTiles = actualSize.height()/childWall.minSize.height;
  IF ( NOT (childWall.minSize.height <= actualSize.height <= childWall.maxSize.height ))
    invalid fit, throw error
  IF (childWall.maxSize.height()*maxNumberOfVerticalTiles < actualSize.height)
    invalid fit, throw error
  childWall.targetSize.height = actualSize.height/maxNumberOfVerticalTiles;

IF (orientation == Horizontal OR orientation == Both)
  ...// Same as Vertical case, replacing Vertical with Horizontal and swapping height and width
```

- Wall List: $WL \rightarrow W_1, W_2, ..., W_n$
  For a vertical orientation, children are allocated all the same width and allocated height based on their minimum, maximum and preferred heights. Vice-versa for a horizontal orientation.

```
IF (orientation == Vertical)
  FOREACH chilWall W_i
    IF ( NOT (W_i->minSize.width( <= actualSize.width() && actualSize.width() <= W_i.maxSize.width() ))
      invalid fit, throw error


  IF ( NOT (SUM(childWalls.minSize.height) <= facadeSize.height <= childWalls.maxSize.height ))
    invalid fit, throw error

  FOREACH chilWall W_i
    W_i.targetSize.width = actualSize.width


  IF (allocating each child its own preferred height \
                will not cause the total height to \
                exceed the parent's height)
    //allocate those heights
    FOREACH chilWall W_i
      W_i.targetSize.height = W_i.prefSize.height
    IF (total height of children is less than parent's height)
      // then there is more to allocate
      add height to each child until the parent's height is reached
      // height added is a ratio of remaining height \
              (childWall.maxSize.height - childWall.targetSize.height)

  ELSE
    // not valid, because it would be too high
    allocate each child it's minimum size
    add a fraction of their preferred heights such that the total will equal parent's height
    // if there are no child walls with preferred heights, then the maximum size is used instead

ELSE if (orientation == Horizontal)
  ...// Same as Vertical case, replacing Vertical with Horizontal and swapping height and width
```