Conditions for Procedural 3D Shape Synthesis

Vaida Laganeckiene

Kongens Lyngby 2012
IMM-M.Sc.-2012-93

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk IMM-M.Sc.-2012-93

Summary (English)

The goal of the thesis is to expand and improve general procedural modelling tool *Generic Graph Grammar* (G^3). In particular the emphasis is laid on conditions for selecting a set of primitives for which to apply certain rules. Better conditions serve to improve expressibility of the grammar and enables the creation of more interesting models.

The G^3 framework is mostly improved in three aspects. Firstly, the merging of coinciding primitives is implemented. This additional feature allows to avoid duplicate geometric information and helps to create topologically consistent meshes, for which conditions are more easily applied. Secondly, intersection tests are implemented with the possibility to cancel the procedural command, if model intersects itself. It helps to create more realistic models without self-intersection or can used as additional artistic option. Finally, implementation of compound commands makes it possible to query primitives for conditions within different scopes and allows to define more flexible rules.

Furthermore, several small improvements are made, such as the adjustments to the user interface, which facilitate the practical use of the framework.

Several models created with improved G^3 method are presented in the report.

Preface

This thesis was prepared at the department of Informatics and Mathematical Modelling at the Technical University of Denmark in fulfilment of the requirements for acquiring an M.Sc. in Informatics.

The thesis deals with conditions for procedural 3D modelling and other improvements to Generic Graph Grammar.

The thesis consists of several chapters. In the related work analysis, alternative procedural modelling techniques are presented and Generic Graph Grammar method is described. Method chapter describes three main issues solved during this project – merging of coinciding primitives, model self intersection checking and different scopes for conditions enabled with compound commands. Implementation chapter explains implementation issues, including algorithm for deleting existing primitives, code design structure and user interface description. Result chapter presents some models created with this framework and the final conclusion and future work chapter concludes the report.

Lyngby, 29-August-2012

Vaida Laganeckiene

Acknowledgements

I would like to thank my supervisor Jakob Andreas Bærentzen for guidance, useful ideas and suggestions, Asger Nyman Christiansen for help understanding original framework and my husband Mindaugas Laganeckas for moral support and patience.

Contents

Summary (English) **i**

Preface **iii**

Acknowledgements **v**

1 Introduction **1**
- 1.1 What is procedural modelling? . . . 1
- 1.2 The goal of the project . . . 3

2 Related Work **5**
- 2.1 L-systems . . . 5
- 2.2 *CGA shape* grammar . . . 7
- 2.3 General procedural modelling . . . 8
- 2.4 Generic Graph Grammar . . . 8
  - 2.4.1 Object representation . . . 9
  - 2.4.2 Generations of the graph . . . 9
  - 2.4.3 Parameters . . . 9
  - 2.4.4 Conditions . . . 10
  - 2.4.5 Commands . . . 11
  - 2.4.6 Rules . . . 13

3 Method **15**
- 3.1 Merging of primitives . . . 15
  - 3.1.1 Undesirable topological situations in the mesh . . . 17
  - 3.1.2 Merging algorithm . . . 18
- 3.2 Self-intersection checking . . . 20
  - 3.2.1 Intersection of two line segments . . . 23
  - 3.2.2 Triangle and line segment intersection . . . 24

3.2.3 Intersection of two triangles 26
3.2.4 Triangulation of polygons 26
3.3 Compound commands 27
3.3.1 Concept of compound command 28
3.3.2 Isolated conditions 29
3.3.3 Primitives accessed by more than one compound command 30
3.3.4 Relative parameters 31
3.3.5 Compound commands containing compound commands . 33
3.4 Other improvements of the framework 33

4 Implementation **37**
4.1 Deletion of primitives 38
4.2 Hierarchical class structure of commands 39
4.3 Description of user interface 40

5 Results **43**

6 Conclusion and future work **47**

Bibliography **49**

Chapter 1

Introduction

1.1 What is procedural modelling?

Entertainment industry, such as computer games and movie makers, requires some economical solutions for producing 3D models. Even though computers become more and more powerful, the work or the artists remains highly valuable asset and should not be wasted. Imagine, for example, how much time it would take for an artist to manually model a city with hundreds of houses or a forest with all the plants (Figure 1.1). Therefore procedural modelling is increasingly used, where large amount of content is required and precise control of the details is not necessary.

In procedural modelling content is usually created using a set of rules, instead of manually defining all details. Sometimes these rules are applied in slightly random fashion, which allows to make many objects that are similar, yet not exactly the same. This can be useful, for example, when creating many instances of some plant species [PL96].

Procedural modelling allows to save space in computer memory. The 3D object can be generated in runtime from the set of rules and rules usually take much less space than precise definition of all geometric information. In addition, when generating content in runtime, it is possible to alter it depending on the context.

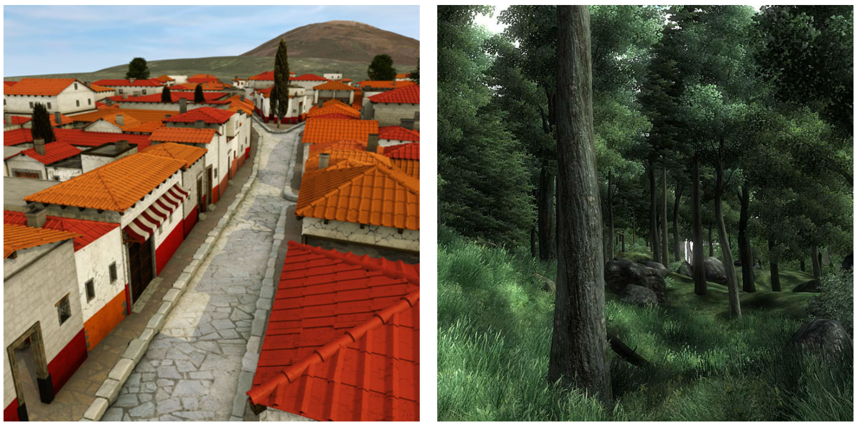

(a) Procedural Pompeii generated with CityEngine (b) Forest generated with SpeedTree in game The Elder Scrolls IV Oblivion ©2012 Bethesda Softworks LLC

Figure 1.1: Complex scenes generated procedurally [1]

For example, road net could be procedurally generated in runtime using terrain as an input.

When procedurally generating content, it is easier to follow some domain specific rules that may be important in specific situation. For example, plants will look more realistic, if we model them according to biological rules. There can be rules concerning number of branches, height of the plant with respect to age, factorial pattern of the plant and similar. There could also be some architectural rules, for example, that door should be at the ground level and facing the street and windows should not intersect with the wall of another building.

Because of these domain specific details, most of the procedural methods are designed for specific type of models, for example, plants, [PL96], terrain [SdKG$^+$09] or buildings [MWH$^+$06]. However it is also important to have some general method that could handle many types of models. This would allow to easier combine different types of models in one scene. In addition, it is faster to learn how to use one framework, than separate one for each type of model. Besides, there are models that can not be easily assigned to some broad category and it can still be useful to model them procedurally.

[1] City picture is from `http://www.esri.com/software/cityengine/casestudies`, forest picture is from `http://www.speedtree.com/gallery/`

1.2 The goal of the project

This project is based on the work of A. N. Christiansen. He designed general procedural method - Generic Graph Grammer(G^3)[Chr11]. G^3 is simple and general method, which can be used to create various type of 3D models: both skeletal structures and surfaces. This method uses the set of parametric rules, which are applied to the primitives of the model - nodes, edges and faces. Which primitives are selected for execution of the rule depends on conditions. The method is described in more detail in section 2.4.

As it appears, G^3 framework can be improved in several ways.

- G^3 framework can produce meshes that contains several primitives with the same geometrical information. By merging these coinciding primitives, topological relations of the mesh can be greatly improved, which helps to control the outcome of the rules (see section 3.1).
- G^3 framework can produce self-intersecting meshes. Various intersection tests can be performed before applying the rule. This improvement would help to control more precisely the resulting mesh and to create wider variety of models (see section 3.2).
- The conditions of G^3 are quite rigid. More advanced conditions with different scope would allow to select primitives more easily. As a result, more sophisticated models could be created.
- It is difficult to define parameters in such a way that the same rule could be used for separate model parts with different positions and orientation. Both conditions and parameters can be improved with the use of compound commands described in section 3.3.

The goal of this project is to overcome above mentioned drawbacks and to improve expressive power of G^3.

CHAPTER 2

Related Work

Grammars used in procedural modelling are mostly applied in some specific domain. Most fundamental grammar - L-systems - was designed by Lindenmayer and applied in computer graphics by Prusinkiewicz [PL96]. L-systems are mostly used for simulating a biological growth. For artificial human-made objects, such as buildings, shape grammars are more suitable. Further L-systems and *CGA shape* grammar will be described in more detail.

2.1 L-systems

L-systems is string rewriting grammar. It consists of some alphabet of symbols V, initial string of symbols ω and a set of productions P, which simultaneously replace each symbol in the string with other strings in each derivation step.

Example of simple L-system:

$$\begin{aligned} \omega &: b \\ p_1 &: a \to ab \\ p_2 &: b \to a \end{aligned}$$

Result of it would be:

$$\begin{aligned}
n &= 0, b\\
n &= 1, a\\
n &= 2, ab\\
n &= 3, aba\\
n &= 4, abaab\\
n &= 5, abaababa\\
n &= 6, abaababaabaab
\end{aligned}$$

For being able to use L-systems for creation of 2D and 3D objects, turtle interpretation is introduced. Alphabet for turtle interpretation consists of symbols F and f. These symbols instruct LOGO-style turtle to go forward with and without drawing the line. In addition, there are rotation symbols. In 2D rotation symbols for turning left is $+$ and for turning right is $-$. For example, *quadratic Koch curve* in figure 2.1 can be generated using this L-system:

$$\begin{aligned}
&\omega : F\\
&p : F \rightarrow F - F + F + FF - F - F + F
\end{aligned}$$

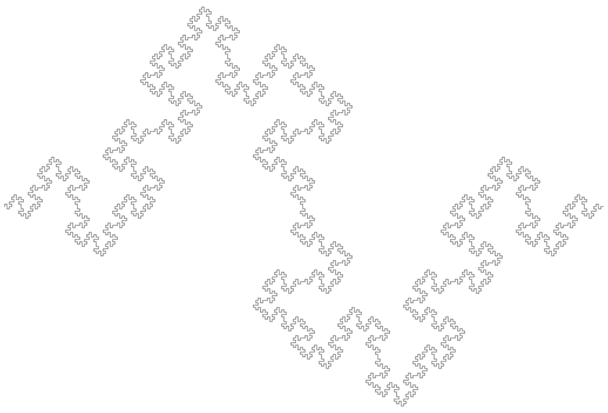


Figure 2.1: Quadratic Koch curve from Wikipadia [1]

In 3D orientation of the turtle is defined by three directions: *heading* H, *left* L and *up* U, satisfying equation $H \times L = U$. Rotation in 3D is defined as turn – rotation around U, pitch – rotation around L and roll – rotation around H. This concept of directions in 3D space is also used in the G^3 framework.

[1] http://en.wikipedia.org/wiki/File:Quadratic_Koch.png

There are several extensions of simple L-systems.

- Bracketed L-systems are introduced for representing branched structures. They have additional symbols for pushing and popping the state of the turtle to the stack and from it.
- Stochastic L-systems allow probabilistic choosing of productions.
- Context-sensitive L-systems take into consideration symbols going before and after the symbol, for which production is applied.
- Parametric L-systems are used to implement more complex growth functions. In this case, productions include logical and arithmetical expressions with parameters.

L-systems are mostly used for modelling plants as they allow to model branching patterns and factorial appearance of plants. However they were also successfully applied in other areas, for example, in modelling road networks.

G^3 framework also have some useful features similar to those of L-systems. It can be used to make branched structures, where branch segments are represented by edges. There is also possibility to apply commands with some probability. Context sensitivity is emulated by the conditions. Commands are parametric, however it is not possible to use logical and arithmetical expressions.

2.2 *CGA shape* grammar

Müller et al. 2006 [MWH$^+$06] explains new shape grammar *CGA shape* for modelling of buildings. Grammar allows to prioritise the rules so that firstly mass model is created, assembled from simple volumetric shapes, such as boxes and cylinders, and later details to the building façade can be added. Rules mostly consist of simple transformations of the scope of the model - translation, rotation and scaling, in addition to subdivide and repeat rules. Subdivision is based on the split grammar by Wonka et al. 2003 [WWSR03]. There is also a rule that allows to split the model to the components of lesser dimensions. This is useful for façade modelling. Furthermore, occlusion checking is implemented, which allows to avoid positioning any components in the intersection of the models. Another interesting feature of snapping makes it possible to snap subdivision lines to some dominant lines of the model.

Grammar has readable rules, but still is mostly understood by people with computer science background. It could be used to generate complex urban environments with a lot of details, using small set of rules. *CGA shape* grammar is integrated in commercial tool *CityEngine*.

The behaviour of subdivision rules of *CGA shape* grammar is emulated by *Split Face* command of G^3 framework.

2.3 General procedural modelling

In addition to procedural techniques designed for specific domain, there were several attempts to design general method, suitable for all kinds of models.

Havemann [Hav05] recommend to change modelling process of 3D shapes and to use functions, more precisely Euler operations, instead of geometric primitives. He developed Generative Modeling Language (GML), which can handle Catmull/Clark subdivision surfaces, BRep meshes and Euler operations. Our approach resembles his in such way that we also use commands similar to Euler operations that create geometric primitives.

Krecklau et al. [KPK10] expended *CGA shape* [MWH$^+$06] grammar with the possibility of defining non-terminal classes for non-terminal symbols. This way the user can create several dynamic modules, like windows, balconies or other façade elements and combine them using some abstract façade templates. Flags are introduced to support local changes of the scene. Authors claim their approach to be general and illustrate it with the cases of architectural modelling and plant modelling.

Both of these approaches resulted in quite complex scripting languages, which can be difficult to learn for an artist. Whereas the approach described in the next section seeks to be easily understandable and user-friendly.

2.4 Generic Graph Grammar

This thesis is further development of Generic Graph Grammar (G^3), which is the procedural method developed by A.N.Christiansen [Chr11]. It combines the capabilities of L-systems and shape grammars and is able to produce both organic models, represented by skeletal structures, and human made objects, represented by surfaces.

Further the main features of G^3 will be described.

2.4.1 Object representation

Directed cyclic graph G is chosen for representation of the object. It is a set of primitives. Primitives can be of three types.

Nodes A node $n \in G$ has position $p \in \mathbb{R}^3$ and radius $r \in \mathbb{R}$. In addition to that it contains a list of neighbouring edges.

Directed edges A directed edge $e \in G$ is represented by two nodes – pre-node n_{pre} and post-node n_{post}, $e = (n_{pre}, n_{post})$. It also contains the list of the neighbouring faces.

Faces A face $f \in G$ is defined as the ordered set of nodes $f = \{n_0, n_1, \ldots, n_n\}$, where each two consecutive nodes are connected by an edge: $e_0 = (n_0, n_1), e_1 = (n_1, n_2), \ldots, e_n = (n_{n-1}, n_n)$. A face not necessary has to be planar.

This representation is chosen, because edges nicely represent skeletal structures and faces - surfaces.

2.4.2 Generations of the graph

Graph G is derived by consecutively applying commands to graph primitives. Initial graph usually contains one node, $G_0 = \{n_0\}$. After an application of the command the new generation of graph is created. All generations of the graph are saved in the list, so after applying n commands we would have the list of generations $G_0, G_1, \ldots, G_n$. The approach of saving all generation was chosen, because command should affect all primitives satisfying some conditions in the same way, no matter in which order it is applied. A command could change the topology of the graph, so if the conditions were checked and the command was applied on the same object, the result of the command for some primitives could make the condition invalid for some other primitives.

2.4.3 Parameters

All command have an input of the same variable parameters, that could be set by the user.

Radius r defines the radius of the new node, after the command is applied.

Directions H, L, U are used to define a direction of newly created edge. This idea is taken from 3D turtle representation of L-systems. H represents heading of the turtle, L – direction to the left and U – direction up. They are all unit length and perpendicular to each other: $H = L \times U$. New edge is created in the direction of heading H.

Length l defines the distance of the new node position p from some starting position p_0 in the direction H, $p = p_0 + l \cdot H$.

Variations in radius σ_r, length σ_l and directions - roll σ_ρ, pitch σ_ϕ, turn σ_θ introduce random variation in all parameters. Variation is sampled from random uniformly distributed variable $U(-\sigma, \sigma)$.

Parameters can be defined in absolute or in relative manner. If the parameters are chosen to be relative, they are defined as the portion of the parameters of some preceding primitives. For example, radius r_1 can be defined relative to the radius r_0 of the preceding node $r_1 = r \cdot r_0$, while in absolute case it would be $r_1 = r$. In case when more than one node or one edge precedes new one, parameters of preceding primitives are averaged. When direction are defined in relative manner, directions of preceding primitives are used as the base vectors, forming a coordinate system.

Only two options for parameters – either being absolute or relative to preceding primitive – are quite restrictive. In addition, there are no global parameters, so that the whole model could be scaled or rotated easily. Parameters are improved during this project with the help of compound commands (see section 3.3).

2.4.4 Conditions

A condition is Boolean statement, which tells, if the primitive has some property or not. Conditions are checked before application of the command and if all conditions defined by the user are satisfied for some primitive, command is applied to that primitive. Condition consists of three parts.

- The property of the primitive.
- The comparison operator (more, less, equal or not equal).
- The value defined by the user.

All the commands can be applied to only one type of the primitives – nodes, edges or faces, so conditions are also divided into three sets: the properties that could be checked for nodes, edges and faces.

Node conditions include random value, age of the node, number of neighbouring edges, number of faces, number of outgoing and incoming edges, position.

Edge conditions include random value, age of the edge, number of neighbouring edges, number of faces, number of outgoing and incoming edges, position of its pre-node, direction.

Face conditions include random value, age of the face, number of nodes in the face, number of edges, position of the first node.

Improvement of conditions is one of the main goals for this master thesis.

2.4.5 Commands

Commands transform one graph generation to another, by replacing one primitive, satisfying the conditions, with the set of other connected primitives. All commands create exactly one node and at least one edge, which position and direction are defined by parameters. The command can change topological relations between primitives, but geometrical information remains the same for old primitives.

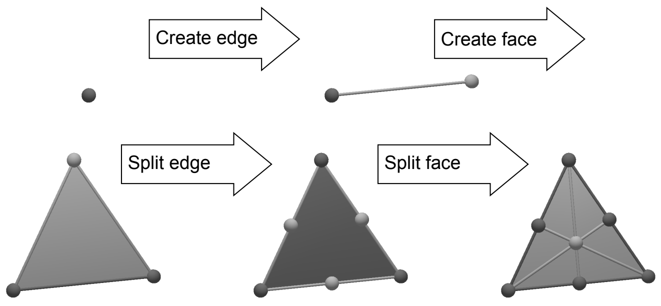


Figure 2.2: Examples of four basic commands

There are four commands, that can be applied to primitives. Examples of their usage can be seen in figure 2.2. Further each of these commands is described in more detail.

Create edge command is applied to node n_0. It creates a new node n_1 and a

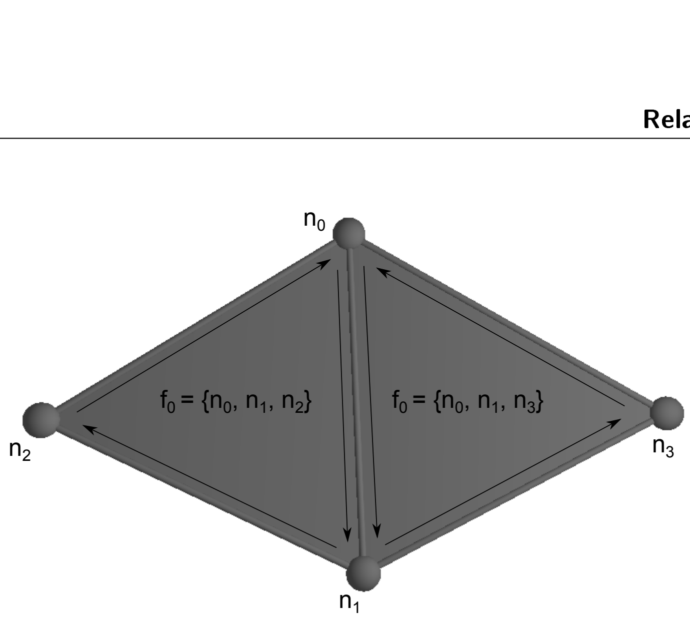


Figure 2.3: *Create face* command originally did not created consistently oriented faces

new edge $e_1 = (n_0, n_1)$. Radius of new node and length and direction of new edge are defined by parameters.

Split edge command is applied to edge $e_0 = (n_0, n_1)$. It creates a new node n_2 and a new edge $e_1 = (n_0, n_2)$. Radius of new node and length and direction of new edge are defined by parameters. In addition old edge is change to $e_0 = (n_2, n_1)$.

Create face command is applied to edge $e_0 = (n_0, n_1)$. It creates a new node n_2 and a new edge $e_1 = (n_1, n_2)$. Radius of new node and length and direction of new edge are defined by parameters. In addition one more edge is created $e2 = (n_2, n_0)$ and a face is created $f = \{n_0, n_1, n_2\}$. Notice that in this way two faces created from the same edge will not be consistently oriented, so one face will be oriented clockwise and one counter-clockwise (figure 2.3). This feature of an old framework was fixed so that newly created face is either $f = \{n_0, n_1, n_2\}$ or $f = \{n_1, n_0, n_2\}$ depending on the orientation of the neighbouring face.

Split face command is applied to face $f_0 = \{n_0, n_1, \ldots, n_n\}$. It creates a new node n_{n+1}. Its position and radius are defined by parameters with respect to the barycentre of the face $b = \frac{1}{n+1} \sum_{i=0}^{n} p_i$. This command has secondary conditions for the nodes of the face. If node n_i satisfies the condition, then new edge $e_i = (n_i, n_{n+1})$ is created. Furthermore a set of new faces is created.

Original framework contained an additional command *Grow face*, which was being applied to faces and increased the number of the vertices in the face.

However this command was removed, because it gave the same results as *Split edge* command (figure 2.4). It is due to the fact that new node created with *Split edge* command not necessary has to be somewhere inside the edge. Actually its position is defined by parameters of length l and heading direction H of the command.

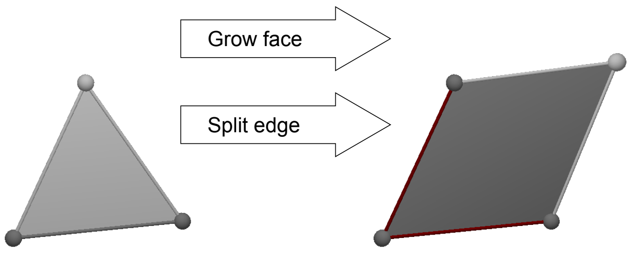


Figure 2.4: *Grow face* command can be replaced by *Split edge* command

There exists one more procedure *Edit node*. It is used to edit geometric information. It is similar to commands in its usage, however the main difference is that it doesn't change topological information and it changes geometric information, such as position and radius of nodes. For convenience it will also be called command later in the document.

Edit node command is applied to node n_0 with position p_0 and radius r_0. Its position is changed to $p_1 = p_0 + H \cdot l$ and its radius is changed to r. Heading H, length l and radius r are parameters of the command.

2.4.6 Rules

Rules were introduced to improve the ease of modelling and to avoid repetitive tasks. A rule is just a sequence of commands with the given name, that could be saved to a file. So instead of applying commands one by one, the user can just apply one rule. For example, we could make a sequence of commands that make a window and define it as a rule. Then we could apply this rule repeatedly with slightly different parameters for the same house or for different kind of houses. Rules can also consist of other rules, so for example, the rule for the house could consist of the rules for the windows, door and other commands. However all rules are just the lists of commands, so if the rule contains another rule, their lists of commands are merged to form one bigger list.

Different kind of hierarchical rules called compound commands were implemented in this project and are described in section 3.3. Compound commands have some advantages against simple rules. Simple rules are very vulnerable to the starting configuration of the model, so some very unexpected results can occur, if starting model is not exactly as it should be. Compound commands are more isolated from the context. Besides, if a simple rule uses the same parameter in many commands and we want to change it, we would have to change it in every one of them, while in compound command case, all parameters are defined in relation to initial parameters, so it is enough to change only initial parameter.

CHAPTER 3

Method

In this chapter the main techniques used to improve G^3 framework will be described. The original framework sometimes produced coinciding primitives. Solution for this problem, which helps to improve topology of the model, is presented in section 3.1. Another problem of model self-intersection is solved in section 3.2. One approach of improving conditions and expressiveness of the grammar is presented in section 3.3. Finally, some other improvement to G^3 framework are listed in section 3.4.

3.1 Merging of primitives

During the modelling process with original framework, sometimes situations arise, where not connected primitives stand in close or the same geometric positions.

Let's examine, for example, the process of creating a box depicted in figure 3.1. After we apply *Split edge* command to hypotenuses of the triangles, we create one node with the position at the box corner and two edges. In this way we form box sides. However commands are applied without looking at the context, so new node and new edge are created, even though there already exist vertical

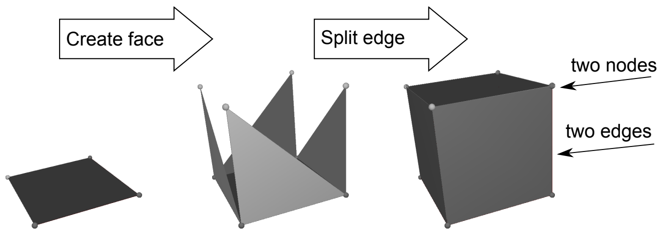


Figure 3.1: Coinciding primitives appears in the process of creating a box

Primitive	Number without merging	Number with merging
Nodes	12	8
Edges	16	12
Faces	5	5

Table 3.1: Number of primitives in the box with and without merging

edges and nodes at the corners of the box. Therefore we get duplicate edges and nodes and the box sides are not connected.

Its is not efficient to store duplicate primitives. As you can see in the table 3.1, we get four too many nodes and four too many edges if we do not use merging of the coinciding primitives. This difference can be much more noticeable for bigger models.

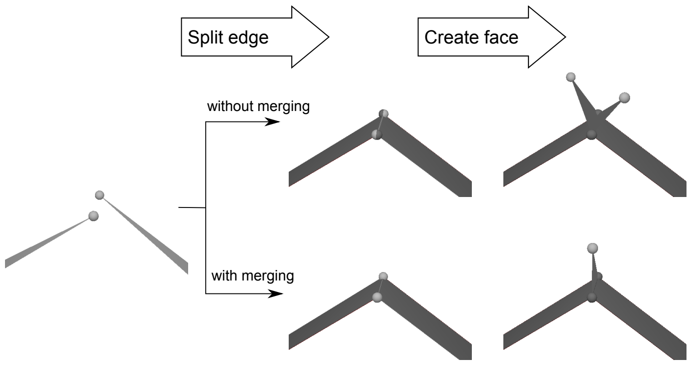


Figure 3.2: Commands produce different results, if applied with or without merging

In addition, it could be very confusing for the user if he sees only one node, when there in fact are two of them. The expected result of the modelling can be different from the actual one. One example is presented in figure 3.2. You can see that if the command *Split edge* is applied without merging, it creates two faces that are not connected on the top and there are actually two edges on the top of the model. So after we apply *Create face* command, it creates two triangles. However if *Split edge* command is applied with merging, it glues together two faces in the top. Consequent *Create face* command results in only one triangle pointing upwards. So merging also allows to model some new situations, that were not possible without merging, and as a result it increases expressiveness of the grammar.

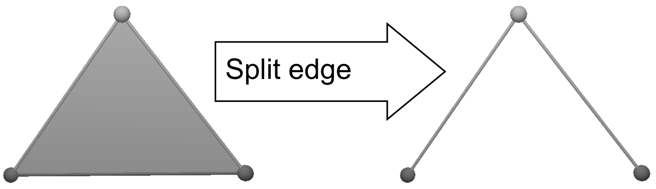


Figure 3.3: *Split edge* together with merging results in deleted face

Moreover, it is much easier to apply conditions to topologically consistent mesh, which results from merging. If all the parts of the model visually close to each other are connected, you would never be confused with how many neighbouring primitives specific primitive has.

One of the problems with merging is that in the original framework you could always know how many new primitives are produced after applying of the command. For example, *Create Edge* command always produced one new node and one new edge. With merging it is not always the case. *Create edge* could produce one new edge, but new node would be merged with existing one. It could even happen that primitives need to be deleted. As the example in figure 3.3 demonstrates, *Split edge* command applied to the bottom edge creates a new node which is merged with the top node of the triangle and the old face has to be deleted. Deletion of the primitives was not possible in the original framework so it had to be implemented. It is described in the section 4.1.

Further, undesirable situations that could arise in the process of merging are described in the section 3.1.1 and main algorithm of merging is described in section 3.1.2.

3.1.1 Undesirable topological situations in the mesh

Below is the list of the situations that are not tolerated and have to be fixed:

1. Two nodes have the same positions. **Solution** - one node is deleted, neighbouring edges are assigned to the other node.
2. Edge has the same begin and end node. **Solution** - edge is deleted. Care should be taken for what happens with neighbouring faces.
3. Two edges have the same end nodes. **Solution** - one edge is deleted, neighbouring faces are assigned to the other edge.
4. The face has only two nodes. **Solution** - face is deleted.
5. The face has the same node more than once. **Solution** - if nodes are consecutive, only one node of those left, otherwise face is split.
6. The face has edge connected sub-cycles. **Solution** - the face is split.
7. Two faces have the same indices. **Solution** - delete one face.

Applying the rules of the grammar without merging can only produce the first situation. Merging actually means avoiding the first problem. However merging the two nodes can produce all the other situations.

3.1.2 Merging algorithm

Merging algorithm consists of three parts.

- Merging of nodes fixes undesirable situation 1.
- Correcting of edges fix situations 2 – 4.
- Correcting of faces fix situations 5 – 7.

3.1.2.1 Merging of nodes

For finding out coinciding or very close node position, k-d tree is used [Ben75]. I have used k-d tree implementation that is part of DTU framework [GEL].

After application of each command, positions of the nodes of the new generation are inserted into k-d tree. Afterwards for each new node n_{new} (or for each old node with the changed position in the *Edit node* case) k-d tree is searched in the very small distance from the position of n_{new}. If some n_i node is found, then nodes n_{new} and n_i are merged. New radius is calculated as $r = \frac{r_{new}+r_i}{2}$

and new position is calculated as $p = \frac{p_{new}+p_i}{2}$. These radius and position are assigned to node n_i and node n_{new} is deleted.

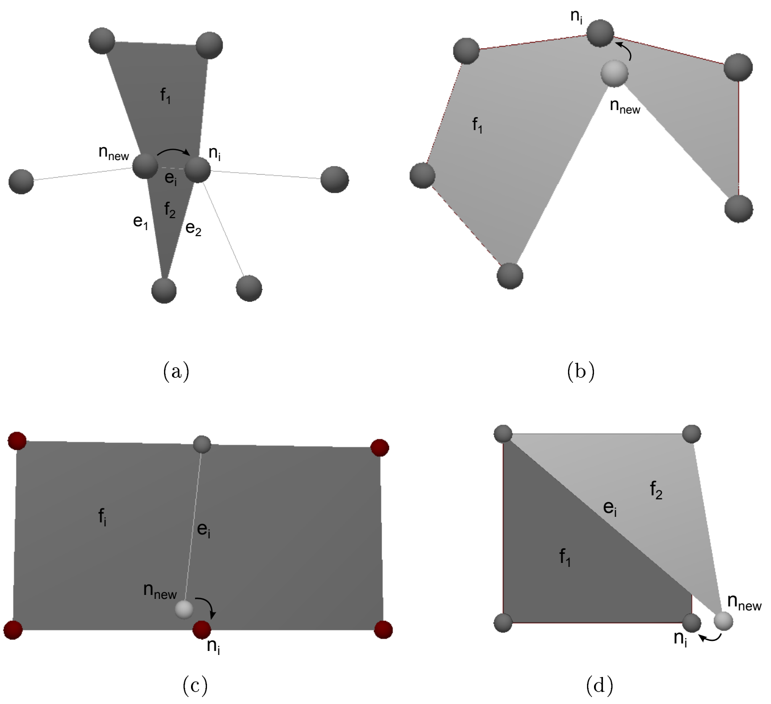


Figure 3.4: Various situations, where merging is performed. (a) edge e_i and face f_2 are deleted, edges e_1 and e_2 are merged together. (b) face f_1 is split into two. (c) after nodes n_{new} and n_i are merged, face f_i contains two sub-loops, that have common edge e_i, therefore face f_i is split into two. (d) face f_2 is deleted and face f_1 is split into two.

3.1.2.2 Correcting of edges

After nodes n_{new} and n_i have been merged, it is checked, if there exists an edge $e_i = (n_{new}, n_i)$ connecting nodes n_{new} and n_i. If so, it has to be deleted, so that we would avoid situation 2 of edge having the same begin and end nodes. Then we need to check edge e_i neighbouring faces. If a neighbouring face has more than three vertices, such as f_1 in figure 3.4(a), one of its vertex is removed. However if the face has three vertices, such as f_2 in figure 3.4(a), it has to be deleted, otherwise it would contain only two vertices and the situation 4 would appear.

All neighbouring edges of nodes n_{new} and n_i hava to be checked if there exists such a pair of edges that share an end node, such as edges e_1 and e_2 in figure 3.4(a). If such edges exist they have to be merged, otherwise situation 3 would appear.

Edges e_1 and e_2 are merged in such way. Firstly, all the faces of e_1 are assigned to e_2. Then average left direction $L = \frac{L_1+L_2}{2}$ and up direction $U = \frac{U_1+U_2}{2}$ are calculated and, if necessary, fixed to be perpendicular. Finally, edge e_1 is deleted.

As the last step of correcting of edges, all neighbouring edges of node n_{new} are assigned to node n_i.

3.1.2.3 Correcting of faces

Firstly, in all affected faces index of node n_{new} is replaced by index of node n_i.

Then all these faces are checked, if they contain double indices, as would happen in situation depicted in figure 3.4(b). Let's say, that we find a face $f = \{n_0, \ldots, n_{i_1}, n_i, n_{i+1}, \ldots, n_{j-1}, n_j, n_{j+1}, \ldots, n_n\}$, where $n_i = n_j$. Then we split these indices into two sets and form two new faces: $f_1 = \{n_j, n_{j+1}, \ldots, n_n, n_0, \ldots, n_{i-1}, n_i\}$ and $f_2 = \{n_i, n_{i+1}, \ldots, n_{j-1}, n_j\}$. However new face is formed only if the set of indices has more than two indices.

Next, all affected faces are checked for sub-loops, as we want to fix situation 6. For each inside edge found, such as e_i in figure 3.4(c) and e_i in figure 3.4(d), face is split into two. Additionally all triangular faces that already exist as the part of some face, such as the face f_2 in figure 3.4(d) is part of the face f_1, are deleted. In this way we avoid situation 7.

3.2 Self-intersection checking

One of the flaws of the original framework was that there was no way to prevent self-intersection of the model. This could result in unrealistic models, for example, plants with intersecting leaves and branches (see figure 3.5).

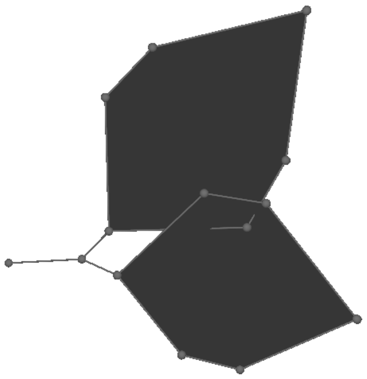


Figure 3.5: A model of the branch with intersecting leaves

In addition to allowing to model more realistic objects, intersection testing could also be used as an artistic tool. It would be as additional option to create more interesting models. For example, with the use of intersection testing we can create the tree in the box model in figure 3.6. The branches of the tree are modelled using *Create edge* command. This command is only applied, if newly created edge would not intersect with anything including the sides of the box. The box and the tree depicted in figure 3.6 is one connected model, so for practical usage of this tree the sides of the box would need to be removed manually.

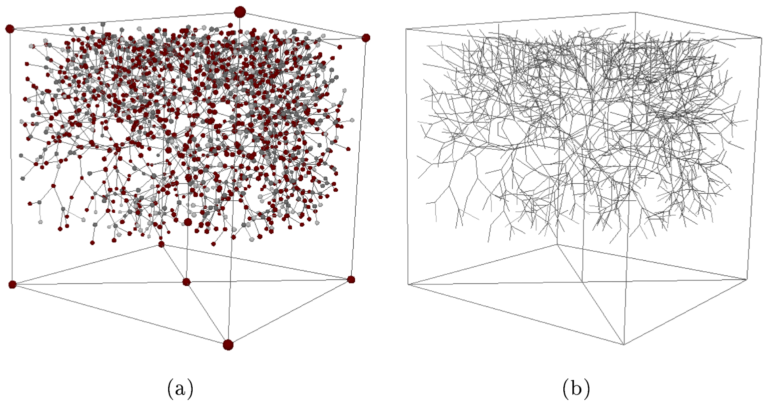


Figure 3.6: Tree in the box model. Intersection testing enforce the tree to stay within the box

Intersection testing is implemented similarly to conditions. The main difference is that we need to know the result of command in beforehand, for judging if command should be applied or not, which is not the case with conditions. As with the conditions, user can choose, if he want to perform intersection testing or not. Actually user has three options: to perform intersection testing, merging algorithm or none of these. Intersection testing and merging can not be performed in the same time, because coinciding vertices are also treated as case of intersection.

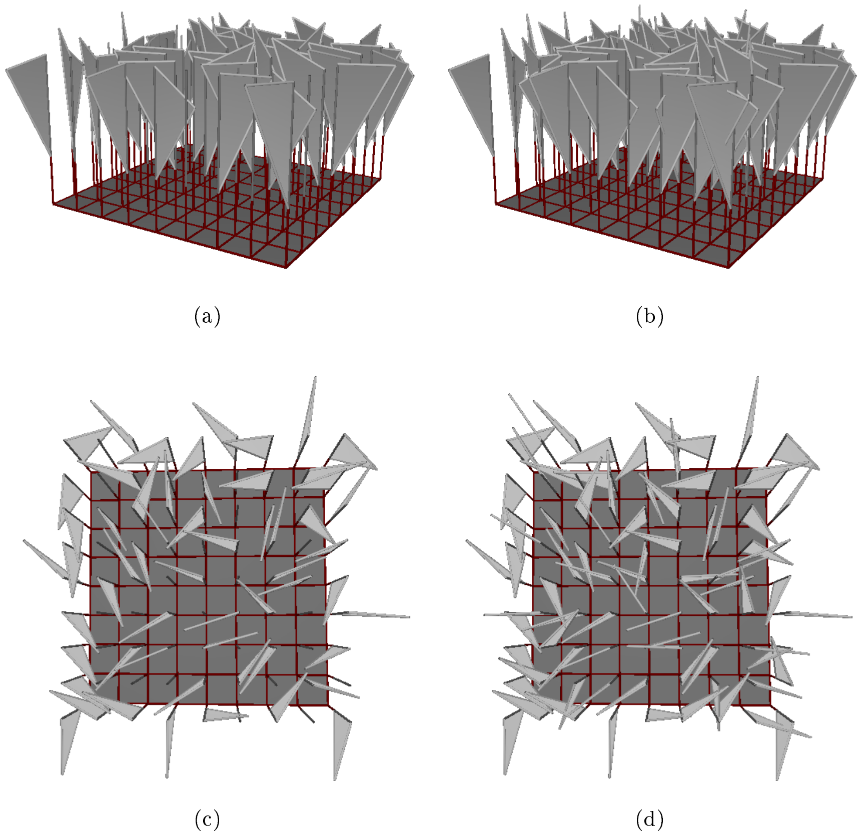


Figure 3.7: Model of flags with intersection testing (a), (c) and without (b), (d)

One limitation of intersection testing is that it can only be applied to one command. If the newly created primitives resulting from the command would intersect with any existing primitives, the command is not applied. Currently it is not possible to discard the whole rule, so if intersection occurs for some command in the middle of the rule, result of earlier commands would still remain.

You can compare the model of the flags with intersection testing and without in figure 3.7. The poles of the flags are modelled with *Create edge* command and afterwards *Create face* is applied to create triangles with random orientation. Intersection testing is performed for *Create face* command, so triangles that would intersect with other flags are not created. However poles of the flags are still present, as they were created before intersection testing.

The result of command can be new nodes, edges and faces. Nodes can not exist without edges connecting them, therefore three cases of intersection can occur: intersection between edges, intersection between edge and face and intersection between faces. Edges are represented as line segments, while faces are represented as triangles or polygons, that can be triangulated. These three cases and triangulation algorithm are described below.

3.2.1 Intersection of two line segments

Intersection of two line segments is implemented as described in [AMHH08] section 16.16.2. Lines are represented as rays:

$$\mathbf{r}_1(s) = \mathbf{o}_1 + s\mathbf{dir}_1 \tag{3.1}$$

$$\mathbf{r}_2(t) = \mathbf{o}_2 + t\mathbf{dir}_2 \tag{3.2}$$

There $\mathbf{o}_1$ and $\mathbf{o}_2$ are end points of the line segments, $\mathbf{dir}_1$ and $\mathbf{dir}_2$ are directional vectors of the lines and s and t are parameters. The intersection point of these lines can be found using formulas:

$$s = \frac{\det(\mathbf{o}_2 - \mathbf{o}_1, \mathbf{dir}_2, \mathbf{dir}_1 \times \mathbf{dir}_2)}{||\mathbf{dir}_1 \times \mathbf{dir}_2||^2} \tag{3.3}$$

$$t = \frac{\det(\mathbf{o}_2 - \mathbf{o}_1, \mathbf{dir}_1, \mathbf{dir}_1 \times \mathbf{dir}_2)}{||\mathbf{dir}_1 \times \mathbf{dir}_2||^2} \tag{3.4}$$

Then there can be three cases:

1. Lines are parallel, if $||\mathbf{dir}_1 \times \mathbf{dir}_2|| = 0$. In this case line segments do not intersect, unless they belong to the same line, than line segments are checked for overlapping.

2. Lines do not share a plane. Then calculated s and t can be used to find closest points between these lines.

3. Lines intersect. Then the calculated s and t are checked against zero and the lengths of the line segments.

Edges that do not have any neighbouring faces are usually used to represent skeleton structures. They are actually only center lines of cylinders with thickness defined by the radii of the end nodes. Therefore we have to take into consideration this thickness h.

Then in the case 1, distance between parallel lines is calculated $d = ||\mathbf{o}_1 - \mathbf{o}_2 - (\mathbf{o}_1 - \mathbf{o}_2) \cdot \mathbf{dir}_1||$. If this distance is bigger than h, then edges do not intersect. Otherwise both line segments are projected on the same line. If they overlap, they intersect, otherwise they do not intersect.

In the case 2, distance between the closest points on the lines is calculated. If it's bigger than h, edges do not intersect. Otherwise it is treated the same as case 3. Parameters s and t are checked against $-h$ and $l + h$, where l is the length of the edge. If they both correspond to the points inside edges extended by thickness h, they intersect, otherwise they do not intersect.

3.2.2 Triangle and line segment intersection

Algorithm for triangle and line segment intersection is described in [AMHH08] section 16.8.1.

Each point on the triangle $\mathbf{f}(u,v)$ can be expresses as the weighted average of triangle vertices $\mathbf{p}_0$, $\mathbf{p}_1$ and $\mathbf{p}_2$:

$$\mathbf{f}(u,v) = (1-u-v)\mathbf{p}_0 + u\mathbf{p}_1 + v\mathbf{p}_2 \tag{3.5}$$

Barycentric coordinates u, v, and $w = 1 - u - v$ should all be between 0 and 1, for the point $\mathbf{f}(u,v)$ to be inside the triangle. You can see the various examples of barycentric coordinates in figure 3.8.

The line that we are checking intersection with is defined as a ray $\mathbf{r}(t) = \mathbf{o} + t\mathbf{dir}$, so for finding intersection point we need to solve equation

$$\begin{aligned} \mathbf{r}(t) &= \mathbf{f}(u,v) && (3.6)\\ \mathbf{o} + t\mathbf{dir} &= (1-u-v)\mathbf{p}_0 + u\mathbf{p}_1 + v\mathbf{p}_2 && (3.7) \end{aligned}$$

Solution to this equation can be found using formulas:

$$t = \frac{\det(\mathbf{s}, \mathbf{e}_1, \mathbf{e}_2)}{\det(-\mathbf{dir}, \mathbf{e}_1, \mathbf{e}_2)} \tag{3.8}$$

$$u = \frac{\det(-\mathbf{dir}, \mathbf{s}, \mathbf{e}_2)}{\det(-\mathbf{dir}, \mathbf{e}_1, \mathbf{e}_2)} \tag{3.9}$$

$$v = \frac{\det(-\mathbf{dir}, \mathbf{e}_1, \mathbf{s})}{\det(-\mathbf{dir}, \mathbf{e}_1, \mathbf{e}_2)} \tag{3.10}$$

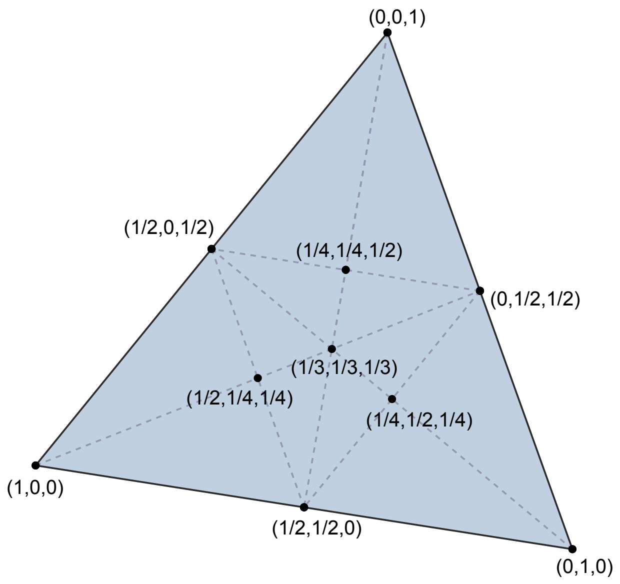


Figure 3.8: Barycentric coordinates of various points in the triangle

There $\mathbf{e}_1 = \mathbf{p}_1 - \mathbf{p}_0$, $\mathbf{e}_2 = \mathbf{p}_2 - \mathbf{p}_0$ and $\mathbf{s} = \mathbf{o} - \mathbf{p}_0$.

Barycentric coordinates u and v are checked to be inside of the triangle, so they should be positive and their sum should be less than one. Also parameter t, defining line segment, should be positive and smaller than the length of the line segment, for the intersection to appear.

If line $\mathbf{r}(t)$ is parallel to triangle plane, so $\det(-\mathbf{dir}, \mathbf{e}_1, \mathbf{e}_2) = 0$, then the distance d between the line and the plane is calculated. If $d \neq 0$, then line segment and triangle do not intersect. Otherwise, line segment belongs to triangle plane. In this case, the triangle vertices are checked, in which side of the line segment they are. If they all are on the same side, then line segment and triangle do not intersect. Otherwise, two ending points of the line segment are checked, in which side of each triangle edges they are. If in any case they both are on the opposite side as the remaining triangle vertex, then line segment and triangle do not intersect, otherwise they intersect.

3.2.3 Intersection of two triangles

Simple algorithm for intersection of two triangles is implemented. Each edge of both triangles is checked for intersection with the other triangle. If in any case intersection occurs, then triangles intersect, otherwise, they do not intersect.

3.2.4 Triangulation of polygons

The faces in *Generic Graph Grammar* are presented as ordered list of vertices, therefore they doesn't have to be flat nor convex and can have any number of vertices. This can cause some problems when checking intersection. As you can see in figure 3.9, the same face can be divided into two triangles in two different ways.

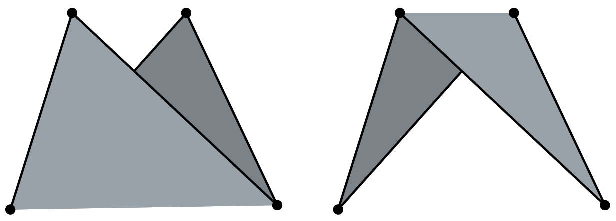


Figure 3.9: The same face defined by the cycle of edges is displayed in two ways.

For the result of intersection to be defined without ambiguities, triangulation of the polygons was implemented. The algorithm is simple and consists of these steps:

1. Select two non-neighbouring vertices of the polygon.
2. Check if diagonal defined by these two vertices intersect with any of the edges of the polygon. If so, select other two vertices and repeat from step 2.
3. Check if diagonal is completely inside or outside of the polygon. For this crossing test described in [AMHH08] section 16.9.1 is used. However crossing test works only in two dimensions. Therefore the axes aligned bounding box of the polygon is calculated and the dimension of smallest extent is dropped. If the center of the diagonal found to be inside the polygon, the polygon is divided into two using this diagonal and algorithm is applied recursively until only triangles are left. In the case of the diagonal

being outside, other two vertices are selected and algorithm is repeated from step 2.

The result of triangulation algorithm can be seen in figure 3.10

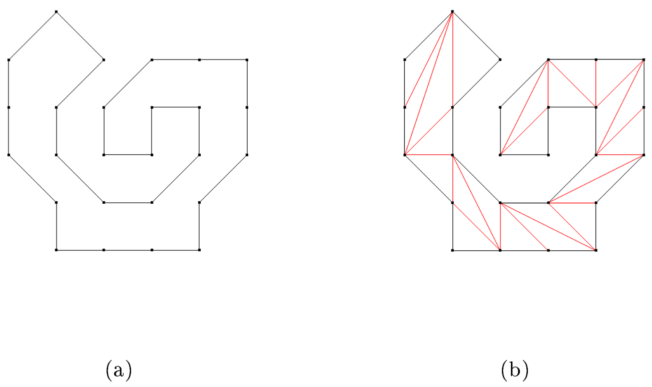


Figure 3.10: Triangulated non-convex polygon

Triangulation result could be improved by selecting pairs of vertices not randomly, but according to the distance between them, so that shorter diagonals are checked firstly. However this would require to calculate distances between vertices ant to sort them according to this distance, therefore it would slow down the algorithm. For intersection testing speed is more important than configuration of the triangles, so simpler approach was chosen. However even faster algorithms exist, that can, for example, be found in [AMHH08].

3.3 Compound commands

The main weakness of original framework is conditions. It is difficult to define precisely, which primitives to select. The conditions are mostly of two kinds - checking geometrical and topological information.

The problem of geometric conditions is that it is not invariant to the scale or translation. So for example, if we select nodes that has x coordinate less than one and then apply the same condition for smaller model, different nodes will be selected. The same would happen if we want to make the model in a different place. So we could not make two identical models standing side by side, using just geometric information.

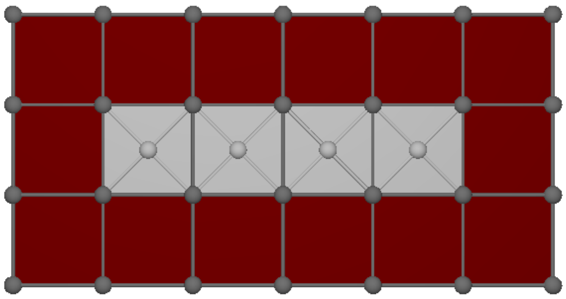


Figure 3.11: Command *Split face* applied for faces that have 8 neighbouring faces. Conditioned faces are coloured grey. It is difficult to find the condition that would select second and fifth face in the middle row.

Topological conditions have another problem, sometimes they are too general and correspond to very common situation. In addition, they can query only local topological properties of the model. These limitations make it hard to select just the primitives you want and nothing else. For example, in figure 3.11, *Split face* command is applied with condition of a face having eight neighbouring faces. You can see that in this case we get four conditioned faces, for which the command is applied. However it would be difficult to split only the second and the fifth face in the middle row, because all four faces have the same topological information.

Conditions can be improved by limiting them to some part of the model that interests us. In this way no unexpected primitives would be selected from some other part of the model. For this reason, compound commands were implemented. They actually have even more advantages than just improved conditions. For example, they can be used instead of rules, but more flexibly, because the user can define initial parameters and conditions.

3.3.1 Concept of compound command

Compound command is a command and like simple command it replaces one primitive with the set of connected primitives. Compound command, the same as simple commands, have parameters and conditions. Also, as the name suggests, compound command is compounded of the list of other commands.

Parameters describe the reference point for contained commands. All the parameters of contained commands would be relative to these initial parameters. This way we can have some part of object rotated in space just by changing initial parameters of compound command.

Conditions are used to select initial primitive. This initial primitive, which could be node, edge or face, defines the scope of conditions for the contained commands.

Commands, contained within compound command, could be either simple commands or also compound commands.

Main features of compound commands will be described in more detail in following sections.

3.3.2 Isolated conditions

The most useful feature of the compound command is that it has limited scope for applying conditions. Scope is limited to the set of primitives that compound command creates, including initial primitive it was applied to. If initial primitives has sub-primitives, they are also included to the scope set. So if initial primitive is face, its border edges and corner nodes are included and if initial set is edge, its ending nodes are included. After applying each new command contained within compound command, scope set is increased by newly created primitives. The topological relations are queried only within this scope.

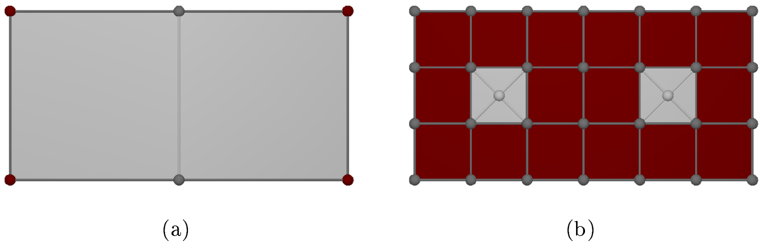


Figure 3.12: Compound command is applied to two faces in figure (a). Compound command splits faces into nine smaller faces and then split the face that has eight neighbours. Result is in (b).

This feature allows to solve the problem illustrated in figure 3.11. The desired result can be acquired using compound commands and it is shown in figure

3.12(b). Compound command is applied to each of two squares in 3.12(a). So the big square is initial primitive of the compound command and it defines the scope of it. Compound command divides big square into nine small ones. Afterwards compound command splits faces that have eight neighbours. Only second and fifth faces in the middle row are split, because in the scope of compound command third and fourth faces have only five neighbouring faces.

Conditions isolated to the scope of the compound command allow to apply the compound command the same way, no matter what is the context of initial primitive. So if you create the compound command on the isolated face, the same topological conditions will be valid if you apply it to the face surrounded by other faces.

There is one problem that has to be addressed with this approach. It is described in the following section.

3.3.3 Primitives accessed by more than one compound command

It could happen that the same primitive belongs to two scope sets, for example in the figure 3.12(a) edge in the middle belongs both to the set of the left square and the right square. It has to be split twice, but it could actually be split four times, twice when applying the compound command to the left square and twice, when applying it to the right square, even though it is not desired.

This problem is solved by checking if the command already been applied to some primitive and it is not repeated, when applying compound command for different primitive set. However even though command is not repeated the second time for the shared primitive, newly created primitives resulting from that command are inserted into both scope sets. So nodes created after splitting the edge are inserted both to the set of the left square and to the set of the right square.

However this would not produce the correct result, if two compound commands were supposed to split shared edge with different parameters, because newly created nodes had different positions. Nevertheless, it is difficult to define what would be the correct result in this case. Therefore this case is treated the same way as the one with the same parameters and more correct solution could be one of future improvements.

(a) Data

Scope	Parameter value
Initial primitive	Edge length $l_{ini} = 3$
Compound command C_{cc}	Length $l_{cc} = 5$
...	
Preceding primitive	Edge length $l_{pre} = 0.5$
Contained command C_i	Length $l_i = 2$
...	

(b) Calculation

C_{cc} mode	C_i mode	Calculation
Absolute	Absolute	$l = l_{cc} \cdot l_i = 5 \cdot 2 = 10$
Absolute	Relative	$l = l_{pre} \cdot l_i = 0.5 \cdot 2 = 1$
Relative	Absolute	$l = l_{ini} \cdot l_{cc} \cdot l_i = 3 \cdot 5 \cdot 2 = 30$
Relative	Relative	$l = l_{pre} \cdot l_i = 0.5 \cdot 2 = 1$

Table 3.2: Example of calculating length parameter

3.3.4 Relative parameters

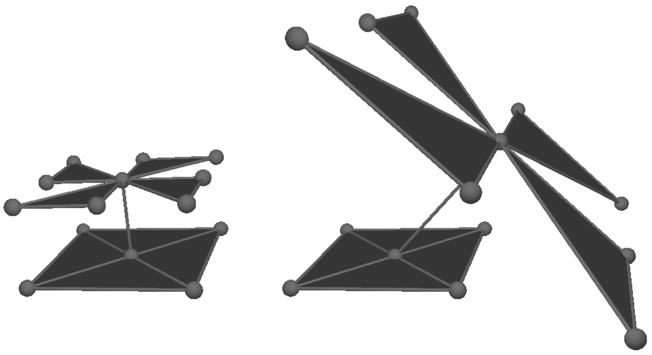
All the parameters of commands contained within compound command are defined in relative mode – either relative to some preceding primitive or relative to initial parameters of compound command. Relative relation means that all scalar parameters, such as radius of the node or length of the edge are defined as the proportion of the corresponding parameters. Direction parameters in relative mode are defined using relative directions as base vectors.

Initial parameters of compound command itself can be defined in an absolute manner or in relation with the parameters of initial primitive.

Example of calculation of length parameter is presented in table 3.2. You can see that if contained command is in relative mode, its parameter is multiplied by the length of respective preceding primitive. Otherwise, its parameter is multiplied by the parameter of compound command, which is one again multiplied by the length of initial primitive, if compound command is in relative mode.

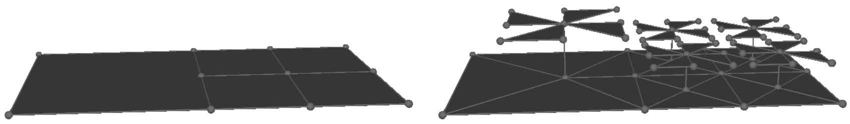
The effect of relative primitives can be illustrated with simple compound command of primitive wind mill, that you can see in the Figure 3.13(a).

It is very easy to scale and rotate the result of compound command by using parameters of compound command. For example in Figure 3.13(b) you can see the same compound command, but its length parameter is set to two and



(a) (b)

Figure 3.13: (a) Compound command of primitive wind mill (b) Compound command of the same mill with twice as big length parameter and different direction parameters



(a) (b)

Figure 3.14: Compound command from Figure 3.13(a) is applied to the model on the left, consisting of one big and four small squares. Result is on the right.

direction parameters are changed. The resulting model is scaled by two and rotated, even though initial face remains the same.

By changing initial face we can also change the resulting model, but only if compound command is set to be relative to this initial primitive. This is illustrated in figure 3.14. You can see that bigger square result in bigger mill than those resulting from smaller squares. You will also notice that even though the compound command is defined using quad as initial primitive, it could easily be applied to face with five vertices as well.

3.3.5 Compound commands containing compound commands

As it was mentioned before, compound commands can contain both simple commands and other compound commands. This presents us with two challenges – how to define parameters and the scope of conditions.

The problem with parameters is solved by simply applying the calculation illustrated in table 3.2 at each level of compound commands. So parameters of higher level of commands are multiplied by parameters of lower level of commands, unless lower level command is in relative mode, then its parameters are multiplied by parameters of preceding or initial primitive.

The problem with scope is solved by defining the scope set of lower level command to be the subset of scope set of higher level command. For being able to query these sets easily, they all are stored in stack data structure, so that the lowest level compound command set is on top and highest on the bottom of the stack. So the scope set of primitives is pushed to the stack when new compound command is started being executed and is popped from the stack, when it is finished.

3.4 Other improvements of the framework

Several other improvements to the framework were implemented. They are described below.

New conditions Firstly the set of conditions was incremented. Two geometrical conditions we added – length of the edge and the area of the face.

The area of the face was calculated by firstly triangulating the face and then adding together areas of triangles. Also one additional topological condition of the number of neighbouring faces for faces was added. It is used for example in figure 3.11. In addition, condition that query positions is checked for all vertices in the face and both nodes of the edge, not just the first vertex in the face and preceding node of the edge. This is changed, because it is not easy to tell, which of the nodes is the first one or the preceding one, just by looking at the model.

Improved user interface Main improvement of user interface was the ability to define parameters precisely, by inserting values to text fields instead of setting them with mouse. It is described in more detail in section 4.3.

Improved shading The shading of the models was improved by defining normals. Normal at each vertex $\mathbf{p}_i$ of the face was defined as cross-product of directions defined by the previous vertex and next vertex:

$$\mathbf{n} = \frac{(\mathbf{p}_{i+1} - \mathbf{p}_i) \times (\mathbf{p}_{i-1} - \mathbf{p}_i)}{||(\mathbf{p}_{i+1} - \mathbf{p}_i) \times (\mathbf{p}_{i-1} - \mathbf{p}_i)||} \tag{3.11}$$

It was possible to define normals consistently oriented, because faces were made to be consistently oriented, as described in section 2.4.5 and illustrated in figure 2.3. In addition, it is now possible to invert all normals, as this can result in better shading. You can see differences in shading, depending whether normals are defined or not in figure 3.15.

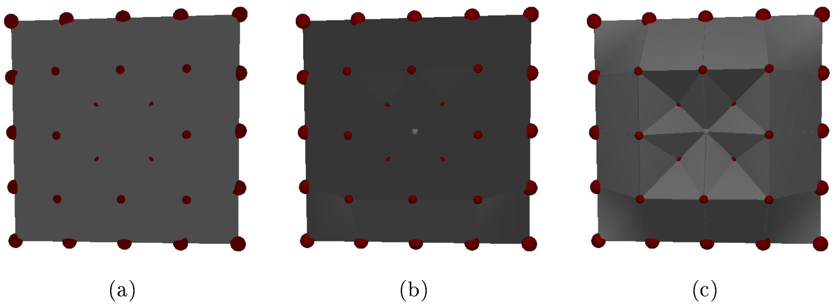


Figure 3.15: Shading depending on the defined normals. (a) no normals defined, (b) normals more or less facing opposite direction to the screen, (c) inverted normals of (b).

Relative directions Relative directions were defined in different way. In original framework each of the directions H, U, L were rotated by the angle between directions of relative primitive H_r, U_r, L_r and respectively x-axis, y-axis and z-axis. However this method wasn't transparent so it was changed to the method, where H_r, U_r, L_r were used as base vectors:

$$H_{new} = H[0] \cdot H_r + H[1] \cdot U_r + H[2] \cdot L_r \tag{3.12}$$
$$U_{new} = U[0] \cdot H_r + U[1] \cdot U_r + U[2] \cdot L_r \tag{3.13}$$
$$L_{new} = L[0] \cdot H_r + L[1] \cdot U_r + L[2] \cdot L_r \tag{3.14}$$

Here H, U, L are parameters of the command and H_{new}, U_{new}, L_{new} are calculated directions used for defining position of the new node and direction of the new edge. $[i]$ is the i^{th} coordinate of the 3D vector.

Split face command *Split face* command was changed in the case, when there were only two conditioned nodes. Before no matter how many nodes were conditioned, new node was inserted in the barycentre of the face. Now, in the case of two conditioned nodes, the face is split into two faces and the conditioned nodes are connected by a new edge, but no new nodes are created, because it is not necessary for splitting the face. If needed, new node can be inserted with subsequent *Split edge* command.

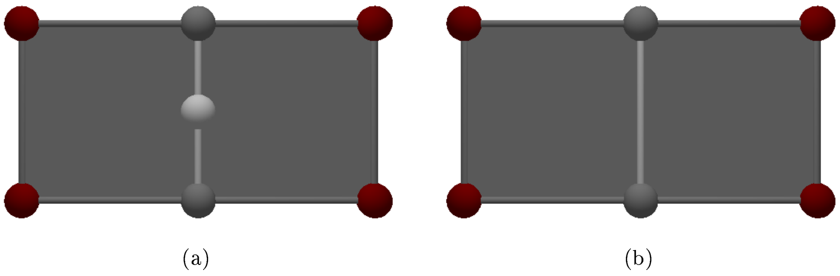


Figure 3.16: Result of the *Split face* command with two conditioned nodes: (a) – before, (b) – now.

Changes in design Several changes to the design of the code were made, mostly in the class hierarchy of the commands, which made it easier to include new commands. It is described in more detail in section 4.2.

Chapter 4

Implementation

G^3 framework was implemented in C++ language. Additionally, these libraries were used:

OpenGL was used for visualization of the model

Glui was used to make user interface

TinyXML was used for reading and writing to the xml files.

GEL was used for operations with 3D vectors and quaternions. In addition implementation of k-d tree from this library was used.

As was mentioned before, this project is extension of the framework developed by A.N.Christiansen [Chr11].

Further several implementation issues will be described. The method for deleting of primitives, which is necessary for merging algorithm, is specified in section 4.1. Hierarchical class structure of commands is presented in section 4.2. Finally, the user interface is described in section 4.3.

4.1 Deletion of primitives

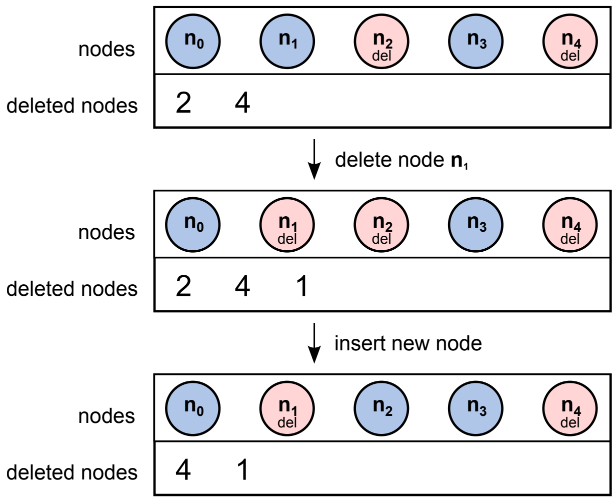


Figure 4.1: Example of deleting a node and inserting a new node. Deleted nodes are marked *del* and coloured in pink.

All the primitives of the model are saved in three lists – one for nodes, one for edges and one for faces. Additional topological information, such as neighbouring edges for each node, is saved as the list of indices. Therefore deleting of the primitives is not so trivial to implement, because deleting of one primitive would change the indices of all succeeding primitives in the list.

This problem was solved by keeping deleted primitives in the list, with the boolean flag that it is deleted. Additionally, the indices of deleted primitives were recycled for newly created primitives. For being able to search easier for unused indices, additional lists for indices of deleted primitives were created.

The example of deleting a node and inserting a new one is presented in figure 4.1. As you can see, after deleting the node n_1, it is still kept in the list of nodes, but it is marked as deleted and is not used anywhere in modelling. Also its index is appended to the end of deleted node list. When new node is created, its index is taken form the start of the deleted node list and this new node is stored in place of the deleted node n_2.

4.2 Hierarchical class structure of commands

Class structure of the code was changed to allow easier extension of the framework with new commands. Current class hierarchy of commands presented in figure 4.2. Firstly, new class for each of the five simple commands was created. They all expend expends class *Command*. Class *Rule* also expends class *Command*. In addition to that it also contains a list of commands. One additional class for compound commands was added. Class *CompoundCommand* has both features of *Command* and features of *Rule*, because it also has list of commands and we should be able to write it to file. For this reason, class *CompoundCommand* expands class *Rule*.

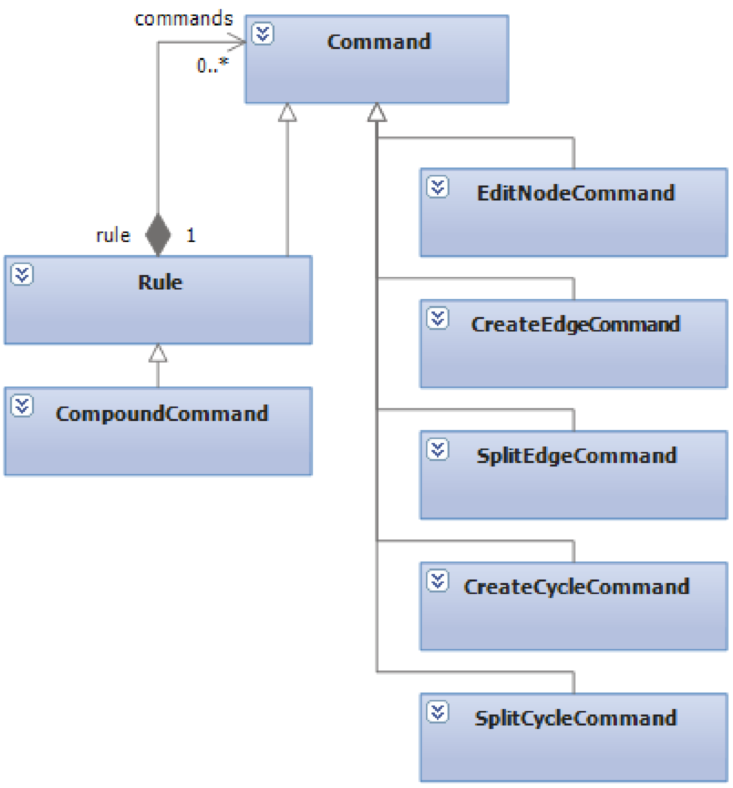
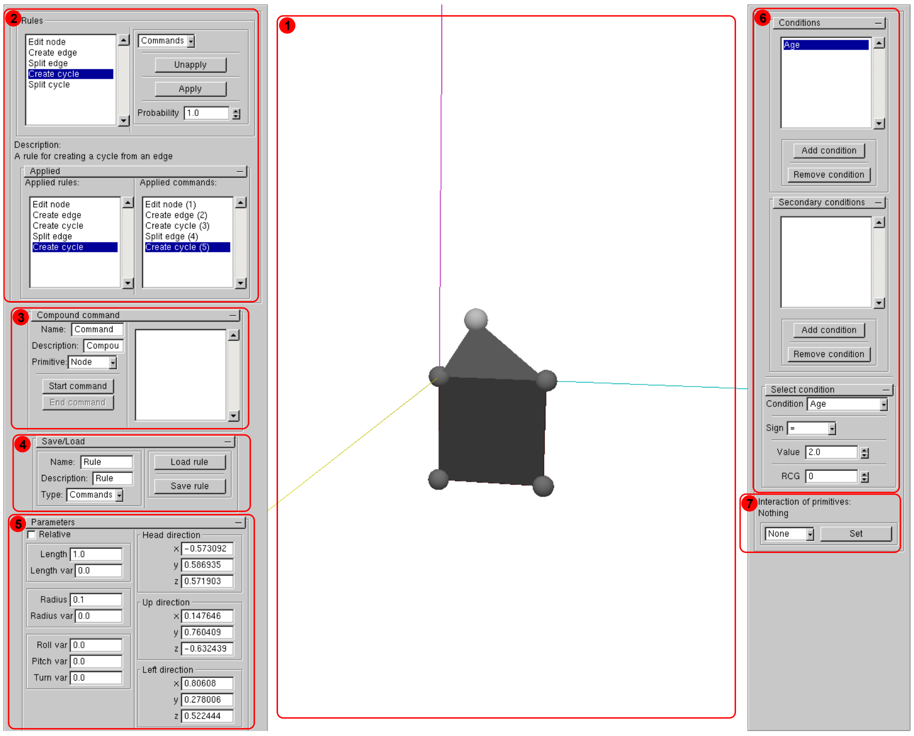


Figure 4.2: Class hierarchy of commands

All of the commands has virtual function *execute*. The function *execute* of compound command calls functions *execute* of all contained commands one by one. This architecture allows easier implementation of compound command composed of other compound commands. Moreover, it is possible to add new types of commands without too many changes in the framework.

4.3 Description of user interface

Firstly the components of the main window, presented in figure 4.3, will be explained. They are marked with red rectangles and numbers in the corner.



Edit node
Create edge
Split edge
Create cycle
Split cycle

Probability 1.0

Description:
A rule for creating a cycle from an edge

Applied rules:
Edit node
Create edge
Create cycle
Split edge
Create cycle

Applied commands:
Edit node (1)
Create edge (2)
Create cycle (3)
Split edge (4)
Create cycle (5)

Name: Command
Description: Compou
Primitive: Node

Name: Rule
Description: Rule
Type: Commands

Relative
Length 1.0
Length var 0.0
Radius 0.1
Radius var 0.0
Roll var 0.0
Pitch var 0.0
Turn var 0.0

Head direction
x −0.573092
y 0.586935
z 0.571903

Up direction
x 0.147646
y 0.760409
z −0.632439

Left direction
x 0.80608
y 0.278006
z 0.522444

Age

Condition Age
Sign =
Value 2.0
RCG 0

Interaction of primitives:
Nothing
None

Figure 4.3: Screen-shot of user interface

1. This is the OpenGL part of the window, where the model is visualized. Yellow, magenta and cyan lines represent main coordinate system. Nodes are visualized as spheres with corresponding radii, edges – as either line segments or cylinders and faces – as polygons. This window was already implemented.

2. In this part of the window, we select which rules or commands to apply. Applied rules and commands are presented in two lists. This was already implemented.

3. This part of the window is for creating new compound commands. For starting a new compound command, you should enter its name, description, select the type of initial primitive – node, edge or face and press *Start command* button. Then all the traditionally applied commands or rules would be the part of this new compound command and appear in the list on the right, until you press button *End command*. This is my contribution.

4. Here you can load and save rules to and from xml files. This was already implemented.

5. Here you can set values for all parameters, by typing numbers to text fields. The same text fields are used to show values of parameters of the current command. You can choose relative mode for command with check-box *Relative*. This is my contribution.

6. This is the part of the window related with conditions. You select one of the conditions in the list and which value you want it to be compared to, then it appears in either conditions list or secondary conditions list. Secondary conditions are for selecting secondary primitives. For example, in the *Split face* case, we select primary conditions for faces and secondary conditions for nodes in the selected faces. This part was already implemented.

7. This is where you can select one of three options: check for intersection, merge primitives or do nothing. This option can be set for each command separately. This is my contribution.

Further I will explain some keyboard short-cuts, that can be used. Some functions already existed, some were implemented by me.

Already implemented functions:

'esc' exits the program.

'p' prints all the nodes, edges and faces to the command line.

'1' toggles drawing of nodes

'2' toggles drawing of edges

'3' toggles drawing of cycles

’4’ toggles drawing of orientation of each node

’8’ toggles drawing of edges as cylinders

’9’ toggles lighting on and off

’0’ toggles drawing of axes of main coordinate system

’e’ exports model to *obj* file

’w’ exports model to *txt* file

’r’ resets command to default parameters.

’z’ undoes last command.

’x’ undoes commands until selected command.

’c’ undoes all commands

’<’ go up applied commands list

’>’ go down applied commands list

’ ’ apply selected command

Additionally I implemented:

’t’ triangulate all faces. This function is just for visualization, faces of the model used for conditions remain the same.

’n’ inverts normals of the model.

’s’ saves current view.

’l’ loads saved view.

Chapter 5

Results

In this chapter several models made with improved *Generic Graph Grammar* framework are displayed. All the renderings are done with Blender. Different materials are also selected with Blender, G^3 is only responsible for the geometry of the models.

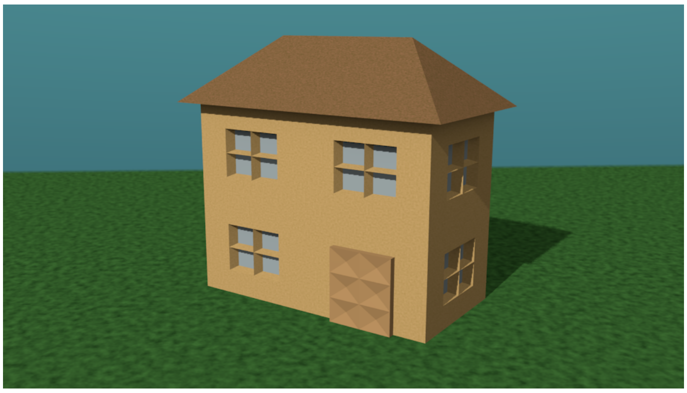


Figure 5.1: The model of a house

The model of the house in figure 5.1 is made using compound command for creating windows. Using compound commands, windows can easily be made equally spaced, no matter what are the orientation and position of the walls. Another compound command was used for door.

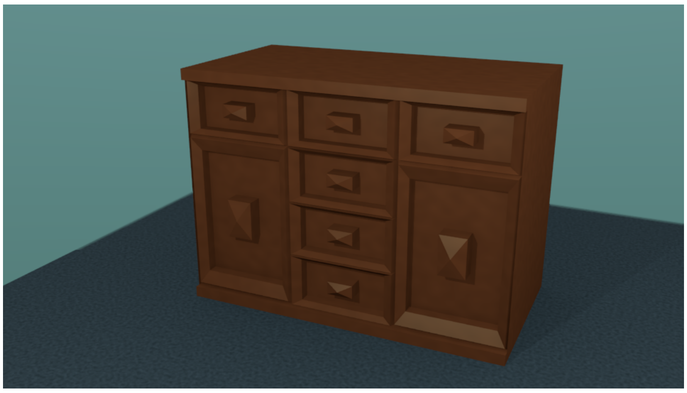


Figure 5.2: The model of a cupboard

For the cupboard model in figure 5.2 the compound command for drawers was used. As you can see the same compound command can be applied to narrow faces to get drawers and to wider faces to get the doors for the sections of the cupboard.

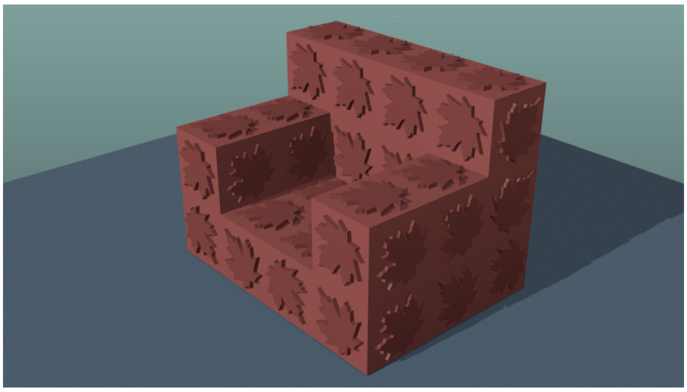


Figure 5.3: The model of a sofa

You can see sofa model with maple leaf pattern in figure 5.3. Compound command allows to position maple leaves at the surface of the sofa with any orientation.

Merging was used for the first three the models to make topologically consistent models, to avoid duplicate primitives and to allow topological queries for conditions with predictable results.

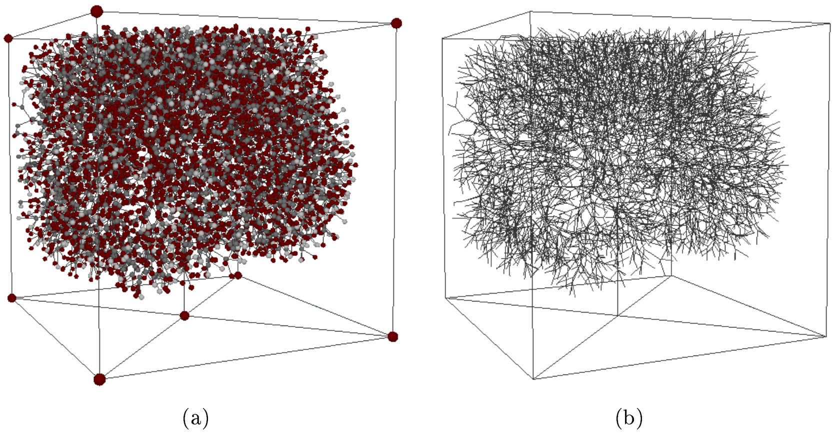


Figure 5.4: Tree in the box model.

Model displayed in figure 5.4 illustrates the use of intersection testing for artistic purposes. This model is made using almost the same rule as previously described model in figure 3.6. This model is not rendered with Blender, because the only faces that it has belong to restricting box. This model has skeleton structure, which needs to be further processed to get 3D model that could be rendered, therefore it is visualized as the screen-shot of the program.

Some statistics, concerning the production of the models is listed in table 5.1. As

	House	Cupboard	Sofa	Tree
Number of nodes	494	206	6824	6285
Number of edges	1088	417	10447	6292
Number of faces	596	213	3619	9
Generation time	0.31 s	0.16 s	50.22 s	6.14 s
Number of commands	53	61	90	42
Size of the model	29 KB	12 KB	366 KB	346 KB
Size of rule file	76 KB	69 KB	122 KB	58 KB

Table 5.1: Statistics of four models

you can see, generation time is quite small for smaller models, but it can increase up to one minute, when the number of polygons exceeds several thousands. This increase of time is mostly because conditions are check quite slowly within such a big set of primitives.

Also you can notice that for the smaller models it is more efficient to store geometric representation of models, while for bigger models rule files take much less space than model files. First three models are saved in *obj* format and tree in a box model is simple text file with positions and radii of the nodes and indices of edges. These model files are compact, while rules are saved in *xml* file format, which is quite wordy. If more compressed file format would be chosen for rules, the saving of space could be noticed even in the smaller models.

All the timing were measured on the computer with 64-bit Windows 7 operating system, Intel(R) Core(TM)2 Duo CPU processor and 4 GB RAM.

Chapter 6

Conclusion and future work

The general procedural modelling tool *Generic Graph Grammar* was extended with three main features. Merging of coinciding primitives enabled us to create topologically consistent models, without any excess of duplicate primitives, which not only resulted in reliable application of conditions, but also helped to create different models. Intersection testing between different primitives helped to avoid unrealistic self-intersecting models, in the same time being interesting modelling option in itself. Compound commands allowed us to limit the checking of conditions within some scope and additionally, served as the rules that could be flexible positioned and oriented. Their usefulness in creating various patterns is illustrated with the models presented in results chapter. Furthermore, several small improvements were made to facilitate the usage of G^3, including adjustments to commands, addition of new conditions and improved user interface.

However, there are still areas where the framework could be improved. Firstly, intersecting testing can be accelerated with faster intersection testing algorithms and using bounding volume hierarchy. In addition, the result of intersection test could have an effect to the whole rule not just one command. Further, the main limit of compound commands is that conditions are always used in absolute manner, so compound commands are not entirely invariant to scaling and translation. Implementation of relative conditions would be a great improvement. Finally, the whole framework could be improved with the use of user defined parameters and possibility to have arithmetic and logical expressions with them.

Bibliography

[AMHH08] Tomas Akenine-Möller, Eric Haines, and Natty Hoffman. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2008.

[BE92] Marshall Bern and David Eppstein. Mesh Generation and Optimal Triangulation. In F. K. Hwang and D. Z. Du, editors, *Computing in Euclidean Geometry*. World Scientific, March 1992.

[Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975.

[Chr11] Asger Nyman Christiansen. Generic graph grammer: A simple graph grammer for generic procedural modelling. In *Computer & Graphics*, 2011.

[dBSvKO08] Mark de Berg, Otfried Schwarzkopf, Mark van Kreveld, and Mark Overmars. *Computational Geometry Algorithms and Applications*. Springer-Verlag, Berlin Heidelberg, third edition edition, 2008.

[FYK10] David Fletcher, Yong Yue, and Majid Al Kader. Challenges and perspectives of procedural modelling and effects. In *2010 14th International Conference Information Visualisation*, 2010.

[GEL] Geometry and linear algebra framework. `http://www2.imm.dtu.dk/projects/GEL/`.

[Hav05] Sven Havemann. *Generative Mesh Modeling*. PhD thesis, Institute of Computer Graphics, Braunschweig Technical University, 2005.

[KM] George Kelly and Hugh Mccabe. Itb journal a survey of procedural techniques for city generation.

[KPK10] Lars Krecklau, Darko Pavic, and Leif Kobbelt. Generalized use of non-terminal symbols for procedural modeling. *Comput. Graph. Forum*, 29(8):2291–2303, 2010.

[MWH$^+$06] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural modeling of buildings. *ACM Trans. Graph.*, 25:614–623, July 2006.

[PL96] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The algorithmic beauty of plants.* Springer-Verlag New York, Inc., New York, NY, USA, 1996.

[SdKG$^+$09] Ruben M. Smelik, Klaas Jan de Kraker, Saskia A. Groenewegen, Tim Tutenel, and Rafael Bidarra. A survey of procedural methods for terrain modelling. In *Proceedings of the CASAWorkshop on 3D Advanced Media In Gaming And Simulation (3AMIGAS)*, 2009.

[TMW02] Robert F. Tobler, Stefan Maierhofer, and Alexander Wilkie. A multiresolution mesh generation approach for procedural definition of complex geometry. *Shape Modeling and Applications, International Conference on*, 0:35, 2002.

[WWSR03] Peter Wonka, Michael Wimmer, François Sillion, and William Ribarsky. Instant architecture. *ACM Trans. Graph.*, 22(3):669–677, July 2003.