Procedural Modelling

Anton Haglund

October 10, 2009
Master's Thesis in Computing Science, 30 ETCS credits
Supervisor at CS-UmU: Frank Drewes
Examiner: Per Lindström

Umeå University
Department of Computing Science
SE-901 87 UMEÅ
SWEDEN

Abstract

By combining generative modelling with the power of grammatical model generation beautiful results can be obtained in the world of procedural modelling. This report covers a comparison between the two techniques, as well as a description of how they work and how they can be used, both separately and in combination. It has been claimed that procedural modelling may increase the start up times for software, but with modern hardware, the right theory and algorithms this is insignificant, concerning models whose level of detail is more than enough. Procedural modelling is an efficient, space saving method for describing complex models. The purpose with this thesis is to find out if it is possible to use procedural modelling to generate models for a graphical user interface for mobile phones. Upon implementing, aesthetic graphical results were obtained and these are presented in the paper. Are you about to write your own procedural modeller, or just interested in the way things really work, all the basics to get you started are covered in this paper.

Contents

List of Figures

Chapter 1

Introduction and problem description

This chapter contains a short introduction to the area and a description of the master thesis.

1.1 Introduction

The primary purpose of this master thesis can be divided in three steps. The first step is to gain further knowledge about procedural modelling, which is an umbrella term in computer graphics to create 3D models and textures from a set of rules [8]. Fundamentally procedural modelling can be divided into grammatical and generative modelling. The basic idea is to use algorithms to produce a scene or a model. The generation is configurable by parameters. The second step is to implement grammatical and generative modelling to be able to use it in the third step, which is to to create models that can be used in a graphical user interface for mobile telephones. The company Tactel is interested in using these methods when developing mobile telephone software.

There are a couple of benefits in using procedural modelling for e.g. a company which develops applications for mobile telephones. One of the benefits for applications using the techniques of procedural modelling is that they leave a very small footprint on a device. Another benefit is that the digital distribution of the application is easier, which is connected to the trend to fast supply with smaller and cheaper applications. The huge size reduction of the software is interesting. A benefit of creating an own modeller is that it is dynamic and it is possible to introduce small subtle changes to the created object.

The company Tactel[1] is a leading developer of mobile applications, providing solutions and consulting services to many of the world's major network operators and mobile handset vendors. They have seen that it is possible to use the techniques of procedural modelling to generate a graphical user interface to be used on a mobile telephone smartphone.

[1] http://www.tactel.se/

1.2 Problem description

The problem is to make it possible to generate 3D models that will be used as icons in a graphical user interface of mobile telephones. The models shall be visually attractive, unique and reproducible. This will be accomplished by develop and implementing a procedural modelling language which will bring generative and grammatical model generation together.

1.2.1 Problem statement

The possibility to make use of 3D graphics in a mobile telephone is now a fact. This enables a graphical user interface in 3D as well. However, the storage space on mobile telephones is still limited, and therefore the 3D-models have to be implemented in a way that doesn't consume a lot of memory. In this project the two most common powerful methods for creating models are combined to create an even more powerful method. The two methods for generating models are generative modelling and grammatical model generation.

The first method, generative modelling, which will from now on be mentioned as stacking, is an approach for functionally creating models, typically solids, with a set of parameters. The next step is to stack modifier operators on top of the created solid. It is also possible to connect multiple solids to finalize a model.

Grammatical model generation is the other method for generating models procedurally. The method that will be looked at here build upon L-systems, which is a type of a formal grammar known from formal language theory and will be further described in Chapter 2.3. Such an L-system generates a sequence of strings which, then, is interpreted as a description of a 3D-model.

1.2.2 Goal and purpose

The final goal is to make it possible to create a 3D model-icon with procedural modelling. Dividing that final goal leads to stacking, grammatical modelling and a combination of these. With stacking it is possible to create solids and apply modifiers by utilising them in a sequence. Grammatical modelling makes it possible to generate models by means of a grammar. The last sub part is to identify possible methods for combining stacking and grammatical modelling.

The usefulness of combinations of procedural modelling techniques can be discussed and it is often claimed that the start up and loading times for such application is very long. With modern hardware and fast CPUs (even in hand-held devices) this is negligible. Efficient algorithms and data structures thoroughly worked out make the loading times unnoticeable, concerning models whose level of detail is more than enough.

Another issue with procedural modelling is that randomized models often get very ugly. By creating a tool for generating models procedurally a designer can create graphics that are appealing to the user. Another way to deal with the problem is to make it possible for the user to mark the models that are attractive. By rating generated models by how pleasing they are to the user this is no longer a difficulty. Another question that has to be added here. For how long do we have to test different models before the result is good?

The list of purposes for using procedural modelling can be long. One of the purposes are connected to communication. Procedural modelling simplifies the digital distribution

of software. It opens up the opportunity to be able to fast supply with smaller and cheaper applications.

An application using procedural modelling leaves a very small footprint on the device. By footprint in this sense we refer to stored data. The reduction of size is immense for the binary data of and along the application. When using procedural modelling it is no longer necessary to supply the software with large models. In the matter of milliseconds the application can create detailed 3D-models hundred times the size of the executable.

1.2.3 Related work

Some work has been done about procedural modelling, the most notable are mentioned in this subsection. We can clearly divide procedural modelling in three parts and look at it from the different viewpoints. These parts are computer graphics, 3D modelling and grammatical model generation.

Computer graphics A couple of modelling techniques that are suitable for real-time rendering and that have been very useful for this master thesis are explained in [1]. The explanation and comparison of different subdivision techniques for example, as well as the description of data structures.

Interesting aspects of the subject can be found in [6], where for example Perlin Noise is presented which have been extensively used to increase the appearance of realism in computer graphics.

3D modelling Nearly all 3D models can be divided in two categories; solid and boundary.

Solid modelling is the representation of the solid parts of an object, that are suitable for computer processing. Surface models and wire frame models are included in other modelling methods.

Primary uses of solid modelling are for CAD, engineering analysis, computer graphics and animation, rapid prototyping, medical testing, product visualization and visualization of scientific research.

Boundary models represent the surface, that is, the shell of the object. These are easier to work with than solid models. Almost all visual models used in games and film are shell models.

There are many different 3D computer graphics software tools, the major are 3ds Max (Autodesk), AC3D (Invis), Aladdin4D (DiscreetFX), Blender (Blender Foundation), Cinema 4D (MAXON), Electric Image Animation System (EI Technology Group), formZ (AutoDesSys, Inc.), Houdini (Side Effects Software), Hypershot, LightWave 3D (NewTek), Massive, Maya (Autodesk), Modo (Luxology), Silo (Nevercenter), SketchUp Pro (Google), Softimage (Autodesk), solidThinking (solidThinking Ltd), SolidWorks (SolidWorks Corporation), trueSpace (Caligari Corporation), Vue 7 (E-on Software), ZBrush (Pixologic).

The basic concepts in modelling are transmigration operations and Euler Boolean operations. In parametric feature based modellers it is common to represent every operation in the creation of a model. They change state in order to maintain information about building model and use expressions to keep rules between the geometric sub objects. This feature makes it possible to regenerate the representation of the model, and is known as a transmigration operation. The series of modifications which preserve the Euler characteristics at every stage in the boundary representation is known as the

Euler Boolean operation. If we fail to maintain the Euler characteristics in each state, we will end up with geometrical artwork depicted by M. C. Escher [7].

Solid models can be created in several ways; Sweeping [13], Boundary representation [2], Parametrized primitive instancing [8], Spatial occupancy enumeration [8], Constructive Solid Geometry [8], Function representation [15] and Facet modelling [23] for example[2].

One example of a modelling language is The Generative Modelling Language (GML), which is a very simple programming language for the concise description of complex 3D shapes [9].

Grammatical model generation In [17] the most about L-systems is covered, although the approach is directed to grammatical modelling of plants, flowers and trees. An example of using L-systems for other things than botanical modelling can be found in [14] where they show how to use L-system etc. to model cities. More ideas about grammatical computer graphics is discussed in [5].

[2]Extrude, which is a type of Sweeping, is discussed in Chapter 2. Constructive Solid Geometry is mentioned in Chapter 2 as well. Parametrized primitive instancing is one of the techniques used in Chapter 4.

Chapter 2

Methods for generating models

There are different methods for procedural modelling. Two of the most common will be presented in this chapter, as well as the concept for combining these. The methods are generative modelling and grammatical model generation. In order to use any of these we need to have a good data structure.

2.1 Data structure

Logically, a polygon mesh is a set of polygons in 3D, a collection of vertices, edges and faces that defines the shape of a polyhedral object. A mesh usually represents a 3D object, where the polygons represent the faces. For simplicity, we drop the distinction and speak of faces rather than polygons. Naturally, a triangle mesh is the special case of a mesh in which all faces are triangles. In the following, we will restrict our attention to triangle meshes, since they have been used in the implementation. Furthermore, with a few exceptions, the generalization to arbitrary polygon meshes is rather straightforward. For each vertex, in addition to its position, data such as colours, shading normal, etc. can be stored. For each face the shading normal is usually stored. Polygon meshes may be represented in a variety of ways, using different methods to store the vertex, edge and face data. These are listed, discussed and explained in [21]. In the following, we describe a popular data structure for the implementation of a polygon mesh, the half-edge data structure [10, 21].

When the computation of model is completed and the model is about to be transferred to the graphics hardware, it has to be converted to a raw data format. For this format triangle strips and triangle fans are often used. For this reason they are presented in this section as well. Because of their repetitive properties these techniques can very helpful when generating models, which is explained in the next section.

2.1.1 The Half-Edge data structure

The half-edge data structure is an efficient way to store meshes as described in [3]. Every edge is divided into two half-edges. Every half-edge have four pointers, one to the next half-edge, to a vertex, to the face and one to the pairing half-edge. Every vertex has a pointer to one of the half-edges emanating from that vertex. Each face has a pointer to one of the half-edges belonging to that face. In Figure 2.1 on the following page the

part of the half-edge data structure corresponding to one face is depicted in a schematic manner.

By using the half-edge data structure the complexity for model creation and modification is decreased drastically, compared to a naive implementation. Instead of for example traversing every pair of vertices to be able to decide which face is next to the current face it is possible to directly access the next face by using the pointers included in the Half-Edge data structure.

The structure for the Half-Edge can be described in pseudo code like this:

```
struct HEedge
{
    HEedge_ptr next;   // The next half-edge.
    HEface_ptr face;   // The face.
    HEvert_ptr vert;   // Vertex at the end of the half-edge.
    HEedge_ptr pair;   // Oppositely oriented adjacent half-edge.
};
```

Typically, traversing all the half-edges belonging to a face is done like this:

```
HEedge_ptr edge = face->edge;
do
{
    // Do something with edge here.
    edge = edge->next;
} while (edge != face->edge);
```

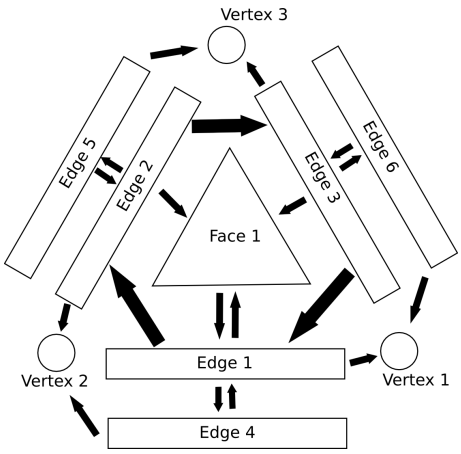


Figure 2.1: The half-edge data structure is a way to represent the model structure where every half-edge has pointers to the next half edge, the face, the vertex and the pairing half-edge. Every vertex has a pointer to one of the half-edges emanating from the vertex. Every face has a pointer to one of the half-edges belonging to that face.

2.1.2 Multiple connected triangles

Simple structures where it is possible to combine several triangles are very common. When dealing with connected triangles it is often more effective to represent the sharing vertices or edges only once. Two of the mostly used techniques are triangles strips and triangles fans, which is explained in this subsection as well as in [1].

Triangle strip The triangle strip is a sequence of triangles $T_1, T_2, \ldots, T_i, T_{i+1}, \ldots, T_n$ where T_i and T_{i+1} share an edge for each $i \in \{1, 2, \ldots, n\}$. n is the size of the triangle strip, i.e. the number of triangles.

A triangle strip describes the first triangle by using three vertices, after that every new triangle is defined by adding one vertex and using the last two vertices defined for the previous triangle. See Figure 2.2 for an example triangle strip.

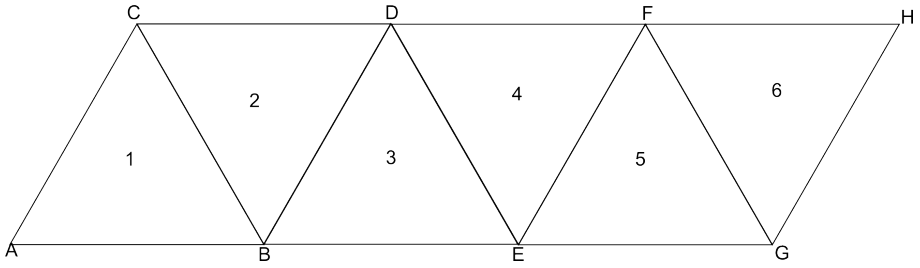


Figure 2.2: The triangle strip ABCDEFGH describes the triangles ABC BCD CDE DEF EFG FGH.

To automatically determine that a set of connected triangles actually is a triangle strip is a very time-consuming task. The final time is though decreased because it is in the end faster to use triangle strips than it is to use triangles.

Triangle strips reduce the amount of data to be stored for describing a series of triangles. The number of vertices stored in memory is reduced from $3n$ to $n + 2$, where n is the number of triangles to be drawn. This allows for less use of memory space, as well as making them faster to load into RAM [19].

Triangle fan The connected set of triangles in a triangle fan shares one central vertex. It only takes $n + 2$ vertices to describe the triangle fan instead of $3n$ vertices, where n is the number of triangles [10]. See Figure 2.3 on the following page for an example triangle fan.

2.2 Generative modelling

Generative modelling, or stacking, is the method used in almost every modelling software. The usual procedure for creating a 3D model is to create one or more solids, and then apply modifiers and deformators to them. By using combination operators these are then turned into a complete model. In the most popular modellers only a few of the basic primitives are used.

Why should one use the same tools and operators that are available in current 3D modelling software? A simple application that make use of generative modelling doesn't

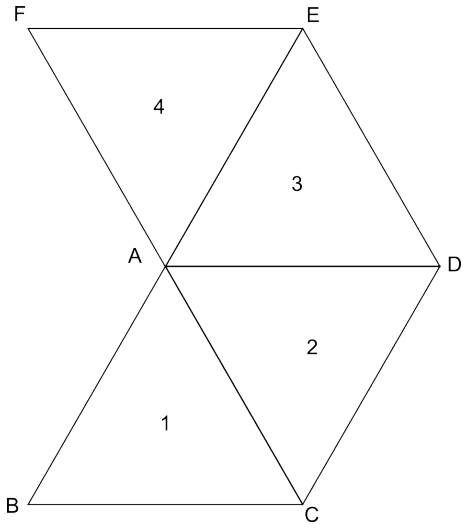


Figure 2.3: The triangle fan ABCDEF describes the triangles ABC ACD ADE AEF.

need each and every tool available in current 3D modelling software. The application is going to be too complex if all the advanced functionality is added. If only a fraction of the tools available, and only the juiciest parts are selected the application is scalable. In 3D modelling software there is of course an already thoroughly worked out foundation of tools and operations, and therefore it is good if we compare with these before we create our own application.

In this section the necessary primitives, modifiers and deformators are presented along with a comparison between functions in common 3D modelling software such as Blender, Maya, 3D Studio Max, AC3D and SketchUp.

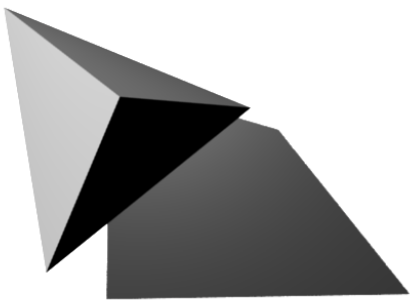
2.2.1 Primitives

In generative modelling, primitives are simple geometric shapes such as the cube, cylinder, sphere, cone, pyramid and the torus. Supplemented by the platonic solids, which is tetrahedron, dodecahedron, octahedron and icosahedron (and hexahedron, but that is the same as a ordinary cube). Some of the primitives are depicted in Figure 2.4 on the next page.

Example A cylinder can be implemented with two triangle fans, one on the top and one on the bottom, together with a triangle strip along the side. The function for generating a cylinder takes three arguments, two for height and width, and one for deciding how many segments the cylinder shall have. When the function knows that n segments shall be used it is possible to create the $2n + 2$ vertices at the right places and then generate a set of triangles. The set is generated according to the techniques described as triangle strips and triangle fans.

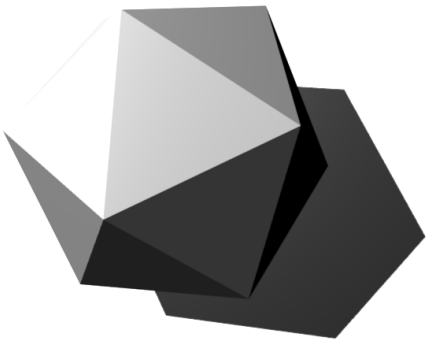
The function for generating a cylinder can be implemented in such a way that the half-edge data structure is set up during creation time. Every pointer is set during the generation which makes the procedure efficient.

Figure 2.4: Primitives



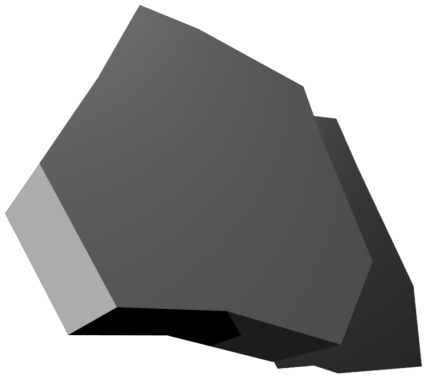
(a) Tetrahedron

(b) Octahedron



(c) Icosahedron

(d) Torus



(e) Prism

2.2.2 Operators

There are four different ways to operate on existing objects. It is possible to transform, deform or combine objects along with applying surface tools. These categories of operators are explained in this subsection.

Transform A transform in the sense of this report is an affine transform, a well known concept from affine geometry. Thus, a transform can be composed of (or decomposed into) three basic operators, namely translation, scaling and rotation. In 3D graphics the transformation is described with a transformation matrix. All the points in the object is multiplied with the matrix and the object is then transformed. The, to every operators, corresponding matrices will be presented in homogeneous coordinates. Assume that L is the linear transformation and $\hat{x}$ is the column vector $L(\hat{x}) = \mathbf{A}\hat{x}$ for some matrix $\mathbf{A}$, which is called the transformation matrix of L.

A translation is to move an object in space, that is, an operation to change the current position of an object with respect to a single point, such as the origin. The translation is represented by the matrix in Equation 2.1.

$$\begin{pmatrix} 1 & 0 & 0 & v_x \\ 0 & 1 & 0 & v_y \\ 0 & 0 & 1 & v_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.1}$$

Scaling is a linear transformation that enlarges or diminishes objects. When scaling an object its size is changed. There are two types of scaling, uniform and non-uniform. For a uniform scaling the scale factor is the same in all three dimensions. A non-uniform scaling is performed when at least one of the scaling factors differs from the others. The scaling matrix describing the scaling by the factors v_x, v_y and v_z in the x-, y- and z-dimension respectively, is given in Equation 2.2.

$$\begin{pmatrix} v_x & 0 & 0 & 0 \\ 0 & v_y & 0 & 0 \\ 0 & 0 & v_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.2}$$

An object can be rotated relative the origin around any axis with an arbitrary angle. The matrix representation of such a rotation is given in Equation 2.3. It is possible to use Quaternion based rotations (which avoids gimbal lock) as well, but they will not be explained in this report.

$$\begin{pmatrix} \hat{u}_x & \hat{v}_x & \hat{w}_x & 0 \\ \hat{u}_y & \hat{v}_y & \hat{w}_y & 0 \\ \hat{u}_z & \hat{v}_z & \hat{w}_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.3}$$

Here $\hat{u}$, $\hat{v}$ and $\hat{w}$ are the three units which form the basis for the rotated coordinate system.

Surface tools A surface tool is an operation to make a model or a object less rough, to change its appearance, but keep its general form. The primary subjective goal is to enhance its visual appearance. The subdivision and bevel operations can be counted as surface tools

A subdivision surface of the faces of an object is performed to give a smooth appearance, to enable modelling of complex smooth surfaces with simple objects having only a few vertices. A more detailed description of subdivision is given below.

The Bevel Modifier adds the ability to bevel the edges of the object that it is applied to. Allowing control of how and where the bevel is applied to the object.

Example A subdivision surface is a way to give an object a smooth appearance, enable modelling of complex smooth surfaces with simple objects as a base. There are different techniques for subdividing. Some of them are: Catmull-Clark, Loop, Doo-Sabin and $\sqrt{3}$. An explanation of Loop subdivision [12] will be presented here.

The subdivision consists of two steps; splitting faces and smoothing. The split step creates new vertices at the midpoint of each edge, along with new edges and faces. A total of three new vertices, four faces, and three edges is created. When using the half-edges data structure there are initially three half-edges and one face. When the splitting phase is done there are a total of twelve half-edges, three per face. The process of splitting a face is depicted in Figure 2.5.

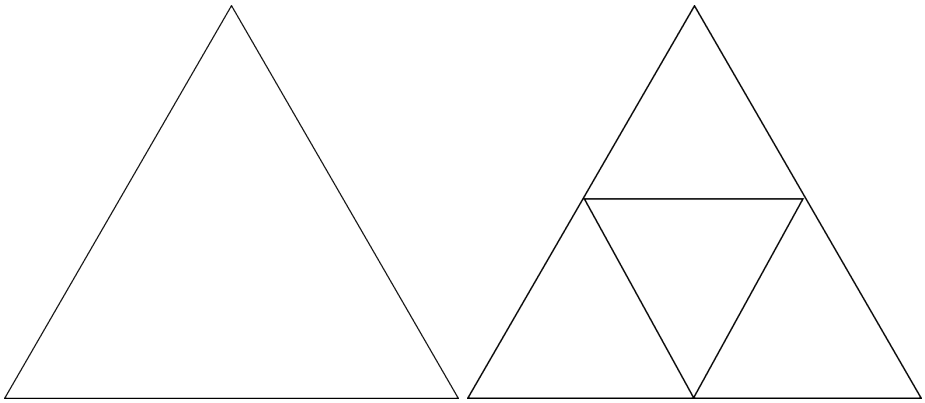


Figure 2.5: Before and after splitting one triangle in the mesh with Loop subdivision. The underlying data structure is not important here and therefore the edges are not visualised as half-edges.

The next step in Loop subdivision is smoothing, which means that that the vertices will be repositioned with respect to the positions of the other vertices. Consider an existing vertex p^k, where k is the number of subdivision steps. If the valence of p^k is n, then p^k has n neighbouring vertices, p_i^k , $i \in \{0, 1, \ldots, n-1\}$. With neighbouring vertices we mean all the vertices, through an edge, connected to the current vertex. As described in [1], the rules for updating the position of an existing vertex p^k into p^{k+1} and creating a new vertex p_i^{k+1} between p^k and each p_i^k are

$$p^{k+1} = (1 - n\beta)p^k + \beta(p_0^k + \cdots + p_{n-1}^k) \tag{2.4}$$

$$p_i^{k+1} = \frac{3p^k + 3p_i^k + p_{i-1}^k + p_{i+1}^k}{8}, i = 0 \ldots n-1 \tag{2.5}$$

Here, the factor β depends on the vertex valence n, as follows:

$$\beta = \frac{1}{n}\left(\frac{5}{8} - \frac{(3 + 2\cos(2\pi/n))^2}{64}\right) \tag{2.6}$$

Other functions (that require less computations) have been defined for β, but this is the one used in the implementation.

The position update for a new vertex (Equation 2.5 on the preceding page) can be visualized by a mask, which is depicted in Figure 2.6.

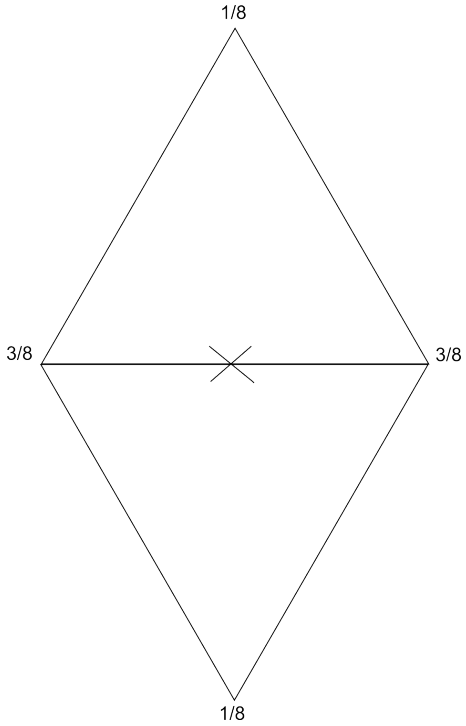


Figure 2.6: The new vertex (marked with X), is repositioned according to old positions of nearby vertices with weights 1/8 and 3/8 as explained in Equation 2.5 on the previous page. Nearby vertices in this case is vertices that belong to the two triangles we are operating on.

It is possible to use this method not just for subdividing surfaces to give them a smoother appearance. By modifying the weights in the smoothing step different shapes can be obtained for extreme values. See Figure 2.7 on the facing page for example.

Another option for extending the usage of the subdivision function is to set $\beta = 0$, thus skipping the smoothing step completely in order to make the mesh more tessellated.

Deform The deformation operations are used to contort, distort or wring an object. They can, for example, cause an object to assume a crooked or angular form.

A deformation changes the overall form of an object, not necessarily to make it look better. This distinguishes it from a modifier operation whose purpose is to enhance the subjective appearance of an object. The specific deformation operations explained here is the displacement-, mirroring-, twist-, bend- and extrude operation.

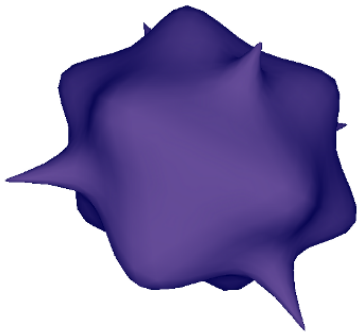


Figure 2.7: By applying extreme values to the weight function in the smoothing step in subdivision interesting results are obtained. In this figure an octahedron is subdivided once with weight 1.4 and once with 1.8, and finally twice without modifying the weight values for smoothing.

For the displacement modifier there are two approaches. The first approach is to translate a selection of vertices in a mesh with a specified translation matrix. The other coming is to displace vertices in a mesh based on the intensity of a texture. Either procedural or image textures can be used.

Displacement can be along a particular local axis or the vertex normal. If a texture is used the separate RGB components can be used to determine the direction and the amount of the displacement.

The mirror modifier creates a mirror image of an object along the local X, Y or Z axis which passes through the object centre. All the vertices and edges in the mesh are duplicated and reflected along the selected axis.

When twisting the mesh, vertices are rotated around a local axis with a certain amount depending of how far the vertex is from the origin. Only selected vertices are affected by the twist modifier. In a 3D modelling tool the selection of vertices is natural - mark and select the vertices. An automatic randomized selection of vertices is more complex. A real life example of twisting would be to wring out a rag.

By bending an object, vertices are repositioned according to a set of rules. A common method is by using a bone structure and bending the object with respect to the bones as their position changes. Bone structures often support twisting as well. A real life example of bending is to bend a wooden (or some other semi-elastic material) stick.

Extrude operates on a selection of faces. What the deformer does is that it create copies of the vertices of the face. The new vertices are translated in the direction of the face normal. Afterwards, new faces are created to keep the model solid. For a triangle face six $(2 \cdot 3)$ new faces are created for the three rectangles. The same face can be extruded several times, and the vertices of the face can change positions with respect to each other in every step. It is possible to tilt the direction in which the extrusion is made, but by default in the direction of the face normal is used. How much each selected face should be extruded can be configured.

Example With the extrude operation a variety of results can be obtained. In the implementation made in this project either all faces or a random number of faces are selected for extrusion. The face is extruded in the direction of the face normal, unless a tilt is added to the extrusion direction. Defining the extrusion length is possible as well as changing the distances between the extruded vertices. The number of extrusions per face is also possible to modify. To the left in Figure 2.8 an object is shown whose faces were extruded multiple times with a tilt of the direction vector as well as a slight change of the distance between vertices. Another variant of extrude is depicted to the right in the same figure (Figure 2.8) where the extrusion length is randomized between a minimum and maximum value. All faces in the mesh have been affected by this modifier in the latter figure.

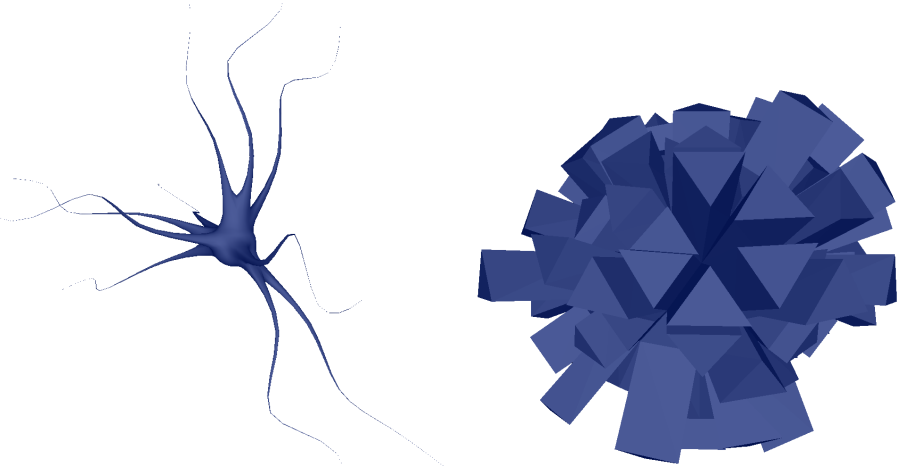


Figure 2.8: The left object is an example of extruding a face multiple times, whereas the right object gives an example for extruding all faces with different lengths.

Combine It is possible to combine objects, by in some way taking their compound. One way is to reuse the same object in different combinations.

An example of a way to combine objects is the array operation. The array operation creates an array of copies of an object along an axis or a curve. Each copy is being offset from the previous one in different ways.

The other major topic when combining objects is *Constructive Solid Geometry*, which is a way to create a complex surface or object by using boolean set operators to combine objects [8]. Objects can be combined with the set operators like union, difference and intersection. Union merges two objects into one. Set difference is the subtraction of one object from another. Boolean intersection is the portion common to two objects.

2.3 Grammatical model generation

A grammar is basically built on the concept of generating objects, which not necessarily need to be geometrical, with the use of rules that replaces primitive parts with more complex sub objects. The sub objects can be anything from strings over graphs to trees

for graphical objects, and the primitive parts that is substituted is often some kind of non-terminal. The traditional kind of grammar is the one that operate on strings. To be able to use that string-based grammar to generate graphical objects we generate strings that consist of symbols with a geometrical purpose. Then it is possible to interpret every generated string as a geometrical object. This type of string generating grammars that have been used most extensively in this context is the Lindenmayer system, which is known as L-system for short.

An L-system is a parallel rewriting system, a type of grammar that has extensively been used to model the growth processes of plants and cell structures. L-systems can also be used to generate self-similar fractals. The definition of a context-free L-system (OL-system) presented by Lindenmayer in [17] is given in Definition 2.3.1 followed by the corresponding definition of derivations in Definition 2.3.2.

Definition 2.3.1. The string L-system, which is said to be context-free, is described by the triplet $G_L = \langle V, \omega, P \rangle$. The non-empty word $\omega \in V^+$ is called the axiom of the L-system. The finite set of production rules is denoted by $P \subset V \times V^*$. A production is a letter a together with a word u and formally written as $a \to u$. The letter a and the word u represents the predecessor and the successor in a production rule. It is required that, for every $a \in V$, there is at least one production of the form $a \to u$ in P. Sometimes we do not explicitly specify a production for a letter a, namely if the production is $a \to a$. This production is known as the identity production. A context free L-system is said to be deterministic if for every $a \in V$ there is exactly one production of the form $a \to u$ in P.

Definition 2.3.2. . Let $\mu = a_1 \dots a_m$ be an arbitrary word over V. The word $v = u_1 \dots u_m \in V^*$ is *directly derived* from μ, denoted $\mu \Rightarrow v$ if $a_i \to u_i$ for every $i = 1, \dots, m$. A word v is generated by G in a derivation of length n if there exists a derivation of words, $\mu_0, \mu_1, \dots, \mu_n$ such that $\mu_0 = \omega$, $\mu_n = v$ and $\mu_0 \Rightarrow \mu_1 \Rightarrow \dots \Rightarrow \mu_n$.

Using this definitions it is possible to specify a table of L-systems, called T0L. $T0L = \langle V, \omega, \{P_1, \dots, P_n\} \rangle$ where every $\langle V, \omega, P_i \rangle$ is a L-system according to Definition 2.3.1. Derivations has the form $\mu_0 \Rightarrow_{P_{i_1}} \mu_1 \Rightarrow_{P_{i_2}} \dots \Rightarrow_{P_{i_n}} \mu_n$ with $i_1, \dots, i_n \in \{1, \dots, n\}$. T0L-Systems is said to be deterministic if every $\langle V, \omega, P_i \rangle$ is deterministic for $i = 1, \dots, n$ [1].

As mentioned, strings generated with L-systems can be interpreted as geometrical objects. How this interpretation is done will be explained in the following subsection.

2.3.1 Visualization

The turtle interpretation provide a simple language for describing 3D models. In this context we only look at the terminal alphabet consisting of the symbols F, f, +, -, ∧, &, \, /, |, [and]. To be able to interpret such a string we have to look at it from left to right and interpret the symbols as an imaginary turtle (or, more technically, a 3D plotter) that views them as instructions to draw an object. The turtle starts at a predefined position, the origin in the Cartesian coordinate system, and orientation in space. The position and orientation of the turtle is known as its status. The interpretation of the symbols is presented below. Length d and angle α is two parameters that is set globally.

[1] The implementation is using one table of rules for the generation of a string, and another table of rules for termination.

F Move forward a step of length d from $\bar{x}$ to $\bar{x}'$. A line segment between $\bar{x}$ and $\bar{x}'$ is drawn.

f Move forward a step of length d without drawing a line.

\+ Turn left by angle α.

- Turn right by angle α.

$\wedge$ Pitch up by angle α.

& Pitch down by angle α.

\ Roll left by angle α.

/ Roll right by angle α.

| Turn around.

[Push state onto stack.

] Pop state off stack.

The symbols [and] makes it possible to implement a branching behaviour. At the beginning [pushes the current state, which contains position and orientation, onto a stack. At the end of the branch] is reached, the saved state is restored and the process continues.

2.3.2 Non-deterministic L-Systems

In the implementation only deterministic L-System is used, but as mentioned L-Systems can be non-deterministic as well. By defining a couple of different rules for the same substitution, where every rule have a specific probability to be chosen, implies that the L-System is non-deterministic.

Example We can use the production rule 3.1 on page 18 from the deterministic L-system as a starting point and create a set of new production rules. The set represent a non-deterministic L-System and the set might look something like this.

$$L \quad \rightarrow_{.50} \quad LF + F + FL \tag{2.7}$$
$$L \quad \rightarrow_{.25} \quad LF - F - FL \tag{2.8}$$
$$L \quad \rightarrow_{.25} \quad LF + L + FL \tag{2.9}$$

Where the subscript to the arrow specifies the probability that this particular rule is selected.

Chapter 3

Accomplishment

This chapter contains some of the thoughts and ideas that has seen the light of the day during the implementation of various techniques described in the previous chapter.

The subsection about *Using the data structure* is about generative modelling.

3.1 Preliminaries

One of the goals has been to create and to run an application for a mobile telephone. To be able to reach that goal the proper tools had to be used.

The device used during implementation is running Windows Mobile[1]. With use of C++, SDL, OpenGL, OpenGL ES and Boost C++ Libraries in Visual Studio 2008 [4, 18, 16, 22, 11] it was possible to develop a demo application for the device.

3.2 Using data structure

When generating objects the half-edge data structure is used, as mentioned in the previous chapter. A good data structure is crucial for generating objects because it is by far the best way to locate specific vertices, edges or faces. To be able to generate objects efficiently it is necessary to use the data structure in the right way.

In order for the data structure to work properly all the pointers have to be set during the generation of objects. This process can be called edge-pairing. There are two ways to pair up edges, sloppy and direct. The two methods will now be described and compared.

Sloppy edge pairing is done like so, the list of edges are traversed in two nestled loops and all the vertices are compared. If the edges are next to each other the vertices should match. This algorithm is roughly of complexity $O(n^2)$ where n is the number of edges. It does not take a rocket scientist to figure out that this is very ineffective.

The direct edge pairing is collecting information about the structure during the generation. In the end of the generation all the pointers are set directly. In a simple scenario when for example creating a box it is quite trivial to pair all edges directly, but further down in the stacking process it becomes more and more complex.

[1] http://www.microsoft.com/windowsmobile/61/default.mspx Information about Windows Mobile 6.1 using WinCE API.

Example For subdivision a direct method for pairing faces have been implemented. As mentioned in the weblog[2] 92160KB of vertex and normal data was generated in 16 seconds, which was a huge optimization compared to using the sloppy method, which took several minutes for less subdivision steps.

3.3 Grammatical generation

The grammatical model generation method used in the implemented system is based on L-system which is described in Section 2.3 on page 14. The grammatical model generator can be called GMG for short. The GMG consist of three parts, a specification, a generator and a visualizer. In this section these three will be described.

3.3.1 Specification and generator

The specification for the GMG consists of parameters and production rules. The File format for the specification is presented in Appendix A. When the parsing of a specification is done the generator generates a string according to the rules obtained. The generator create the string successively by using search and replace-regular expression with boost::spirit [4].

3.3.2 Visualizer

The task of the visualizer is to create a model from the generated string. The string is parsed and interpreted.

A triangular pipe represent the generated structure. For example the grammar with the production rule

$$L \rightarrow LF + F + FL \tag{3.1}$$

with angle 25° iterated five times, with all the L's substituted with F in the end of the run. The width of the triangular pipe is 1.7 units and the length of each pipe segment is 0.7. The model produced is depicted in Figure 3.1.

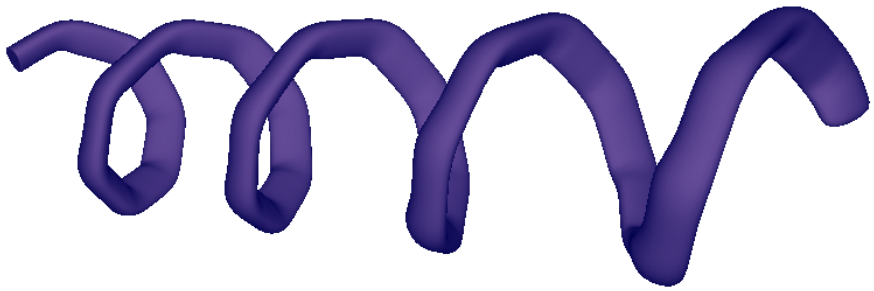


Figure 3.1: An example of a model generated with L-system.

[2] http://proceduralmodeling.wordpress.com/2008/09/19/subdivision-optimized/

Chapter 4

Results

In this chapter the result of the master thesis will be presented. The different parts are *Implementation*, *Example Models* and closes up with a *Discussion*.

4.1 Implementation

Some of the techniques described in this report have been realised and tested on implementation for two platforms. A goal was to create 3D-models for a mobile telephone device, which have been done. When starting developing it is quickly easy to realise that another version for a ordinary computer is needed as well. The purpose for implementing two versions was a matter of speed. With the Software Developer Kit (SDK) for the mobile telephone came an emulator to be able faster run the application created for the device. But this emulator was not efficient and it was clumsy to work with. Because everything that should be implemented was supposed to be fairly general and easy to deploy everywhere the natural choice was to create two versions. In this section these two implementations, and what they do, are presented.

4.1.1 Device

The implementation for the device generates a couple of models and displays them in a grid view as icons for a potential graphical user interface. Every model can be generated with respect to a configuration in plain text and should be easy to connect to some service. Such a service could for example be the Short Message Service, that is very common on mobile telephones.

4.1.2 Computer

For the implementation on the computer there are some differences apart from the Device-implementation. The implementation on the computer is primarily used for just looking at the generated models, generate new models and changing values.

To be able to look at the generated models I used the Arcball technique for rotating the object [20]. The Arcball is the use of spherical coordinates to be able to manipulate the rotation of an object.

Generating new models can be done, as mentioned, with configuration with strings of plain text. This can be used for debugging purposes, to be able to fast recreate a

new, different, model. For larger specifications, as for the L-systems, a file format is used and listed in appendix A.

Changing values in real time is done with simple sliders. These sliders are set through the application programming interface.

4.2 Creating a model

Each model is created through the application programming interface, which consist of a set of tools which will be explained in this section. The primitive generators are listed as follows.

Tetrahedron The primitive shown in Figure 2.4 on page 9 with three vertices and four faces. In the programming interface it is only possible to set the size, which is the length of one edge.

Box An ordinary box are specified either with only size (which is edge length) or height, width and depth in the programming interface

Octahedron The octahedron is depicted in Figure 2.4 on page 9 as well. Again it is only possible to set the edge size in the programming interface.

Icosahedron 12 vertices, 20 faces and 60 half edges makes a icosahedron depicted in Figure 2.4 on page 9.

Cylinder A detailed description of the cylinder is explained in Section 2.2.1 on page 8.

The concept of stacking is to apply a modifier to the generated primitive. It is a natural step after creating a primitive to apply a modifier. There is no limit of how many different modifiers that can be applied to the model. The modifiers implemented in the application programming interface are as follows.

Subdivide The subdivision operation is explained in detail in Section 2.2.2 on page 11. The subdivision function takes two parameters, the first parameter is affecting the β constant in the smoothing step, and the second parameter is a boolean telling if the smoothing step should be done or not. If not the smoothing step is done, the subdivision function is used to increase the tessellation of the object.

Randomize The randomize function displace vertices on the unit sphere spanned by the radius between two vertices randomly. The distance the vertices should be displaced are scaled by the parameter delivered with the function.

Spikeify A special operation that substitute every face on the object with a tetrahedron. The height of the tetrahedron is specified by the parameter.

Extrude The extrude operation is explained in detail in Section 2.2.2 on page 13. A face-selector class is used to specify what faces the extrude operation shall operate on. The two parameters `move` and `tilt` specify how much each extrusion shall be resized each step and how much the face normal shall be tilted. There is another parameter to tell how many extrusions that shall be made, by default there is only one extrusion step. The last option is to specify min and max height for every extrusion, a random height is generated between those two limits.

Vertex displace The vertex displacer select a set of vertices to operate on. The position of the selected vertices are changed.

Grammatically generated models are created according to the file format in Appendix A.

Apart from the Randomize-function there is the class `Randomizer` which chooses from a set of models with some random parameters. It is used for randomly generate a couple of models. It might be a good idea to look at the examples in the class to get a grip of how to use the application programming interface as well.

4.3 Example models

Through the report figures have been displayed, some of these are example models generated by the implementation. The models generated are the subdivided model in 2.7 on page 13, the extruded model in 2.8 on page 14, the corkscrew (L-system) in 3.1 on page 18 and the primitives Tetrahedron, Octahedron and Icosahedron in 2.4 on page 9.

We will now look at some examples of generation of models from the `Randomizer`-class. This will give a brief look at how to use the application programming interface. To start with we shall look at the `Randomizer::primitive(const float dimension)` function. The function produces a random model, just to give some initial primitive to apply modifiers to. The primitive-generator simply select either a Box, Tetrahedron, Octahedron, Icosahedron, Cylinder, or a grammatically generated model. All these, except the grammatically generated model, are specified size-limited by the dimension parameter.

The first example is a simple subdivision.

```
HEmodel_ptr model;                //A model pointer.
model = primitive(1.2f);          //Create a random primitive with
                                  //dimension 1.2 units.
model = model->subdivide(1,true); //Subdivide once without weight to the
                                  //beta-parameter and with the
                                  //smoothing step.
model = model->subdivide(1,true); //Subdivide one more time.
return model;                     //Return model pointer.
```

The second example is showing how to use the vertex randomiser. An example of a hexahedron where the positions of the vertices are randomized is showed in Figure 4.1 on the next page.

```
HEmodel_ptr model;
model = primitive(1.5f);        //Create a random primitive with
                                //dimension 1.5 units.
model = model->randomize(0.4f); //Displace vertices randomly at maximum
                                //40% of the shortest distance between
                                //two vertices.
return model;
```

The third example shows how to use the Spikeify-operation depicted in Figure 4.2 on the following page.

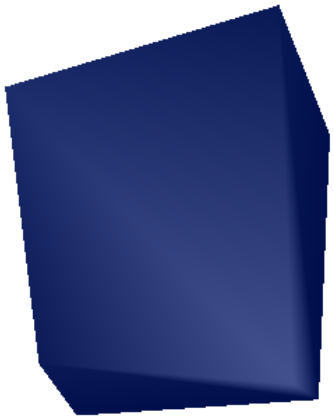


Figure 4.1: A hexahedron where the positions of the vertices are randomized.

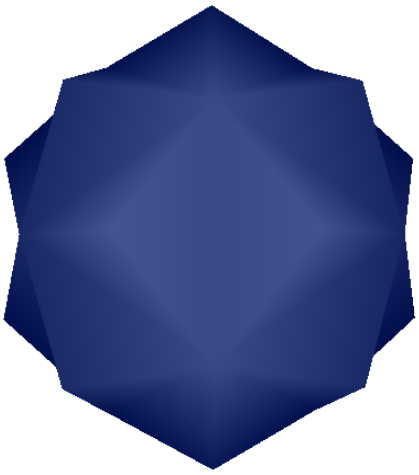


Figure 4.2: An example of how a icosahedron will look after being modified by the Spikeify operation.

```
HEmodel_ptr model;
model = primitive(2.1f);       //Create a random primitive with
                               //dimension 2.1 units.
model = model->spikeify(0.1f); //Replace each face with a tetrahedron
                               //with height 0.1 units.
return model;
```

Example number four is how to use Extrude.

```
HEmodel_ptr model;                    //A model pointer.
```

```
model = primitive(0.7f);              //Create a random primitive with
                                      //dimension 0.7 units.
if(model->getFaces().size() < 40)     //If the created primitive have
    model = model->subdivide(1,true); //less than 40 faces: Subdivide it
Selector_ptr s = New Selector();      //Create a new face selector.
s->setFaces(model->getFaces());       //Tell the selector to use all
                                      //faces in the created model.
s->setMove(0);                        //Specify that the extrusion
                                      //should not change size in the
                                      //different steps, i.e. not move
                                      //the vertices in the direction
                                      //perpendicular to the face
                                      //normal.
s->setTilt(1,1,1);                    //Don't tilt the face normal
                                      //between the extrusions.
const float max = 0.5f + ((float)(std::rand()%1000)/999.0f)*1.0f;
                                      //Set a max length for
                                      //each extrusion.
s->extrudeFaces(model,0,max);         //Extrude the faces of the model
                                      //and set the minimum extrusion
                                      //length to zero and the maximum
return model;                         //to max. Return model pointer.
```

Example five is showing another way to use Extrude.

```
HEmodel_ptr model;                      //A model pointer.
model = primitive(0.4f);                //Create a random primitive with
                                        //dimension 0.4 units.
model = model->subdivide(1,true);       //Subdivide the model once.
model->setNVertices();                  //Tell the model to count it's
                                        //vertices.
Selector_ptr s = New Selector();        //Create new face selector.
s->setFaces(model->getRandomFaces(10)); //Assign 10 random faces
                                        //from the model to
                                        //the face selector.
s->setMove(1.5f);                       //Vertices will move 50%
                                        //perpendicular to the face
                                        //normal.
s->setNExtrusions(4);                   //The selector will extrude four
                                        //times (default is one).
s->setTilt(1.2f,1,1);                   //Tilt the face normal 20%
                                        //in the x-direction.
s->extrudeFaces(model);                 //Extrude faces.
return model;                           //Return model pointer.
```

The sixth example demonstrate how to use the vertex displacer, and an example is depicted in Figure 4.3 on the next page.

```
HEmodel_ptr model;                    //A model pointer.
model = primitive(0.6f);              //Create a random primitive with
                                      //dimension 0.6 units.
```

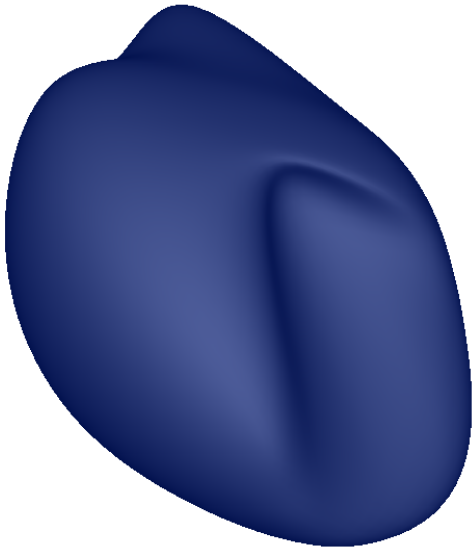


Figure 4.3: This is an example of how the result would be if the vertex displacer is applied on a subdivided octahedron.

```
Selector_ptr s = New Selector();      //Create a face selector.
const int n=std::rand()%model->getNVertices(); // Get n random vertices
s->setVertices(model->getRandomVerts(n)));     // from model and
                                          // assign them to the selector
vr::Matrix m;                         //Create translation matrix.
const float trans[3] = {              //Randomized.
0.5f + ((float)(std::rand()%1000)/999.0f)*1.0f,
0.5f + ((float)(std::rand()%1000)/999.0f)*1.0f,
0.5f + ((float)(std::rand()%1000)/999.0f)*1.0f};
m.makeTranslate(trans[0],trans[1],trans[2]);
s->addTransform(m);                   //Assign the translation to
                                      //the selector and
s->apply();                           //apply the displacement.
model = model->subdivide(1,true);     //Subdivide twice to make
model = model->subdivide(1,true);     //the model look smoother.
return model;                         //Return model pointer.
```

The seventh and last modifier example show how to make use of adding a weight to the β-function in the smoothing step.

```
HEmodel_ptr model;                  //A model pointer.
model = primitive(5.5f);            //Create a random primitive with
                                    //dimension 5.5 units.
const float val =                   //Chose a random value.
((float)(std::rand()%1000)/999.0f)*4.0f;
model = model->subdivide(val,true); //Subdivide twice with
model = model->subdivide(val,true); //weight to beta.
```

```
model = model->subdivide(1,true);   //One ordinary subdivision.
return model;                       //Return model pointer.
```

These where some examples of how to use the modifier functionality. The primitive generators are intuitive, and will not be mentioned here. Grammatical model generation on the other hand is a primitive generator as well. One example will show how to use the grammatical model generator shall be used.

```
std::stringstream ss; //Create a stringstream for the configuration.
ss << "0.2 ";      //Specify the length of each graphical
                   //segment to 0.2.
ss << "0.3 ";      //Set the width of a segment to 0.3.
ss << "4 ";        //Number of rewriting iterations will be four.
ss << "25 ";       //Rotation angle for every bend.
ss << "L ";        //Starting symbol for the grammar.
ss << "1 ";        //Number of production rules.
ss << "L ";        //First rule: Substitute every "L" with
ss << "LF&F&FL ";  //"LF&F&FL".
ss << "1 ";        //Number of substitutions.
ss << "L ";        //Replace every "L" in the final
ss << "F";         //string with the letter "F".
Turtle_ptr t = New Turtle();         //Create a turtle for the
                                     //visualisation.
Generator_ptr g = New Generator(ss); //Generate a string from the rules.
t->setRotation(g->getRotation());    //Tell the turtle about rotation,
t->setLength(g->getLength());        //length and scaling for the
t->setScaling(g->getScaling());      //grammar.
return t->parser(g->string());       //Parse the generated string with
                                //the turtle and return the final model
```

4.4 Discussion

Another result of this master thesis that I need to mention is the usage of the two different techniques; stacking and grammatical modelling. In my opinion stacking is easier to work with for generating models. It is much easier to keep control of the generation of the object. By changing a parameter the model changes naturally linear against it. Grammatical modelling with L-systems are not as intuitively to work with. By changing some parameter just a little the generated model can change shape completely. This can of course be a big strength, but in my opinion it is a drawback.

Chapter 5

Conclusions

This is the ending chapter of the master thesis. The goal to present 3D model-icons with procedural modelling have been reached. Restrictions, limitations and future work will be discussed in this chapter.

5.1 Restrictions and limitations

In this section the restrictions and limitations for the application is presented. Some of these are of course possible to work on in the future. The primary limitations are that the appearance of a model is very subjective and that models are converted to raw data.

5.1.1 Subjective models

When generating a model randomly the appearance is more or less random. It is difficult to decide if a model is looking good or not when generating it. This is a highly subjective matter and is difficult to control.

This is of course possible to control by letting a person deciding which models to keep and which to discard by for example assigning a score to every model.

5.1.2 Raw data

Once each model is generated the mesh is converted to raw data and transferred to the graphics card to save memory. Only information about the bounding volume and translations is not discarded. This is a limitation if we for example want to add an additional modifier.

5.2 Future work

Is there any future in procedural modelling? The short answer is definitely yes. The motivations are divided in two parts, quality and functionality, and are presented in this section. This section will end up with a discussion about combining stacking and grammatical modelling.

5.2.1 Qualities

It is possible to use procedural modelling commercially. As mentioned it will be possible to use it for generating a graphical user interface. Procedural modelling is extensively used foremost in the gaming industry, which is very big and profitable.

5.2.2 Functionality

Some suggestions for extensions have come up, these suggestions are listed here:

I **Looks and colour** Procedural textures is used to give a model an even better look.

II **Sharing** It will be possible to share settings to friends and even be able to combine two graphical user interfaces to create a new one. This can be solved with a genetic algorithm e.g. The goal is to be able to describe the whole user interface with only one SMS.

III **Ordering** How the icons are placed in relationship to each other is generated procedural as well. With the correct configuration the same behaviour is obtained every time.

IV **Animations** Some icons will move and eventually change shape in real-time, this will be described with a simple key as well.

V **Randomize** It will be possible to create a brand new graphical user interface randomly.

VI **External interaction** Most mobile telephones have built in media players. The graphical user interface will be affected in real-time of the music played.

VII **Edit** In some, limited, sense it will be possible to change the looks of the graphical user interface in a controlled way.

5.2.3 Combining stacking and grammatical modelling

The idea of combining the concepts of stacking and grammatical modelling is interesting (although it has not been implemented in this project). The first idea is to use a grammatical model generator instead of an extrusion. The other idea is to replace a terminal symbol in a grammatically generated model with a complex object. The two ideas are quite similar, the difference is that the first one start off with a non-grammatical generated model and apply grammatically generated parts to it. In the other idea the initial model is generated grammatically and the endpoints (terminals) are substituted with non-grammatical generated models. These ideas are explained further in this section.

Grammatical generation instead of extrude An ordinary extrude, as explained in 2.2.2 on page 13, which is applied on a single face can be replaced by a grammatically generated model. The grammatically generated models often have tree like structures which reminds very much of the models obtained with extrude.

The principle is to select one or more faces on a primitive and use it as the starting symbol in the grammatically generated model. In this combination the face is used as a starting point for the generation.

Which grammatical generated model should be used instead of a extrusion? It is possible to have a selection of predefined grammatical generated models to choose from and extrude the faces with that system. It might be a option for the users to specify their own models as well.

The overall size of the grammatically generated model will probably be dependent on the size of the starting symbol. If the faces are very small, because of e.g. subdivision, the starting symbol will be very small as well. This will result in a very (to the relative size) tiny model. The solution to this is to select three vertices at a suitable distance from each other. A suitable distance is the one that have an area that is larger than a certain threshold. The three vertices are selected and a new face is created. The small faces inside the new faces will be removed. A grammatical model will be generated with that new, larger, will take place from the new larger face.

Change terminal in grammatical generation A grammatical generated model produces a set of terminal symbols. This terminal symbol is often represented by one very simple object. The idea here is to extend the set of objects representing the terminal symbol.

One approach for generating objects as terminal symbols, is to add the functionality to the visualizer. When the visualizer reaches a terminal symbol the object to be used is generated. When generating an object a couple of parameters have to be be specified, defining e.g. orientation, position and size. The orientation and position can depend on the current state in the visualization. What about the size? It is probably not feasible to specify the size for each individual terminal symbol. This is too much to keep track of. The solution to this is that we end up with two options. The first option is to let all the objects have the same fixed size. The other solution is to randomize the size within some suitable interval.

The method for extending the set of terminals with objects can probably be applied for non-terminals as well, as long as all the geometrical rules are are fulfilled.

Chapter 6

Acknowledgements

I would like to give acknowledgements to my supervisor Frank Drewes at the department of Computing Science, to Mats Thor and all the people at Tactel. My friends deserve a great deal of credit as well. Finally I want to give appreciation to my girlfriend and my family for great support.

References

[1] T. Akenine-Moller, T. Moller, and E. Haines. *Real-Time Rendering*. AK Peters, Ltd. Natick, MA, USA, 2002.

[2] P. Benko, R.R. Martin, and T. Varady. Algorithms for reverse engineering boundary representation models. *Computer-Aided Design*, 33(11):839–851, 2001.

[3] M. De Berg, O. Cheong, and M. van Kreveld. *Computational geometry: algorithms and applications*. Springer, 2008.

[4] J. de Guzman et al. The Boost Spirit Library.

[5] F. Drewes. *Grammatical picture generation: a tree-based approach*. Springer, 2006.

[6] D.S. Ebert, S. Worley, F.K. Musgrave, D. Peachey, K. Perlin, and K.F. Musgrave. *Texturing and modeling: a procedural approach*. Academic Press, Inc. Orlando, FL, USA, 1998.

[7] MC Escher. *Graphic Work of MC Escher*. Ballantine Books, 1971.

[8] J.D. Foley. *Computer graphics: principles and practice*. Addison-Wesley Professional, 1995.

[9] S. Havemann and D.W. Fellner. *Generative mesh modeling*. Braunschweig, 2005.

[10] Ø. Hjelle and M. Dæhlen. *Triangulations and applications*. Springer Verlag, 2006.

[11] S. Lantinga. Simple directmedia layer, 2005.

[12] C. Loop. Smooth subdivision surfaces based on triangles. *Master's thesis, University of Utah, Department of Mathematics*, 1987.

[13] RR Martin and PC Stephenson. Sweeping of three-dimensional objects. *COMP. AIDED DES.*, 22(4):223–234, 1990.

[14] Y.I.H. Parish and P. Müller. Procedural modeling of cities. In *Proceedings of ACM SIGGRAPH 2001*, pages 301–308, 2001.

[15] A. Pasko, V. Adzhiev, A. Sourin, and V. Savchenko. Function representation in geometric modeling: concepts, implementation and applications. *The Visual Computer*, 11(8):429–446, 1995.

[16] D.S. Platt. *Introducing Microsoft. Net*. Microsoft Press Redmond, WA, USA, 2002.

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

[18] K. Pulli, J. Vaarala, V. Miettinen, T. Aarnio, and K. Roimela. *Mobile 3D Graphics: With OpenGL ES and M3G*. Morgan Kaufmann, 2007.

[19] W.J. Schroeder. Modeling of surfaces employing polygon strips, October 1 1996. US Patent 5,561,749.

[20] K. Shoemake. ARCBALL: a user interface for specifying three-dimensional orientation using a mouse. In *Graphics Interface*, volume 92, pages 151–156, 1992.

[21] C. Smith. *On Vertex-Vertex Systems and Their Use in Geometric and Biological Modelling*. PhD thesis, The University of Calgary, April 2006.

[22] B. Stroustrup. The C++ Programming Language, Special Ed, 1997.

[23] L. Wilkinson. *The grammar of graphics*. Springer, 2005.

Appendix A

File format

The file format describing an ET0L-system for the model generation is straightforward. In the list below each row is presented and what it specifies.

I Global setting for the length of each graphical segment in the visualization.

II Global setting for the width of each graphical segment in the visualization.

III Number of iterations for the grammar.

IV Rotation angle in every bend in the visualization.

V The axiom - the starting symbol for the grammar.

VI Number of production rules in the table.

VII Predecessor letter in the production rule.

VIII The successor word in the production rule.

IX Number of substitutions. These last three rows to specify which letters in the alphabet that should be substituted by symbols readable by the visualizer.

X Substitute all the occurrences of this symbol with the symbol on the next row.

XI The symbol to be substituted.

There is a format[1] for describing stacking as well. This is represented by a string. Every letter in the string have a function. These letters are listed and their function is explained below. Every letter take a real value as parameter. If no parameter is added it defaults to 1.0. For the primitives the parameter specifies the size.

b Create the new primitive box.

t Create the new primitive tetrahedron.

o Create the new primitive octahedron.

c Create the new primitive cylinder.

[1] Extrude etc. is too complex to specify in this format.

r Randomize vertices on the unit sphere spanned by the radius between two vertices. Scaled by the parameter.

x Change the seed value used in the randomizing function.

s Subdivide on time. The parameter specifies the weight multiplied in the smoothing step.

w Spikeify. Substitute every face with a tetrahedron with height specified by the parameter.