

# Non-interactive Modeling Tools and Support Environment for Procedural Geometry Generation



Submitted in partial fulfilment  
of the requirements of the degree  
of Master of Science  
of Rhodes University

Chantelle Morkel

January 2006

## **Abstract**

This research examines procedural modeling in the field of computer graphics. Procedural modeling automates the generation of objects by representing models as procedures that provide a description of the process required to create the model.

The problem we solve with this research is the creation of a procedural modeling environment that consists of a procedural modeling language and a set of non-interactive modeling tools.

A goal of this research is to provide comparisons between 3D manual modeling and procedural modeling, which focus on the modeling strategies, tools and model representations used by each modeling paradigm.

A procedural modeling language is presented that has the same facilities and features of existing procedural modeling languages. In addition, features such as caching and a pseudo-random number generator is included, demonstrating the advantages of a procedural modeling paradigm.

The non-interactive tools created within the procedural modeling framework are selection, extrusion, subdivision, curve shaping and stitching.

In order to demonstrate the usefulness of the procedural modeling framework, human and furniture models are created using this procedural modeling environment.

Various techniques are presented to generate these objects, and may be used to create a variety of other models. A detailed discussion of each technique is provided.

Six experiments are conducted to test the support of the procedural modeling benefits provided by this non-interactive modeling environment. The experiments test, namely parameterisation, re-usability, base-shape independence, model complexity, the generation of reproducible random numbers and caching.

We prove that a number of distinct models can be generated from a single procedure through the use parameterisation. Modeling procedures and sub-procedures are re-usable and can be applied to different models. Procedures can be base-shape independent. The level of complexity of a model can be increased by repeatedly applying geometry to the model. The pseudo-random number generator is capable of generating reproducible random numbers. The caching facility reduces the time required to generate a model that uses repetitive geometry.

## **Acknowledgements**

First and foremost, I thank God for providing me with the opportunity to return to Rhodes and do my masters.

To my parents and Jock, thank you for always believing in me and standing by me through the good and the bad and for always urging me to work harder. Mom, thank you, Dad, rest in peace and Jock, you rock my world!

I would like to thank my supervisor, Professor Shaun Bangay, for his time, patience and constant re-assurance that this research was still on track. This document would not be in existence had it not been for him.

To the people that have gotten me through this year, I would like to thank you all for always believing in me and not allowing me to give up. Special thanks go to Marie Dercksen, Candice Egan, Adele Lobb, Hannah Slay, Zukhanye Kwinana, Jo Janse van Rensburg, Danny Zhao and Merle Naidoo.

Extra special thanks go to Kevin Glass - you're my hero, Ellen Ku - for always being there for me, standing by me and for all the hours spent helping me decipher what I have written and Dominic White - for always encouraging me and providing ample entertainment.

Thanks to the Computer Science Department at Rhodes University and to Eskom for allowing me to come back to do masters and for providing power for our computers.

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Problem Statement . . . . .	4
1.2	Background . . . . .	4
1.3	Motivation . . . . .	5
1.3.1	The Benefits of Procedural Approaches . . . . .	7
1.3.1.1	Support Environment . . . . .	7
1.3.1.2	Re-usability and Model Complexity . . . . .	9
1.3.1.3	Database Amplification . . . . .	10
1.3.1.4	Base-shape Independence . . . . .	10
1.3.1.5	Caching . . . . .	10
1.3.1.6	Animation . . . . .	11
1.3.1.7	Random Numbers . . . . .	11
1.4	The Problem Statement Revisited . . . . .	12
1.4.1	The Support Environment . . . . .	12
1.4.2	The Non-Interactive Modeling Tools . . . . .	12
1.4.3	Research Objectives . . . . .	13
1.5	Terminology . . . . .	15
1.6	Thesis Structure . . . . .	15
<b>2</b>	<b>Related Work</b>	<b>18</b>
2.1	Modeling . . . . .	19
2.1.1	Manual 3D Modeling Strategies . . . . .	19
2.1.1.1	Box Modeling . . . . .	20
2.1.1.2	Flat Mesh Modeling . . . . .	20
2.1.1.3	Patch Modeling and Lofting . . . . .	20
2.1.1.4	Curve Modeling . . . . .	21



2.1.1.5	Deformation Modeling . . . . .	21
2.1.2	Procedural Modeling . . . . .	22
2.1.2.1	Lindenmayer Systems . . . . .	22
2.1.2.2	Fractals . . . . .	23
2.1.2.3	Fractional Brownian Motion . . . . .	24
2.1.2.4	Particle Systems . . . . .	24
2.1.2.5	Compositing Geometry . . . . .	25
2.1.2.6	Detail Modeling . . . . .	26
2.1.2.7	Deformation Modeling . . . . .	26
2.1.3	Applications of Procedural Modeling . . . . .	27
2.1.3.1	Organic Objects . . . . .	27
2.1.3.2	Man-made Objects . . . . .	30
2.1.3.3	Phenomena . . . . .	31
2.1.3.4	Summary . . . . .	32
2.2	Modeling Languages . . . . .	32
2.2.1	3D Modeling Packages - Scripting . . . . .	32
2.2.2	Procedural Modeling Languages . . . . .	33
2.2.2.1	Modeling Environment (Menv) . . . . .	34
2.2.2.2	Animation Language (AL) . . . . .	35
2.2.2.3	Interactive Animation . . . . .	36
2.2.2.4	Authoring Solid Models . . . . .	37
2.2.2.5	The Renderman Shading Language . . . . .	38
2.2.2.6	Summary . . . . .	38
2.3	Tools and Representations . . . . .	39
2.3.1	Interactivity . . . . .	39
2.3.2	Modeling Tools . . . . .	41
2.3.2.1	3D Modeling Package Tools . . . . .	41
2.3.2.2	Procedural Modeling Constraints . . . . .	43
2.3.2.3	Procedural Modeling Tools . . . . .	43
2.3.3	Model Representations . . . . .	44
2.3.3.1	3D Modeling Package Representations . . . . .	44
2.3.3.2	Procedural Modeling Representations . . . . .	44
2.4	Summary . . . . .	45

<b>3</b>	<b>The Procedural Modeling Language</b>	<b>46</b>
3.1	Requirements . . . . .	46
3.1.1	Procedural Modeling Requirements . . . . .	47
3.1.1.1	Modeling Strategies . . . . .	47
3.1.1.2	Parameterisation . . . . .	47
3.1.1.3	Level of Detail and Lazy Evaluation . . . . .	48
3.1.1.4	Data Structures, Looping Constructs and Control Constructs . .	48
3.1.1.5	Variables . . . . .	49
3.1.1.6	A Pseudo-Random Number Generation Facility . . . . .	49
3.1.1.7	Caching . . . . .	49
3.1.2	Supported Procedural Modeling Benefits . . . . .	50
3.1.2.1	Re-usability . . . . .	50
3.1.2.2	Increased Model Complexity . . . . .	50
3.1.2.3	Base-Shape Independence . . . . .	50
3.2	Design . . . . .	50
3.2.1	Standardisation . . . . .	51
3.2.1.1	Bounding Volumes . . . . .	51
3.2.1.2	Procedure Declaration . . . . .	52
3.2.1.3	Body . . . . .	53
3.2.2	Flags . . . . .	53
3.2.3	Mode . . . . .	53
3.2.4	Parameterisation . . . . .	54
3.2.5	Level of Detail and Lazy Evaluation . . . . .	55
3.2.6	Data Structures, Looping Constructs and Control Constructs . . . . .	55
3.2.7	Variables . . . . .	56
3.2.8	A Pseudo-Random Number Generation Facility . . . . .	56
3.2.9	Caching . . . . .	56
3.3	Implementation . . . . .	57
3.3.1	Implementation Decisions . . . . .	57
3.3.2	Standardisation . . . . .	57
3.3.3	Parameterisation . . . . .	58
3.3.3.1	Parameter Rewriting and Default Parameter Values . . . . .	58
3.3.3.2	Parameter Signatures . . . . .	60
3.3.4	Procedure Call Rewriting . . . . .	60

3.3.5	Level of Detail and Lazy Evaluation . . . . .	61
3.3.6	A Pseudo-Random Number Generation Facility . . . . .	62
3.3.7	Caching . . . . .	62
3.4	Conclusions . . . . .	64
3.4.1	Summary . . . . .	64
3.4.2	Conclusions . . . . .	65
<b>4</b>	<b>Procedural Modeling Tools</b>	<b>66</b>
4.1	Tool Specifications . . . . .	67
4.1.1	Experiment . . . . .	67
4.1.2	Comparison of Strategies . . . . .	70
4.1.3	Selection . . . . .	70
4.1.4	Design . . . . .	70
4.1.4.1	Indices . . . . .	71
4.1.4.2	Labels . . . . .	72
4.1.4.3	Regions . . . . .	72
4.1.4.4	Summary . . . . .	73
4.1.5	Extrusion . . . . .	75
4.1.5.1	Design . . . . .	75
4.1.6	Curve Shaping Tool . . . . .	76
4.1.6.1	Design . . . . .	76
4.1.7	Subdivision . . . . .	76
4.1.7.1	Subdivision Schemes . . . . .	77
4.1.7.2	Design . . . . .	78
4.1.8	Stitching . . . . .	79
4.1.8.1	Design . . . . .	79
4.2	Data Structures . . . . .	79
4.2.1	Model Representation . . . . .	80
4.2.1.1	Design . . . . .	80
4.2.2	Selection Representation . . . . .	81
4.2.2.1	Design . . . . .	81
4.3	Implementation . . . . .	81
4.3.1	Tools . . . . .	82
4.3.1.1	Selection . . . . .	82
4.3.1.2	Extrusion . . . . .	82

4.3.1.3	Curve Shaping . . . . .	84
4.3.1.4	Subdivision . . . . .	85
4.3.1.5	Stitching . . . . .	86
4.3.2	Data Structures . . . . .	89
4.3.2.1	Model Representation . . . . .	89
4.3.2.2	Selection Representation . . . . .	90
4.4	Conclusions . . . . .	91
4.4.1	Summary . . . . .	91
4.4.2	Conclusions . . . . .	92
<b>5</b>	<b>Applications of Procedural Modeling</b>	<b>93</b>
5.1	Procedural Modeling Tools . . . . .	93
5.1.1	Selection . . . . .	94
5.1.2	Extrusion . . . . .	96
5.1.3	Curve Shaping Tool . . . . .	96
5.1.4	Subdivision . . . . .	97
5.1.5	Stitching . . . . .	98
5.1.6	Models . . . . .	99
5.1.6.1	Organic Objects . . . . .	99
5.1.6.2	Man-Made Objects . . . . .	100
5.2	Model Creation . . . . .	101
5.2.1	Hierarchical Models . . . . .	101
5.2.2	Human Modeling . . . . .	101
5.2.2.1	Requirements . . . . .	102
5.2.2.2	Design . . . . .	102
5.2.2.3	Implementation . . . . .	104
5.2.2.4	Results . . . . .	105
5.2.3	Furniture . . . . .	107
5.2.3.1	Requirements . . . . .	108
5.2.3.2	Design . . . . .	108
5.2.3.3	Implementation . . . . .	109
5.2.3.4	Results . . . . .	109
5.3	Discussion . . . . .	113

<b>6</b>	<b>Results</b>	<b>116</b>
6.1	Experiment 1 - Parameterisation . . . . .	117
6.1.1	Objective . . . . .	117
6.1.2	Design . . . . .	117
6.1.3	Implementation . . . . .	118
6.1.4	Results . . . . .	118
6.2	Experiment 2 - Re-usability . . . . .	121
6.2.1	Objective . . . . .	121
6.2.2	Design . . . . .	121
6.2.3	Implementation . . . . .	121
6.2.4	Results . . . . .	122
6.3	Experiment 3 - Base-shape Independence . . . . .	124
6.3.1	Objective . . . . .	124
6.3.2	Design . . . . .	124
6.3.3	Implementation . . . . .	125
6.3.4	Results . . . . .	125
6.4	Experiment 4 - Model Complexity . . . . .	128
6.4.1	Objective . . . . .	128
6.4.2	Design . . . . .	128
6.4.3	Implementation . . . . .	129
6.4.4	Results . . . . .	129
6.5	Experiment 5 - Reproducible Randomness . . . . .	130
6.6	Experiment 6 - Caching . . . . .	131
6.6.1	Objective . . . . .	131
6.6.2	Design . . . . .	131
6.6.3	Implementation . . . . .	132
6.6.4	Results . . . . .	132
6.7	Conclusion . . . . .	135
<b>7</b>	<b>Conclusions</b>	<b>136</b>
7.1	Challenges . . . . .	136
7.2	Objective 1 - Review of Related Work . . . . .	138
7.2.1	Summary . . . . .	138
7.2.2	Conclusions . . . . .	139

7.3	Objective 2 - Determine Whether a Procedural Modeling Language can be Created that Encompasses the Requirements and Benefits of Procedural Modeling . .	140
7.3.1	Summary . . . . .	140
7.3.2	Conclusions . . . . .	140
7.4	Objective 3 - The Requirements Specification, Design and Implementation of the Procedural Modeling Language . . . . .	140
7.4.1	Summary . . . . .	140
7.4.2	Conclusions . . . . .	141
7.5	Objective 4 - The Design and Implementation of a Set of Procedural Modeling Tools . . . . .	141
7.5.1	Summary . . . . .	141
7.5.2	Conclusions . . . . .	142
7.6	Objective 5 - The Application of the Procedural Modeling Tools in Model Creation	142
7.6.1	Summary . . . . .	142
7.6.2	Conclusions . . . . .	143
7.7	Objective 6 - Experiments to Test the Support of the Procedural Modeling Environment . . . . .	143
7.7.1	Summary . . . . .	143
7.7.2	Conclusions . . . . .	144
7.8	Conclusions . . . . .	144
7.9	Research Contributions . . . . .	144
7.10	Future Work . . . . .	145
<b>A</b>	<b>Bison Grammar</b>	<b>146</b>
<b>B</b>	<b>Colour Models</b>	<b>148</b>
<b>C</b>	<b>Examples of Modeling Procedures</b>	<b>150</b>
C.1	Human Modeling Procedure . . . . .	150
C.2	Torso Shaping Procedure . . . . .	152
C.3	Foot Modeling Procedure . . . . .	154
C.4	Chair Procedure . . . . .	156
C.5	Luggage Modeling Procedure . . . . .	159
<b>D</b>	<b>Code for Selected Procedural Modeling Tools</b>	<b>162</b>
D.1	Selection . . . . .	162

D.2 Curve Shaping . . . . .	163
D.3 Extrusion . . . . .	164

# List of Figures

2.1	An L-system tree. Image courtesy of Kevin Glass . . . . .	23
2.2	A fractional Brownian motion displacement shader . . . . .	24
2.3	Vegetation made from particle systems . . . . .	25
2.4	The existing procedural modeling techniques based on the level of proceduralism and interactivity . . . . .	40
4.1	Examples of the human models created during the modeling experiment . . . . .	68
4.2	The incorrect (left) and correct (right) removal of extruded faces . . . . .	71
4.3	An elephant model enclosed within its bounding volume . . . . .	75
4.4	Examples of selections . . . . .	83
4.5	Original donkey (a), 1 level of subdivision (b) and 3 levels of subdivision (c) . . .	86
4.6	Subdivision with a smoothing factor of 1.0 (top), 0.5 (middle) and 0.0 (bottom) .	87
4.7	The selection from each model is shown in yellow . . . . .	87
4.8	Deciding which direction to traverse the contours in . . . . .	88
4.9	An example of stitching two models together . . . . .	89
5.1	Examples of selection, using the whole model a) and using a named selection b) .	95
5.2	Examples of selections. The regions in yellow demarcate selections . . . . .	95
5.3	The same set of selection ratios applied at different levels of detail . . . . .	96
5.4	A chair model created by extruding a single cube to form the base, legs, back and cushion . . . . .	97
5.5	Curve shaping applied to varying levels of detail . . . . .	97
5.6	Different levels of subdivision resulting in a reduction of model size and different degrees of model smoothness . . . . .	98
5.7	An example of using values outside the range $[0; 1]$ for the smoothness operator .	98
5.8	An example of stitching . . . . .	99
5.9	The human figure (by Loomis) . . . . .	103



5.10	Human model - front and side views . . . . .	108
5.11	A scene containing chairs, tables and the luggage . . . . .	112
6.1	A human model with the abdomen and buttocks shaping parameters changed . .	119
6.2	A group of human models created by using random height, abdomen and but- tocks values . . . . .	120
6.3	Human models with different height, abdomen and buttocks parameter values . .	120
6.4	A human model with legs and feet . . . . .	122
6.5	The luggage model with legs and feet . . . . .	122
6.6	A chair and couch model with human legs . . . . .	123
6.7	A chair and couch model with standard legs that have been shaped using the leg shaping procedure . . . . .	123
6.8	A human model . . . . .	125
6.9	The "Humpty-Dumpty" model created from an egg-shaped primitive . . . . .	126
6.10	Chair models from different base-shapes . . . . .	126
6.11	A chair model created from a disc base-shape . . . . .	127
6.12	Table models . . . . .	127
6.13	Luggage model with 6, 12, 18, 24 and 30 legs respectively. . . . .	129
6.14	L-system trees with 3, 4, 5 and 6 levels of recursion . . . . .	130
6.15	Models used in caching experiments . . . . .	132
B.1	Textured human model . . . . .	148
B.2	Textured human model with different parameter values for stomach and buttocks	149
B.3	Textured "Humpty-Dumpty" model . . . . .	149
B.4	Textured luggage . . . . .	149

# List of Tables

2.1	Applications of procedural modeling . . . . .	32
4.1	The Modeling Results for Participants 1 - 8 . . . . .	69
4.2	A comparison of advantages and disadvantages associated with each extrusion approach . . . . .	70
4.3	The advantages and disadvantages of each selection strategy . . . . .	74
4.4	Various Subdivision Schemes . . . . .	77
6.1	The number of polygons in the luggage model increases as the number of legs increases . . . . .	129
6.2	Average time taken to generate a model in seconds . . . . .	133
6.3	Average time taken to generate a model in seconds, with additional experiments to test new hypotheses . . . . .	134

# Chapter 1

## Introduction

### 1.1 Problem Statement

The problem we address is the creation of a development environment for performing procedural modeling, along with modeling tools adapted for such a non-interactive environment.

The development environment must provide support for modeling operations and processes that retain the benefits of procedural modeling.

The modeling tools required must function without run-time human intervention, but support well-known and understood modeling strategies.

The aim of this research is thus to create a modeling language and set of algorithmic tools to provide a non-interactive alternative for model construction.

Existing procedural techniques have several desirable benefits associated with them. The benefits that we want to incorporate into our development environment are abstraction, variable levels of detail, increased model complexity, database amplification, pseudo-randomness and re-usability, parameterisation, base-shape independence and caching.

### 1.2 Background

The complexity of computer generated models is steadily increasing. The modeling of characters and objects is typically performed manually through the use of modeling packages. This is a time-consuming process that is exacerbated by the need to create highly detailed and complex models. As a model grows in complexity, the amount of storage space required to store the model also grows.

Computer generated models are designed to play a specific role or perform a specific task

within a scene or context, which does not allow the models to be readily re-usable in other contexts or scenarios. The models can be adapted to conform to the new environment, but this is done at the cost of extra effort and time to the modeler.

Procedural modeling is capable of alleviating these problems. Procedural modeling uses scripted process to automate the generation of models. The procedures used contain all of the information required to create a complete model. The necessary calculations and transformations are incorporated into each procedure.

Procedural modeling focuses on the automation of model creation. This solves the problem of having to create models by hand, as models can be created algorithmically. A procedure to model an object is written, and is called once the model needs to be generated.

The complexity of a model equates to how intricately an object has been modeled. Such detail can be added in a procedural manner, by repeatedly applying a tool or refinement operation.

The procedures used to generate models are a lot smaller, in size, than the models they create. Irrespective of how complex a model is, the model generation process is stored, not the actual model. These procedures are evaluated as they are needed.

The procedures used to generate models are parameterised. This parameterisation enables a modeler to generate different versions of the same model and provides a means of re-using the models.

Section 1.3 discusses each of the benefits of procedural modeling, and identifies why each of these benefits is advantageous to procedural modeling. Each of the benefits is realised through the use of modeling facilities and a stable support environment.

## **1.3 Motivation**

This section looks at the four computer graphics fields that have benefited from the use of procedural approaches. These fields are procedural shading, procedural texture synthesis, procedural modeling and procedural scripting or animation.

### **Procedural shading**

Shaders are procedures that are used to determine the colour of each point on a surface [Apodaca and Gritz, 2000, Olano and Lastra, 1998, Upstill, 1993] and are widely used. Shaders are also responsible for calculating the interaction between light sources in a scene and the surface that the shader is being applied to.

The use of procedural shading means that procedures can be modified and tweaked to produce the desired results [Olano and Lastra, 1998]. The introduction of randomness into a shader creates an element of chaos on the surface that is being shaded. By doing this, the surface looks more realistic. Since shaders are concerned with point colours and the interaction of lights with the surface model, a change in the viewing parameters causes the shaded surface to change.

### **Procedural Texture Synthesis**

Procedural texture synthesis creates textures through the use of procedures. The synthesis process is not reliant on the use of a sample texture [Qin and Yang, 2002]. An advantage of using procedural texture synthesis is that the resulting texture does not have any seams that need to be concealed, nor are there any visible discontinuities in the texture.

A disadvantage of procedural texture synthesis is that small changes to parameter values can have a big impact on the resulting texture. This makes the process unpredictable.

There are several forms of procedural texture synthesis, which include cellular texture [Legakis et al., 2001], reaction-diffusion [Turk, 1991, Witkin and Kass, 1991], noise [Perlin, 1985] and hypertexture [Perlin and Hoffert, 1989].

The non-procedural texture synthesis methods use sample images, which are replicated over the surface of a model [Bangay and Morkel, 2006, Bonet, 1997, Turk, 2001, Wei and Levoy, 2000].

### **Procedural Scripting**

Procedural scripting uses algorithmic descriptions of movement to animate a model [Thalmann and Thalmann, 1996]. SoftImage software is capable of creating and procedurally controlling individual actors and crowds for motion pictures and games. Piccolo, is a scripting environment specifically created for the purpose. Actors are loaded and their actions dictated by Piccolo through the use of commands and a hierarchical finite state machine [Koga et al., 2004].

Procedural scripting is also used to assign random movements to models during periods of inactivity [Perlin and Goldberg, 1996].

### **Procedural modeling**

Procedural modeling automates the model creation process, by describing modeling processes algorithmically [Cutler et al., 2002]. The cost of procedural modeling is associated with the creation of the modeling procedure and not with model generation. Model descriptions are contained within procedures [Cutler et al., 2002, May et al., 1996, Reeves et al., 1990] and are

evaluated as required [Hart, 1994]. This provides savings in both computational resources, as procedures are only evaluated when required and storage space as models are stored as procedures in scripts.

Procedural modeling promotes model and procedure re-usability through the use of parameterisation. A number of different variations of a model can be created from a single modeling procedure.

### **1.3.1 The Benefits of Procedural Approaches**

This section looks at the various benefits of procedural techniques. The discussion has been categorised into benefits directly associated with the support environment and the remaining individual benefits.

The benefits catered for by the support environment include aspects such as abstraction, parameterisation, level of detail and model evaluation.

The remaining benefits include re-usability, model complexity, database amplification, base-shape independence, caching, animation and random numbers.

#### **1.3.1.1 Support Environment**

This section reviews the components of each of the procedural approaches that support the creation of a versatile support environment. The purpose of this section is to create a comprehensive list of attributes that can be used to create a versatile support environment.

One of the attributes associated with the use of a support environment is abstraction. The support environment is responsible for hiding implementation specific details from the user, while simultaneously providing access to the underlying functionality. Several procedurally oriented systems [Cutler et al., 2002, May et al., 1996, Reeves et al., 1990, Upstill, 1993] abstract the inner workings of the procedural technique by providing a high-level interface to the underlying facilities. All the modeler or shader writer need know is what functionality is available and how to apply it. A basic knowledge of how the facilities work, what they do and what they return is preferable.

Another important aspect of the support environment is the provision of data structures or representations and built-in functionality. The procedural techniques that are used to create models need to have some way of creating these models and storing them. Primitive geometric types are used to create more complex models, which in turn need to be stored.

The provision of built-in functionality provides the user with a set of basic operations that are

commonly used. This aids in the model creation, shading, texturing or scripting process as the user does not have to provide for these functions and operations for themselves. Several of the procedural approaches [Cutler et al., 2002, Upstill, 1993] allow a user to write support functions for use in shading or modeling.

Support environments provide different types of variables, namely predefined or provided variables and local variables [May et al., 1996, Reeves et al., 1990, Upstill, 1993]. The predefined variables are provided by the support environment and need to be set by the modeler. These variables are used to hold information about the current model or point which is used during the rendering process. Local variables have the same scoping requirements as those found in programming languages. The Renderman shading language local variables require a type and a storage class [Apodaca and Gritz, 2000]. The storage class determines if the variable is varying or uniform over the surface of the object. This distinction provides optimisation hints to the compiler, as a variable labeled as uniform need only be read in and evaluated once.

A very important aspect of the support environment is the use and handling of parameters. Parameters promote re-usability and variation. Many unique versions of a model can be created by changing parameter values. The behaviour of shaders can be varied by adjusting parameter values. Each of the procedural approaches discussed supports the use of parameters [Apodaca and Gritz, 2000, Cutler et al., 2002, May et al., 1996, Reeves et al., 1990]. The Renderman shading language is the most stringent in terms of parameter requirements. The shading language enforces the use of default values for each parameter in a shader's parameter list. This provides a failsafe mechanism in an instance when a parameter value is not set. The parameters and their values may be assigned in any order during a call to a shader. This makes it easier to assign values to parameters, as the value is assigned directly to the parameter by using the parameter name.

Parameterisation also allows parts of a scene to be altered, by adjusting parameter values, without affecting the rest of the scene [Marshall et al., 1980].

Parameter values that are not specified by the modeler are inferred from the provided information according to the relationship between the parameters [Marshall et al., 1980, Reeves and Blau, 1985]. In some cases assigning values to each of the parameters in a parameter list is unnecessary. This, however, depends on how the procedure has been written. It is possible to make the parameters of a procedure dependent on each other.

Support environments dictate the layout and requirements of modeling functions and shaders [Cutler et al., 2002, Upstill, 1993]. This provides a standardisation mechanism that allows interoperability with other functions or shaders.

A specific example of how this is enforced is the creation of shaders in the Renderman shading language. The Renderman shading language provides six distinct types of shader [Upstill, 1993]. The shader writer selects the type of shader to create according to the shading requirements. The different types of shader are identified through the use of keywords, in conjunction with the inputs required, and the output generated. By specifying the parameters required for each shader, the Renderman shading language provides an additional consistency check. Parameters need to conform to the basic requirements of the shader, if they do not, the environment is able to catch this error.

Another aspect that is catered for by the support environment is the evaluation of a model or shader. There are two forms of evaluation, namely level of detail and lazy evaluation. Both of these rely on the current viewing parameters of the camera within a scene. Several procedural approaches make use of bounding volumes [Hart, 1994, Upstill, 1993] to determine the required level of detail and what needs to be evaluated.

Level of detail refers to the geometric complexity of a model. Procedural modeling techniques are able to adjust the level of detail of a model based on the current viewing parameters [Apodaca and Gritz, 2000, Marshall et al., 1980, Reeves, 1983]. The amount of detail required differs according to the distance between the camera and model.

Lazy evaluation [Hart, 1994] is a method of evaluating procedures as they are needed. The scene is assessed to determine which geometry is visible. Only those procedures that contain the required geometry are processed. The scene needs to be re-appraised each time the viewing parameters change.

The sections that follow discuss the benefits associated with procedural approaches as a whole. These aspects are not necessarily related to the support environment.

### **1.3.1.2 Re-usability and Model Complexity**

An advantage of a procedural strategy is the re-usability of procedures. Procedures can be reused, in different scenarios and with different parameter values.

The equivalent manual modeling strategy is not as versatile. Models created in this manner can be reused, at a cost. Most models are created with a specific purpose in mind. Should they be needed outside of this scope, the models need to be adapted, which means that extra time and effort needs to be expended on this process.

The use of procedural modeling techniques allows for a greater degree of scene complexity [Marshall et al., 1980, Parish and Muller, 2001, Sims, 1991] as the modeling process is not performed manually. Modeling operations can be applied repeatedly through the use of procedures



and looping constructs.

Manual modeling techniques tend to be labour intensive. Procedural modeling generates models automatically using parameter values. It is also much easier to introduce variation in a procedural modeling environment by varying the parameter values.

The complexity of a model can be increased by adding extra detail to it. Repetitive geometry is one manner of introducing complexity into a model

#### **1.3.1.3 Database Amplification**

Procedural techniques promote database amplification. This is the process of generating a large amount of information from a small input set of data [Apodaca and Gritz, 2000]. The specifications for a model or shader are small in size, as they are script files that describe the process required, whereas the resulting model or shaded object contains a large amount of detail and is several orders of magnitude larger than the script used to generate it in the first place. For example, shaders generate surface detail of arbitrary complexity for which the corresponding texture map would be sizable.

#### **1.3.1.4 Base-shape Independence**

Base-shape independence is a desirable attribute to have, as it provides a procedural approach with extra flexibility. A procedure that can be re-used on an arbitrary shape or object makes a procedural strategy more versatile [Morkel and Bangay, 2006]. This concept holds true for shading, texture synthesis and modeling. The effect of the procedure must be predictable irrespective of what the starting condition is.

#### **1.3.1.5 Caching**

Caching is the process of storing information that has been evaluated or loaded for possible re-use. If a cache is available, and in use, it is first checked to see if a model is cached before re-evaluating or reloading the information or object. If the information or object is not in the cache, then it is evaluated or loaded and then stored in the cache. If, however, the item does exist in the cache, it is returned. This results in a memory overhead, as a cache structure needs to be maintained. Should an item exist in the cache, there is no need to evaluate the procedure or process to get the item. If it is not in the cache, only a small amount of time is used to search through the cache which does not result in a significantly bigger time overhead.

Caching can be used to re-instance geometry from procedures that have already been evaluated [Deussen et al., 1998, Ebert et al., 2002]. The re-use of existing geometry means that the cost of evaluating procedures is reduced, as a procedure need only be evaluated once. The down-side is an increase in memory usage, as the cache needs to be stored and updated during the modeling process.

#### **1.3.1.6 Animation**

Procedural techniques support the animation of dynamic objects or scenes. As an example, if we are draping a procedurally generated cloth model over a table, a function that takes in the current value of a time variable is responsible for returning the shape of the cloth. To animate the system, the time step is increased over time to simulate time passing. With each change to the time variable, the current shape of the cloth is returned to give the impression of falling cloth. Objects can also be controlled procedurally [Koga et al., 2004, Perlin and Goldberg, 1996].

#### **1.3.1.7 Random Numbers**

Many procedural approaches use randomness to either introduce variation into a model or depict chaos or random behaviour in movement or animation. The use of unpredictable random numbers is undesirable as results cannot easily be replicated. By using reproducible random numbers an identical model may be generated by providing the seed value originally used. The use of seeded pseudo-random numbers overcomes this drawback. Seed values provide an initial condition for the generation of random numbers.

In light of this, modeling techniques [Chen et al., 2002, Reeves, 1983] that rely on the use of pseudo-random numbers store the seed values used to generate the random numbers. If the process is repeated with the stored seed values, the initial model and conditions are recreated.

Perlin noise [Perlin, 1985] is a form of pseudo-randomness that is predictable [Apodaca and Gritz, 2000]. Noise is repeatable, but appears to be random at given levels of resolution. Shaders often use noise to give the appearance of a desired material or pattern. Noise can also be used to create surfaces with any predictable random features, at different levels of resolution [Velho et al., 2001]. A similar use is the creation of fractal terrain that relies on noise to perturb subdivided surfaces to give a jagged appearance [Musgrave et al., 1989].

## **1.4 The Problem Statement Revisited**

Section 1.1 introduces the problem statement of this research. This is an extension to the initial problem statement, and discusses the objectives of this research.

The primary objective of this research is the creation of a procedural modeling environment. The procedural modeling environment supports several of the procedural modeling benefits described in Section 1.3.1. The benefits that are supported are database amplification, procedure re-usability, increased model complexity and base-shape independence.

The two components making up the procedural modeling environment are a support environment and a set of non-interactive modeling tools.

### **1.4.1 The Support Environment**

The support environment we have created is in the form of a procedural modeling language that provides a number of facilities. Similar languages are used to facilitate modeling in existing procedural modeling paradigms (see Section 2.2.2). The procedural modeling languages provide a mechanism for accessing the underlying functionality of the procedural modeling environment while abstracting the implementation details of these facilities from the modeler. The modeler does not need to know how the tools or data structures are implemented, they only need know what tools, representations and facilities are available. The use of a high-level language makes the modeling paradigm implementation independent [Cutler et al., 2002], thus enabling different representations and tools to be used and added to the model development environment.

The use of a procedural language provides a standardisation facility as procedures are forced to conform to the layout and specification dictated by the language. This facilitates re-usability of procedures and interoperability of procedures with other procedures. The languages facilities that have been included are parameterisation, data structures, control and looping constructs, predefined variables, caching and psuedo-random numbers.

### **1.4.2 The Non-Interactive Modeling Tools**

The other component of our procedural modeling environment is our set of non-interactive modeling tools. These tools are accessed through the procedural modeling language.

Each procedural modeling tool that we have created either resembles a tool found in 3D modeling packages or it is a generalisation of a modeling technique into a tool. The tools based on existing 3D modeling tools are selection, extrusion, subdivision and stitching. The tool based

on a modeling technique is the curve shaping tool. An advantage of basing our tools on existing tools or techniques is that our procedural tools have an aspect of familiarity, as the function of the tool is predictable.

### 1.4.3 Research Objectives

The objectives of this research are:

1. Perform a series of investigations into the fields of work related to this research. Focusing on these areas:
  - (a) Identify the modeling strategies employed to create models in 3D modeling packages. Determine if any of these modeling strategies can be adapted for use in a procedural modeling environment.
  - (b) Review existing procedural modeling techniques and identify the modeling strategies employed by each. Identify the application areas of each procedural modeling technique in terms of the models that can be generated by this strategy.
  - (c) Review existing procedural modeling languages and determine what features and facilities are provided by these languages. Identify the similarities between each of the languages.
  - (d) Identify the modeling tools used in 3D modeling strategies. Determine if any of these tools is suitable for adaptation to a procedural modeling environment.
  - (e) Identify the modeling tools used in procedural modeling techniques. Determine if there is any overlap between the 3D modeling tools and the procedural modeling tools.
  - (f) Identify the model representations found in 3D modeling packages and existing procedural modeling techniques. Determine which model representations are common to both and identify which representation is most frequently found.
2. Determine whether a procedural modeling language can be created that encompasses the requirements and benefits of procedural modeling.
3. Create a requirements specification for a procedural modeling language that is based on the features and facilities identified in existing procedural modeling languages and the procedural modeling benefits listed in 1.3.1. Determine if all of the requirements can be met,

and if not, identify why it is not possible to meet the requirement. Design the procedural modeling language based on the requirements identified. Determine what design strategies can be used to provide the facility or feature embodied by the requirement. Implement the procedural modeling and discuss the strategy used to do so.

4. Determine whether it is possible to create procedural modeling tools based on existing 3D modeling tools or techniques. Determine what design strategies are required for this purpose. Discuss the implementation choices made. Determine whether what type of model representation is best for our purposes and identify in what way the representation meets our needs.
5. Determine the uses and limitations of each procedural modeling tool. Demonstrate what modeling strategies can be used to create models in the procedural modeling environment.
6. Determine to what extent the procedural modeling language supports the procedural modeling benefits of Section 1.3.1. Determine which procedural modeling benefits are supported by the procedural modeling language benefits by performing six experiments:
  - (a) Determine whether parameterisation is capable of producing a number of distinct models and whether modeling procedures are robust under a range of parameter values.
  - (b) Determine whether modeling procedures can be used on different models and in different contexts. Determine whether or not components of modeling procedures are also re-usable.
  - (c) Determine whether or not it is possible for modeling procedures to be base-shape independent.
  - (d) Determine whether model complexity can be increased by repeatedly applying modeling operations to create the same kind of geometry.
  - (e) Determine whether the procedural modeling language support for pseudo-random numbers works.
  - (f) Determine whether there are situations in which caching is beneficial. Determine whether caching reduces the amount of time required to create a model.

The procedural modeling paradigm is a move away from conventional modeling strategies that involve the use of a 3D modeling package. As such, an inexperienced modeler is not able to use

the system without first learning how it works. Our procedural modeling environment does not support immediate visual feedback.

The purpose of this research is to determine the feasibility of using a procedural modeling paradigm. Each of the models presented is a proof-of-concept and is used to illustrate the functionality of the modeling environment we have created.

## 1.5 Terminology

- **Modeler** - in the context of 3D modeling packages, a modeler is the person that creates models manually by using the representations and tools found in the packages. In a procedural modeling context, a modeler is the person that writes the modeling procedure or script.
- **Modeling package** - a 3D modeling environment that is used to create models through the use of tools and representations
- **Strategy** - the process or technique used to create a model.
- **Tool** - a facility that is used to create, extend or deform an object to create a model.
- **Constructivist** - a modeling strategy that combines components of existing models to create new ones.
- **Extrusionist** - a modeling strategy that creates an entire model from a single starting primitive through the use of modeling tools.
- **Non-interactive** - requires no modeler intervention

## 1.6 Thesis Structure

Chapter 2 discusses the existing procedural modeling strategies and their applications. Three categories of model have been identified to group the applications of the various procedural modeling techniques. These categories are organic, man-made and phenomena which are used again in the discussion of the application of our procedural modeling environment in Section 5.1.

Another aspect of existing procedural modeling strategies that is explored, is the use of procedural modeling languages. Several of the existing techniques rely on a procedural modeling language to facilitate the modeling process. Each of these languages is discussed to determine

what facilities each provides, and what motivating factors were considered when creating the language.

A discussion of 3D modeling strategies is provided in Chapter 2. This discussion presents the various strategies used to create models in a 3D modeling package. By documenting the manual modeling strategies we have a comparison to the procedural modeling strategies discussed. We are also able to identify 3D modeling strategies that can be adapted for use in a procedural modeling environment.

A discussion of the various modeling tools and model representations found in 3D modeling packages and procedural modeling environments is presented in Chapter 2. The discussion of tools found in 3D modeling packages is important, as we have based our procedural modeling tools on the tools found and strategies used in 3D modeling packages. The model representations found in 3D modeling packages and procedural modeling environments are discussed. We have created a model representation that is compatible with most 3D modeling packages and some procedural modeling environments.

Chapter 3 discusses the requirements, design and implementation of our procedural modeling languages. The requirements are based on the facilities and features identified in existing procedural modeling languages.

The design aspect discusses how each of the procedural modeling language requirements have been met. This is a high-level design strategy which is implementation independent.

The implementation aspect discusses the implementation-specific decisions we made during the implementation process.

Chapter 4 discusses each of our procedural modeling tools. Each of the modeling tools provided in 3D modeling packages is revisited and discussed. This forms part of the motivation for our choice of modeling tools. We have chosen tools that can be adapted to work in a non-interactive modeling environment. The criteria used to determine this is based on the level of interactivity required to use and manipulate the tool. Tools that are too reliant on modeler interaction are disregarded, as they are inappropriate for a non-interactive modeling environment.

The tools we have identified (selection, extrusion, subdivision, curve shaping and stitching) satisfy the procedural modeling constraints identified in Section 2.3.2.2.

Chapter 5 discusses the uses of our procedural modeling tools, and shows applications of each of them. Several procedural models have been created, and are discussed in Chapter 5. The algorithms used to create each of the models are presented and discussed.

Chapter 6 presents the experiments of Objective 6. Each experiment, with its design and implementation strategy is presented and discussed. The result of each experiment is presented

as a discussion with accompanying illustrations of the experiment.

Chapter 7 discusses the conclusions reached during the course of this research. Aspects discussed include the challenges faced during the research process and contributions made as a result of the research performed.



# Chapter 2

## Related Work

We are creating a non-interactive modeling environment that supports the procedural creation of 3D objects. To do this, we have created a support environment in the form of a procedural modeling language and a set of non-interactive modeling tools to facilitate the modeling process.

The next three chapters discuss the procedural modeling language, the procedural modeling tools and applications of these procedural modeling facilities. This chapter discusses the work related to each of these aspects.

To gain a thorough understanding of how modeling is performed procedurally and manually, we have investigated several modeling strategies of each approach.

Through this process we are able to identify traditional procedural modeling techniques and new categories which are comprised of the remaining procedural techniques. We have identified procedural strategies that procedurally model objects by: combining existing models to create new ones, adding detail to models to create new objects and deforming objects.

A review of manual modeling strategies provides an opportunity to examine each strategy and determine which ones can be adapted to a procedural modeling context. We are providing an alternative to the traditional manual modeling process, which is versatile enough to adapt manual modeling strategies to generate procedural models.

Before we look at the tools and representations available in 3D modeling packages and existing procedural modeling paradigms, we discuss the scripting and language facilities of each modeling approach. Most 3D modeling packages support both a visual feedback modeling environment and a scripting interface. These scripting facilities allow a modeler to perform limited procedural modeling with a 3D modeling package. An advantage of this is the visual feedback supported by such a modeling environment. The scripting languages support the tools and model representations used in the manual modeling approach.

The procedural modeling languages are used to facilitate the modeling process. These languages provide an interface to the underlying modeling facilities. The languages also provide tools and representations to aid the modeling process. Before we can take a look at the tools and representations, we need to determine how the language makes use of them. The language is a high-level interface, as such, the tools and representations can be changed, without changing the core functionality of the procedural modeling paradigm.

Each of the manual modeling strategies uses a set of tools to perform modeling processes. 3D modeling packages provide a variety of tools that can be used to model an object. We have composed a list of the tools available in 3D modeling packages. The list of tools is used to identify tools that can be adapted to a non-interactive modeling environment. Not all of these tools are adaptable to a procedural modeling environment as they are reliant on interaction from the modeler.

A comparison between the modeling tools found in procedural modeling and those found in 3D modeling packages is performed to determine where an overlap of functionality occurs.

Similarly, a comparison of the model representations available in procedural modeling and 3D modeling packages is discussed. Common model representations allow the models generated using one modeling strategy to be displayed and used in the other.

## **2.1 Modeling**

There are two forms of 3D modeling that is of interest to us, these are manual modeling with a 3D modeling package and procedural modeling. Each of these modeling paradigms use different modeling strategies to create models. We are interested in the techniques that are used by both manual modeling and procedural modeling.

The manual modeling strategies provide insight into how models are created by hand. These strategies can be adapted to work in a procedural modeling environment.

The procedural modeling strategies that we discuss, identify approaches used to create procedural modelings in existing non-interactive modeling environments.

By combining the strategies discussed in each of these sections, we are able to create new procedural modeling strategies that are based on familiar techniques.

### **2.1.1 Manual 3D Modeling Strategies**

There are several different approaches to manually modeling an object. These vary depending on the context, and the object that is being modeled. The most common modeling strategies are

discussed in the sections that follow.

This topic is of great importance, as we are identifying possible modeling strategies that can be adapted for use in a non-interactive modeling environment.

### **2.1.1.1 Box Modeling**

The box modeling strategy uses a cube or plane as the starting primitive for a model. This primitive shape is extended by adding new vertices, edges and faces through the use of extrusion, subdivision and the knife tool. This process is continued until the extended primitive matches is a coarse approximation of the object being modeled.

Once the necessary detail, in terms of vertices, edges and faces, has been added, the model is shaped to match the object it represents. Shaping is done by displacing vertices, edges or faces to match a perceived shape [Ratner, 2003] or according to template images [Saastamoinen, 1999].

This modeling technique has successfully been used to model humans [Ratner, 2003] and plants [Fleming, 2000].

### **2.1.1.2 Flat Mesh Modeling**

This method is very effective in the creation of highly detailed models [Fleming and Schrand, 2001]. The process starts by creating a flat polygonal mesh that is roughly the same shape as the object that is being modeled. Detail is added to the mesh to facilitate the later creation of object attributes, for example if a hand is being modeled, extra detail is for the joints of the fingers and nails. Once the basic shape and detail are done, extrusion is used to add depth to the model.

### **2.1.1.3 Patch Modeling and Lofting**

This modeling technique uses spline surface patches (such as NURBS patches) to create models. The surface of the object is modeled as a series of patches which are shaped and stitched together to form a complete object model. This strategy can be used to model humans [Ratner, 2003], creatures [Fleming and Schrand, 2001] and other objects such as knives, spoons and spatulas [Ratner, 2003].

Lofting is similar to patch modeling [Fleming and Schrand, 2001], the surface of the object is approximated with a set of splines that have an equivalent number of control points. The 3D modeling package is then responsible for *skinning* the object between the splines.

#### 2.1.1.4 Curve Modeling

Curve modeling builds on the box modeling, by providing an alternative means of shaping an object. The major curves of an object are traced onto a side and front viewpoint of a template image. The object is shaped by displacing the vertices of the object along these curves.

The steps used to create a model are:

1. Using the box modeling strategy, create a coarse approximation of the object that is being modeled. This consists of the process:

```
while image not covered
    select interior faces
    extrude to increase coverage
```

2. Displace the vertices on the object boundary to line up with the template.
3. Additional vertices may be added to improve the fit of the model, or to smooth the geometry.
4. Mirror the geometry to create a whole model.

This strategy is used to model human figures [Saastamoinen, 1999] and insects [Fleming, 2000].

Grossman et al. [2002] use a 3D tape drawing metaphor to create curve models of cars. This system is based on a 2D technique that models curves by pasting photographic tape on a drawing surface. The curves in the 3D drawing metaphor are also 2D, but are projected into 3D through the use of a depth curve. This depth curve may be created or selected from one of the existing curves. This is a feasible approach, as the curves used to model cars are created in relation to each other. The final result of the modeling process is a 3D outline of a car.

#### 2.1.1.5 Deformation Modeling

The *MakeHuman* plug-in for Blender morphs previously modeled human characters to create new models [MakeHuman Team, 2005]. The human models are based on accurate anatomical references and medical writings. The plug-in caters for the addition of parameters to create new characters. These parameters include control over body mass, gender and type of physique of the model.

Meta objects are implicit surfaces, which have logical operations, such as addition, intersection and subtraction, performed on them [Roosendaal et al., 2004].

Meta objects are suited to modeling the overall shape of an object, not explicit detail of the object, as they are shaped through the use of logical operations. Organic-looking objects can be modeled with ease through the use of meta objects, as they are rounded [Fleming and Schrand, 2001].

Maya has a curve shaping tool known as *wires* [Singh and Fiume, 1998]. This tool is used by a modeler to shape an object by adding curves to control the deformation of an object. These curves are used to approximate the object, and by moving the control points of the curves, the underlying object is deformed. This deformation techniques is independent of the level of detail provided in the model.

### 2.1.2 Procedural Modeling

This section discusses the various forms of procedural modeling. Each type of procedural modeling technique is suited to a specific modeling domain [Cutler et al., 2002, Green and Sun, 1988].

The procedural modeling techniques have been grouped according to the manner in which objects are modeled. Our procedural modeling categories are L-systems, fractals and fractional Brownian motion, particle systems, compositing geometry, detail modeling and deformation modeling. L-systems, fractals and particle systems are well-known procedural modeling techniques. The remaining categories, compositing geometry, detail modeling and deformation modeling, have been created according to the types of modeling the procedural techniques in these categories perform.

The compositing geometry category discusses procedural techniques that use a combination of existing models to create new models, and to add detail to scenes.

The detail modeling category discusses procedural techniques that detail, such as hair or spikes, to models procedurally.

The deformation modeling category discusses procedural techniques that use deformations to create and shape models.

Section 2.1.3 discusses the application of procedural modeling techniques to various types of models.

#### 2.1.2.1 Lindenmayer Systems

Lindenmayer systems (L-systems) are parallel rewriting grammars [Prusinkiewicz and Lindenmayer, 1990]. Rewriting is the process of creating a complex string by replacing components of



Figure 2.1: An L-system tree. Image courtesy of Kevin Glass

the initial starting axiom. This process is performed in parallel, so all of the replaceable characters in a string are replaced in a single pass. Rewriting rules and productions dictate what, and under which conditions, replacements occur [Prusinkiewicz and Lindenmayer, 1990]. This process is repeated until it is no longer possible to perform rewriting, or when a specified recursive depth has been reached.

Once the the rewriting process has been completed, the characters in the string are interpreted as modeling commands or primitive objects.

L-systems are very well suited to modeling branching structures and are used to model plants [Deussen et al., 1998, Prusinkiewicz and Lindenmayer, 1990] and feathers [Chen et al., 2002]. An example of a tree generated by an L-system is shown in Figure 2.1. The grammar for this L-system tree was adapted from [Prusinkiewicz and Lindenmayer, 1990].

### 2.1.2.2 Fractals

Fractals are complex geometric objects that are made up of pieces similar to the entire model in some form or other, over a range of scales, which causes fractals to look the same, irrespective of the scale [Ebert et al., 2003, Feder, 1989]. This is a property known as self-similarity. Fractals are statistically self-similar [Ebert et al., 2003, Fournier et al., 1982], which means that while the smaller components making up the larger geometric object are similar at different scales, they are not identical.

Fractals are band-limited, as there is an upper and lower scale limit to self-similarity [Ebert et al., 2003]. Viewing the fractal at a scale above or below these limits causes the self-similarity

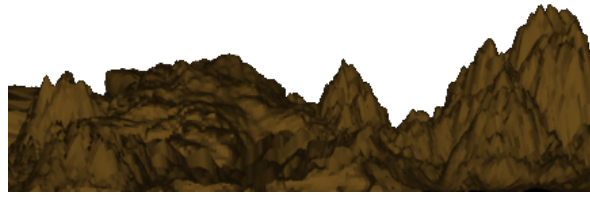


Figure 2.2: A fractional Brownian motion displacement shader

property of the fractal to be ineffectual.

Fractals are ideal for creating visual complexity in models which require detail at different resolution levels [Ebert et al., 2003]. Naturally occurring examples of fractals include clouds, water and mountainous terrain, each of which is self-similar over a range of scales.

### 2.1.2.3 Fractional Brownian Motion

Fractional Brownian motion (fBm) can be created by either adding or multiplying frequencies to obtain a noise value [Apodaca and Gritz, 2000, Ebert et al., 2003]. The dimension, or jaggedness, of such a fractal model differs according to its location. Fractional Brownian motion is also statistically self-similar [Ebert et al., 2003, Fournier et al., 1982].

These patterns are visible in nature, in the form of mountainous terrain. Terrain contains a large amount of detail at varying frequency levels, for example, it is possible to have large open plains, with undulating hills and jagged mountain ranges [Ebert et al., 2003]. Each of these terrain components is represented by detail at a different frequency level. When these frequencies are combined, a terrain-like model is the result.

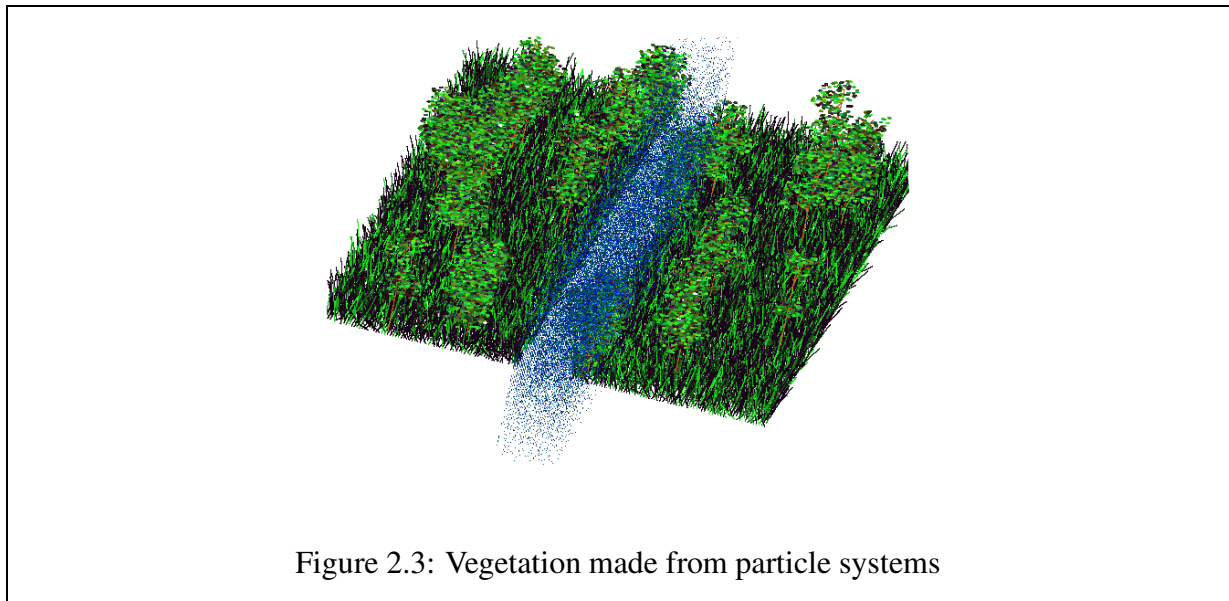
An example of fractional Brownian motion is illustrated by the displacement shader in Figure 2.2.

### 2.1.2.4 Particle Systems

A particle system approximates the volume of an object through the use of clouds of primitives (particles) [Reeves, 1983]. The particle system is controlled procedurally.

Particle systems are best used for models that are dynamic, have no clear edges, and that are unpredictable. Examples of objects include fire, water, clouds and vegetation.

Particle systems have varying levels of detail which are adapted to the current viewing parameters.



A combination of particle systems have been used to generate the vegetation scene in Figure 2.3. The waterfall and river are dynamic particle systems that change over time. The grass blades are particles that have been drawn as a series of line segments. The blades depict the limited motion of the particle over time. The trees are also made from particles. A strategy similar to the grass blades has been used for the tree trunks, and the leaves are depicted as particles.

### 2.1.2.5 Compositing Geometry

Procedural modeling is versatile enough to allow the combination of existing geometric elements into more complex models.

Marshall et al. [1980] use existing geometric elements to add detail and complexity to terrain models. The procedures encapsulate the calls to the existing geometry, while allowing control of the model creation process through the use of parameters. An example of this is the generation of a tree model. The leaves, trunk and branches are the basic elements that are combined to create a tree.

Cutler et al. [2002] combine layers of material to create volumetric models. A surface model of an object is used as the starting primitive. Layers of materials are created from the initial surface.



### 2.1.2.6 Detail Modeling

Perlin and Hoffert [1989] make use of noise to generate detail on objects. The system is given a base shape which is modeled as arrangements of density (a hard region which is definite, and a soft region which is vague). Noise is used in the system, in conjunction with shaping operators, to introduce controllable randomness into the signal used to generate detail. Noise is implemented in the system as a sum of pseudo-random spline knots.

Velho et al. [2001] make use of procedural techniques to synthesize shapes on subdivided surfaces. Shape synthesis is performed procedurally, and can be used to synthesize shapes on any base shape. To synthesize shapes, details are added to each level of the subdivided shape (multi-resolution surfaces). Details can be specified at a limited number of levels. Subdivided surfaces allow shapes to be displayed and edited at different levels of the surface. By using this attribute of subdivided surfaces, Velho et al. [2001] obtain varying levels of detail.

### 2.1.2.7 Deformation Modeling

Procedural modeling is able to create detail by deforming an object. Object deformation is used as a method of sculpting an object into the shape required. Szeliski and Tonnesen [1992] allow a user to deform a particle surface by pushing objects (knives and spheres) through the surface. The surfaces adapt by stretching or adding in new particles to ensure a watertight surface.

Cutler et al. [2002] create volumetric models by creating layers of material. A starting object is loaded, and layers are added to the interior and exterior to create layered objects. An example of this is candy, which can be made of different types of sweets.

The layers used to create models are combined through the use of logical operations such as addition, subtraction and intersection.

Models may be deformed through the use of simulation tools, for example a hammer and chisel tool or an erosion tool. These tools are applied to the model and are restricted to work with a specific type of material.

To a large extent, the work by Velho et al. [2001] (see Section 2.1.2.6) can also be considered deformation modeling. Although one is adding detail to an object, it is possible to deform the details created, or to deform the object itself to create the detail.

Lewis and Jones [2004] use a series of deformations to create models. The modeling process begins by selecting a basic shape. This shape is deformed and shaped until the desired object is achieved. Their system is versatile enough to allow a user to go back to any step within the deformation process, to redo it, or to view it.

The HyperFun system, created by Cartwright et al. [2005], creates models through the application of logical functions, such as addition and subtraction, to implicit modeling components (eg cubes, spheres and cylinders). Deformation effects are achieved by applying the set operators provided.

### **2.1.3 Applications of Procedural Modeling**

This section discusses the applications of the procedural modeling strategies discussed in Section 2.1.2.

We have grouped the applications of procedural modeling according to the types of objects that are modeled. To make this grouping as general as possible, we have identified three main categories of objects which are modeled by existing procedural techniques.

These categories are used again later to describe the types of models we are able to construct using our procedural modeling environment. These models are discussed in Chapter 5. By using common categories, we are able to directly contrast the applications of existing procedural techniques with our own procedural modeling strategy.

The organic object category is discussed first, with a discussion on the man-made object category preceding the naturally occurring phenomena category.

#### **2.1.3.1 Organic Objects**

Categories are organic objects, man-made objects and naturally occurring phenomena.

The organic category describes objects that occur naturally, but has been extended to include objects that are smooth and lacking jagged edges. Objects such as plants, animals, terrain, insects and humans are classified as natural models according to our definition of organic, while cloth is an example of a unnatural object that is considered to be an organic model.

#### **Plant Modeling**

L-systems are commonly used to model vegetation and its growth [Deussen et al., 1998, Prusinkiewicz and Lindenmayer, 1990, Prusinkiewicz, 2000]. Information pertaining to light and water is passed through in the form of input files, or are specified by hand.

Reeves and Blau [1985] make use of particle systems to generate vegetation.

### Feather Modeling

Chen et al. [2002] make use of L-Systems to generate feathers. The texture, barbs and curves of the feather to be generated are specified manually. The feather is created through the use of a parametric L-system that uses external forces to add realism to the resulting feather.

Once the feather has been generated, a number of key feathers are placed on the model to facilitate the interpolation of feathers over the object to cover it.

L-systems are used because of the inherent branching nature of feathers [Chen et al., 2002]. Both the random seed and the parameter values are stored for feather re-creation.

### Terrain Modeling

Fournier et al. [1982] make use of an approximation to fractional Brownian motion to create terrain. This approximation adds noise functions together, weighting each of the noise frequencies according to how high or low the frequency is. This process is used to generate a scalar displacement value. This scalar value is added to the height (y axis) component of the surface model to generate terrain. The approximation to fractional Brownian motion makes the scalar values dependent on previous values.

Musgrave et al. [1989] make use of noise and erosion to create eroded fractal terrains. Each point in the terrain is determined, independently of its neighbours, through the use of noise synthesis. To create the effect of eroded terrain, the system trickles water on each vertex in a fractal height field. The water is permitted to run off the landscape, eroding the terrain as it goes. The water also serves another purpose, the deposition of eroded material on the terrain.

Marshall et al. [1980] make use of procedural modeling and predefined sets of data to create complex scenes (see Section 2.1.2.5). The output of a procedure can form the input for another procedure. The cost for this procedural modeling approach is a high computational overhead (as the procedures are evaluated as required). This is because the scene is re-evaluated with every change made.

The savings associated with this procedural approach are achieved in terms of storage space. Terrain descriptions no longer have to be very large, which results in storage savings [Marshall et al., 1980].

The trade-off this approach makes is storage savings for computational overhead.

Visually satisfying scenes that have a degree of noisiness are generated by using this approach. This noisiness is achieved through the use of random numbers and parameter values.

The parameters used by Marshall et al. [1980] are dependent on each other. As an example, the procedure to produce trees requires that the specification of two out of three possible param-

eters (the available parameters are `leaf_density`, `ave_branch_size` and `num_leaves`). The third parameter is inferred from the supplied two [Marshall et al., 1980].

Hultquist et al. [2006] use procedural techniques such particle systems and L-systems to generate terrain. The parameter values are set through the use of adjectives that describe the desired scene. Each adjective has a scalar associated with it, this scalar is set and then used to generate the vegetation and terrain of a scene.

### **Human Modeling**

Animation Language [Scheepers et al., 1997] is a procedural modeling language (see Section 2.2.2) that is used to create human musculature models. The main objective of the modeling is to create an accurate representation of the muscles that lie beneath the skin, and further to that, to allow realistic movement (i.e. bulging and stretching) of these muscles.

Non-interactive modeling tools have been used to model humans [Morkel and Bangay, 2006]. The tools used are extrusion, a curve shaping tool and subdivision. Extrusion is used to extend a primitive shape to create a coarse approximation of the human model. The curve shaping tool is used to fit the model to a set of curves. Subdivision is used to smooth the model and add extra detail, vertices, edges and faces, for shaping.

### **Cloth Modeling**

Particle systems can be used to model cloth [Choi and Ko, 2002, Oshita and Makinouchi, 2001]. The particles represent the vertices of the cloth model. Forces, in the form of springs, are used to make the particles influence each other. The forces ensure that the cloth retains its shape.

### **Surface Modeling**

Similar to cloth modeling, [Szeliski and Tonnesen, 1992] make use of particles to model surfaces. The particles exert forces on each other, which ensures that the surface will not come apart. The surface is malleable, as the user can manipulate it by using a knife tool, or by pushing an object through it. The surface will adapt by adding in extra particles if the requirements for doing so are met.

Szeliski and Tonnesen [1992] make use of oriented particles to model surfaces. Oriented particles use external forces and constraints to force the particles to form a surface. There are internal forces, between particles, which ensures that the surface does come apart.

The surface formed by oriented particles is malleable, and can be deformed. During a deformation, an oriented particle surface is able to add extra particles to prevent the surface from

tearing apart. To add particles, two conditions have to be met. The first condition is that there is sufficient space between particles to add new ones, and the second condition is that the number of neighbours in the region of interest is within a valid range (i.e. falls between the minimum and maximum number of neighbours). If these conditions are met, new particles are added to the surface model. If these conditions are not met, then no new particles are added.

### **Candy Modeling**

Cutler et al. [2002] create models of candy by combining layers of different materials to create volumetric models. Each layer of the model represents a different layer of the sweet being modeled.

#### **2.1.3.2 Man-made Objects**

The man-made category describes objects that made by humans or machines. Objects in this category tend to have definite edges, and are angular in shape. These objects do not occur naturally, and include objects such as cities, furniture and cars.

### **City Modeling**

Parish and Muller [2001] use two L-systems to generate city models. The first is an extended L-system which is used to generate the road network of the city. The second is a parametric L-system which is used to generate the buildings of the city.

The layout of the road network is based on the type of input map provided to the extended L-system. The type of maps that can be used include elevation and population density maps. Through the use of rules and constraints, the road network is made to conform to the input map.

Once the road network has been generated, the lots of land between roads is subdivided to determine the placement of buildings. The parametric L-system is then responsible for creating the buildings, through the use of transformations and extrusion.

Sun et al. [2002] use a similar strategy by making use of input maps to create road network templates. These templates describe Voronoi, raster and radial road network patterns.

A combination of procedural techniques is used by *Glass et al. [2006]* to model informal settlements. The strategy adopted uses a combination of procedural modeling techniques to determine the placement of dwellings. Voronoi diagrams, L-systems and subdivision are used to create flat meshes. Noise is added to the subdivided surfaces to add randomness to the placement of dwellings. Dwellings are placed at each of the vertices of the resulting mesh.

Greuter et al. [2003] generate procedural cities based on the current viewpoint. The ground plane of the city is divided into a 2D grid, with each block demarcating a building. The buildings use pseudo-random numbers whose seeds are calculated according to the location of the block in the grid. The building is extruded upwards from a floor plan.

The amount of buildings generated in a scene is decided by the amount of the grid visible from the current viewpoint. Only the currently visible buildings are evaluated and cached to save on re-evaluation costs.

### **Furniture Modeling**

Non-interactive modeling tools can be used to create furniture models, such as chairs, couches and tables Morkel and Bangay [2006]. The models used are extrusion, a curve shaping tool and subdivision. A starting primitive is extruded to create additional detail such as vertices, edges and faces. This detail is used during the deformation process to shape the furniture models. Subdivision is used to smooth the models.

### **Statue Modeling**

Cutler et al. [2002] create volumetric statue models. These models are made up of different layers, which are represented as different materials. An example presented by Cutler et al. [2002] is a bronze cat model that is embedded in a layer of clay material. This material is removed through the use of simulation tools and the bronze model is revealed.

#### **2.1.3.3 Phenomena**

The phenomena category describes naturally occurring events such as rain, water, clouds and fire. These models are not part of the organic category as they are fuzzy in nature [Reeves, 1983], and are not volumetric models. Their composition approximates a volumetric shape or form, as they are made up of smaller components.

### **Fire Modeling**

There are two forms of procedural modeling that can be used to model fires, these are L-systems and particle systems respectively.

Zaniewski and Bangay [2003] illustrate that an L-system extended with environmental feedback can be used to model fires in factory buildings. Reeves [1983] makes use of particle systems to model fire.

Procedural Technique	Researcher	Terrain	Plant	Cloud	Human	Surface	Candy	City	Room	Scene	File
Landscape	[Chen et al., 2007]	[Hidalgue et al., 2006]	[Deussen et al., 1998] [FractiBionic and Lindenmayer, 1999]					[Glick et al., 2006] [Roth and Muller, 2001]			[Castelnati and Wang, 2003]
Particle		[Mangano et al., 1989]									
Fire		[Foster et al., 1987]									
Particle Systems			[Reeves and Elus, 1983]	[Choi and Ko, 2007] [Gama and Makiouchi, 2001]		[Schikl and Tonnesen, 1997]					[Reeves 1983]
Compositing Geometry		[Marshall et al., 1985]					[Kutler et al., 2007]			[Kutler et al., 2007]	
Deformation Modeling					[Moriel and Wang, 2006] [Schroeder et al., 1997]				[Moriel and Wang, 2006]	[Kutler et al., 2007]	

Table 2.1: Applications of procedural modeling

Reeves makes use of random number seeds, and adjustable parameters, to create particles. The seed values of each new particle are written to a table of seed values at each frame. This allows the system to start generating particles at an arbitrary frame. The system will start generating particles several frames before the desired frame, this means that the desired frame will consist of particles that contribute to it from previous frames [Reeves, 1983].

### 2.1.3.4 Summary

Table 2.1 summarises the applications of procedural modeling, and the procedural techniques which may be used to create these models.

## 2.2 Modeling Languages

This section looks at the various forms of procedural modeling languages and the equivalent scripting languages available in modeling packages. The use of scripting languages allows a modeler to perform limited procedural modeling in an interactive modeling environment.

### 2.2.1 3D Modeling Packages - Scripting

Many 3D modeling packages provide a scripting or plug-in interface that allows some degree of procedural geometry generation [Autodesk, Inc, 1999b, Bayne et al., 2004, Brooks et al., 2001,

NewTek, 2001, Roosendaal and Selleri, 2004]. In some cases the manual modeling process can be recorded as the sequence of modeling commands, and can be reused in a script [Lewis and Jones, 2004]. Upon completion of the model, any step of the model creation process may be modified and the resulting model regenerated.

An advantage of these systems is the ability to visualise a model and select changes to be made directly on the model. They provide a visual programming facility for procedural modeling.

Scripted modeling in Maya is performed through the use of MEL (Maya Embedded Language) scripts [Brooks et al., 2001]. Manual modeling can be expressed as MEL scripts through the use of echo commands in Maya, meaning that MEL supports all of the modeling operations available. Various manual modeling tools are available through the scripting interface.

SoftImage XSI caters for the modeling and rendering aspects of scenes. A range of scripting languages can be used to create procedural scripts [Bayne et al., 2004]. Similar to Maya, the scripting languages can be used to perform modeling tasks, and are especially suited to performing similar operations repetitively.

### **2.2.2 Procedural Modeling Languages**

Procedural modeling languages are used to facilitate the procedural generation of models. They provide a high-level way of accessing the underlying facilities, representations and tools. The languages provide a means of standardising modeling processes for re-usability and inter-operability. The level of abstraction afforded by a procedural modeling language allows the modeling environment to be independent of any particular implementation. Procedural modeling languages are extensible and flexible which allows a modeler to add functionality, tools and representations to it [Reeves et al., 1990].

Each of the procedural modeling languages discussed extend existing programming languages [Cutler, 2003, Green and Sun, 1988, May et al., 1996, Reeves et al., 1990]. There are two advantages to using such a strategy. The modeling language inherits all of the functionality, data structures, control constructs and libraries available in the programming language. The other advantage is that using this strategy, the implementation time and effort required to create a procedural modeling language is greatly reduced. Another reason for using an existing programming language is that these languages are capable of solving a large variety of computer and graphics based problems [May et al., 1996]. Programming languages have the functionality to facilitate the modeling process, but do not promote experimentation [Green and Sun, 1988]. The use of a procedural modeling language facilitates the procedural modeling process, as access



to the underlying functionality is provided through the modeling language. A standard way of accessing these facilities is provided by the language.

Each procedural modeling language is discussed individually. The discussion focuses on how each of the languages caters for the various facilities required by such a modeling environment. These facilities include level of detail, re-usability, complexity, database amplification, base-shape independence and parameterisation.

### 2.2.2.1 Modeling Environment (Menv)

The Menv system provides an environment that encourages procedural modeling and animation [Reeves et al., 1990]. The Menv philosophy is that a modeler should have high-level control over both the modeling and animation of objects, the underlying facilities are controlled by the system. The Menv modeling environment provides visual feedback by employing model views similar to a 3D modeling package. The Menv user interface is windows based.

The Menv modeling environment uses Modeling Language (ML), which is extensible and modularised. The modularisation aspect is used for the modeling tools, which are modules that are loaded into the modeling environment for use. The tools available are a save workspace tool, input and output tool, display tool, camera tool, object or avar tool, calculator, 3D digitiser tool, curve tool, sweep tool, avar key-frame tool, avar spline tool, muscle-based animation tool, armature-based animation tool, playback, model-to-text tool, grid tool, patch editor, pick tool and grease pencil tool [Reeves et al., 1990].

The communication between tools and modeling processes is done through the use of a central storage facility that each process has access to. Messages are treated as events, and are passed in a ring of message tokens.

Menv supports hierarchical modeling, which is also used for animation purposes. Animation primitives are bound to the various components of an object hierarchy. This facilitates the animation of the object. The Menv system uses a two-pass process to optimise the animation process.

Reeves et al. [1990] identify three advantages of using the Menv system, which are replication, precision and parameterisation.

Replication is the process of instancing geometry to be used in a model. The language is concerned with instancing geometry and transforming it to facilitate placement of the geometry in the model. The replication of geometry promotes model complexity and database amplification.

Complexity is added to a model by adding detail that is repetitive and tedious to do by hand. By using a procedural strategy, the level of complexity can be adjusted to suit the requirements

of the model.

Database amplification is supported by replication, as the same type of geometry is used to create a model. A highly complex and detailed object can be created by using a small model specification. This is due to the re-usability of the geometry.

The precision afforded by a procedural modeling approach is far greater than that possible by manual modeling techniques. A procedural model is created algorithmically, so the language can be used to perform exact alignment calculations when combining objects.

Parameterisation controls the model creation process. By using parameters, the model can be varied, and is not reliant on hard-coded values within the program.

A variety of primitives are provided by Menv, each of which may be used to model an object. Each model is stored as a procedure and contains all of the necessary commands to create it. The application of these procedures to other starting primitives is feasible, as the only change that needs to be made is the selection of the shape.

### **2.2.2.2 Animation Language (AL)**

AL is a procedural modeling and animation language [May et al., 1996] that provides similar modeling and animation facilities as Menv. The AL system is an extension to the Scheme programming language. The models are represented as AL procedures which may be stored and re-used. AL uses Renderman compliant renderers to generate high-quality images. All types of Renderman shader are also supported by AL.

The geometric primitives provided by AL are stored as parameterised procedures which are called as needed. Each procedure has default parameter values which are used if a parameter value is not supplied in a procedure call. Hierarchical modeling is also supported by AL.

Replication, precision and parameterisation are also supported by AL and provide similar benefits. In addition to parameterisation, AL uses predefined variables to maintain the current graphics state of the scene [May et al., 1996]. The graphics state variables store cumulative transformations and a table of model attributes, and return the current state of each of these when requested to do so. The graphics state is used during the rendering process to provide details about the model to the renderer. Time is another predefined variable which is used for animation of the model.

No underlying data structures are created during the model evaluation process, instead a set of rendering commands are outputted which are used to render the final image of a model.

### 2.2.2.3 Interactive Animation

The interactive animation system is based on MML, which is a special purpose language developed for procedural modeling and animation [Green and Sun, 1988]. The MML language is an extension of the C programming language, which does not use a compiler, only a preprocessor to convert MML code to C code.

The purpose of this system is to provide a high-level notation for describing procedural models and motion. Models are described and stored as MML procedures. MML procedures have a definite structure, and must include primitives, generation, motion and rendering conditions.

The primitives section defines the primitives to be used in a model. Each primitive definition contains a primitive name and a list of properties, which includes the type of the primitive.

The generate section specifies how primitives are generated. This makes use of a rule-based model which contains parameter definitions to initialise variables, and a rule section that describes how the primitives are generated. Each rule has a condition and action component. The condition component is used to determine when an action is applied. The action component specifies the action, in C code, to be performed.

The motion section contains the motion verb declarations. Motion verbs describe the motion of an object. Motion is defined in terms of the type of primitive, and not individual primitives.

The render section describes how each of the modeling primitives is converted into a display primitive.

When a procedure is evaluated, the preprocessing stage gathers information about the generation and motion verbs. This information includes the names, types, ranges and motion verbs used in the model. This information is used to construct a user interface, which is model specific.

The interface consists of four components, which are the view, generate, motion and edit components.

- View - sets parameters and determines if frames need to be recorded for playback. This is also responsible for playback. Parameter values are adjusted through the use of potentiometers, which are generated when the interface is created. There is a potentiometer for each of the parameters used in the model declaration.
- Generate - This is concerned with the creation of primitives, which also uses potentiometers, one for each parameter and another to control the number of generations before the first animation frame.
- Motion - Controls the application of motion verbs. This is motion-based, once a verb is

selected, a bank of potentiometers appears, one for each parameter and another to specify the number of frames a motion verb is active in.

- Edit - This provides access to the motion table, which may be manually edited.

#### 2.2.2.4 Authoring Solid Models

The procedural modeling environment created by Cutler et al. [2002] uses a scripting language to create layered, volumetric models.

The modeling language extends the C++ programming language, and is grammar-based. This language is concerned with the specification of geometric and material properties of a model, which may be varied over time to alter the model.

The premise of this modeling language is that the internal structure of a model can be derived from the surface representation of the object.

The procedural framework provides a controlled, systematic way to specify the geometric and material properties of a solid model and to vary these as a function of time. The geometry and material layers are declared and combined procedurally to create layered objects. The layered objects are created through the use of logical functions, such as addition, subtraction and intersection.

Material properties are defined through the use of rendering and simulation parameters, each of which has a default value. An existing library of materials can be supplemented by modeler-defined scripts.

C++ functions can be defined to aid the modeling process. These functions must have default parameter values for optional parameters. A procedure is called with its parameters assigned to values, for example height = 3.0. If the parameters are optional, they do not need to be included in the procedure call.

Simulation tools, such as weathering, are used to deform and shape models. Each tool contains default values for each of the standard parameters catered for, and can be material-specific, i.e. the tool only functions on a specific type of material.

The tools are re-usable, as are the model definitions. The complexity of a model is determined by the amount of detail added and layers used to create the model. A small model definition can be used to create a large and complex model.

The base-shape of a model is determined by the model that is loaded into the system. The modeling techniques and tools are generic enough to work with any type of base-shape, which makes this procedural modeling paradigm base-shape independent.

### 2.2.2.5 The Renderman Shading Language

The Renderman shading language provides a limited procedural modeling facility [Upstill, 1993], which supports the creation of hierarchical models. This allows complex models to be made from the simple primitives available in Renderman.

Models are specified as procedures which define all aspects of the model. An advantage of using a procedural representation of a model is the storage savings. Database amplification is supported by this modeling approach, as the specification of a model is often many times smaller than the resulting model. Modeling procedures are invoked during the rendering process. Bounding volumes are used to determine whether or not a procedure should be evaluated.

Another aspect of procedural modeling that is supported by Renderman is the use of varying levels of detail. Bounding volumes are used to determine the amount of detail required for a model. The larger the bounding volume, the more detail is added to a model, the opposite is also true. This varying level of detail provides a trade-off between how much detail can be visually appreciated and the amount of work the renderer has to perform [Upstill, 1993].

Although Renderman supports procedural modeling, the focus of the Renderman language is shading. An aim of the Renderman shading language is the separation of the modeling environment from the rendering environment. This approach gives the renderer control over what is rendered in the final image. This is due to the renderer being responsible for determining what needs to be evaluated and shaded according to the visibility of the object in the scene.

### 2.2.2.6 Summary

A similarity between the procedural model modeling languages discussed is the representation of models as programs of the respective languages [Cutler et al., 2002, Green and Sun, 1988, May et al., 1996, Reeves et al., 1990]. Representing a model in this manner has two advantages, documentation and storage savings. A model that is represented as a program is its own documentation [Reeves et al., 1990], as each of the steps taken to create the model is recorded in the program.

Procedural modeling language tools and representations are reusable, as many of the them are stored as procedures that are evaluated when required [Cutler et al., 2002, May et al., 1996].

The geometry created during modeling is also reusable, and can be repeatedly applied to a model to create complexity [May et al., 1996, Reeves et al., 1990]. Detail that is tedious to create and apply to a model manually can be done with ease through the use of a procedural modeling language, due to the procedural nature of the system. Looping structures can be used to repeatedly apply detail to models.

The procedural modeling tools used by these languages are base-shape independent [Cutler et al., 2002] and can be applied to a variety of models. New tools can be created and added to the procedural modeling language, as the languages are modularised [Cutler et al., 2002, May et al., 1996, Reeves et al., 1990] and new components can be added to it.

Another aspect that is common in all of the procedural modeling languages is the use of parameters. In some cases, the use of default parameter values is enforced [Cutler et al., 2002, Green and Sun, 1988, May et al., 1996, Upstill, 1993].

## 2.3 Tools and Representations

This section discusses the tools and representations available in 3D modeling packages and the tools and representations used in the different procedural modeling strategies (Section 2.1.2) and the procedural modeling language (Section 2.2.2).

The first aspect discussed in this section is the relationship between procedural modeling and the level of interactivity. Which is followed by the discussions on modeling tools and model representations.

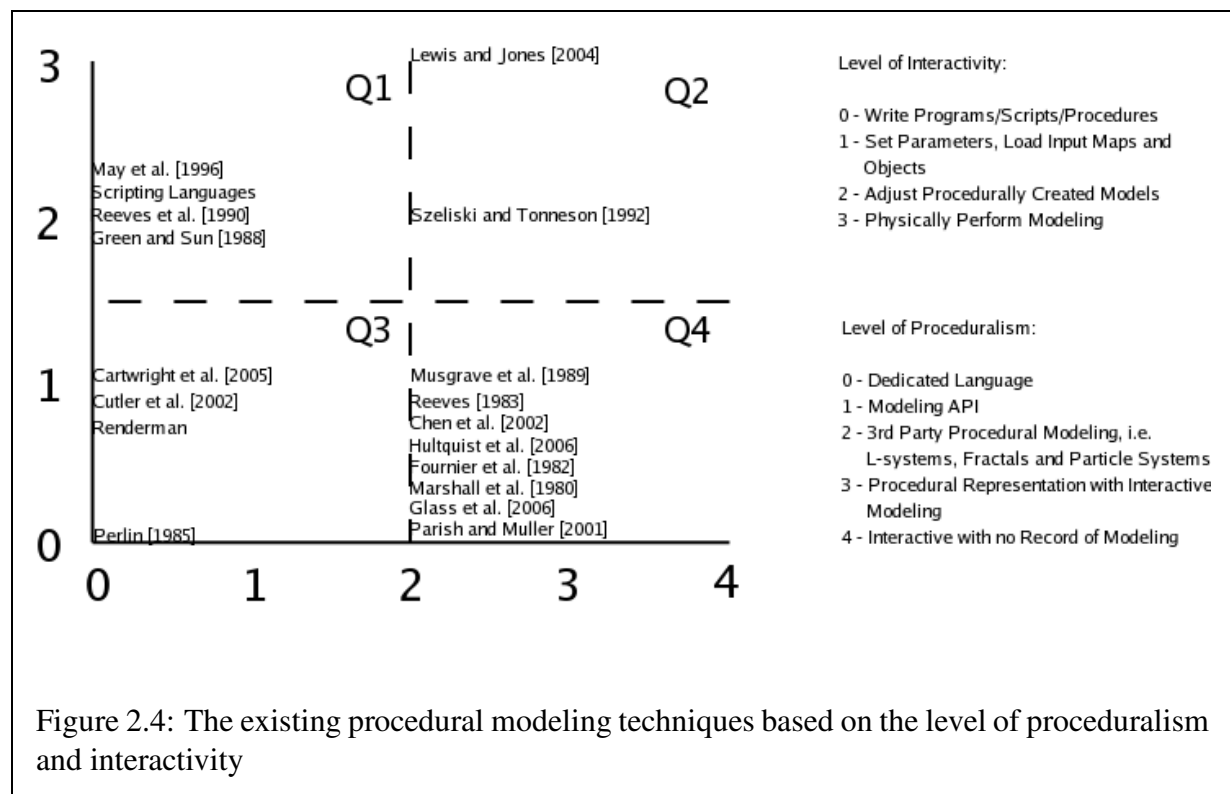
### 2.3.1 Interactivity

The goal of this section is to determine what type, if any, of relationship exists between the level of procedural modeling and the level of interactivity of the procedural modeling techniques we have reviewed.

We have devised two scales, namely level of proceduralism and level of interactivity, to categorise the procedural techniques discussed. The procedural modeling scale is as follows:

- 0 - Dedicated procedural modeling language;
- 1 - Modeling API is provided;
- 2 - Third part procedural modeling, i.e. L-systems, particle systems, fractals and compositing geometry;
- 3 - Procedural representation with interactive modeling;
- 4 - Interactive with no record of modeling.

The interactivity scale is defined as:



- 0 - Write programs, scripts or procedures;
- 1 - Set parameters, load input maps and objects;
- 2 - Adjust procedurally created models;
- 3 - Physically perform modeling.

Figure 2.4 shows one way of determining the relationships between the level of proceduralism and the level of interactivity of a procedural modeling technique. This method also allows us to identify trends in terms of these relationships between the different techniques.

The first quadrant of Figure 2.4 (denoted as *Q1*) shows that several of the procedural modeling languages and interfaces are highly procedural in nature, but provide facilities for the interactive manipulation of models. The second quadrant (*Q2*) shows that it is possible to have procedural modeling techniques that are not very procedural in nature and rely on modeler interaction. Quadrant three (*Q3*) shows the remainder of the procedural modeling language-like techniques are highly procedural in nature but provide very little support in terms of interactivity. The final quadrant (*Q4*) illustrates that the majority of procedural modeling techniques rely on the use of

third party procedural modeling strategies, including L-systems and fractals, to create models. These techniques also provide very little interaction support to a modeler.

From these findings, we conclude that procedural modeling languages tend to be highly procedural in nature and vary from very little interaction (writing programs and setting parameters) to a significant amount of interaction (adjustment of procedurally generated models).

Another conclusion is that a technique may be considered procedural even though it is highly reliant on interaction. These techniques do not occur as frequently as the little to no interaction techniques, but they exist.

The final conclusion reached is that third party procedural modeling techniques are commonly used to generate models. The majority of procedural techniques rely on these modeling strategies. These techniques require very little interaction and are well-documented.

### **2.3.2 Modeling Tools**

This section contains information relevant to the various modeling tools found in procedural modeling environments and modeling packages. A description of these tools is relevant, as decisions need to be made regarding which modeling tools are to be adapted to function in our procedural modeling environment.

#### **2.3.2.1 3D Modeling Package Tools**

There are large quantities of tools offered for use in 3D modeling packages. This discussion focuses on the most commonly used and available tools, as these are the most likely candidates for adapting to a procedural modeling environment. These tools include subdivision and subdivision surfaces, the knife tool, extrusion, lofting, constructive solid geometry, deformations and stitching.

The tools commonly used to model the coarse shape of a model are shaping, extrusion, subdivision and the knife tool. Each of these tools creates new vertices, edges and faces in the model.

In a manual modeling environment, selections are performed interactively. If the scripting interface of a 3D modeling package is used to model an object another means of performing selection needs to be employed. The general strategy employed by these packages is the labeling of each unique object in a scene, and then referencing individual faces and vertices in terms of their indices within desired object. This introduces several problems. One problem is the need to continuously update indices, as faces or vertices may be removed and new ones added



during modeling operations. This can result in the selection or removal of incorrect faces or vertices at a later stage during modeling due to errors in indexing. Another problem is the need to keep track of individual faces on an object, which is a cumbersome and unrealistic expectation. Several of the 3D modeling packages [Autodesk, Inc, 1999b, Bayne et al., 2004, Brooks et al., 2001, NewTek, 2001, Roosendaal and Selleri, 2004] provide a recording facility that stores the modeling commands used in a manual modeling scenario for re-use in a scripted modeling situation.

Extrusion is simplistic in basic function, and only requires an initial selection and a direction and magnitude in which to “pull out” this selection. Subdivision is far more complicated than extrusion, but once provided with an initial selection, the process continues happily without interaction. The *Metaform* tool found in Lightwave is a form of subdivision that uses the original shape as a bounding volume for a smooth, more organic-looking shape [NewTek, 2001]. If the original shape has a large amount of detail, it is preserved during this process. The shaping continues in the same manner as ordinary box modeling. This approach can be used to model creatures [Fleming and Schrand, 2001]. The knife tool is also relatively simple in function, but requires exact user interaction in the form of specifying where new vertices should be added to a model. In a manual modeling environment, the knife tool adds vertices at the locations specified when the modeler drags the knife tool over the model. The functionality of the extrusion tool and the knife tool are similar, but the extrusion tool is a lot easier to port to a non-interactive modeling paradigm. As such, the tools that have the most potential for use in procedural modeling are extrusion and subdivision. Implementation of the knife tool can be done at a later stage.

Shaping tools are used to deform a model to match the shape of the object that is being modeled. There are at least two ways of performing shaping, one is the manual displacement of vertices to match a given (or imagined) shape and the other is by using the various deformation and shaping tools provided in the modeling package. The first of these options is not feasible, as we are performing procedural modeling with as little modeler interaction as possible. The second shaping option is more appropriate, as a tool is used to perform the deformations. There are a number of deformation and shaping tools provided in 3D modeling packages, these tools include armatures, lattices, bevel and sweeps.

Armatures are used to animate an object as armature deformations are used to simulate movement. Armatures are placed and associated with components of an object manually.

Lattices are capable of deforming multiple sets of vertices at a time [Roosendaal et al., 2004]. The displacement of a single lattice point affects all of the object vertices associated with that point. This makes deformations easy to perform, as the majority of the work lies in associating

the object vertices with lattice control points. Another advantage is that the displacement of a single lattice control point influences a number of object vertices.

The bevel tool is used to smooth the edges of a model, by reducing the size of faces and inserting new faces to fill up spaces. The new faces are curved to introduce smoothing.

The sweep tool is used to create cylindrical shapes by sweeping the object around an axis [NewTek, 2001]. This tool requires the modeler to perform the sweep by modeling the initial object, and then selecting how and about which axis it should be swept.

None of the deformation tools discussed are appropriate for use in a non-interactive modeling environment, as they are too reliant on manual application to the model. Saastamoinen [1999] documents a human modeling tutorial that uses curves depicted on a background as basis for the displacement of vertices. This approach has great potential in terms of use in a procedural modeling environment. Provided a strategy can be found that handles the displacement of vertices to match a curve, we have found a mechanism for performing model shaping.

### **2.3.2.2 Procedural Modeling Constraints**

All of the 3D modeling packages provide a significant amount of modeling tools and facilities. Each of which is interactive in nature. The creation of a large set of tools for procedural modeling is unnecessary. A smaller, non-interactive, tool set that provides the operations required to create and shape a model is adequate.

Each of our modeling tools is applied procedurally. The use of selections provides a standardised form of input, which can be used by each of the tools. The tools are robust enough to handle malformed and empty selections. This is necessary, as there is no interaction to correct a process that has gone awry.

The tools are applicable to a variety of models. Similar to their manual modeling counterparts, the procedural modeling tools can be used to create a variety of objects.

### **2.3.2.3 Procedural Modeling Tools**

The operators provided by procedural modeling languages include union, intersection and subtraction [Cartwright et al., 2005, Cutler et al., 2002], the synthesis of shapes on existing geometry [Perlin and Hoffert, 1989, Velho et al., 2001], weathering of geometry [Cutler et al., 2002, Musgrave et al., 1989], cutting operations [Szeliski and Tonnesen, 1992] and extrusion [Parish and Muller, 2001, Reeves et al., 1990, May et al., 1996, Lewis and Jones, 2004].

The geometric operations a system needs to create a model depends on what model the system is creating. Operations include transformations [May et al., 1996, Reeves et al., 1990],

subdivision and curve shaping [Morkel and Bangay, 2006].

### 2.3.3 Model Representations

This section looks at the various model representation available in 3D modeling packages and existing procedural modeling strategies. These are important, as our model representation needs to be compatible with existing modeling systems.

#### 2.3.3.1 3D Modeling Package Representations

Modeling packages support numerous object representations. These include polygon meshes, spline surfaces, subdivision surfaces, implicit surfaces, and particle systems [Autodesk, Inc, 1999a, Corporation, 2001, NewTek, 2001, Roosendaal and Selleri, 2004].

The polygonal meshes include planes, cubes, tubes, cylinders and cones. The spline surfaces include Bezier curves and circles and NURBS curves and circles. Subdivision surfaces are objects that have had subdivision applied to them. Implicit surfaces include all variations of *meta object*. Particle systems are primitive points that can be represented by small spheres.

#### 2.3.3.2 Procedural Modeling Representations

The data structures that procedural modeling techniques use differ according to the requirements of the system. Most systems use a polygonal surface representation.

The Modeling Environment system makes use of bi-cubic patches, polygons, quadric surfaces and spheres [Reeves et al., 1990].

Cutler et al. [2002] make use of tetrahedral meshes and signed distance fields to represent volumetric models. Fournier et al. [1982] make use of fractal polygons, and meshes to represent terrain. Marshall et al. [1980] make use of predefined elements to create a scene, these elements are meshes. The models are capable of storing additional information about the objects, such as the region the object occupies and the object complexity. The shape synthesis system of Velho et al. [2001] uses a combination of meshes and subdivision surfaces.

The Oriented Particle system creates surfaces through the use of particles. The particles are positioned in a grid-like fashion [Szeliski and Tonnesen, 1992]. Reeves [1983] makes use of particles to create volumetric models such as fire and clouds.

Parish and Muller [2001] make use of several data structures to model, and then represent models. The road map is a graph, the building allotments are polygons, the buildings themselves are strings, and the resulting geometry is also polygonal.

Glass et al. [2006] use Voronoi diagrams and polygonal meshes to model informal settlements.

## 2.4 Summary

This chapter discusses aspects pertaining to the modeling strategies, modeling tools and model representations of both 3D modeling packages and procedural modeling techniques. The facilities and features of existing procedural modeling languages are reviewed. This allows us to identify a set of requirements for our procedural modeling language.

The common 3D modeling strategies identified include the box modeling, flat mesh modeling, patch modeling and lofting, curve modeling and deformation modeling. Several of these strategies are potentially suited to adaption in a procedural modeling context.

The investigation of modeling tools found in 3D modeling packages and procedural modeling techniques shows that logical operators, subdivision and extrusion are common to both modeling paradigms. The 3D modeling tools that we have identified for adaptation to a procedural modeling context are a selection mechanism, extrusion, subdivision and stitching.

The model representations that are common to both 3D modeling packages and procedural modeling include spline surfaces, polygonal meshes and implicit surfaces. The mostly commonly found model representation, in both modeling paradigms, is a polygonal mesh.

Based on the comparison of existing procedural modeling languages, we conclude each of the languages extends an existing programming language which provides looping and control constructs and data structures for the procedural language. Common features include abstraction, standardisation, parameterisation, predefined variables and default parameter values.

# Chapter 3

## The Procedural Modeling Language

This chapter focuses on the procedural modeling language created for our non-interactive modeling environment.

The first aspect discussed is the requirements of the procedural modeling language in terms of facilities that are provided and the procedural modeling benefits that are supported.

The next aspect is the design strategy used for the procedural modeling language, which addresses what approaches are followed to create the procedural modeling language.

The final aspect considered is the implementation of the procedural modeling language. This discussion focuses on implementation-specific decisions that are made.

This chapter concludes by providing a summary of the capabilities of the procedural modeling language being discussed.

The procedural modeling language supports two types of modeling strategies, a constructivist modeling approach and an extrusionist modeling approach. The constructivist modeling approach creates new models by combining existing models or components. The extrusionist modeling approach creates models by extending a simple primitive shape.

The functionality for each of these modeling techniques is built into the modeling language.

### 3.1 Requirements

Section 2.2.2 discusses the existing procedural modeling languages and identifies the advantages of using a procedural language and the facilities of each of these languages. These advantages are abstraction, extensibility and standardisation.

The requirements of the procedural modeling language, as identified in Section 2.2.2, are parameterisation, language controlled level of detail, data structures, looping constructs, control

constructs, variables, a pseudo-random number generation facility and caching. Two types of model creation strategies have been identified, and the provision of both of these is a requirement of the procedural modeling language.

The benefits of procedural modeling (see Section 1.3.1) that the language supports are re-usability, increased model complexity and base-shape independence.

### **3.1.1 Procedural Modeling Requirements**

This section describes each of the procedural modeling language requirements, listed in Section 3.1, in more detail.

#### **3.1.1.1 Modeling Strategies**

Two types of modeling strategies, namely extrusionist and constructivist, are supported by the procedural modeling environment. The extrusionist approach constructs an entire model from a single starting primitive. The constructivist modeling approach combines components of existing objects to create new models.

The type of modeling being performed is specified by the modeler, and provision of this facility is necessary.

#### **3.1.1.2 Parameterisation**

Many procedural techniques use parameters to control model generation [Chen et al., 2002, Cutler et al., 2002, Parish and Muller, 2001, Reeves, 1983, Upstill, 1993]. In a non-interactive modeling environment, parameters provide sufficient, if limited, control over the model creation process.

The use of parameters makes modeling procedures re-usable, as a number of model variations can be created from a single procedure (see Section 6.1.2). The use of default parameter values provides a failsafe mechanism if a parameter value is not specified. Several procedural modeling languages force the use of default parameter values [Cutler et al., 2002, May et al., 1996, Upstill, 1993].

An additional facility provided by some procedural modeling languages [Cutler et al., 2002, Upstill, 1993] is the option to provide parameters and their values in any order. This is a useful facility as the order of the parameters in a procedure becomes irrelevant, which removes the need to constantly refer to the order in which parameters have been declared in procedures. Another advantage is that parameters are set by assigning a value to the parameter name, which is of the

format *parameter = value*. This makes procedure calls meaningful as the parameter names are assigned values in the call which makes it immediately obvious what values have been assigned to which variables.

Considering the benefits of parameterisation, it is a requirement of our procedural modeling language. Default parameter values are a valuable contribution and have proved to be a feature of the languages that provide them. The ability to assign values to parameter names and call procedures with the parameters in an arbitrary order is a desirable facility for the reasons discussed previously.

### 3.1.1.3 Level of Detail and Lazy Evaluation

The level of detail of an object is the amount of refinement performed on it according to its visibility in a scene, which is determined by the renderer [Hart, 1994, May et al., 1996, Reeves et al., 1990, Upstill, 1993].

The size of the bounding volume, as seen by the camera, determines how much refinement needs to be performed on a model. The larger the bounding volume, the more refinement is performed. The smaller the bounding volume, the fewer the refinements to a model. If the bounding volume is not visible at all, no refinement is done on the model as it is not currently in view of the camera.

The evaluation of a model as it is required is known as lazy evaluation and is supported by existing procedural modeling techniques [Hart, 1994, Upstill, 1993]. The use of this strategy provides a savings in computational overheads as models are only evaluated when they are required.

This is a valuable facility, as it provides a means of evaluating only the models that are required in the current scene. It also reduces the amount of work performed on a model to what can visually be appreciated.

### 3.1.1.4 Data Structures, Looping Constructs and Control Constructs

The existing procedural modeling languages provide looping and control constructs and data structures for use during modeling [Cutler et al., 2002, May et al., 1996, Reeves et al., 1990, Upstill, 1993]. Each of these languages extends an existing programming language, and these facilities are inherited from the base language.

The provision of these components in a procedural modeling language is essential. Much of the work done in procedural modeling is the repeated application of modeling tools or geometry to an object. The use of looping constructs facilitates this. The data structures are used to store

models or values required for the models to be generated. The control constructs provide the procedural language with the ability to perform logical operations.

### **3.1.1.5 Variables**

The use of variables in procedures allow intermediate model values to be stored, manipulated and retrieved during the modeling process. Each of the procedural modeling languages discussed in Section 2.2.2 caters for the declaration and use of variables.

Several of the languages provide predefined variables and parameters which are required for model creation and rendering processes [May et al., 1996, Upstill, 1993]. The advantage of providing predefined variables is that it takes the guesswork out of model creation and rendering. It is not up to the modeler to determine what needs to be set for the modeling environment, instead the modeler only needs to assign the relevant values the variables provided.

### **3.1.1.6 A Pseudo-Random Number Generation Facility**

Many existing procedural modeling techniques use reproducible random numbers generated from seed values [Chen et al., 2002, Greuter et al., 2003, Reeves, 1983] during modeling. The seed values are stored and can be re-used to generate the same pseudo-random number.

The provision of such a facility is desirable as it provides a means of generating pseudo-random numbers that can be repeatedly generated.

### **3.1.1.7 Caching**

Two forms of caching have been identified in existing procedural modeling techniques. The first is used by the particles in an oriented particle surface to keep track of neighbouring particles [Szeliski and Tonnesen, 1992]. The second form is used to store seed values for particle systems [Reeves, 1983] to facilitate the re-generation of scenes.

Caches are used to temporarily store objects for quick retrieval should the object be required again. A cache in the procedural modeling language allows model components to be re-used when using the constructivist modeling approach. The re-use of model components results in faster model generation times and less computational overhead as components do not have to be re-evaluated repeatedly (see Section 6.6). The reduction in model generation is not always significant and depends on the amount of repeated detail in the model.



### **3.1.2 Supported Procedural Modeling Benefits**

The procedural modeling benefits identified in Section 3.1 are supported by the procedural modeling language.

#### **3.1.2.1 Re-usability**

An advantage of using a procedural modeling approach is the re-usability of modeling procedures on different models in diverse modeling contexts. A procedural modeling language facilitates re-usability by enforcing the standardisation of procedures, which promotes inter-operability and re-usability of the procedure.

#### **3.1.2.2 Increased Model Complexity**

Another advantage of using a procedural modeling strategy is an increase in the complexity of models. The complexity of a model is increased by adding repetitive detail to the model. This is often done through the use of looping structures or repeated procedure calls.

#### **3.1.2.3 Base-Shape Independence**

An additional advantage of using a procedural modeling approach is the base-shape independence of modeling procedures. Arbitrary starting primitives can be used by modeling procedures, moreover, the result of applying a procedure to an arbitrary starting primitive is predictable.

Procedural modeling languages support base-shape independence by providing robust tools and facilities for model generation and manipulation

## **3.2 Design**

Section 3.1 discusses the requirements of the procedural modeling language. This section discusses how each of these requirements is catered for in the design strategy.

Based on the success achieved by existing procedural modeling languages and the benefits of extending an existing programming language, we have chosen to base the procedural modeling language on an existing language. This strategy reduces the amount of time required to create the programming language, while providing access to the existing functionality, support libraries, data structures, control constructs and looping constructs of the base language.

A full listing of the grammar is available in Appendix A.

### 3.2.1 Standardisation

The procedures created with the procedural modeling language are forced to conform to a standard layout, to ensure re-usability and interoperability. Although not essential, the use of standardisation provides error checking, as the grammar expects a specific procedure layout and returns an error should this requirement not be met. Standardisation also makes coding of procedures simpler, as the layout is fixed, predictable and cannot be deviated from.

Each procedure consists of several required and optional components. The required components include a return type, a procedure name, the type of procedure being created, a parameter list (even if it is empty) and the procedure body. The optional components include a bounding volume and a cache.

This is similar to the layout of procedures in programming languages, with the exception of the bounding volume, cache and type of procedure.

The layout of the procedures is best illustrated by the relevant grammar segment and discussion of what each aspect represents.

```
PML : boundingvolumedeclaration
      proceduredclaration
      body
```

#### 3.2.1.1 Bounding Volumes

The first aspect, the *boundingvolumedeclaration*, is used to specify a bounding volume for the model being created. This bounding volume is used to determine the level of detail of the model (see Section 3.2.5). The *boundingvolumedeclaration* nonterminal symbol is defined by this grammar segment. The use of a bounding volume is optional, as shown by the empty string as the final alternative of the grammar segment.

```
boundingvolumedeclaration : BOUNDINGVOLUME
                           boundingbody
                           |  $\epsilon$ 
```

The *BOUNDINGVOLUME* token is used to specify that a bounding volume is being used. The *boundingbody* nonterminal symbol is responsible for reading in the declaration of the bounding volume before the actual procedure declaration which follows the bounding volume information.

The *boundingbody* nonterminal symbol is defined by the following grammar segment.

```
boundingbody : '{' contents '}'
```

The *contents* nonterminal symbol is used to describe a body of code. The use of curly braces, '{' and '}', demarcates the boundaries in which the body of code must lie. The *contents* nonterminal symbol has the following specification.

The *contentselements* nonterminal symbol is defined by the grammar segment described next.

```
contentselements: BLOCK
                | PMLPROC
                | contentselements
                contentselements
```

The *BLOCK* token is used to describe the body of a procedure.

The *PMLPROC* token is used to identify calls to other modeling procedures that need to be rewritten to provide a full parameter list as required by the procedure.

### 3.2.1.2 Procedure Declaration

The layout of a modeling procedure is based on the format of procedures in programming languages, with two exceptions. The first is the optional specification of a flag before the declaration of the procedure and the second identifies what type of procedure is being created. A segment of the grammar has been included for discussion purposes.

```
proceduredeclaration: flags MODEL mode NAME arglist
```

The terminal symbol *proceduredeclaration* is used to define the procedure declaration being parsed.

- The *flags* nonterminal symbol is used to specify whether or not the procedure is cacheable.
- The *Model* token is used to specify that a model object is returned by the procedure. Every procedure written using the procedural modeling language is forced to return a model object. A facility to create supporting procedures for modeling is not provided. Each procedure is required to cater for all of its own calculations and modeling operations.
- The *mode* nonterminal symbol is used to identify whether the procedure being created uses a constructivist modeling approach or an extrusionist modeling approach to generate models.
- The *Name* nonterminal symbol is used to retrieve the name of the procedure. The *arglist* nonterminal symbol is used to return the parameter list of the procedure.

### 3.2.1.3 Body

The final component of a modeling procedure is its body. The *body* nonterminal symbol is used to parse the contents of a procedure and identify which parameter calls need to be rewritten. The *body* nonterminal symbol re-uses the *contents* nonterminal symbol discussed in Section 3.2.1.1. The exact grammar specification of the *body* nonterminal is provided in the following grammar segment.

```
body : '{' contents '}'
```

The grammar specification of the *body* nonterminal symbol is identical to the *boundingbody* nonterminal symbol (see Section 3.2.1.1).

### 3.2.2 Flags

The *flags* nonterminal symbol in a procedure declaration (see Section 3.2.1.2) is used to determine whether or not a procedure is cacheable. It is an optional setting, if a procedure is non-cacheable then no flag value is set. This can be seen in the grammar segment as the alternative option to the *CACHEABLE* token is an empty string. A grammar segment showing the definition of the *flags* nonterminal symbol is shown for illustrative purposes.

```
flags : CACHEABLE |  $\epsilon$ 
```

Currently, the only flag supported is caching. The *cacheable* keyword is used to specify that a model produced by a procedure may be added to the cache (see Section 3.2.9). The list of flags can be extended to support other procedure descriptors.

### 3.2.3 Mode

The *mode* nonterminal symbol in a procedure declaration (see Section 3.2.1.2) is used to determine whether or not a procedure uses a constructivist or extrusionist modeling approach. One of two keywords is used to specify which modeling strategy is used. The keywords are *out* and *inout* which describe the type of modeling approach used.

The *out* keyword describes a constructivist modeling approach. This is apt, as a model is not passed into a constructivist procedure, instead the model is created within the procedure and returned.

The *inout* keyword describes the extrusionist modeling approach. Again, this is an appropriate description as a model into an extrusionist procedure as a parameter. This model is then manipulated and then returned.

The grammar segment dealing with the *mode* nonterminal symbol is included for discussion purposes.

```
mode : OUT | INOUT
```

The type of modeling strategy employed by a procedure is required by the grammar. If a value is not provided, a syntax error is given by the compiler.

### 3.2.4 Parameterisation

As discussed in Section 3.1.1.2, the use of parameterisation in the procedural modeling language is advantageous. The procedural modeling language is parameterised and makes use of default parameter values. This is enforced by the grammar.

The procedural modeling language caters for parameters through the use of a standard procedure layout. The *arglist* nonterminal symbol from Section 3.2.1.2 is used to retrieve the parameter list of the procedure, and is defined as follows.

```
arglist : '(' argumentdecl ')'
argumentdecl : type NAME '=' NAME | argumentdecl
              ',' argumentdecl
type : INT | DOUBLE | STRING | VECTOR | BOOLEAN | CURVE
```

The *argumentdecl* nonterminal symbol takes parameters and their default values as input. This is repeatable, as procedures often have more than a single parameter. The input is of the form *parameter = value*, which enforces the use of default parameters. A syntax error occurs if a default parameter value is not provided.

The *type* token specifies the data type of each parameter. The data types catered for are the standard data types found in all programming languages.

Similar to existing procedural modeling languages, our language provides predefined parameters and variables (see Section 3.2.7). The parameters provided are for a camera object, a seed value for use with the pseudo-random number generator (see Section 3.2.8) and a default transformation.

The camera object is used to determine the level of detail of an object. The seed value can be used to generate a pseudo-random number if one is required by the model. The default transformation is for the initial placement of the model in a scene.

If the extrusionist modeling approach is used in a procedure, a model parameter is added to the parameter list. The extrusionist modeling strategy creates a model from a single object, thus requiring the model to be available in different modeling procedures that are used to construct the model.

### **3.2.5 Level of Detail and Lazy Evaluation**

The level of detail component makes use of the camera object provided as a predefined parameter. A bounding volume facility has been created for use by the level of detail calculations. The level of detail is provided as a predefined variable that can be used by the modeler to control the refinement levels of the model.

A model is enclosed within a bounding volume and is then evaluated by the camera object to determine whether or not the bounding volume is in the line of sight of the camera. If it is not, the model is not evaluated at all. If it is, the model is generated with a refinement level appropriate to the distance from the bounding volume and the camera object. The further away the object, the fewer the number of refinement steps are required.

Bounding volumes are also used to determine whether or not a model is visible in a scene. If the object is not visible, there is no need to evaluate it. If the bounding volume of a model is visible, in terms of the current viewing parameters of the camera, the model is evaluated. If not, the modeling procedure is not evaluated. Thus, lazy evaluation results prevents the unnecessary evaluation of models cannot be visually appreciated.

### **3.2.6 Data Structures, Looping Constructs and Control Constructs**

As discussed in Section 3.2, the procedural modeling language extends an existing programming language. The data structures, looping constructs and control constructs are inherited from the base programming language.

The data types have been supplemented with the addition of a model object, a curve object and a selection object.

### 3.2.7 Variables

In addition to predefined parameters, the procedural modeling language provides predefined variables that are required to generate a model.

A number of predefined variables are provided. A set of vectors is predefined to determine directions. A point of origin is also provided.

A predefined model variable is provided which is used to return the model at the end of the procedure evaluation. The initial value of the model variable differs according to whether the procedure uses a constructivist or extrusionist modeling approach.

If the constructivist approach is selected, the model variable is assigned a new instance as the initial value. If the extrusionist approach is selected, a new instance of the model variable is created and then assigned the value of the model that is added to the parameter list (see Section 3.2.4).

### 3.2.8 A Pseudo-Random Number Generation Facility

As discussed in Section 3.1.1.6, the provision of a pseudo-random number generation facility is desirable. Calls to the pseudo-random number generator are treated in the same manner as calls to modeling procedures (see Section 3.2.1.1). Pseudo-random numbers are used to introduce predictable chaos into a model.

### 3.2.9 Caching

Procedures that make use of caching are declared as such through the use of the *cacheable* keyword (see Section 3.2.2).

The decision of whether or not to cache a model is left to the modeler. The type of procedure being created (constructivist or extrusionist) determines whether or not a cache is usable.

The constructivist modeling strategy uses existing models to create new ones. The caching of components that are repeatedly re-used is desirable as it reduces the amount of time required to evaluate components and create a model.

The extrusionist modeling approach creates a model from a single starting primitive. The caching of such a model, although possible, does not aid in the model creation process. Before the cache can effectively be used with the extrusionist modeling approach, a strategy of caching and then re-using modeling operations and components needs to be devised. This is beyond the scope of this research and is left as a possible future extension to the procedural modeling language.

### 3.3 Implementation

The preceding sections discussed the requirements (Section 3.1) and design (Section 3.2) of our procedural modeling language. This section discusses the implementation details of the procedural modeling language with specific emphasis on each of the requirements listed previously.

#### 3.3.1 Implementation Decisions

In Section 3.2 it was decided to base our procedural modeling language on an existing programming language. We have chosen the Java programming language as the base language. The reason for this decision is that the procedural modeling language works with an existing modeling framework, RHoVeR (Rhodes virtual reality system) [Bangay, 2001, Bangay et al., 1996].

The grammar for the procedural modeling language uses Bison [Donnelly and Stallman, 2005], which is an open-source parser generator.

#### 3.3.2 Standardisation

The grammar used to provide a standardisation mechanism is described in Section 3.2.1. This section describes how the grammar is interpreted to create a Java class and procedure declaration.

The items that are necessary to describe a procedure in the procedural language: the flag of the procedure (Section 3.2.2) which is optional, the return type of the procedure (Section 3.2.1.2), the mode of the procedure (Section 3.2.3), the procedure name and the parameter list (Section 3.2.1.2).

All of this information is parsed and outputted into a Java class, which is compiled and then executed to create the model. The parsing process uses the following steps:

1. Create an empty class that has the same name as the procedure being parsed.
2. Add an import statement to the class file. This is to enable the use of the Java vector representation which is used extensively by the model representation (see Section 4.2).
3. Create a class by re-using the procedure name. Add the delimiting braces of the class.
4. If the procedure is marked as cacheable, add a static cache declaration to the class. A static declaration is used so that a single cache can be used over multiple calls of the procedure.
5. Add parameter list signature as a comment before procedure declarations. This is used for parameter rewriting, which is discussed in Section 3.3.4.



6. If a bounding volume is declared, rewrite it as a procedure of the class. This precedes the main modeling procedure.
7. Declare the modeling procedure to be a public static method that has the same name as the class.
8. Retrieve the parameter list provided by the modeler and add the predefined parameters (Section 3.3.3) to the beginning of the list. Determine if the procedure uses an extrusionist modeling approach by identifying the type of procedure that is being written. If it is an extrusionist approach, add a predefined model parameter to the parameter list. Update the parameter list to use Java wrapper classes to enable the use of default parameter values (Section 3.3.3.1).
9. Insert the predefined variables at the beginning of the procedure body. If caching is enabled insert code to deal with the cache into the procedure body (Section 3.3.7).
10. Rewrite modeling procedure calls (Section 3.3.4) replacing the original calls in the body of the procedure.
11. Write the body of the procedure to the class.

Once the interpretation is complete, the Java class can be compiled and executed.

### 3.3.3 Parameterisation

Parameterisation in the procedural modeling language is handled in the same manner as a programming language handles parameters, with the exception of the default parameter values. Java does not allow default parameter values to be declared, so we have provided the facility to do so in the procedural modeling language.

#### 3.3.3.1 Parameter Rewriting and Default Parameter Values

One of the requirements of the procedural modeling language is the ability to create modeling procedures with default parameter values. To force a procedure to use the default value associated with a parameter, a *null* value is passed to a procedure. Java does not allow the basic data types to be assigned a *null* value. Considering both of these shortcomings, we have devised a strategy that makes use of wrapper classes.

During the first pass of the compiler, each parameter along with its type and default value are stored in temporary variables for use during the rewriting step. During the parameter rewriting process, each parameter type is changed to use the associated wrapper class, for example an `int` becomes an `Integer` and a `double` becomes a `Double`.

Each modeler-defined parameter is declared as a variable within the procedure body. The type of the original parameters becomes the type of the variable declared within the procedure. The parameter name is used as the variable parameter, as the parameters of the procedure are used to perform modeling tasks. Conditional statements are also added to the body of the procedure to test whether a parameter value is set to *null*. If a *null* value is encountered, the default value of the parameter is assigned to it.

The name of the parameters in the procedure declaration are altered to differentiate between the wrapper class parameters and the locally declared variables.

The predefined parameters are added to the parameter list of the procedure, followed by the altered parameter declarations.

An example of the final result of this process is shown for illustrative purposes. The input procedure declaration is the first line of the example. The remainder of the lines is the output generated and added to the Java class file.

```
Model out Torso (double ratio = 0.6, int arms = 2)

static public Model Torso (Camera camera_w,
VRTransformation transformation_w, long randomBase_w,
Double ratio_w, Integer arms_w)

double ratio;
if (ratio_w == null)
    ratio = 0.6;
else
    ratio = ratio_w.doubleValue ();

int arms;
if (arms_w == null)
    arms = 2;
else
    arms = arms_w.intValue ();
```

### 3.3.3.2 Parameter Signatures

A parameter signature is created for each procedure for use with procedure call rewriting (Section 3.3.4).

An initial pass is done over each procedure that is being compiled. In this pass, each parameter name, type and default value is extracted and stored. The parameter signature is created by combining the type of procedure (out or inout) and each parameter's type and name. Each component is separated by a colon (:).

The data type of each parameter is not written to the signature in its primitive form, instead the associated wrapper class for each data type is used. This is necessary for the signature to be consistent with the parameter list that gets updated to use wrapper classes 3.3.3.1.

The signature is added as a comment to the Java class file, before the procedure declarations.

An example of a parameter signature is shown below the parameter list used to generate it:

```
Model out Torso (double ratio = 0.6, int arms = 2,  
                String name = "torso")
```

```
////Type Signature:OUT:Double,ratio:Integer,arms:String,name
```

Java does not allow procedures to have default parameter values, so we cannot rewrite the parameter list with the default values to the Java class file. Our solution is to implement a strategy that allows us to use default parameter values despite Java's shortcomings.

### 3.3.4 Procedure Call Rewriting

As discussed in Section 3.1.1.2, parameter rewriting is a useful facility to provide a modeler. A procedure call that is to be rewritten is marked, in this case, with a hash symbol '#'. In fact, any symbol that does not have special significance in Java is suitable for this purpose. The symbol is added to the beginning of the parameter call to notify the parser that standard parsing of the script needs to be halted to perform procedure call rewriting.

The full name of a procedure (*Class.ProcedureName*) is not used during a procedure call. If the procedure call is earmarked for rewriting, the class information for the procedure is automatically added by the parser. The values required for the predefined parameters of a procedure being called are also added automatically.

The parameters that are not being assigned values are left out of the procedure call. They are assigned their default values, as it is assumed that a *null* value is being passed to the procedure.

If a modeling procedure needs to be rewritten, the parser searches through each of the class files in the same directory as the current class file to determine if a matching procedure can be found. Once a corresponding class file is found, the parameter signature is retrieved and stored temporarily. Each parameter in the procedure call is retrieved and compared to the parameter signature to determine whether it is a valid parameter. If a match in the parameter signature is found, the corresponding wrapper class type is retrieved from the signature. The parameter in the procedure call is rewritten by removing the parameter name and replacing it with a new instance of its wrapper class, the assigned value is assigned to the wrapper class parameter.

For example, if a procedure with this declaration:

```
Model out Torso (double ratio = 0.6, int arms = 2,
                 String name = "torso")
```

is called like this:

```
#Torso (ratio = 0.6)
```

the final procedure call is:

```
Torso.Torso (camera_w, transformation_w, randomBase_w,
             new Double(0.6), null, null)
```

When the procedure is evaluated, the *ratio* parameter is assigned the value 0.6, the *arms* parameter is assigned the default value 2 and the *name* parameter is assigned the value "torso".

### 3.3.5 Level of Detail and Lazy Evaluation

Although the facilities have been provided (the camera object and bounding volume), the level of detail and lazy evaluation facilities remain incomplete.

We have provided a bounding volume facility which is used in combination with the camera object to determine whether or not an object is visible. A level of detail value is calculated based on the number of pixels a model covers on the screen.

This concept carries over to lazy evaluation as well. Both features are reliant on the use of bounding volumes, although, it is beneficial to perform lazy evaluation first, as there is no need to consider an object for refinement if it is not visible at all.

### 3.3.6 A Pseudo-Random Number Generation Facility

Seeded pseudo-random numbers are generated by a Java class created for the purpose. The constructor permits the modeler to decide between the use of Perlin noise and a Java pseudo-random number generator.

The noise function returns a 3D noise value. The code for the noise function is taken from Perlin [2002]. The Java random number generator takes as input a seed value and returns a pseudo-random number.

The random seed value associated with each procedure is used as the default seed value to the random number generator. This seed value may be supplemented with a modeler-specified seed value, which is then added to the call to the pseudo-random number generator as a parameter.

### 3.3.7 Caching

The extrusionist modeling approach is not suited to caching as the modeling strategy uses a single primitive to create a model.

Caching provides a saving on computational overheads by re-using models, or components of models. A trade-off is made, as the cache is maintained in memory. Although a model is not repeatedly evaluated, the cache is searched each time a model is required.

The cache facility provided makes use of Java hash tables [Goodrich and Tamassia, 2001].

A procedure declares a cache object if the model it produces is cacheable (Section 3.2.9). The cache declaration is static so that a single cache object is used for each call of the procedure (see Section 3.3.2).

```
public static Cache cache; //declaration
```

If a procedure is cacheable, a check is added to the procedure body to facilitate the instantiation of the cache object.

An example of the check statements added to a procedure are shown to aid the discussion.

```
//If the cache is null, create a new one
if (cache == null)
{
    cache = new Cache ();
}
```

A procedure signature is used to compare whether or not a model is in the cache. This signature is created by combining the values of the current procedure call into a string, including the random seed value of the procedure. Each parameter value is separated by a percentage symbol (%).

An example of the declaration of a procedure signature is shown.

```
//create a signature for comparisons
String signature = "%" + (ratio) + "%" + (arms)
                  + "%" + (name) + "%" + (randomBase_w);
```

To test whether a model is already in the cache, another check is added to the procedure. This check is used to compare the current procedure signature with those already in the cache. If a match is found, then the model associated with the signature is returned. If a match is not found, the model is added to the cache and its signature is used to reference it in the cache.

An example of the check used is shown for illustrative purposes.

```
//Check the cache for an existing entry
if (cache.checkSignature (signature))
{
    model = (Model)cache.getObject (signature);
    Model cacheModel = new Model ();
    cacheModel.mesh = model.mesh.copyMesh ();
    return cacheModel;
}
else
{
    CreateModel ();
    //If we have fallen through, we need to add the model
    //to the cache
    Model cacheModel = new Model ();
    cacheModel.mesh = model.mesh.copyMesh ();
    cache.putObject (signature, cacheModel);
    return model;
}
```

To facilitate the addition and retrieval of models to and from the cache, a new model is created. This is necessary to overcome the references to objects that Java maintains.

If a new model is not created each time, the reference to the object in the cache is retrieved, not the object.

## 3.4 Conclusions

This section presents a brief summary of the related work covered in this chapter, which is followed by a discussion on the challenges, contributions and conclusions from this chapter.

### 3.4.1 Summary

The requirements for our procedural modeling language are based on the features and facilities found in existing procedural modeling languages. These requirements are abstraction, standardisation, parameterisation, default parameter values, predefined variables and programming constructs. The list of requirements is supplemented by the procedural modeling benefits, which includes support for language-controlled level of detail, increased model complexity, base-shape independence and procedure re-usability. Other requirements identified are procedure call rewriting, caching and a pseudo-random number generator.

Abstraction is a result of using a procedural modeling language. The standardisation of procedures is enforced by controlling the layout of procedures.

The use of parameters is also enforced. Even if no parameters are specified by a modeler, a procedure has a parameter list consisting of predefined parameters. Each of the parameters specified by the modeler is forced to have a default parameter value.

Predefined variables and parameters are provided in each compiled procedure. These are used to provide support for the modeler and to return a model to be rendered.

The programming constructs for the procedural modeling language are provided by Java, as this is the programming language being extended.

The procedure call rewriting facility transposes a given procedure call to match what is required by the procedure.

The caching facility provides a means of temporarily storing geometry for re-use during model generation.

The pseudo-random number generator provides a facility for generating reproducible random numbers from seed values.

### 3.4.2 Conclusions

The ability to use parameters in procedures provides a way of creating a number of distinct models from a single modeling procedure. The use of default parameter values makes the modeler's task simpler as there is no need to provide parameter values for a model.

The procedure call rewriting facility provides several benefits to a modeler. These include the ability to assign parameters values in a logical way, the specification of these parameters in any order and the use of default parameter values for parameters not assigned values. Each of these contributes to making the procedure writing process simpler.

The caching facility provides temporary storage for evaluated geometry which can be re-used in a model. This facility reduces the amount of time required to create models as components can be re-used.

The language-controlled level of detail facility controls the number of refinement steps required to generate a visually satisfying model. The visual satisfaction of a model equates to the amount of detail that can be visually appreciated. The lazy evaluation aspect determines what is visible in a scene and what is not.

The modeler is thus not responsible for what is generated in a scene and how much refinement is required to generate a model. This is an automatic process, and does not rely on modeler interaction.

The predefined variables provided make the procedure writing process easier as these are used to interface with the renderer. The modeler need only set these variables. A model is enclosed within a bounding volume and is then evaluated by the camera object to determine whether or not the bounding volume is in the line of sight of the camera. If it is not, the model is not evaluated at all. If it is, the model is generated with a refinement level appropriate to the distance from the bounding volume and the camera object. The further away the object, the fewer the number of refinement steps are required.



# Chapter 4

## Procedural Modeling Tools

Modeling tools are required for use in our procedural modeling environment. These tools are non-interactive and are applied to models procedurally.

The modeling language discussed in Chapter 3 provides a means for accessing our procedural modeling tools.

We have created a set of procedural modeling tools that are inspired by the modeling tools available in 3D modeling packages. The tool set consists of:

- A selection tool, which is used to identify portions of a model for manipulation.
- An extrusion tool, which is used to grow a model by creating new vertices, faces and edges.
- A curve shaping tool, which is used to deform selections of a model to shape it.
- A subdivision tool, which is used to create new vertices, faces and edges for a model, and simultaneously smooth the model. The level of smoothing is controlled through the use of a smoothing factor.
- A stitching tool, which is used to join parts of a model, or to create a new model from other models.

These tools have been selected, as they can be adapted to function non-interactively.

The remainder of the chapter discusses the following aspects: Section 4.1 discusses the specification of each tool. Section 4.2 discusses the model representation we have created for the models and selection representations. Section 4.3 discusses the implementation of the procedural modeling tools and the data structures. Section 4.4 concludes the chapter by discussing the challenges faced, and contributions made.

## 4.1 Tool Specifications

An experiment to determine the strategies used to create human models by hand is presented first. The aim of this experiment is to document the approaches used by a group of participants to model a human.

The remainder of this section discusses the procedural modeling tools listed in Section 4. This includes a discussion of the design strategies considered and used.

### 4.1.1 Experiment

There are various ways of modeling the human form, and these techniques are largely reliant on the modeling context (for example art or anatomy) and the individual performing the modeling. This experiment aims to document as many of these approaches as possible. The experiment focuses on the methods used by eight participants while creating a human model.

#### **Aim:**

The aim of the experiment is to determine what modeling strategies are used to create and shape a human model by hand.

#### **Experiment Setup:**

- The modeling medium selected is play-dough.
- The human models have to be constructed on a piece of blank white paper.
- No tools are provided, but anything the participant finds available and useful is permitted.
- The participants were seated in a room away from casual observation, in a horseshoe, with all of the participants facing inwards.

#### **Measurements:**

The only measurements taken are in terms of the modeling approach adopted by each of the participants. A video clip records the modeling process for each participant. The final model is photographed.



Figure 4.1: Examples of the human models created during the modeling experiment

### Participants:

The experiment consisted of eight participants in total. These participants were members of a Rhodes Computer Science special interest group.

### Interaction:

During the experiment, our interaction with the participants was limited to asking questions about how details and models were added and constructed. The entire experiment was recorded, which was done by walking around the tables, viewing the models.

### Results:

Table 4.1 briefly describes the method used by each of the participants to create his or her human model.

Figure 4.1 shows the result of this modeling experiment. An image of each participant is shown.

One of the eight participants used an extrusion-based approach to model his human. The remaining seven participants broke their dough into pieces and shaped each body part individually. Once the parts had been modeled, they were stuck together to create the final human model.

From this, we can conclude that extrusion is not an intuitive way of modeling humans with play-dough. The constructivist modeling approach, which creates models from existing components, is used more frequently.

Participant	Technique
Participant 1	Divide the dough into quarters and create body parts from the pieces. The arms and legs were modeled from single pieces of dough (i.e. one piece of dough for the legs, and one for the arms). To model detail, for example a hand. Create the arm by using a slightly flattened sausage. Make small sausages for the fingers and attach to the hand.
Participant 2	Separate the dough into pieces to perform modeling. Detail was added through the use of small pieces of dough, or separation (knife tool - finger nails)
Participant 3	Broke the piece of dough into smaller pieces which were individually modeled into parts of a human. Detail was added using a knife tool (finger nails and pen) to create toes, and musculature. Speedo is a piece of differently coloured dough.
Participant 4	Used one piece and extruded parts of the body from this piece. Detail was added by using differently coloured pieces of dough - the clothing. Pens were used to add details such as eyes, mouth and decorations on the clothing
Participant 5	Flattened the piece, and then started folding it into a body. To get proportions, he sketched an outline of a human on a piece of paper, and moulded his person according to that. The face was made of a different colour piece of dough
Participant 6	Broke dough into pieces, and moulded body parts. Hair was small sausages. Fingers made with a piece of cardboard, effectively a knife tool
Participant 7	Used one piece of dough, from which he extruded all body parts. Details were added with pens and finger nails. Shoes, hair and facial hair were added by using green coloured play dough. A cigar was added, which was half a sweet
Participant 8	Broke dough into pieces. Added a different coloured head, used sweets for eyes and headphones. Nail varnish was added with blue pieces of dough. The human was sitting in a meditative pose

Table 4.1: The Modeling Results for Participants 1 - 8

	Advantages	Disadvantages
Original Model	No need to copy the model, as operations are performed directly on the original. This removes problems associated with referencing.	Should the need for an unchanged copy of the original model arise, none will be available.
Copy of the Model	The original model remains intact, as the operations are performed on the copy.	A referencing problem between the original model and the copy occurs, as references are no longer to the same object.

Table 4.2: A comparison of advantages and disadvantages associated with each extrusion approach

### 4.1.2 Comparison of Strategies

Each of the tools operate on the original model. An alternative to this approach is to perform the operations on a copy of the model, which is then returned to the calling function. Each method has advantages and disadvantages. Table 4.2 illustrates the advantages and disadvantages of each approach.

### 4.1.3 Selection

A selection mechanism is required to identify faces, vertices and edges of a model for manipulation. In a manual modeling environment, selections are performed interactively.

Scripting languages in 3D modeling packages require a method of identifying faces, vertices and edges for manipulation. Interactive selection is infeasible, as the scripting interface of the modeling package is used. A brief survey of the available scripting languages shows that the selection routines available in scripting are very limited [Bayne et al., 2004, Brooks et al., 2001, NewTek, 2001]. The objects created are assigned unique names, which are then used in the identification and selection of faces, for example *cube.f[4]* represents the selection of the fourth face from an object named *cube* [Brooks et al., 2001].

### 4.1.4 Design

Two design strategies were suggested to determine the best method of implementing a selection tool. Each of these strategies is discussed, and the strengths and weaknesses of each approach are identified.

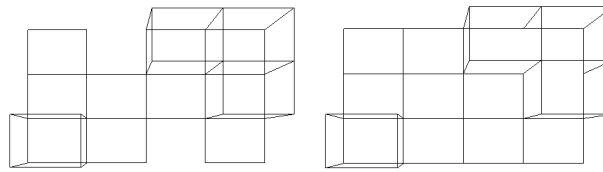


Figure 4.2: The incorrect (left) and correct (right) removal of extruded faces

The first strategy considered is the use of face, vertex and edge indices. This strategy is the same approach as that used by the scripting languages in 3D modeling packages.

The second strategy uses labels and regions to identify faces of interest. Each face of the model is assigned a label, which is used to identify which component of the model the face belongs to. The regions refer to areas of model space which are used to perform selections. If a face falls within a specified region of space, it is considered selected and a corresponding label is assigned to it.

#### 4.1.4.1 Indices

The initial design of the selection tool follows the same principles of the scripting languages reviewed. We make selections based on index numbers. The indices relate to the position of the face, vertex or edge in the lists maintained by the model. This form of identification works, but is less than ideal as it requires knowledge of the inner workings of the model. The selection of indices is a matter of guesswork and relies on the use of numbers to specify index values. Another pitfall of this approach is that indices change due to operations (such as deletions and extrusions) being performed on the model, which results in the incorrect vertex, face or edge being extruded or deleted. Figure 4.2 illustrates both the incorrect and correct removal of extruded faces because of changing face indices.

There are several shortcomings associated with this design strategy which can be summarised as follows.

- The use of indices requires knowledge of the underlying mesh structure of a model.
- The use of indices requires a large amount of guesswork and experimentation, as the indices tend to change due to operations.

- Indices change due to the removal of originating faces, and if more than one face is extruded in a single extrusion operation, it is a process of trial-and-error when attempting to locate the correct face to perform operations on.
- The use of indices has an undesirable effect if a particular instance has been catered for, for example catering for an odd number of extrusions and using an even number.

The shortcomings associated with the use of indices prompts the exploration of an alternative design strategy.

#### **4.1.4.2 Labels**

The use of labels provides a means for identifying the region to which a face belongs. Labeling solves the problems associated with face identification, as there is no longer any reliance on numerical values. The value of a label is not changed when a modeling operation is performed, instead the value is only altered by assigning a new value. This makes the use of labels reliable. Several problems arise, however, if each face requires a unique label. One problem is that it becomes difficult to maintain a list of unique labels. This situation is no better than using indices, as we still have to guess which faces are which. Another problem is the assignment of these unique labels.

An alternative to labels are attributes which are more versatile. A face can have a variety of attributes, which is not restrictive in terms of a single property. MEL (Maya) uses a similar approach [Brooks et al., 2001], a set of existing attributes are provided with each primitive, but it is possible to add attributes as the need arises. Due to the similarity in the approaches, the attribute alternative shares the same triumphs and shortcomings of the labeling strategy.

#### **4.1.4.3 Regions**

By making use of labeling or attributes, it is possible to group faces that make up a particular area of a model into specific regions of space. Faces that form part of a region are distinguishable from other regions through the use of a keyword, or region specific identifier. The initial identifier need only be set once, and all faces that result from extrusions within a given area have the same properties.

Guessing and experimentation no longer have to be performed to find the correct face to assign a label to. By using regions of space to identify faces, we have created a scale-independent selection mechanism. To aid the labeling of selected faces, we can use inheritance. Every new

face that is created from a face within the selection inherits the label of the parent. By doing this, we do not have to repeatedly perform selections to cater for the new faces that have been added.

An aspect that we have to consider is how to demarcate the regions of space. One method of doing this is to use bounding volumes. The use of bounding volumes also creates the opportunity for hierarchical modeling. Each selection has a bounding volume associated with it, and is capable of keeping track of its parentage. By doing this, we are able to create parent-child structures. For example if a human is being modeled, the torso selection is the parent to the arms, legs and neck. The arms are parents to the hands, the legs are parents to the feet and the neck is parent to the head. In this manner, we are able to create an hierarchical model of a human. The use of hierarchical models is also useful for animation purposes. Each selection is easily identified by the region of space that it occupies, so it is easy to associate deformation primitives with the relevant portions of the model. The fact that selections keep track of their parent selections is also useful for connecting the deformation primitives.

An additional aspect that we can cater for using this approach is the storage of selections. By storing selections in the model, we are able to retrieve them later. It is impractical to store every selection performed, however, so it is possible to use selections without storing them.

#### **4.1.4.4 Summary**

The advantages and disadvantages of each selection strategy are summarised in Table 4.3.

The disadvantages of using indices for selections far outweigh the advantages. This has led us to choose the second selection strategy. A summary of the selection strategy that we have chosen is discussed.

We use regions of space, enclosed in bounding volumes, to identify selections. Figure 4.3 illustrates a model enclosed within its bounding volume. A selection is made by locating the faces that fall within a particular area defined relative to the dimensions of the bounding volume.

Such a selection strategy works well if the geometry concerned fits the bounding volume. A region selected relative to a bounding box then encompasses the relevant set of primitives in a manner unaffected by fine geometric detail, resulting in a robust selection mechanism.

Selections can be created in a hierarchical fashion, by allowing a selection within a selection. This representation supports the creation of hierarchical models, as components of a model can individually referenced, and extended by another procedure.

Two categories of selection are supported. Anonymous selections have the scope of local variables, and are used for intermediate modeling operations. Named selections are associated with the model, are globally accessible, and are used to identify important geometric compo-



Strategy	Advantages	Disadvantages
Indices	<ul style="list-style-type: none"> <li>• Numbers are an easy way of identifying objects</li> </ul>	<ul style="list-style-type: none"> <li>• Requires knowledge of the underlying mesh structure</li> <li>• Indices change due to operations applied to the model</li> <li>• Applying tools to the model in a different order affects indexing</li> </ul>
Labeling and Regions	<ul style="list-style-type: none"> <li>• Labels are an easy way of identifying objects</li> <li>• Labels are only changed when a command is issued to do so</li> <li>• The assignment of labels requires minimal knowledge of the mesh structure</li> <li>• Labels are inherited</li> <li>• Regions are scale-independent</li> </ul>	<ul style="list-style-type: none"> <li>• The use of regions of space can be inaccurate due to the shape of the model</li> <li>• All of the selections a face is associated with are recorded as attributes, which can make it difficult to find a specific face</li> </ul>

Table 4.3: The advantages and disadvantages of each selection strategy

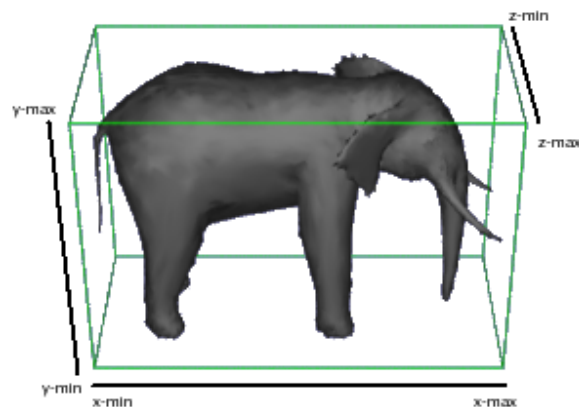


Figure 4.3: An elephant model enclosed within its bounding volume

nents. Named selections are preserved during modeling operations.

The hierarchical relationship amongst selections is explicitly maintained, and accessible by the modeling procedure. This hierarchical structure is intended to facilitate later operations on the model, such as the placement of a skeleton for the animation of the model.

### 4.1.5 Extrusion

The need for an extrusion tool stems from the use of such a tool in a manual modeling environment, and has successfully been used in human modeling [Ratner, 2003, Saastamoinen, 1999]. Extrusion is used to pull parts out of a model, which results in the creation of new vertices and faces. The selection tool is used to demarcate regions for extrusion.

#### 4.1.5.1 Design

To make the extrusion tool as flexible as possible, we need to ensure that it is capable of extruding several faces at a time. This saves on procedure calls, as we are able to extrude multiple faces with a single call. The ability to extrude multiple faces also results in water-tight meshes, as edges between extruded faces are removed. Only the edges that lie on the boundary result in new faces. A capping face is created for each of the original faces that was extruded. A facility to return the capping faces of an extrusion is useful, as the selection of capping faces no longer has to be performed.

### 4.1.6 Curve Shaping Tool

The idea for a curve shaping tool stems from the human modeling tutorial by Saastamoinen [1999]. Additionally, Grossman et al. [2002] use curves to represent the major contours of automobiles in a car design context.

Both of these systems rely on the use of human intervention, one for the shaping of models, and the other for the creation of the curves. We believe that the concept of curve-shaping is a good candidate for adapting to a non-interactive modeling environment.

#### 4.1.6.1 Design

Saastamoinen [1999] shapes the outline of a human model by aligning the model boundary with an image template containing a projection of the desired model. A non-interactive version of the same process provides a useful tool for modeling humans and other objects.

The curves depicting boundaries can either be created explicitly (manually) based on an image template, or preferably in a procedural fashion, with control points being influenced by parameters passed to the procedure.

A corresponding set of primitives on the boundary of the object is selected, and individual control points are displaced to match the curve. The quality of the resulting object can be improved by increasing the number of vertices (for example through the use of subdivision) which ensures a more accurate fit of the curve.

### 4.1.7 Subdivision

There are many subdivision schemes available, each of which has its own advantages and disadvantages. A survey has been conducted on some of the more commonly used subdivision schemes. This survey has led to the identification of several desirable attributes of subdivision which form the determining factor for why a particular subdivision scheme is chosen.

#### Desirable Attributes:

- The smallest possible increase in the number of faces at each subdivision step.
- The applicability of the subdivision scheme to a mesh of arbitrary topology.
- The resulting mesh, in terms of composition (either triangles or quadrilaterals). DeRose et al. [1998] note that it is better to use quadrilaterals to model objects such as arms, legs and fingers, as they are better able to capture symmetry.

Scheme	Face/Vertex Split	Refinement - A/I	Tiling - T/Q	Increase
Loop	Face	A	T	4 times
Modified Butterfly	Face	I	T	4 times
Catmull-Clark	Face	A	Q	4 times
Kobbelt	Face	I	Q	4 times
Doo-Sabin	Vertex	A	Q	4 times
$\sqrt{3}$ Subdivision	Face	A	T	3 times
Mid Edge	Vertex	A	Q	2 times
4-8	Face	A	Q	2 times
$\sqrt{2}$ Subdivision	Face	A	Q	2 times

Table 4.4: Various Subdivision Schemes

- Adaptive mesh refinement to obtain regularity over the model.

Zorin and Schroder [2000] identify two types of refinement that are used in most subdivision schemes, these refinements are *face split* and *vertex split*.

Face split refinement splits each face in a mesh into four new faces. The original vertices are retained, while new vertices for each edge are added, with an additional vertex in the center of the original face for quadrilaterals.

Vertex split refinement removes the original vertices, and replaces them with a number of new vertices (one new vertex for each face adjacent to the old vertex). The original faces of the mesh are retained, and one new face is added for each new vertex.

The face split refinement strategy can be broken down further, into *interpolating* and *approximating* [Zorin and Schroder, 2000]. Interpolating schemes retain the original vertices of a mesh throughout the entire refinement process. This results in a set of vertices at different levels of subdivision that are all the same vertex. The problem with an interpolating strategy is a loss in quality. The more interpolatory the strategy, the more quality is lost.

Approximating schemes estimate the control mesh, as the strategy is analogous to using a spline patch to shape the control mesh.

#### 4.1.7.1 Subdivision Schemes

Table 4.4 lists some subdivision schemes and their attributes [Li et al., 2004, Zorin and Schroder, 2000]. The refinement is either approximatory or interpolary (A or I), and the resulting tiling is either triangular or quadrilateral (T or Q).

Based on our initial criteria, the schemes under consideration are the  $\sqrt{2}$  subdivision scheme

[Li et al., 2004], the midedge subdivision scheme [Peters and Reif, 1997] and the 4 – 8 subdivision scheme [Velho and Zorin, 2001].

The midedge subdivision scheme works by adding in new edge points at each refinement step, and connecting these new points to form new faces.

The 4 – 8 subdivision scheme works on a surface that is comprised of quadrilaterals, that are represented as two isosceles triangles lying on the diagonal of the face. The subdivision process works by inserting and then connecting vertices on the internal edges of faces to form new faces. This scheme supports adaptive subdivision.

The  $\sqrt{2}$  subdivision scheme is a variation of the 4 – 8 subdivision scheme.

### Comparison of Subdivision Schemes

The midedge subdivision scheme is one of the first schemes to introduce a two-fold increase in the number of faces at each subdivision step. Two applications of the midedge scheme are equivalent to a single step in the Doo-Sabin subdivision scheme, which has a four-fold increase in face number.

The 4 – 8 subdivision scheme and  $\sqrt{2}$  subdivision scheme are similar in functionality. The 4 – 8 scheme works on a *tri-quad* (quadrilaterals split into two triangles) meshes, and must therefore adapt a given control mesh to this type of layout. The  $\sqrt{2}$  subdivision scheme does not have this requirement, it is capable of subdividing any given mesh without extra steps being required.

On the whole, the  $\sqrt{2}$  subdivision scheme suits our needs the best. The scheme supports a two-fold increase in faces at each step, does not require a specific mesh layout to function, supports adaptive subdivision and results in meshes containing quadrilateral faces.

#### 4.1.7.2 Design

The  $\sqrt{2}$  subdivision scheme of Li et al. [2004] is used to facilitate the generation of additional faces, and the smoothing of models. There are several benefits to using this subdivision scheme which include:

- The number of polygons at each step increases by a factor of two, unlike most other schemes which usually increase the polygon count by three or four. Providing finer grained control over level of detail.
- The approach works for meshes containing polygons with variable numbers of sides.

- The  $\sqrt{2}$  subdivision scheme supports adaptive subdivision, allowing refinement to apply only to specific portions of the model.

Adaptive subdivision is used to control the level of detail within a model, as well as the refinement of larger polygons to get an even spread of regularly sized polygons over the entirety of the model. Adaptive subdivision makes use of selections to identify where it must be applied.

Through the use of a smoothing factor, we are able to control the level of smoothing that occurs at each step of subdivision.

Each subdivided face inherits the labels from the original face.

### 4.1.8 Stitching

The stitching facility makes it possible to create new models from existing ones, or to use model components from within the cache. Funkhouser et al. [2004] create new models through the use of an object model database and stitching. It is possible to search the object database to find desirable components. These components are removed from the originating model through the use of intelligent scissors, and are pasted onto the new model through the use of stitching.

#### 4.1.8.1 Design

The design for the stitching method is the same as that of Funkhouser et al. [2004]. The process starts with the identification and selection of two boundary contours (on two separate models). The closest vertices in each contour are identified first to start the stitching. Once these vertices are found, new faces are created between them, and the next vertex in line. After the initial case has been dealt with, the process continues by traversing one boundary and finding the closest corresponding vertex in the other mesh. This process is continued until no vertices are left in either contour.

Each face in a model keeps a record of its vertices. The list of vertices are stored in the order that they appear in the face. By using this information, we are able to determine which direction to traverse the vertices in each boundary contour.

## 4.2 Data Structures

Two types of data structure are discussed in this section. The first of these is the actual model representation, while the other is the selection representation which is used by each of the modeling tools.

### 4.2.1 Model Representation

Procedural modeling strategies use a variety of model representations (see Section ??), but one of the most commonly used representations is a mesh structure [Cutler et al., 2002, Fournier et al., 1982, Marshall et al., 1980, Parish and Muller, 2001, Reeves et al., 1990]. Modeling packages also offer a large variety of representations, one of the most prominent is a mesh representation. In light of this, we have decided to use a mesh structure as our fundamental data structure.

The non-interactive modeling tools can be used on a variety of model representations, provided they are implemented to do as much. The same design for the tools can be used.

#### 4.2.1.1 Design

In addition to providing the basic mesh structure, we need to provide facilities with which a traversal of the mesh components may be performed. An adaptation of the *Winged-Edge* representation [Joy et al., 2002] is used. The mesh is comprised of sets of faces, vertices and edges. Each vertex maintains a list of edges and faces that it is part of. Each edge maintains a list of faces that it is part of, and it keeps track of its starting and ending vertices. Each face keeps track of the vertices and edges that comprise it. In addition, the mesh class maintains individual lists of vertices, edges and faces respectively.

The vertex and face lists of the mesh need to be maintained and updated after each modeling operation that is applied to the mesh. We have two choices regarding the update of the edge list, we can either maintain the list as we maintain all of the other lists, or we can update the edge list as it is required. The former approach has the advantage that the mesh structure is always consistent and up to date. The disadvantage of this approach is that it requires time and resources to maintain this list.

The alternative approach has the advantage that the edge list is updated only when it needs to be. The disadvantage is that the vertex, edge and face lists are not synchronised and contain outdated information.

We have chosen to update all of the mesh components at the same time. There are two advantages to using this approach. A consistent mesh makes the traversal of vertices, edges and faces simple as the stored information is up to date. The second advantage is that updates to the mesh are not as resource intensive, as only the components that need to be updated are. If the edge list is maintained concurrently with the other components, it requires less time to update the lists, as continuous updates are performed.

### 4.2.2 Selection Representation

Selections are frequently used in 3D modeling packages to identify vertices, faces or edges for manipulation. We have designed a procedural equivalent to manual selection (see Section 4.1.3).

Selections are a significant component of procedural modeling and as such we use a standardised selection representation which is used throughout the procedural modeling environment. This allows compatibility between the modeling tools, as selections are reusable and are capable of working on the same structures.

#### 4.2.2.1 Design

There are two forms of selection that we wish to represent. These are *anonymous* and *named* selections (See Section 4.1.3). Both types of selection store a set of faces with their corresponding lists of vertices and edges. An anonymous selection is a temporary selection that is used to perform an operation, but is not stored for later use. A named selection is stored in the model representation for later retrieval. The selections that are stored can be retrieved from the mesh to be used as a reference selection. The ability to store specific selections is also useful for future extensions, such as animation, to the procedural modeling paradigm.

We need a selection representation that can be used as either anonymous or named selections. A named selection needs a unique identifier for retrieval, whereas the anonymous selection is temporary, so the value assigned to its identifier is irrelevant.

Another aspect of the representation required, is a facility in which to store the actual selection, in this case, a set of faces. Faces are used in the selections, vertices and edges are available through the connectivity information stored within each face. There is no reason that vertices or edges cannot be used in the selections.

Each selection has a parent property which is set to the identifier of the selection from which it is chosen. In this way, selections are capable of keeping track of their lineage. This results in a hierarchical model, as a parent-child structure is maintained by storing named selections in the model. By creating and maintaining a hierarchical structure, we are able to introduce animation primitives at a later stage.

## 4.3 Implementation

This section discusses the implementation of each of these tools and data structures. The implementation of the tools is discussed first. An example of the application of each tool is also



provided and discussed. The implementation of our data structures is discussed last.

### 4.3.1 Tools

This section discusses the implementation of each of our tools. Each discussion includes the motivation for the implementation choices we have made.

#### 4.3.1.1 Selection

The selection tool takes as input a minimum and maximum value for each dimension, a parent selection and a name for the current selection.

The minimum and maximum values specified are used to determine the valid ranges from which faces may be selected. These values correspond to the minimum and maximum extremes of the parent selection's bounding volume. The minimum and maximum parameter values are interpreted as a ratio along each of the bounding volume's axes, and faces that fall in these regions are selected. A face is considered selected if and only if all of its vertices are within the valid selection ranges.

Each of the selected faces is assigned the current selection's name, and is stored in a selection data structure.

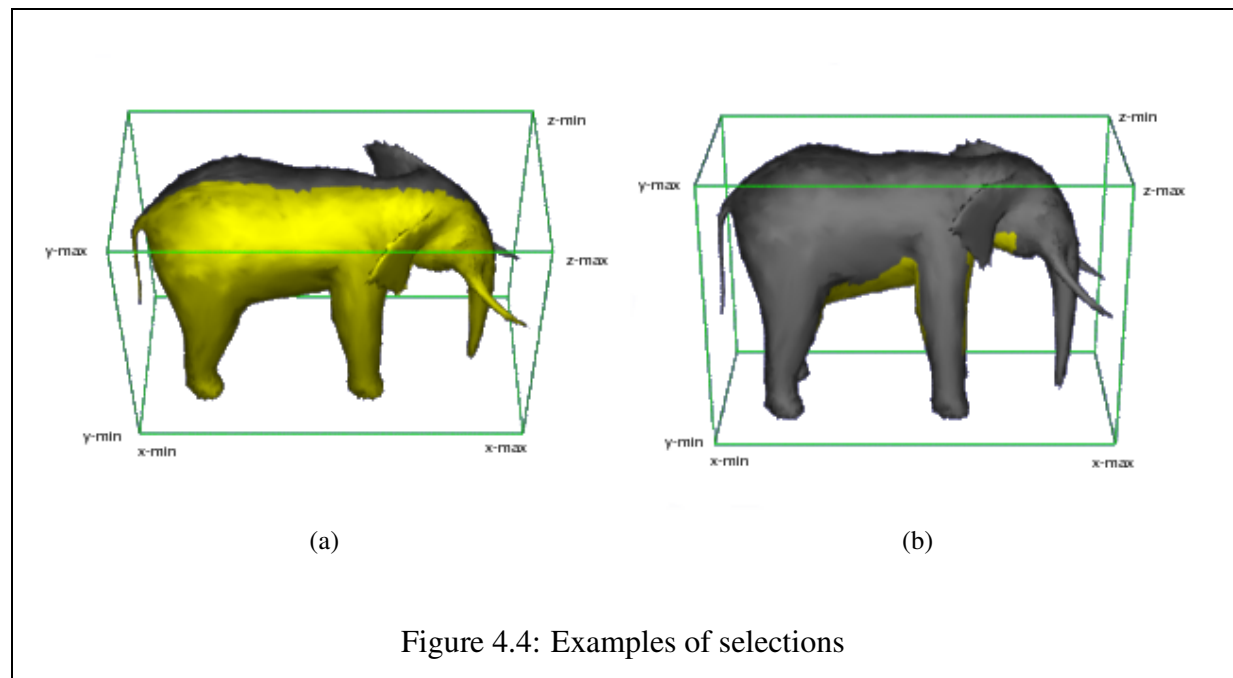
Implementation specific choices that were made include:

- Each of the minimum and maximum values is represented as a real value (double) to allow for the use of fractions.
- The bounding volume used is an axis-aligned parallelepiped. These bounding volumes are easy to implement, and are cubical, which makes the use of our selection ratios easier.

Two examples of selection are shown in Figure 4.4a) and b). The selection parameters for Figure 4.4a) are as follows:  $xmin = 0.0$ ,  $xmax = 1.0$ ,  $ymin = 0.0$ ,  $ymax = 1.0$ ,  $zmin = 0.5$  and  $zmax = 1.0$ . The selection parameters for Figure 4.4b) are as follows:  $xmin = 0.25$ ,  $xmax = 0.75$ ,  $ymin = 0.25$ ,  $ymax = 0.75$ ,  $zmin = 0.25$  and  $zmax = 0.75$ .

#### 4.3.1.2 Extrusion

The extrusion tool takes as input a selection and a vector specifying the magnitude and direction of the extrusion. A corresponding new vertex needs to be created for each vertex in the supplied set. The position of the new vertex is the position of the old vertex with the specified displacement



applied to it. These new vertices are then connected to the old ones to form new faces. The extrusion process is completed when a cap face is placed over the extrusion. This cap is made up entirely of new vertices.

To cater for the extrusion of multiple faces, the set of faces passed as input is traversed to firstly remove duplicate vertices and secondly to identify boundary edges.

Each vertex is copied and then displaced to create the new vertices to create new faces. Adjacent faces have vertices in common, and this is what leads to duplicate vertices. If these duplicates are not removed, the following problems arise:

- Extra vertices are created, as a new vertex is created for each existing vertex. This is a problem, as the mesh contains vertices that are not required.
- Several displacement operations need to be performed to move all of the newly created vertices. This is an unnecessary use of resources.

We are able to remove duplicates by storing each of the non-duplicate vertices in a list. Before adding a new vertex, the existing list of vertices is traversed to check whether or not the vertex has already been added. If the vertex is already in the list, the next vertex is retrieved and the process repeated. If the vertex has not been added to the list, it is appended to the end.

Edges are deemed to be boundary edges if they occur only once in the list of faces provided as input. Each boundary edge is used to create a new face for the extrusion process. The faces created on boundary edges and caps inherit the label information of the input set of faces.

The penultimate extrusion step is completed by capping the newly created faces with a new face for each of the originally provided faces.

The final step in extrusion is the deletion of the original faces passed through as input. This preserves mesh integrity and ensures that manifold meshes remain so, which is required by the  $\sqrt{2}$  subdivision scheme.

A feature that we have added to our extrusion tool is the ability to return the cap faces of the last extrusion. This is a useful facility as multiple extrusions can be performed by using the returned cap faces as the selection. The alternative is to repeatedly perform selections to identify the faces that need to be extruded.

In terms of actual implementation specifics, we have made the following choices:

- The set of input faces is passed through to the extrusion tool as a selection. Each of the modeling tools uses selections, so by using this data structure, we have made our extrusion tool compatible with the rest of the modeling tools. The capping faces that are returned are also in the form of a selection.
- The displacement and magnitude of the extrusion is represented as a vector. Vectors store both direction and magnitude.
- All of the lists used during the extrusion process are Java vector data structures. If we use an array to store this data, we have to traverse each of the faces, edges and vertices to obtain the number of elements required for the array. Thereafter we are able to work on each of these components. By using a vector class, we do not need to know the number of elements beforehand, a vector is able to expand to accommodate each element added to it.

#### 4.3.1.3 Curve Shaping

Bezier splines, B-Splines or Lagrange splines are used to create the curves to which the model is to be shaped. These types of curves are used in 3D modeling packages to shape objects.

The curve shaping tool takes as input a selection, a curve, a projection vector and a direction in which to displace control points. The selected region of the object is displaced to fall onto the surface formed from the curve extended along the projection vector. We do this by projecting both control points and curve onto a 2D plane perpendicular to the projection vector. This reduces

the complexity of the intersection calculation from a 3D problem to a 2D one. If we do not project the curve and model control points into 2D space, we have to test for intersections in 3D space.

Since any class of spline curve to be used, intersection is performed by approximating the curve with a pair of straight line segments, and testing for ray intersection on each of these. The closest curve section is then recursively subdivided, and the process repeated.

The new position of each model control point falls at the intersection of the projected curve and ray from the 2D control point in the direction of the displacement vector. The offset from the current position of the control point to the point of intersection is used to displace the model control point so that it falls on the curve.

Each of the curves is represented as a set of control points, with a curve degree that is controlled by the number of control points and a number of knots which is also dependent on the number of control points. The curves are renderable, which allows them to be displayed once a model has been generated. The curve itself is approximated by a number of small line segments.

#### 4.3.1.4 Subdivision

Subdivision of a model is viewed as the application of adaptive subdivision to the entire surface of the model, as such, we are going to discuss the implementation of adaptive subdivision. Adaptive subdivision may be applied to individual selections from a model.

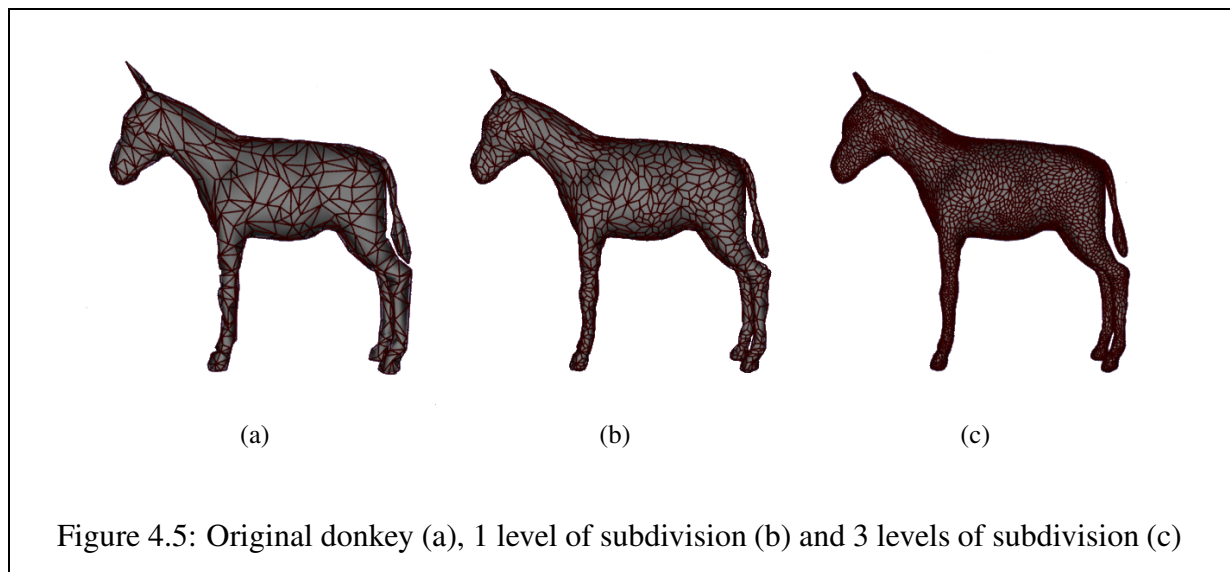
The  $\sqrt{2}$  subdivision scheme inserts a center point into each face, which is then used to create new quadrilateral faces. Edges are used to form the new faces by connecting each of their end points to the face points of adjacent faces. The resulting mesh contains quadrilaterals, although triangles can be introduced into the mesh on boundaries during adaptive subdivision.

Adaptive subdivision takes as input a set of faces that need to be subdivided, a number dictating how many levels of refinement need to be applied and a smoothing operator.

The input set of faces are known as separable faces [Li et al., 2004]. Before subdividing the selection, the adaptive subdivision method searches through all of the faces in the model, marking the faces that are separable and those that are inseparable. Inseparable faces are not subdivided. If a separable face has an inseparable face for a neighbour, triangles are introduced into the model. This is because the vertex that should have been located in the center of the inseparable face now lies on the edge between the separable and inseparable faces.

Figure 4.5 illustrates the application of subdivision to a model of a donkey.

Subdivision is responsible for adding extra detail to a model, by adding new vertices, edges and faces. These items may be displaced to shape a model, or they may be used to aid in the selection of a set of faces.



Subdivision is also responsible for smoothing a model. During each subdivision step, the original vertices in a selection are displaced inwards towards the center of the face they make up. New vertices are added to the center of each face, and the center points of adjacent faces are connected to the vertices of the edge in common. This also has a smoothing affect on a model, as the sharp edges of models are removed, and new ones created. The continuous refinement of a model results in the size of the faces of a model decreasing, as each face is subdivided to create new ones. The overall effect is that of a smooth model, or a model without jagged edges.

The smoothness operator is implemented as an interpolation. The new value for a subdivision displacement is calculated as normal, with the operator determining how far between the original point and the newly calculated point the vertex is displaced. Figure 4.6 illustrates the effect of varying levels of subdivision on a cube with the smoothing operator set to 1.0, 0.5, and 0.0 respectively. Note how the original vertices of the cube remain in place when the smoothing factor is set to 0.0, but the rest of the cube is pulled inwards. This is due to the subdivision scheme used, and the manner in which subdivision is performed. The center (or face) vertices inserted at each step move inwards, the smoothing operator restricts the movement of the original vertices, and confines them to their original positions.

#### 4.3.1.5 Stitching

The stitching tool takes as input two models and a selection from each model that specifies where the join is to be made. The selection of the first model indicates what is to be replaced by

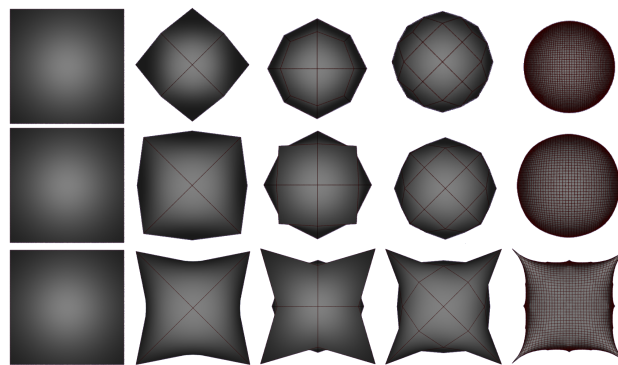


Figure 4.6: Subdivision with a smoothing factor of 1.0 (top), 0.5 (middle) and 0.0 (bottom)



(a) Triceratops Selection (b) Donkey Selection

Figure 4.7: The selection from each model is shown in yellow

a portion of the second model. The selection of the second model denotes what is to be replaced by the selection from the first model.

As an example of stitching, we are going to combine a triceratops model and a donkey model. In this example, the triceratops is the first model and the donkey is the second model. The selection from the triceratops model is going to replace the selection from the donkey model. Figure 4.7a) shows the triceratops selection, and Figure 4.7b) shows the donkey selection. Note that both selections denote the top of the models. The final, stitched model is shown in Figure 4.9.

The selections provided as input parameters are used to create boundary contours. The contours are created by traversing the faces that lie on the boundaries between selected and non-selected faces. After obtaining these contours, the faces in the first model that are not part of the

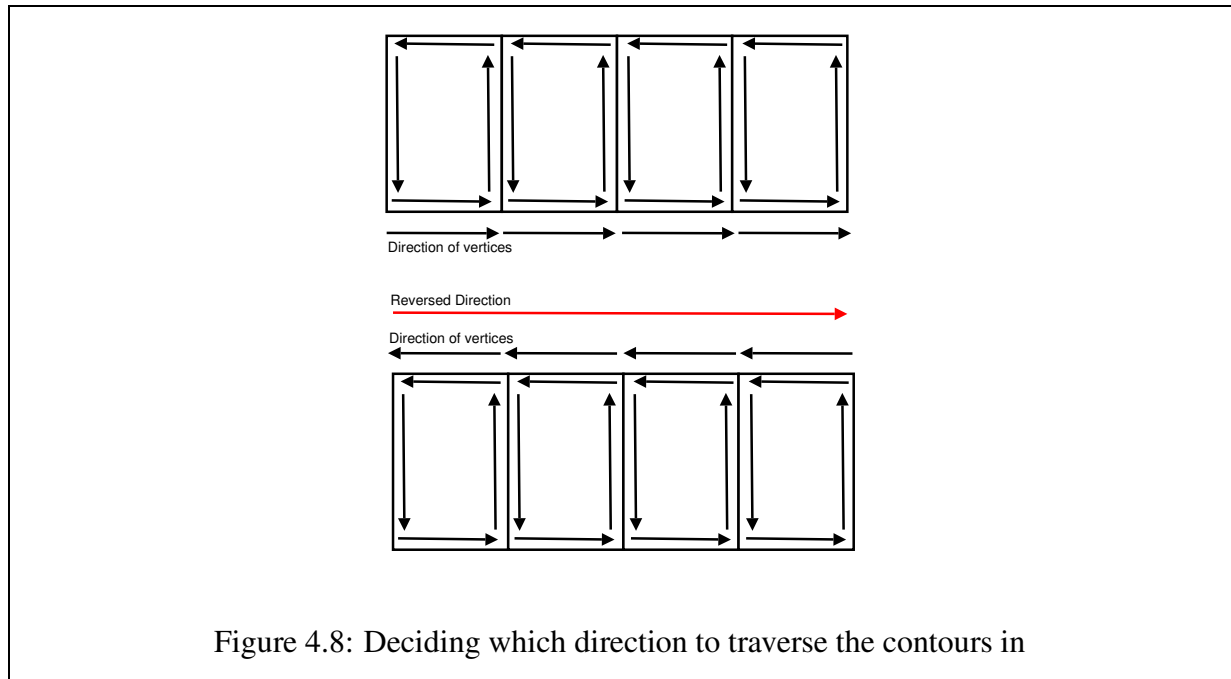


Figure 4.8: Deciding which direction to traverse the contours in

selection are deleted, as the faces that are not selected are going to be replaced. The selection of the second model deleted, as the selection denotes what is to be replaced in the second model.

Once the face deletions have been performed, a brute force search is done to locate the two closest (one from each model) vertices.

Once we have obtained the starting vertices from each model, we need to determine which order each of the contours needs to be traversed in. A possible strategy is discussed, but is unimplemented.

The direction of traversal for the first model can be fixed by specifying the next neighbour on the current edge. For the second model, we obtain the neighbour vertices on each side of the starting vertex, and through the use of the connectivity information stored in each face, we decide which neighbour is the next to be visited. Each face stores its vertices in an anti-clockwise order. The traversal of each contour must be in the same direction as the traversal of the other contour. If this is not the case, the stitching goes awry and connects the vertices inside out. Figure 4.8 illustrates the determination of the direction of traversal for the second model.

Once the direction of traversal has been determined, we visit every vertex in each contour, and connect it to the closest vertex in the other model. This often results in a vertex being chosen more than once, as it is deemed to be the nearest vertex.

Figure 4.9 illustrates the effectiveness of our stitching method. The red band on the model

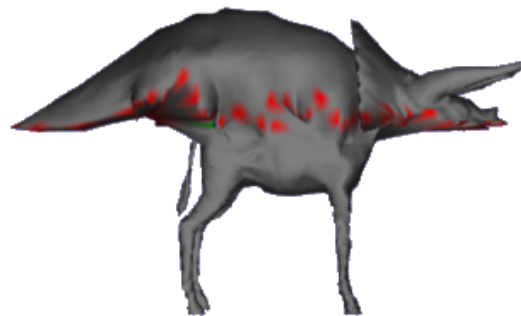


Figure 4.9: An example of stitching two models together

shows where the join took place on each model.

### 4.3.2 Data Structures

This section discusses the implementation of our modeling data structures. The first data structure to be discussed is the model representation. The last data structure to be discussed is the selection representation.

#### 4.3.2.1 Model Representation

Models are represented by mesh structures. A mesh is made up of three essential components, namely edge, vertex and face lists. Each of the procedural modeling tools that are implemented within the mesh class are charged with the task of leaving each of these components in a consistent state. Whenever a new vertex, face or edge is added, each of the relevant parts of the mesh need to be updated.

We also need to be able to query this representation to retrieve information that is pertinent to modeling, but this needs to be done in a manner that does not involve knowing the low level details of the model representation.

#### Faces

A face stores its vertex and edge lists in Java vectors. The reason for using vectors is that we cannot guarantee that a model always has three or four vertices and three or four edges per face.



If we use arrays, or another structure that is not able to grow dynamically, we end up with a restrictive mesh representation. If, for example, a model is made up of octahedral faces and not triangles or quadrilaterals, our mesh structure is not able to cope with it. If we use a data structure that grows as required, we do not have a problem with this kind of mesh.

A face also stores a normal vector that specifies the direction of the normal. This vector is obtained by calculating the normals of each of the vertices in the face and adding them.

A face has a vector of labels which are used to perform selections. Whenever a face forms part of a new selection, the selection identifier is added to the label vector of the face. This vector of labels is generalisable to become a list of attributes. So far there has been no need to use anything other than labels for identification.

### **Vertices**

Each vertex maintains a list of faces and edges of which it is a member. These lists are implemented as Java vectors for the same reasons stated previously.

A point, which stores the position of the vertex in space is also maintained by each vertex. A normal vector is stored by each vertex. The normal of the vertex contributes to the normals of the faces of which the vertex is a component.

### **Edges**

Each edge keeps track of its starting and ending vertex. By doing this, connectivity of the mesh is maintained even if the vertices are displaced.

Each edge maintains a vector of the faces of which it is a member.

### **Mesh**

The mesh class maintains lists of vertices, edges and faces which it is made up of. Each of these lists is implemented as a Java vector.

#### **4.3.2.2 Selection Representation**

To store our selections, we have created a selection representation. This structure is used to store the faces identified during the selection process. Each of procedural modeling tools has been designed and implemented to work with the selection representation.

Although this structure is used to store faces, we are able to retrieve the vertices and edges associated with each face by using the connectivity information maintained by each face. The

implementation of additional selection facilities that are able to select individual vertices and edges is desirable, but has not yet been implemented. We have had no need to select individual vertices or edges thus far. The curve shaping tool uses the connectivity information stored in each face, in the input selection, to retrieve individual vertices for displacement.

Each selection representation stores a list of faces in a Java vector. Once again, vectors are used because of their dynamic nature, the size of the structure grows as more elements are added. A structure such as an array is unsuitable as the size of the selection is unknown until the end of the actual selection process.

Each selection representation also stores an identifier in the form of a string. This identifier is used to retrieve the stored selections from the mesh. The identifier of the parent selection is also stored in the selection representation. This enables a selection to trace its lineage back to the starting primitive.

## **4.4 Conclusions**

This chapter has discussed the motivation, design and implementation of each of our procedural modeling tools and data structures.

The remainder of this section discusses the challenges we faced while choosing, designing and implementing our tools and data structures, and concludes with the contributions we have made in terms of procedural modeling.

### **4.4.1 Summary**

This chapter presents and discusses the design and implementation of our non-interactive modeling tools. The tool-set consists of a selection tool, an extrusion tool, a curve shaping tool, a subdivision tool and a stitching tool.

- The selection tool uses regions of space to identify faces for manipulation. The boundaries of these regions are demarcated through the use of bounding boxes.
- Extrusion is a tool commonly found in 3D modeling packages, which has been adapted for use in a non-interactive modeling environment. This tool is used to extend a model by adding detail.
- The curve shaping tool is based on a modeling strategy as opposed to an existing modeling tool. It is used to shape a model according to prespecified curves.

- The subdivision tool is another commonly found tool, which is used to add detail and smooth a model. The level of smoothness is controlled by a smoothing factor.
- Stitching tools are frequently found in 3D modeling packages and are used to combine geometry to create new models.

Two representations have also been discussed. We have chosen a polygonal mesh representation for models.

The selection representation is used to store the faces identified during a selection operation. A label is associated with each set of faces.

#### 4.4.2 Conclusions

The selection mechanism we have created is scale independent, as it uses regions of space to identify faces. Each selection is axis-aligned and propagates the labels associated with it. Hierarchical modeling is supported by our selection mechanism as sub selections can be made.

All of the other non-interactive modeling tools rely on the use of selections to perform their tasks.

The extrusion tool is an adaptation of a manual modeling tool that preserves mesh integrity.

The curve shaping tool is level of detail independent. The more detail, however, the closer the approximation to the curve.

The subdivision tool provides a means of creating similarly sized polygons over the surface of an object, as faces are subdivided into smaller faces.

The stitching tool is a non-interactive version of a similar tool found in 3D modeling packages.

Code listings of selected tools, namely selection, curve shaping and extrusion may be found in Appendix D.

# Chapter 5

## Applications of Procedural Modeling

Chapter 2.1.3 discusses three broad categories of models, namely organic objects, man-made objects and phenomena. This chapter evaluates our procedural modeling approach by creating models from these categories.

Models from two categories are created, these categories are the organic and man-made objects. We have chosen not to model phenomena. An example from each category is used to illustrate the application of our procedural modeling environment to models. Before discussing the models, we take an in-depth look at each of the procedural modeling tools available in our modeling environment. The discussion focuses on the uses of a tool, its limitations and an example of the usage of the tool.

Section 5.2 discusses the creation of our organic and man-made models. We model a human to illustrate the organic category of modeling, and furniture to show the man-made category. Each model is discussed in the form of an experiment, where we detail the requirements of the experiment, the design strategy used to solve for each of these requirements and the implementation of each of our design strategies.

This chapter concludes with a discussion on the challenges faced during the modeling processes, the strengths and weaknesses of our procedural modeling strategy and the contributions made through the use of our strategy. We also discuss the experiences gained during the model creation process.

### 5.1 Procedural Modeling Tools

Chapter 4 discusses the design and implementation of each modeling tool in detail. This discussion focuses on the uses of each tool, how the tools are applied and the shortcomings of each tool.

The tools being discussed are selection, extrusion, curve shaping tool, subdivision and stitching.

### 5.1.1 Selection

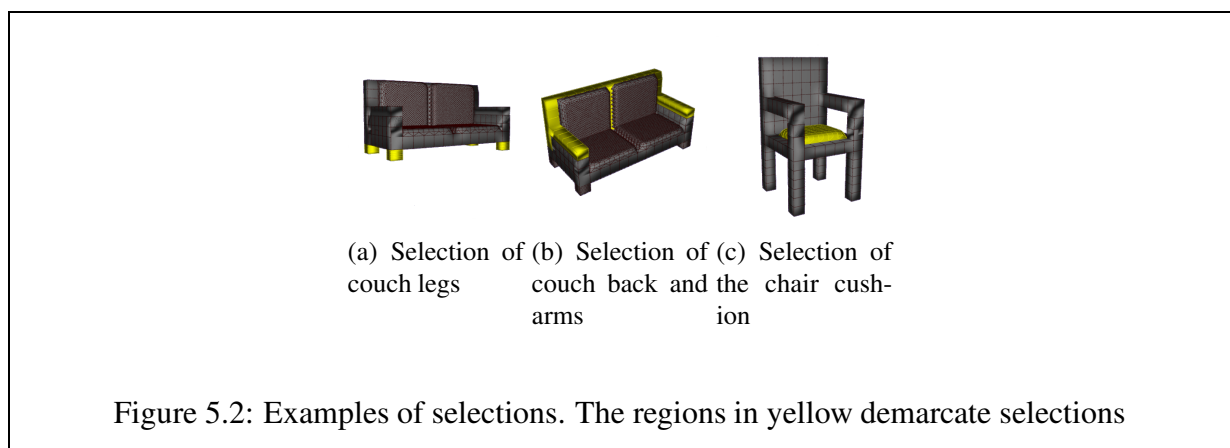
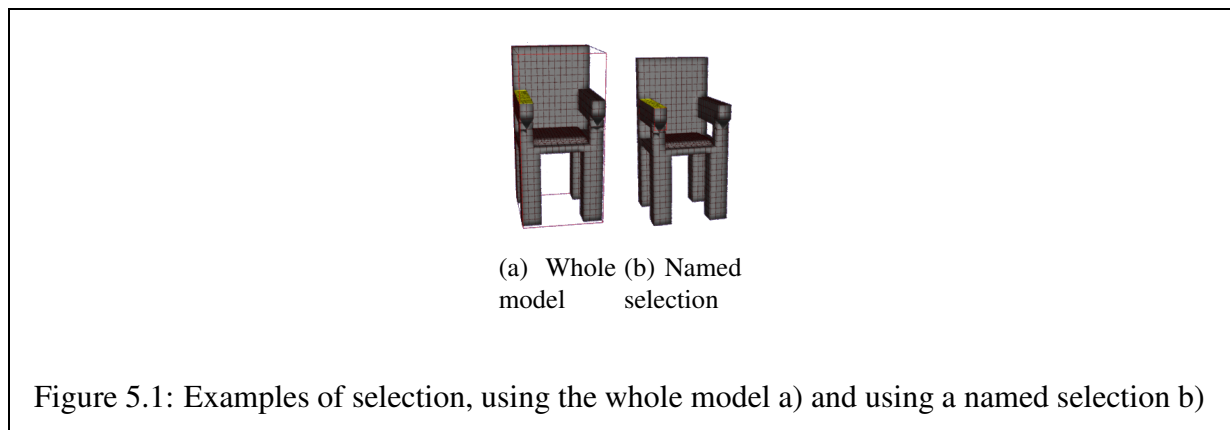
The selection tool is used most often and is also the most important tool in our set of tools as it is used to identify faces for manipulation. Each of the other modeling tools is adapted to use selections either as input or during processing. The parameters used to perform selections are scale independent.

Selection is responsible for the identification and storage of a set of faces. Two types of selection are available namely *anonymous* selections and *named* selections.

Anonymous selections are used during the model generation process to temporarily store selections that do not need to be retained by the model for later use. An example of anonymous selection usage is during an extrude operation. The selection is used to identify which faces for extrusion, but do not have to be retained once the process is finished.

Named selections are stored in by the model for later retrieval and use. Named selections make hierarchical modeling possible, as individual regions of a model can be identified and stored. This is a useful facility for two reasons.

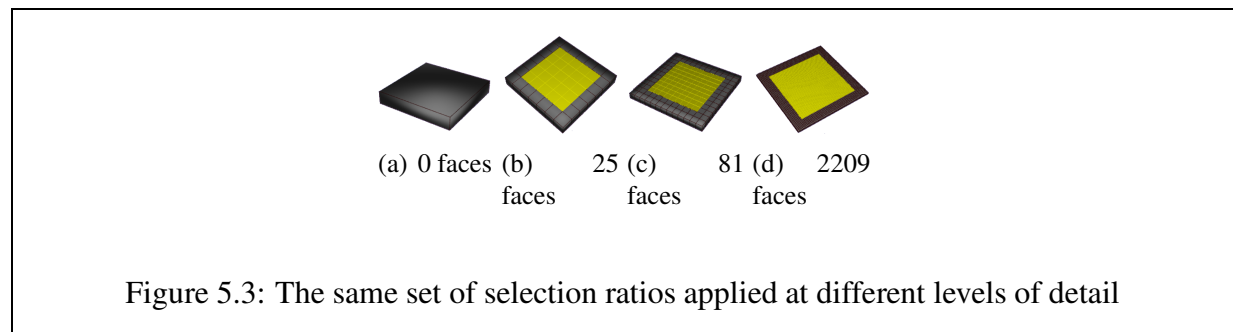
1. The use of named selections makes subsequent selections easier. All selections use bounding volumes to identify member faces. If a selection is stored, new selections can be based on them as opposed to performing selections from the whole model. As a model grows in size, so too does its bounding volume making it necessary to adapt the ratios used for selection. Named selections solve this, as new selections are performed based on existing ones. Figure 5.1 a shows a selection that was performed on the entire chair model. The parameter values have to be varied to compensate for the size and height of the model. The exact parameters used are ( $xMin = 0, xMax = 0.3, yMin = 0.7, yMax = 0.8, zMin = 0, zMax = 1$ ). Figure 5.1 b is the selection of the same faces, but using a named selection that is stored in the model. The parameters used are ( $xMin = 0, xMax = 1, yMin = 0.9, yMax = 1, zMin = 0, zMax = 1$ ). Fewer parameter values have to be changed to achieve the same selection.
2. Hierarchical models are commonly used for animation. The use of named selections makes it easy to insert animation primitives into a model. An animation primitive is associated with a set of vertices, which are displaced as the primitive moves. By using named selections, it is simpler to associate vertices with an animation primitive.



The use of shapes with sharp features is preferable, as our selection mechanism uses bounding volumes to perform selections. Axis-aligned bounding boxes are used for selections. Using shapes that are not smooth makes the selection more accurate as the shape coincides with the shape of the bounding volume. Smooth shapes can be used for modeling, the selection parameters need to be updated to cater for the curve of the object. Section 5.1.3 discusses a strategy that alleviates this problem.

The selection tool is used frequently during modeling. Some examples of selection are shown in Figure 5.2 a, b and c.

A limitation of the selection tool is the level of detail of a model. The less detail a model has the smaller the number of faces that are selected. A face is selected if each of its vertices fall within the region specified by the selection parameter values. It is possible to create an object that is comprised completely of large faces. This makes the selection of faces difficult. As the level of detail of the model increases, the easier it becomes for the selection tool to perform



selections. Figure 5.3 illustrates this.

### 5.1.2 Extrusion

The extrusion tool is frequently used to model objects, both in manual and procedural modeling contexts. Extrusion is used to *grow* a model by adding vertices, edges and faces to it. The box modeling strategy (see Section 2.1.1) uses extrusion to create a model from a primitive object. Our procedural modeling strategy uses the same technique.

The application of the extrusion tool is done algorithmically, and is applied to selected regions of a model (see Section 5.1.1). The modeler specifies the direction and magnitude of the extrusion by using the predefined direction vectors.

As an illustration of extrusion, we show the creation of a chair from its primitive form to the final product. Figure 5.4 depicts how a chair model is created from a cube.

The initial cube is extruded to in both the x and z directions to form a base for the chair. A chair back is extruded in the positive y direction. A cushion is extruded in the positive y direction from the centre faces of the base. The legs are extruded from the base of the chair in a negative y direction.

### 5.1.3 Curve Shaping Tool

The curve shaping tool is used to shape a model. Selections of a model are deformed to follow a curve, thus shaping the model. The use of this tool alleviates the need to select vertices, edges or faces and move them individually.

Whilst experimenting with the curve shaping tool, we found that this tool has other applications. It can be used in conjunction with extrusion. In instances when the shape being worked with does not have pronounced edges, the curve shaping tool can be used to make curved selections sharper.

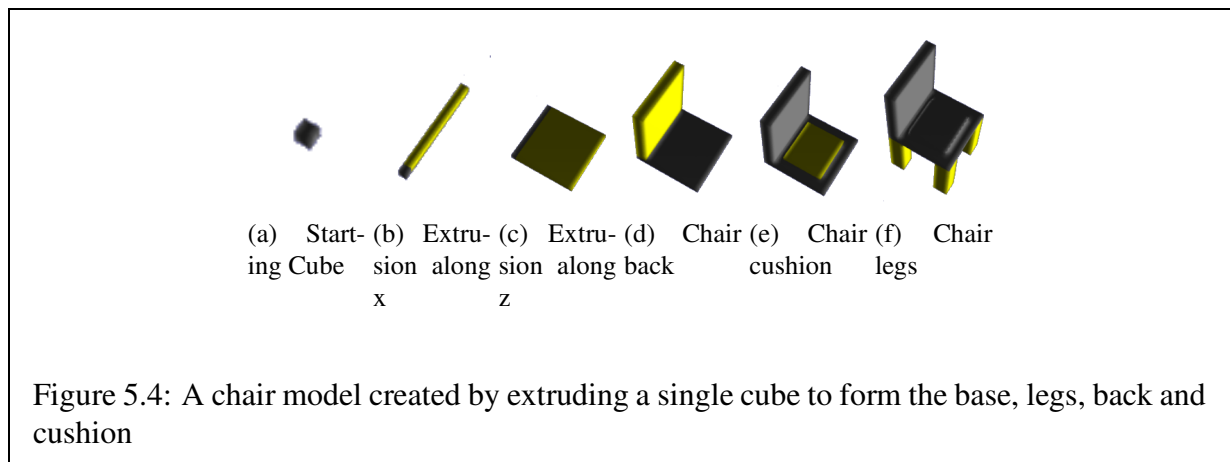


Figure 5.4: A chair model created by extruding a single cube to form the base, legs, back and cushion

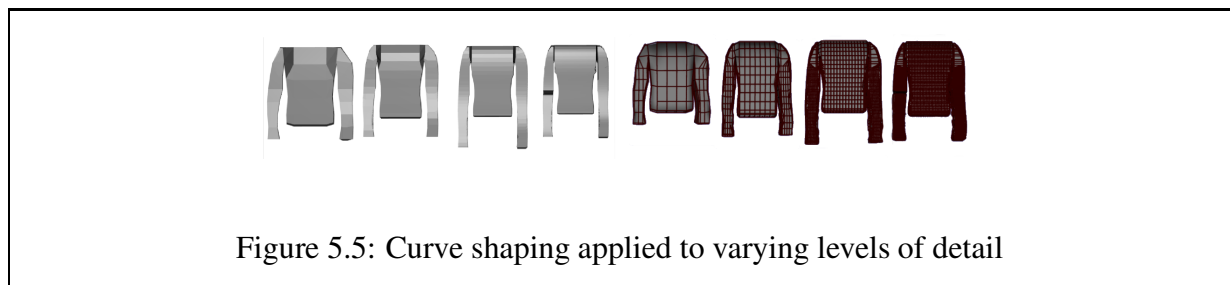


Figure 5.5: Curve shaping applied to varying levels of detail

The curve shaping tool is level of detail independent. The more detail available in a model, the better the fit to a curve. Figure 5.5 illustrates how the approximation to a curve improves as the level of detail of an object increases.

The first model has a small amount of detail, and the approximation to the curve is jagged. The next two models have more detail, but are still slightly jagged, although the approximation is improving. The final model has a large amount of detail and is a good approximation to the curves used for shaping, resulting in a smooth model.

#### 5.1.4 Subdivision

The subdivision tool serves two purposes, the addition of detail and smoothing of a model. The addition of detail to a model comes in the form of vertices, edges and faces. The subdivision tool has been adapted to work with sets of faces and functions on both open and closed meshes. The smoothing aspect of subdivision is controlled by a smoothing factor, which is set by the modeler. The use of this smoothing factor determines how much smoothing occurs during subdivision. In some instances it is desirable to subdivide a set of faces without applying any smoothing. The



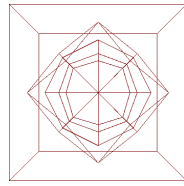


Figure 5.6: Different levels of subdivision resulting in a reduction of model size and different degrees of model smoothness

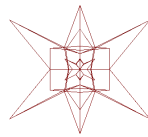


Figure 5.7: An example of using values outside the range  $[0; 1]$  for the smoothness operator

application of subdivision in these scenarios is to create extra detail for manipulation or selection.

Figure 5.6 shows a cube at varying levels of subdivision. The cubes have been embedded within each other to illustrate the smoothing that takes place, and the reduction in model size due to smoothing. The outer cube has no subdivision applied to it. The next cube has one level of subdivision applied to it, and the innermost cube has two levels of subdivision applied to it.

Figure 5.7 shows an example of a star-shaped object (2 levels of subdivision with smoothness set to 2.0) contained within a cube (no subdivision) contained within a star-shaped object (2 levels of subdivision with smoothness set to -2.0).

The subdivision tool requires a manifold surface on which to operate.

### 5.1.5 Stitching

The stitching tool is used to combine components of objects, or entire objects, to create new models. This strategy promotes re-usability, as we are able to re-use pieces of previously created objects.

The constructivist modeling strategy relies on the use of a stitching tool. We can also use stitching in an extrusionist modeling context. This strategy relies on the modeler creating a model



Figure 5.8: An example of stitching

using extrusion, but stitching together components of the model to create a closed structure. Figure 5.8 shows a couch model that has had stitching applied to it. An arm has been extruded from the back and base of the couch respectively. Stitching has been used to join these two arms to create a single arm on each side of the couch.

Figure 5.8 illustrates that stitching can be applied to a single model to join components of that model. The stitching tool is designed to make use of two separate models, which makes the stitching of a model to itself problematic. The stitching process itself is successful, but anomalies are visible in the model. The anomalies are due to the way in which sections of the model are chosen for combination.

### 5.1.6 Models

This discussion looks at the properties of each type of model discussed in Section 5.1 and how our procedural modeling tools are capable of meeting these requirements.

#### 5.1.6.1 Organic Objects

Organic objects depict naturally occurring objects, these include humans, flora, fauna and sea life. The shapes of these objects lack angular definition, and have smooth edges.

By using a combination of our tools, we are able to create organic shapes.

The selection tool is used to select parts of the model to which new vertices, faces and edges are added and the parts of the model which are deformed to shape the model.

The extrusion tool creates angular models, unless the initial selection is smooth. The displacement specified by the modeler is applied to each vertex within a selection, thus retaining

the start shape. The extrusion tool does, however, result in straight edges that do not appear organic in nature. Again, through the use of the curve shaping tool, the faces created using extrusion can be displaced to create organic models.

The curve selection tool is primarily concerned with shaping. A selection is deformed to follow a curve. A curve may be a straight line, in which case, the deformation is angular. If a non-linear curve is used, the resulting deformation is organic in nature, as there are no sharp edges in the model.

Subdivision is an ideal tool to create organic models. By setting the smoothing factor to its maximum value, smoothing of a model occurs. The smoothing process rounds off the edges of a model, creating an organic-looking object.

The stitching tool can also be used to create organic models. Through the use of initial organic models, we are able to retain the organic properties of the original models. If we start the modeling process with sharp-edged models, we can use the stitching tool to create a coarse model, and then apply either subdivision or the curve shaping tool depending on the requirements of the model.

#### **5.1.6.2 Man-Made Objects**

Man-made objects do not occur naturally and are created through human effort. Examples include buildings, automobiles and furniture. These objects tend to be angular in shape, with definite edges. Some of our procedural modeling tools support the creation of angular shapes without needing to rely on other tools.

The selection tool is used to identify parts of the model for manipulation. The results of applying the selection tool to an angular model is more predictable than the application of selection on a curved object. This is ideal for modeling man-made objects, as they are angular in nature.

The extrusion tool results in angular objects, unless the initial selection is non-angular. The resulting faces have sharp edges, the exception to this are the cap faces of the extrusion.

The curve shaping tool is used to deform an object to have sharp edges and creases. By shaping this object to a line, angular models are created. Another aspect to consider is the level of detail of the model. If a model has a low level of detail, the result of applying the curve shaping tool is very angular and jagged.

The subdivision tool is more suited to the smoothing of an object to create organic shapes. We are able to use subdivision to model man-made objects through the use of our smoothing factor. If we set the smoothing factor to the lowest possible factor, we obtain an object that is only slightly rounded off.

The stitching tool creates man-made models by combining angular models. If an initial model is organic, we can apply the curve shaping tool to create edges.

## **5.2 Model Creation**

Section 5.1 discusses how the procedural modeling tools are used. This section focuses on the application of these tools in modeling scenarios. The aspects of model creation that are focused on are the tools used, how each tool is applied and the procedural modeling language features being used by the model. We also take a look at which of the procedural modeling benefits (see Chapter 1) are apparent in these models. The algorithms to create each object are supplied to illustrate the model creation process.

We begin by discussing the creation of hierarchical models. All of the models discussed in this chapter are created and stored hierarchically by using named selections. The first model we discuss is the organic object. Our choice of organic object is the human form, and the experiment to create the human model is discussed in Section 5.2.2. The second type of model discussed is the man-made object. We have chosen to model furniture in the form of the luggage, chairs, tables and couches. Each of these models is discussed in detail in Section 5.2.3.

### **5.2.1 Hierarchical Models**

Our procedural modeling strategy supports the creation of hierarchical models. Our modeling strategy primarily uses the extrusionist modeling approach. A model is generated from a single primitive, which is extended and deformed to match the shape of the desired object. The hierarchical nature of the model generation process is due to the way in which the model is created. The modeling process starts with a simple primitive, which is gradually extended to create larger components of the model. These larger components serve as parents to the sub-components which are created. For example, if we are modeling a human figure we start with a base shape, and create the torso. Selections from the torso form the arms, legs and head respectively. Selections from these components are in turn used to create hands, feet and ears.

### **5.2.2 Human Modeling**

One of the most prominent organic shapes found in nature is the human form. To illustrate the applicability of our modeling approach to organic shapes, we have chosen to model a human.

### 5.2.2.1 Requirements

The human model is based on an existing manual human modeling technique. Saastamoinen [1999] uses a box modeling strategy to obtain the basic shape of the human model. The shaping is done by hand to follow the curves depicted on a template image which is loaded into the background of a modeling package.

This human modeling strategy is adapted to a procedural modeling context. To do this, the procedural modeling tools are used in the same way as the tools in the manual modeling strategy. This provides us with a comparable technique.

The human modeling strategy of Saastamoinen [1999] is discussed in Section 2.1.1. The requirements of our experiment are listed below.

1. Our human modeling strategy should mirror the manual modeling strategy of Saastamoinen [1999], as we are adapting this strategy for use in a procedural modeling environment. The specific modeling requirements include:
  - (a) The specification of a template-like method that depicts the curves required to shape the model. We are unable to use a template image for the deformation of a model. The image template is used in a manual modeling context, as the modeler is responsible for displacing selections to match a curve. Our procedural technique does not rely on this strategy, instead we need to specify accurate curves which may be used for the deformation of our model
  - (b) The selection of a starting primitive or base shape. One of our aims is to create a human model that results in the same basic shape irrespective of the starting primitive.
  - (c) A method of growing this base shape into a coarse, approximatory shape of our desired model. The manual modeling strategy uses a box modeling technique. We need to devise a method of using the box modeling strategy to our advantage.
  - (d) Once we have the coarse model shape, we need to be able to deform the model to match the specified curves. This is the last step in the model creation process.

### 5.2.2.2 Design

This section discusses how each of the human modeling requirements is met.

The first requirement is the creation of a template to shape the model. Since we are using a procedural modeling strategy, the use of a template image to perform the actual shaping is not a viable solution, instead, we have devised a manner in which we can translate a template image

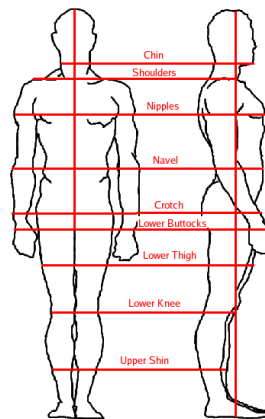


Figure 5.9: The human figure (by Loomis)

into a set of curves. We make use of a proportionally correct human figure, taken from Loomis [1943]. The proportions of the human figure are calculated manually to create a set of curves, which we use to deform the model. These proportions are calculated as a set of ratios based on measurements taken from the sketches. Figure 5.9 shows the front and side views of the sketches used to determine the curves of the human model. Each component of the model, i.e. the torso, arms, legs, neck and head, is dealt with individually.

The second requirement is the creation and placement of an initial starting shape. A parallelepiped is used to begin the modeling process. This is a base-shape independent model, so different starting shapes may be used. We are not restricted to use a cubical shape, we can use other shapes (see Section 6.3).

The manual box modeling strategy discussed in Section 2.1.1 is adapted to create a procedural equivalent. The next requirement relies on the starting primitive being extended to create a coarse approximation of our desired model. We have made use of procedural modeling tools to cater for this aspect of the modeling process.

The penultimate requirement is the shaping of the coarse object to create a shaped model of our human. The curves are already specified in step one of the modeling process, all that remains is the fitting of the model to these curves.

Section 5.2.2.4 discusses the results of our human modeling experiment.

### 5.2.2.3 Implementation

This section discusses the implementation of the human modeling procedure. We discuss the choices made during implementation, and present a set of algorithms to illustrate the discussion.

The first requirement is the definition of a set of curves that are used to shape our model. Section 5.2.2.2 discusses the use of a template image and manually performed calculations to obtain proportions. These proportions are recorded and used as control points for the required curves. We have shaped our models in two dimensions to match the front and side views of our template image. Each component of the human figure, i.e. the arms, legs, torso, head and neck, has four curves associated with it. The proportions are calculated with respect to each individual component of the human model. This mirrors our hierarchical modeling strategy, as we have the capability of dealing with individual components of a model alone. Each curve is created in relation to the corresponding component of the model.

The second requirement is the creation and placement of a base-shape to begin the modeling process. We make use of primitive procedural base-shapes. These shapes are obtained through procedure calls within the human modeling procedure. The placement of the base-shape is irrelevant, as the modeling procedure behaves in the same manner.

To extend our starting primitive, we have used the same procedure as the manual modeling strategy: repeated application of the extrusion tool. Regions of the primitive are selected and then extrude with the purpose of creating a coarse human model.

The entire human model is created before shaping. The coarse model has the correct number of limbs, and is proportionally correct. Once we have this coarse model, we make a series of calls to our curve shaping tool to fit the various components of our model to the required curves.

Algorithm 2 provides one possible torso modeling strategy. It creates a rectangular parallelepiped consisting of many small adjacent cubes occupying a volume with the given dimensions. Extrusion is used to provide the vertices required for later shaping. The size of the basic cube determines the level of detail of the model. If the cube size decreases many more are required to fill the same volume, and more vertices are available during later shaping steps. Other torso modeling strategies could also be used without affecting the working of the rest of the model (section 6.3.2).

Curve shaping is used to refine the shape of the basic body components. The process for the torso is outlined in Algorithm 3. The curves are created dynamically, and proportioned to match the geometry provided. Vertices are displaced to match the desired outline. The fit of the model to a curve depends on the level of detail in the model. Subdivision can be used between geometry creation and curve fitting to increase the level of detail of the final model.

**Algorithm 1** A Human Modeling Algorithm

---

```

HumanModel ( )
    createTorso (0.288, 0.398, 0.611, 6.0)
    headSel←torso.select(17.5%,82.5%,99%,100%,0%,100%)
    head←createHead (headSel)

    leftArmSel←torso.select(99%,100%,0%,100%,0%,100%)
    rightArmSel←torso.select(0%,1%,0%,100%,0%,100%)
    leftArm←createArm (left, leftArmSel)
    rightArm←createArm (right, rightArmSel)

    leftLegSel←torso.select(50%,100%,0%,1%,0%,100%)
    rightLegSel←torso.select(0%,50%,0%,1%,0%,100%)
    leftLeg←createLeg (left, leftLegSel)
    rightLeg←createLeg (right, rightLegSel)

    torsoCurves (torso)
    headCurves (head)
    armCurves (left, leftArm)
    armCurves (right, rightArm)
    legCurves (left, leftLeg)
    legCurves (right, rightLeg)

```

---

To make the foot modeling procedure robust, we have devised an algorithm that creates additional detail as it is required. Algorithm 4 shows the algorithm used to select and create a foot and toes.

The number of toes for each foot is specified as a parameter, as is the threshold level. The threshold value controls how detailed a selection is. For example if the threshold value is set to 10, a selection is valid if it has at least 10 faces. If any selection contains less than the specified number of faces, the region from which the toes are selected is adaptively subdivided. This process is repeated until all of the toes are valid (in terms of the threshold value) selections.

**5.2.2.4 Results**

Figure 5.10 shows a progression of the model creation process. The model starts as a cube, which is extended to create a parallelepiped. The limbs of the human model are extruded from the sides of the parallelepiped to give a coarse approximation of the human figure. The last image shows the result of shaping the coarse model to create the final human model.



---

**Algorithm 2** A Torso Creation Algorithm

---

```

createTorso (Depth, Width, Height, unit)
    Create a cube of unit size

    repeat Depth/unit times
        select front faces of the torso cube
        extrude faces along the forward axis

    repeat Width/unit times
        select right faces of the torso cube
        extrude faces along the right axis

    repeat Height/unit times
        select bottom faces of the torso cube
        extrude faces along the downward axis

```

---



---

**Algorithm 3** Shaping the Torso Model to Curves

---

```

torsoCurves (torsoSelection)
    bv←torsoSelection.BoundingBoxVolume ()
    minMax←bv.minX + bv.maxX

    //Assign Control Points
    CP1←(bv.maxX, bv.maxY, 0.0)
    CP2←(bv.maxX-(0.15*minMax), bv.maxY/2, 0.0)
    CP3←(bv.maxX-(0.1*minMax), bv.minY, 0.0)

    rightCurve←BezierSpline (CP1, CP2, CP3)

    //Re-assign Control Points for the front,
    //back and left curves
    leftCurve←BezierSpline (CP1, CP2, CP3)
    frontCurve←BezierSpline (CP1, CP2, CP3)
    backCurve←BezierSpline (CP1, CP2, CP3)

    shapeToCurve (torsoSelection, rightSideCurve)
    shapeToCurve (torsoSelection, leftSideCurve)
    shapeToCurve (torsoSelection, frontSideCurve)
    shapeToCurve (torsoSelection, backSideCurve)

```

---

**Algorithm 4** A Foot Creation Algorithm

---

```

FootModel (numToes, threshold, initialSelection)
  toeSize ← 1.0/numToes
  allSelectionsValid ← false
  while NOT allSelectionsValid
    anonSelect ← initialSelection.select
                      (0%,100%,0%,40%,75%,100%)
    xMin ← 0.0
    xMax ← toeSize
    valid ← true
    for i from 1 to numToes
      toei ← anonSelect.select
                      (xMin,xMax,0%,100%,0%,100%)
      if toei.numFaces() is less than threshold
        valid ← false
        xMin ← xMax
        xMax ← xMax + toeSize
    if NOT valid
      anonSelect.adaptivelySubdivide ()
    allSelectionsValid ← valid

  for i from 1 to numToes
    extrude toei

```

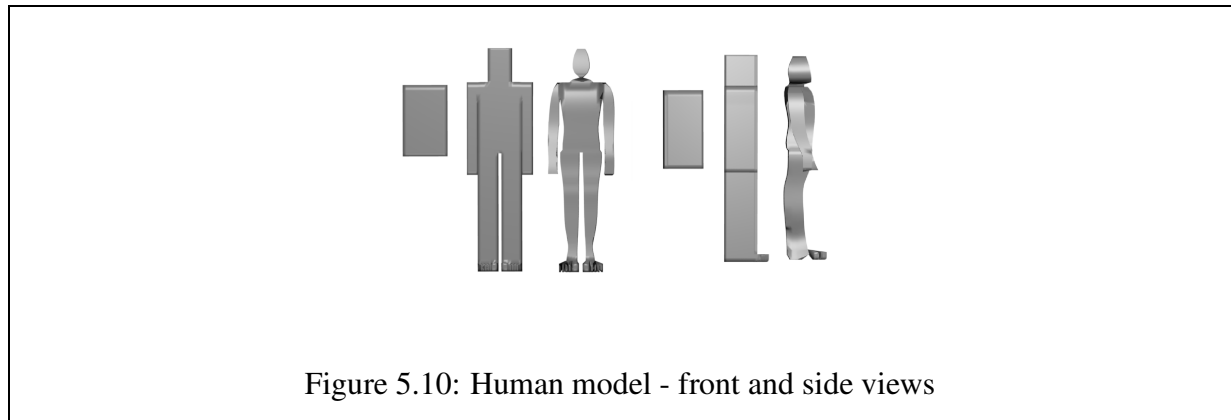
---

The coding of the procedures required to generate the human model takes a significant amount of time. This is due to the proportion calculations that are done by hand. The procedures used to generate the human model are re-usable (see Section 6.3).

An alternative modeling strategy that applies performs curve shaping after the creation of each model component can also be used. The results are sometimes unpredictable, as the shaping affects potential selections. It does, however, have the benefit of retaining some of the curvature of the parent component in the child.

### 5.2.3 Furniture

The realm of man-made objects is extensive and includes buildings, cars and furniture. We have elected to model furniture to illustrate the applicability of our modeling tools to man-made models. A reason for choosing furniture is the abundance of examples encountered in every day life. Basic furniture models can be created in a matter of minutes, with the more complicated



pieces taking more time.

A variety of furniture pieces which include chairs, tables, couches and a model similar to Terry Pratchett’s “The Luggage” [Pratchett, 1985] are created.

#### 5.2.3.1 Requirements

We have created four individual procedures, one for each piece of furniture that we are modeling. Each procedure is required to produce a model that is identifiable as the type of furniture being modeled. For example a couch must resemble a couch and a table a table.

We need to parameterise as many of the values used to create the furniture models as possible. By doing this, we are able to generate distinct models from a single procedure.

Through the use of furniture models, we have shown that our procedural modeling tools are capable of modeling objects that have edges. The processes that we have used are adaptable to create other man-made objects.

The procedures should be parameterised and base-shape independent. This makes the procedure flexible and allows distinct models to be created from a single procedure.

#### 5.2.3.2 Design

The design of the furniture models is based on rough approximations of furniture examples found in everyday life. The box modeling strategy is used to create each of the furniture models.

The luggage model uses the legs and feet of the human model described in 5.2.2.3.

### 5.2.3.3 Implementation

This section discusses the tools and strategies used to create the furniture models. An algorithm for each procedure is presented for discussion.

Each of the models we have created make extensive use of extrusion.

The chair model makes use of a cubic primitive to begin the modeling process. This cube is extruded to create a base for the chair. Once that is done, the back faces are selected to create a chair back. A cushion is created from the faces in the center of the base. Four legs are created by performing selections and extrusion on them. Algorithm 5 illustrates this process in more detail.

The couch model follows a similar strategy to the chair model. A parameter controls the number of cushions created for a couch. The couch model grows in length to accommodate any given number of cushions. Additionally, the modeler can choose to have corresponding back cushions made on the couch. Algorithm 6 shows this process in more detail.

Stitching is used to create arms for both the chairs and couches.

The luggage model makes use of an extruded cube to create a base. Once this is complete, the edges of the luggage are selected to extrude the sides. The back most side is extruded further to create a straight lid. The bounding volume of the luggage is used to create two curves with which to shape the lid. The last step in the creation of the luggage model is the addition of feet. The modeler is able to specify the number of feet that are required. The selections for these feet are made from the base of the luggage, and the human foot procedure is called to grow feet on the luggage. Algorithm 7 illustrates this process in more detail.

The table model provides the modeler with a choice of starting primitives. By setting the value of a boolean parameter, the modeler is able to select either an oval base-shape or a cubic base shape. Once the base shape has been selected, the legs of the table are created in the same manner as the chair and couch. Algorithm 8 shows the table creation process.

### 5.2.3.4 Results

This section discusses the results of the furniture modeling exercise. Figure 5.11 shows a scene that is comprised of chairs, tables and the luggage. Each of these models has been created by using the algorithms in Section 5.2.3.3. The variations in the chairs and tables are due to changes in parameter values.

Although the furniture is not highly refined or intricate in detail, the items of furniture are recognisable. The variation afforded by the parameters of each procedure has saved a large amount of modeling time, as a procedure can be called repeatedly and generate different results.

---

**Algorithm 5** A chair creation algorithm

---

```
createChair (unit, length, depth, backLength, arms,  
            bottomarms, shape)
```

```
Create a cube of unit size  
Repeat length/unit times  
    select faces closest to the maximum x range  
    extrude faces along right axis  
Repeat depth/unit times  
    select faces closest to the maximum z range  
    extrude faces along forward axis  
//create a chair back  
Repeat backlength/unit times  
    select faces closest to the min z range and  
        closest to the max Y range  
    extrude faces along upward axis  
//Extrude the center-most faces of the base  
//to create a cushion,  
Select faces in the center of the chair base  
    extrude faces along upward axis  
    subdivide extruded faces  
for 1 to 4 legs  
    Select faces from base of chair  
    extrude faces along the downward axis  
//create arms from the back of the chair  
if (arms)  
    for 1 to 2 arms  
        select faces from chair back  
        extrude faces in the positive Z-direction  
    if (bottomarms)  
        for 1 to 2 arms  
            select faces from chair base  
            extrude faces along upward axis  
        stitch backarms and bottomarms  
//If the shape option is selected, shape the back of the chair  
if (shape)  
    create curve with offSet displacement  
    select topmost faces of chair back  
    shapeToCurve (faces, curve)
```

---

---

**Algorithm 6** A couch creation algorithm

---

```
createCouch (unit, length, depth, backLength,  
            arms, bottomarms, shape numCushions,  
            backCushions)
```

```
Create a cube of unit size  
for 1 to numCushions  
    Repeat length/unit times  
        select faces closest to the maximum x range  
        extrude faces along right axis  
Repeat depth/unit times  
    select faces closest to the maximum z range  
    extrude faces along forward axis  
Repeat backlength/unit times  
    select faces closest to the min z range  
    and closest to the max Y range  
    extrude faces along the upward axis  
for 1 to numCushions  
    select faces from base  
    extrude faces along upward axis  
if (shape)  
    create curve with offSet displacement  
    select topmost faces of chair back  
    shapeToCurve (faces, curve)  
if (backCushions)  
    for 1 to numCushions  
        select faces from couch back  
        extrude faces along forward axis  
for 1 to 4 legs  
    Select faces from base of couch  
    extrude faces along downward axis  
if (arms)  
    for 1 to 2 arms  
        select faces from couch back  
        extrude faces along forward axis  
if (bottomarms)  
    for 1 to 2 arms  
        select faces from couch base  
        extrude faces along upward axis  
stitch backarms and bottomarms
```

---

**Algorithm 7** A luggage creation algorithm

---

```

createLuggage (unit, length, depth, height,
               lidLength, offSet, numFeet)

    Create a cube of unit size
    Repeat length/unit times
        select faces closest to the maximum x range
        extrude faces along the left axis
    //Extrude along the Z axis next
    Repeat depth/unit times
        select faces closest to the maximum z range
        extrude faces along the backward axis
    for 1 to 4 trunkSides
        Select faces from base of trunk
        extrude faces along the upward axis
    Repeat lidLength/unit times
        select faces closest to the min z range
            and closest to the max Y range
        extrude faces along the upward axis

    Create inner and outer lid curves using the
    bounding volume of the trunk and offSet

    //Shape the lid
    select outside faces of lid
    shapeToCurve (faces, outerCurve)
    select inside faces of lid
    shapeToCurve (faces, innerCurve)
    for 1 to numFeet
        Select faces from base of trunk
        createFoot (faces)

```

---



Figure 5.11: A scene containing chairs, tables and the luggage

---

**Algorithm 8** A table creation algorithm

---

```
createTable (roundTable, length, width, unit, smoothingFactor)

  if (roundTable)
    Create elongated cube with length and width
    Subdivide cube (smoothingFactor)
  else
    Create a cube of unit size
    Repeat length/unit times
      select faces closest to the maximum x range
      extrude faces along right axisn
    Repeat width/unit times
      select faces closest to the minimum z range
      extrude faces along backward axis
  for 1 to 4 legs
    Select faces from base of couch
    extrude faces along the downward axis
```

---

### 5.3 Discussion

Sections 5.1 and 5.2 discuss the application of the procedural modeling tools and two modeling experiments, respectively. This section discusses the experience gained and modeling strategies used during the experiments. This section also discusses the findings, the challenges we faced and the contributions made in terms of our modeling tools and our modeling strategy.

Whilst using our modeling tools to create our human and furniture models, we encountered a few difficulties with the geometry that we applied our tools to. We overcame these problems by applying these strategies.

- In instances when a selection needs to be performed on a curved surface, we can use the curve shaping tool to “flatten” the area of interest prior to performing the selection. We use a straight line, which is at the maximum axis range of the potential selection. A larger than required area of geometry is selected, and then fitted to the curve to make the geometry a uniform shape. The selection then proceeds as normal. This strategy can also be used as an intermediate step during extrusion operations. Alternatively, our extrusion tool is capable of returning the capping faces of the last extrusion performed. This alleviates the need of applying curve shaping prior to a selection, and the need to perform a selection altogether. If we already have the faces which require extruding, there is no need to select them again.



- Another valuable lesson learned is the use of bounding volumes. Bounding volumes are used to perform selections, but in addition to this purpose, bounding volumes can be used to calculate curves to which a model may be fitted. All of the curves used in our modeling experiments are calculated according to individual selections and their bounding volumes. For simple curves that need to span the axes, the use of bounding volumes are ideal.
- The use of adaptive subdivision has also proved essential during the modeling process. Situations have occurred in which we require additional detail in a specific region of a model during a selection. In some instances, selections need to be made from models that contain little detail. We can remedy this lack of detail by repeatedly applying the subdivision tool to the region of interest. This strategy is used to model the feet of the human and the luggage models (see Algorithm 4). A region of interest is identified at the front of the foot, and selections are made to identify a modeler-specified number of toes. If the quantity of faces of any of these selections is below a threshold value, the region of interest is subdivided and the process is repeated until each selection is valid.
- The use of our hierarchical modeling facility has also proved invaluable. By modeling an object in a hierarchical manner, we are able to refer to the stored selections at later stages of the model creation process. This functionality was used extensively during the human modeling experiment.

In terms of experimental results, we are able to conclude that our modeling tools provide an effective method of generating human and furniture models. As stated in Section 5.2, we are interested in the proof-of-concept aspect of our experiments. The human model is proportional, as it is based on accurate calculations. Details such as hands, eyes, ears and nose can be added to the models in the same manner as the models were created.

During the human modeling experiment, we discovered the benefits of using a hierarchical modeling strategy. The size of the bounding volume of a model increases as the object is grown through extrusion or displacement. This has an adverse affect on selections, as we have to adjust the selection scales continuously to accommodate this growth. By using named selections, and thus a hierarchical strategy, we no longer have to worry about this. Each sub-selection can be performed in terms of the named selection that contains the region of interest. The bounding volumes tend to be a smaller in size, which alleviates the need to continually adjust our selection parameters.

The stitching tool is first used during the furniture modeling experiments. The tool is effective in terms of stitching models together. It does, however, have a problem when the two models it is

stitching together are components of the same model. The stitching process works, but artifacts of the join remain. This problem might be resolved if the two components of the model are made into individual models that are then stitched together.

The contributions we have made during our experiments are two-fold.

- We have illustrated how our tools may be applied to create believable models. The script writing process, in most cases, takes a matter of minutes, more complex models require more time. Once a script has been created, it can be used repeatedly to recreate models, and to generate new models. All of our scripts are parameterised, which allows variations of models to be generated simply by changing a parameter value.
- We have documented modeling strategies that were beneficial to us during our experiments. Although these techniques are readily available, it is through a significant amount of time spent working with the modeling tools that we have identified these approaches. Each of these strategies has been used while generating the models in this chapter.

A colour image of the human and luggage models may be seen in Appendix B. Examples of the modeling scripts used to generate some of these models are available in Appendix C.

# Chapter 6

## Results

Chapter 5 discusses two applications of procedural modeling, human modeling and furniture modeling. This chapter focuses on the benefits of procedural modeling discussed in Chapter 1. Six experiments have been devised to show that our modeling strategy provides the benefits of procedural modeling.

The models that are used for illustrative purposes are the models discussed in Chapter ??, with the exception of the L-system trees.

The first experiment discusses the benefit of parameterisation. All of our models are parameterised to enable us to re-use the modeling procedures. The experiment focuses on the variety we are able to attain through the use of parameters. We also show the benefit of specifying default parameter values.

The second experiment tests the benefits of re-usability. The parameterisation experiment explores an aspect of re-usability by creating a variety of models from the same procedure. This experiment focuses on the application of procedures to different models in different contexts.

The third experiment investigates base-shape independence. This experiment is related to the re-usability experiment, as we are re-using our procedures. The focus of this experiment, however, is the application of procedures to different starting primitives.

The fourth experiment tests the increased level of complexity attainable in procedural models. One of the benefits of procedural modeling is the ability to create complex models. Procedural modeling creates complexity by adding detail that is tedious to create in models manually.

The fifth experiment tests the benefit of pseudo-randomness in procedural modeling. This experiment compares the use of pseudo-random numbers to ordinary random numbers. An illustration of the effects of each are shown.

The sixth experiment quantifies the benefits of using a cache during the modeling process.

## **6.1 Experiment 1 - Parameterisation**

The use of parameters in a procedural modeling context provides a control mechanism over the model generation process. Variations of a model are achieved by altering parameter values. By forcing procedures to make use of default parameter values, the model generation process becomes easier, as parameter values do not have to be set. Each parameter that is not assigned a value makes use of its default value.

The purpose of this experiment is to explore the benefits of parameterisation. This includes the ability to generate a number of unique models from a single procedure, the ability of the modeling procedure to cope with changes to parameter values and the benefit of having default parameter values.

### **6.1.1 Objective**

Two hypotheses are tested in this experiment.

1. The first hypothesis is that a number of distinct models can be generated from a single procedure by changing parameter values.
2. The other hypothesis is that the modeling procedures used for this experiment are robust enough to handle a range of parameter values and produce the expected result.

Another aspect which is under consideration is the use of default parameters and the effects they have on model generation. This result is not quantifiable, but is a relevant point.

### **6.1.2 Design**

To test these hypotheses, we make use of the human modeling procedure described in Section 5.2.2.

A number of human models are generated from this human modeling procedure. The height, abdomen and buttocks parameter values are assigned random values to create unique models. The abdomen and buttocks parameters are used during the model shaping process to displace the shaping curves around the abdomen and buttocks regions of the model.

An advantage of using the human model, especially in terms of our second hypothesis, is that each resulting model is guaranteed to be proportional. The proportions of the human model are fixed and scale according to the height value provided. The curves used to shape the human model are based on these proportions and are created by using the bounding volumes of the

current model. As the height value increases or decreases so do the components of the human model, which in turn enlarges or reduces the bounding volumes of the model and its components. As the size of the bounding volumes vary so do the shapes and lengths of the shaping curves.

### 6.1.3 Implementation

The implementation of the human model procedure is as discussed in Section 5.2.2.

The random height values are generated through the use of a random seed and the pseudo-random number generation facility provided in the procedural modeling language.

The abdomen and buttocks values are random numbers generated by the random number generator in the Java Math class.

Fifty human models are generated using a for loop. At each iteration, a new random value for the height, abdomen and buttocks parameters is generated. These parameter values are the only ones provided to the human modeling procedure call. The remaining parameters are set to use the default values associated with them.

### 6.1.4 Results

A similar parameterisation experiment was previously conducted to test whether or not distinct models could be generated from a single procedure [Morkel and Bangay, 2006]. The experiment used the same human model as described in Section 5.2.2. Two parameter values were altered to create variations in the model. These parameters are used to shape the abdomen and buttocks regions of a human model. Four different models were shown to illustrate the variety achievable. Each of the models has a differently shaped abdomen and buttocks. This result was achieved by changing these shaping values to influences how pronounced the shaping of each components is.

Figure 6.1 shows the models used in this experiment. A colour version of this image is available in Appendix B.

The test for this experiment is more comprehensive than that used in the previous experiment. This experiment makes use of an additional parameter, which increases the number of distinct models that can be generated.

Fifty human models were generated through the use of a loop. During each iteration, a random number for the height, abdomen and buttocks was generated. The human modeling procedure was called with these parameter values. The rest of the parameters required for the human model are left unassigned, and the default values are used.

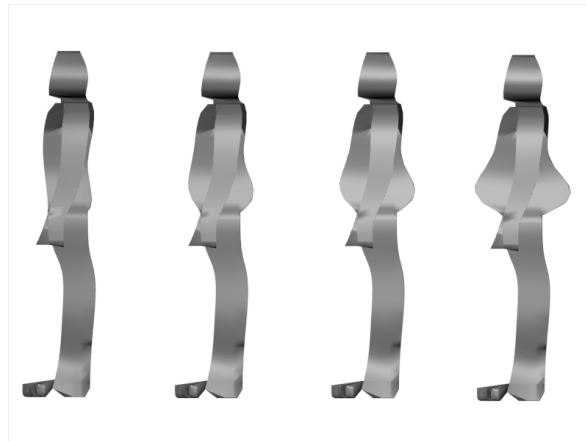


Figure 6.1: A human model with the abdomen and buttocks shaping parameters changed

Figure 6.2 illustrates the effects achievable by varying the height, abdomen and buttocks parameters. Each of the human models in this scene has been created with random values, and is different from every other model in the scene.

Based on the results from the previous experiment and this one, we conclude that it is possible for a single procedure to generate distinct models through the use of parameterisation.

To test whether or not the modeling procedures are robust enough to handle a range of parameter values, we have generated an additional seven human models that increase in height. For the second hypothesis to hold true, the shaping of the human models must be identical, just on smaller or larger scales.

The abdomen and buttocks parameters are assigned random values, with the exception of the abdomen and buttocks regions, the models are identically shaped. The human modeling procedure is thus capable of using a range of parameter values to produce distinct, but predictable models. Figure 6.3 shows the models used.

In conclusion, it is possible to generate a number of distinct models from a single procedure. This is done by altering the parameters used to generate a model. The more parameters provided for modeling purposes, the larger the amount of variation that is possible. If modeling values are specified within the procedure, it becomes more difficult to generate different models using a construct such as a loop, as each of these values has to be manually changed within the procedure.

Another conclusion reached during this experiment is that it is possible for modeling procedures to be robust enough to handle a range of parameter values. A procedure is able to scale

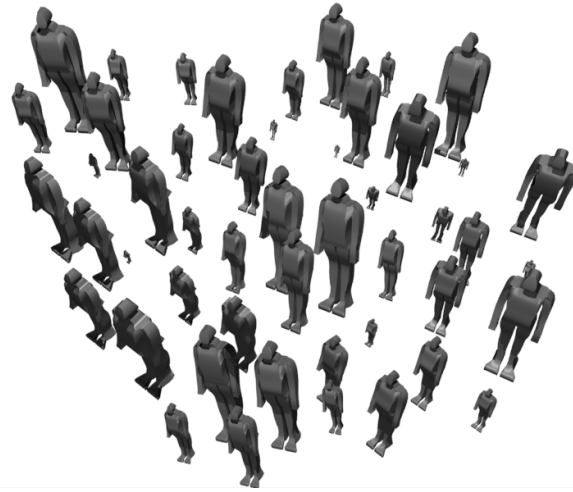


Figure 6.2: A group of human models created by using random height, abdomen and buttocks values

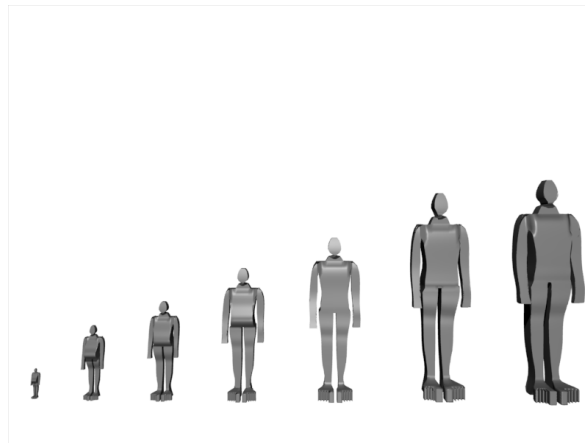


Figure 6.3: Human models with different height, abdomen and buttocks parameter values

to match a range of given parameter values, provided it is written to do so. A procedure that uses manually specified values within the procedure is not able to scale automatically. In this instance, the values need to be updated in the procedure before being applied to the model.

## **6.2 Experiment 2 - Re-usability**

An advantage of using a procedural modeling strategy is the ability to reuse modeling procedures. These procedures can be used to generate entirely new models, or components of these procedures can be applied to different models, thus saving both effort and time.

The purpose of this experiment is to test whether modeling procedures and their individual components can be reused.

### **6.2.1 Objective**

The hypothesis of this experiment is that modeling procedures can be reused on different models. A further test of re-usability is on whether the components of a model can also be reused.

To determine the success of these tests, a comparison is done between the original model and the new models. This is done by determining if the model components reused resemble the original application under the original conditions. For example, if a human leg model is applied to a centipede, the resulting leg should resemble the one on the human model, but in a different scale.

### **6.2.2 Design**

To test our hypothesis, we reuse the leg procedure created for the human model on a variety of objects, which include the luggage model, chairs and couches.

A further test is done to determine if the shaping procedure used for the human leg models can be reused. This test is performed on a chair and couch model with standard chair and couch legs.

### **6.2.3 Implementation**

The luggage model is as described in Section 5.2.3.

The chair and couch models have been updated to provide an option between standard furniture legs and human legs.



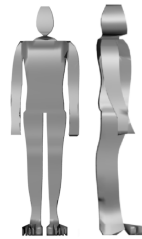


Figure 6.4: A human model with legs and feet

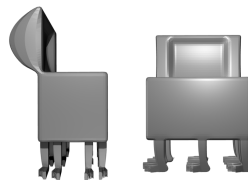


Figure 6.5: The luggage model with legs and feet

#### 6.2.4 Results

Figure 6.4 is a human model with legs created by the leg modeling procedure. This model is provided for comparison purposes, and does not form part of the experimental results.

The first experimental model discussed is the luggage model. The luggage model has twelve human legs grown from its base, which can be seen in Figure 6.5. The number of legs is parameterised and can be varied. An example of a luggage model with more legs can be seen in Figure 6.4.

Each of the legs of the luggage model resembles a shaped human leg. The legs of the luggage are smaller than the legs of the human model, which is catered for by a parameter, but are otherwise created and shaped in the same manner.

The second and third models discussed are the chair and couch models that have human legs, respectively. The legs of the couch model are exaggerated in both this experiment and the next. This is to provide a comparable leg model. Both of these models can be seen in Figure 6.6.

By comparing the human model in Figure 6.4 and the chair and couch models, it can be seen

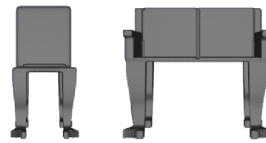


Figure 6.6: A chair and couch model with human legs

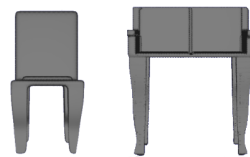


Figure 6.7: A chair and couch model with standard legs that have been shaped using the leg shaping procedure

that the legs on the chair and couch resemble those on the human model.

In light of this, we conclude that the leg modeling procedure created for the human model is reusable in different contexts and by different models.

A further test of re-usability is conducted to see if the process used to shape the human leg model can be applied to the standard chair and couch legs. To determine whether this experiment is successful, we refer to the human model in Figure 6.4. The shape of the chair and couch models should match the shape of the human legs.

Figure 6.7 shows the chair and couch models that have shaping applied to their legs.

The first indication that the shaping procedure has been applied to the chair and couch models is that the legs are not as angular as normal. If one compares the shape of the legs of the chair, couch and human models, it is possible to see similarities, the same leg shape is common in all three models.

Based on these tests, we are able to conclude that modeling procedures can be reused. Components used to create a model may also be reused and applied to different models.

A colour image of the luggage may be seen in Appendix B.

## 6.3 Experiment 3 - Base-shape Independence

Procedures are base-shape independent if they can be applied to arbitrary starting primitives and still achieve predictable results. This is a useful property to have, as it makes the modeling procedures more flexible. It also increases the re-usability possibilities of the modeling procedures.

This experiment tests the base-shape independence of our modeling procedures.

### 6.3.1 Objective

The purpose of this experiment is to determine whether or not our modeling procedures are applicable to primitives of arbitrary shape.

To determine the success of the experiment, we discuss the expected outcomes of applying the modeling procedures to each of the primitives. An example of each of the original models is provided for comparison purposes.

A visual evaluation of each experimental model is performed to determine whether or not the model meets the expected outcomes listed.

### 6.3.2 Design

To test base-shape independence several modeling procedures are applied to a variety starting primitives. The human, chair and table models are used to test base-shape independence.

The parallelepiped torso of the human model is replaced with an egg-shaped primitive. The expected outcome of this experiment is the creation of a “Humpty-Dumpty” model, that has human arms and legs. A head is not added to the model, as the egg-shaped torso is considered the head. The torso of the model is also not shaped, as we wish to retain the egg shape.

The chair modeling procedure is applied to a variety of base shapes, which include a disc-shaped base and a thin parallelepiped base shape. The latter shape is the standard shape used to model a chair. The experiment is, however, run on this shape at different levels of detail.

The table modeling procedure uses a thin parallelepiped base shape as the default starting primitive. To test base-shape independence, we apply the table modeling procedure to a smoothed disc-shaped base and a non-smoothed disc-shaped base which provides an interesting starting primitive.



Figure 6.8: A human model

### 6.3.3 Implementation

The egg-shaped torso primitive used for the “Humpty-Dumpty” model is created by performing smoothing subdivision on a parallelepiped. The resulting shape is a 3D egg-shaped object.

The base parallelepiped for the chair model is created by extending a single cube, through the use of extrusion, to create a base-shape. The number of cubes used to create this base-shape is determined by a modeler-specified parameter. The parameter controls the number of extrusion steps required to make the base-shape. The higher the parameter value, the more extrusion steps required and the larger the amount of detail available for manipulation.

To obtain the different levels of detail required for the experiment, the number of extrusion steps is varied.

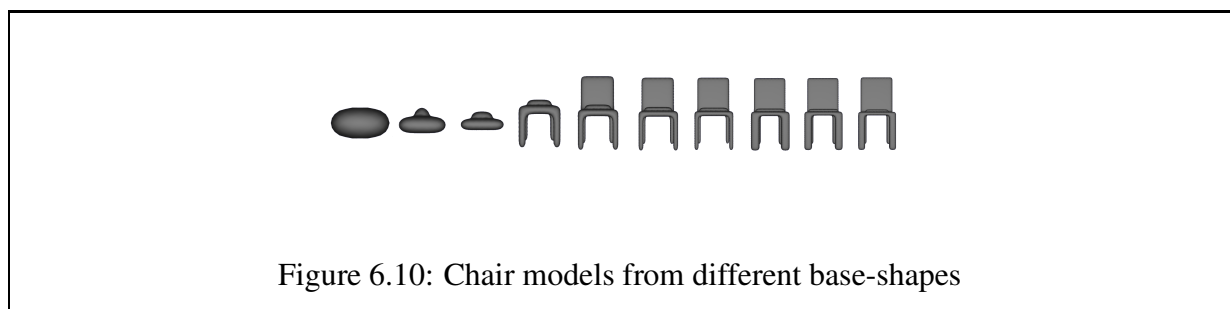
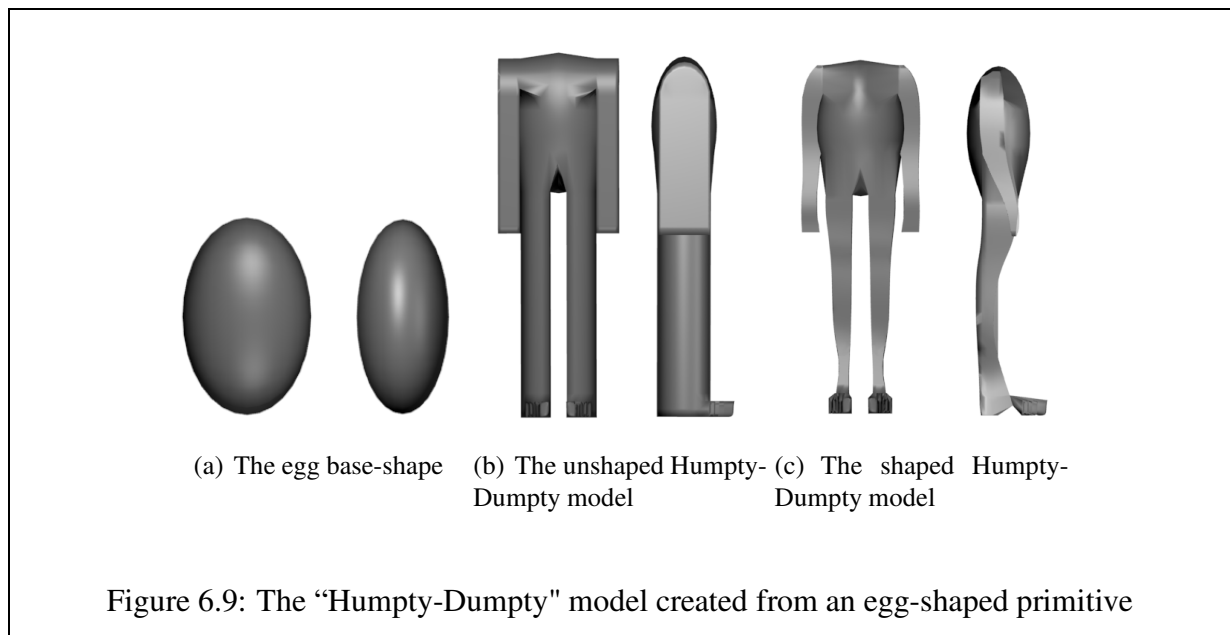
The disc base-shape used for the table model is created in the same manner as that of the egg-shaped primitive for the “Humpty-Dumpty” model. To create the other disc-based shape, the smoothing parameter for subdivision is set to 0.0, thus creating a non-smoothed disc base-shape.

### 6.3.4 Results

An example of a human model is again provided for comparison purposes, and can be seen in Figure 6.8.

The first test is used to create the “Humpty-Dumpty” model discussed in Section 6.3.2. A series of steps from the model creation process have been shown. Figure 6.9a) shows the egg-shaped primitive. Figure 6.9b) shows the egg model with human arms and legs, no shaping has been done. Figure 6.9c) shows the final “Humpty-Dumpty” model.

The requirement of this model that it produce an egg-shaped “Humpty-Dumpty” figure with no head, no torso shaping and shaped human arms and legs. Figure 6.9 shows that this has in fact been accomplished. The arms and legs of the model are comparable to the arms and legs of the



human model in Figure 6.8. A side and front view of each model is provided for comparisons.

Colour images of the “Humpty-Dumpty” and human models are available in Appendix B.

The first base-shape independence test is a success. Several tests remain to test the hypothesis of base-shape independence further.

The second test is used to determine whether or not the chair modeling procedure is base-shape independent.

A starting primitive does not necessarily have to be a completely different shape. It is possible to derive a set of different base-shapes from a single primitive. This is possible by using different levels of detail. For this test, the level of detail strategy discussed in Section 6.3.3 is used. Figure 6.10 shows the chair generation attempts for the detail levels 1 – 10.

The first four chair creation attempts are met with failure, as there is simply not enough detail from which to create a cushion, chair back and legs. The last six attempts result in full



Figure 6.11: A chair model created from a disc base-shape



Figure 6.12: Table models

chair models. The size of the legs change, as more faces are selected with a higher level of detail with the same selection parameters. This is due to a decrease in face size as more detail is added to the model. A larger quantity of smaller components are used to create the same sized parallelepiped, which in turn results in more detail available for manipulation.

The second base-shape independence experiment has met with partial success, as some of the chair generation attempts failed to produce a complete chair model. The chair modeling procedure is robust enough to function on a primitive shape at different levels of detail, provided there is sufficient detail to create the model. In some cases, only part of the model is generated, this is a partial success, but is not sufficient to prove that the chair modeling procedure is always base-shape independent.

Another test performed using the chair modeling procedure is the creation of a chair model from a disc-like shape. This exercise has proved successful, as the chair contains all of the components required to be recognisable as a chair. Figure 6.11 shows the result of this experiment.

The final base-shape independence test that is performed is using the table modeling procedure. The default base-shape for a table is a flat parallelepiped. This base-shape is replaced with a smoothed disc-like surface and a non-smoothed disc-like primitive (see Section 6.3.3) for the purpose of this experiment.

The results of this experiment are shown in Figure 6.12. The first table, which uses a smoothed disc-like shape, resembles an oval table which is a plausible table model. The second table, which uses a non-smoothed disc-like shape, resembles an artistic coffee table.

The table modeling procedure passes the test of base-shape independence.

In conclusion, modeling procedures that can be applied to different types of starting primitive are considered to be base-shape independent. Each of the modeling procedures tested in this experiment have passed this test. Those that did fail, the chair model, failed because of too little detail to create a model with.

## **6.4 Experiment 4 - Model Complexity**

One of the benefits of procedural modeling is the ability to create highly complex models that are tedious to create manually. The complexity is in terms of the amount of repeated geometry in an object.

The procedural modeling environment we have created applies repetitive geometry without modeler intervention. The amount of complexity in a model is controlled by changing a parameter value.

### **6.4.1 Objective**

The hypothesis of this experiment is that procedural modeling techniques are capable of modeling highly complex models. The complexity referred to is in terms of repetitive geometry or structures within a model.

The success of the experiment is determined by the amount of detail visible in the model and the manner in which this detail is added.

### **6.4.2 Design**

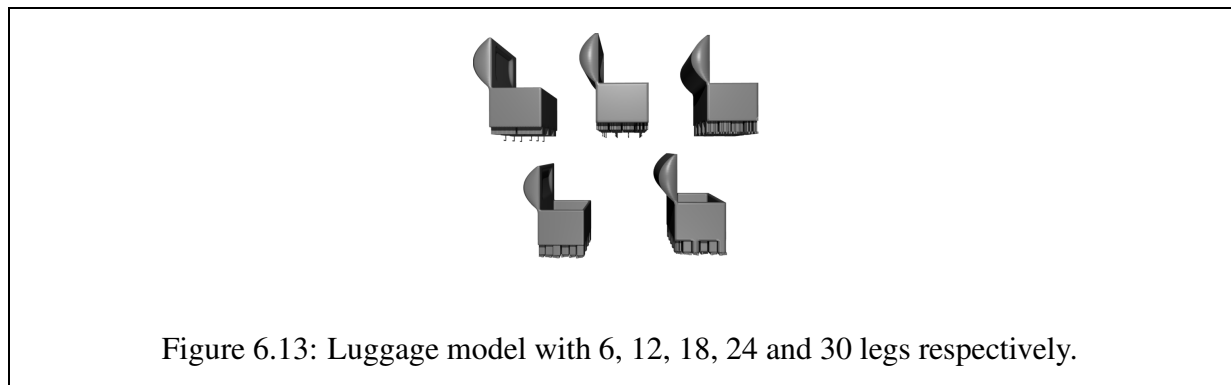
To test this hypothesis, we have devised two experiments that create a more complex model with each time the model is generated.

The first experiment uses the luggage model. To increase model complexity, we generate a series of luggage models, each with a different number of legs.

The second experiment illustrates complexity by generating L-system trees with different levels of recursion.

Number of Legs	Polygon Count
6	29310
12	38919
18	44416
24	57557
30	63086

Table 6.1: The number of polygons in the luggage model increases as the number of legs increases



### 6.4.3 Implementation

The different luggage models are generated through the use of a for loop. Each iteration increases the number of legs added to the model. The number of legs increases by a factor of six each time. The number of rows of legs also increases to accommodate the legs.

The strings for the L-System trees are created by string generator. Each tree has an increased recursion level, which results in more model complexity as more branches for the tree have to be created at each step.

### 6.4.4 Results

Five luggage models have been generated with an increase of six legs at each iteration. As an indication of the increase in complexity of the luggage models, Table 6.1 shows the number of polygons in each of the luggage models. The number of polygons shows how the model grows in size (in terms of polygons) as more detail is added to the model.

Figure 6.13 shows the five luggage models created for this experiment.

The base of each luggage model is subdivided to provide additional detail to create legs with.





Figure 6.14: L-system trees with 3, 4, 5 and 6 levels of recursion

The geometry required to create the legs for each luggage model is created through the use of a looping structure. Each model is generated without modeler interaction.

The detail (legs) in the luggage model is repetitive, which fits the definition of complexity provided in Section 6.4. Manually adding the legs to the luggage model requires a significant amount of time as the geometry needs to be created, then duplicated and stitched to the model.

Based on the hypothesis for this experiment, we conclude that the successive luggage models display increasing levels of complexity.

The second test generates a number of L-system trees with different levels of recursion. Figure 6.14 shows the four L-system trees generated with recursion levels 3, 4, 5 and 6 respectively. As the level of recursion increases, so do the number of branches that are generated for the model. The tree model grows larger and has more child branches created. This adds to the complexity as the number of branches increases significantly at each recursion step.

The tree models are generated without modeler interaction.

The detail generated for the tree models agrees with the definition of complexity provided in Section 6.4. We can thus conclude that procedural modeling permits models of increased complexity to be created without modeler intervention.

A colour image of the luggage may be seen in Appendix B.

## 6.5 Experiment 5 - Reproducible Randomness

Many existing procedural modeling techniques rely on the use of reproducible random numbers Chen et al. [2002], Greuter et al. [2003], Reeves [1983] to create models. The seed values used for this purpose are either stored Chen et al. [2002], Reeves [1983] or can be recreated Greuter et al. [2003] based on a scene specific factor, such as location in space.

An advantage of using pseudo-random numbers that are generated through the use of seed values, is the ability to reproduce these numbers at a later stage. The only requirement is the same seed value originally used be used to reproduce the pseudo-random number.

A pseudo-random number generator has been provided in our procedural modeling environment. This pseudo-random number generator has been tested to determine whether it is functioning as expected. A number of tests are run, with the same seed value consistently provided, to ascertain whether or not the same number is generated each time. The results are indeed as expected, as each test resulted in the same psuedo-random number.

## **6.6 Experiment 6 - Caching**

Caching is a widely used medium of temporarily storing information for quick retrieval. We have provided a caching facility in our procedural modeling environment to provide the same benefits to modelers.

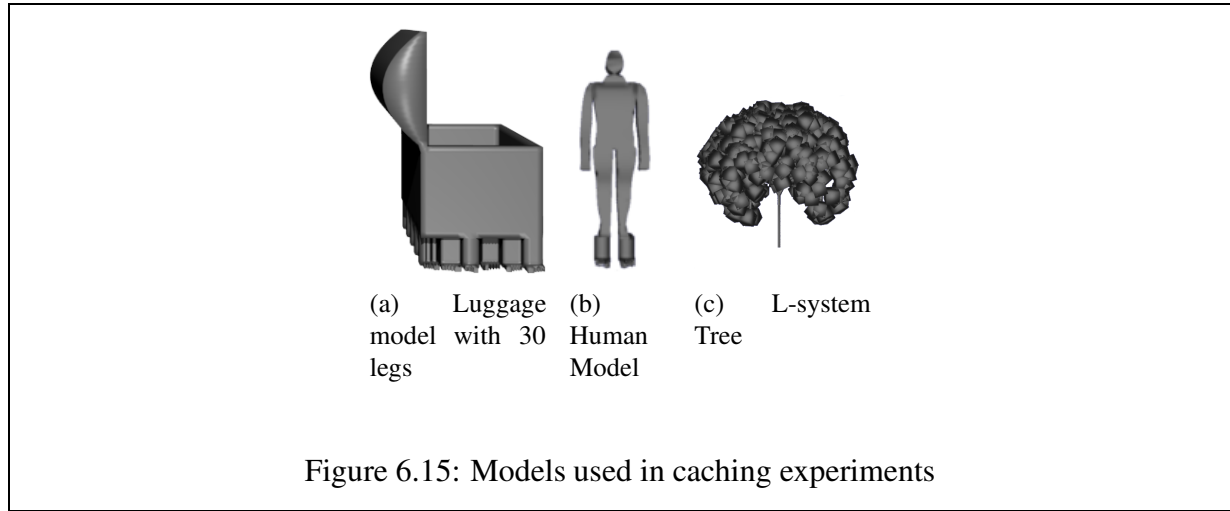
### **6.6.1 Objective**

This experiment tests the caching facility provided in our procedural modeling language. The hypothesis we are testing is that the time taken to generate a model while using caching is significantly less than the time taken without a cache.

### **6.6.2 Design**

To test the hypothesis of this experiment, we have devised three sub experiments, which tests the caching facility on different models.

1. Measure the amount of time taken to create 30 legs for the luggage model using caching, and then without using caching.
2. Repeat experiment 1, using only 2 legs for a human model
3. This experiment is itself split into two smaller experiments. We use an L-system tree for the third experiment which can itself be cached, as can the cylinders which are used to create branches for the tree.
  - (a) Measure the amount of time taken to create an L-system tree that has been cached, and then without a cached model.
  - (b) Measure the amount of time taken to create the same L-System tree with the cylinders cached, and then not cached.



### 6.6.3 Implementation

A stand-alone foot model has been created for the purposes of this caching experiment and is used in the caching tests performed on the luggage and human models. For loops are used to generate the models used for the caching experiments.

### 6.6.4 Results

Figure 6.15 shows examples of the models used for the caching experiments.

A number of experiments were performed to test the influence of caching on the amount of time required to generate a model. Each experiment consists of 500 samples, and is broken up into 50 groups of 10 samples each.

The results of these experiments have been summarised in Table 6.2. For each experiment, two averages are provided. The first average is based on the time taken to generate the first model in each of the 50 groups.

The formula used to calculate this average, per experiment, is  $average = \frac{\sum_1^{50} (timetakenforsample\ 1)}{50}$ . The first average is important, as the experiments that have caching enabled take longer the first time the model is generated. This is because the cache is empty, and the models have to be generated.

The second average provided is calculated by taking the remaining time and samples numbers and dividing. The formula used for each experiment is  $average = \frac{\sum_2^{10} \sum_1^{50} (timetakenforsample\ n)}{(500-50)}$ .

We have identified two anomalies in the results presented in Table 6.2.

- The time taken for sample 1 to generate in the human and tree experiments is faster than

	No Caching	Caching	Improvement (%)
Luggage - first	10.31s	5.51s	46.56
Luggage - remainder	9.66s	4.67s	51.66
Human - first	23.89s	24.27s	-1.57
Human - remainder	23.7s	23.67s	0.13
L-System Tree - first	3.08s	3.71s	-16.98
L-System Tree - remainder	2.53s	0.5s	80.24
Cylinders - first	3.08s	2.91s	5.52
Cylinders - remainder	2.53s	2.41s	4.73

Table 6.2: Average time taken to generate a model in seconds

the time taken to generate a cached equivalent. We hypothesise that this is due to the cache being initialised and a check to an empty cache being performed. To test this hypothesis, we run all the caching tests again this time adding a cache declaration that is never used to the non-cached experiment.

- The average time taken for the remaining non-cached samples of all of the experiments to generate is less than the time taken to generate the first sample. This is an unexpected result. We hypothesise that this is due to the loading of support libraries and initialisations during the first model generation. To test this hypothesis, we run the all the caching tests again. A single model is generated and then discarded before the timings begin.
- The last column in Table 6.2 shows the improvement from not using caching to using it in the form of a percentage. A negative percentage value indicates that the caching performed worse than the non-cached equivalent. The majority of models show a positive improvement. The two models that do not show an improvement are the human and L-system tree models. In both instances, the first sample of the non-cached model takes less time than the cached model. This is causing the negative percentage. The remaining cached models require less time than the non-cached equivalents, albeit slight in the case of the human model.

Table 6.3 shows the results of these additional experiments.

Based on the results in Table 6.3, we conclude the following:

- The addition of a caching statement to a non-cached experiment results in an increase in the time taken to generate a model. In the case of the human model and tree models with cached cylinders, the hypothesis that caching requires more time for the first sample is

	No Caching	Caching	Caching Declaration	Improvement (%)
Luggage - first	9.43s	5.01s	-	46.87
Luggage - remainder	9.42s	4.69s	-	50.21
Human - first	22.57s	22.26s	22.66s	1.37
Human - remainder	22.43s	21.99s	22.57s	1.96
L-System Tree - first	2.29s	2.83s	2.76s	-19.08
L-System Tree - remainder	2.26s	0.46s	2.86s	95.86
Cylinders - first	2.29s	2.18s	2.51s	4.80
Cylinders - remainder	2.26s	2.18s	2.49s	3.54

Table 6.3: Average time taken to generate a model in seconds, with additional experiments to test new hypotheses

invalidated. The caching experiment took less time than the non-cached equivalent experiment. This hypothesis does however, hold true for the L-system tree.

- The last column in Table 6.3 also shows the improvements in time taken to generate cached models as opposed to non-cached models. These improvements are presented as percentages. All of the experiments, with the exception of one, show a positive improvement and thus a reduction in the amount of time required to generate a model when using a cache. The L-system tree model shows a negative improvement for the first samples of the experiments. By adding a caching statement to the non-cached experiment which is never used, we can see that the resulting time for the first sample is closer to the cached time.
- The cached luggage model requires significantly less time to generate than the non-cached model. This is due to the luggage making repeated use of cached leg models. The leg models need only be evaluated once, and they can be repeatedly retrieved from the cache whilst constructing the current model.
- Caching works well for models that contain repetitive components, in this case the luggage and L-system tree models. Caching has reduced the amount of time required to generate a luggage model by more than half, while reducing the amount of time needed to generate an L-system tree by almost three quarters.
- Models that are not highly repetitive in nature, such as the human model, do not benefit very much from the use of caching. A single foot model is insufficient, as a left and right foot model are created. In this experiment it is only the feet of the model that are cached,

further improvements in terms of time required to generate a human model can be gained by caching more of the components, for example the arms and legs.

## 6.7 Conclusion

The parameterisation experiment leads us to conclude that a number of distinct models can be generated from a single modeling procedure. The robustness of a modeling procedure is dependent on how the procedure is written. The modeling procedures used during this experiment are scale-independent and work regardless of the parameter values passed.

The re-usability experiment leads to the conclusion that a modeling procedure can be applied to models other than those it was originally created for. Sub-components of a modeling procedure can also be re-used and applied to different models. This property makes modeling procedures flexible and saves on coding time as modules from existing procedures can be reused.

The base-shape independence experiment allows us to conclude that it is possible for modeling procedures to use arbitrary starting shapes.

The complexity experiment shows that repeated detail can be used to increase the complexity of a model. As the detail is added procedurally, there is no extra cost in terms of programmer effort. The detail can be added through the use of a loop.

The experiment to test the pseudo-number generator provided in the procedural modeling language shows that this facility functions as it should. The same pseudo-random number is generated for each test.

The caching experiment leads us to conclude that caching is most beneficial to models that contain a large amount of repetition. In cases where the repetition is small, there is little improvement in the time taken to model an object.

# Chapter 7

## Conclusions

We have created a procedural modeling environment that is composed of a support environment and set of non-interactive modeling tools. The support environment is a procedural modeling language that facilitates modeling.

The research objectives are presented in Section 1.4.3. This chapter discusses how each of these objectives is satisfied by the work presented in this thesis.

## 7.1 Challenges

One challenge was the identification of a set of tools that are appropriate for use in a procedural modeling environment. The majority of the tools were chosen as they are commonly found in 3D modeling packages:

- A selection tool is essential to modeling and forms the basis of all of our other modeling tools. We reviewed the scripting interfaces of a number of 3D modeling packages in an effort to identify strategies that we can use in our procedural modeling environment. No helpful strategies were found, as the indexing technique used is similar to our initial selection strategy. Scripting languages name each of the objects in the scene with a unique identifier, and then refer to the vertices, edges or faces of the object through the use of index values. In light of this, we explored a number of alternative selection strategies before choosing our current selection mechanism. We chose our selection mechanism as it is scale-independent, the same selection mechanism can be used on a large or small object. We used a parallelepiped bounding volume, which is best suited to perform selections on objects that are of similar shape to itself. Selections on other objects can be performed, but the selection parameters need to be adjusted to give the desired result.

- The curve shaping tool is based on a human modeling tutorial that uses curves on a template image to shape an object. A procedural equivalent of this tool has been added to the tools provided in the procedural modeling environment. The implementation of the curve shaping tool proved challenging, as we had to devise a strategy that allows a 3D point to intersect with a 2D curve. To do this, we project both the points and curves onto the same plane, and then test for intersections.

Several challenges were faced in the creation of our procedural modeling language:

- The parameter rewriting facility: We had to devise a means of firstly retrieving the parameter list from a procedure and then rewriting the procedure call to match the procedure requirements. A two-pass compilation process is used, the first of which identifies the parameters and their types in a parameter list and creates a parameter signature which is used by other modeling procedures to rewrite procedure calls. The second pass is used to rewrite the procedure calls and to format each of the procedures by adding the default parameters and predefined variables.
- The provision of the caching facility: The implementation of the cache was trivial, as an existing hash table data structure was used. The ability to store and retrieve models from the cache is more complicated. The initial approach was to simply store the created model, and retrieve the model from the cache. This approach is problematic, as references to the models are retained. This means that there can never be multiple copies of a model, as the same model is repeatedly returned. To overcome this problem, we create a new model object to be stored in the cache. This removes the referencing problem, as a new model entirely is stored. A copy of the model in the cache is returned when retrieving an object from the cache.

Several challenges were encountered whilst modeling humans and furniture:

- If the model uses a smooth base-shape, the ratios used with the selection mechanism need to be adjusted to perform the selection accurately. To overcome this, the curve shaping tool can be used to adjust a set of faces to make them more accessible to the selection tool. This can be done by shaping the faces to a straight line curve positioned at the maximum range of this set of faces. This makes the geometry in the region of interest a uniform shape. The selection can then proceed as normal. This strategy can also be used between extrusion calls.



- Another challenge faced during modeling is the lack of detail available for manipulation of a model. Scenarios arose in which there was insufficient detail to perform selections. To overcome this, we repeatedly subdivide the region of interest until a threshold is reached, and then perform the selections as required.
- As primitives are extended to create object models, the size of the bounding volume of the model grows to accommodate the expanding model. This has an adverse affect on the ability to perform selections, as the ratios used need to be adjusted to work with a larger model. To overcome this, we use hierarchical modeling. This allows us to perform selections in terms of smaller, already existing selections.

## **7.2 Objective 1 - Review of Related Work**

### **7.2.1 Summary**

The review of related work discusses comparisons between 3D modeling packages and procedural modeling techniques in a number of areas.

The first discussion is based on the modeling strategies used in 3D modeling packages and procedural modeling techniques. A number of 3D modeling strategies have been identified. The descriptions for each of these strategies is well-documented. Several strategies are identified as potential candidates for adaptation to a procedural modeling environment.

The modeling strategies employed by procedural modeling techniques are the same as the techniques themselves. A variety of objects can be modeled by procedural modeling techniques. The application areas of the various techniques are reviewed and summarised.

The next aspect discussed is the similarity between existing procedural modeling languages. The results of this comparison are used to formulate a requirements specification for our own procedural modeling language. The existing procedural modeling languages are contrasted with the scripting interfaces found in 3D modeling packages.

The modeling tools found in 3D modeling packages are compared with those found in procedural modeling techniques. A number of modeling tools are common to both modeling paradigms. Several 3D modeling tools are selected to be adapted to use in a procedural modeling environment.

A comparison of model representations between 3D modeling packages and procedural modeling techniques is performed to determine which tools are commonly found and to identify the most frequently occurring model representation.

### 7.2.2 Conclusions

A number of 3D modeling strategies have been identified from books and tutorials. These strategies are well-documented as sets of instructions and commonly used to model objects in 3D modeling packages. We conclude that 3D modeling strategies can be adapted for use in a procedural modeling environment, as the strategies are algorithmic in nature.

We have shown that the box and curve modeling strategies can be adapted for use in procedural modeling as they are methodical and repetitive in nature. They also support the extrusionist modeling approach.

A number of procedural modeling techniques have been reviewed and we conclude that a procedural modeling strategy is no different from a procedural modeling technique. A number of application areas can be addressed with these procedural techniques, including vegetation, city and fire modeling.

A number of procedural modeling languages exist, with the dates of publication ranging from 1988 to 2002. This leads us to conclude that although a number of attempts have been made to create a fully fledged procedural modeling language, there is room for improvement. This makes the research presented here relevant as it is based on existing strategies, but presents new contributions as well (see Section 7.9). The similarities between the existing procedural modeling languages include the use of parameterisation and predefined variables. However, because of the extra functionality in our environment such as caching and pseudo-random number generation, we conclude that our procedural modeling environment is sufficiently different from existing systems to warrant the creation of a procedural modeling language.

A variety of modeling tools are provided by existing 3D modeling packages. The number of tools is finite. Tools that are commonly found in 3D modeling packages include extrusion, subdivision and a knife tool. In addition, we found that a number of the modeling tools used by procedural techniques overlap with the tools found in 3D modeling packages. This overlap includes tools such as extrusion and subdivision. A number of the identified modeling tools are suitable for adaptation, but we have chosen selection, extrusion, subdivision and stitching tools as a basis for a procedural modelling system. We have proved these tools are sufficient in a non-interactive modeling environment by demonstrating their use in the creation of procedural human and furniture models.

We find that the number of model representations used in existing 3D modeling packages is finite. In particular, modeling strategies and modeling tools are dependent on model representations. For example, certain tools are only applicable to a mesh representation, while others are applicable only to implicit surface representations. There is, however, an overlap of model

representations between procedural modeling techniques and 3D modeling packages. These representations include implicit surfaces and spline patches. We have found that the most frequently occurring model representation in procedural modeling techniques and 3D modeling packages is a polygonal mesh.

## **7.3 Objective 2 - Determine Whether a Procedural Modeling Language can be Created that Encompasses the Requirements and Benefits of Procedural Modeling**

### **7.3.1 Summary**

A number of procedural modeling languages are identified and discussed in the literature review. Each of these languages provides support for some or all of the identified procedural modeling benefits. Each of these languages also meets the requirements of procedural modeling by providing a non-interactive model creation facility. The models themselves are stored as procedures and are evaluated as required. Several of the existing procedural modeling languages provide immediate visual feedback and allow the modeler to interactively modify the procedurally created model.

### **7.3.2 Conclusions**

We conclude that it is possible to create a procedural modeling language that meets the requirements. This can be done by combining the facilities provided by each of the existing procedural modeling languages, which results in a language that meets the requirements of procedural modeling and provides the benefits associated with this modeling strategy.

## **7.4 Objective 3 - The Requirements Specification, Design and Implementation of the Procedural Modeling Language**

### **7.4.1 Summary**

We have based the requirements specification for the procedural modeling language on existing procedural modeling languages. The list of requirements is supplemented by a list of procedural modeling benefits.

The procedural modeling language extends an existing programming language, namely Java. The basic functionality, control and looping constructs and data structures are provided by the base language.

A strict procedure declaration layout is enforced as is the use of default parameter values. These support standardisation and provide a fail-safe mechanism for model creation, respectively. Predefined variables and parameters are provided in each compiled procedure. These are used to provide support for the modeler and to return a model to be rendered.

A number of facilities are provided by the modeling language. A procedure call rewriting facility, which allows for default parameter values. In addition, a caching facility is provided which allows for the temporary storage of geometry for re-use during model generation. A pseudo-random number generator is provided which is a facility for generating reproducible random numbers from seed values.

Most of the requirements identified are implemented. The level of detail and lazy evaluation mechanism is incomplete, but the supporting methods have been provided.

### **7.4.2 Conclusions**

We conclude that it is possible to create a procedural modeling language based on the requirements derived from existing languages, and encompassing the benefits of procedural modeling. We found that it is possible to provide each of the requirements identified.

We found that by extending an existing programming language the procedural language is able to inherit the underlying programming constructs, libraries and facilities. This reduces the amount of time and effort required for the creation of the procedural language.

From our experience in the use of the procedural modeling language, we find that the use of default parameters and procedure call rewriting provides an interface which is more intuitive to use, simplifying the procedural modeling process.

## **7.5 Objective 4 - The Design and Implementation of a Set of Procedural Modeling Tools**

### **7.5.1 Summary**

A set of procedural modeling tools is created based on the tools and modeling strategies identified in 3D modeling packages. The tool-set consists of a selection, extrusion, curve shaping, subdivision and stitching.

Two data-structures are also discussed, namely the polygonal mesh and a selection representation which stores the faces identified and corresponding identifiers during a selection operation.

### **7.5.2 Conclusions**

We conclude that it is feasible to create non-interactive tools based on existing modeling tools, given that extrusion, subdivision and stitching tools are adapted from 3D modeling package tools. However, we have also shown that tools may be derived from modeling strategies, such as the curve shaping tool. While the concept of particular tools remain similar we found it necessary to adapt their implementation for the non-interactive environment, for example the selection tool, where a novel approach which uses regions of space is required to identify faces.

We have found that all of our procedural modeling tools are well-suited to performing non-interactive operations on a polygonal mesh. Each tool leaves the mesh in a consistent state. Future work may include the implementation of more model representations and adapting the modeling tools to work on these representations.

## **7.6 Objective 5 - The Application of the Procedural Modeling Tools in Model Creation**

### **7.6.1 Summary**

The uses and limitations of procedural modeling tools are discussed in detail. This is in an effort to identify the scenarios in which the use of a specific tool is appropriate. Two sets of models, namely humans and furniture, are created to demonstrate the use of the procedural modeling tools and adapted modeling strategies. These models also test the facilities provided by the procedural modeling language.

During the modeling process, we encountered several difficulties with the models. One issue is the inability to perform efficient selection operations on smooth objects. This is solved by shaping regions of a model prior to performing selections. Another problem encountered relates to the use of the bounding volume of the entire model to perform selections. This is alleviated by using named selections, as the region of interest becomes localised to a specific area of the model. One other problem encountered is a lack of detail to perform selections on a model. This can be remedied by repeatedly applying subdivision to the region of interest until a sufficient selection is made.

### 7.6.2 Conclusions

The successful creation a variety of models leads us to conclude that each non-interactive tool is usable in a procedural modeling context. We have demonstrated that the box and curve modeling strategies can be adapted to create procedural models. We conclude that the box modeling strategy works well with a parallelepiped starting primitive but is not as efficient with smooth primitives. The curve modeling strategy is level of detail independent. The higher the level of detail, the closer the approximation of the model to a curve.

Certain limitations were identified regarding a number of tools. For example, the selection tool is not able to function efficiently if the level of detail of a model is too low, or if the surface of an object is too smooth. The subdivision tool requires a manifold mesh on which to operate.

A major advantage of the modeling environment is made evident through the process of model creation. For example, we have found the curve shaping tool to be level of detail independent. Each of the procedural tools is base-shape independent.

## 7.7 Objective 6 - Experiments to Test the Support of the Procedural Modeling Environment

### 7.7.1 Summary

Six experiments are conducted to test that the benefits identified are provided by our procedural modeling language. Specifically the following benefits are tested:

- parameterisation;
- modeling procedure re-usability;
- base-shape independence;
- increased model complexity;
- the ability to create reproducible random numbers;
- caching.

### 7.7.2 Conclusions

The parameterisation experiment leads us to conclude that a number of distinct models can be generated from a single modeling procedure. The robustness of a modeling procedure is dependent on how the procedure is written. The modeling procedures used during this experiment are scale-independent and work regardless of the parameter values passed.

The re-usability experiment leads to the conclusion that a modeling procedure can be applied to models other than those it was originally created for. Sub-components of a modeling procedure can also be re-used and applied to different models. This property makes modeling procedures flexible and saves on coding time as modules from existing procedures can be reused.

The base-shape independence experiment allows us to conclude that it is possible for modeling procedures to use arbitrary starting shapes.

The complexity experiment shows that repeated detail can be used to increase the complexity of a model. As the detail is added procedurally, there is no extra cost in terms of programmer effort. The detail can be added through the use of a loop.

The experiment to test the pseudo-number generator provided in the procedural modeling language shows that this facility functions as it should. The same pseudo-random number is generated for each test.

The caching experiment leads us to conclude that caching is most beneficial to models that contain a large amount of repetition. In cases where the repetition is small, there is little improvement in the time taken to model an object.

We conclude that sufficient evidence exists to show that the benefits of procedural modeling have been realised by our procedural modeling environment.

## 7.8 Conclusions

We conclude that we have created a procedural modeling environment that consists of a procedural modeling language and a set of non-interactive modeling tools.

## 7.9 Research Contributions

The contributions that have been made by this research are as follows:

- A comprehensive literature review of existing procedural modeling techniques has been conducted.

- The creation of a scale-independent selection mechanism which works with regions of space.
- A curve shaping tool that deforms a model non-interactively and is level of detail independent is presented.
- A smoothing factor is provided to control the amount of smoothing a model undergoes during subdivision.
- A procedural modeling language that supports the benefits of procedural modeling is defined.
- A caching facility is provided which allows objects to be temporarily stored and retrieved during modeling. The use of this facility reduces the amount of time required to create a model that is composed of repetitive geometry.
- A demonstration of the use of the procedural modeling language to generate a variety of models is provided, which exposes the advantages and limitations of the procedural modeling environment and its tools.
- Certain modeling lessons related to procedural modeling have been identified and documented during the modeling process.
- A number of experiments have been defined and executed in order to validate the benefits of a procedural modeling environment.

## **7.10 Future Work**

There are several facilities that can be added to this procedural modeling environment. These include support for transformations, completion of the language-controlled level of detail facility, animation, a knife tool and a lattice deformation tool. Additional model representations can be added to the procedural environment to provide a variety to the modeler. The tools have to be updated or re-implemented to work with other model representations.



# Appendix A

## Bison Grammar

```
PML : boundingvolumedeclaration
    proceduredeclaration
    body
boundingvolumedeclaration : BOUNDINGVOLUME
                           boundingbody
                           |

boundingbody : '{'
             contents
             '}'

proceduredeclaration: flags
                   MODEL
                   mode
                   NAME
                   arglist

flags : CACHEABLE
      |

mode : OUT
     |
     INOUT

arglist : '('
```

```
argumentdecl
    ')'

argumentdecl : type NAME '=' NAME
              | argumentdecl
                ','
                argumentdecl

type : INT
      | DOUBLE
      | STRING
      | VECTOR
      | BOOLEAN
      | CURVE

body : '{'
      contents
      '}'

contents : contentselements
          FINALBLOCK
          |
          FINALBLOCK

contentselements: BLOCK
                 |
                 PMLPROC
                 |
                 contentselements contentselements
```

# Appendix B

## Colour Models

The textured models are provided to enhance the view of the geometry of the models and not to improve the realism of these models.

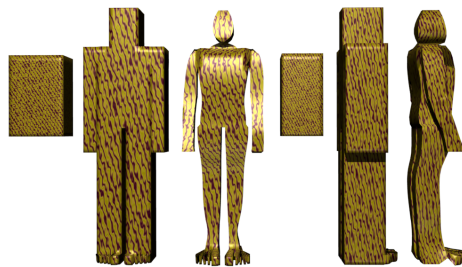


Figure B.1: Textured human model

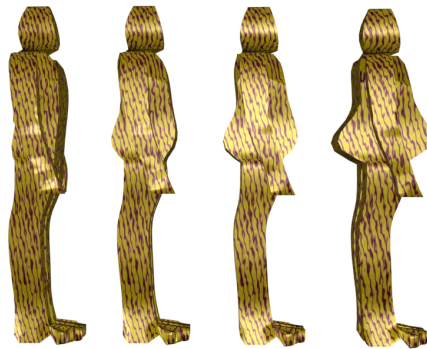


Figure B.2: Textured human model with different parameter values for stomach and buttocks

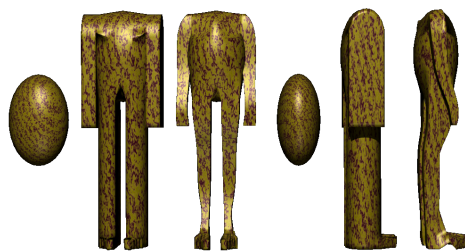


Figure B.3: Textured "Humpty-Dumpty" model



Figure B.4: Textured luggage

# Appendix C

## Examples of Modeling Procedures

### C.1 Human Modeling Procedure

```
Model out HumanFigure (double height = 6.0, String gender = "male", boolean cubical = true,
                        double tummy = 0.0, double butt = 0.0)
{
    ///Conversions
    double feet = height;
    double cm = feet * 30.48;
    double m = cm / 100;
    double ml = cm * 10;

    double head = m/8.0;

    double heightRatio = 61.5/23.0;

    double lengthRatio = 40.0/23.0;
    double widthRatio = 29.0/23.0;

    ///Ratios taken from Loomis
    double torsoHeight = heightRatio * (head);
    double torsoLength = lengthRatio * (head);
    double torsoWidth = widthRatio * (head);

    String name = "torso";

    double divBy = 12.0;

    if (cubical)
        model = #Torso (divBy = divBy, height = torsoHeight, length = torsoLength, width = torsoWidth, name = name);
    else
        model = #Egg (height = torsoHeight, length = torsoLength, width = torsoWidth, name = name);
    BoundingBox preArms = new BoundingBox (model.mesh);

    ///Work on the arms...the total height of the arm (with extended fingers) is 3 2/3 heads, 3 1/2 heads.
    ///The length of the arm is 9ml (at the widest spot), which equates to a ratio of 9/23. The width of the
    ///arm (at the widest point) is 18ml, so a ratio of 18/23. We are going
    ///to ignore the width for the time being, as we are working with the global width (torso width).

    ///The curves should fix this.
    double armLength = (9.0/23.0) * head;

    ///Ratio for where to extrude the arms: ratio = 16/61 (the shoulders span from the top of the torso 0, to 16ml down)
    double ratio = 16.0/61.0;
    FaceSet armSet = model.mesh.select (0.0, 1.0, (1.0 - ratio), 1.0, 0.0, 1.0, model.mesh.getFaceSet (name), "arms");

    FaceSet rightArm = model.mesh.select (0.0, 0.01, 0.0, 1.0, 0.0, 1.0, armSet, "rightShoulder");
```

```

FaceSet leftArm = model.mesh.select (0.99, 1.0, 0.0, 1.0, 0.0, 1.0, armSet, "leftShoulder");

int count = 0;

while ((rightArm.numFaces () < 1) || (leftArm.numFaces () < 1))
{
    model.mesh.sqrt2AdaptiveSubdivide (armSet, 2, 1.0);
    rightArm = model.mesh.select (0.0, 0.3, 0.0, 1.0, 0.0, 1.0, armSet, "rightShoulder");
    leftArm = model.mesh.select (0.7, 1.0, 0.0, 1.0, 0.0, 1.0, armSet, "leftShoulder");
    count++;
}

rightArm.setParent (name);
leftArm.setParent (name);

model.mesh.removeLabelFromFaceSet (rightArm, "torso");
model.mesh.removeLabelFromFaceSet (leftArm, "torso");

model.mesh.addFaceSet (rightArm);
model.mesh.addFaceSet (leftArm);
model = #Arm (divBy = (divBy/2.0), height = head, length = armLength, width = torsoWidth,
    heads = 3.0, name = "rightShoulder");

model = #Arm (divBy = (divBy/2.0), height = head, length = armLength, width = torsoWidth,
    heads = 3.0, name = "leftShoulder");
///  
Extrude the neck and head...the required ratio is 7ml/40ml on each side of the shoulders
ratio = 7.0/40.0;

FaceSet neck = model.mesh.select ((0.0 + ratio), (1.0 - ratio), 0.99, 1.0, 0.0, 1.0,
    model.mesh.getFaceSet (name), "neck");
neck.setParent (name);
model.mesh.removeLabelFromFaceSet (neck, "torso");
model.mesh.addFaceSet (neck);

model = #Head (height = head, length = torsoLength, width = torsoWidth, name = "neck");
///  
Work on the Legs
FaceSet check = model.mesh.select (0.0, 1.0, 0.0, 0.01, 0.0, 1.0,
    model.mesh.getFaceSet (name), "check");

FaceSet rightThigh = new FaceSet ();
FaceSet leftThigh = new FaceSet ();

if (check.numFaces () > 0)
{
    rightThigh = model.mesh.select (0.0, 0.5, 0.0, 0.01, 0.0, 1.0,
        model.mesh.getFaceSet (name), "rightThigh");
    leftThigh = model.mesh.select (0.5, 1.0, 0.0, 0.01, 0.0, 1.0,
        model.mesh.getFaceSet (name), "leftThigh");
}

if ((rightThigh.numFaces () < 10) || (leftThigh.numFaces () < 10))
{
    BoundingBox postArms = new BoundingBox (model.mesh);

    double diff = 0.0;
    if (postArms.min[0] >= 0.0)
        diff = (preArms.min[0] - postArms.min[0]);
    else
        diff = (postArms.min[0] - preArms.min[0]);

    double fullLength = (postArms.max[0] - postArms.min[0]);
    double xRatioMin = diff/fullLength;

    diff = postArms.max[0] - preArms.max[0];
    double xRatioMax = diff/fullLength;

    FaceSet legSet = model.mesh.select ((0.0), (1.0), 0.0, 0.3, 0.0, 1.0,
        model.mesh.getFaceSet (name), "legs");
    model.mesh.addFaceSet (legSet);

    double legRatio = 0.25;

```

```

while (((rightThigh.numFaces () < 10) || (leftThigh.numFaces () < 10)))
{
    rightThigh = model.mesh.select (0.0, 0.5, 0.0, 1.0, 0.0, 1.0,
        model.mesh.getFaceSet ("legs"), "rightThigh");
    leftThigh = model.mesh.select (0.5, 1.0, 0.0, 1.0, 0.0, 1.0,
        model.mesh.getFaceSet ("legs"), "leftThigh");
}

rightThigh.setParent (name);
leftThigh.setParent (name);

model.mesh.removeLabelFromFaceSet (rightThigh, "torso");
model.mesh.removeLabelFromFaceSet (leftThigh, "torso");

model.mesh.addFaceSet (rightThigh);
model.mesh.addFaceSet (leftThigh);

////The ratio for the legs = 4 heads. Since we are catering for the leg in four pieces -
////thigh, knee, shin and ankle, we pass a single head at a time
double legHeight = head;

///First right leg
model = #Leg (numToes = 5, threshold = 3, height = legHeight, length = torsoLength, width = torsoWidth,
    heads = 4.0, name = "leftThigh");

//Next is left leg
model = #Leg (numToes = 5, threshold = 3, height = legHeight, length = torsoLength, width = torsoWidth,
    heads = 4.0, name = "rightThigh");
model = #FitToCurves (height = torsoHeight, length = torsoLength, width = torsoWidth, head = head,
    tummy = tummy, butt = butt);

model.mesh.updateNormals ();
}

```

## C.2 Torso Shaping Procedure

```

Model inout TorsoCurves (double height = 1.0, double length = 1.0, double width = 1.0, double head = 1.0,
    double tummy = 0.0, double butt = 0.0, double handles = 0.0, String name = "torso")
{
    VRPoint points [] = new VRPoint [10];

    double val = height/12.0;
    FaceSet torso = model.mesh.getFaceSet (name);
    BoundingBox myBox = new BoundingBox (torso);

    double minX = myBox.min[0];
    double minY = myBox.min[1];
    double minZ = myBox.min[2];
    double maxX = myBox.max[0];
    double maxY = myBox.max[1];
    double maxZ = myBox.max[2];
    double midX = (minX + maxX)/2.0;
    double midY = (minY + maxY)/2.0;
    double midZ = (minZ + maxZ)/2.0;

    double thirdY = (minY + maxY)/3.0;

    double offSet = 0.02;

    points[0] = new VRPoint (maxX, maxY + offSet, 0.0);
    points[1] = new VRPoint (maxX, maxY, 0.0);
    points[2] = new VRPoint (maxX, maxY - head, 0.0);
    points[3] = new VRPoint ((maxX - (5.0/40.0 * (minX+maxX))), midY, 0.0);
}

```

```

points[4] = new VRPoint ((maxX - (7.0/40.0 * (minX+maxX))) + handles,
    maxY - (head + head*(2.0/3.0)), 0.0);
points[5] = new VRPoint ((maxX - (6.0/40.0 * (minX+maxX))) + handles,
    maxY - head*2, 0.0);
points[6] = new VRPoint ((maxX - (4.5/40.0 * (minX+maxX))) + handles,
    maxY - (head*2 + head/6.0), 0.0);
points[7] = new VRPoint ((maxX - (4.0/40.0 * (minX+maxX))) + handles,
    maxY - (head*2 + head/3.0), 0.0);
points[8] = new VRPoint ((maxX - (3.0/40.0 * (minX+maxX))) + handles,
    maxY - (head*2 + head/3.0 + head/6.0), 0.0);
points[9] = new VRPoint ((maxX - (4.0/40.0 * (minX+maxX))) + handles, minY, 0.0);

Curve rightCurve = new BezierSpline (points.length - 1, points);

points[0] = new VRPoint (minX, maxY + offSet, 0.0);
points[1] = new VRPoint (minX, maxY, 0.0);
points[2] = new VRPoint (minX, maxY - head, 0.0);
points[3] = new VRPoint ((minX + (5.0/40.0 * (minX+maxX))), midY, 0.0);
points[4] = new VRPoint ((minX + (7.0/40.0 * (minX+maxX))) - handles,
    maxY - (head + head*(2.0/3.0)), 0.0);
points[5] = new VRPoint ((minX + (6.0/40.0 * (minX+maxX))) - handles,
    maxY - head*2, 0.0);
points[6] = new VRPoint ((minX + (4.5/40.0 * (minX+maxX))) - handles,
    maxY - (head*2 + head/6.0), 0.0);
points[7] = new VRPoint ((minX + (4.0/40.0 * (minX+maxX))) - handles,
    maxY - (head*2 + head/3.0), 0.0);
points[8] = new VRPoint ((minX + (3.0/40.0 * (minX+maxX))) - handles,
    maxY - (head*2 + head/3.0 + head/6.0), 0.0);
points[9] = new VRPoint ((minX + (4.0/40.0 * (minX+maxX))) - handles, minY, 0.0);

Curve leftCurve = new BezierSpline (points.length - 1, points);

FaceSet leftSide = model.mesh.select (0.0, 0.01, 0.5, 1.0, 0.0, 1.0,
    model.mesh.getFaceSet ("torso"), "leftside");
FaceSet rightSide = model.mesh.select (0.99, 1.0, 0.5, 1.0, 0.0, 1.0,
    model.mesh.getFaceSet ("torso"), "rightside");

model.mesh.shapeToCurve (rightSide, rightCurve, new VRVector (1, 0, 0), new VRVector (0, 0, 1));
model.mesh.shapeToCurve (leftSide, leftCurve, new VRVector (-1, 0, 0), new VRVector (0, 0, 1));

leftSide = model.mesh.select (0.0, 0.1, 0.0, 0.6, 0.0, 1.0,
    model.mesh.getFaceSet ("torso"), "leftside");
rightSide = model.mesh.select (0.9, 1.0, 0.0, 0.6, 0.0, 1.0,
    model.mesh.getFaceSet ("torso"), "rightside");

model.mesh.shapeToCurve (rightSide, rightCurve, new VRVector (1, 0, 0), new VRVector (0, 0, 1));
model.mesh.shapeToCurve (leftSide, leftCurve, new VRVector (-1, 0, 0), new VRVector (0, 0, 1));

//Start working on the front and back curves...
points = new VRPoint [ 11 ];

points[0] = new VRPoint (midX, maxY + offSet, (maxZ + (10.5/29.0 * (minZ+maxZ))));
points[1] = new VRPoint (midX, maxY, (maxZ + (10.5/29.0 * (minZ+maxZ))));
points[2] = new VRPoint (midX, maxY - (head * (1/3)), (maxZ + (3.5/29.0 * (minZ+maxZ))));
points[3] = new VRPoint (midX, maxY - (head * (2/3)), maxZ);
points[4] = new VRPoint (midX, maxY - head, maxZ);
points[5] = new VRPoint (midX, midY, (maxZ + (0.5/29.0 * (minZ+maxZ))));
points[6] = new VRPoint (midX, midY - (head * (1/3)), (maxZ + (0.5/29.0 * (minZ+maxZ))));
points[7] = new VRPoint (midX, midY - (head * (2/3)), (maxZ + (1.5/29.0 * (minZ+maxZ))));
points[8] = new VRPoint (midX, midY - head, (maxZ + (2.5/29.0 * (minZ+maxZ))) + tummy);
points[9] = new VRPoint (midX, (midY - head) - (head * (1/3)), (maxZ + (4.5/29.0 * (minZ+maxZ))));
points[10] = new VRPoint (midX, minY, (maxZ + (3.5/29.0 * (minZ+maxZ))));

Curve frontCurve = new BezierSpline (points.length - 1, points);

double y = 0.1;

points[0] = new VRPoint (midX, maxY + offSet, (minZ - (4.0/29.0 * (minZ+maxZ))));
points[1] = new VRPoint (midX, maxY, (minZ - (4.0/29.0 * (minZ+maxZ))));
points[2] = new VRPoint (midX, maxY - (head * (1/3)), (minZ - (0.5/29.0 * (minZ+maxZ))));
points[3] = new VRPoint (midX, maxY - (head * (2/3)), minZ);

```



```

points[4] = new VRPoint (midX, maxY - head, minZ);
points[5] = new VRPoint (midX, midY, (minZ - (5.5/29.0 * (minZ+maxZ))));
points[6] = new VRPoint (midX, midY - (head * (1/3)), (minZ - (9.5/29.0 * (minZ+maxZ))));
points[7] = new VRPoint (midX, midY - (head * (2/3)), (minZ - (7.5/29.0 * (minZ+maxZ))));
points[8] = new VRPoint (midX, midY - head, (minZ - (2.5/29.0 * (minZ+maxZ))));
points[9] = new VRPoint (midX, (midY - head) - (head * (1/3)), (minZ - (0.5/29.0 * (minZ+maxZ))) - butt);
points[10] = new VRPoint (midX, minY - 0.1, (minZ - (5.5/29.0 * (minZ+maxZ))) - butt);

Curve backCurve = new BezierSpline (points.length - 1, points);

FaceSet back = model.mesh.select (0.0, 1.0, 0.0, 1.0, 0.0, 0.2, model.mesh.getFaceSet ("torso"), "front");
FaceSet front = model.mesh.select (0.0, 1.0, 0.0, 1.0, 0.8, 1.0, model.mesh.getFaceSet ("torso"), "front");

model.mesh.shapeToCurve (front, frontCurve, new VRVector (0, 0, 1), new VRVector (1, 0, 0));
model.mesh.shapeToCurve (back, backCurve, new VRVector (0, 0, -1), new VRVector (1, 0, 0));
}

```

## C.3 Foot Modeling Procedure

```

Model inout Foot (int numToes = 5, int threshold = 5, int needed = 0, double bigToe = 0.3,
                  double extrudeUnit = 1.0, String name = null)
{
    // Creates a foot. Down corresponds to -Y.
    VRPoint a = new VRPoint (3.0, 2.0, 3.0);
    VRPoint p = #transform (coordinates = "world", point = a);

    FaceSet t = model.mesh.getFaceSet (name);
    VRVector direction;
    FaceSet cap = new FaceSet ();
    t = model.mesh.select (0.0, 1.0, 0.0, 1.0, 0.0, 1.0, t, name);

    // squares off the foot under subdivision.
    int ankle = 6;

    if (needed == 1)
    {
        for (int i = 0; i < ankle; i++)
        {
            cap = new FaceSet ();
            direction = VRVector.multiply ((-extrudeUnit/(double) ankle), Y);
            #extrude (faceList = t, extrudeBy = direction, partName = name, capFaces = cap);
            t = cap;
        }
    }
    t = model.mesh.select (0.0, 1.0, 0.0, 0.35, 0.9, 1.0,
                          model.mesh.getFaceSet (name), name + "toes");
    model.mesh.removeLabelFromFaceSet (t, name);
    model.mesh.addFaceSet (t);

    for (int i = 0; i < ankle/3; i++)
    {
        cap = new FaceSet ();
        direction = VRVector.multiply ((extrudeUnit / (double) ankle), Z);
        #extrude (faceList = t, extrudeBy = direction, partName = name, capFaces = cap);
        t = cap;
    }

    double remainder = 1.0 - bigToe;

    double toeSize = remainder / ((double) (numToes - 1)); //-1, as we have catered for the big toe already

    boolean allToes = false; //Assume we have no selections for all the toes

    int counter = 0;

    FaceSet toes [] = new FaceSet [numToes];
}

```

```

double minX;
double maxX;

double minZ = 0.75;
double maxZ = 1.0;

double minY = 0.0;
double maxY = 1.0;

while (!(allToes))
{
    System.out.println(counter);

    counter++;
    if (counter > 30)
        break;
    boolean temp = true;

    FaceSet toeRegion = model.mesh.select (0.0, 1.0, minY, maxY, minZ, maxZ,
                                           model.mesh.getFaceSet (name + "toes"), name + "toeRegion");

    minX = 0.0;
    maxX = bigToe;
    for (int i = 0; i < numToes; i++)
    {
        double mx = minX;
        double Mx = maxX;

        if (name.charAt(0) != 'l')
        {
            mx = 1.0 - maxX;
            Mx = 1.0 - minX;
        }
        toes[i] = model.mesh.select (mx, Mx, 0.0, 1.0, 0.0, 1.0, toeRegion, name + "toe" + i);
        temp = temp && !(toes[i].numFaces () < threshold);
        minX = mx;
        maxX = Mx;
    }

    if (!(temp))
    {
        model.mesh = model.mesh.sqr2AdaptiveSubdivide(toeRegion, 2, 0.5);
    }

    allToes = temp;
}

model.mesh.visibleFace = name + "toeRegion";

for (int i = 0; i < toes.length; i++)
{
    model.mesh.removeLabelFromFaceSet (toes[i], name);
    model.mesh.removeLabelFromFaceSet (toes[i], name + "toeRegion");
    model.mesh.addFaceSet (toes[i]);
}

for (int i = 0; i < toes.length; i++)
{
    t = model.mesh.select (0.0, 1.0, 0.0, 1.0, 0.0, 1.0,
                          model.mesh.getFaceSet (name + "toe" + i), "toefront");

    cap = new FaceSet ();
    direction = VRVector.multiply (extrudeUnit/(i+2), Z);
    #extrude (faceList = t, extrudeBy = direction, partName = name + "toe" + i, capFaces = cap);
}
}

```

## C.4 Chair Procedure

Model out Chair (double length = 2.0, double depth = 2.0, double legLength = 2.0, double backLength = 1.5, int numLegs = 4, double segments = 6.0, int type = 0, double offSet = 2.0, boolean arms = false, boolean shape = false, boolean cubic = true, boolean feet = false, boolean file = false, String fileName = null)

```
{
    ///Control the amount of detail
    double heightUnit = backLength/segments;
    double lengthUnit = length/segments;
    double depthUnit = depth/segments;
    double legUnit = legLength/segments;

    String name = "chair";

    FaceSet t = new FaceSet ();

    VRVector direction;
    FaceSet cap = new FaceSet ();
    double cushXMin = 0.0;
    double cushXMax = 0.0;
    double cushZMin = 0.0;
    double cushZMax = 0.0;
    if (cubic)
    {
        model = #Cube (dimension = (lengthUnit/2.0));
        FaceSet chairSet = new FaceSet (name, model.mesh.faces);
        model.mesh.addFaceSet (chairSet);

        t = model.mesh.select (0.9, 1.0, 0.0, 1.0, 0.0, 1.0, model.mesh.getFaceSet (name), name + "len");

        for (double i = lengthUnit; i <= length; i += lengthUnit)
        {
            cap = new FaceSet ();
            direction = VRVector.multiply (lengthUnit, X);
            #extrude (faceList = t, extrudeBy = direction, partName = name, capFaces = cap);
            t = cap;
        }

        t = model.mesh.select (0.0, 1.0, 0.0, 1.0, 0.99, 1.0, model.mesh.getFaceSet (name), name + "len");

        for (double i = depthUnit; i <= depth; i += depthUnit)
        {
            cap = new FaceSet ();
            direction = VRVector.multiply (depthUnit, Z);
            #extrude (faceList = t, extrudeBy = direction, partName = name, capFaces = cap);
            t = cap;
        }
        cushXMin = 0.1;
        cushXMax = 0.9;
        cushZMin = 0.2;
        cushZMax = 0.9;
    }
    else
    {
        model = #Disc ();
        FaceSet chairSet = new FaceSet (name, model.mesh.faces);
        model.mesh.addFaceSet (chairSet);
        cushXMin = 0.25;
        cushXMax = 0.75;
        cushZMin = 0.25;
        cushZMax = 0.8;
    }
    ///Model the back of the chair, simple to begin with
    t = model.mesh.select (0.0, 1.0, 0.65, 1.0, 0.0, 0.2, model.mesh.getFaceSet (name), name + "back");

    for (double i = heightUnit; i <= backLength; i += heightUnit)
    {
        cap = new FaceSet ();
        direction = VRVector.multiply (heightUnit, Y);
        #extrude (faceList = t, extrudeBy = direction, partName = name, capFaces = cap);
    }
}
```

```

    t = cap;
}

t = model.mesh.select (0.0, 1.0, 0.05, 1.0, 0.0, 0.2, model.mesh.getFaceSet (name), name + "back");
model.mesh.removeLabelFromFaceSet (t, name);
model.mesh.addFaceSet (t);

t = model.mesh.select (cushXMin, cushXMax, 0.05, 1.0, cushZMin, cushZMax, model.mesh.getFaceSet (name), name + "cushion");
model.mesh.removeLabelFromFaceSet (t, name);
model.mesh.addFaceSet (t);

cap = new FaceSet ();
direction = VRVector.multiply (depthUnit*(0.75), Y);

#extrude (faceList = t, extrudeBy = direction, partName = name, capFaces = cap);
t = model.mesh.select (0.0, 1.0, 0.0, 1.0, 0.0, 1.0, model.mesh.getFaceSet (name + "cushion"), name + "subdiv");
model.mesh = model.mesh.sqrt2AdaptiveSubdivide (t, 2, 1.0);

///Model the legs of the chair...
FaceSet legs = new FaceSet ();

//Leg 1
t = model.mesh.select (0.0, 0.25, 0.0, 0.25, 0.0, 0.25, model.mesh.getFaceSet (name), "r" + name + "leg1");
model.mesh.removeLabelFromFaceSet (t, name);
model.mesh.addFaceSet (t);

legs.mergeFaceSets (t);

//Leg 2
t = model.mesh.select (0.75, 1.0, 0.0, 0.25, 0.0, 0.25, model.mesh.getFaceSet (name), "l" + name + "leg2");
model.mesh.removeLabelFromFaceSet (t, name);
model.mesh.addFaceSet (t);

legs.mergeFaceSets (t);

//Leg 3
t = model.mesh.select (0.75, 1.0, 0.0, 0.25, 0.75, 1.0, model.mesh.getFaceSet (name), "l" + name + "leg3");
model.mesh.removeLabelFromFaceSet (t, name);
model.mesh.addFaceSet (t);

legs.mergeFaceSets (t);

//Leg 4
t = model.mesh.select (0.0, 0.25, 0.0, 0.25, 0.75, 1.0, model.mesh.getFaceSet (name), "r" + name + "leg4");
model.mesh.removeLabelFromFaceSet (t, name);
model.mesh.addFaceSet (t);

model.mesh.visibleFace = name + "leg";

legs.mergeFaceSets (t);

System.out.println ("Done legs");

if (feet)
{
    ///Add feet to the chairs

    model = #Leg (numToes = 5, threshold = 3, height = 0.5, name = ("r" + name + "leg1"));
    model = #Leg (numToes = 5, threshold = 3, height = 0.5, name = ("l" + name + "leg2"));
    model = #Leg (numToes = 5, threshold = 3, height = 0.5, name = ("l" + name + "leg3"));
    model = #Leg (numToes = 5, threshold = 3, height = 0.5, name = ("r" + name + "leg4"));

    model = #LegCurves (right = ("r" + name + "leg1"), left = ("l" + name + "leg2"));
    model = #LegCurves (right = ("r" + name + "leg4"), left = ("l" + name + "leg3"));
}
else
{
    System.out.println ("Here!");

    for (double i = legUnit; i <= legLength; i += legUnit)
    {
        cap = new FaceSet ();
    }
}

```

```

        direction = VRVector.multiply (-legUnit, Y);
        #extrude (faceList = legs, extrudeBy = direction, partName = name, capFaces = cap);
        legs = cap;
    }
}
//Arms
if (arms)
{
    FaceSet armSet = new FaceSet ();

    t = model.mesh.select (0.0, 0.2, 0.25, 0.5, 0.99, 1.0, model.mesh.getFaceSet (name + "back"), name + "arm1");
    //model.mesh.removeLabelFromFaceSet (t, name + "back");
    model.mesh.addFaceSet (t);

    armSet.mergeFaceSets (t);

    t = model.mesh.select (0.8, 1.0, 0.25, 0.5, 0.99, 1.0, model.mesh.getFaceSet (name + "back"), name + "arm2");
    //model.mesh.removeLabelFromFaceSet (t, name + "back");
    model.mesh.addFaceSet (t);

    armSet.mergeFaceSets (t);

    for (double i = depthUnit; i <= depth; i += depthUnit)
    {
        cap = new FaceSet ();
        direction = VRVector.multiply (depthUnit, Z);
        #extrude (faceList = armSet, extrudeBy = direction, partName = name, capFaces = cap);
        armSet = cap;
    }

    armSet = new FaceSet (); //Reset the arm faceset, as we are going to extrude arms from the base of the chair

    t = model.mesh.select (0.0, 0.2, 0.99, 1.0, 0.8, 1.0, model.mesh.getFaceSet (name), name + "arm3");
    armSet.mergeFaceSets (t);
    t = model.mesh.select (0.8, 1.0, 0.99, 1.0, 0.8, 1.0, model.mesh.getFaceSet (name), name + "arm4");
    armSet.mergeFaceSets (t);

    for (double i = heightUnit; i <= backLength*0.4; i += heightUnit)
    {
        cap = new FaceSet ();
        direction = VRVector.multiply (depthUnit, Y);
        #extrude (faceList = armSet, extrudeBy = direction, partName = name, capFaces = cap);
        armSet = cap;
    }

    FaceSet arm1 = model.mesh.select (0.8, 1.0, 0.2, 0.4, 0.8, 1.0, model.mesh.getFaceSet (name + "back"), name + "arm1");
    FaceSet arm2 = model.mesh.select (0.8, 1.0, 0.0, 0.99, 0.8, 1.0, model.mesh.getFaceSet (name), name + "arm2");

    model.mesh.join (arm1, model.mesh, arm2);

    arm1 = model.mesh.select (0.0, 0.2, 0.2, 0.4, 0.8, 1.0, model.mesh.getFaceSet (name + "back"), name + "arm1");
    arm2 = model.mesh.select (0.0, 0.2, 0.0, 0.99, 0.8, 1.0, model.mesh.getFaceSet (name), name + "arm2");

    model.mesh.join (arm1, model.mesh, arm2);

}

if (file)
{
    if (fileName != null)
    {
        MeshUtil.writeGeometry (model.mesh, fileName);
    }
    else
    {
        System.out.println ("The filename may not be null!");
    }
    System.exit(0);
}
model.mesh.updateNormals ();
}

```

## C.5 Luggage Modeling Procedure

```

Model out Luggage (double height = 2.0, double width = 1.0, double length = 2.0, int numLegs = 4,
                    int legRows = 4, double divBy = 30.0, boolean file = false, String fileName = null)
{
    double cubeDim = height/divBy;
    double extrudeDim = cubeDim*2.0;

    model = #Cube (dimension = cubeDim, name = "luggage");

    String name = "luggage";

    FaceSet t = new FaceSet (name, model.mesh.faces);
    model.mesh.addFaceSet (t);
    VRVector direction;
    FaceSet cap = new FaceSet ();
    t = model.mesh.select (0.0, 1.0, 0.0, 1.0, 0.0, 0.1, model.mesh.getFaceSet (name), name);
    for (double i = extrudeDim; i <= length; i += extrudeDim)
    {
        cap = new FaceSet ();
        direction = VRVector.multiply (-extrudeDim, Z);
        #extrude (faceList = t, extrudeBy = direction, partName = name, capFaces = cap);
        t = cap;
    }

    t = model.mesh.select (0.0, 0.01, 0.0, 1.0, 0.0, 1.0, model.mesh.getFaceSet (name), name);

    for (double i = extrudeDim; i <= width; i += extrudeDim)
    {
        cap = new FaceSet ();
        direction = VRVector.multiply (-extrudeDim, X);
        #extrude (faceList = t, extrudeBy = direction, partName = name, capFaces = cap);
        t = cap;
    }

    FaceSet sides = model.mesh.select (0.0, 0.1, 0.99, 1.0, 0.0, 1.0,
                                       model.mesh.getFaceSet (name), "sides");
    sides.mergeFaceSets (model.mesh.select (0.9, 1.0, 0.99, 1.0, 0.0, 1.0,
                                             model.mesh.getFaceSet (name), "sides"));
    sides.mergeFaceSets (model.mesh.select (0.01, 0.99, 0.99, 1.0, 0.0, 0.1,
                                             model.mesh.getFaceSet (name), "sides"));
    sides.mergeFaceSets (model.mesh.select (0.01, 0.99, 0.99, 1.0, 0.9, 1.0,
                                             model.mesh.getFaceSet (name), "sides"));
    model.mesh.addFaceSet (sides);

    for (double i = extrudeDim; i < height; i += extrudeDim)
    {
        cap = new FaceSet ();
        direction = VRVector.multiply (extrudeDim, Y);
        #extrude (faceList = sides, extrudeBy = direction, partName = name, capFaces = cap);
        sides = cap;
    }

    t = model.mesh.select (0.0, 0.1, 0.99, 1.0, 0.0, 1.0, model.mesh.getFaceSet ("sides"), "lid");
    model.mesh.addFaceSet (t);

    for (double i = extrudeDim; i < width; i += extrudeDim)
    {
        cap = new FaceSet ();
        direction = VRVector.multiply (extrudeDim, Y);
        #extrude (faceList = t, extrudeBy = direction, partName = name, capFaces = cap);
        t = cap;
    }

    BoundingBox myBox = new BoundingBox (model.mesh.getFaceSet ("lid"));

    double minX = myBox.min[0];
    double minY = myBox.min[1];
    double minZ = myBox.min[2];
    double maxX = myBox.max[0];
    double maxY = myBox.max[1];

```

```

double maxZ = myBox.max[2];

VRPoint points [] = new VRPoint [ 3 ];

double offSet = 2.5;

points [0] = new VRPoint (minX, maxY, 0.0);
points [1] = new VRPoint (minX - offSet, (maxY + height)/2, 0.0);
points [2] = new VRPoint (minX, maxY - height, 0.0);*/

points [0] = new VRPoint (minX, maxY, 0.0);
points [1] = new VRPoint (minX - offSet*2, (maxY + height)/2, 0.0);
points [2] = new VRPoint (minX, minY, 0.0);

Curve lidCurve = new BezierSpline (points.length - 1, points);

points [0] = new VRPoint (minX, maxY, 0.0);
points [1] = new VRPoint (minX - offSet/4, (maxY + height)/2, 0.0);
points [2] = new VRPoint (minX + extrudeDim, maxY - height, 0.0);

Curve lidInnerCurve = new BezierSpline (points.length - 1, points);

FaceSet lid = model.mesh.select (0.0, 0.01, 0.0, 1.0, 0.0, 1.0,
                                model.mesh.getFaceSet ("lid"), "outerlid");

FaceSet innerLid = model.mesh.select (0.99, 1.0, 0.15, 0.85, 0.2, 0.8,
                                    model.mesh.getFaceSet ("lid"), "innerlid");

double quarter = (minZ + maxZ)/4.0;

offSet = 0.5;
FaceSet base = model.mesh.select (0.0, 1.0, 0.0, 0.01, 0.0, 1.0, model.mesh.getFaceSet (name), "base");
model.mesh.addFaceSet (base);

///

```

```

{
    t = model.mesh.select (xMin, xMax, 0.0, 1.0, zMin, zMax - overlap,

        model.mesh.getFaceSet ("rightbase"), "rightfoot" + i + j);

    model.mesh.addFaceSet (t);
    model = #Leg (numToes = numToes, threshold = threshold, height = 1.0, length = 0.5,
        width = 1.0, heads = 1.0, name = ("rightfoot" + i + j));

    t = model.mesh.select (xMin, xMax, 0.0, 1.0, zMin, zMax - overlap,

        model.mesh.getFaceSet ("leftbase"), "leftfoot" + i + j);

    model.mesh.addFaceSet (t);
    model = #Leg (numToes = numToes, threshold = threshold, height = 1.0, length = 0.5,
        width = 1.0, heads = 1.0, name = ("leftfoot" + i + j));

    zMin = zMax + overlap;
    zMax = zMax + offSetFoot;
}
zMin = 0.0;
zMax = offSetFoot;
xMin = xMax;
xMax = xMax + xRatio;
}

model.mesh.updateNormals ();

if (file)
{
    if (fileName != null)
    {
        MeshUtil.writeGeometry (model.mesh, fileName);
    }
    else
    {
        System.out.println ("The filename may not be null!");
    }
}
}

```



# Appendix D

## Code for Selected Procedural Modeling Tools

### D.1 Selection

```
// Returns a list of faces that fall within the bounds given.
public FaceSet select (double ratiominx, double ratiomaxx,
                      double ratiominy, double ratiomaxy,
                      double ratiominz, double ratiomaxz,
                      FaceSet currentSet, String selectionName)
{
    BoundingBox bb = new BoundingBox (currentSet);
    Vector selection = new Vector (0);
    if (currentSet != null)
    {
        for (int i = 0; i < currentSet.numFaces (); i++)
        {
            Face currFace = currentSet.getFace (i);
            boolean found = true;
            for (int j = 0; j < currFace.numVertices (); j++)
            {
                Vertex currVertex = (Vertex) currFace.vertices.elementAt (j);
                double x =
                    (currVertex.position.coord[0] - bb.min[0]) / (bb.max[0] -
                                                                bb.min[0]);
                double y =
                    (currVertex.position.coord[1] - bb.min[1]) / (bb.max[1] -
                                                                bb.min[1]);
                double z =
                    (currVertex.position.coord[2] - bb.min[2]) / (bb.max[2] -
                                                                bb.min[2]);
                if (!
                    ((x >= ratiominx) && (x <= ratiomaxx) && (y >= ratiominy)
                     && (y <= ratiomaxy) && (z >= ratiominz)
                     && (z <= ratiomaxz)))
                {
                    found = false;
                    break;
                }
            }
            if (found)
            {
                selection.addElement (currFace);
            }
        }
    }
}
```

```

    }
    FaceSet selectedSet = new FaceSet (selectionName, selection);
    return selectedSet;
}

```

## D.2 Curve Shaping

```

public void shapeToCurve (FaceSet faces, Curve curve, VRVector direction,
                          VRVector projectionDirection)
{
    //Ensure that we are using a normalized projection plane.

    projectionDirection = projectionDirection.normalize ();

    ///Start by projecting all of the control points onto our projection plane.

    ///This only needs to be done once per method call

    VRPoint projectedPoints[] = new VRPoint[curve.numberControlPoints ()];
    for (int i = 0; i < projectedPoints.length; i++)
    {
        projectedPoints[i] =
            projectPoint (curve.getControlPoint (i), projectionDirection);
    }

    ///Create a new curve that has already been projected onto our projection plane.
    Curve projectedCurve =

        new BezierSpline (projectedPoints.length - 1, projectedPoints);

    ///Project our direction vector onto our projection plane. This also only needs to

    ///be done once.
    direction = projectVector (direction, projectionDirection);

    direction = direction.normalize ();

    ///Start working with the faces in the faceset. To begin with, we are going to traverse
    ///each vertex of a face, and displace it. This can be extended to work on the face...
    ///Find all duplicate vertices and remove them. This way we are only displacing each vertex

    ///once

    Vector allVertices = new Vector (numVertices ());
    for (int i = 0; i < faces.numFaces (); i++)
    {
        Face currFace = faces.getFace (i);
        for (int j = 0; j < currFace.numVertices (); j++)
        {
            Vertex currVertex = currFace.getVertex (j);
            boolean found = false;
            for (int k = 0; k < allVertices.size (); k++)
            {
                Vertex current = (Vertex) allVertices.elementAt (k);
                if (currVertex == current)
                {
                    ///This means we have a duplicate entry
                    {
                        found = true;
                        break;
                    }
                }
            }
        }
    }
}

```

```

        }
    }
    if (!(found))
    {
        allVertices.addElement (currVertex);
    }
}

}

for (int j = 0; j < allVertices.size (); j++)
{
    Vertex currVertex = (Vertex) allVertices.elementAt (j);
    VRPoint projectedPoint =
        projectPoint (currVertex.position, projectionDirection);
    VRVector displacement =
        getDisplacement (projectedCurve, projectedPoint, direction,
            projectionDirection);
    currVertex.position = VRPoint.add (currVertex.position, displacement);
}
}

```

## D.3 Extrusion

```

public Mesh extrude (FaceSet faceList, VRVector extrudeBy, String partName, FaceSet capFaces)
{
    Vector allVertices = new Vector (0);
    FaceSet partSet = null;
    if (partName != null)
    {
        partSet = getFaceSet (partName);
        if (partSet == null)
            partSet = new FaceSet (partName, new Vector ());
    }
    for (int i = 0; i < faceList.numFaces (); i++)
    {
        Face currFace = faceList.getFace (i);
        for (int j = 0; j < currFace.numVertices (); j++)
        {
            Vertex currVertex = (Vertex) currFace.vertices.elementAt (j);
            boolean found = false;
            for (int k = 0; k < allVertices.size (); k++)
            {
                Vertex current = (Vertex) allVertices.elementAt (k);
                if (currVertex == current)
                {
                    ///This means we have a duplicate entry
                    {
                        found = true;
                        break;
                    }
                }
            }
            if (!(found))
            {
                allVertices.addElement (currVertex);
            }
        }
        ///end inner for
    }
    ///end outer for
    ///Keep track of the new vertices added
    Vertex newVertices[] = new Vertex[allVertices.size ()];
    ///Go through each vertex and extrude it
    for (int i = 0; i < allVertices.size (); i++)
    {
        Vertex currVertex = (Vertex) allVertices.elementAt (i);
        ///Extrude the vertex
        VRPoint newVertex = VRPoint.add (currVertex.position, extrudeBy);
        newVertices[i] = addVertex (newVertex);
    }
}

```

```

    }
    //end for
    ///Go through each face looking for boundary edges from which to extrude new faces
    for (int i = 0; i < faceList.numFaces (); i++)
    {
        Face currFace = faceList.getFace (i);
        ///Keep track of vertex indices in the old vertex list and the new one
        int indices[] = new int[currFace.numVertices ()];
        ///For each vertex in the current face, find its corresponding position in allVertices,
        ///and use these to construct the new faces
        for (int z = 0; z < currFace.numVertices (); z++)
        {
            Vertex currVertex = (Vertex) currFace.vertices.elementAt (z);
            for (int g = 0; g < allVertices.size (); g++)
            {
                Vertex current = (Vertex) allVertices.elementAt (g);
                if (currVertex == current)
                {
                    {
                        indices[z] = g;
                        break; //break if we find the vertex, otherwise we are wasting time
                    }
                }
            }
        }
        ///Traverse all the edges to find which ones are boundaries.
        ///The boundary edges will have faces extruded from them
        for (int j = 0; j < currFace.numEdges (); j++)
        {
            Edge currEdge = (Edge) currFace.edges.elementAt (j);
            int occurrences = 1;
            for (int k = 0; k < currEdge.numFaces (); k++)
            {
                Face edgeFace = (Face) currEdge.faces.elementAt (k);
                if (edgeFace == currFace)
                {
                    {
                        continue;
                    }
                }
                else
                {
                    {
                        for (int l = 0; l < faceList.numFaces (); l++)
                        {
                            {
                                Face currentFace = faceList.getFace (l);
                                if (currentFace == edgeFace)
                                {
                                    {
                                        occurrences++;
                                        break;
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
        if (occurrences < 2)
        {
            ///If the occurrence of this edge is < 2, then we are dealing with a boundary edge
            {
                Vector t = new Vector (0);
                t.addElement (allVertices.
                    elementAt (indices
                        [(j +
                            1) % currFace.numVertices ()]));
                t.addElement (newVertices
                    [indices[(j + 1) % currFace.numVertices ()]]);
                t.addElement (newVertices[indices[j]]);
                t.addElement (allVertices.elementAt (indices[j]));
                Face temp = addFace (t);
                temp.labels = (Vector) currFace.labels.clone ();
                if (partSet != null)
                {
                    {
                        partSet.addFace (temp);
                    }
                }
            }
        }
        //end edge for
        Vector t = new Vector (0);
        for (int k = 0; k < currFace.numVertices (); k++)
        {

```

```

        t.addElement (newVertices[indices[k]]);
    }
    Face temp = addFace (t);
    temp.labels = (Vector) currFace.labels.clone ();
    if (partSet != null)
    {
        partSet.addFace (temp);
    }
    if (capFaces != null)
    {
        capFaces.addFace (temp);
    }
    } //end outer for
    ///Delete the original faces
    for (int i = 0; i < faceList.numFaces (); i++)
    {
        Face delFace = faceList.getFace (i);
        deleteFace (delFace);
    }
    for (int i = 0; i < numFaceSets (); i++)
    {
        FaceSet temp = getFaceSet (i);
        temp.updateFaceSet (temp.getID (), faces);
    }
    if (partSet != null)
    {
        addFaceSet (partSet);
    }
    return this;
}

```

# Bibliography

Anthony A. Apodaca and Larry Gritz. *Advanced Renderman: Creating CGI for Motion Pictures*. Academic Press, San Diego, California, 2000.

Autodesk, Inc. *3D Studio Max Release 3, Reference Volume I*. Autodesk, Inc., 1999a.

Autodesk, Inc. *3D Studio Max Release 3, Reference Volume II*. Autodesk, Inc., 1999b.

Shaun Bangay. Experiences in porting a virtual reality system to java. In *AFRIGRAPH 2001: Proceedings of the 1st international conference on Computer graphics, virtual reality and visualisation*, pages 33–37, 2001.

Shaun Bangay and Chantelle Morkel. Graph matching with subdivision surfaces for texture synthesis on surfaces. In *AFRIGRAPH 2006: Proceedings of the 4th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa*. ACM Press, 2006.

Shaun Bangay, James Gain, Greg Watkins, and Kevan Watkins. Rhover; building the second generation of parallel/distributed virtual reality systems. In *First Eurographics Workshop on Parallel Graphics & Visualisation*, pages 991–1000, 1996.

Judy Bayne, Grahame Fuller, Erik Goulet, Edna Kruger, Luc Langevin, Jamal Rahal, and Gino Vincelli. Softimage|XSI version 4.0 tutorials 1, 2004.

Jeremy S. De Bonet. Multiresolution sampling procedure for analysis and synthesis of texture images. In *Proceedings of ACM SIGGRAPH 1997*, pages 361–368. ACM Press/Addison-Wesley Publishing Co., 1997.

Steven Brooks, Susan-Belle Furguson, Lisa Ford, Claude Macri, Susan Park, Diane Ramey, and Linda Rose. *Mel Version 4*. Alias | Wavefront, 2001.

- Richard Cartwright, Valery Adzhiev, Alexander A. Pasko, Yuichiro Goto, and Tosiyasu L. Kunii. Web-based shape modeling with hyperfun. *IEEE Computer Graphics and Applications*, 25: 60–69, 2005.
- Yanyun Chen, Yingqing Xu, Baining Guo, and Heung-Yeung Shum. Modeling and rendering of realistic feathers. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 630–636. ACM Press, 2002.
- Kwang-Jin Choi and Hyeong-Seok Ko. Stable but responsive cloth. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 604–611, San Antonio, Texas, 2002. ACM Press.
- Calgari Corporation. *trueSpace 5 Users's Guide*. Calgari Corporation, California, 2001.
- Barbara Cutler, Julie Dorsey, Leonard McMillan, Matthias Muller, and Robert Jagnow. A procedural approach to authoring solid models. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 302–311. ACM Press, 2002.
- Barbara M. Cutler. *Procedural Authoring of Solid Models*. PhD thesis, Massachusetts Institute of Technology, 2003.
- Tony DeRose, Michael Kass, and Tien Truong. Subdivision surfaces in character animation. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 85–94. ACM Press, July 1998.
- Oliver Deussen, Pat Hanrahan, Bernd Lintermann, Radomir Mech, Matt Pharr, and Przemyslaw Prusinkiewicz. Realistic modeling and rendering of plant ecosystems. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 275–286. ACM Press, 1998.
- Charles Donnelly and Richard Stallman. Bison (the yacc-compatible parser generator), 2005. URL <http://www.gnu.org/software/bison/manual/>. [Last Accessed: 10 November 2005].
- David S. Ebert, F. Kenton Musgrave, Darwyn Peachy, Ken Perlin, and Steven Worley. *Texturing and Modeling*. Morgan Kaufmann, third edition, 2002.
- David S. Ebert, F. Kenton Musgrave, Darwyn Peachy, Ken Perlin, and Steven Worley. *Texturing and Modeling*. Morgan Kaufmann Publishers, San Francisco, third edition, 2003.

- Jens Feder. *Fractals*. Plenum Press, New York, April 1989.
- Bill Fleming. *Mastering 3D Graphics: Digital Botany and Creepy Insects*. John Wiley and Sons, Inc, New York, 2000.
- Bill Fleming and Richard H. Schrand. *3D Creature Workshop*. Charles River Media, Inc., Hingham, Massachusetts, 2nd edition, 2001.
- Alain Fournier, Don Fussell, and Loren Carpenter. Computer rendering of stochastic models. *Communications of the ACM*, 25(6):371–384, June 1982.
- Thomas Funkhouser, Michael Kazhdan, Philip Shilane, Patrick Min, William Kiefer, Ayellet Tal, Szymon Rusinkiewicz, and David Dobkin. Modeling by example. In *ACM Trans. Graph.*, pages 652–663. ACM Press, 2004.
- Kevin R. Glass, Chantelle Morkel, and Shaun D. Bangay. Duplicating road patterns in south african informal settlements using procedural techniques. In *AFRIGRAPH 2006: Proceedings of the 4th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa*. ACM Press, 2006.
- Michael T. Goodrich and Roberto Tamassia. *Data Structures and Algorithms in Java*. John Wiley & Sons, Inc, second edition, 2001.
- Mark Green and Hanqiu Sun. A language and system for procedural modeling and motion. *IEEE Computer Graphics and Applications*, 8(6):52–64, November 1988.
- Stefan Greuter, Jeremy Parker, Nigel Stewart, and Geoff Leach. Real-time procedural generation of ‘pseudo infinite’ cities. In *GRAPHITE '03: Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, pages 87–94. ACM Press, 2003.
- Tovi Grossman, Ravin Balakrishnan, Gordon Kurtenbach, George Fitzmaurice, Azam Khan, and Bill Buxton. Creating principal 3d curves with digital tape drawing. In *CHI 2002: Proceedings of the SIGCHI conference on Human factors in computing systems*, volume 4. ACM Press, 2002.
- John C. Hart. On efficiently representing procedural geometry, 1994. URL <http://graphics.cs.uiuc.edu/~jch/papers/>. [Accessed: 18 January 2005].



- Carl Hultquist, James Gain, and David Cairns. Affective scene generation. In *AFRIGRAPH 2006: Proceedings of the 4th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa*, 2006.
- Kenneth I. Joy, Justin Legakis, and Ron MacCracken. Hierarchical approximation and geometric methods for scientific visualization. In Gerald Farin, Hans Hagen, and Bernd Hamann, editors, *Data Structures for Multiresolution Representation of Unstructured Meshes*. Springer-Verlag, Heidelberg, Germany, 2002.
- Yotto Koga, Michael Svihura, Edna Kruger, and Luc Langevin. *SOFTIMAGE:BEHAVIOR Version 2.0 User Guide*. SOFTIMAGE AVID, 2004.
- Justin Legakis, Julie Dorsey, and Steven Gortler. Feature-based cellular texturing for architectural models. In *SIGGRAPH 2001: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 309–316, New York, NY, USA, 2001. ACM Press.
- T. Lewis and M.W. Jones. A system for the non-linear modelling of deformable procedural shapes. *Journal of WSCG (Winter School of Computer Graphics)*, 12(1-3), February 2004.
- Guiqing Li, Weiyin Ma, and Hujun Bao.  $\sqrt{2}$  subdivision for quadrilateral meshes. *The Visual Computer*, 20(2):180–198, May 2004.
- Andrew Loomis. *Figure Drawing, For All It's Worth*. Viking Press, New York, 1943.
- MakeHuman Team. Makehuman project, 2005. URL <http://www.makehuman.org>. [Last Accessed: 29 September 2005].
- Robert Marshall, Rodger Wilson, and Wayne Carlson. Procedure models for generating three-dimensional terrain. In *SIGGRAPH 1980: Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, pages 154–162, New York, NY, USA, 1980. ACM Press.
- Stephen F. May, Wayne E. Carlson, Flip Phillips, and Ferdi Scheepers. Al: A language for procedural modeling and animation. Technical report, Ohio State University and CSIR, 1996. URL [url{http://accad.osu.edu/~smay/al.pdf}](http://accad.osu.edu/~smay/al.pdf).
- Chantelle Morkel and Shaun Bangay. Procedural modeling facilities for hierarchical object generation. In *AFRIGRAPH 2006: Proceedings of the 4th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa*. ACM Press, 2006.

- F. K. Musgrave, C. E. Kolb, and R. S. Mace. The synthesis and rendering of eroded fractal terrains. In *Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, pages 41–50. ACM Press, 1989.
- NewTek. *Lightwave 3D 7 Reference Guide*. NewTek, San Antonio, Texas, 2001.
- Marc Olano and Anselmo Lastra. A shading language on graphics hardware: the pixelflow shading system. In *SIGGRAPH 1998: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 159–168, New York, NY, USA, 1998. ACM Press.
- M. Oshita and A. Makinouchi. Real-time cloth simulation with sparse particles and curved faces. In *Proceedings of Computer Animation 2001*, pages 220–227, November 2001.
- Yoav I. H. Parish and Pascal Muller. Procedural modeling of cities. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 301–308. ACM Press, 2001.
- K. Perlin and E. M. Hoffert. Hypertexture. In *Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, pages 253–262. ACM Press, 1989.
- Ken Perlin. An image synthesizer. In *Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 287–296. ACM Press, 1985.
- Ken Perlin. Improved noise reference implementation, 2002. URL <http://mrl.nyu.edu/~perlin/noise/>. [Last Accessed: 28 June 2005].
- Ken Perlin and Athomas Goldberg. Improv: a system for scripting interactive actors in virtual worlds. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 205–216. ACM Press, 1996.
- Jorg Peters and Ulrich Reif. The simplest subdivision scheme for smoothing polyhedra. *ACM Transactions on Graphics*, 16(4):420–431, 1997.
- Terry Pratchett. *The Colour of Magic*. Corgi Adult, 1985.
- Przemyslaw Prusinkiewicz. Simulation modeling of plants and plant ecosystems. *Communications of the ACM*, 43(7):84–93, 2000.
- Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag, New York, 1990.

- Xuejie Qin and Yee-Hong Yang. Estimating parameters for procedural texturing by genetic algorithms. *Graphical Models*, 64(1):19–39, 2002.
- Peter Ratner. *3-D Human Modeling and Animation*. Wiley, second edition, 2003.
- W. T. Reeves. Particle systems - a technique for modeling a class of fuzzy objects. *ACM Transactions on Graphics*, 2(2):91–108, 1983.
- William T. Reeves and Ricki Blau. Approximate and probabilistic algorithms for shading and rendering structured particle systems. In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 313–322. ACM Press, 1985.
- William T. Reeves, Eben F. Ostby, and Samuel J. Leffler. The Menv modelling and animation environment. *Journal of Visualization and Computer Animation*, 1(1):33–40, August 1990.
- Ton Roosendaal and Stefano Selleri, editors. *The Official Blender 2.3 Guide : free 3D creation suite for modeling, animation and rendering*. No Starch Press, USA, 2004.
- Ton Roosendaal, Stefano Selleri, and Blender Documentation Team. Blender documentation volume i - user guide, 2004. URL <http://download.blender.org/documentation/htmlI/>. [Accessed: 2 January 2005].
- Olli-Pekka Saastamoinen. Modeling a human body, 1999. URL <http://www.dlc.fi/~ops/tutorial/human.htm>. [Last Accessed: 25 July 2005].
- Ferdi Scheepers, Richard E. Parent, Wayne E. Carlson, and Stephen F. May. Anatomy-based modeling of the human musculature. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 163–172, 1997.
- Karl Sims. Artificial evolution for computer graphics. In *SIGGRAPH 1991: Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, pages 319–328, New York, NY, USA, 1991. ACM Press.
- Karan Singh and Eugene Fiume. Wires: a geometric deformation technique. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 405–414. ACM Press, 1998.

- Jing Sun, Xiaobo Yu, George Baciuc, and Mark Green. Template-based generation of road networks for virtual city modeling. In *Proceedings of the ACM symposium on Virtual reality software and technology*, pages 33–40. ACM Press, 2002.
- Richard Szeliski and David Tonnesen. Surface modeling with oriented particle systems. In *SIGGRAPH 1992: Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pages 185–194, New York, NY, USA, 1992. ACM Press.
- Nadia Magnenat Thalmann and Daniel Thalmann. Computer animation. *ACM Computing Surveys*, 1996.
- Greg Turk. Generating textures on arbitrary surfaces using reaction-diffusion. In *Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, 1991.
- Greg Turk. Texture synthesis on surfaces. In *Proceedings of ACM SIGGRAPH 2001*, pages 347–354. ACM Press, August 2001.
- Steve Upstill. *The Renderman Companion: A Programmer's Guide to Realistic Computer Graphics*. Addison-Wesley, Boston, 1993.
- Luiz Velho and Denis Zorin. 4-8 subdivision. *Computer-Aided Geometric Design*, 18(5):397–427, 2001.
- Luiz Velho, Ken Perlin, Lexing Ying, and Henning Biermann. Procedural shape synthesis on subdivision surfaces. In *Proceedings of the 14th Brazilian Symposium on Computer Graphics and Image Processing (SIBGRAPI)*, Florianopolis, Brazil, October 2001.
- Li-Yi Wei and Marc Levoy. Fast texture synthesis using tree-structured vector quantization. In *Proceedings of ACM SIGGRAPH 2000*, pages 479–488. ACM Press/Addison-Wesley Publishing Co., 2000.
- Andrew Witkin and Michael Kass. Reaction-diffusion textures. In *Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, pages 299–308, 1991.
- Tomasz Zaniewski and Shaun Bangay. Simulation and visualization of fire using extended lindenmayer systems. In *AFRIGRAPH '03: Proceedings of the 2nd international conference on Computer graphics, virtual Reality, visualisation and interaction in Africa*, pages 39–48, Cape Town, South Africa, 2003. ACM Press.

Denis Zorin and Peter Schroder. Subdivision for modeling and animation, 2000. URL <http://mrl.nyu.edu/publications/subdiv-course2000/>. SIGGRAPH 2000 course notes, [Last Accessed: 21 February 2005].