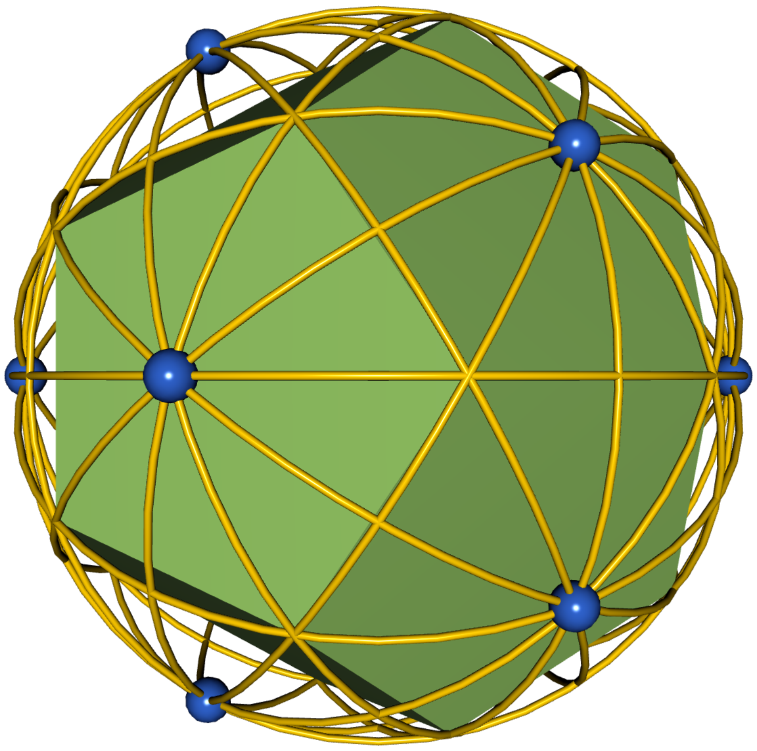
CHALMERS
UNIVERSITY OF TECHNOLOGY



Procedural Modeling and Animation with Quaternions

Using design grammars for procedural shape synthesis

Master's thesis in Computer Science

MATTIAS ANDERSSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2015

Master's thesis 2015:06

Procedural Modeling and Animation with Quaternions

Using design grammars for procedural shape synthesis

MATTIAS ANDERSSON

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden 2015

Procedural Modeling and Animation with Quaternions
Using design grammars for procedural shape synthesis
MATTIAS ANDERSSON

© MATTIAS ANDERSSON, 2015.

Supervisor: Ulf Assarsson

Master's Thesis 2015:06
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Sweden
Telephone +46 31 772 1000

Cover: Dodecahedron inscribed within the Coxeter complex.

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2015

Procedural Modeling and Animation with Quaternions
Using model design grammars for procedural shape synthesis
MATTIAS ANDERSSON
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

This master's thesis investigates a novel approach to procedural modeling using shape grammars. Model Design Grammars (MDG), a derivative of Context-Free Design Grammars (CFDG), are presented as a powerful tool for constructing three-dimensional models. Part of the thesis is about how to represent transforms, such as rotations, reflections and translations using quaternions. We study the use of production rules to generate hierarchical procedural models and we add the notion of compound transformations for animating objects in a bone hierarchy. We show how the same idea can be used as an additional tool for procedural modeling. We show that there exists an equivalent representation of a 0L-system as an MDG grammar.

Keywords: shape grammars, L-systems, procedural modeling, skeletal animation, quaternions, transformations, CFDG.

Acknowledgements

I wish to thank my supervisor, Ulf Assarsson, for proofreading, giving advice and good feedback on my thesis work.
Thanks to Anders Berg, for accepting the role as an opponent — it should be noted that he has prepared well, with more than a hundred games as a Wordfeud opponent.
One person that deserves special credits is Michael Hansen, who introduced me to context-free design grammars in the first place and who has always been a great friend with a lot of creative ideas.
Thanks to Jon Lennart Aasenden and Eric Grange for creating the development tool Smart Mobile Studio [2], which has been crucial for the implementation of the MDG language and for the development of the SaaS online service OnlineModeler.com [4].
I wish to thank my boss, Maral Alaghi, for giving me the opportunity to finish this thesis work, despite my many other duties.
Thanks to Johan Lodin for many interesting conversations about programming and for his refreshing insights about functional programming.
Finally, I would also like to thank Ole-Martin Christensen, Vladimir Conde, Bojana Dukic, Hans Brasch, Donal Murtagh, Joakim Möller, Nima Jamaly, Srikanth Gopal, Atul Yadav and Patrik Rynell for great friendship and many interesting discussions throughout the years.

Mattias Andersson, Gothenburg, June 2015

Contents

List of Figures

1 Introduction

This thesis will present a novel approach to procedural shape synthesis based on the use of quaternions and shape grammars. We will examine previous work within three seemingly distinct fields — shape grammars, quaternions and procedural modeling — and we will show how these fields can be used in conjunction to provide a powerful syntactic tool for describing a wide variety of 3D objects.
The main contribution of this report will be to highlight why quaternions is a useful tool in the context of procedural modeling and why our implementation of shape grammars, which we have chosen to term Model Design Grammars (MDG), provide a compact representation of such models.
Moreover, we will look at the theory of rotations and reflections and we will demonstrate how rigid transformations can be used to build complex geometrical structures from a set of predefined rules.
We will look at how common techniques used in modeling, such as extrusions and surface revolutions can be easily constructed by combining special profile rules with curve sweeping operations. Additionally, we will examine how to efficiently evaluate grammar rules and what selection criteria to use for our rule selection. We will study the topic of skeletal animation and we will show how quaternions can be used to describe the compound transformations of hierarchical skeletons.

1.1 Motivation

Procedural modeling is a common technique employed by both animators and designers when constructing three-dimensional objects. Oftentimes, the procedural generator has been hardcoded within the modeling software itself and is exposed only as a set of adjustable parameters for the designer. This thesis emphasizes a different approach, where the designer needs to provide a syntactic description of the model, before it is generated. This approach gives a greater freedom in representing a large variety of objects. This approach also encourages a scientific understanding of geometry, where the designer can explore many different shapes and geometries, by making small adjustments to the model descriptions.
There are many different scientific fields that would benefit from a compact syntactical representation of procedural models. In physics, chemistry and biology, it is of great interest to be able to find different representations of molecular structures and crystal lattice models. In architecture, it is essential to be able to explore new designs and to be able to represent symmetries and geometries in a simple way [32]. Mathematical descriptions of geometry often provide a great aesthetic value, and

this is can be used for more efficient development of designs in the jewelry industry, where conventional modelmaking is often a major bottleneck [48].
Computer games could benefit from procedural models for the same reasons as any of the above fields, but here we also have the additional benefit of being able to save disk space by representing complex geometries with simple formulas.
The most common way to represent transformations in Euclidean space is probably by using transform matrices. What are quaternions and why would we want to use them? There are many reasons why quaternions may be a better representation than matrices. The following is a list of some of the benefits of using quaternions (the list is partially extracted from the book Visualizing Quaternions by Andrew J Hanson [21]):

Shape A unit quaternion represents a point on the hypersphere $SO(3)$.

Metric Provides a meaningful metric to compare and understand different orientations.

Interpolability It is possible to perform smooth interpolation between two orientations.

Gimbal lock Quaternions avoid the problem of gimbal lock that is present with Euler angles.

Memory A quaterion can represent a rotation using only 4 numbers, whereas a 3×3 rotation matrix requires 9 numbers.

Spinors Unlike rotation matrices, quaternions provide a way to represent *spinors*, making it possible to distinguish between 360° and a 720° rotations.

1.2 Problem statement

The thesis project was conceived from a few independent observations made by the author:

- Shape grammars is a powerful syntactic tool for representing many different geometries[1].
- With the notable exception of L-systems, there is little research on other types of shape grammars for procedural modeling in $\mathbb{R}^3$ space.
- Recursive evaluation of production rules gives additional expressive power that is not available in other modeling languages.

Designing a new shape grammar will involve finding an answer to a few substantial questions:

- What modifications and additions would make this grammar different from shape grammars in $\mathbb{R}^2$ space, such as CFDG?

[1]This observation was made by studying shape grammars for representing two-dimensional vector objects, see e.g. [15]

- What set of geometric transformations would be useful in $\mathbb{R}^3$ space?
- What would be a suitable representation for vertices and normals?
- Could shape grammars be used to describe camera paths and compound transformations?
- Could shape grammars provide a language for animation?
- What criteria would be used for rule selection and for terminating recursions?

2

Previous Work

2.1 Grammars and modeling languages

Noam Chomsky is often credited as the father of modern linguistics. He was a pioneer in the theory of syntax and in the classification of formal grammars [10, 11]. Chomsky classified different grammars according to their expressive power. Context Free Grammars (CFG) was one of the formal grammars described by Chomsky. These grammars have also been important in the development of procedural shape synthesis, where they can be used to describe a large number of procedural objects.

2.1.1 Turtle graphics

Seymour Papert developed the LOGO programming language in 1967 [17] and he also invented the notion of a moving *turtle* that would change location and orientation through a set of commands. The `FORWARD` and `BACKWARD` commands would advance the turtle a number of units, optionally leaving a trail behind it. The `LEFT` and `RIGHT` commands would change the orientation by a specified angle. Papert proposed turtle graphics as a tool within education to teach young children about mathematics [35].

2.1.2 Lindenmayer systems

L-systems and parallel rewriting grammars (PRG) were introduced by Lindenmayer in 1968 [26]. These systems provided a simple formalism for constructing visually complex imagery of biologically developing systems and plant-like organisms [38].
0L-systems represent the most basic instances of L-systems, where production rules apply to each symbol independently of its neighbors in the symbol string. This system is said to be context-free, since the selection process does not depend on the context. A 2L-system[1] will take into account the symbol before and the symbol after each symbol and use that as a condition for rule expansion. This system can then be said to be context-sensitive.
The final output of an L-system is a string of symbols, that is later evaluated in a separate pass to build the procedural model. With bracketed L-systems, bracket parentheses are used to denote that the current transform space should be pushed or popped from a stack. This means that it is possible to return to a prior state by

[1]A 1L-system will use only one more symbol as a condition – either to the left or to the right of the current symbol.

encapsulating a subtree within brackets. The tree-like fractals generated by these systems are sometimes referred to as *graftals* [45].
Przemyslaw Prusinkiewicz has been a major contributor in the research about L-systems [39, 40, 8]. His research about self-organizing growth processes of trees [34] demonstrates how L-systems can be used to achieve a high level of realism. The TreeSketch system [27], one application of this research, explores the concept of using a procedural brush for drawing trees.

2.1.3 Shape grammars

James Gips and George Stiny are generally credited as having introduced the concept of shape grammars in 1971 [47, 18, 19]. Shape grammars built upon the grammar definitions investigated by Chomsky and added a set of rules for substituting symbols with a shape. The grammars presented by the authors were very closely related to L-systems, in terms of representing transforms and shapes as strings of symbols.

2.1.4 GEOMED

An early example of using computers for procedural modeling is the GEOMED software, developed by Bruce Baumgart in 1974 [6, 7]. GEOMED provided a compact notation for many complex geometric objects, including polyhedra. It supported features such as curve sweeps along a path using a profile curve and advanced geometric transformations. While the software was interactive, updating a push-down stack when commands were entered, the notation of these commands formed a modeling language that could probably be classified as a shape grammar. Baumgart hardly cited any references in his papers, which makes it difficult to tell what were the main influences of his work (possibly because this was also part of a classified ARPA research program.) Unmistakingly, this work was highly original and even by today's standards it is interesting to look at some of the aspects of his implementation.

2.1.5 ASAS

An important contribution in the domain of modeling languages is the Actor/Scriptor Animation System (ASAS), introduced by Craig Reynold in 1982 [42]. ASAS was implemented in LISP and was inspired by partly by formal natural language described by Terry Winograd [49]. There were several generations of the ASAS system and it was developed into a very versatile tool for both modeling and animation (it was also used in many TV productions.) Reynolds did not explicitly make any references to shape grammars, but the language should definitely be considered one of the precursors to MDG in that it allowed similar recursive definitions of transforms.

2.1.6 GENMOD

Snyder et al. rigorously define the mathematical framework needed for generative modeling [46]. The authors examine concepts such as *generators* and *manifolds*. Advanced methods for curve sweeping are investigated, where curves are used as

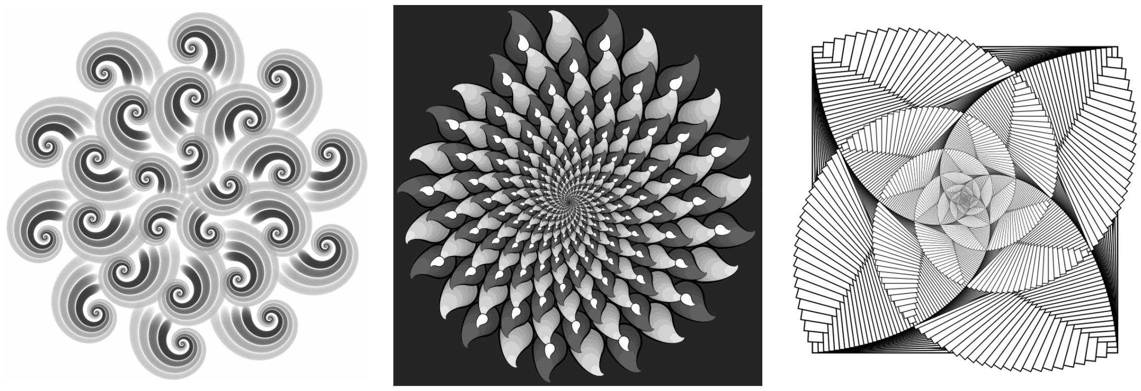


Figure 2.1: Artwork generated with Context Free.

generators to define profiles and cross sections of objects. Additionally it is demonstrated how *constructive planar geometry* (CPG), can be used to build complex geometry, such as the tip of a screwdriver. The GENMOD system incorporates many different symbolic operators, such as differentiation, integration and vector algebra operations.

2.1.7 AL

May et al. used Scheme (one of the main LISP dialects) to implement the ASAS derivative language AL [31]. May introduced the notion of *articulation functions* in animation systems. An articulation function is described as a "time-dependent function that will interactively animate parameters and variables within the model".

2.1.8 CGA

CGA is a shape grammar developed by Müller et al. for the purpose of generating large city models [33]. It is based on the parallel rewriting grammars introduced in the L-systems, but it is different in one important aspect – rather than using rules for rewriting strings, this grammar will rewrite the shapes themselves (i.e., one shape is replaced by another.) Additionally, the CGA shape rules were derived by studying domain-specific concepts in architecture.

2.1.9 CFDG

Context Free Design Grammars (CFDG) were introduced by Chris Coyne in 2005 and has since been popularized by the open-source software project Context Free [15].
There is a fairly limited number of publications that have examined the potential benefits of using CFDG as tool for procedural modeling. Saunders and Grace [43] examined how CFDG can be used as a tool in the education of design students in order to improve their understanding of generative processes and evolutionary design. The authors emphasize that design students do not generally get the same level of practice with programming languages as computer science students do and that it is therefore useful to be able to present a domain-specific design language

that does not add too many new concepts and abstractions. An engine for creating CFDG grammars using genetic algorithms was implemented in Processing[2].
Machado and Nunes [28, 29] investigated how CFDG could be used as a tool for evolutionary art, where genetic algorithms would transform a set of different grammars using mutation and crossover operators.

2.1.10 StructureSynth

StructureSynth, a software developed by Mikael Christensen in 2009 [13, 12], incorporates a CFDG derivative language for threee-dimensional modeling that is probably the closest relative to the MDG modeling language. The author introduces the notion of *rule retirement* and *substitutions*, which is similar to the rule selection mechanisms using recursion depth presented in this thesis.
Christensen also highlights the benefits of using CFDG-based systems for constrained systems and generative art. Design grammars are restricted in what kind of objects they are able to encode, unlike most procedural languages and hence there is a constrained *exploration space* that can be exploited in order to construct new models by making small alternations to the grammars and to the model parameters.

2.1.11 Fugu

Fugu is a modeling system based on the Lua scripting language [37]. The implementation can be used for modeling, manipulation and animation and it is capable of generating complex organic models. The system encourages both rapid prototyping and experimentation through a set of flexible 3D mesh operations.

2.2 Quaternions

Quaternions were conceived by William Rowan Hamilton already in 1843 [20]. Hamilton was a strong proponent of using quaternions as the standard algebraic entity to encode vectors in Euclidean $\mathbb{R}^3$ space and for describing their algebraic operations [21]. In fact, vector algebra operations, such as the cross product and the dot product were discovered only after analyzing the properties of quaternions. Perhaps the most important quaternion property, namely that of representing orientations in $\mathbb{R}^3$ space, was demonstrated by Hamilton's contemporary, Arthur Cayley, in 1845 [9].
Clifford algebras provide a generalization of geometrical transformations in Euclidean $\mathbb{R}^n$ space, in which quaternions constitute a special case for $n = 3$ [5, 23].
H.S.M. Coxeter observed that a quaternion rotation can in fact be represented as the product of two reflections [14].
Shoemake offers a strong argument for using quaternions for camera trajectories in computer animation [44]. Why do so many use Euler angles, despite their many disadvantages? Shoemake argues that this is due to quaternions being a subject that is generally introduced after the mathematics of Euler angles in higher education.

[2]Processing is a software tailored specifically for education within media arts [41].

The use of splining quaternions in keyframe animation systems has also been studied by Duff [16] and by Pletinckx [36].
A major application area for quaternions is within the aerospace industry, where it has been used in satellite navigation systems for many years [25].
Some recent developments within the fields of visualization and quaternions is the use of *quaternion maps* for representing orientation frames within protein molecular structures [22]. Quaternions also provide a useful similarity metric for comparison of protein structures.

3 Theory

We present a theoretical basis for understanding the concepts of our implementation. We will be concerned mostly with the aspects of transformations in Euclidean $\mathbb{R}^3$ space. The mathematical component for representing transformations in our implementation is quaternions. This is one alternative that provides certain advantages and disadvantages compared to other representations.

3.1 Grammars

3.1.1 Context-free grammars

Chomsky defined a context-free grammar as a 4-tuple (V, Σ, R, S), where

1. V is a set of non-terminal symbols,
2. Σ is a set of terminal symbols,
3. R is a set of production rules that map from V to $(V \cup \Sigma)$,
4. S is the initial symbol.
5. a grammar G generates a language L[1].

Each production rule is defined as a symbol $v \in V$ that maps to a string of symbols $s_1, \ldots, s_n$, such that $s_k \in (V \cup \Sigma), \forall s_k$.
Both CFDG and the 0L-systems are examples of context free languages. The production rules are derived from exactly one non-terminal symbol. The so called IL-systems are context-sensitive, since a production rule is subject to the condition of neighboring symbols in the input string. MDG is a completely context-free language, meaning that any production rule can map only from one non-terminal symbol. However, MDG also includes the notion of conflicting rules, such that there are multiple production rules that maps from the same non-terminal. This is in fact one of the central aspects of this language, and it allows us to implement very specific rule selection mechanisms.

[1]A language L is said to be context-free, if there exists a G such that $L = L(G)$.

3.2 Rotations

A few common ways of representing rotations and orientations are outlined. This is an essential component in rigid body dynamics, where the orientation will determine the motion of a rigid body.

3.2.1 Euler angles

Probably one of the most intuitive ways of representing rotations, in Euclidean geometry, is through the use of *Euler angles*. Three angles, α, β and γ are used to denote the rotation about three different axes.
Additionally, there is a distinction between *classic Euler angles* and *Tait-Bryan* or *Cardan* angles[2]. It is important to realize that for Euclidean geometries in $\mathbb{R}^3$ or higher, rotations are not commutative and it makes a difference in which order you rotate about the axes.

3.2.2 Axis-angle

Another convenient way to represent rotations, is through the axis-angle representation, which consists of a vector a and a rotation θ about the same vector:

$$(\text{axis}, \text{angle}) = \left(\begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix}, \theta \right) \tag{3.1}$$

3.2.3 Matrices

Matrices is a common way of representing rotations in computer graphics.
We can use separate matrices in order to describe a rotation around each axis in $\mathbb{R}^3$:

$$R(\phi, \theta, \psi) = \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{bmatrix}}_{R_x(\phi)} \cdot \underbrace{\begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}}_{R_y(\theta)} \cdot \underbrace{\begin{bmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{R_z(\psi)} \tag{3.2}$$

Here ϕ, θ and ψ corresponds to the Tait-Bryan angle representation of a rotation. Similarly, Euler angles may be used, by replacing the third rotation matrix with $R_x(\psi)$.

3.2.4 Quaternions

In quaternion algebra, the vector $\mathbf{v}$ is rotated by the quaterion $\mathbf{q}$ through the formula

$$\mathbf{v}' = \mathbf{q}\mathbf{v}\mathbf{q}^*. \tag{3.3}$$

[2] *Classic Euler angles* refers to angles where α and γ represents a rotation about the same axis, while *Tait-Bryan* or *Cardan* angles refers to rotations about three distinct axes (also the same thing as *yaw*, *pitch* and *roll*.)

3.3 Quaternions

We define a *quaternion* $\mathbf{q}$ as a 4-tuple (q_x, q_y, q_z, q_w) with the following properties:

$$\mathbf{q} = (q_x, q_y, q_z, q_w) = iq_x + jq_y + kq_z + q_w \tag{3.4}$$
$$i^2 = j^2 = k^2 = -1, jk = -kj = -i, ki = -ik = j, ij = -ji = k. \tag{3.5}$$

It is usually convenient to divide the quaternion into its vector part $\mathbf{q_v}$ and its scalar part q_w:

$$\mathbf{q}_v = (q_x, q_y, q_w) = iq_x + jq_y + kq_z \tag{3.6}$$
$$\mathbf{q} = (\mathbf{q}_v, q_w) = iq_x + jq_y + kq_z + q_w = \mathbf{q}_v + q_w \tag{3.7}$$

Multiplication between two quaternions $\mathbf{q}$ and $\mathbf{r}$ is defined as:

$$\begin{aligned}\mathbf{qr} =& (iq_x + jq_y + kq_z + q_z)(ir_x + jr_y + kr_z + r_w)\\ =& i(q_z r_z - q_z r_y + r_w q_x + q_w r_x)\\ &+ j(q_z r_x - q_x r_z + r_w q_y + q_w r_y)\\ &+ k(q_x r_y - q_y r_x + r_w q_z + q_w r_z)\\ &+ q_w r_w - q_x r_x - q_y r_y - q_z r_z\\ =& (\mathbf{q}_v \times \mathbf{r}_v + r_w \mathbf{q}_v + q_w \mathbf{r}_v, q_w r_w - \mathbf{q}_v \cdot \mathbf{r}_v)\end{aligned} \tag{3.8}$$

We define the *quaternion conjugate* as

$$\mathbf{q}^* = (\mathbf{q}_v, q_w)^* = (-\mathbf{q}_v, q_w) = (-q_x, -q_y, -q_z, q_w), \tag{3.9}$$

and the *quaternion norm* as

$$\mathrm{N}(\mathbf{q}) = \mathbf{qq}^* = \mathbf{q}^*\mathbf{q} = \mathbf{q}_v \cdot \mathbf{q}_v + q_w^2 = q_x^2 + q_y^2 + q_z^2 + q_w^2. \tag{3.10}$$

For every quaternion, there exists a corresponding *unit quaternion*, also called a *versor*:

$$\mathrm{U}(\mathbf{q}) = \frac{\mathbf{q}}{\sqrt{\mathrm{N}(\mathbf{q})}} \tag{3.11}$$

A *pure quaternion* is defined as a quaternion where $q_w = 0$. These will satisfy the relations $\mathbf{q}^* = -\mathbf{q}$ and $\mathbf{q}^2 = \mathrm{N}(\mathbf{q})\mathbf{q}$. A point $\mathbf{p} = (p_x, p_y, p_z)$ in Euclidean space $\mathbb{R}^3$, may be represented as a pure quaternion quaternion $\hat{\mathbf{p}} = (\mathbf{p}, 0) = (p_x, p_y, p_z, 0)$.

3.3.1 Combining transformations

This section demonstrates how quaternion *reflection*, *rotation* and *scaling* transformations are combined (similarly to how multiple affine transformations are combined with rotation matrices.)
A unit quaternion $\mathbf{q}$ will rotate a point $\mathbf{x}$ through the formula

$$\mathrm{R}(\mathbf{x}, \mathbf{q}) = \mathbf{qxq}^*. \tag{3.12}$$

Let us consider the same function $\hat{\mathrm{R}}(\mathbf{x},\mathbf{q})$, defined for *any* quaternions (i.e. both unit and non-unit quaternions):

$$\hat{\mathrm{R}}(\mathbf{x},\mathbf{q}) = \mathbf{q}\mathbf{x}\mathbf{q}^* \tag{3.13}$$
$$\hat{\mathrm{R}}(\mathbf{x},\mathbf{q}) = \sqrt{\mathrm{N}(\mathbf{q})}\mathrm{U}(\mathbf{q})\mathbf{x}\sqrt{\mathrm{N}(\mathbf{q}^*)}\mathrm{U}(\mathbf{q}^*) \tag{3.14}$$
$$\hat{\mathrm{R}}(\mathbf{x},\mathbf{q}) = \sqrt{\mathrm{N}(\mathbf{q})}\sqrt{\mathrm{N}(\mathbf{q}^*)}\mathrm{U}(\mathbf{q})\mathbf{x}\mathrm{U}(\mathbf{q}^*) \tag{3.15}$$
$$\hat{\mathrm{R}}(\mathbf{x},\mathbf{q}) = \mathrm{N}(\mathbf{q})\mathrm{U}(\mathbf{q})\mathbf{x}\mathrm{U}(\mathbf{q}^*) \tag{3.16}$$
$$\hat{\mathrm{R}}(\mathbf{x},\mathbf{q}) = \mathrm{N}(\mathbf{q})\mathrm{R}(\mathbf{x},\mathbf{q}) \tag{3.17}$$

Hence, the transformation $\hat{\mathrm{R}}(\mathbf{x},\mathbf{q})$, represents a rotation of the vector $\mathbf{x}$ by $\mathrm{U}(\mathbf{q})$, and a scaling by the norm $\mathrm{N}(\mathbf{q})$. Rather than discarding the information about scale, by using only unit quaternions, this may be used as an additional parameter in the transformation.
A series of rotations represented by $\mathbf{q}_1,\mathbf{q}_2\ldots,\mathbf{q}_n$, can be represented using a single composite quaternion $\mathbf{q}$:

$$\hat{\mathrm{R}}(\mathbf{x},\mathbf{q}_1,\mathbf{q}_2\ldots,\mathbf{q}_n) = \mathbf{q}_1\mathbf{q}_2\cdots\mathbf{q}_n\mathbf{x}\mathbf{q}_n^*\mathbf{q}_{n-1}^*\cdots\mathbf{q}_1^* = \mathbf{q}\mathbf{x}\mathbf{q}^* \tag{3.18}$$

Similarly, a series of reflections can be represented in the same way:

$$\hat{\mathrm{F}}(\mathbf{x},\mathbf{q}_1,\mathbf{q}_2\ldots,\mathbf{q}_n) = \mathbf{q}_1\mathbf{q}_2\cdots\mathbf{q}_n\mathbf{x}\mathbf{q}_n\mathbf{q}_{n-1}\cdots\mathbf{q}_1 = \mathbf{q}\mathbf{x}\mathbf{q} \tag{3.19}$$

What is the general quaternion transformation that combines both rotations, reflections and scaling transformations? Consider the composite function Ψ,

$$\Psi(\mathbf{x},\mathbf{q}_1,\mathbf{q}_2\ldots,\mathbf{q}_n) = \Psi_1(\Psi_2(\ldots\Psi_n(\mathbf{x},\mathbf{q}_n)\ldots),\mathbf{q}_2),\mathbf{q}_1), \tag{3.20}$$

where

$$\Psi_k(\mathbf{x},\mathbf{q}) = \Psi_k^L(\mathbf{q})\mathbf{x}\Psi_k^R(\mathbf{q}) = \begin{cases}\hat{\mathrm{F}}(\mathbf{x},\mathbf{q}) & \text{if reflection}\\ \hat{\mathrm{R}}(\mathbf{x},\mathbf{q}) & \text{if rotation}\end{cases} \tag{3.21}$$

with

$$\Psi_k^L(\mathbf{q}) = \mathbf{q}, \text{ for both rotations and reflections,} \tag{3.22}$$
$$\text{and } \Psi_k^R(\mathbf{q}) = \begin{cases}\mathbf{q} & \text{for reflections,}\\ \mathbf{q}^* & \text{for rotations.}\end{cases} \tag{3.23}$$

The formula can further be written as

$$\Psi(\mathbf{x},\mathbf{q}_1,\mathbf{q}_2\ldots,\mathbf{q}_n) = \Psi_1^L(\mathbf{q}_1)\cdots\Psi_k^L(\mathbf{q}_n)\mathbf{x}\Psi_k^R(\mathbf{q}_n)\cdots\Psi_k^R(\mathbf{q}_1) = \mathbf{q}^L\mathbf{x}\mathbf{q}^R \tag{3.24}$$

with $\mathbf{q}^L$ and $\mathbf{q}^R$ being the composite left and right side quaternions describing the general transformation of rotation, reflection and scaling.

3.3.2 Relation to other representations

Quaternions are conveniently converted into any other representation of rotations. Sometimes a different representation is a better option for a specific domain, such as shaders, where optimized matrix algebra routines can provide some advantages compared to quaternions. Rotation matrices also simplify the operations of non-uniform scaling and skewing.

3.3.2.1 Rotation matrices

Conversion of a quaternion (q_x, q_y, q_z, q_w) into a 3×3 affine matrix:

$$\begin{bmatrix} 1-2q_y^2-2q_z^2 & 2q_xq_y-2q_zq_w & 2q_xq_z+2q_yq_w \\ 2q_xq_y+2q_zq_w & 1-2q_x^2-2q_z^2 & 2q_yz-2q_xq_w \\ 2q_xq_z-2q_yq_w & 2q_yq_z+2q_xq_w & 1-2q_x^2-2q_y^2 \end{bmatrix} \tag{3.25}$$

A similar method for converting back to quaternions is described by Akenine-Möller et al. in [3].

3.3.2.2 Axis-angle

A quaternion is easily obtained from the axis-angle representation:

$$(\mathbf{q}_v, q_w) = \left(\sin\left(\frac{\theta}{2}\right)\mathbf{v}, \cos\left(\frac{\theta}{2}\right)\right) \tag{3.26}$$

The inverse operation (for $q_w \neq 1$):

$$(\mathbf{v}, \theta) = \left(\frac{\mathbf{q}_v}{\sqrt{1-q_w^2}}, 2\arccos(q_w)\right) \tag{3.27}$$

3.3.2.3 Euler angles

Euler angles to quaternion conversion is straight-forward:

$$\mathbf{q}_{\phi,\theta,\psi} = \begin{bmatrix} \cos(\psi/2)\cos(\theta/2)\sin(\phi/2) & - & \sin(\psi/2)\sin(\theta/2)\cos(\phi/2) \\ \cos(\psi/2)\sin(\theta/2)\cos(\phi/2) & + & \sin(\psi/2)\cos(\theta/2)\sin(\phi/2) \\ \sin(\psi/2)\cos(\theta/2)\cos(\phi/2) & - & \cos(\psi/2)\sin(\theta/2)\sin(\phi/2) \\ \cos(\psi/2)\cos(\theta/2)\cos(\phi/2) & + & \sin(\psi/2)\sin(\theta/2)\sin(\phi/2) \end{bmatrix} \tag{3.28}$$

3.3.3 Obtaining the rotation from two vectors

Suppose $\mathbf{u}$ and $\mathbf{v}$ are two known vectors and that $\mathbf{v} = \mathbf{q}\mathbf{u}\mathbf{q}^*$ (i.e., $\mathbf{v}$ is the vector $\mathbf{u}$ rotated by the quaternion $\mathbf{q}$.)
In this case, it can be shown that

$$(\mathbf{q}_v, q_w) = \left(\mathbf{u} \times \mathbf{v}, \mathbf{u} \cdot \mathbf{v}\sqrt{\mathrm{N}(\mathbf{u})\mathrm{N}(\mathbf{v})}\right) \tag{3.29}$$

4
Implementation

This chapter will detail the specifics of the implementation of MDG. Moreover, this chapter will cover the syntactical framework used for the implementation of the language itself. It will also cover the methods used for evaluating an MDG grammar in order to synthesize a shape (any procedural model represented in the MDG language, is considered to be a grammar in itself.) Finally, we will dissect the topic of compound transformations and animation.

4.1 Syntax and semantics

In this section we present the syntactical elements that are needed to understand MDG. A reference of the grammar that is used by the MDG parser can be found in A.1. Additionally a description of the different language elements is provided in B.

4.1.1 Directives

At the most basic level, an MDG model is defined by a set of global directives, such as `RULE`, `STARTSHAPE` and `BACKGROUND`. The definition of an MDG grammar consists of one or more `RULE` declarations.
Support for isosurfaces has been implemented through the `ISOSURFACE` directive, which generates a mesh model from evaluating a mathematical function on a 3D voxel grid, with an explicitly defined bounding box[1].
Additionally, some directives are used to control global variables, such as `BACKGROUND`, `WEIGHT`, `CAMERAFOV`, `MINSIZE` and `MAXDEPTH`.
A few directives are related to the shading system. The `PROGRAM` directive allows us to control the rendering process of the graphics pipeline, by specifying vertex and fragment shaders. It supports rendering in several passes as well as rendering to an off-screen texture. The `UNIFORM` directive allows us to set the values of specific uniform variables in the shaders (such as the position of light source.)

4.1.2 Production rules

Production rules are declared according to the pattern where `WEIGHT` is a value used as a metric for rule selection. A rule may have multiple declarations, such that there would be a conflict in name resolution in most procedural programming languages.

[1]Isosurfaces are generated by the use of an Open Source implementation of the marching cubes algorithm by Aaron Hochwimmer.

```
rule RULENAME WEIGHT {
  RULEBODY
}
```

MDG will handle such a conflict according to a rule selection process that depends on the weight of each conflicting rule.
The `RULEBODY` is a list of either *iterators* or references to other production rules that will be evaluated sequentially when this rule is executed.
The `STARTSHAPE` directive is used to determine which rule will be evaluated first.
In addition to explicitly defined production rules, there are some built-in implicit rules.

4.1.3 Drawables

While a production rule can represent both a terminal and a non-terminal, MDG also provides a few built-in terminal rules, which have been termed *drawables*:

- The `VERTEX` drawable will emit a vertex, transformed by the current transform space. This will also include information about the current normal vector. Every three vertices will be interpreted as a triangle by the rendering system.
- The `MOVETO` and `LINETO` drawables, will connect two vertices by using a separate profile rule, defined by the `LINERULE` directive.
- The `RESET` drawable will reset the transform space to the last coordinate and orientation of either `MOVETO` or `LINETO`.
- The `ISOSURFACE` command defines new drawables that are generated by evaluating a parametric function in $\mathbb{R}^3$-space.

MDG uses a modular approach for drawables, where new drawables can be registered dynamically. Possible extensions would be to import objects from different 3D file formats.

4.1.4 Namespaces

Rules may be logically grouped into different namespaces, using the `NAMESPACE` directive. Namespaces are particularly useful when using the `MOVETO` and `LINETO` commands (see Section 4.3.) Each namespace will maintain its own buffer for these commands, which allows us to have multiple lines that are connected using separate coordinates. It is possible to reference a namespace that has not been declared (e.g. `X.LINETO`) and it will remember the last coordinate in this particular namespace. This could also be used as a mechanism for reading back rotation and coordinate information at higher recursion levels, if combined with the continuation operator (see Section 4.2.4.)

4.1.5 Adjustments

One of the main differences between L-Systems and CFDG-based systems, is that CFDG encodes transformations within the production rules themselves (rather like many procedural languages.)
The following example will perform a rotation of 20° around the x-axis and 30° around the y-axis, moving 10 units along the z-axis and flipping across the z-plane, while simultaneously adding 0.2 and 0.3 to the color parameters `R` and `G`:

```
rule X{
  Y {rx 20 ry 30 z 10 fz R 0.2 G 0.3}
}
```

4.1.6 Iterators

CFDG provides an intuitive method for iteration. Loops have a similar structure as in many procedural languages, but the code does not include a loop index. The following rule will perform 10 iterations, rotating 5° around the x-axis and moving 3 units along the z-axis in each iteration:

```
rule X {
  10 * { rx 5 z 3 } Y {}
}
```

These kind of constructs have not been studied in great detail in other shape grammars. Parametric L-systems permit the use of parameter values associated with each symbol. Such parameters allows a kind of controlled expansion, where terminal rules can be associated with a parametric value. However, this is not the same as iteration, since it requires that all symbols in the output string are rewritten before the next parametrically controlled expansion step.
A special construct was added to the syntax of iterators that we term *divisors*. Instead of specifying only the number of iterations m, the iterations can also be specified as m/n, where n is a number that divides the transformation into n smaller steps.

4.1.7 Macros

A macro preprocessor was implemented. The preprocessor supports several useful directives:

`@IMPORT` directive for including other MDG projects;

`@FRAGMENTSHADER` and `@VERTEXSHADER` directives for defining shaders;

`@DEFINE`, `@IFDEF`, `@ELSE`, `@ENDIF` for conditional parsing;

`@MACRO` defines a macro and allows parametric expansions;

`@EXPAND` expands a previously defined macro;

`@PRESETS` defines presets that can be accessed in the user interface;

`@PARAM` parameter value associated with a preset (`int`, `float` or `bool`.)
The system fully supports the use of custom shaders. There are many reasons not to use a static shader in a procedural modeling language. Shader design is a whole research field in its own and by combining this with a solution for procedural modeling, we get a solution with maximum flexibility in terms of visualization options.

4.2 Evaluation

Our production rules are compiled into a derivation tree that can be evaluated either through depth-first or breadth-first traversal.
Other CFDG-based implementations, such as Context Free and StructureSynth use a breadth-first evaluation. This has some advantages, but it will not give the same results if we use some of the novel constructs that are introduced in this section, such as *continuation* and *sequential selection*.

4.2.1 Transforms

For each rule in the grammar, we precompute a composite transformation, representing translation, scaling, reflection and rotation.
Each expression of adjustments, maintains a compact internal representation of the transformations needed to be carried out. Rather than computing every transformation in the evaluation pass of the syntax tree, a composite transformation is precomputed for each transform expression. This representation will respect the order of the all of the transformations (scaling, translation, rotation, reflection and rotation), and it will be represented by three distinct quaternions $\mathbf{q}$, $\hat{\mathbf{q}}$ and $\mathbf{u}$.

4.2.1.1 Internal representation of transforms

The internal representation of transforms requires that we maintain a set of three quaternions for each transform expression and for each stack frame. The quaternions associated with a transform expression are computed from the transform expression according to the procedure outlined below.

Initialization of variables:

$$\mathbf{u}_0 = 0 \qquad \text{(translation)} \tag{4.1}$$

$$\mathbf{q}_0 = \hat{\mathbf{q}}_0 = 1 \qquad \text{(orientation)} \tag{4.2}$$

Translation by $\mathbf{v}$:

$$\mathbf{u}_{k+1} = \mathbf{u}_k + \mathbf{q}_k \mathbf{v} \hat{\mathbf{q}}_k \tag{4.3}$$

Rotation by $\mathbf{r}$:

$$\mathbf{q}_{k+1} = \mathbf{q}_k \mathbf{r} \tag{4.4}$$

$$\hat{\mathbf{q}}_{k+1} = \mathbf{r}^* \hat{\mathbf{q}}_k \tag{4.5}$$

Reflection by $\mathbf{r}$:

$$\mathbf{q}_{k+1} = \mathbf{q}_k \mathbf{r} \tag{4.6}$$

$$\hat{\mathbf{q}}_{k+1} = \mathbf{r}\hat{\mathbf{q}}_k \tag{4.7}$$

Scaling by a scalar s:

$$\mathbf{q}_{k+1} = s\mathbf{q}_k \tag{4.8}$$

$$\hat{\mathbf{q}}_{k+1} = s\hat{\mathbf{q}}_k \tag{4.9}$$

The evaluation process is similar. We initialize our variables:

$$\mathbf{p}_0 = 0 \qquad \text{(translation)} \tag{4.10}$$

$$\mathbf{w}_0 = \hat{\mathbf{w}}_0 = 1 \qquad \text{(orientation)} \tag{4.11}$$

When a rule is invoked , the precomputed transform is applied as follows:

$$\mathbf{p}_{k+1} = \mathbf{p}_k + \mathbf{w}_k \mathbf{u} \hat{\mathbf{w}}_k \tag{4.12}$$

$$\mathbf{w}_{k+1} = \mathbf{w}_k \mathbf{q} \tag{4.13}$$

$$\hat{\mathbf{w}}_{k+1} = \hat{\mathbf{q}} \mathbf{w}_k \tag{4.14}$$

Note that these are all the computations that are needed in order to perform all of the transformations in each adjustment formula. No extra step is needed for scaling – it is implicit in the multiplication by the (non-unit) quaternions.
Note that any value could be chosen for the initialization of p_w. The quaternion vectors sent to the vertex shader will be multiplied by the projection matrix and then it is necessary to set $p_w = 1$ for uniform scaling. However, p_w could also serve as an additional parameter for controlling the the scale of our model, since in effect, the coordinates will be mapped to $(x/w, y/w, z/w)$ after multiplication with a projection matrix.

4.2.2 Rule selection

One of the main ideas of CFDG, and consequently, of MDG, is that it permits multiple conflicting production rules. When a rule that has been declared multiple times is evaluated, only one of its definitions is selected. This does not violate the principle of the corresponding grammars being context-free.
CFDG resorts exclusively to a stochastic selection process, whereas MDG supports several different alternatives for selection. This is also a feature that provides significant advantages to L-systems, which have limited capabilities in terms of rule selection (although parametric, stochastic and context-sensitive L-Systems provide some mechanisms for rule selection.)

4.2.2.1 Probabilistic selection

MDG inherits the default stochastic method for rule selection that is also found in CFDG. The weight value associated with a rule denotes the probability that this rule will be selected.

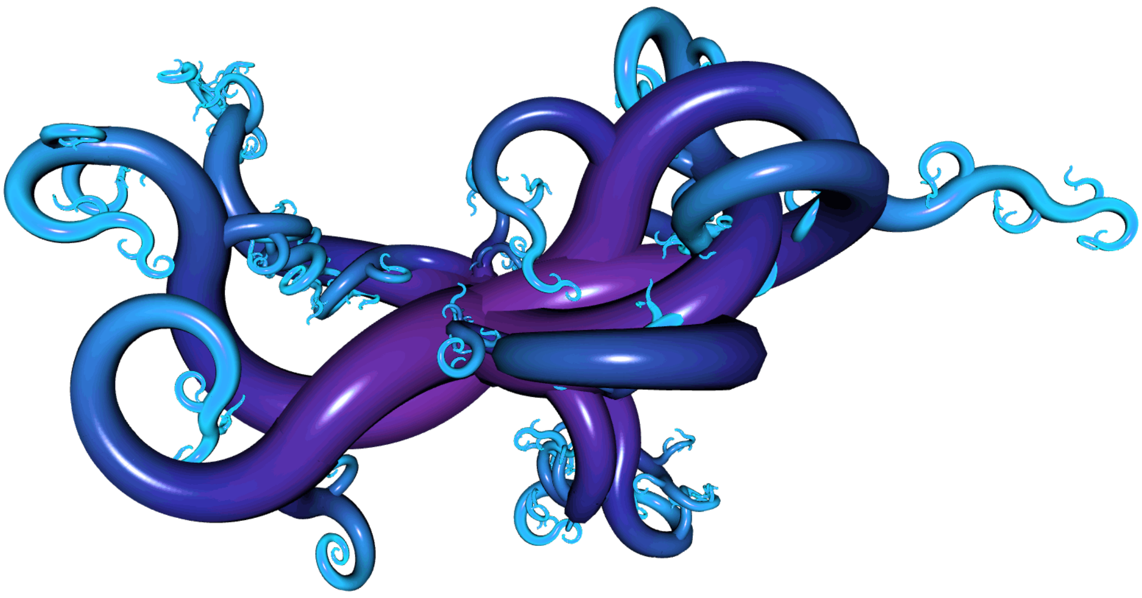


Figure 4.1: Model generated through probabilistic selection of rules.

Listing 4.1: Example of a grammar for generating a stochastic model

```
selector A frequency
rule A 0.01 {}
rule A 0.1 {
  A{z 1}
  A{rz 0.3 rx 0.35}
rule A 0.9 {
  SPHERE{}
  A{z 1 ry 0.01}
}
```

4.2.2.2 Recursion depth selection

Rule selection can also depend on recursion depth. The weigh value is interpreted as the recursion depth and a rule is selected only if its weight value is less than or equal to the current recursion depth.

4.2.2.3 Scale selection

Scale selection determines which rule to use depending on the scale of the current transform space. Since scale information is encoded within the quaternions that we use for transformations, this is simply a matter of using the norm of the quaternion representing our orientation, i.e. $N(\mathbf{q})$, as our rule selection metric.

If rotation matrices were used instead of quaternions, this would have been a more expensive computation. Typically this would involve computing the determinant of the matrix, in order to get a metric of uniform scale.

4.2.2.4 Functional selection

Rule selection can be made by evaluating a custom multivariate function $f : \mathbb{R}^n \rightarrow \mathbb{R}$. In our current implementation this function can be a custom expression of our current position, but this could also be extended to include other parameters, such as the current orientation, or any other custom parameters that are passed on the stack in the evaluation (similarly to conditions in parametric L-systems.)

4.2.2.5 Sequential selection

Sequential selection provides an ordered deterministic approach to rule selection. Each rule will be executed sequentially n times, where n is the rule weight. Note that this kind of scheme will cause different results depending on whether depth-first or breadth-first evaluation is used.

4.2.3 Termination criteria

From the previous section, it should be evident that we may terminate a recursion by using a suitable selection criteria.
However, it is also useful to have some default criteria for implicit termination. Two such criteria have been implemented: Evaluation may stop after a specified number of recursions, through the `maxdepth` directive. It may also stop when the scale of the current transform becomes sufficiently small, by using the `minsize` directive. This corresponds to *depth selection* and *scale selection* when using explicit rule selection.

4.2.4 Continuation

Each rule will evaluate each of the production rules defined within its body. This forms *parent-child* relationship between productions. The child rule automatically inherits the transform space of its parent rule (transformed by any intermediate adjustment expressions.) When the next rule is evaluated, again it will be relatively to the parent transform space.
However, sometimes it is desirable to actually use the transform space of the preceding rule within the rule body. This concept is denoted *continuation*, since we will continue from where the last rule last updated the transform space (possibly at a completely different recursion depth.)
Below is shown an excerpt from the code used to generate an SiC crystallographic structure (see C.4 for the complete model):

```
rule ABCB {
  SiC {} \     // A
  SiC {} \     // B
  SiC {fx} \   // C (flip x axis)
  SiC {} \     // B
  MOVETO{fx}   // (flip x axis -- for the next ABCB)
}
```

Each invocation of the `SiC` rule will adjust the translation of the transform space. By using continuation, we will place the next SiC molecule such that it connects to its neighbor and forms a crystallographic structure. Finally, we use `MOVETO` in order to update the transform space through reflection of the yz-plane, so that the next ABCB layer will be correctly transformed.

4.3 Curve sweeping

Techniques for performing *curve sweeps* were implemented. A line is constructed by extruding a profile curve, defined by the `LINERULE` command, along a trajectory or a motion path.
A generalized cylinder is constructed for every line segment that is defined with the `MOVETO` and `LINETO` commands[2]. When `LINETO` is called an extrusion is constructed from the coordinate saved by the last `MOVETO` or `LINETO` command. The extrusion connects the vertices of two profile curves with a triangle strip.
The `BLENDTO` command was implemented as a special variant of `LINETO` that creates a smoothly interpolated path between the two vertices.

4.3.1 Smooth joints

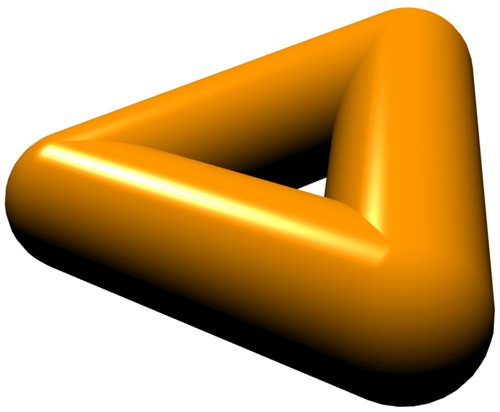
A common problem in modeling is to connect two separate objects that are defined in two separate transform spaces, so that they join together smoothly. Consider two different cylinders that are separated by some distance and oriented in different directions. We want to find an intermediate surface that connects one cylinder to another. This problem exists not only in modeling, but also in animation, where different joints of a skeleton model need to be animated.
One technique to address this problem is known as *vertex blending* or *skinning*. We know the shape of the surfaces that we want to connect (circles in the case of cylinders) and we know the transforms that define their origin and orientation. By interpolating the two transform spaces using a weight, $w \in [0, 1]$, we are able to compute new vertices that form a smooth joint (our representation of a transform space and our interpolation function, will determine the smoothness of the surface[3].)

4.3.2 Spline interpolation

Quaternions have been studied extensively for the purpose of interpolating orientations along motion paths and trajectories. In animation systems this is sometimes referred to as *guiding*. The spherical interpolation scheme investigated by Shoemake [44], shows how to perform interpolation such that the orientation changes with even steps along the unit sphere. This scheme was combined with a Bezier spline interpolation scheme for coordinates. Duff proposed another interpolation scheme based on B-splines [16]. Yet another scheme was investigated by Pletinckx [36].
The `BLENDTO` drawable was implemented in MDG in order to not only interpolate position and orientation, but also to interpolate information about the scale. The

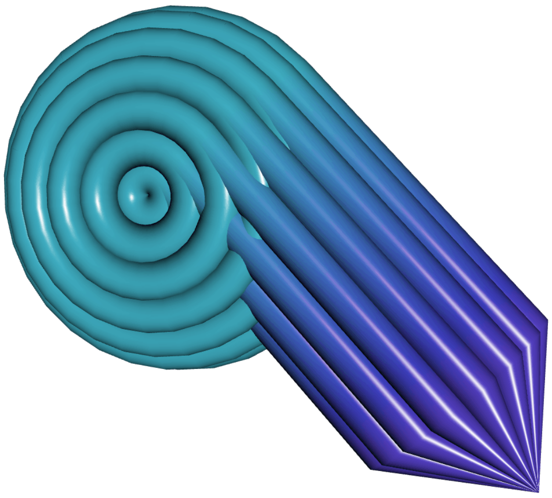
[2]These commands are meant to be 3D analogues of the same commands found in many 2D canvas libraries.

[3]It has been demonstrated that Dual Quaternions provide a better solution to this problem than regular quaternions[24].



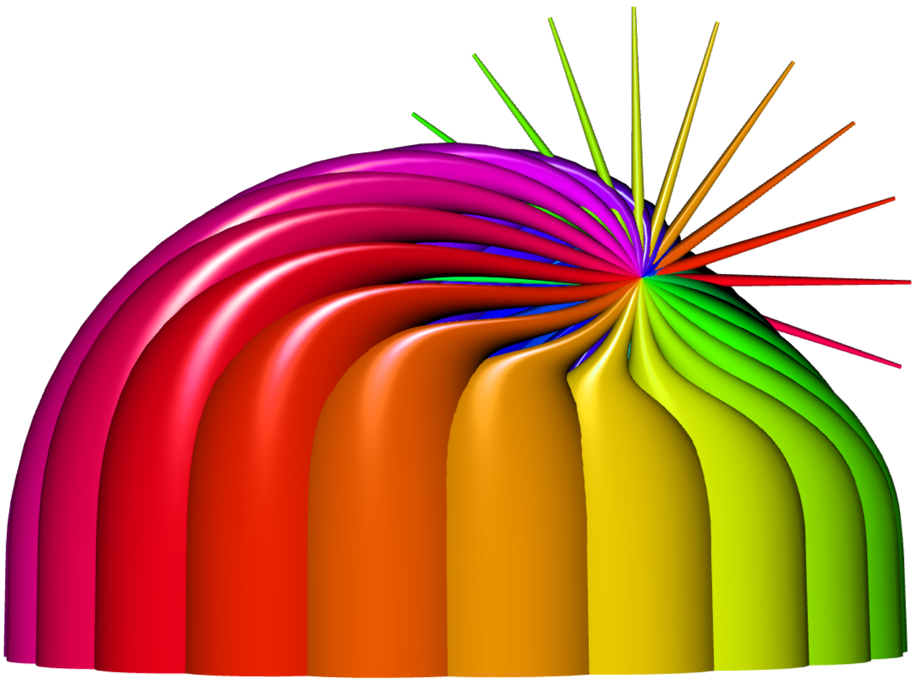
```
rule SMOOTH_TRIANGLE {
  LINETO{z 1} \
  10/10 * {rx 120} LINETO{} \
  LINETO{z 1} \
  10/10 * {rx 120} LINETO{} \
  LINETO{z 1} \
  10/10 * {rx 120} LINETO{} \
}
```

(a) Smooth joints



```
rule REVOLVE_EXTRUDE {
  NEWLINE{}
  30/30 * [ry 360] LINETO{}
  MOVETO{}
  LINETO{x -15 R 1} \
  LINETO{x -5 R 1 s 0}
}
```

(b) Extrude and lathe



(c) Vertex blending: two coordinates with different orientation are connected using a smooth spline curve.

Figure 4.2: Different techniques for creating smooth joints.

implementation assumes that the line is going to continue in the z-direction[4] from the source coordinate and orientation pair $(\mathbf{u}_1, \mathbf{q}_1)$ to the target coordinate and orientation $(\mathbf{u}_2, \mathbf{q}_2)$. The implementation below was derived experimentally:

$$D = \sqrt{\mathrm{N}(\mathbf{u}_1 - \mathbf{u}_2)} \cdot \lambda \tag{4.15}$$

$$\mathbf{v}_1 = \mathbf{q}_1 \left(0, 0, \frac{D}{\mathrm{N}(\mathbf{q}_1)}, 0\right) \mathbf{q}_1^* \tag{4.16}$$

$$\mathbf{v}_2 = \mathbf{q}_2 \left(0, 0, -\frac{D}{\mathrm{N}(\mathbf{q}_2)}, 0\right) \mathbf{q}_2^* \tag{4.17}$$

The interpolated coordinate and orientation $(\mathbf{u}, \mathbf{q})$ is computed for the interpolant $w \in [0, 1]$:

$$s = \mathtt{SMOOTHSTEP}(w) \tag{4.18}$$

$$\mathbf{u} = \mathtt{LERP}(\mathtt{LERP}(\mathbf{u}_1, \mathbf{v}_1, s), \mathtt{LERP}(\mathbf{u}_2, \mathbf{v}_2, s), w) \tag{4.19}$$

$$\hat{\mathbf{q}} = \mathtt{LERP}(\mathbf{q}_1, \mathbf{q}_2, w) \tag{4.20}$$

$$\mathbf{q} = \hat{\mathbf{q}} \sqrt{\frac{\mathtt{LERP}(\mathrm{N}(\mathbf{q}_1), \mathrm{N}(\mathbf{q}_2), s)}{\mathrm{N}(\hat{\mathbf{q}})}} \tag{4.21}$$

Figure 4.2c demonstrates the effect of using this technique with the parameter $\lambda = 0.75$, when splining between different transform spaces.

4.3.3 Straight lines

The `LINETO` and `BLENDTO` commands connect two coordinates with different scale and orientation using curve sweeping along a line. The orientation at each coordinate does not necessarily need to correspond to the orientation of the line.
However, sometimes it is desirable to make the extrusion orthogonal to the line segment. The `SLINETO` command was added for this purpose. In order to find the quaternion that represents the orientation of our line, we need to use Equation 3.29.

4.4 Animation

Support for animation was included by observing that in a tree-like recursive representation of a model, each branch or each shape along the path could be an object that could be animated separately.
By assigning an *articulation function* [31] to each bone, we are able to animate multiple attributes, such as orientation, position and color. This has been implemented in a way that permits real-time animation of bones through the use of customized vertex shaders.

[4]It is possible to use any direction as a convention.

4.4.1 Animation frames

The concept of *animation frames* was introduced. In order to understand the implementation, we need to distinguish between *model rules* and *transform rules*. A *transform rule* is a rule that will emit *frames*, while a *model rule* is a rule that will emit *shapes* or *vertices*.
Each rule is associated with a number of *transform rules*. Animation is achieved through a number of different steps:

1. Evaluate the procedural model and construct a hierarchical bone structure, where each bone is a model rule with at least one associated transform rule;
2. Sample animation frames at time t for each bone in the generated model[5];
3. Build transform matrices used to animate each bone in the vertex shader[6]. This step requires conversion from quaternions to matrices, but it will be done only once for each transform rule and it has not proven to be a performance bottleneck.

Listing 4.2: The shape generated by *model rule* A is associated with a number of *transform rules*, $X_1, \dots, X_n$.

```
rule A : X1 : X2 : ... : Xn {
  ...   // generate our model
}

rule X1 {
  // rotate 360 degrees around z axis and move 10 steps
  // along x axis (emit 100 frames in total)
  100 / 100 * {rz 360 x 10} FRAME{}
}

rule X2 {
  FRAME{}
  X2{z 0.1}    // move 0.1 steps along z axis (infinite recursion)
}
```

How do we rewind to an earlier frame, which has already been evaluated? Each transform rule maintains its own cache of sampled animation frames. Hence, rewinding the animation is simply a look-up process in the local cache of each transform rule.

4.4.2 Camera trajectories

The camera is assigned a rule of its own through the `CAMERARULE` directive. This rule describes a time-dependent trajectory for the motion and orientation of the camera. Model parameters can be used to animate certain properties of the camera, such as field-of-view or look-at direction.

[5]Transform rules will emit a frame at a time interval specified by the `fps` parameter (which can be changed globally and can also be modified individually for each rule by adjustment expressions.)

[6]We will maintain an index for every vertex that will determine which bone matrix to use in the shader.

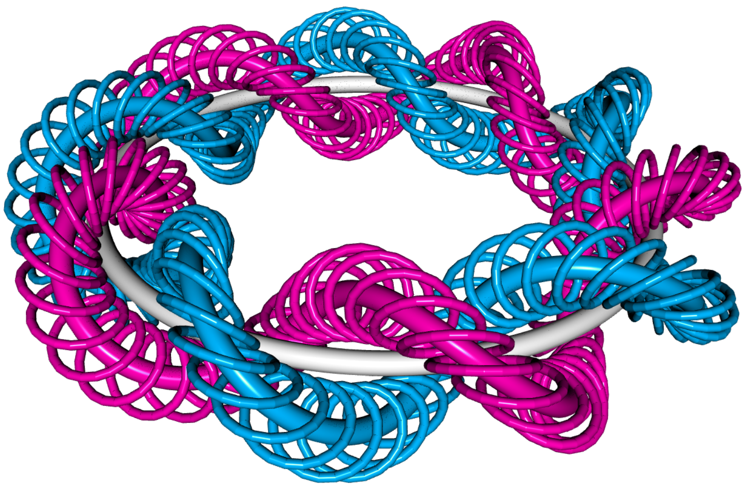


Figure 4.3: Demonstration of a model created with composite transform rules. Each transform represents a rotation around the path of the parent transform. See code in C.3.

4.5 Modeling with transform rules

It was observed that there exists a class of procedural objects that are not easily described by the generic MDG grammars. Figure 4.3 shows an object that is constructed from tangled transforms. These curves correspond to the trajectories of an animation with one, two and three orthogonal transforms, respectively.
The key component for this kind of composition is to select one rule to be used as an *iterator*. The `spiralring` rule in the following example, will sample every frame of the `ringpath` iterator rule and it will composite this transformation by the same frame from the `firstspiral` and `secondspiral` rules:

```
rule spiralring * ringpath : firstspiral : secondspiral {
  SLINETO{s 0.1}
}
```

Note that in this situation, we do not perform a time-dependent sampling, as is necessary in animations, but rather we sample by the index of each frame (it is also necessary that we sample only a finite number of frames, i.e. the rule needs to terminate.)

5 Results

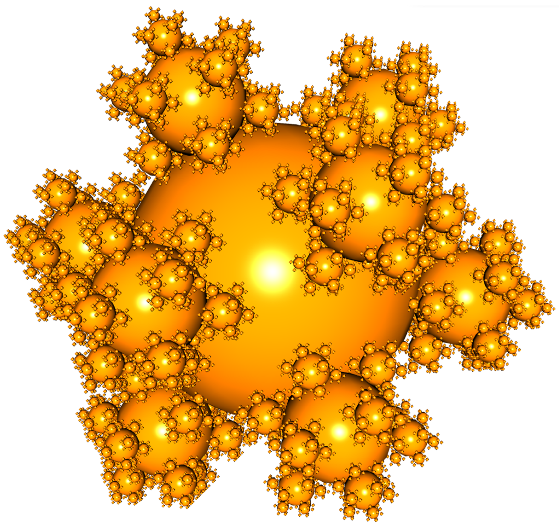
A motivating factor in this thesis project has been to find a generic language for the representation of a great number of geometric objects. Ideally, this should be a language that favors simplicity and that can easily be interpreted, even by a non-expert.
MDG grammars provide a tool for modeling that not only can represent an object in a concise way, but can also provide an insight about the geometry of the object at hand. It is useful to think about geometries in terms of their descriptions in MDG and to use this as a method for exploration of new structures. In the context of molecular physics, you might want to ask "what are the different representations of this specific crystal lattice?" or "how can we describe its symmetries in terms of a set of transforms?" MDG grammars provide an answer that is usually meaningful to the interpretation of these kind of physical or theoretical entities.
This section will illustrate with a number of examples how MDG grammars can be used for many different applications, including molecular structures, fractals and polyhedra.

5.1 Level of detail

Multiple approaches have been devised to adjust level-of-detail (LOD) when rendering complex scenes with many objects. The method known as *continuous LOD*, will adjust the complexity of a model according to some heuristic function (typically distance to the camera.) However, in our recursively evaluated grammars, we may want to specify a different heuristics, such as the current scale of our transform space. Figure 5.1 demonstrates how macros can solve the problem of LOD, where each expansion of a macro corresponds to an isosurface of a different resolution and a corresponding rule with a weight that is used as a metric for scale selection. The smallest spheres are represented as an octahedron with no more than eight vertices.

5.2 L-systems

One of the questions that arose during the course of this thesis work was whether or not L-systems can be represented in MDG. As we shall see, there is in fact a straight-forward conversion between 0L-systems and MDG grammars.
This involves using *depth selection* to limit the recursion depth and *continuation*, to continue from the coordinate and transform of the last rule. Figure 5.2 shows



```
@macro SPHERE_LOD(SIZE,RES)
isosurface SPHERE(RES) {
  xyz -1 1 (RES)
  function { 1.0 - (x*x+y*y+z*z) }
}
rule SPHERE (SIZE) {
  SPHERE(RES) {}
}
@endmacro

@expand SPHERE_LOD(0.02,2)
@expand SPHERE_LOD(0.04,3)
@expand SPHERE_LOD(0.08,4)
@expand SPHERE_LOD(0.60,6)
@expand SPHERE_LOD(1.00,9)
selector SPHERE scale
```

Figure 5.1: Sphereflake model, rendered using continuous LOD.

two very basic L-systems that have been converted to their corresponding MDG grammars.

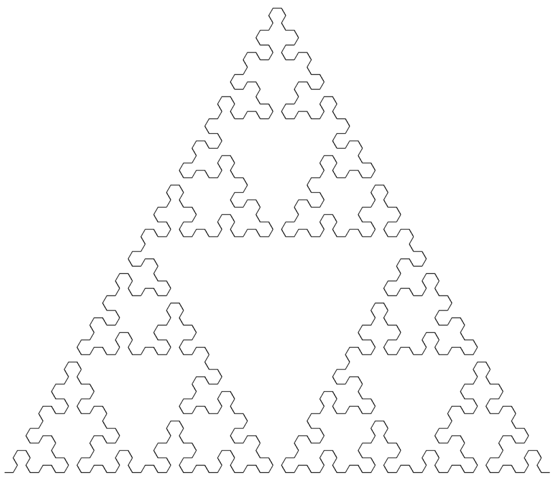
5.3 Molecular models

During the course of the thesis project, I was contacted by M.Sc. student Maria Karani. Maria had seen an early prototype of my system and inquired about whether it would be possible to use it for rendering a crystallographic lattice model of a specific silicon carbide molecule. This proved to be a rather trivial structure to represent in MDG. Figure 5.3a shows the rendered version of the 4-H SiC molecule examined by Maria (see code in C.4.)

5.4 Polyhedra

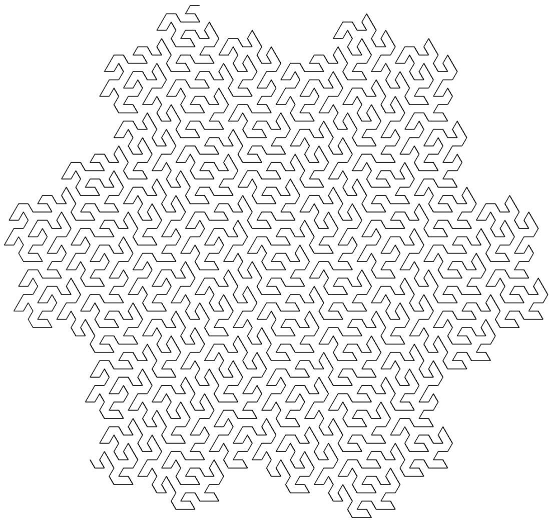
MDG provides compact representations of many geometrical objects. Regular polyhedra is an interesting class of objects that can be modeled in a straight-forward way with MDG (provided that you manage to devise the appropriate angles and distances.)
The following code will generate the complete skeleton model of a truncated icosahedron as shown in Figure 5.3b:

```
rule truncated_icosahedron{
  5/5*{rx 1} 2/2*{ry 1} 1*{ry -.2376862591} 2*{ry .3838602364} {
    NEWLINE{}
    6/6*[rz 1] LINETO{rx 0.4410563588t z 10}
  }
}
```



```
n = 6, d = 4, δ = 60°
YF
XF → YF+XF+YF
YF → XF-YF-XF
```

(a) The Sirepinski arrowhead curve and its representation using L-system notation. One of the models presented in Prusinkiewicz's seminal paper [39].



```
n = 4, d = 5, δ = 60°
XF
XF → XF+YF++YF-XF--XFXF-YF+
YF → -XF+YFYF++YF+XF--XF-YF
```

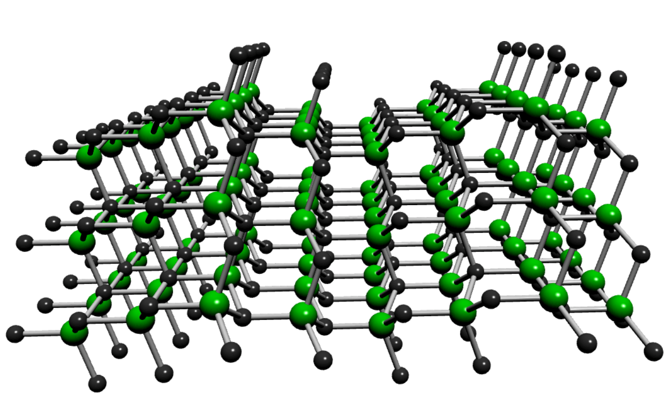
(b) Peano-Gosper space-filling curve, originally named *flowsnake*. One of the fractals studied by Mandelbrot in 1977 [30].

Figure 5.2: There is a straight-forward conversion between the syntax of L-systems and the MDG grammars. By selecting rules based on *recursion depth*, we will achieve the same effect as an L-system, but without the need to first construct a symbolic string representation.

```
startshape YF
selector XF, YF depth

rule XF 6 { YF{} \ XF{rx 60} \ YF{rx 60} }
rule YF 6 { XF{} \ YF{rx -60} \ XF{rx -60} }
rule XF 7 { SLINETO{z 1} }
rule YF 7 { SLINETO{z 1} }
```

Listing 5.1: Equivalent Sirepinski arrowhead curve written in MDG. Note the use of the continuation character in order to continue from the position and orientation of the last terminal rule.


(a) 4-H SiC ABCB structure examined by Maria Karani, in her M.Sc. thesis.

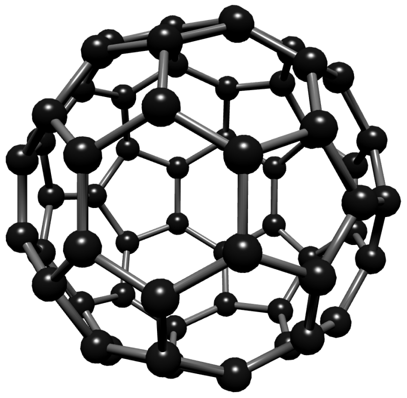

(b) Model of a fullerene molecule, also known as a *buckeyball.*

Figure 5.3: One domain, where compact descriptions of models is essential, is within the field of nanoscale molecular science.

The constants used in this code have been derived analytically, through the formula for dihedral angles of an icosahedron:

$$a = \frac{4r}{3+\sqrt{5}} \tag{5.1}$$

$$r = 1 \quad \Rightarrow \quad a_0 = 0.7639320224 \tag{5.2}$$

$$r = 1/\sqrt{3} \quad \Rightarrow \quad a_1 = 0.4410563588 \tag{5.3}$$

$$\frac{\arctan a_0 + \arctan 2}{2\pi} = 0.3838602364 \tag{5.4}$$

$$\frac{1}{4} - \frac{3\arctan a_0 + \arctan 2}{2\pi} = -0.2376862591 \tag{5.5}$$

5.5 Knots

Some experimenting with compound transforms resulted in an interesting family of knot-like models, as shown in Figure 5.4. Despite the simple description of this model (see C.2), it lends itself to experimentation by adjusting the number of revolutions about each axis. It was also discovered that if the number of revolutions about each axis is denoted n_x, n_y, n_z, then at least two of these numbers need to be co-prime in order for the path to be connected.

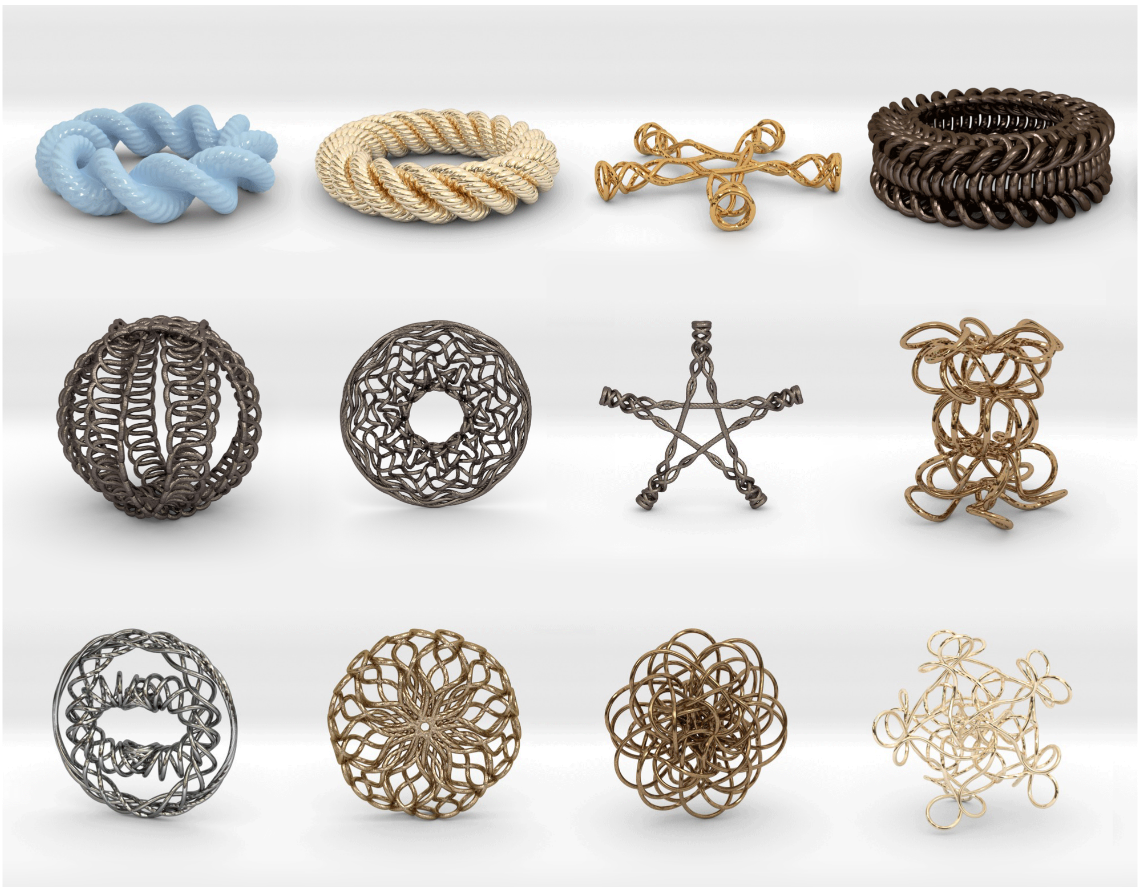


Figure 5.4: Examples of different knot structures generated with MDG. The models have been exported as .STL and uploaded to online 3D printing service Shapeways [1], which permits 3D printing of models in a wide variety of materials.

6 Discussion

While MDG has been inspired by CFDG, it has also been augmented with several language features that are not part of the latter. The use of divisors in iterated statements has been very useful in that it provides an easy way to divide a sequence of transformations into smaller steps. A common task in modeling is to perform a rotation around an axis through a number of steps, which is simply a matter of dividing by the number of iterations.

Unlike CFDG, which performs a breadth-first traversal of the syntax tree, MDG performs depth-first traversal. This requires less memory (and memory is more expensive in 3D, due to the overhead of additional parameters passed on the stack.) The major drawback of depth-first evaluation is when the evaluation process is visualized in real-time, since it will show only sub-branches of the final model (consider a tree that grows by one arm at a time, rather than all arms growing by a small amount simultaneously.) Breadth-first evaluation might also be an advantage in context-sensitive grammars, such as IL-systems, since the context should be sensitive to the current growth iteration. Probably MDG will never be able to support the kind of context-sensitive features described by IL-systems (since that actually relies on the parallel rewriting paradigm.)

There are also certain observations to be made about the use of transforms in MDG. Should a transform be implemented as a 3×3 matrix or as a quaternion? There are many reasons to choose one representation or the other. Matrix-matrix and matrix-vector multiplication is generally faster than the corresponding quaternion methods. However, quaternions allow us to reduce our memory footprint, by storing only four values. Additionally, interpolation of orientations is a non-trivial task with transform matrices. For this application quaternions provide a great advantage.

Many textbooks refer to unit quaternions as the only feasible entity in which rotations are properly represented. The author of this thesis feels that this is a vaguely misleading observation. Sometimes it is desirable to use unit quaternions (it simplifies some of the analytical expressions and it can be a way to avoid floating-point drift when performing many compound transformations.) However, as has been shown in this thesis, a quaternion will still represent a rotation if it is scaled. Additionally, the magnitude of the quaternion provides a useful metric for rule selection.

Rule selection can also be made interactive, by reading the state of an input device, such as a mouse or a keyboard. This would be particularly interesting in the context of transform rules, where an input device could be used to control the animation. Some other interactive aspects would be to make it possible to select different objects defined by the transform hierarchies and to manually adjust their orientation and parameter values.

7 Conclusions

Development of MDG has been an exciting journey and a fruitful learning experience. The result of this thesis is an advanced solution for both animation and modeling. Using shape grammars for procedural modeling is not a new phenomenon, but research has been fairly constricted to a few different shape grammars, such as the many variations of L-systems. CFDG-based modeling systems, prove to be a very powerful alternative to these grammars. The similarity with procedural languages, makes it easy for people with little prior experience of shape grammars to start modeling. By embedding transformations within the production rules of the language, we add clarity and readability that is sometimes lost when representing grammars only as symbols.

The approach towards motion paths and compound transformation is a novel concept in MDG, and has little resemblance with previous implementations. By uniting the two processes of modeling and animation, it is possible to think about these concepts not as distinct abstractions, but rather as two interchangeable methods, where there is no firm boundary between an animation path and a shape that is generated procedurally along the same path.

MDG provides several options for intuitively and easily representing many different scientific models, such as planetary systems, rigid body dynamics, molecular models and plant models. It offers a solution for rapid prototyping and for experimental designs in generative art. There is also a strong case to be made for the use of MDG as a tool within education. Similarly to how CFDG has been used in the education programs of design students [43], MDG will provide the same kind of benefits and it will extend the design space into the realm of three dimensions.

While it is a subject that deserves further research, this thesis also shows that MDG provides an excellent option for representing complex geometry, such as regular polyhedra. These kind of models will also benefit from the reflectional symmetry of quaternion transformations.

We have demonstrated that there exists en easy transformation for converting a Lindemayer 0L-system into an MDG grammar. Curiously, 0L-systems are evaluated breadth-first, while our implementation requires a depth-first evaluation combined with *continuation* in order to simulate the same system. While it was not demonstrated, it should also be noted that this transformation applies also to stochastic 0L-systems. What about bracketed and parametric L-systems? It has not been demonstrated how these systems can be represented in MDG, but this is certainly another possible path of future research.

Bibliography

[1] Shapeways. `http://www.shapeways.com/`.

[2] Jon Lennart Aasenden and Eric Grange. Smart mobile studio. `http://smartmobilestudio.com/`.

[3] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-time rendering*. CRC Press, 2008.

[4] Mattias Andersson. Onlinemodeler. `http://www.onlinemodeler.com/`.

[5] Michael F Atiyah, Raoul Bott, and Arnold Shapiro. Clifford modules. *Topology*, 3:3–38, 1964.

[6] Bruce G Baumgart. Geomed: Geometric editor. Technical report, DTIC Document, 1974.

[7] Bruce Guenther Baumgart. Geometric modeling for computer vision. Technical report, DTIC Document, 1974.

[8] Frédéric Boudon, Christophe Pradal, Thomas Cokelaer, Przemyslaw Prusinkiewicz, and Christophe Godin. L-py: an l-system simulation framework for modeling plant architecture development based on a dynamic language. *Frontiers in Plant Science*, 3, 2012.

[9] Arthur Cayley. Xiii. on certain results relating to quaternions: To the editors of the philosophical magazine and journal. 1845.

[10] Noam Chomsky. Syntactic structures. 1957.

[11] Noam Chomsky. *Aspects of the Theory of Syntax*, volume 11. MIT press, 1969.

[12] Mikael Hvidtfeldt Christensen. Structuresynth. `http://structuresynth.sourceforge.net/`.

[13] Mikael Hvidtfeldt Christensen. Structural synthesis using a context free design grammar approach. In *Generative Art International Conference*, 2009.

[14] HSM Coxeter. Quaternions and reflections. *American Mathematical Monthly*, pages 136–146, 1946.

[15] Chris Coyne, Mark Lentczner, and John Horigan. Context free art. `http://www.contextfreeart.org`, 2010.

[16] Tom Duff. Splines in animation and modeling. *State of the Art in Image Synthesis (SIGGRAPH'86 course notes Number 15, Dallas, TX)*, 1986.

[17] Wallace Feurzeig and Seymour Papert. The logo programming language, 1967.

[18] James Gips. *Shape grammars and their uses*. PhD thesis, Stanford University Palo Alto, CA, 1974.

[19] James Gips. A syntax-directed program that performs a three-dimensional perceptual task. *Pattern Recognition*, 6(3):189–199, 1974.

[20] William Rowan Hamilton. On quaternions. In *Proceedings of the Royal Irish Academy*, volume 3, pages 1–16, 1847.

[21] Andrew J Hanson. Visualizing quaternions. In *ACM SIGGRAPH 2005 Courses*, page 1. ACM, 2005.

[22] Andrew J Hanson and Sidharth Thakur. Quaternion maps of global protein structure. *Journal of Molecular Graphics and Modelling*, 38:256–278, 2012.

[23] David Hestenes and Garret Sobczyk. *Clifford algebra to geometric calculus: a unified language for mathematics and physics*, volume 5. Springer Science & Business Media, 1987.

[24] Ladislav Kavan, Steven Collins, Jiří Žára, and Carol O'Sullivan. Skinning with dual quaternions. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 39–46. ACM, 2007.

[25] Jack B Kuipers. *Quaternions and rotation sequences*, volume 66. Princeton university press Princeton, 1999.

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

[27] Steven Longay, Adam Runions, Frédéric Boudon, and Przemyslaw Prusinkiewicz. Treesketch: interactive procedural modeling of trees on a tablet. In *Proceedings of the international symposium on sketch-based interfaces and modeling*, pages 107–120. Eurographics Association, 2012.

[28] Penousal Machado and Henrique Nunes. A step towards the evolution of visual languages. In *First International Conference on Computational Creativity, Lisbon, Portugal*, 2010.

[29] Penousal Machado, Henrique Nunes, and Juan Romero. Graph-based evolution of visual languages. In *Applications of Evolutionary Computation*, pages 271–280. Springer, 2010.

[30] Benoit B Mandelbrot. *Fractals: form, chance, and dimension.* WH Freeman San Francisco, 1977.

[31] Stephen F May, Wayne Carlson, Flip Phillips, and Ferdi Scheepers. Al: A language for procedural modeling and animation. Technical report, Citeseer, 1996.

[32] William J Mitchell. *The logic of architecture: Design, computation, and cognition.* MIT press, 1990.

[33] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. *Procedural modeling of buildings*, volume 25. ACM, 2006.

[34] Wojciech Palubicki, Kipp Horel, Steven Longay, Adam Runions, Brendan Lane, Radomír Měch, and Przemyslaw Prusinkiewicz. Self-organizing tree models for image synthesis. In *ACM Transactions on Graphics (TOG)*, volume 28, page 58. ACM, 2009.

[35] Seymour Papert. Teaching children to be mathematicians versus teaching about mathematics. *International journal of mathematical education in science and technology*, 3(3):249–262, 1972.

[36] Daniel Pletinckx. Quaternion calculus as a basic tool in computer graphics. *The Visual Computer*, 5(1-2):2–13, 1989.

[37] Ben Porter, Jon McCormack, James Wetter, and Alan Dorin. A procedural 3d modelling and animation system based on lua. Technical report, 2011.

[38] Aristid Lindenmayer Przemyslaw Prusinkiewicz, Aristid Lindenmayer, James S Hanan, F David Fracchia, and Deborah Fowler. The algorithmic beauty of plants with. 1990.

[39] Przemyslaw Prusinkiewicz. Graphical applications of l-systems. In *Proceedings of graphics interface*, volume 86, pages 247–253, 1986.

[40] Przemyslaw Prusinkiewicz, Radoslaw Karwowski, Radomír Měch, and Jim Hanan. L-studio/cpfg: A software system for modeling plants. In *Applications of Graph Transformations with Industrial Relevance*, pages 457–464. Springer, 2000.

[41] Casey Reas and Ben Fry. *Processing: a programming handbook for visual designers and artists*, volume 6812. Mit Press, 2007.

[42] Craig W Reynolds. Computer animation with scripts and actors. In *ACM SIGGRAPH Computer Graphics*, volume 16, pages 289–296. ACM, 1982.

[43] Rob Saunders and Kazjon Grace. Teaching evolutionary design systems by extending "context free". In *Applications of Evolutionary Computing*, pages 591–596. Springer, 2009.

[44] Ken Shoemake. Animating rotation with quaternion curves. In *ACM SIGGRAPH computer graphics*, volume 19, pages 245–254. ACM, 1985.

[45] Alvy Ray Smith. Plants, fractals, and formal languages. *ACM SIGGRAPH Computer Graphics*, 18(3):1–10, 1984.

[46] John M Snyder and James T Kajiya. Generative modeling: A symbolic system for geometric modeling. In *ACM SIGGRAPH Computer Graphics*, volume 26, pages 369–378. ACM, 1992.

[47] George Stiny and James Gips. Shape grammars and the generative specification of painting and sculpture. In *IFIP Congress (2)*, pages 1460–1465, 1971.

[48] Somlak Wannarumon, Erik LJ Bohez, and Kittinan Annanon. Aesthetic evolutionary algorithm for fractal-based user-centered jewelry design. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 22(01):19–39, 2008.

[49] Terry Winograd. Understanding natural language. *Cognitive psychology*, 3(1):1–191, 1972.

A

Appendix 1

Listing A.1: A subset of the MDG language grammar.

$$
\begin{aligned}
\mathit{commands} &\rightarrow \mathit{command}\ [\ \mathit{command}\ ]\\
\mathit{command} &\rightarrow \langle\ \mathit{startshape} \mid \mathit{rule} \mid \mathit{namespace} \mid \mathit{program} \mid \mathit{selector} \mid\\
&\quad\ \mathit{linerule} \mid \mathit{camerarule} \mid \mathit{normal} \mid \mathit{background} \mid \mathit{weight} \mid \mathit{color} \mid\\
&\quad\ \mathit{function} \mid \mathit{isosurface} \mid \mathit{maxiter} \mid \mathit{maxdepth} \mid \mathit{minsize} \mid\\
&\quad\ \mathit{duration} \mid \mathit{uniform} \mid \mathit{cameramode} \mid \mathit{camerafov} \mid \mathit{fps} \mid \ldots\ \rangle\\
\mathit{startshape} &\rightarrow \texttt{startshape}\ \mathit{rulename}\\
\mathit{rule} &\rightarrow \texttt{rule}\ \mathit{rulename}\ [\ \mathit{transform}\ ]\ \mathit{weight}\ \{\ \mathit{rulebody}\ \}\\
\mathit{program} &\rightarrow \texttt{program}\ \{\ \mathit{programcommands}\ \}\\
\mathit{programcommands} &\rightarrow \langle\ \mathit{shader} \mid \mathit{target} \mid \mathit{render}\ \rangle\\
\mathit{shader} &\rightarrow \langle\ \texttt{vertexshader} \mid \texttt{fragmentshader}\ \rangle\ \mathit{shadername}\\
\mathit{shadername} &\rightarrow \texttt{STRING}\\
\mathit{target} &\rightarrow \langle\ \texttt{screen} \mid \texttt{texture}\ \rangle\\
\mathit{selector} &\rightarrow \texttt{selector}\ \mathit{rulenameref}\ \langle\ \texttt{depth} \mid \texttt{scale} \mid \texttt{sequential} \mid\\
&\quad\ \texttt{pattern} \mid \texttt{function}\ \mathit{functionref}\ \rangle\\
\mathit{function} &\rightarrow \texttt{function}\ \mathit{functionname}\ \mathit{functionbody}\\
\mathit{functionname} &\rightarrow \texttt{STRING}\\
\mathit{functionbody} &\rightarrow \{\ \texttt{EXPR}\ \}\\
\mathit{functionref} &\rightarrow \langle\ [\ \mathit{namespaceref}\ ]\ \mathit{functionname} \mid \mathit{functionbody}\ \rangle\\
\mathit{transform} &\rightarrow \langle\ \mathit{convolve} \mid \mathit{animate}\ \rangle\ [\ \mathit{transform}\ ]\\
\mathit{convolve} &\rightarrow *\ \mathit{rulenameref}\\
\mathit{animate} &\rightarrow :\ \mathit{rulenameref}\\
\mathit{rulebody} &\rightarrow [\ \langle\ \mathit{ruleoriterator} \mid \mathit{terminal}\ \rangle\ ]\ [\ \mathit{rulebody}\ ]\\
\mathit{ruleoriterator} &\rightarrow \langle\ \mathit{iterator} \mid \mathit{ruleparams} \mid \backslash \mid \{\ \mathit{rulebody}\ \}\ \rangle\\
\mathit{terminal} &\rightarrow \langle\ \texttt{VERTEX} \mid \texttt{MOVETO} \mid \texttt{LINETO} \mid \texttt{FRAME} \mid \ldots\ \rangle\\
\mathit{iterator} &\rightarrow \mathit{iterations}\ [\ /\ \mathit{divisor}\ ]\ *\ \{\ \mathit{transforms}\ \}\ \mathit{ruleoriterator}\\
\mathit{ruleparams} &\rightarrow \mathit{rulenameref}\ \{\ \mathit{transforms}\ \}\\
\mathit{transforms} &\rightarrow \langle\ \mathit{translate} \mid \mathit{rotate} \mid \mathit{flip} \mid \mathit{scale} \mid \mathit{color}\ \rangle\ [\ \mathit{transforms}\ ]\\
\mathit{namespace} &\rightarrow \mathit{namespacename}\ \{\ \mathit{commands}\ \}\\
\mathit{rulename} &\rightarrow \texttt{STRING}\\
\mathit{rulenameref} &\rightarrow [\ \mathit{namespaceref}\ ]\ \mathit{rulename}\\
\mathit{namespaceref} &\rightarrow \mathit{namespacename}\ .\ [\ \mathit{namespaceref}\ ]\\
\mathit{namespacename} &\rightarrow \texttt{STRING}\\
\mathit{iterations} &\rightarrow \texttt{NUMBER}
\end{aligned}
$$

$$\begin{aligned}
\textit{divisor} &\rightarrow \texttt{NUMBER} \\
\textit{translate} &\rightarrow \langle\ \texttt{x}\ X\ [\ Y\ [\ Z\ ]\ ]\mid \texttt{x}\ X \mid \texttt{y}\ Y \mid \texttt{z}\ Z\ \rangle \\
\textit{rotate} &\rightarrow \langle\ \texttt{r}\ X\ [\ Y\ [\ Z\ ]\ ]\mid \texttt{rx}\ X \mid \texttt{ry}\ Y \mid \texttt{rz}\ Z\ \rangle \\
\textit{flip} &\rightarrow \langle\ \texttt{f}\ X\ [\ Y\ [\ Z\ ]\ ]\mid \texttt{fx}\ X \mid \texttt{fy}\ Y \mid \texttt{fz}\ Z\ \rangle \\
\textit{scale} &\rightarrow \texttt{s}\ S \\
X, Y, Z, S &\rightarrow \texttt{NUMBER}
\end{aligned}$$

B

Appendix 2

Table B.1: MDG directives

Directive	**Meaning**
startshape rulename	shapename is the default rule to be rendered when the program is executed
rule rulename [: ta [: tb [: tc ...]]] { ... }	prototype code for a rule with associated transforms ta, tb, tc, ...
program { ‹commands› }	prototype code for a program
linerule rulename	default rule to use for MOVETO and LINETO operations
camerarule rulename	transformation rule used for adjusting camera position and direction
namespace ns { ... }	defines a namespace section; rule names will be prepended ns
duration dur	the duration in seconds of the animation
isosurface name { ‹commands› }	declaration of an isosurface
function name { ‹expr› }	declaration of a function (mathematical expression with variables x, y, z)
selector rulename selectorname	defines a selector for the rule (valid options for selectorname is 'random', 'scale' and 'sequential'.)
normal x y z	the untransformed normal used by the VERTEX operation
fps num	num is the default fps (frames per second)
color a b c d	a, b, c, d are the strings that represent the four components of the color attribute (R, G, B, A by default)
weight a b c d	a, b, c, d are the strings that represent the four components of the weight attribute (u, v, w, q by default)
randseed num	integer that will be used as a seed value for the random number generator
uniform uniformname x [y [z [w]]]	sets a uniform in the shader program(s) to the values given by (x y z w)
uniform uniformname rulename prop	sets a uniform in the shader program(s) to the value given by property prop of the current frame of a rule (prop can be 'pos', 'dir', 'conj', 'color', 'weight' and 'normal')
angle anglemode	changes how angles are interpreted (anglemode may be one of 'degrees', 'arcdegrees', 'radians', 'orbits' and 'frequency')
minsize value	a minimum threshold value of the scale to indicate when recursion should stop
stacksize value	a maximum threshold value for the stack, to indicate when recursion should stop (advanced)
buffersize value	how many values to buffer in each asynchronous computation (advanced)
invalidateskipframes value	how many frames to skip before invalidating the current scene buffers
camerafov value	field-of-view angle of camera in degrees (default 40°)
cameranearplane value	camera near plane (default 0.1)
camerafarplane value	camera far plane (default 1000)
background r g b a	change the background color

Table B.2: MDG preprocessor directives

Directive	Meaning
@import project1 [project2 [project3 ...]]	imports source code and textures from projects
@presets preset1 [preset2 [preset3 ...]]	define preset names
@param int paramname defvalue minvalue maxvalue	defines an integer parameter
@param float paramname defvalue minvalue maxvalue	defines a float parameter
@param bool paramname defvalue	defines a boolean parameter
@fragmentshader shadername ‹shadercode› @endshader	defines a fragment shader named 'shadername'
@vertexshader shadername ‹shadercode› @endshader	defines a vertex shader named 'shadername'
@macro macroname(arg1 [, arg2 [, arg3 ...]]) ‹macrodefinition› @endmacro	defines a macro named 'macroname' with parameters named arg1, arg2, arg3, ...
@expand macroname(arg1 [, arg2 [, arg3 ...]])	expands a macro with the parameter values given by arg1, arg2, arg3, ...

Table B.3: MDG adjustment operators

Adjustment	Meaning
x x1 [y1 [z1]]	translation along the x-axis by x1, y-axis by y1, and z-axis by z1
y y1	translation along the y-axis by y1
z z1	translation along the z-axis by z1
r x1 [y1 [z1]] or rotate x1 [y1 [z1]]	rotation around the x-axis by angle by x1, y-axis by y1, and z-axis by z1
rx x or pitch x	rotation around the x-axis by angle x
ry y or yaw y	rotation around the y-axis by angle y
rz z or roll z	rotation around the z-axis by angle z
down [x]	rotate -90° or -x° around the x-axis
up [x]	rotate 90° or x° around the x-axis
left [y]	rotate -90° or -y° around the y-axis
right [y]	rotate 90° or y° around the y-axis
rollleft [z]	rotate -90° or z° around the z-axis
rollright [z]	rotate 90° or z° around the z-axis
s num	scale by num
f x y z	flip along the plane with a normal (x, y, z)
fx	flip along the x-axis
fy	flip along the y-axis
fz	flip along the z-axis
fps num	adjust fps by num
R num	adjust color component R by num (can be renamed)
G num	adjust color component G by num (can be renamed)
B num	adjust color component B by num (can be renamed)
A num	adjust color component A by num (can be renamed)
u num	adjust weight component u by num (can be renamed)
v num	adjust weight component v by num (can be renamed)
w num	adjust weight component w by num (can be renamed)
q num	adjust weight component q by num (can be renamed)

Table B.4: MDG program commands

Command	**Meaning**
vertexshader shadername	select a vertex shader for the next rendering step
fragmentshader shadername	select a fragment shader for the next rendering step
target screen	render to the screen (default)
target texture texturename	render to a texture (n.b. unimplemented)
render	render the shape given by the startshape command

Table B.5: MDG isosurfaces

Command	**Meaning**
x xmin xmax xres	defines the minimum and maximum values as well as the resolution along the x-axis
y ymin ymax yres	defines the minimum and maximum values as well as the resolution along the y-axis
z zmin zmax zres	defines the minimum and maximum values as well as the resolution along the z-axis
xyz min max res	defines the minimum and maximum values as well as the resolution for all axes
function { ‹expr› }	mathematical expression to be evaluated at each grid point of the isosurface
function funcname	reference to a function that will be evaluated at each grid point of the isosurface

Table B.6: MDG rule operators

Operator	**Meaning**
VERTEX {}	a vertex is defined at the current position (every three vertices makes up a triangle)
[namespace.]**MOVETO** [rulename] **{}**	moves to the context of the namespace current position, rasterizing the path of the rule 'rulename' (if provided) or else the rule given by the 'linerule' command
[namespace.]**LINETO** [rulename] **{}**	works like MOVETO, but also emits triangles that connects with the last MOVETO/LINETO position
[namespace.]**BLENDTO** [rulename] **{}**	similar to LINETO, but smoothly joins two orientation frames through several smaller line segments
FACENORMALS {}	recomputes the normals for the last three vertices so that they represent the normal of the triangle plane
FRAME {}	emits an animation frame at the current position
[namespace.]**RESET {}**	reset transforms to the MOVETO context of the associated namespace
[namespace.]**DEBUG {}**	write information to debug console output

C

Appendix 3

Listing C.1: Basic vertex and fragment shaders used for models.

```
@fragmentshader FS
  precision mediump float;

  varying vec3 normal;
  varying vec4 vertPos;
  varying vec3 cameraPos;
  varying vec4 weight;
  varying vec4 color;

  uniform vec3 lightPos;
  uniform vec3 specColor;
  uniform float phongPower;

void main() {

  vec3 normal = normalize(normal);
  if (!gl_FrontFacing) {
    normal = -normal;
  }

  vec3 lightDir = normalize(vertPos.xyz - lightPos);
  float lambertian = max(dot(lightDir,normal), 0.0);
  float specular = 0.0;

  if(lambertian > 0.0) {
    vec3 reflectDir = normalize(reflect(lightDir, normal));
    vec3 viewDir = normalize(cameraPos - vertPos.xyz);
    float specAngle = max(dot(reflectDir, viewDir), 0.0);
    specular = pow(specAngle, phongPower);
  }

  gl_FragColor = vec4( lambertian*color.rgb +
                         specular*specColor, 1.0 + color.a);
}
@endshader
@vertexshader VS
     attribute vec4 aVertexPosition;
     attribute vec3 aVertexNormal;
     attribute vec4 aVertexWeight;
     attribute vec4 aVertexColor;
     attribute float aVertexBone;

     uniform mat3 uNMatrix;
     uniform mat4 uPMatrix;
     uniform mat4 uBoneMatrix[60];
     uniform vec3 uCameraPos;

     varying vec4 vertPos;
     varying vec3 cameraPos;
     varying vec3 normal;
     varying vec4 weight;
     varying vec4 color;

     mat3 getNormalMat(mat4 mat) {
        return mat3(mat[0][0], mat[1][0], mat[2][0], mat[0][1], mat
               [1][1], mat[2][1], mat[0][2], mat[1][2], mat[2][2]);
     }

     void main(void) {
       mat4 uMVMatrix = uBoneMatrix[int(aVertexBone)];
       vertPos = uMVMatrix * aVertexPosition;
       cameraPos = uCameraPos;
       gl_Position = uPMatrix * vertPos;
       normal = aVertexNormal * getNormalMat(uMVMatrix);
       weight = aVertexWeight;
       color = aVertexColor;
     }
@endshader

program {
  vertexshader VS
  fragmentshader FS
  target screen
  render
}
```

Listing C.2: Model describing a class of knots with a number of revolutions about each axis.

```
startshape START
linerule LINERULE
angle orbits

rule LINERULE {
  12/12*{rz 1} VERTEX{x 1}
}

rule START{
  NEWLINE{}
  ring{s 2 R 1 G 0.3}
}

rule ring * zpath : ypath : xpath {
  ALINETO{z 0.2 s 0.3}
}

rule zpath {
  4000/4000*[rz 9] FRAME{}
}

rule ypath {
  4000/4000*[ry 81] FRAME{}
}

rule xpath {
  4000/4000*[rx 16] FRAME{}
}
```

Listing C.3: Model generated with transform rules.

```
uniform lightPos 0 0 -400
uniform specColor 1 1 1
uniform phongPower 40

startshape START
minsize 0.0015
duration 60

linerule LINERULE
camerarule CAMERARULE

normal 0 0 1

// adjust camera distance here (-5)
rule CAMERARULE {
  FRAME{z -20}
  CAMERARULE {}
}

// 12-sided extrusion along the line
rule LINERULE {
  12/12*[rx 360] VERTEX{z -1}
}

// render two spiral rings
rule START : transform{
  NEWLINE{}
  spiralring{R 1 B 0.5}
  NEWLINE{}
  spiralring{rz 36 B 1 G 0.5}
}

// spiralring uses ringpath as its 'iterator'
rule spiralring * ringpath : firstspiral : secondspiral {
  LINETO{s 0.25}
}

// the outer circle (2000 steps)
rule ringpath {
  2000/2000*[rz 360] FRAME{x 5}
}

// spiral with 5 twirls (2000/400)
rule firstspiral {
  2000/400*[ry 360] FRAME{x 0.75}
}

// spiral with 100 twirls (2000/20)
rule secondspiral {
  2000/20*[rz 360] FRAME{y 0.65}
}

// animation frames
rule transform{
  FRAME{}
  transform{ry 1 rx 0.7 rz 0.3}
}
```

Listing C.4: Silicon carbide molecular model.

```
rule SILICON {
  // the 's' parameter adjusts the scale
  SPHERE{s 0.2 s 1.17 G 0.8}
}

// Carbon, 0.25 * (0.077/0.117)
rule CARBON {
  // the 'R', 'G', 'B' parameters adjust the color
  SPHERE{s 0.2 s 0.77 R 0.2 G 0.2 B 0.2}
}

rule BOND {
  MOVETO{}
  LINETO{z 1}
}

//selector SiC function { max(x,y,z) }
// face-edge-face angle: 70.5288 degrees
rule SiC {
  // CARBON{z -1}
  BOND{z -1}
  SILICON{}
  3*{rz 120} 1*{ry 70.5288} {
    CARBON{z 1}
    BOND{}
  }
  // move to the right position for the next SiC
  MOVETO{z 1 ry 70.5288 z 1 ry -70.5288}
}

rule ABCB {
  SiC {} \     // A
  SiC {} \     // B
  SiC {fx} \   // C (flip x axis)
  SiC {} \     // B
  MOVETO{fx}   // (flip x axis -- for the next ABCB)
}

rule LAYERS {
  // height of pyramid: sqrt(6)/3 = 0.8164965809
  // height of two pyramids: 2*sqrt(6)/3 = 1.632993162
  // edge-to-opposite-edge x 2: sqrt(2) = 1.414213562
  // 4 repetitions along the [1120] direction
  // 3 repetitions along the [1100] direction
  // 2 repetitions along the [0001] direction
  4 * {x 1.414213562 y 0.8164965809} 3 * {y 1.632993162} 2 * {} \ ABCB{}
}

rule START {
  LAYERS{x -3 y -4 z -2}  // adjust the origin
}
```

Listing C.5: Dodecahedron inscribed within the Coxeter complex (cover page illustration.)

```
@import projects/basic-shaders

uniform lightPos 0 0 -400
uniform specColor 1 1 1
uniform phongPower 40
background 0.4 0.6 0.8 1

startshape START
minsize 0.0015
duration 60

maxiter 4000
maxdepth 1000

linerule LINERULE
camerarule CAMERARULE

normal 0 0 1
angle orbits

rule CAMERARULE {
  FRAME{z -74}
  CAMERARULE {}
}

rule TRANSFORM {
  FRAME{}
  TRANSFORM{ry 1d rx 0.7d rz 0.3d}
}

rule LINERULE {
  12/12*[rx 1] VERTEX{z 0.1}
}

isosurface SPHERE {
  xyz -1 1 6
  function {1.0 - (x*x+y*y+z*z)}
}

// a = 4*r/(3+sqrt(5))
// r = 1  =>  a = 4/(3+sqrt(5)) = .7639320224
rule PVERTEX{
  VERTEX{ry -0.7639320227t z 10}
}

rule pentagon{
  NEWLINE{}
  PVERTEX{}
  PVERTEX{rz 0.2}
  PVERTEX{rz 0.4}
  FACENORMALS{}
  PVERTEX{rz 0.4}
  PVERTEX{rz 0.6}
  PVERTEX{rz 0.8}
  FACENORMALS{}
  PVERTEX{}
  PVERTEX{rz 0.8}
  PVERTEX{rz 0.4}
  FACENORMALS{}
}

rule START : TRANSFORM {
  dodecahedron{s 2 B 0.4 R 0.6 G 0.8}
  symplanes{s 2 G 0.8 R 1}
}

rule dodecahedron {
  dodec_faces{fz}
  5/5*{rx 1} dodec_faces{ry -2t}
}

rule dodec_faces {
  pentagon{ry -0.25}
  pentagon{ry 0.25}
}

rule symplanes {
  5/5*{rx 1} dihedral{}
}

rule dihedral {
  circle{fz}
  1*{ry -2t} 2/5*{rx 1} circle{}
}

rule circle {
  SPHERE{ry 0.25 z 10 B 0.8 G -0.4 R -0.8 s 0.6}
  SPHERE{ry -0.25 z 10 B 0.8 G -0.4 R -0.8 s 0.6}
  torus{}
}

rule torus {
  NEWLINE{}
  MOVETO{z 10}
  48/48*{ry 1} LINETO{z 10}
}
```