A system for modelling deformable procedural shapes.

Lewis, Timothy Luther

A System for Modelling Deformable Procedural Shapes

Timothy Luther Lewis

A thesis submitted to the University of Wales in
candidature for the degree of Philosophiae Master

Department of Computer Science
University of Wales, Swansea
March 2004

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

DECLARATION

This work has not previously been accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Signed . .....(candidate)

Date.....30/7/2004.....

STATEMENT 1

This thesis is the result of my own investigations, except where otherwise stated.

Other sources are acknowledged by footnotes giving explicit references. A bibliography is appended.

Signed ....(candidate)

Date.....30/7/2004.....

STATEMENT 2

I hereby give consent for my thesis, if accepted, to be available for photocopying and for inter-library loan and for the title and summary to be made available to outside organizations.

Signed . .....(candidate)

Date.....30/7/2004.....

ACKNOWLEDGMENTS

The author wishes to thank the following individuals and organizations, in no particular order, without the help of whom I could not have completed this thesis:

Dr. Mark W Jones, Swansea University.

Professor Min Chen, Swansea University.

Martin Robinson, CEO Themekit LTD.

福原居士, CEO AiCube LTD.

Euler, Euclid, Pythagoras, Blinn et al.

Mark Kiddel, Hau Lin Zhou, Ann Smith, Geraint Howell and everyone else in 'the lab' for their friendship and support.

Chani Weaver, for her love and showing me how to carry on.

Tirion Awen Lewis, for providing an excellent reason to hurry up and finish.

My Family, without whose unwavering support and encouragement, I could not have continued.

And finally,

Japanese Immigration, Otemachi, Tokyo. Without whose professional lack of sympathy, I really wouldn't be here today.

ABSTRACT

This thesis presents a new procedural paradigm for modelling. The method combines the benefit of compact object descriptions found in procedural modelling along with the advantage of the ability to interact in real-time as is found with interactive modelling techniques.

The three main components to this paradigm are geometry generators (the creation of basic object shapes), selectors (the specification of a selection volume), and modifiers (the object transformation functions).

The user interacts in real-time with the object, and has complete control over the object formation process. Interaction is stored within appropriate nodes in a creation-history list which can be replayed or partially replayed at any time during the creation process. The parameters associated with each interaction are stored within the node, and are available for editing at any time during the creation process. The concepts presented here remove the problems that most modelling software have, in that the arbitrary editing of object parameters is destructive, in the sense that changing the parameter of one node may cause the object to behave unpredictably. This takes place in real-time, rather than off-line. In some cases real-time interaction is made possible by trading visual quality vs. speed of rendering. This results in the object being rendered at a lower quality, and therefore decisions on whether the object parameters need adjustment may be predicated upon a poor representation of the object. The work presented herein attempts to bridge the divide between the two approaches by providing the user with a powerful and descriptive procedural modelling language that is entirely generated through real-time interaction with the geometric object via an intuitive user interface. The main contributions of this work are that it allows:

- Procedural objects are specified interactively.
- Modelling takes place independently of representation (meaning the user does not base their modelling on the (mesh) representation, but rather on the shape they see)
- Changes to the object are coherent and non-destructive.

Procedural modelling, deformation, CSG, 3D modelling software, scenegraph, node.

Terms and definitions

For the remainder of this document, the following terms and definitions shall be used:

- **Primitive**: Any base primitive defined by the system.
- **Modifier**: Any operation that changes or distorts a base primitive so that it's original form is altered in a way that renders the original mathematical definition of the object invalid.
- **Node**: Any operation on or definition of an object or resource.
- **Selection**: This refers to a user-defined region of an object on which a modification will take place.
- **View**: This refers to the view-port in which a selection was made or a modification performed. In the case of the software in which the method described herein is concerned, this either refers to an orthographic view or a perspective view.
- **'The System' or 'The Application'**: this refers to the software in which the method described in the document has been implemented.
- **Material**: These nodes specify a shading model for 3D objects. These nodes often take other nodes (such as textures, or even other materials) as inputs and composite these inputs when rendering an object.
- **Texture**: A 2D or 3D image based texture; either procedural or stored as individual pixels or voxels. Textures are usually inputs to materials and cannot be used to specify shading of an object without the use of a material.
- **Palette**: Palettes can be considered as one dimensional textures and contain colour table data. This class of node is often used as an input to a texture or material.
- **Resource**: Typically this refers to texture, material, palettes and other auxiliary scene data.
- **Gizmo**: A 3D user interface element used for manipulating objects in 3D display views.
- **I/O Filter**: Refers to program code used to import data from other applications.
- **Geometry**: Refers to vertex position data. In the case of the system described, this term is used to describe lists of vertices.
- **Topology**: This refers to the connections between vertices. For example, the geometry of a cube would be the 8 vertices of that cube, the topology is considered to be the faces of the cube that reference the vertices. When discussing topology, the two core elements are edges and faces.
- **Scenegraph**: This term refers to the structure of data within a 3D visualization or modelling system. This term is used when referring both to the data organization within the system described herein and when referring to the manner in which other 3D applications and systems organize their data. This term refers both to the runtime data representation of scene data and also to scenegraph file formats.

TABLE OF CONTENTS

Chapter 1

INTRODUCTION

1.1 Summary

In this thesis, a new 3D modelling system, called ClayWorks, will be presented. This system presents a new method for creating, generating and storing both 3D data and the auxiliary objects that make up a 3D scene.

1.1.1 Procedural modelling

In this thesis we define procedural modelling as a process by which complex data can be generated from limited parameterized input. This is opposed to explicit geometry and bitmaps and techniques for compressing such data; this thesis only briefly touches upon areas of data compression. The main focus is on data generated and managed by user input and importantly, what useful things can we do with the knowledge of how a user built an object, including using this information for both compression (what redundant steps can be removed or re-expressed?) and future editing. In a sense, the steps taken to describe an object become in themselves tools for the future editing of an object.

This thesis describes not only a means of describing objects procedurally but also gives methods for creating and interacting with these procedural methods in an intuitive an meaningful way. This is significant as procedural methods are usually limited to grammar based 3D modelling systems, with little visual user feedback. There's a definite place for such grammar based automatic generation of scenes but this should compliment intuitive user input, rather than replace it.

Real-world metaphors for the creation of 3D data include clay modelling, machining, lathing and other techniques that we can intuitively grasp. The aim of the user interface portion of this project is to create procedural models but to do so in a manner that the user of the software feels in control of the output.

These techniques are not limited to 3D data but also to 2D illustration and 1D control curves. With one system for describing visual data and operations upon that data we can create and store 3D scenes of great complexity but at a fraction of the storage space (and therefore, transmission cost) and also have, as part of their description, useful semantics about their creation embedded within the scene.

The greater aim of this project is to be able to use it to create interactive 3D and/or 2D applications where resource data is mostly generated, rather than stored. The system described in this thesis is also designed to be flexible, where third party additions to the system have the same level of access and control as those that form the base functionality; if a third party developer is unhappy with the implementation of a certain part of the system, they should be free to replace it with their own component and, as long as the interfaces with existing components are correctly implemented, new additions to the system can be integrated seamlessly.

1.1.2 **Existing modelling software**

The system described in this thesis is not intended to replace existing 3D design packages; such programs, including 3D Studio Max [DISCREET1], Maya [MAYA1] and Softimage [SOFTI1] are mature and industry tested applications but focus largely on the creation of output intended for recorded media such as film and television. As such, these programs are largely unconcerned with interaction and the efficient means of describing 3D data. However, these programs are used for the creation of interactive content simply because no better alternative exists and there *is* a great deal of overlap between the needs of computer graphics in film and in interactive media such as games.

1.1.3 **Novel features of this system**

However, the system described here should not be considered a poor cousin to such programs or as a 'low-polygon' modeller. Many of the tools available within this system are similar to those found in other programs; the real change is in the focus of how objects are stored, away from a final explicit representation such as a triangular mesh or bitmap and instead focusing on the creation of the object as a means to store, describe and control the final shape. 3D Studio Max utilizes a similar concept with its modifiers but this system will often produce unexpected results when the user tries to change the parameters of past modifiers. Also, the libraries used to generate the objects are not available to third party developers for use in their own systems so when these objects are to be used in another system, they must first be stored in some explicit format.

The system presented in this thesis specifies a file format that references instances of 'node' classes and how these are connected together; this graph describes how objects are constructed and also how they are displayed, in a similar manner to the 'observer' design pattern. The class library that is provided by the system is extensible in a manner similar to java; third parties add their own classes and add to the functionality of the language. In this

case, the language is a formal means of describing graphical data, behaviour and interaction and the system for creating this data is an intuitive WYSIWYG tool.

In this way a game or other interactive 3D application would, instead of reading in large mesh and texture files, include the generative part of the modelling software within their distribution classes. The 3D data within the game would be generated by instancing the same classes that the modelling system itself uses. One of the largest bottlenecks in modern interactive graphics applications is in downloading large data files (textures and meshes) either from a disk or over a network connection. Generating this data procedurally transfers the load to the processor and/or graphics processor which opens up new avenues for optimizing the data. Also, with the core modelling tools built into the end application software, real-time deformation of data utilizing the same suite of tools used to model that data becomes a possibility.

Whilst the system still needs improvement, the results so far are impressive. The following images are screen shots of the application in action. The triangular mesh output for the Harley Davidson model (Figure 1) contains over 15,000 triangles yet compressed, only occupies 14.4kb of disk space. The object takes less than a second to generate from its procedural description. The lighthouse scene (Figure 2) takes 40k, mostly due to the height map data used to describe the island itself. This scene takes a little longer to generate (approximately two seconds) due to the current implementation of Perlin noise (used to generate the waves and the clouds), which could be far more efficient.

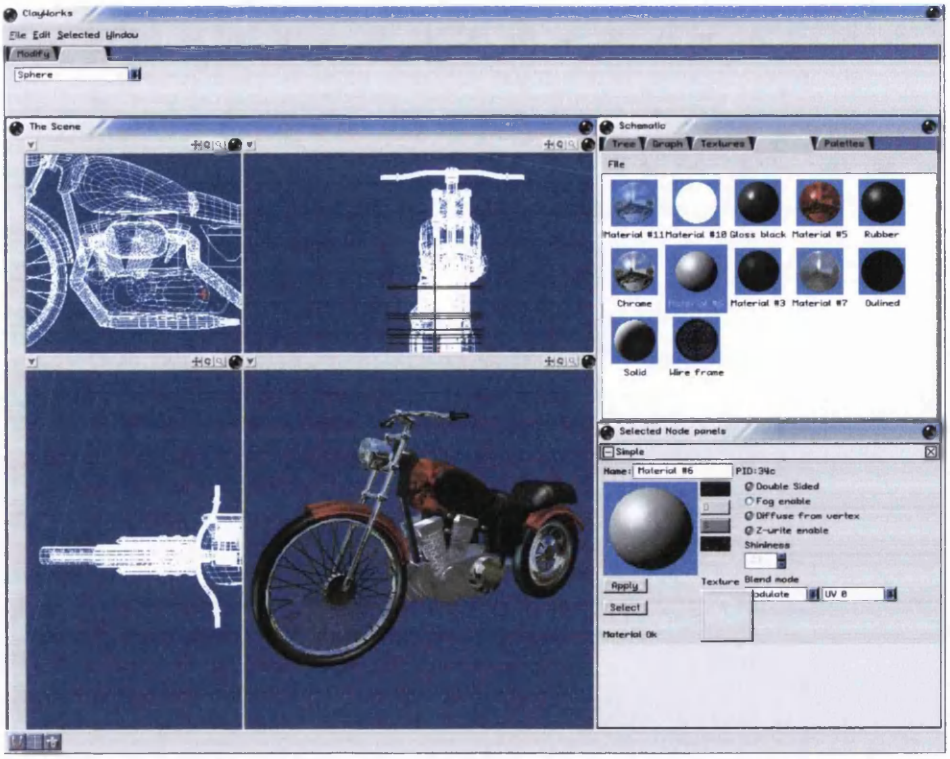


Figure 1. Screen shot of the application in action showing the Harley Davidson model.

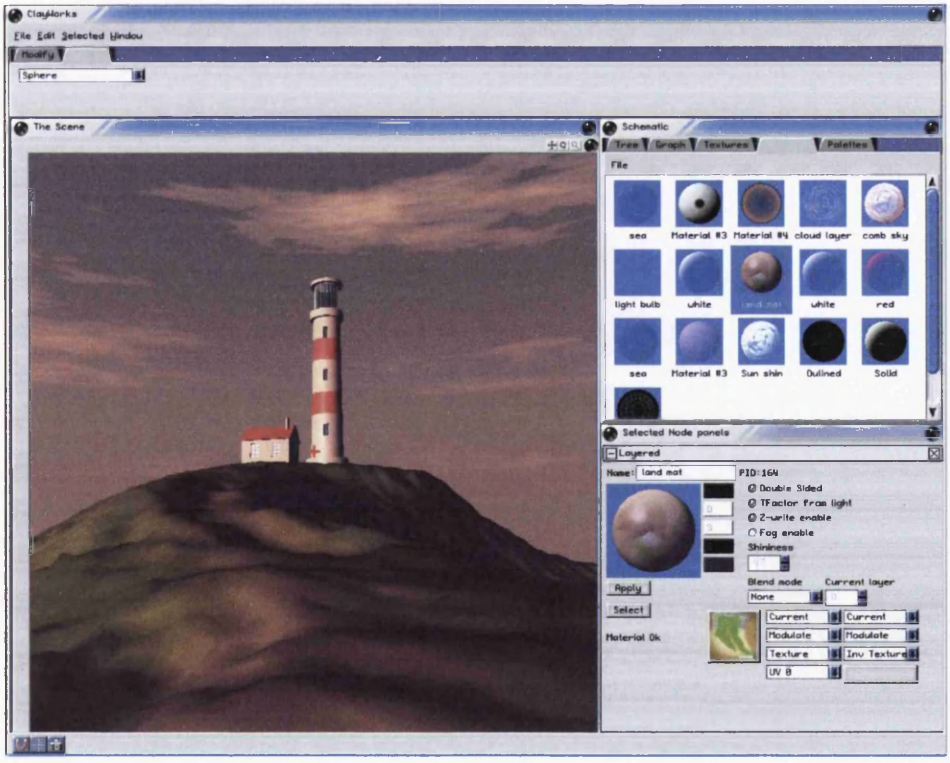


Figure 2. Screen shot of the application in action, showing the Island Scene.

This work has been presented as a full paper at WSCG which was held in February 2004 at The University of West Bohemia in Plzen [LEWS]. The talk outlined the work and emphasized the space savings gained by storing models in a generative, procedural manner. A live demonstration of the software was given, showing clearly that the time taken to load and then generate complex scenes was minimal.

Chapter 2

PREVIOUS WORK

2.1 Introduction

The system described in this thesis strives to provide real-time manipulation of 3D objects created using procedural methods and also modified using procedural modifiers. In the following section we shall examine various methods used to manipulate 3D objects on a 2D display and also methods to both generate and manipulate 3D objects in a procedural manner.

Procedural modelling has been used extensively to generate naturally occurring phenomena and Section 2.2 contains an overview of some of these methods. Most of the methods for creating and manipulating 3D objects procedurally do so in an off-line non-interactive manner. This includes *generative modelling* [SNYD] a technique used to describe procedurally generated and manipulated shapes where the high-level description of the how an object was created is essentially the object definition, rather than a lower level description such as a polygon mesh or voxel dataset. Generative modelling techniques are discussed in the next section.

Section 2.3 covers ways to represent and manipulate 3D forms, including deformation lattices, lofting and lathing of 2D shapes in three dimensions and also a section on generative modelling, an important method for the specification of 3D shapes.

Section 2.4 is an investigation into generative modelling techniques, which forms a counterpoint to techniques for storing and compressing low-level (e.g, triangle meshes) descriptions of data, which are discussed in section 2.5.

Section 2.6 contains a detailed overview of various methods used to interact with 3D shapes on a 2D display. Many of these techniques are used in commercially available modelling systems and some of these programs are reviewed in Section 2.7.

2.2 Procedural modelling

Procedural modelling refers to a means of creating objects parametrically. That is to say,

complex or dense low-level geometry is created from terse high-level descriptions. The low-level representation of the object might be a polygon mesh or voxel data set. Intermediate means of storing and representing objects, such as NURBs, also exist and these offer more control over the curvature of a shape than a polygonal mesh or voxel data set but are usually used to approximate higher-level (and terser) descriptions of base primitives such as spheres, cubes, cylinders and so on.

Procedural modelling covers a large area and a great deal of different methods for creating different classes of object.

Landscapes [LYCH] [MAND] [MAXN] [MILL], clouds [NISH1] [NISH2] [NISH3] [NISHI4] [SCHP] [DOBA] [DOBA2] [MIYA], water [TSND] [NISHI4], plant life [LIND], cities [PAMU] and many other natural forms have been modelled using a wide variety of techniques ranging from fractal algorithms, noise generators, L-systems, cellular automation, CSG, and many other techniques. Wide though this field is, all of these techniques share one factor in common: the generation of complex visual effects from a small set of parameters.

One area that has received a lot of attention in the field of procedural modelling is the creation of natural phenomena that would be extremely tedious and time consuming to create by hand.

2.2.1 L-systems

Lindenmayer-Systems (hereafter referred to as L-systems) are a class of string re-writing algorithms developed by Lindenmayer [LIND] to study the form and growth of various organisms.

L-systems refer to a class of algorithms that utilize grammar rules to constrain and control the growth of various natural systems. L-systems have been used to generate many different types of phenomena including plants and cities. By supplying a grammar, similar to that used in computational theory, to constrain development of a system, such algorithms can be used to generate a wide variety of complex forms. L-systems differ from computational grammars in that they can apply productions in parallel, rather than the serial operations supported by grammars. Also, there are languages that can be generated using context free L-systems that cannot be generated by context free grammars.

The simplest forms of L-systems are context-free and deterministic. The grammars

found in these L-systems allow basic replacement. Consider the simple alphabet *'ab'*. A simple set of grammar rules for this alphabet might include $a \rightarrow ab$, $b \rightarrow a$. In the context of creating a plant using L-systems, one might have an alphabet such as this:

trunk, branch, leaf

And a grammar for such a system might consist of the following:

trunk → *branch*

branch → *branch/ leaf*

More complex implementations of L-systems in computer graphics introduced turtle like commands for which specify certain drawing operations.

In 2001, Parish and Muller [PAMU] describe a system for generating entire cities using their CityEngine software. Their method relies on underlying topological maps containing height data as well as land, water and vegetation data. Their system also requires sociostatistical maps for population density, zoning (i.e, residential/ industrial/ commercial zones), street patterns for controlling the behaviour of street generation and height maps for controlling the maximal heights of buildings. The system then generates street data and building allotments procedurally. Their system utilizes two different L-systems, one for designing road layout and another for the generation of buildings. They also implement façade texture on buildings using a suitable grid structure, where grid elements correspond to common features on a building, such as doors, windows or wall elements. The relative size of these elements can influence each other; e.g, windows on the ground level or above doors are resized accordingly. Stored textures or procedural textures are then superimposed onto this grid structure, resulting in convincing façades for the surfaces of buildings.

Figure 3. This example shows the real map of Manhattan (bottom image) compared with that generated by CityEngine's [PAMU] 'New York' Rule (centre image).

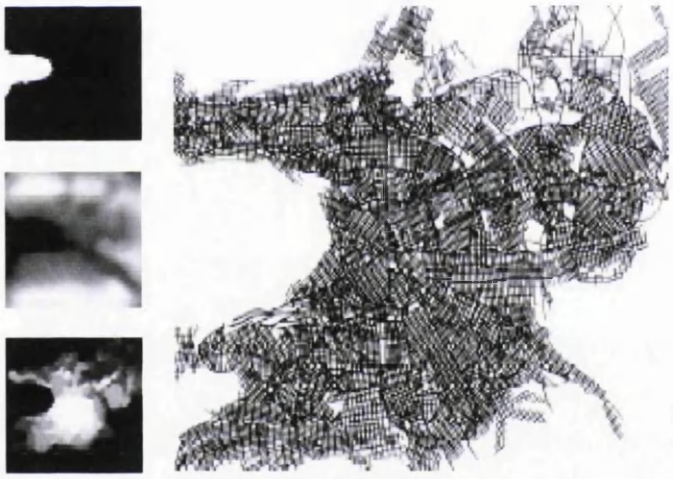


Figure 4. This image shows the various maps used to define behaviour within CityEngine [PAMU]. The maps to the left are used to modify building height, terrain and population density.

The first step in generating a procedural city is to create the street map. Parish and Muller have implemented an L-System for this which is called *extended L-systems*. In their system, strings that are fed into the L-System are modified by *global goals* and *local constraints*. The global goals are defined as the street patterns and population density and the system will endeavor to first create highways linking population centers. The system first finds areas of high population and then shoots rays from these centers of a pre determined length. The population density at each sample point along this ray is summed up and the ray with the

greatest population is chosen for continuing road growth. Smaller roads are then created connecting together main roads.

Their system also relies on several rules, or models for road growth. The type of model used is determined from the street patterns input data map and they fall into the following categories:

Table 1. Patterns used in CityEngine to define growth behaviour.

Pattern name	Pattern
Basic	This pattern simply follows areas of population density. This is the 'natural' model for town or city growth and mimics the layout of older towns. All other rules are based on this rule but with certain constraints for road length and branch angles.
New York	The New York rule imposes a rectilinear pattern upon street development with a given constraint for maximal block length.
Paris	This rule attempts to create a radial development of highways around a central point as can be seen in some large cities such as Paris or Tokyo.
San Francisco	This rule favors streets and highways that have the smallest change in elevation with smaller roads connecting main roads of different elevation.

The L-systems found within *CityEngine* are self-sensitive and the system is aware of

roads that form dead ends or those that create closed, isolated loops. Most L-systems rely on branching operations that end in a leaf or terminal node. In road layouts, terminal dead end roads are the exception rather than the rule. To solve this problem, *CityEngine* parses the road map to search for dead end streets and then modifies the street so that if two roads intersect, an intersection is created, if a road ends near an intersection it is extended to meet it and if a road is close to intersecting with another road, it is extended to form an intersection.

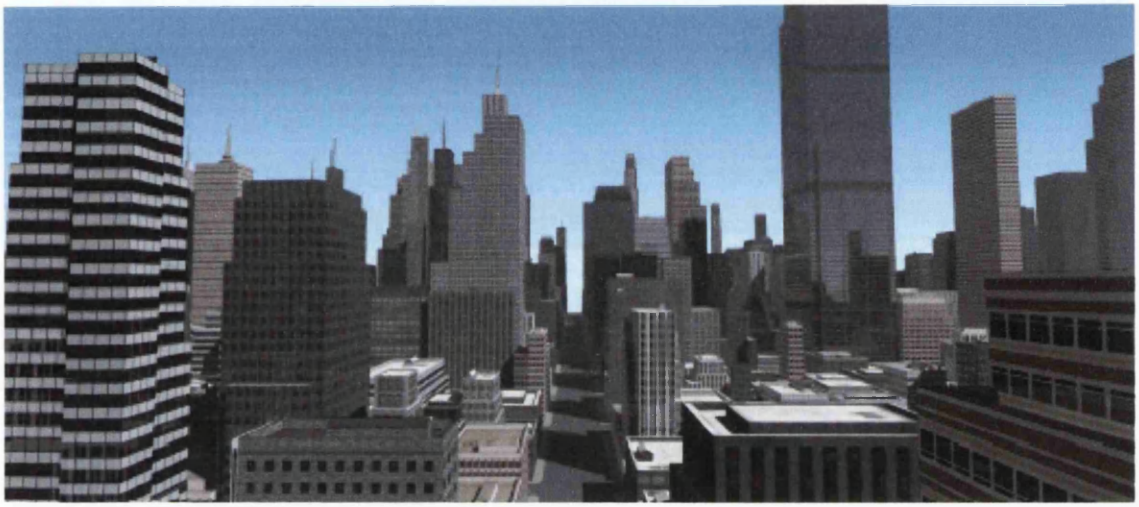


Figure 5. 'Virtual Manhattan', a rendering of a city created by CityEngine [PAMU].

Once a road map is generated, this is passed on to the algorithm that generates allotments which are then extruded to form buildings. The size of the allotments depends on land usage, population density, access to street (allotments with no street access are not generated) and building height, specified by a height map.

L-systems provide a powerful means to create a wide variety of procedurally generated objects and can also be used to study the growth of such systems. As such, L-systems are of use in other fields. Indeed, they were originally created to study the development of plants, rather than as an artist's tool to create 3D objects. In many instances, intuitive creation and scope for artists' license are more important than biological accuracy and another class of procedural algorithms have been developed to cater for this user group.

Kruszewski [KRUS] et al. proposed a system for creating trees and bushes based on their silhouette profile. They describe their system as separate from what they call *developmental* algorithms such as L-systems which model the growth of a plant based on a

grammar. Their algorithm is a top down, rather than a bottom up system as they define a bounding shape which is used to constrain the shape of a given tree. Whilst they freely admit that this technique is not rigorous from the biological science point of view, it does provide a more intuitive tool for the graphics professional who does not have a grounding in biology.

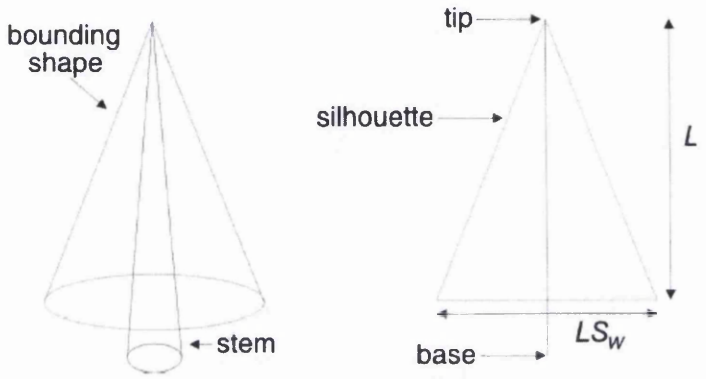


Figure 6. This image shows an example of a bounding shape, used to constrain the shape of trees in Kruszewski's system [KRUS].

They do not, for instance, model nutrient flow or exposure of leaves to sunlight in order to create their plants. However, their algorithm is an ideal artists' tool for creating trees for use in a real-time or rendered environment. In their algorithm, they divide the tree into four main types of feature: the *main stem*, or trunk, the *boughs*, or initial offshoots from the trunk, *limbs* and finally *twigs* which terminate the process. Leaves are modelled as a post processing step and their generation is largely ignored in their paper. Their algorithm depends on several global parameters, for the whole tree and local parameters which are fed through each layer.

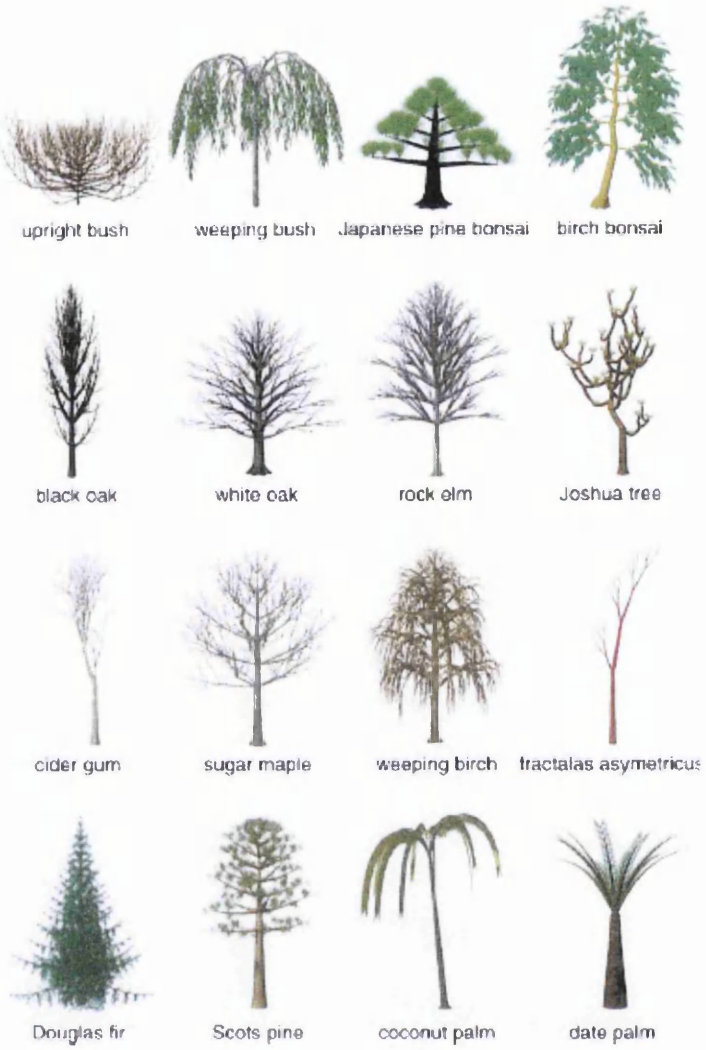


Figure 7. Various plant forms created using Kruszewski's system [KRUS].

2.2.2 Procedural noise generation

Computer graphics have often been accused, rightly or wrongly, of being too 'perfect', too smooth and lacking the random appearance of the real world. Whilst a lot of this criticism is misplaced, methods for simulating the complexity and imperfection of the real world are clearly necessary if we are to create convincing computer generated images of it. To this end, algorithms such as fractals and Perlin noise attempt to add some chaos and naturalistic grime to our virtual scenes.

Perlin [PERL1] devised a noise generation function that is widely used in the generation of procedural effects. Perlin noise is a simple, elegant algorithm that produces a texture or volume by summing together layers of noise, increasing in frequency and decreasing in amplitude in each layer. The noise is interpolated so that low frequencies generate smooth

curves of high amplitude noise that represent the body of the noise. Subsequent passes of the algorithm generate higher frequency, lower amplitude noise for greater detail. Perlin noise has successfully been used to generate a wide variety of naturally occurring phenomena, from landscapes to clouds. By perturbing layers of noise by a sine function, one can use this function to generate convincing wood textures and by adding an exponential function to the layers of noise, it is possible to generate cloud textures. Whilst more specialist approaches to various natural occurring phenomena often generate more convincing results (such as terrain generation through the simulation of natural effects such as erosion or the simulation of clouds using meteorological models), Perlin noise is still often used in conjunction with other methods to create a more random, naturalistic appearance.

The following function shows how Perlin noise is generated.

$$turbulence(x) = \sum_{i=0}^{k} abs(noise(2^i x)/2^i)$$

The *noise(x)* function, in this example, is in one dimension. For each *I* the frequency of the noise increases as a power of 2. The noise is then divided by this power of 2 to lower the amplitude of the noise. At lower frequencies, the noise is interpolated to provide smooth results. Linear interpolation is adequate for real-time generation of Perlin noise but other methods of interpolation are preferred; cubic interpolation produces the best results although cosine interpolation, as shown in the following function, generates good results at a lower computational cost.

$$y = a(1 - \frac{1-\cos(x*\pi)}{2}) + b(\frac{1-\cos(x*\pi)}{2})$$

where a is the first point considered, b is the last and x is a normalized value between 0 and 1. Cosine interpolation achieves noticeably smoother results than linear interpolation and whilst cubic interpolation produces the most natural looking curve, cosine interpolation gives good enough results for most applications and makes considerable saving in computational cost.

$A=(v_0 - v_1)$;
$P=(v_3 - v_2)$-A;
Q=A-P;
$R=v_2 - v_0$

$r= Px^3 + Qx^2 + Rx + v1$

Cubic interpolation, giving very smooth results at a higher computational cost, where v_0, v_1, v_2 and v_3 are the sampled values of the function and x is a normalized value between 0 and 1, representing a point between v_1 and v_2.

The noise generation algorithms must be consistent; i.e, for a given seed, octave and input value, the noise generation algorithm must return the same output value and yet also return convincingly random results. In addition, Perlin noise generators most commonly use different seeds for different octaves of noise, to avoid aliasing in repeating patterns.

Perlin et al. [PERL2] have also devised a system for introducing procedural detail on objects as a part of a subdivision process. Their process does not deal only with 3D volumes as a means to create additional detail, they also deal on the UV domain of a surface representation and introduce procedural detail in a manner somewhat analogous to bump mapping.

Their approach is an extension of mesh subdivision techniques using Catmull-Clark subdivision [CATM]. Whereas subdivision techniques create a smoother version of a coarser polygonal mesh, the technique described in this paper creates additional surface detail as the mesh is subdivided. Perlin also states that whilst it is necessary to synthesize a finer resolution mesh from a coarser hull, it is also necessary to analyze a higher resolution mesh to create a coarser version, in order to add procedural detail to a higher resolution mesh.

A distinction is made between transforms that occur in global space, i.e, the parameters for the procedural shape modification are the x, y and z coordinates of vertices lead to *volumetric shape definitions.* Other deformation techniques based on the surface (u, v) domain lead to *surface shape definitions* where new features are 'grown' on the surface of the shape. Several deformation models are mentioned in this paper, including techniques for creating fractal rocks, berries, tentacles and 'mushroom clouds'. Fractal rocks are possibly the most

familiar of these models, utilizing Perlin noise to add random turbulence at each stage of mesh refinement.

Figure 8. This image, from Perlin et al's system for adding detail to subdivision surfaces [PERL2]. The procedural rock model is shown.

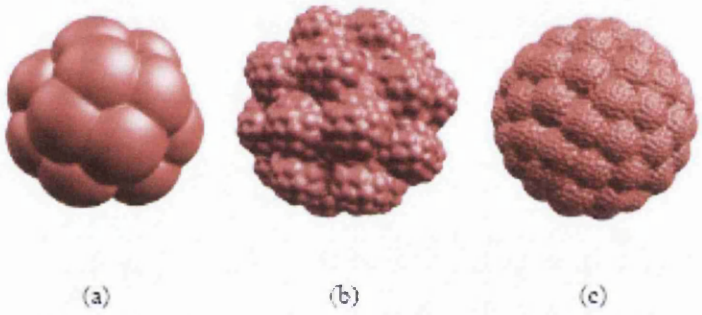


Figure 9. This is an image from Perlin et al's system for adding detail to subdivision surfaces [PERL2]. The berry model is shown.

The berry model, as shown in Figure 9 functions in a different manner; equally placed seed points on the original mesh are used to construct hemispheres on the surface of the mesh which, on the next refinement level, are used to create further hemispheres. The result is a deformation that bulges out from the original shape definition, producing berry like protuberances on the surface of the original shape.

The tentacle model is described as a *surface-morphogenic shape model.* Tentacles are grown from initial seed points, as in the berry model but in this model the direction and length of the displacements is altered by some value at each level, giving finer control over the final look of the object. The tentacle model can be used to model hair, the spikes of a sea-mine or other tenticular extrusions. The model has two parameters; a reference length for the

tentacles and a rotation angle, used to rotate the extrusion. Perlin et al.. also mention that this simple extrusion model could be extended by the use of L-systems to create more interesting extrusions on a surface; i.e, growing plants, or other systems that can be described using L-systems, on the surface of an arbitrary mesh. Finally, the mushroom cloud model creates mushroom like protuberances on the surface of the shape where the mushroom like features grow from a seed point, as in the tentacle example, but are influenced by the 3D coordinates in the region surrounding the seed point.

Perlin et al. also describe the use of alpha mask selections on a mesh that can be used to blend different model for creating surface detail and this raises and interesting number of possibilities; The function used to create the alpha mask might be a user defined selection or based on some surface property, such as gradient. This could allow for surface detail to be added to landscape meshes, based on the type of terrain (rocky formations for steep angles and perhaps plant growth for shallow angles), or different types of procedural detail for different parts of creatures skin. These techniques, whilst fairly straightforward, offer intuitive and simple tools for an artist to add procedural complexity to otherwise simple meshes.

2.2.3 **Procedural clouds and smoke**

Clouds are a complex, beautiful and transitive phenomena that have entranced people for countless generations. The diversity of cloud types seems to defy simple classification and their sheer scale and complex structure makes creating convincing images difficult; to compound this, one only has to gaze out of the window to see that a particular rendering of a cloudy sky falls short of capturing their effervescent grandeur.

The daunting prospect of trying to simulate (or at least represent) nature at its most dynamic aside, a great deal of literature in computer graphics has been devoted to the creation and rendering of atmosphere and clouds ([NISH1] [NISH2] [NISH3] [SCHP] [DOBA] [MIYA]). High-level cirrus clouds can, to some extent, be modelled using a modified Perlin noise texture although to create the icy wisps displayed by this type of cloud, we must apply a distortion to the texture. As we shall see, although standard Perlin noise is not suitable for generating convincing clouds by itself, if combined with other means of representing clouds, it can add some naturalistic turbulence which helps to create convincing clouds.

Many computer graphics papers on clouds generation concentrate on cumulous clouds

[KIKU] [DOBA] [DOBA2] [DOBA3] their relatively solid appearance lending itself to refinements of courser definitions that can be more easily manipulated. In reality, clouds of different meteorological classes blend together to produce dauntingly complex vistas.

Clouds predominately comprise of H_2O in various forms, from vapor to ice crystals. The nature of clouds means that in order to render them effectively, we need to employ light models that handle the scattering of light within the cloud. It is this scattering of light that gives clouds their proverbial 'silver lining'. Cloud rendering models must also be able to handle self-shadowing in order to appear convincing; the difficulties of applying self shadowing are compounded by the extensive scattering of light that occurs within a cloud. Finally, due to the vast scale of clouds in the sky, we need to handle atmospheric scattering and occlusion of the sun by the horizon. At sunrise or sunset, clouds high in the sky will still be lit directly by the sun, even after it has set over the horizon as visible from the ground. Also, dust particles in the atmosphere lead to scattering of light (other than Rayleigh scattering which is caused by gas molecules in the air, which explains the blue hue of our atmosphere), altering its wavelength which leads to the various colours that can be observed at these times of day.

Many methods have been developed for modelling clouds. We can broadly define these into two groups in a similar manner to the earlier distinction between *top-down* models for generating clouds procedurally or heuristically [DSE1] [DSE2] [DSE3] [GARD] [KIKU] [NEYR] with appearance and control being the driving factors and *bottom-up* models, where a more rigorous simulation of the physical behaviour of natural phenomena is required [FOST] [KAJI] [STA1] [STA2] [STA3]. Some techniques for the procedural generation of clouds include Fourier synthesis, fractal generation, spectral synthesis, interpolating vapor density values across a grid and applying a noise function to a set of meta-balls [DOBA2]. Simulation methods include modelling clouds via fluid dynamics, qualitative simulation and dynamic simulation of particles.

Dobashi et al. have developed a system for the generation and near real-time display of clouds [DOBA]. In their system, they utilize meta-balls and cellular automation to create convincing animated images of cumulous clouds. Using a voxel-like grid of values consisting of values representing vapor, clouds and the phase transition from vapor to clouds, animated using cellular automation techniques developed by Nagel [NAGE], they create dynamic animated clouds using meta-balls defined at each cell point. The general shape of the cloud

is defined by iso-surface created by blending the meta-balls together using a power function. Further, smaller meta-balls are created on the surface of the cloud using a fractal method to create a more lifelike fringe.

Dobashi et al. have also used an image based approach, using satellite data as the basis for the generation of 3D volumetric clouds, again using meta-balls [DOBA2]. In their thesis they discuss means of analyzing the satellite data and find the best fit for a series of meta-balls over the visible surface of the cloud. They do this by analyzing the density of pixels of the satellite image and place an initial meta-ball at this point, with it's centre at a pre-computed base plane. They then calculate the error metric between the original image and the meta-ball and adjust accordingly. Their system attempts to use as few meta-balls as possible in order to reduce computation time.

Figure 10. Image from Schpok et al's cloud generation system. [SCHP]

Schpok et al. [SCHP] have developed a real-time cloud rendering system with impressive visual results (see Figure 10). Clouds are rendered as volume slices through an implicit volume which is used to create the general cloud shape. Layers within the cloud are then textured with a four octave 3D noise function to add turbulence and detail. Their system falls into the class of application geared towards visual appearance and intuitive creation, rather than accurately modelling natural phenomena. Although their system does not handle atmospheric scattering, it does include a self shadowing term, although this is performed at the vertex level of the original implicit shape and does not include shadows created by the

more detailed noise function. They do mention that this is one area they wish to address in future versions using modern graphics hardware to compute per pixel shadows in real-time.

Miyazaki et al. [MIYA] take the opposite approach and present a paper based on atmospheric fluid dynamics to generate cloud animations. They use an extended form of cellular automatia called CML (Coupled Map Lattice). The advantage of this meteorologically based model is that they able to create and animate a wide variety of cloud shapes including the familiar cumulus as well as the thunder bearing cumulonimbus, mid-level altocumulus and altostratus and high, icy cirrus cirrocumulus and cirrocumulus clouds as well as the roll-like formations of low level stratus and stratocumulus clouds. To render the clouds, they use Dobashi's meta-ball technique but other methods for generating the images could also be used; the main advantage and focus of their work is in the method they use to actually generate the cloud positions and densities.

In CML lattices, each cell contains a number of state variables and how these variables change depends on the state of surrounding cells. The difference between CML and cellular automaton is that CML uses real scalar values, whereas the later uses discrete binary values. The values in the cells, which control the size of the meta-balls within the cells, are updated over time by computing a series of physically based functions for computing various atmospheric variables including viscosity, pressure, advection, diffusion of water vapor, buoyancy, thermal diffusion and phase transition between ice, water and vapor. Needless to say, a detailed discussion of such atmospheric phenomena is beyond the scope of this thesis but the results of their system speak for themselves. Coupled with accurate rendering, their system provides one of the more complete simulations of clouds in computer graphics to date. Also, CML is not computationally expensive so it is possible to implement their system in real-time which would be a boon to many simulation environments such as flight simulators.

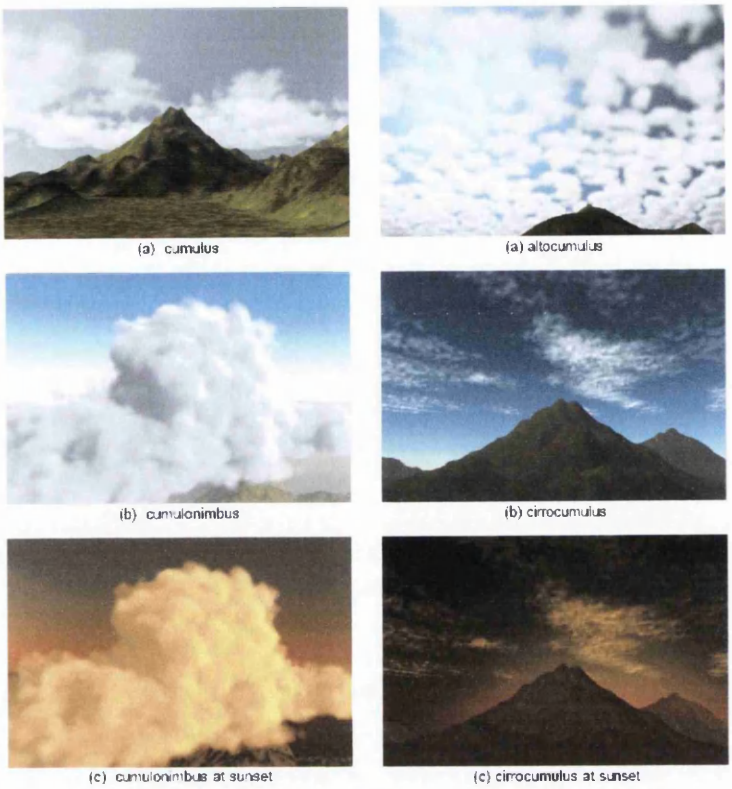


Figure 11. Image showing various types of cloud as created via Miyazaki et al's cloud generation system [MIYA]. This system uses atmospheric fluid dynamics to generate cloud animations.

2.2.4 Procedural Landscapes

Landscapes generated in computer graphics typically fall into two means of representation: elevation maps generated from 2D data and TIN (triangulated irregular networks).

Whilst this approach does not allow features such as overhangs, restricting landscapes to essentially 2D data has many advantages for the efficient storage and rendering of this class of object. Also, on the gross topological scale, landscape surface features rarely exhibit significant overhangs.

2.2.4.1 Generating initial landscapes

Perlin noise [PERL1] and FBM (fractal Brownian motion) can be utilized to create reasonably convincing landscape height maps but in order to create the sort of natural erosion features one would expect from a landscape, some sort of post processing would be required. Fractal or noise based terrains lack features such as rivers, alluvial deposits and deltas and other such features created by erosion. The initial 'mountain building' phase of terrain generation is separate from the subsequent erosion of this terrain and performing an approximation of erosion by various forces as a post-processing stage is a logical step

towards creating convincing looking terrain. The use of fractal Brownian motion to simulate terrains stems from an observation by Benoit Mandelbrot [MAND] that the trace of fractional Brownian motion over time resembles a jagged mountain peek. It is important to note that although there is an observed similarity between fractal terrains and those found in the real world, no formal link between fractal plots and real-world formation of terrain has yet been found.

2.2.4.2 **Applying erosion to height-field data to create more convincing terrains**

Whilst noise based algorithms can create a rough approximation of jagged, eroded terrain, erosion by global systems such as drainage networks caused by rivers and glaciers are more difficult problems. Musgrave [MUSG] lists the main forms of erosion as *fluvial erosion*, *thermal weathering* and *diffusion erosion*.

Fluvial erosion is concerned with erosion by fluids. This is performed over a series of time steps using a formula known as 'erosion law' [ANDR][SCHM] in which water particles are deposited on the height-map and modelling the descent of the water to the lower areas of the height-map terrain. The erosive power of water is a function of its volume, the amount of sediment contained within the water and the local gradient along the path it travels. As the water flows down the terrain, a proportion of sediment is removed and suspended within the water with a constant value specifying how much sediment can be suspended in the water. Another constant determines how long sediment will remain in the water before being deposited further down the flow. The softness of the material at a given location determines how much of it will be converted to water borne sediment. The speed of the water is determined by a gravitational constant, the volume of water in a given channel and the slope of the terrain so fast moving sections of water are less likely to receive sediment deposits whereas slower, flat areas receive more sediment.

Musgrave defines thermal weathering as a generic term for any process that loosens substrate and causes it to fall down a slope to collect in lower, flatter areas. Thermal weathering is simpler to model that fluvial erosion and involves checking the gradient of a given point and, if the gradient is beyond a certain threshold, then a certain amount of sediment from that point is deposited on the neighboring, lower, point.

Finally, diffusion erosion is a process of continual smoothing of the terrain by a number of factors, such as rain splashes, animal footfalls and other such natural elements. This amounts to a low-pass filter of the height map and can be performed in a single pass.

All of the methods for approximating erosion mentioned above benefit from additional information at each element of the terrain height field. Encoding the type of substrate at a given point (or encoding a string specifying different layers of substrate at a given point) would greatly enhance the erosion algorithms as different kinds of rock (e.g, sandstone, limestone, granite) erode at different rates and in different ways.

2.2.4.3 **Optimal methods for rendering terrains**

The ROAM algorithm [DUCH] presents a means to efficiently display dense height-map data. The ROAM algorithm uses a system of binary trees of equilateral triangles with coarser parent triangle representations having two child triangles with a new vertex inserted on one side of the triangle where the transition between coarse and fine representations occur, as can be seen in Figure 12.

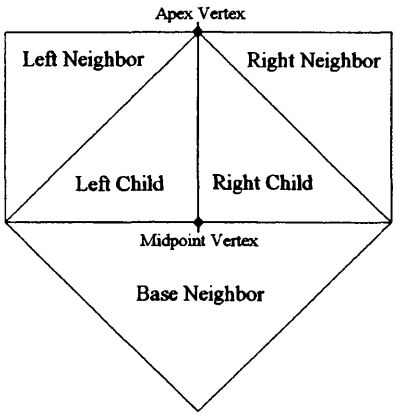


Figure 12. This diagram shows the binary tree triangle structure employed by the ROAM algorithm [DUCH].

To determine what level of detail to use, the ROAM algorithm, and variants, utilize a mixture of distance 'LODing' and analysis of gradient, where triangles further from the viewer and those who are flatter in relation to each other are rendered using courser representations. Also, due to the nature of the binary tree triangle representation, large areas outside of the view frustum can be culled from view until only those triangles at a given resolution that are actually in the view frustum are considered for rendering. The ROAM algorithm also makes use of frame-to-frame coherence where it is assumed that the user will only move to the display by a given rotational and translational increment, further reducing the number of new triangles to consider in a given frame.

2.3 Methods for specifying and distorting 3D shapes

The methods for creating procedural 3D shapes and effects have so far concentrated on systems which are largely automatic and geared towards a specific class of object. Whilst most of these methods require some form of user input, the focus has largely been to automate the generation of complex and dynamic objects and scenes such as clouds, landscapes, plants and cities. Moreover, the user input for these methods is largely unintuitive and disconnected from the familiar methods of altering the shape of solid objects, such as molding clay. In an interactive 3D modelling system it is important to give the artist finer control over the eventual appearance of a given shape, rather than let an algorithm dictate the form of an object entirely. In this section, we shall look at a variety of methods used to deform and manipulate 3D objects in a more intuitive manner. This is closely related to the topic of 3D user interfaces but in this section we shall concentrate on the actual techniques used to distort 3D forms and will cover how the user interacts with these methods in the next section.

2.3.1 Free form deformation lattice

Introduced by Seidenberg and Parry [SEDE] the free form deformation lattice provides a powerful and intuitive control for distorting a 3D shape. Unlike other controls mentioned thus far, the FFD lattice can be used for non-uniform deformations of an object. The lattice is used to deform an object by manipulating the control points on the lattice. At its rest state, the object is unaffected by the lattice but as the control points of the lattice are manipulated, the shape it is affecting also distorts. The resolution and shape of the lattice are flexible and the number of control points can be altered to suit particular modelling needs.

The FFD is a very flexible control and can be used to deform almost any kind of geometry. A given point X is deformed as a control point on the lattice is moved. The FFD is defined in terms of a tensor product trivariate Bernstein polynomial and the lattice forms a parallelepiped (or a prism whose faces are all parallelograms). To transform a point in the 3D object that is to be distorted, we must first transform this point into the local coordinate system of the FFD parallelepiped. Each Cartesian point interior to the parallelepiped can be defined in the space of that object such that

$$X = X_0 + sS + tT + uU$$

where X is the point in Cartesian space, s, t, u define the point within the space of the

parallelepiped, X_0 defines the base of the parallelepiped and *S, T, U* define the trivariate space of the parallelepiped.

Using linear algebra, we can find the *s, t, u* coordinates of a given Cartesian point using the following equation:

$$s = \frac{T \times U(X - X_0)}{T \times U \cdot S} t = \frac{S \times U(X - X_0)}{S \times U \cdot T} u = \frac{S \times T(X - X_0)}{S \times T \cdot U}$$

The *S, T, U* and X_0 vectors define initial shape of the parallelepiped and it is along these vectors that we add control points (P_{ijk}) upon the lattice along the planes defined by the three direction vectors. These form t+1 planes in the *S* direction, n+1 planes in the U direction and m+1 planes in the T direction. A given control point on the lattice has the following Cartesian coordinates $P_{ijk} = X_0 + \frac{i}{l} S + \frac{j}{m} T + \frac{k}{m} U$

To find a given deformed point, we employ a trivariate tensor product Bernstein polynomial. The deformed position X_{ffd} for any given point X is calculated by calculating the *s, t, u* coordinates for that point and then plugging them into the following equation:

$$X_{ffd} = \sum_{i=0}^{l} \binom{l}{i} (1-s)^{l-i} s^i \left[\sum_{j=0}^{m} \binom{m}{j} (1-t)^{m-j} t^j \left[\sum_{k=0}^{n} \binom{n}{k} (1-u)^{n-k} u^k P_{ijk} \right] \right]$$

where X_{ffd} contains the transformed Cartesian point and P_{ijk} is a control point on the lattice. Since this formulae works on arbitrary points within the control lattice, this method of deformation and 3D user interface control can be used to distort any mesh, spline or volumetric data that can be fed into the above formulae.

2.4 Generative Modelling

Snyder has implemented a system utilizing what he terms *generative modelling* [SNYD]. Generative modelling has many elements in common with the system proposed in this thesis but differs in implementation and interface. The defining aspects of generative modelling that are also key aspects of the proposed system are:

- Shapes are defined by base formulae and the subsequent operations upon them.
- Operations made on a base shape can be modified in a non-linear fashion at any time.

Snyder's approach can be seen as an extension of a lofting and extrusion approach, based on low-dimensional entities such as curves and poly-lines that can be transformed into higher dimensional shapes. The real strength of generative modelling as described by Snyder is the way in which the object description can be analyzed. Since objects are stored as parametric representations and parametric operators upon those representations, tests against the object are not made upon an approximation of the object (as a polygonal mesh or NURB patch) but upon the underlying representation.

Snyder's GENMOD software for generative modelling makes use of interval analysis when converting the model to a form amenable to real-time rendering or for ray intersection tests. Whilst he admits that an exhaustive method such as interval analysis has certain disadvantages, such as computational cost, even when combined with non-interval techniques, he states that the advantages of interval analysis (control of error, computation of global maxima and minima) outweigh these disadvantages ([SNYD] p. 123). Whilst the system described in this paper does not currently make use of interval analysis when querying or generating a shape, the methods employed to generative shapes are amenable to such analytical tools.

In his book, Snyder mentions both adaptive and uniform sampling as a means to generate a mesh from a generative description. Using Snyder's generative modelling, a shape is represented by a parametric function:

$$S : \mathrm{R}^n \rightarrow \mathrm{R}^m$$

which is parameterized by the n variables. The shape is generated by the image of S over a rectilinear domain.

$$(x_1, \ldots, x_n) \in [a_1, b_1] \times \ldots \times [a_n, b_n].$$

Several generations of GENMOD, the generative modelling system are described in Snyder's book. The first iteration utilized non-recursive generators and transforms. Non-recursive, as regards generative modelling, refers to the fact that the output of one transform cannot be passed into another. Therefore, shapes are created by specifying input curves to a transform, such as a sweep and the final shape could not be modified further using generative methods. The input curves for these modifiers were constrained to families of functions including constants, lines, arcs and sinusoids. Piecewise cubic curves, created via a

curve editing program and transferred via a file interface, could also be used as input to this first generation generative modelling interface.

The second system featured limited recursion for its transforms. This system allowed an input curve to itself be transformed by other input curves, yielding a more complex variety of output shapes. This system relied on a modelling entity that Snyder refers to as a *u-curve,* which can be represented as a parametric surface:

$$S(u,v):\mathrm{R}^2 \rightarrow \mathrm{R}^3$$

Functions that can be applied to and using these curves include:

- Translation of the u curve.
- Interpolation (lofting) between two U curves to produce a surface.
- Scaling of any coordinate on a u-curve.
- Rotation of the u-curve about any axis.
- Moving a u-curve perpendicular to a trajectory specified by a planar curve or space curve.
- Simultaneous translation and scaling of a u-curve, specified by a planar profile curve.
- Simultaneous translation and scaling of a single coordinate of a u-curve, specified by two 'rail' curves.
- Warping a u-curve by addition of a quadratic function of any of it's coordinates to any other coordinate.

The inputs to generators in this system were one or more u-curves whilst other inputs, for instance the scale or rotation factor of a transformation, were specified by primitive curves.

The second system could encompass all of the functionality present in the first, and an infinite variety of new functions besides. However, the system still lacked generality as

modified primitive curves, in the form of u-curves, could not be used as inputs to many operators.

The third system allowed for fully recursive transformations and generators. After completing the second system, it became apparent that an improved set of modelling operations including vector and matrix operations, arithmetic operations, differentiation and integration, would greatly aid the usability of the system. Thus, these basic modelling tools formed the low-level commands of a simple command line language to specify surface geometry. For example, the following input specifies a cylinder:

```
surface output begin \
    cos mult u twopi \
    sin mult u twopi \
    v
end
```

The surface is defined as the Cartesian product of all the commands between the begin and end. The lines 'cos mult u twopi' and 'sin mult u twopi' define the input of the cos and sin functions respectively with the multiplication of the parametric input 'u' and the constant 2Π. The parametric parameter 'v' is input as is, forming the 'z' component of a surface point.

The authors felt that this third system was the right model to use but that the input specification was clumsy and again, not general enough. The forth and final system, named GENMOD, specification uses a 'C' like syntax and is both easier to use and more general than the third system.

GENMOD is an interpreted system, although certain modules are precompiled to aid speed of execution, and the user can see rapid updates of surface shapes as their specification of those shapes is updated.

Whilst GENMOD is primarily a command driven language, it does have some elements of visual interaction in the form of the curve editors. These form the basis of input parameters to a GENMOD script and as the user edits input curves, the effect on the output model can be rapidly visualized.

Generative modelling is an excellent means to represent an object; the object's high level description is terse yet highly analyzable, level of detail (i.e, output mesh resolution) is simple

to control and this method can be used to represent an infinite variety of shapes; including, if so desired, polyhedral shapes by providing poly-lines as input u-curves as opposed to higher order curves. What is more, the stages used to create an object are in turn parametric tools which can be used to intuitively reshape the object according to need. The main drawback regarding Snyder's generative approach is that it is far more difficult to implement than a simple polygonal modelling language or a system based on any other low-level representation. Also, there is a lack of available graphical modelling tools that use generative modelling as their central paradigm, most graphic artists being unwilling to script an object, rather than design it in an intuitive, real-time interactive editing environment (Aside from the curve editor, generative modelling languages so far rely on scripts to define their shape). Given the ease with which generative modelling can describe existing paradigms as well being rigorously analyzable, the effort needed to implement a generative modelling system is justifiable.

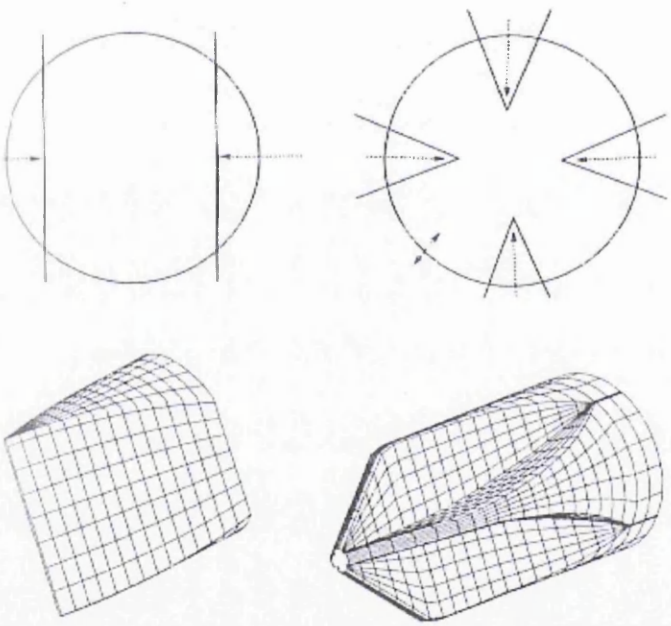


Figure 13. GENMOD example showing flat and Philips style screwdriver heads [SNYD].

Synder's approach also makes use of Computational Planar Geometry or 'CPG'. Similar to CSG (Computational Solid Geometry), CPG is a means of specifying a shape through Boolean operations, only on a planar surface, rather than in 3D space.

One very interesting paper by Ramamoorthi and Arvo [RAAR] describes a technique

for creating generative models from range data. Their technique selects an initial shape and attempts to deform it. This technique not only compresses arbitrary mesh data into a generative description, it also provides useful modifiers that can be altered at a later stage. Since a great deal of data used in 3D graphics is acquired, such techniques for analyzing existing object representations and converting them into an efficient generative description will hopefully provide the stepping stone between previous representations and more compact and useful generative representations.

Generative modelling techniques described in the literature make use of procedural generation and manipulation operators (bend, twist, stretch etc) but do so as part of scripted modelling languages with no graphical user interface and no real-time feedback. In order to provide an intuitive and efficient means of creating 3D objects, some form of graphical user interface is essential as creating objects by tweaking parameters in a text file and spooling these files to an off-line processor is a time consuming and frustrating process.

2.5 3D mesh data compression techniques

The use of Generative modelling can drastically reduce the space needed to store complex, arbitrarily defined 3D shapes. However, techniques to compress mesh data have progressed in recent years and whilst the sort of scaleable compression possible with generative modelling cannot be compared directly with such techniques, mesh compression methods are extremely useful for compressing geometric data, an application which has use both in the transmission of data over a network or even over the data-bus of a computer.

Triangle strips and fans are one method utilized by real-time graphics libraries to improve performance over the data-bus between the CPU and graphics card. Both OpenGL and Direct3D support the use of triangle strips and fans and if utilized well, these can reduce the amount of data transferred.

2.5.1 EdgeBreaker

Jarek Rossignac [JARK] introduced a method for mesh compression called Edgebreaker in his 1999 paper. Edgebreaker attempts to achieve a bit efficient data storage method for mesh connectivity information. In the worst case, the Edgebreaker algorithm achieves 2t bits for any mesh homeomorphic to a sphere or half-sphere where t is the number triangles in the mesh. A naïve approach to mesh storage, assuming 32bits for vertex references, would generate 96 bits per triangle. The later storage method is common for 3D file format representations.

The Edgebreaker algorithm decomposes a mesh into regions, each of which are *simple meshes* (defined in the paper as a mesh homeomorphic to a sphere or half sphere) which can share vertices but not edges. These regions are enclosed by a loop of edges which do not intersect. One edge on each loop is referred to as the 'gate', an edge whose vertex is shared by another loop. When compressing, Edgebreaker deals with one triangle at a time. The triangle to be considered is that which lies on the gate edge on the exterior of a given loop. The algorithm then classified this triangle into one of five different types, as shown on the following diagram.

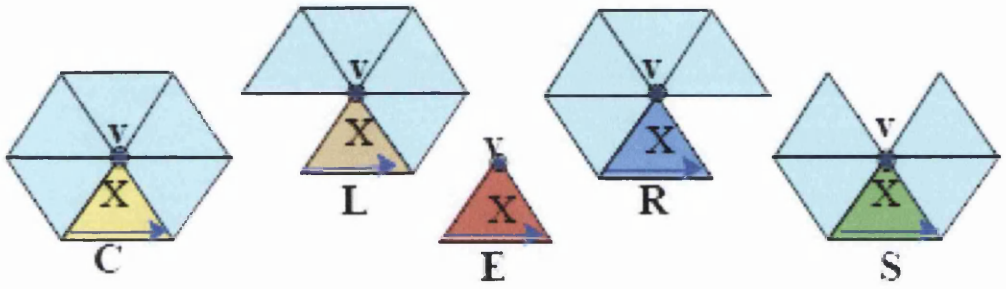


Figure 14. The Five Different cases of the EdgeBreaker algorithm [JARK].

Edgebreaker constructs a history of codes that were uses to compress the mesh and these are used to indicate how to reconstruct the mesh. The algorithm also builds a list of vertex references in the order they are traversed. This history and the vertex reference list is by itself enough to describe the connectivity of a mesh. On each pass of the algorithm, a triangle is removed and the gate reference is updated. The algorithm can also handle objects with holes and Rossignac adds an additional history code, **M**, to identify such cases.

Vertex data can also be compressed in several ways and such compression schemes often make use of variable precision to compress data. Chow's method [CHOW] involves altering the bit length of floating point numbers by analyzing the smallest triangle size or largest surface curvature for a given region and optimizing the bits of precision required for that specific region.

2.5.2 Progressive Meshes

Progressive meshes are another concept of use for interactive 3D and data compression. Progressive meshes, much like their 2D image counterparts, first transmit a low resolution version of the object and then refine this model via vertex insertion as necessary. This approach is interesting as it serves two purposes; the ability to stream 3D data over a

bandwidth limited connection and level of detail encoding, which can be used to render lower resolution meshes at greater distances.

Hoppe [HOP1] in his 1996 paper, *progressive meshes,* describes the generation of such meshes and their use for both LOD encoding and progressive transmission. The basic idea of progressive meshes is an invertible edge collapse operation to simplify a given high-resolution mesh. The key here is the fact that this simplification process can be inverted by the use of an edge split. The low resolution version of the mesh is first transmitted and then, as necessary, extra detail is transmitted in the form of new vertices and the edge reference they are to expand. To avoid 'popping' artifacts, where the visible object jumps from one resolution representation to another, Hoppe employs *geo-morphing* where the newly inserted vertices smoothly interpolate from their collapsed position to their expanded position.

2.5.3 **Summary**

The compression of 3D models for storage and transmission is still an active area of research and there is a clear need for the compression of arbitrary 3D data. However, the use of generative or procedural approaches, combined with a suitable design tool could potentially yield far greater compression of 3D data as well as encoding useful parameters, which can be utilized for animation purposes, as part of the object description. The use of this approach for compressing existing mesh data depends on analysis of the model in question in order to convert it to a procedural description. This is an area that needs more research but the prospect of terse descriptions of models that are actually more useful than standard mesh descriptions is extremely tantalizing. Additionally, this method of storing 3D scene descriptions can be used alongside techniques such as progressive meshes; the progressive mesh generator can be considered another modifier in the chain. Similarly, the EdgeBreaker algorithm is an excellent object connectivity representation format for triangular meshes no matter how the higher level descriptions of such objects are stored and created.

Whilst most 3D modelling tools contain a concept of parameter based modelling and modification, the norm is to rely on data formats that encode objects as triangle meshes or other similarly low-level descriptions. For such an object description format to be accepted by the larger creative community, intuitive tools are required for the generation of such shapes. Therefore, we shall now discuss the topic of 3D interaction and user interfaces.

2.6 User interaction in 3D

The system presented in this thesis strives to provide an intuitive 3D interface to for creating and manipulation 3D shapes and scenes. Whilst in some cases a 3D interface is extraneous and counter-intuitive, for such a system the inclusion of 3D interface elements, rather than relying solely on 2D interface panels, is essential for intuitive use. In his 1995 paper, Kettner [KETT] lists the advantages of 2D interfaces over 3D interfaces as such:

They are still the most popular devices for graphic input. Together with their need for less sensors than 3D input devices, they are more economical.

The forearm can usually rest on the table, while three-dimensional devices fatigue the operator

Normally, 3D interaction is embedded in a 2D window, and switching between the modes can be an unproductive task

As they have one or two degrees of freedom, they are easier to control than a device with a higher number of degrees of freedom.

Physical size is also a limiting factor for the application of a given device

Whilst there have been many attempts to create a standardized set of 3D controls for applications, there is still no *de facto* standard in 3D graphics for the real-time manipulation of objects in a scene. This is not the case for libraries that handle real-time rendering of 3D scenes with PHIGS, GKS, DirectX and OpenGL gaining wide industry and/or academic support. However, these libraries by themselves do not attempt to handle user interface issues and concentrate solely on the rendering aspect of 3D graphics. Straus and Carey [STRAUS] state that the main short-coming of such libraries is that they "are committed to display-orientated application development and provide little help for direct, 3D interaction support". For example, implementing ray-casting callback on OpenGL display lists involves either rendering ID information along with colour information, which can then be read back by the application or else re-rendering the scene with a small 'ray-frustum' and utilizing the `glLoadName()` to assign an ID integer to each object prior to rendering. Both methods can result in sub-optimal picking as OpenGL has no knowledge of any application defined structures which could be used to optimize ray-casting operations. Also, the information gleaned from in-built OpenGL picking operations is limited to the defined ID. As a result of

this, many applications don't utilize the limited interaction support provided by most 3D rendering libraries and instead implement their own, parallel functions.

In response to this short-coming Straus and Carey have created an extendable object orientated library for the manipulation of 3D data known as the Open Inventor Toolkit [OPENI1]. OIT creates what is known as a *retained mode* scenegraph. That is, application data is organized in an object orientated fashion with hierarchical dependencies between objects in a scene. Open Inventor is more than just a scenegraph with support for user feed-back, the toolkit also contains a great deal of 3D modelling tools that could hinder the simplicity of programs which utilize the library.

The methods for manipulating 3D objects described in this section are closely related to the tasks they perform; i.e, the object shape and parameter modifiers. However, it is important to note the same manipulation method can be used to represent several different operations; the same tool used to rotate an object could also be used to place and manipulate a spherical texture coordinate application modifier.

2.6.1 **Models for representing 3D transformations**

Intuitive parameterizations of rotations in 3D are difficult to achieve. Commonly used internal methods to store 3D, such as quaternions, matrices or Euler angles do not translate very well to intuitive numerical controls. Euler angles can result in one axis being canceled out if the object is rotated 90° whereas matrices and quaternions, whilst both are an efficient means to store and apply compound rotations, are totally unintuitive as numerical controls. Axis-angle rotations where a normal vector defines the plane of the rotation and a scalar value depicts the rotation value are an effective and intuitive parametric numerical control and a series of these provide a means to specify rotations parametrically. Whilst rotations are often finally stored in a single compound object (usually a matrix), doing so loses any parametric control. Thus, most 3D interactive widgets for controlling rotation do so by allowing interactive control over the rotation in a given plane before adding this rotation to a matrix when the user releases the mouse.

Chen's [CHEN] virtual sphere is an intuitive means of controlling rotation in a single plane. The interface for Chen's virtual sphere is shown as a thin circle around the object, although later implementations often omitted the circle whilst retaining the behaviour of the virtual sphere. To rotate the sphere, the user simply clicks within the circle and drags the mouse. The rotation is formed by taking the two vectors defined by the first click of the

mouse to the point at which it was released. These vectors are formed by projecting from these intersection points to the centre of the sphere. Taking the cross product of these two vectors produces the normal of the rotation plane. The angle of rotation is taken as arcos of the dot product between these two vectors. Additional rotations can be subsequently performed although it must be noted that these rotations are not transitive. A rotation from A to B to C is not the same as a rotation from A to C.

The turntable metaphor described by Evans et al. [EVANS] describes rotations around the axis perpendicular to the screen with a reference point required to define the centre of rotation. To rotate an object, the user aligns their view at the required plane of rotation and performs circular movements. These movements are converted to an angle by taking creating a vector between the centre of the rotation and the intersection of the pointer ray on clicking and the plane formed by the view vector and the centre of rotation. Another vector is formed by a similar intersection with the current pointer ray and it is the angle between these vectors (both of which lie on the plane formed by the view direction and the centre of rotation) that yields the rotation angle. Variations upon the turntable model include 4 means of rotation; one for each world space or object space axis and one for the axis defined by the view vector. Again, this deviation from the classical turntable model only allows for rotation around one axis at a time but does allow for precise alignment of rotation which can sometimes be difficult to achieve using the basic turntable model and a non-axis aligned viewing system. If the camera axis is found to be the same as an axis aligned rotation control, it can be temporarily disabled to avoid cluttering the display.

The virtual sphere metaphor is perhaps the simplest of all. This means of controlling rotation is not ordinarily accompanied by a 3D widget and simply maps the movement of the mouse in the x and y directions on to rotations in the x and y axis.

2.6.2 **The 3D cursor**

Other methods for manipulating objects in 3D space include 3D cursors, as shown in Figure 15. Variously name *triads* [NLSN], *skitters* [BIER] and *jacks*, these cursors represent single points in space or the orientation, scale and position for various geometric operations. Emmerik [EMRK] used a 3D cursor to directly specify the position, scale and orientation of an object in the scene. This cursor, shaped like a jack which presents a 3D cursor aligned to the object's local space coordinate system. On clicking on the jack, the direction in which the user moves the mouse selects the direction in which the object will translate. Emmerik's

jack also features handles that can be used for non-uniform scaling, and rotation around various axes of an object, thus combining three operations into one metaphor.

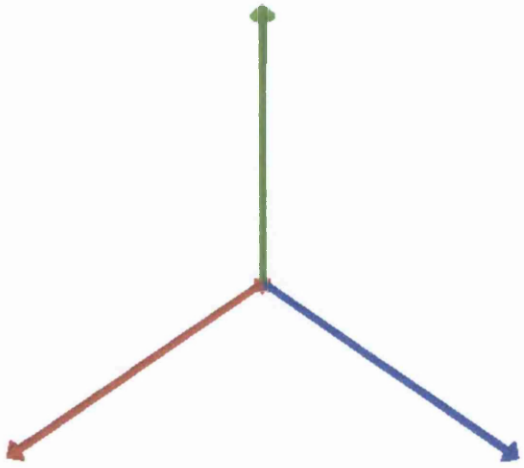


Figure 15. A 3D cursor, used to control the position, scale and orientation of an object.

2.6.3 The Bounding box

The bounding box metaphor consists of a box, often but not necessarily axis aligned around an object or selected region on an object. The bounding box is a versatile transform and has been used variously as a means to translate, scale and rotate an object or form a sub selection within an object. Examples of bounding boxes can be found in Dolner et al. [JDKH] where they exhibit various corner controls to facilitate rotation, scale and translation. The corner controls are shapes in such a way that their function is obvious; Rotation controls located at the centre of each edge of the bounding cube clearly show that they perform rotation around a specific axis whilst the translation and scaling controls are similarly obvious. Marcelo et al. also utilize a bounding box and cite Castier et al [CAST] as their inspiration. Their tool implements scaling by selecting vertices, rotation by selecting and manipulating edges and faces are utilized for translation.

2.6.4 Sketched input

Recently, several systems have been developed that let the user sketch 3D objects in a free-form manner. Igarashi et al. [IGSH] have developed a sketch based 3D input system called Teddy. In this system, the user sketches a rough outline and the system attempts to derive a plausible 3D shape. They achieve this by first triangulating the polygon drawn on the screen. Some polygons are considered illegal and the user is altered to this and the system requests a new stroke. Once the system has an acceptable input polygon, it then attempts to

find the *chordal axis* of that polygon. The chordal axis is defined as the vertices located at the centre of each triangle from the initial triangulation. Vertices along this chordal axis are elevated by an amount directly proportional to their distance from the edge of the polygon. Edges are then elevated, forming a curved region from the initial polygon to the elevated chordal axis and sewn together. The 3D object is thus generated from a 2D polygon with thin areas of the polygon resulting in thin sections in 3D whilst the wide areas of the initial polygon create wide volumes in the final 3D object. The teddy system also supports various cutting and extrusion operations, performed by drawing strokes on the surface of the object. A closed, circular stroke within the shape generates an extrusion base which, followed by a subsequent sketched input from a different angle either creates an extrusion or a 'cave' like imprint on the object. Objects can also be cut by starting a stroke outside of the object and ending it outside the object on the opposite side. Geometry to the left of the cut is removed. Closed cuts that start and end on the same side of the object create cavities within the shape and drawing unclosed regions on the surface of the object add surface details, rather than altering the geometry. While this sketched form of input does not yield very precise objects, it is an extremely intuitive way of creating organic looking 3D objects. It takes only a few strokes to produce a convincing teddy bear object, hence the name 'teddy'.

2.6.5 **3D widget toolkits**

The basic tools mentioned above, along with several more specialized 3D widgets, have been implemented in several 3D interface toolkits. Some of these toolkits attempt to break down 3D interfaces into immutable components [ZELE] upon which more complex manipulators can be created whilst others implement a wide range of manipulator widgets which are none the less flexible in that they can manipulate different types of object. A difficulty in designing a 3D interaction toolkit is in finding an acceptable level of *granularity*. Whereas 3D rendering libraries have now reached a certain level of acceptance and maturity, retained mode scene-graphs, interaction toolkits and means of defining animation and behaviour in said scene-graphs is still an area open to much research. Indeed, it is often difficult to see where the behaviour and animation aspects begin and direct manipulation using widgets ends. Short of developing a full 3D user interface, complete with event model, means of handling collision between various arbitrarily defined scene elements, physical rigid body interaction and various other functions which may be peculiar to a given application, it is difficult to define a concise scene model to enable 3D interaction in the general sense. This may partly be because, unlike a 2D user interface whose closest real-world analogy is a flat surface such as a desk or piece of paper, a 3D environment is analogous to the very world

we inhabit with all the complexity therein. The process of creating and manipulating objects in 3D space and the scene-graphs we use to describe such interaction covers the whole gamut from 3D modelling software where a fine level of control is required, along with manipulators that lack a real-world analogue (we don't rotate real world objects by first covering them with a plastic sphere, we pick them up and rotate them) all the way to virtual environments which require similar underlying mathematical definitions but strive to emulate our real world experience, without otherworldly manipulators to distort our suspension of disbelief. That said, many of the 3D widgets mentioned above, whilst lacking real world counterparts that perform the same function, have become paradigms in their own right and have been implemented in several modelling programs and 3D widget toolkits.

Marcelo et al. describe a system [GMMTK] (called 'MTK' Manipulation ToolKit) that makes use of bounding boxes, trackballs and jacks for the direct manipulation of display lists. They collectively term these manipulators *'draggers'* and their system supports the manipulation of whole objects on the scene using what they term a *smart display list.* Their system is not intended to be a complete object orientated toolkit such as OIT, instead it includes implementations of these three basic 3D manipulators in a context free environment. However, their system still relies on a scenegraph level selection method which is limited to binary selections of whole objects. Sub-object selection, essential for a modelling tool is not implemented in their system.

In 1991, Dollner and Hinrichs [JDKH] describe a system for defining interactive 3D widget. Largely concentrating on whole object selections and manipulation, their system implements a series of manipulators for controlling the orientation, position and scale of objects (the transformer box widget), a widget set for controlling the users position and orientation (the CameraCockpit widget), a pedestal widget which is used to control various display parameters of 3D mesh objects and a series of 3D widgets with direct 2D interface analogues such as buttons and sliders. Their system also introduces the concept of widgets as objects that control other objects in the scene via the use of data connections in the scenegraph. Controller widgets can therefore be re-used and are independent of the objects they manipulate. The aim of their system is to provide a scenegraph capable of abstracting complex elements such as time and event dependent data as well as geometric data into high-level linkable constructs, forming a sort of visual programming language.

Dollner and Hinrichs make a distinction between graph nodes and visible graph nodes

(*graphics objects*). Graphics nodes do have internal methods for rendering themselves. Instead, geometric data stored in a graphical cache called an '*MThing*' is passed to a Virtual Rendering System (VRS) which is an abstraction of an existing rendering library (The paper lists OpenGL and Radiance as two examples of VRS implementations). Ray hit events are also passed to the VMS for processing, largely freeing their system from intensive geometric processing. Non visible nodes perform tasks such as time management and event processing. These nodes define how the visible objects react to either time dependent operations such as stretching an object over a given time frame or how an object reacts when it receives a ray hit.

The Zeleznik et al. [ZELE] have developed a system of 3D widgets where certain base manipulator classes which they call 'widget primitives' that can be combined to produce new manipulator classes. Their base classes consist of a point, represented by a sphere, a ray, represented by and arrow and a more complex base class for representing planes which, although it encapsulates the other two base classes, was defined as a primitive as it is frequently used. Distances in their system can be described by two points in space while two rays, joined at a base point can be used to describe an angle on a plane. Each primitive represents a 0D, 1D, 2D or 3D coordinate system and the coordinate system of these can be constrained by the coordinate systems of other primitives. The primitives change colour as their degrees of freedom are constrained with green indicating that a point is unconstrained, yellow implying a partial constraint (e.g, only orientation or position are constrained) and red indicating that the primitive is fully constrained. By constraining a point to a ray, the point is only able to travel along the length of the ray and is therefore considered partially constrained. Similarly, a point could be constrained to a plane, limiting its movement to that plane. Currently, their system does not have a means of constraining points to a given span of a ray or area of a plane; as such, some compound widgets such as sliders cannot be described in their system as of yet.

Grimm et al. describe a system of more direct use in a modelling program with a set of manipulators used to describe various deformations and sweeps of 3D shapes. Their system utilizes visual control parameters that mirror the deformation taking place. The sweep tool describes in their paper involves the use of a controlling spline that mirrors the effect of the sweep while the warp tool is, by their own definition, slightly unintuitive consisting of a point in space, a circle describing the area affected by the warp and a direction vector indicating the direction and strength (from the magnitude of the vector) of the warp.

2.7 An overview of popular 3D modelling software

Commercial 3D modelling programs are usually split into two camps; those that cater for the CAD/CAM market where precision and compatibility with lathing or other physical manufacturing devices is paramount and the entertainment or design sector where ease of use, animation and visual appearance are more important.

Although both areas have much in common, the approach to programming these systems has become quite different. 3D design software has become more of a catch all approach, incorporating animation, many styles of rendering and increasingly, behaviour to 3D scenes.

Today, this market is dominated by a handful of successful packages and a number of up and coming rival programs that often compete well in some, but not all, areas with the more established programs and are usually much cheaper.

The 'big three' 3D modelling programs at the time of writing are 3D Studio Max [DISCREET1], Alias/Wavefront's Maya [MAYA1] and Softimage [SOFTI1]. Each of these programs exhibit major commonalities which mark them as direct competitors but the subtle differences in approach has enabled each to carve out a niche, albeit always overlapping in some way with their rivals.

2.7.1 3D Studio Max

3D studio Max [DISCREET1] from discreet software is most commonly used in the games and real-time entertainment industry and has a strong suite of polygonal modelling, animation and texture application tools. However, early versions of this software (3D studio for DOS) concentrated on efficient rendering of still scenes as the market for 3D games was very immature at that time. However, with the introduction of 3D studio Max saw a shift in emphasis towards real-time modelling that has continued to this day. 3D studio Max still has an off-line rendering engine of respectable quality and the inbuilt material editors are exclusively geared towards off-line rendering, despite the usefulness of the tool for games and real-time simulation development. Discreet have recently released a program that utilizes the core technology of 3D studio Max (and operates with a similar interface) exclusively for the games industry. GMax, as the software is known, handles real-time material in a native manner and can be tailored to fit the needs of individual game titles by the use of plug-in like products called 'game packs'. These augment and also restrict the functionality of the program so that it best suits the rendering environment of a particular

game, allowing 3rd parties to write modifications easily and also to allow game development houses to write their own game packs when developing their software, freeing them from the need to write proprietary editors.

The release of this software cements Discreet's commitment to the gaming sector and also allows them to compete directly in the film and television sectors that Maya and Softimage currently dominate.

2.7.2 AC3D

AC3D is a shareware polygonal 3D modelling program created by Andrew Coleburn during his PhD at Lancaster University. Due to AC3D's support of multiple platforms and it's generally simple and intuitive interface, AC3D has found quite a lot of success as a low polygon 3D modeller. Originally written for the SGI indigo, the use of the TKL interface toolkit and OpenGL, AC3D was quickly ported to a wide variety of platforms including MacOS, Windows and Linux. There is even a version available for the venerable CBM Amiga. AC3D has a similar feature set to other polygonal modellers but does not support procedural modelling techniques and parameters for objects and modifiers cannot be altered once an object has been created.

2.7.3 Autocad

Autocad has long been a standard in the manufacturing industry. One of Autocad's primary uses is in the creation of blue prints and floor plans. While Autocad is capable of creating 3D rendered views, its main market has been for those wishing to create 2D floor plans. The file format for Autocad files, suffixed by a '.DXF' extension, has become an industry standard.

2.7.4 Blender

Blender is an open source modelling project that has quickly gained a great deal of support, both in terms of users and developers. Blender has an impressive feature set, including support for both polygonal and cubic curve modelling. The software also comes with support for high quality scan-line rendering as well as real-time animation, interaction and rendering. There are a plethora of good ideas contained within Blender; Python script support and gesture based interaction are examples of this. However, Blender is hampered by a non-standard and somewhat overwhelming interface.

2.7.5 Cinema 4D

Cinema 4D, from German developer Maxon, has evolved swiftly into a powerful editor,

supporting polygonal and NURB based primitives, as well as subdivision surface types. The latest release (8.1) supports a schematic view and can compete admirably with more expensive systems. The interface is clean and easy to use although once a sub-object selection is made on an object, the original parameters of that shape cannot be changed.

Cinema 4D also sports a high quality off-line rendering engine and animation support is comprehensive.

2.7.6 **Clayworks v2.45**

Clayworks v2.45 [CLAY245] is a basic, DOS based polygonal modeller, written by the author in the summer of 1992. While basic, this freely available program enjoyed some popularity in the early 1990s.

2.7.7 **Lightwave**

Lightwave from NewTek [LITW] software is a 3D modelling program that has evolved over the years from it's roots on the CBM Amiga. Whilst Lightwave is primarily a polygonal tool, it now contains support for higher order primitives. The range of modelling tools available is extensive and despite, or perhaps because of, the fact that the interface still harks back to the original Amiga program, Lightwave enjoys a large and loyal user base and has been used for film, television and game projects including Babylon 5 and Titanic.

2.7.8 **Maya**

Maya [MAYA1], from Alias-Wavefront is a powerful tool that has found many followers in the game and film industries. Maya provides tools for polygonal, NURB, subdivision and volume primitives, all of which can later be manipulated by the large set of modifier tools available. Maya also features powerful shader tools for describing surface properties although, as with most other 3D modelling programs, these are geared towards off-line rendering methods and have less impact on the visual appearance of the object when manipulating shapes in real-time.

Maya also features a schematic view, known as the 'hyper-graph', which is used to show the data relationships between objects in the scene.

2.7.9 **MilkShape**

MilkShape 3D from Chumbalum software is a low polygon tool aimed at the games market. It is primarily used by people who seek to modify content for existing games, such as half-life and quake. To this end, MilkShape supports a wide variety of file formats for

both import and export. However, the range of modelling tools available is quite limited and the system cannot handle large polygonal datasets. As games software is moving increasingly towards higher definition content, tools such as this will have to evolve into more complex applications in order to compete with larger, more advanced 3D modelers.

2.7.10 **Nendo**

Nendo (Japanese for 'Clay') from Nichiman graphics concentrates solely on polygonal modelling. By limiting the scope of the software, Nichiman have created an extremely easy, and even fun, program to use. The interface is unusual but intuitive, eschewing the traditional four panel draughtsman's view for a single 3D window with a easy to access context menu accessible via the right mouse button. Nendo is suited for low-polygon model development and has therefore captured a niche for interactive entertainment where low polygon count objects are required for efficient rendering.

2.7.11 **Softimage**

Similar in market aspirations to Maya, Softimage [SOFTI1] has traditionally focused on rendering and animation for the film and television market. Softimage integrates with 'MentalRay', an off-line rendering engine used to produce cinematic quality visuals and this continues to be their core market. Despite this, Softimage is also used extensively in games and the company has made attempts to integrate modern real-time visualization tools into their system. The current version of Softimage XSI supports Direct X's pixel and vertex shaders (fragment programs in OpenGL) to better utilize up to date graphics hardware. Softimage XSI claims to be the leader in non-linear editing and creation of 3D animation. Non-linearity, with regards to 3D animation, refers to the creation process of objects, animation and surface attributes and the fact that items can be edited after they have been created and that future changes are non-destructive (i.e, you loose some earlier modification when applying further modifications).

2.7.12 **SolidWorks**

SolidWorks is aimed exclusively at the manufacturing sector. The software makes extensive use of Boolean operations, or CSG, to create forms. However, the file format for SolidWorks contains a polygonal output mesh, negating any storage savings that can be gained through using an object-history description technique such as CSG.

2.7.13 **SolidEdge**

SolidEdge is a similar package to SolidWorks and is also geared towards the

manufacturing industry. The toolset used by SolidEdge does not encourage free-form modelling but this can be seen as an advantage in that the output of this program will be manufactured and needs to be constrained to the capabilities of current manufacturing technology. However, there are some similarities between this constrained approach and this method in that a history of operations must be kept in order to manufacture the final item.

2.7.14 **Truespace**

Truespace from Calgari software is a mid-range 3D modelling program with support for a wide variety of tools and modelling paradigms. Truespace sports an 'unique' interface, placing the menu bar at the bottom of the screen and providing stackable button bars for the modelling tools. Whilst some of the user interface designs are questionable, Truespace does make a great effort to provide tools that can be manipulated in 3D, even for tools that other, more expensive programs relegate to 2D control panels only. For this reason, Truespace is an innovative product insofar as the interface has always attempted to push the boundaries in the difficult area of 3D object manipulation.

2.7.15 **Summary**

There are a wide variety of 3D modelling programs available both commercially and as free products. In turn, many of these programs specialize in various areas with programs which cater for the film and television market concentrating on high quality off-line rendering and linear animation, games and interactive software concentrating on efficient modelling in order to keep polygon counts low and frame rates high and finally, the traditional CAD market where accuracy is more important than rendering quality or rendering efficiency. Each modelling program employs one or more *paradigms*. We define a paradigm as the underlying representation for a model. Many modelling programs restrict their definitions to polyhedra, whilst others include some form of curve representation. Typically, these are low level representations and are usually an approximation to a shape, such as a polyhedral or cubic approximation to a sphere. This is fine if the desired shape *is actually* a roughly spherical collection of polygons but in all the above mentioned sub-domains of 3D modelling, greater accuracy and trueness to an actual shape is desirable and functionally more useful. As Jim Kajiya once said *"Good modelling can save bad rendering, but good rendering can't save bad modelling"*.

2.8 The Scenegraph

The scenegraph is a somewhat nebulous term which can mean many different things to

different people. Most often it refers to retained mode APIs and scene description languages and the manner in which graphical elements in a scene relate to each other.

The concept of the scenegraph is central to any 3D visualization or modelling software. Early examples of software using the scenegraph approach include Open Inventor [OPENI1] which found use in interactive applications that required mouse input, such as modelling and training tools. Scenegraph libraries usually focus on specific areas where their particular strengths are best used. OpenGL optimizer from [SGIO], for example, is a scenegraph system designed with the real-rendering or near real-time rendering of large (usually industrial) datasets is required. Real-time applications have a host of scenegraph APIs to choose from. Open GL performer [SGIP] is a scenegraph API, again from SGI, that focuses on rendering speed and has found use in a number of virtual reality systems.

The games industry is awash with specialist scenegraph APIs (more commonly known in the trade as 'engines') which are often licensed to third party developers who can then sidestep the time consuming task of writing a proprietary scenegraph. Examples of such scenegraph APIs include the Quake/Doom series of APIs from ID software [CARMACK1] and the Unreal engine from Epic software [SWEENY1]. ID and Epic have concentrated on creating game engines for their own games (typically first person view shooting games) and then licensing the scenegraph/engine to a third party as a means to secure additional revenue. The games themselves act as powerful marketing tools but in truth, the licensees typically have to rewrite large portions of the API to accommodate specific features required for their games. Other companies, such as AiCube with their scenegraph/API Xeios [XEIOS1], Critereon with RenderWare [CRIT1] and Touchdown entertainment's Jupiter and Copperhead (formerly known as Lithtech) [LITH1] with their own API concentrate on developing the API as a multipurpose tools suited for many game styles. These companies do not concentrate on developing their own in-house games as epic or ID would but concentrate almost exclusively on developing the scenegraph technology for third party developers.

The scenegraph of a game engine and that of a 3D modelling program might seem like wildly different areas but in fact, they have a lot in common. As 3D computer games and other forms of real-time simulation become more sophisticated, the need for modelling to evolve into more than just shape editors increases. A game engine might not need to contain tools for aligning textures and a 3D modelling tool might not need to have code to

deal with projectile impacts. However, it is possible to have a scenegraph that can do both without containing extraneous code.

In the proposed system, each object, shape modifier, material, texture, palette, AI script, camera or light is a node in the scenegraph. It is not important that the node is visible or not when the 3D scene is rendered, merely that it defined in a rigid manner that specifies what kind of data it can input and output and that it contains functions for processing incoming data and making downstream nodes aware of these changes. The 'bare bones' scenegraph need not have knowledge of any nodes at all, just a base node class and the means to insert nodes into the scene. All the nodes necessary for a game engine, 3D modelling tool or other application are loaded as and when they are needed. The scenegraph, be it intended for a game or productivity application, then handles external events such as mouse movement or joystick control, decides what to do with them and interprets them into events packaged for the scenegraph. In the context of a modelling application, this might mean interpreting a mouse event and repackaging it as a ray event cast into the scene which objects within would then interpret. In a game engine, the mouse could be linked directly into a camera object and pressing the mouse button could release a projectile; On collision with another object, a collision event would be sent to the colliding shapes which would then interpret what to do. An entire game could be sculpted inside the 3D modelling program with developer supplied specialist nodes for handling AI and maintaining the state of the game.

2.9 Differences between existing 3D modelling programs and the approach in this thesis

The 'big three' modelling programs all have one feature in common that is lacking in almost all other 3D modelling programs; A schematic view of the scene showing interconnections between scene objects, assets and deformations. This is known as the 'schematic view' in 3D Studio Max and Softimage and the 'hypergraph' in Maya.

Whilst all of these allow the user to view the scene as a schematic view, they all suffer from one major constraint; It is often impossible to alter the initial parameters of an object (say, the subdivisions of a sphere) and have these changes accurately and correctly propagated through all the subsequent modifications to an object. This is because the programs are tied to the underlying object representation, rather than aiming for a more abstract means of representing object data.

To address this problem, the system described in this thesis endeavours to create a procedural modelling system that is as easy, or easier, to use than existing software and also keeps a robust abstract view of the data such that, for any deformation that one can perform on a 3D dataset, that object can be recreated faithfully with a higher resolution source mesh or even use different modelling paradigms (such as NURBS or volume data) and recreate the object faithfully. Viewed in this manner, the schematic view of a 3D scene is no longer just a means of managing the assets of that scene; it becomes a visual programming language that thoroughly describes the scene in an abstract manner.

The key here is that even though polygon meshes or NURB surfaces are used internally, the program does not rely on these paradigms. A sphere that has had a portion of it deformed, extruded or subdivided is just that; A deformed sphere, rather than a collection of polygons that looks like a deformed sphere, unless of course the desired result actually *is* a polygon approximation to a sphere. The most useful representation of the object or scene is not the final result that is sent to the display, it is the description of how the object was created that is important. Interestingly, this approach solves many problems in one fell swoop and adds a lot of new and functional ways in which the user can edit a scene. Adding infinite undo/redo functionality becomes trivial; merely a case of removing the latest modifier or altering its parameters. There is never any need to save vast amounts of polygonal data to a backup cache in order to perform an undo. Similarly, when saving a disk to file, it is not necessary to save vast numbers of polygons, just the steps taken to create the object.

Where most programs fail to achieve this flexibility is in the manner they handle selections of an object. When an artist or designer is creating a 3D object, most of their time will be spent selecting regions of the object and performing local deformations upon these regions. 3D studio max makes an attempt to solve this problem with the use of volume selections. The user can use boxes, spheres, cylinders or any closed shape in the scene to define a volume selection. However, 3D studio does not store frustum selections as volumes and thus exhibits inconsistency in its design. The system described in this thesis defines **all** selections as a parametric volume, thus removing any reliance on the underlying polygonal (or other rigidly defined) structure. The key is to therefore be able to implement all of the usual selection methods found in 3D modelling programs without resorting to explicit references to element indices.

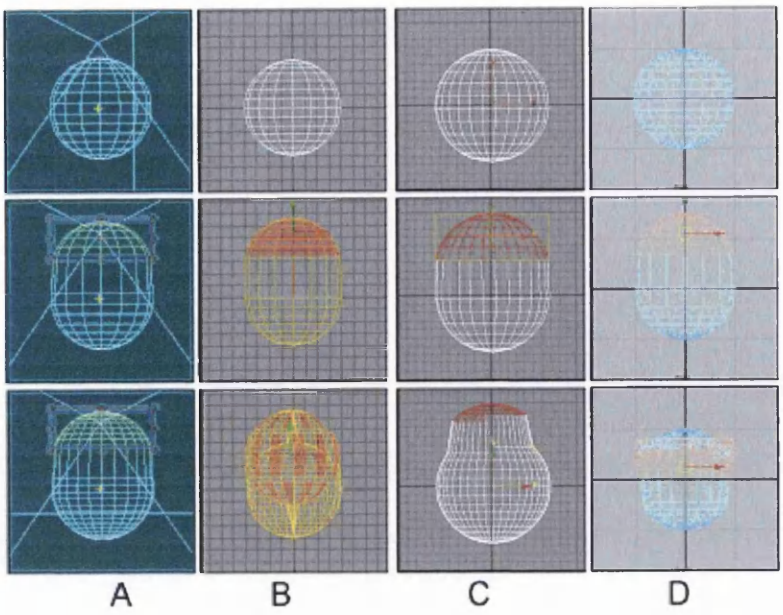


Figure 16. Showing differences between three major software packages and the proposed system. From left to right these images are from the proposed system (A), Softimage [SOFTI1] (B), 3D Studio Max [DISCREET1] (C) and finally, Maya [MAYA1] (D). Each shows how the programs handle selection and translation of a region of a sphere; each performs the operation in a predictable manner until the resolution of the original sphere is altered.

We have looked at a wide range of techniques used to create, describe and manipulate 3D scenes, with emphasis placed on parametric and procedural approaches. A major differentiation between the techniques illustrated here is in the level of interaction the user has. In many procedural systems, objects are created via the creation of off-line scripts which are then compiled into a 3D shape. Some of these systems use parametric representations that are of little use to those in the field of graphic design. L-systems, for example, typically require specialist knowledge to take advantage of their flexibility. Parish and Muller's [PAMU] CityEngine mitigates this difficulty somewhat by providing set grammars for different classes of city but still relying on several input maps for geography, population density and so on that can easily be understood by someone not versed in creating L-System scripts by hand. However, CityEngine is not intended to be a real-time tool and adjusting the generated city to fit a specific need is still a time consuming and non-interactive process.

For some classes of object, it is arguable that an artist would desire limited interaction, rather than micromanaging phenomena. Clouds and trees are examples of this: an artist may indeed wish to create a cloud of a certain shape for dramatic effect in a scene but might not

want to model each individual whisp of cirrus or puff of cumulus, especially if the scene is to be animated. Likewise, the general shape of a tree might be important but if the detail can be procedurally generated, a lot of time and effort can be saved. Similarly, it is likely that nobody would want to design a forested landscape one leaf at a time when we can get a similar result using more general descriptions.

For a design centric, rather than biologically accurate approach Kruszewski's approach to designing trees is ideal. The general shape of the tree is given, as well as some parameters to represent the class of tree. This representation is compact but is flexible enough to give an artist control over individual trees if required. It also fits in with the requirements for the software described in this thesis, albeit for a specialized class of object. That is, the description of the object is terse but still allows for parametric and intuitive control of the final appearance of a shape.

We have also seen that although some commercial 3D modelling programs appear to support a kind of procedural modelling, they fail to perform as expected in many situations due to their reliance on low-level modelling paradigms such as polygon mesh storage or NURBs.

We have also looked in to means to represent and compress 3D mesh data storage. The methods described in this thesis are not intended to replace these methods, more to complement them as the final data structure used for rendering is often a triangular mesh. Rossignac's Edgebreaker would be an ideal replacement for triangle strips and fans if implemented in hardware. Our goal is to keep a useful high-level description of the object at all times and storing and transmitting only the procedural description and only recreating the mesh (at a desired density) as it is needed.

Generative modelling, as described in Snyder's book [SNYD] is a powerful paradigm but is described more as a scripting language for 3D shape generation. Whilst this is acceptable for some visualization purposes, it is again predominately an offline process and much more tedious to use than a WYSIWYG editor. We foresee that it will be possible to implement the generator and modifier nodes described in this thesis in a manner that is as mathematically rigorous as Snyder's generative models, with all the advantages entailed therein. However, our purpose is initially to create a fully interactive system utilizing a similar approach but utilizing a low-level approximation of a shape throughout the creation process, even though the high-level description is implicitly encoded alongside and is used for

transmission and storage of the shape. We also foresee that the methods used for selection and extrusion in Igarashi's 'Teddy' system can provide a very intuitive and powerful means to create new shapes and that this technique is also compatible with Snyder's generative modelling. Although Igarashi concentrated on sketched input, it is also possible to project geometric shapes onto the surface on an object and use these for more precise models, whilst still retaining the analyzability of generative modelling.

We thus arrive at the method for this thesis and will describe how we have created a system that combines the usability of a modern 3D modelling system with the terse descriptions and procedural flexibility of generative modelling.

Chapter 3

OBJECT MODELLING USING GEOMETRY GENERATORS, SELECTORS AND MODIFIERS

3.1 Background

In this chapter, we shall look at the general outline of how the system described in this thesis operates. Firstly, in section 3.2, we shall look at geometry generators. These nodes are used within the system to create initial geometric shapes which can later be modified. The full range of geometry generators is covered in detail but it is possible to add further geometry generator nodes to the system via a plug-in interface.

Section 3.3 covers the subject of selection. Selection is the mechanism by which we mark areas of a geometric shape or highlight control points for later manipulation by modifier nodes.

Section 3.4 covers modifier nodes. Modifier nodes take geometric information as their input and then modify this input in some form and then output this data to either a node which handles rendering of this information or to another modifier.

Section 3.5 covers miscellaneous nodes, such as materials, textures, palettes and logic nodes which do not fit into any of the above categories.

Finally, in Section 3.6 we cover the subject of event based interaction within the framework of the application.

During this chapter, numerous references are made to the flow of data within the system and often, graphs are shown depicting this flow. These graphs are generated by the system itself and show how information (either events, scalar values or more complex data structures) is passed through various nodes to produce an end result. This framework is critical to the manner in which the system operates and a detailed description of its implementation is given in chapter 4.

3.2 Geometry Generators

This section will discuss in detail the various types of standard geometry generators available and how each geometry generator creates mesh data.

The system contains a wide range of base geometry generators. These include (but are not limited to) the sphere, cylinder, toroid, super parabolic quadratic ellipsoid (or super ellipsoid), cone and cube as can be seen in Figure 17. Additional geometry generators can be added to the system at run-time by registering new node types. Each geometry generator has parameters that can be altered at any stage, even after any number of modifiers have been applied to the object.

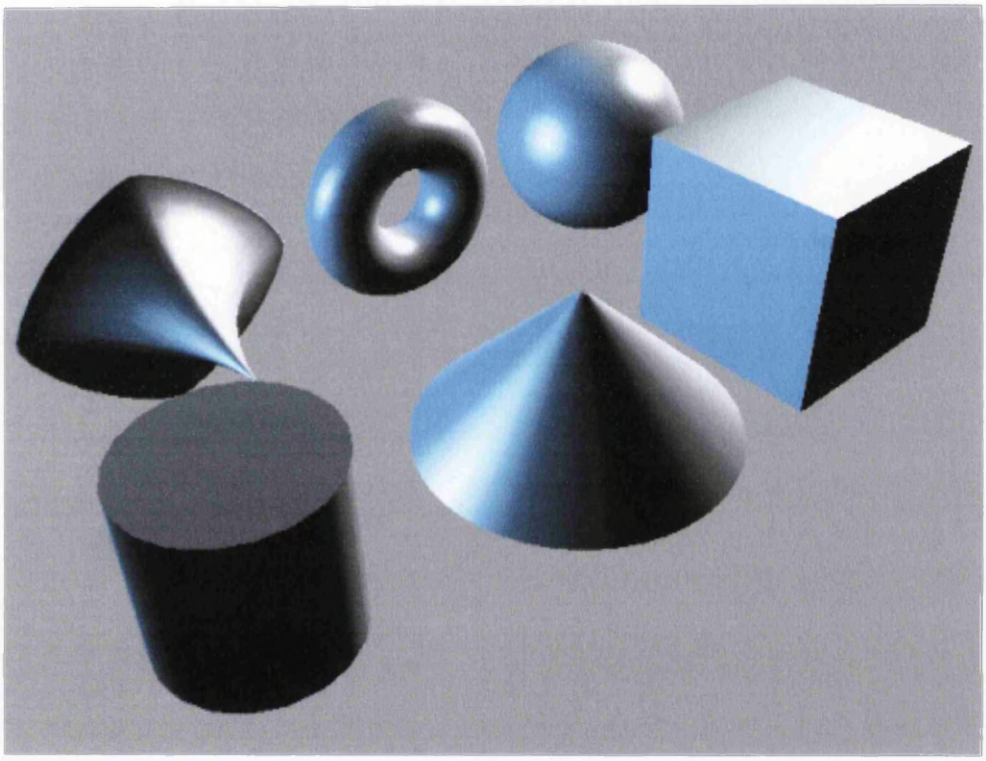


Figure 17. The image above shows a selection of base primitives. From top row first they are the super ellipsoid, torus, sphere, cylinder, cone and cube. The cube, cylinder and cone shows the use of smooth groups to create sharp edges. Smooth groups can be automatically assigned based on the angle between faces.

3.2.1 The Grid

The grid geometry generator creates a simple unit planar grid (of 1.0 unit length on a side) at a given resolution. The grid is constructed from quadrangles, connecting a lattice of vertices. The number of vertices for a grid is defined as

$$N = (x+1)(y+1)$$

where x and y define the number of segments in the x and y axis respectively; the effect of increasing these values can be seen in Figure 19. The orientation and scaling of the grid object are controlled by the world-space transformation node. Texture coordinates are

automatically assigned ranging from 0,0 to 1,1 on the top left and bottom right corners of the grid respectively, as can be seen in Figure 18.

Figure 18. This image shows the default texture mapping properties of a grid at different resolutions. The default grid UV generation interpolates between 0 and 1 across the horizontal and vertical axis. When using lighting interpolated from vertices, the quality of the lighting increases with the resolution of the grid. The specular highlight is not visible in the lowest resolution grid.

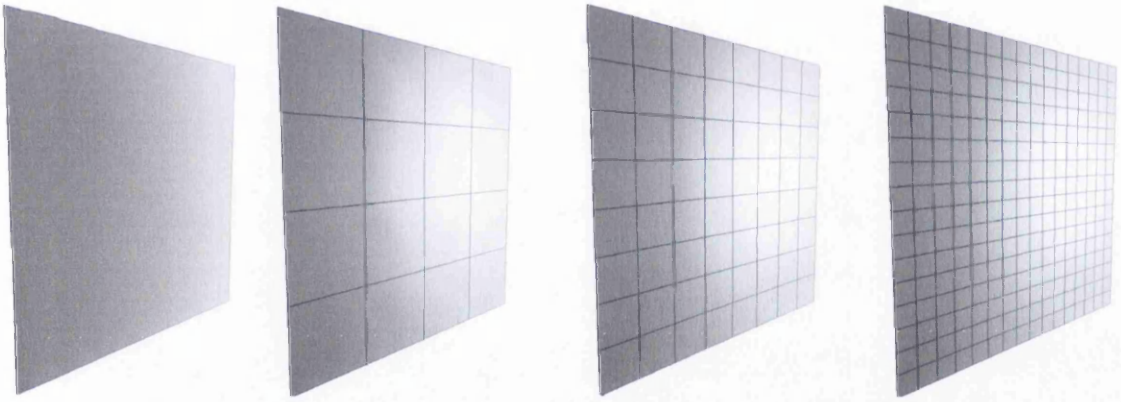


Figure 19. This image shows various grids without texture maps, at the same resolution as the textured versions in Figure 18.

3.2.2 The Cube

The cube geometry generator creates a cube 1 unit of length, breadth and height at the origin. As with most geometry generators, the world-space transformation node controls the scaling, position and orientation of the node. The segmentation of the cube can be controlled parametrically in the x, y and z axis. The number of vertices, edges and polygons used to create a cube can be expressed using the following formulae:

$$NV = 2((x+1)(y+1)) + 2((x+1)(z-1)) + 2((y-1)(z-1))$$

$$NE = 4(2(xy) + 2(xz) + 2(yz))$$

$$NP = 2(xy) + 2(xz) + 2(yz)$$

where x, y and z represent the number of segments in the x, y and z axis respectively. Thus, a cube at the lowest resolution (6 faces, one segment in each axis) yields 8 vertices. The effect of increasing these values can be seen in Figure 20.

Texture coordinates are assigned to each side of the cube. Whereas the vertices are shared along the edges of the cube, the default cube texture coordinates are not. Not sharing the texture coordinates along the edges of a cube eliminates visual artifacts that would other wise be introduced (see Figure 60).

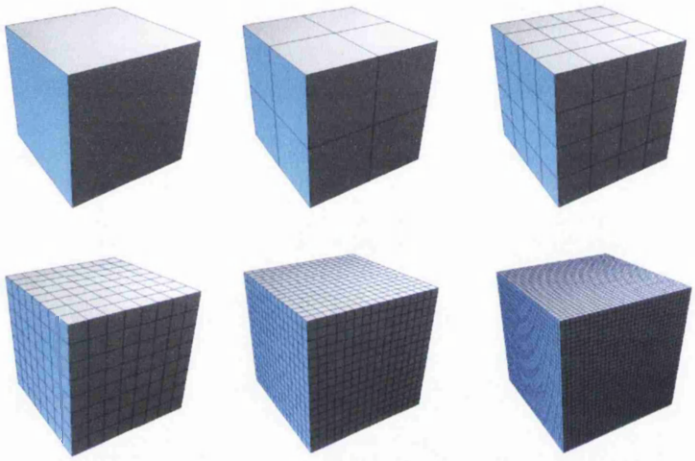


Figure 20. Cubes increasing in resolution on all axis by powers of two.

3.2.3 The Sphere

The sphere geometry generator creates a sphere of radius 1 unit at the origin. Parametric control of the position, scale and orientation is ceded to the world-space transform node that feeds into the node that finally renders the sphere. To avoid redundancy, parametric control of these values is not controlled by the sphere geometry generator. The sphere geometry generator has two parametric controls which are used to define the polygonal resolution in radial and lateral sections, as is shown in Figure 21. Figure 22 shows the effect of increasing this resolution on an object that has been created by modifying a sphere; the increase in resolution of the base shape also has the effect of increasing the resolution of the final shape.

The sphere is constructed by creating two end points and a series of rings down the z axis. Two cones are created by connecting the two end points to adjacent rings and then

connecting the remaining rings together using four sided polygons or quads.

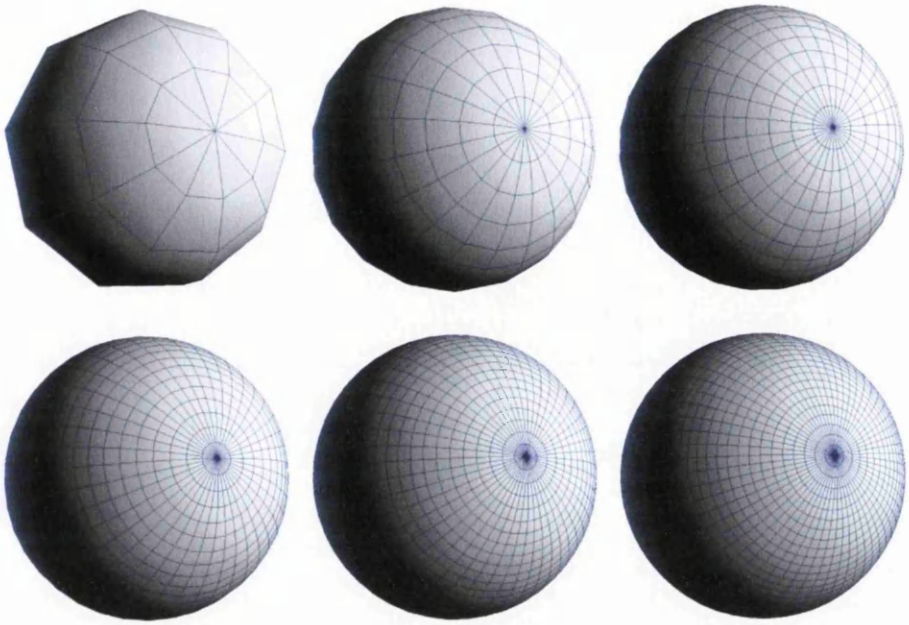


Figure 21. Output of the sphere geometry generator with increasing resolution in radial and lateral segments in steps of 10.

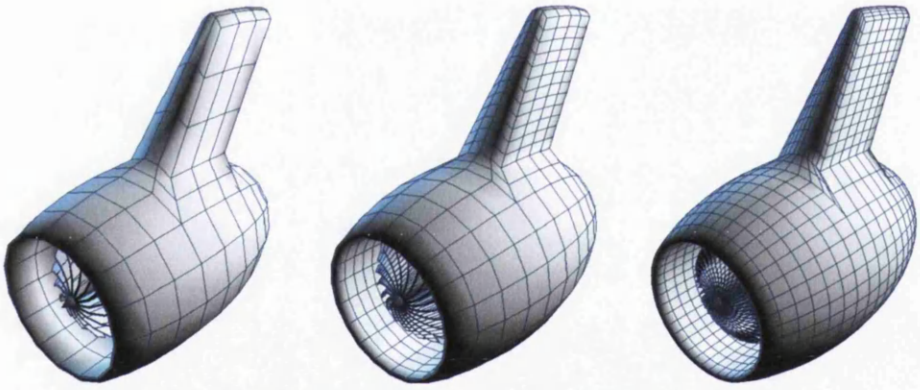


Figure 22. Image showing a jet engine created by modifying a sphere. The resolution of sphere was altered after all modifiers were applied, creating a higher resolution version of the shape.

3.2.4 The Super Quadratic Ellipsoid (or Super Ellipsoid)

Originally introduced by Barr in 1981[BAR1], the super ellipsoid is constructed in a similar manner to the sphere only instead of using circular rings as the lateral sections, super quadratic ellipses are used. Considered on the X/Y plane, the formulae for calculating a super quadratic ellipse is as follows:

$$(|x|^{\frac{2}{e}} + |y|^{\frac{2}{e}}) = 1$$

where e is the exponent of the quadratic ellipse.

Taking this to 3D and replacing the exponent 'e' with 'u' and 'v' for control in two axes, the formulae becomes

$$(|x|^{\frac{2}{u}} + |y|^{\frac{2}{u}})^{\frac{2}{v}} + (|z|^{\frac{2}{v}}) = 1$$

Figure 24 shows the effect of changing the values for u and v from 0 to 2.75. When both n and e are close to 0, the shape appears cuboid but as these values increase, the cuboid becomes rounder before becoming a sphere when both u and v are 1.0. Further increasing the exponents pinches the sides of the shape to form a star like pattern. Adjusting the exponents gives rise to an interesting variety of shapes.

Figure 23 shows the effect of increasing the polygonal resolution in both axes of a super-ellipsoid.

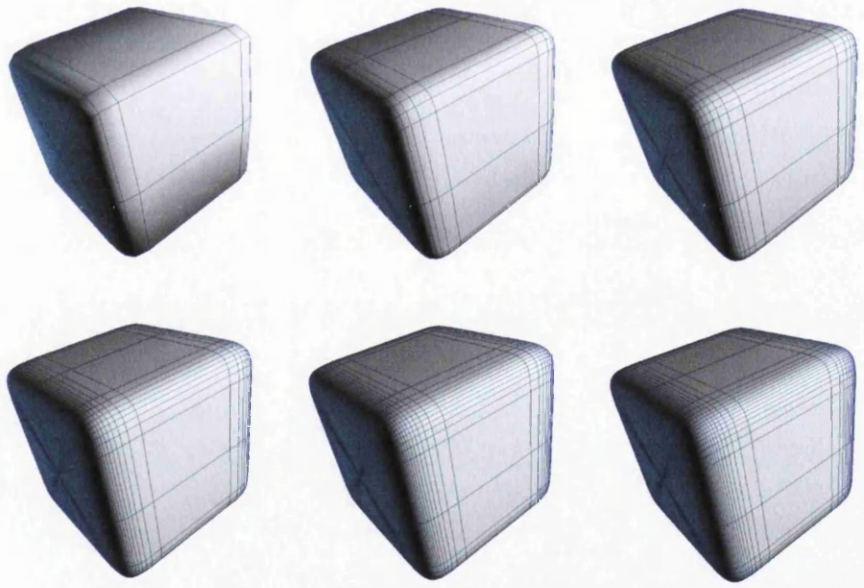


Figure 23. Adjusting the radial and lateral resolution of a super ellipsoid (n and e of 0.2) in steps of 10.

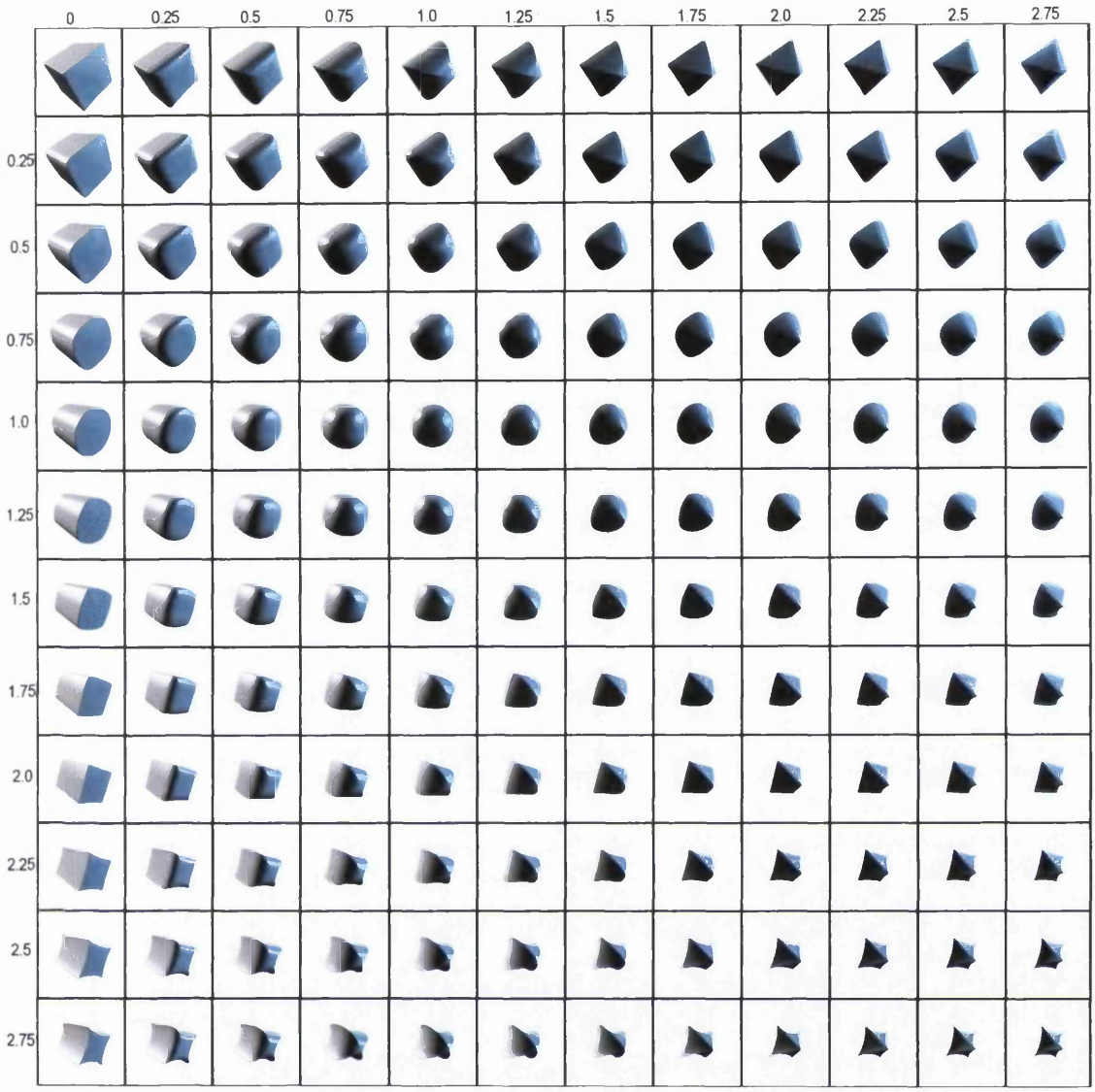


Figure 24. 144 permeations of super ellipsoids with values of n and e ranging from 0 to 2.75.

3.2.5 The Cylinder

The cylinder is another base primitive. Like the sphere and the super ellipsoid, the cylinder allows the parametric alteration of radial and lateral segments along its length as seen in Figure 25. The method of construction differs slightly in that two circles are created at either end which are then stitched to a number of vertex rings down the length of the cylinder. The capping circles and the cylinder body employ separate *smooth groups*, resulting in a visibly sharp edge to the cylinder. The smooth groups instruct the mesh rendering node to create multiple vertex normals per vertex at the edges of the smooth groups, rather than sharing the vertex normal and generating it by the average of the normals of the surrounding polygons, as shown in Figure 26.

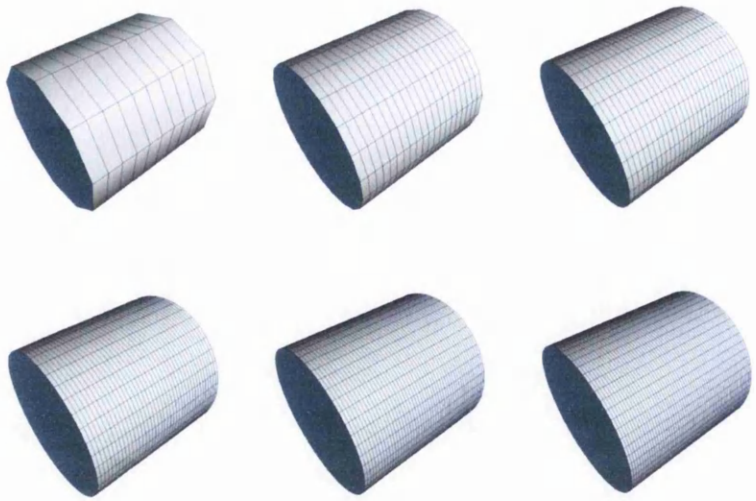


Figure 25. Increasing radial and lateral segments of a cylinder in steps of 10.

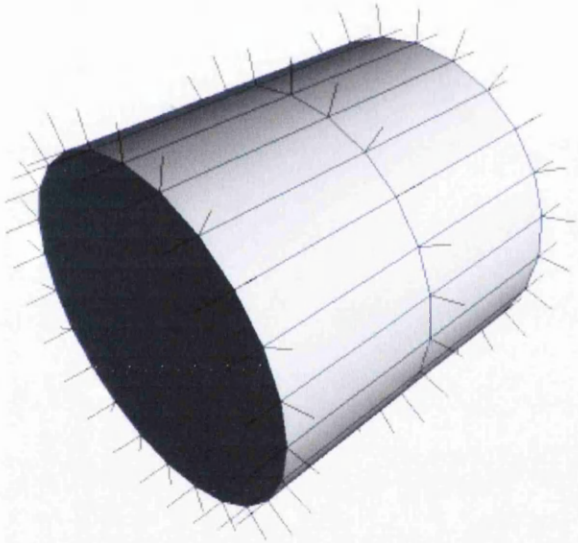


Figure 26. Multiple vertex normals per vertex.

3.2.6 The Cone

The cone is formed from a base circle and further rings of vertices that terminate in a single point at the apex. The distance from the base of the cone to the apex is one unit, the same as the radius of the base circle. As with previously mentioned geometry generators, the cone allows for parametric control of radial and lateral segments, the effect of altering these is shown in Figure 27. The cone also makes use of smooth groups; the base and the body of the cone are assigned to different smooth groups.

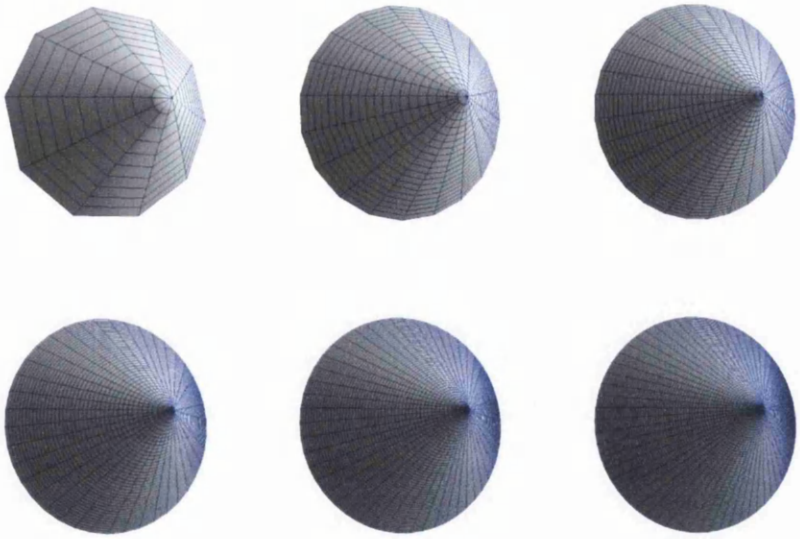


Figure 27. Increasing the radial and lateral segments of a cone in steps of 10.

3.2.7 The (super parabolic) torus

The torus is a formed by creating rings around a central point and connecting them together to form a tube of a given polygonal resolution. Figure 28 shows the effect of increasing this resolution. The cross section and ring shape can also make use of the same super parabolic functions used in the super ellipsoid, allowing for a pinched shape in both the cross section and in the sweep form of the tube.

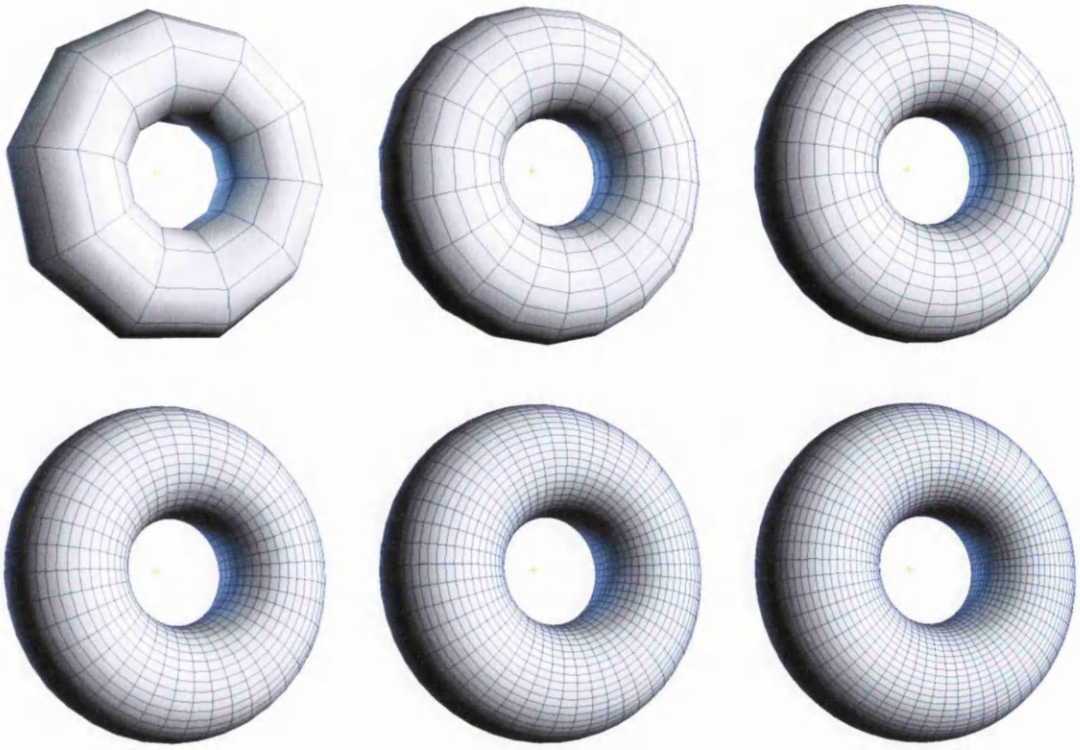


Figure 28. Increasing radial and lateral segments of a torus in steps of 10.

3.2.8 The platonic solids

As well as the cube, the system provides render nodes capable of creating all of the platonic solids; i.e, the dodecahedron, tetrahedron, icosahedron and octahedron, shown in Figure 29. These shapes do not have any parameters but provide interesting base shapes upon which future modifiers can be applied, as seen in Figure 30.

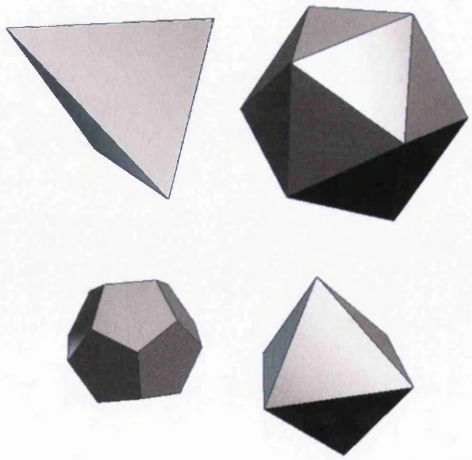


Figure 29. The platonic solids.

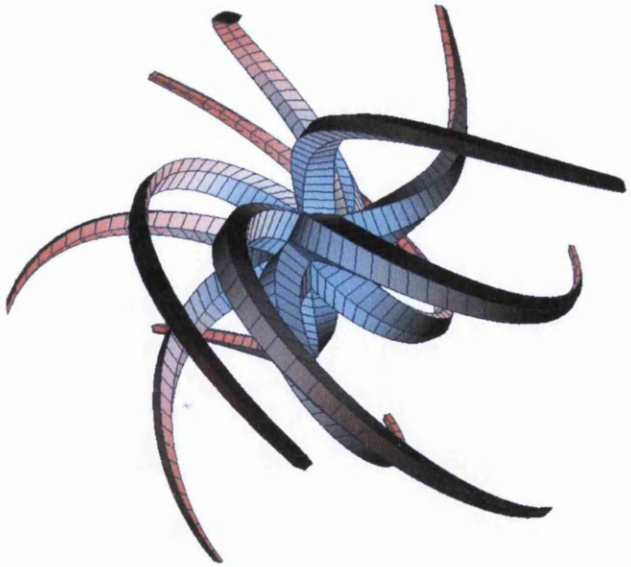


Figure 30. Result of a per-polygon extrude and rotation on a dodecahedron.

3.2.9 The geometry cache node

This node is classified as a geometry generator but doesn't actually procedurally generate anything; it is just a resolution fixed cache of geometric data. This data type is included for compatibility with other 3D data formats that do not store data procedurally. The data from such objects can still, however, be manipulated in a procedural manner, as is shown in Figure 31.

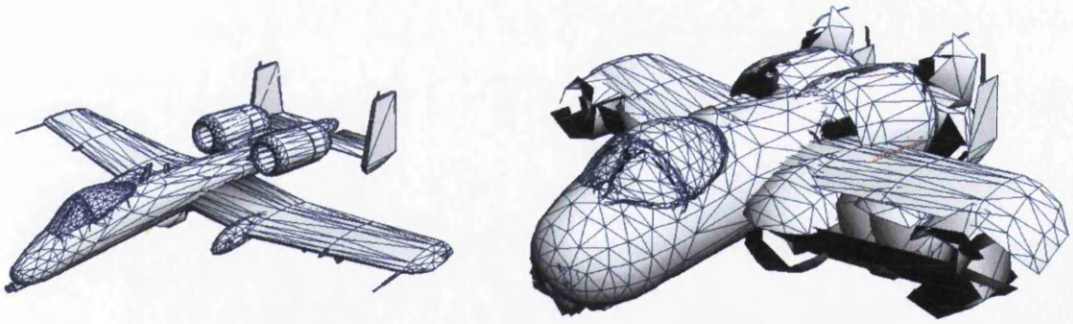


Figure 31. An A10 tank killer plane, imported from a popular online game (Desert Combat, http://www.desertcombat.com) is stored as a geometry cache and deformed using the distance field modifier.

3.2.10 Summary

This section has introduced geometry generator class of nodes. Whilst most of the

geometry generator nodes are parametric in nature, there are some nodes for which it makes little sense to have parameters such as the geometry cache nodes (which represent generic polyhedral shapes and the platonic solids which are specific polyhedral shapes). The mechanism by which the system registers new node classes is designed to be flexible and as geometry data is passed from all geometry generator nodes in a standard format, adding new geometry generator nodes is a trivial matter.

The geometry that leaves the geometry generator nodes is altered by the use of modifiers. However, in order to have some effect, an area of the object (or the whole object) must be selected in order before performing any modifications. The subject of selection is covered in the next section.

3.3 Selection

Selection is one of the most important aspects of any modelling program in as much as the user will identify an area or volume of an object that will be manipulated or modified. In most modelling systems, selections are stored as references to specific control points on the surface of an object, tying the selection to a low-level representation of an object. It is this connection to low-level representations that limits current modelling techniques. This thesis describes a system which breaks this dependence by storing the selection region, rather than a list of selected primitives.

3.3.1 Geometric data flow and selection

The geometry generator nodes output a data type that encapsulates geometric information in a low level format (often polyhedral shapes but curved surfaces can also be represented). This data is split into channels within this data type, including channels for control vertex information (position, UV-coordinates, colour, etc.), topological connectivity (i.e, surfaces formed by references to vertex information) and also a selection channel which refers to the selection status of low-level shapes defined in the other data channels. It is important to note that while the data that is passed from a geometry generator and through subsequent selection modifiers and shape modifiers is low-level in nature, the definition of the object that is stored and manipulated by the user is the higher level description; i.e, the actual geometry generator nodes, selection nodes, modifier nodes and the connectivity information that describes the flow of data between these nodes.

The geometric data flows from the geometry generator (responsible for creating the node), through various geometry modifiers before reaching a node capable of rendering this

data.

The Figure 32 shows a very basic node arrangement consisting of a material, sphere geometry generator, a rendering mesh and a transform.

The material is a resource node and this is shown in deep yellow. This node feeds into the sphere geometry generator and provides the base material for this object. The geometry generator is responsible for creating the mesh data and the flow of this data is represented by the red arrow leading from the node marked 'sphere' into the node that finally renders the object, marked 'render mesh'.

The transform node provides a world space matrix for the object and this also feeds into the render mesh.

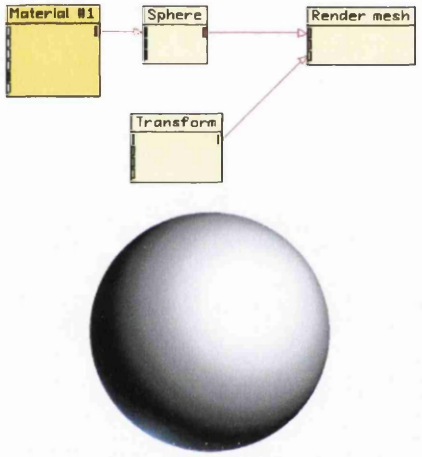


Figure 32. Detail of graph depicting flow of data from a sphere geometry generator to a render mesh with no modifications.

Once modifiers are added, the picture becomes more complex. Figure 33 shows the sphere generator feeding into a selection modifier node. The selection modifier alters the selection channel of the mesh, selecting elements based on the action the user took to select those elements. This in turn feeds into a stretch modifier which alters the vertex channel of the mesh data. This then feeds into a further selection modifier node (this time it clears the selection channel) and finally, as in Figure 32, this feeds into the rendering node.

Figure 34 shows a real world example of an object represented using this system. The plane model took 17 modifier and selection steps to transform a sphere into an object

recognizable as a passenger jet and these steps are shown in Figure 35. Figure 36 to Figure 38 show the effect of altering the resolution of the base node and of the extrusion nodes, showing the increase in resolution of the final shape.

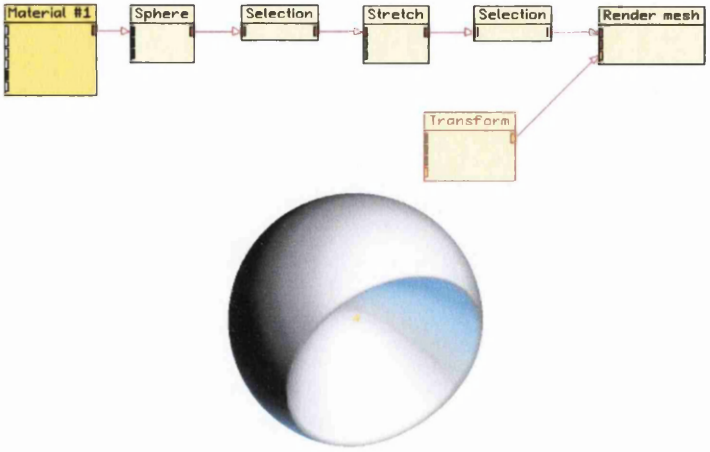


Figure 33. A more complicated flow of data from the sphere generator, through a selection modifier and a stretch modifier.

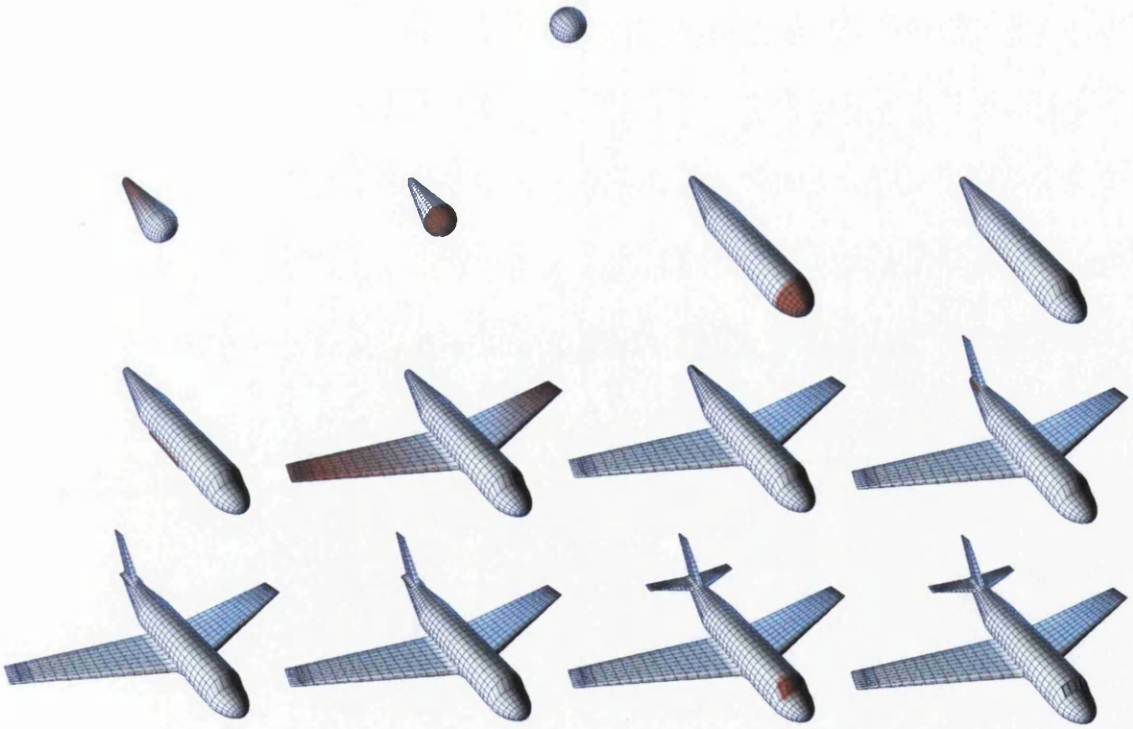


Figure 34. This image shows the steps taken to transform a sphere into a model of a passenger airplane.

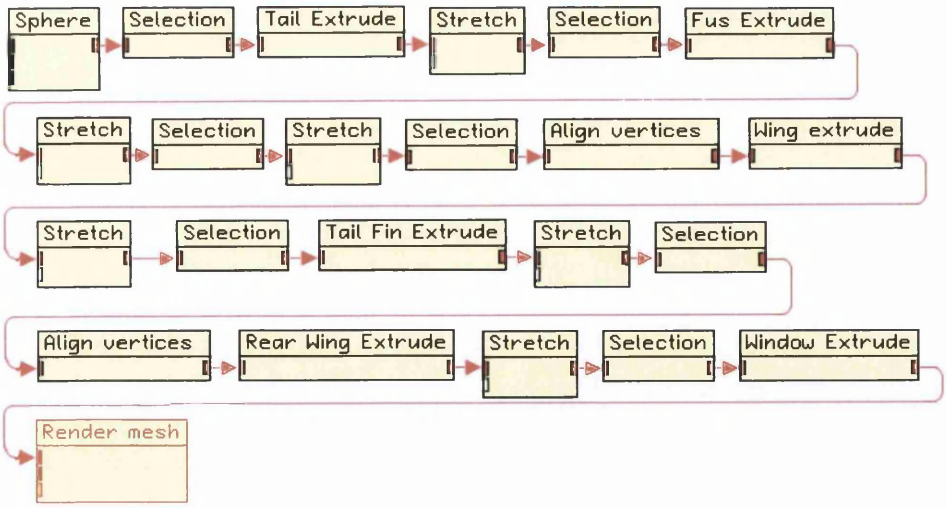


Figure 35. This image shows the view within the graph editing window.

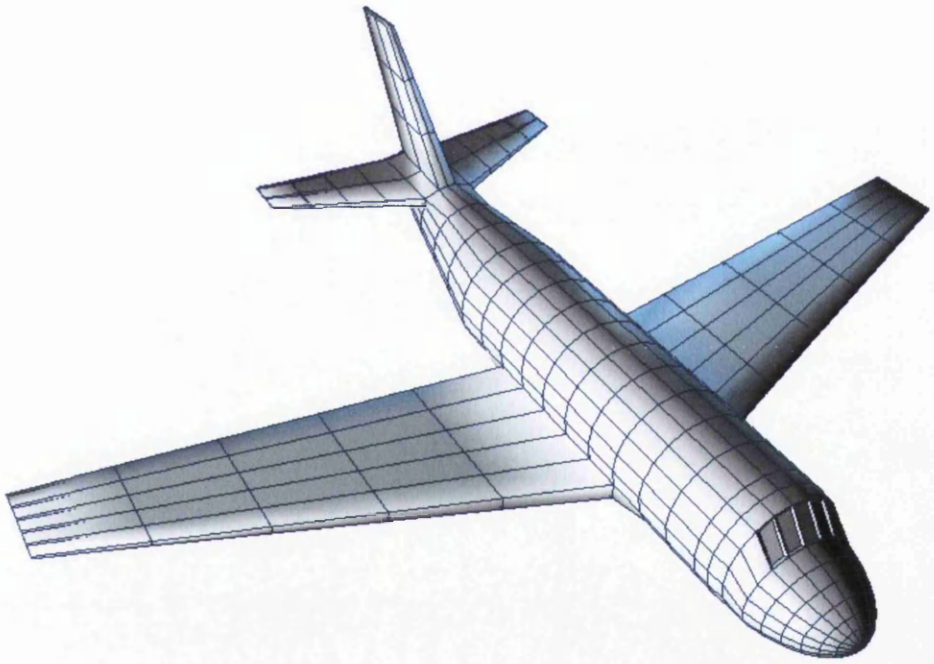


Figure 36. This image shows the plane mesh. The resolution of various parts of the mesh was reduced after the object was created.

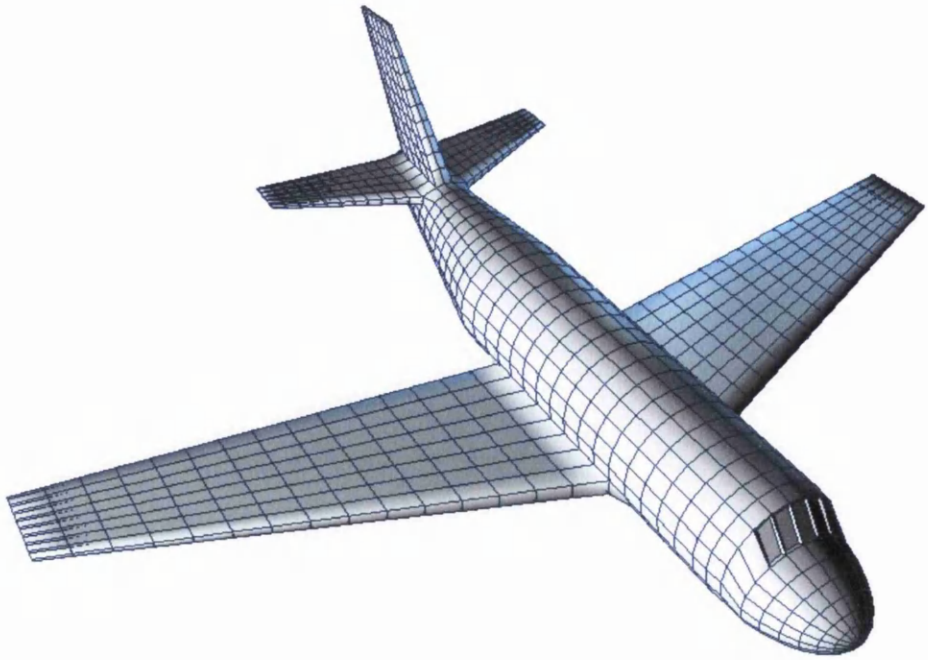


Figure 37. This image shows the plane mesh at the resolution it was created.

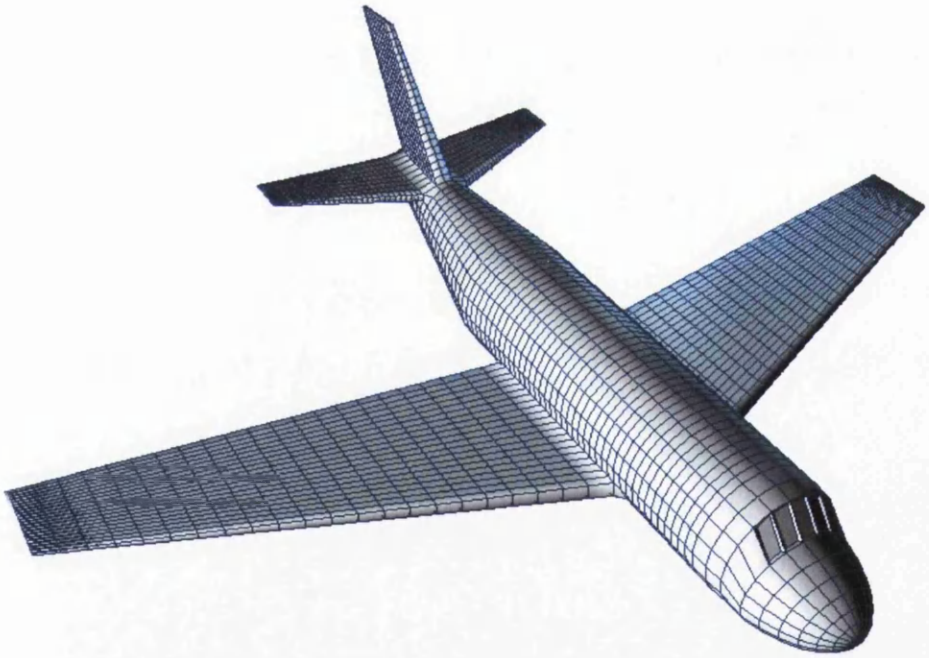


Figure 38. This image shows the plane model at a higher resolution.

3.3.2 Selection Geometry

The selection modifier contains a set of 3D geometric forms called *selection geometry*. The geometry inside this list is tested against the incoming mesh geometry to check for intersections. Every selection geometry in the list has an associated operation attached to it. The following table shows the operations available:

Table 2. Bitwise operations used in selection.

SET	Ignores any previous geometries, the selection includes all elements within the bounding volume.
OR	Adds the elements within the bounding volume to the selection.
XOR	Inverses the selection of any elements within the bounding volume
AND	Only keeps the selection of any elements within the bounding volume; all previously selected elements outside are de-selected.
AND NOT	Removes the elements within the bounding volume from the selection.

Each operation has a specific effect on the selection. Selection geometries are added to the selection modifier via selection events from the scene (scene events and how they interact with objects is discussed in detail in a later section). They are only added if they actually make a change to the selection, e.g, an 'AND' operation on a mesh with no selection will not add a geometry to the list and neither will adding a geometry that reselects elements that are already selected and affects nothing else. By using Boolean operations, complex selections are possible within a single selection modifier.

A selection geometry can be any procedurally defined shape within which it is possible

to do an inside/outside test. i.e, any closed shape could be a possible selection geometry. The most commonly used geometries are extrusions of shapes drawn in 2D onto the view plane of an editing window, such as a frustum or truncated cone. These are the most intuitive shapes to use in a system that displays 3D information in 2D windows.

The following table shows the types of selection geometry that are available. New types of selection geometry can be added to the system at run time using third party extensions.

Table 3. List of selection geometry objects.

Rectangle	A frustum defined by a rectangle 'drawn' onto the view plane, shown from several angles in Figure 39.
Ellipse	A conical projection, taken from an ellipse projected through the view plane.
Lasso	As above only the projection is an arbitrary enclosed shape drawn onto and projected through the view plane. The lasso profile is stored as a cubic curve.
Poly-line	An angular closed polygonal shape, drawn on to the view plane and projected.
Sphere	A sphere in 3D space.
Cube	A cube in 3D space
Cylinder	A cylinder in 3D space
Cone	A cone in 3D space
Pyramid	A pyramid in 3D space
Torus	A torus in 3D space

Super ellipsoid	A super ellipsoid in 3D space
Convex Hull	A convex hull as defined by a list of vertices
Closed Mesh	An input mesh shape that is closed (i.e, no loose edges) and does not self-intersect.

Each type of selection geometry is defined by a function in the form $f(x,y,z)=0$ where the result is 0 if the point defined by the parameters to the function is on the surface. Fall off is calculated by first checking to see if the point falls within the iso-surface range of the selection volume and then normalizing this value and passing it into a function that defines the falloff curve.

The sphere selection geometry is defined by the following function:

$$f(x,y,z)=\sqrt{(x-c_x{}^2+y-c_y{}^2+z-c_z{}^2)}-r=0$$

where r is the radius of the sphere and c is the centre of the sphere. A point is considered selected if the result of the function is less than or equal to zero. The list given above is far from exhaustive; any 3D shape or projected 2D shape can be used as a selection shape. It is also important to note that the shapes used to select a region can be edited at any time. If a projected circle were used to mark a region and this was followed by a delete modifier in order to create a circular hole in an object, this projected circle could later be moved, in effect moving the hole. The same is true for any other modifier; the selection shape itself becomes a tool to edit the object in a parametric fashion

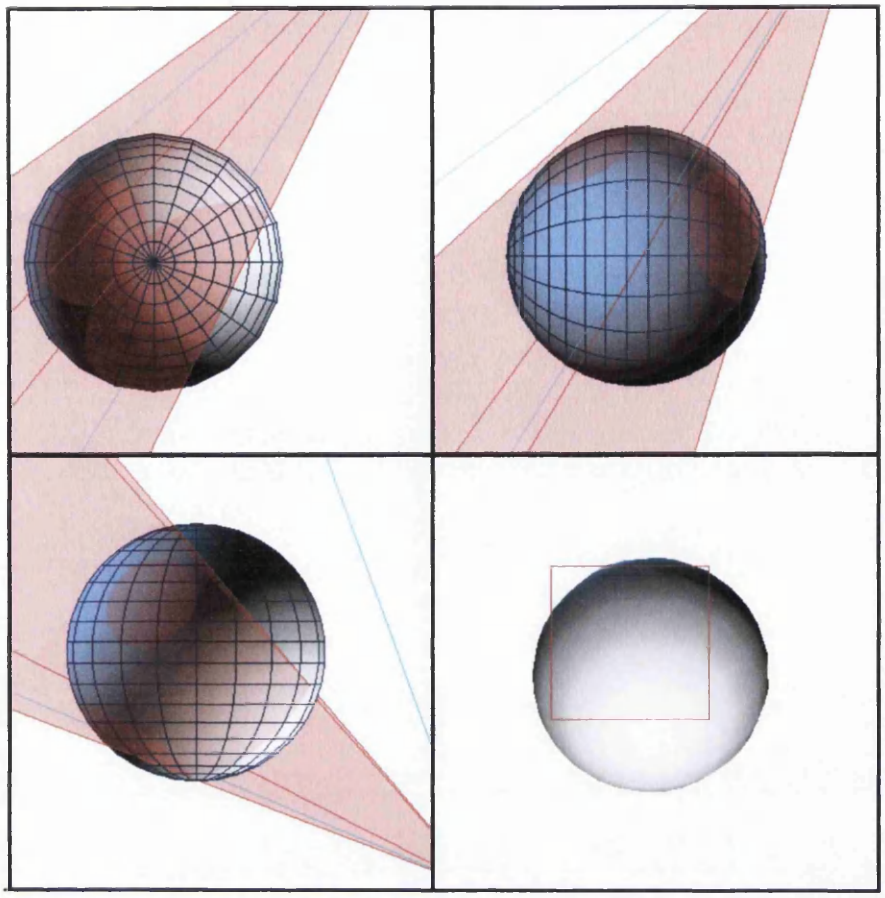


Figure 39. The above image shows a selection frustum as created in the perspective window (bottom right).

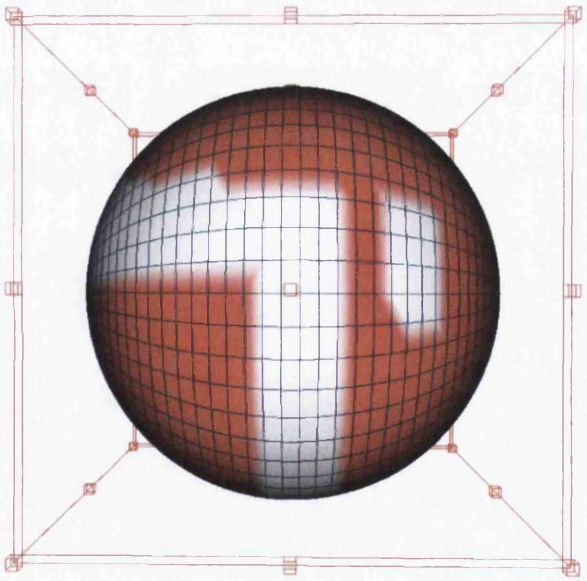


Figure 40. This image shows a complex selection formed from 3 additive frustums and one subtractive *(and not)* frustum.

One area of difficulty when trying to implement a method of selection that relies on volume is how to deal with selections that are actually defined by the underlying topology. Dealing with these forms of selection might seem counter productive as we are trying to step away from reliance on underlying form. However, in practical applications, it is often useful to select an area on a per-element (edge, polygon or vertex) basis. To this end, per-element selections are allowed but in keeping with the volume selection ideal, they are not stored as per-element references. Rather, a convex hull (or series of convex hulls depending on the connectivity of selected areas) are stored that represent the selected area at the resolution it was first defined in. If the resolution of the underlying mesh is altered, the selection will still define the area that the user originally intended. The downside to this approach is in storage space; a selection defined by a convex hull will usually take up more space than a bit array defining selected elements, depending on the complexity of the hull. However, it may be possible to decrease the resolution of the hull and use curve fitting algorithms to describe the area in a more efficient way, using NURB patches as opposed to a convex hull formed of polygons.

3.3.3 **Ideal or 'Cookie Cutter' vs. Snap selection**

Whilst selections are volumetric, the visual result of these selections will appear at the resolution of the underlying mesh; A circle selection will only appear roughly circular, depending on the underlying resolution of the mesh. Such selections are termed 'snap' selections. Another way of handling selections actually changes the underlying geometry of the object by cutting it along the border of the selection volume. Such selections are termed 'ideal' selections.

There are some situations where having the selection snap to the underlying geometry is desirable as ideal selections will create more geometry and add to the complexity of a model. However, ideal selections allow for simple and effective means to add detail to an object. For example, a poly line selection could be used to etch a symbol onto a curved surface which could then be extruded to form a bass relief. Such an operation would otherwise be time consuming a difficult, requiring the user to create the pattern separately and carefully distort it so that it matches the original surface and then perform a Boolean union between the two objects.

Ideal selections also allow for the creation of objects at a lower resolution but with the visual appearance of objects created at a much higher polygonal resolution.

Whilst snap selection is fairly trivial to implement, ideal selection involves more complex operations. The process of selection operates by first defining a list of vertices that are inside the selection volume. The polygons and edges are also flagged as selected if all the vertices they refer to are selected. If only some of the vertices are selected then the polygon is flagged as being 'affected' by the selection, whilst not actually being selected; Affected polygons are often altered in some way by subsequent modifiers; a move operation will move selected polygons but distort affected polygons and often, extra processing has to be performed on these affected regions to insure that polygons do not become non-planar or degenerate. This creates a selection of polygons, edges and vertices that can be referred to by subsequent operations and a list of polygons and edges that surround the selection and will need to be updated if the vertices of the selection move.

In the case of ideal selection, the above procedure is followed but it is taken one step further. The affected polygon list (prior to alteration by fall-off calculations) contains a list of polygons and edges that intersect the selection volume. This information can then be used to perform an efficient cutting operation that inserts new vertices (at a specified density) along the edge of the selection volume. Note that the selection volume itself is formed from a series of shapes via Boolean operations so the selection volume itself may have a very complex form. The system finds the edges where the selection volume intersects the original mesh and inserts new vertices on these edges at the point of intersection. New vertices also must be inserted across the polygons or surfaces that intersect the selection volume. Vertices are inserted along polygonal surface when the tangent of the line formed by the intersection of the view volume and a surface goes above a certain threshold; the lower the threshold, the greater the apparent detail will be for curved selections. However, angular selections will not insert as many new points.

3.3.4 **Selection fall-off**

So far, we have discussed selections as hard-edged intersections between a selection volume and a shape which either select control points that fall within this volume or, in the case of ideal selections, the selection is actually the intersection between this volume and the shape being edited. In many situations, it is desirable to create a smooth edge to this selection and this is achieved by utilizing the selection fall-off feature. Figure 41 and Figure 42 show selection fall-off and the effect of fall-off when applying a stretch modifier to an object. Fall-off information is generated in one of two ways; either by a modifier such as the copy or extrude modifier, which will apply differing levels of influence to copied regions or

extrusion levels or in a geometric fashion, as is the case with the selection modifier.

The selection modifier applies fall off by measuring the distance between a vertex level element of the low-level representation and the edge of the selection geometry boundary.

The selection modifier node allows the user to specify a fall-off curve function, $fcurve(x)$, and a scalar value, τ, which defines the geometric limit of the fall-off. The fall-off function accepts normalized values in the range 0 to 1.

The selection geometry forms a 3D volume and the distance between a point (x, y, z) and this surface is defined as λ.

$$f(x,y,z) = \lambda$$

If a point is inside the selected region (i.e, $\lambda <= 0$) then it considered is wholly selected, with a fall-off value of 1.0.

If the point is outside the selected region but inside its fall-off region (i.e, $\lambda < \tau$) then the fall off value is defined by.

$$falloff = fcurve(\lambda / \tau)$$

If the point is outside the fall-off region (i.e, $\lambda >= \tau$) then it is not considered selected at all and has a fall-off value of 0.0.

3.3.5 **Effect of modifiers on the selection channel**

Although the selection modifier is the central means of controlling the selection channel of the mesh data object, many modifiers will modify the selection channel in some way. Any modifier that adds data to the mesh stream, rather than those that alter mesh information without adding new vertices, edges, polygons or other elements, will in most cases alter the selection channel. The manner in which the selection channel is altered depends entirely on the modifier in question. The next section explains in detail the effect of each modifier in the system, including its effect on the selection channel.

3.4 Modifiers

Modifiers are nodes that receive mesh data and alter it in some way. Different nodes will affect different combinations of mesh data channels; some will only alter the vertex colours

or texture coordinates whilst others will rebuild the whole object based on some procedural function. All of the modifiers implemented in the system make use of the selection channel which is used to determine which areas of the object will be affected by the modification. For example, a modifier that alters the position of vertices in the mesh will only alter the positions of those selected vertices. This includes all vertices selected directly and also those affected by the selection fall-off region.

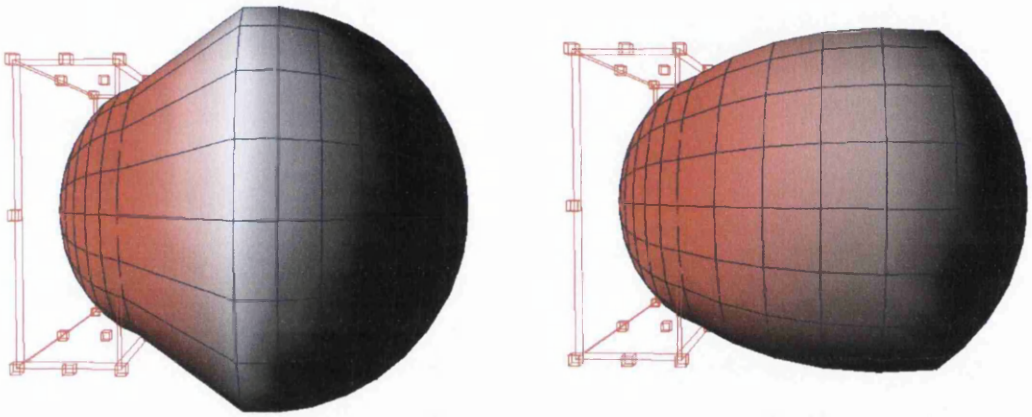


Figure 41. This image shows the effect of altering the selection fall off when a stretch modifier is applied.

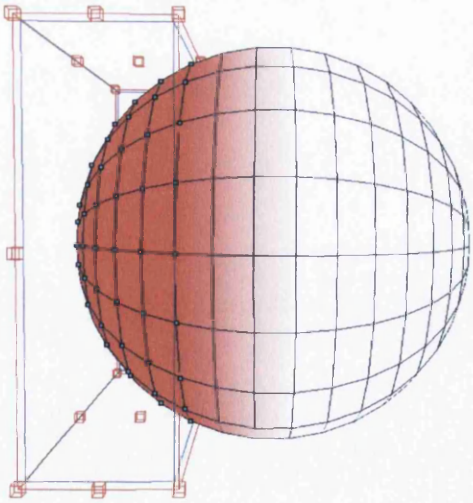


Figure 42. Vertex Selection and fall-off. The red area and the vertices highlighted with squares shows the selected area.

The use of fall-off allows certain modifiers to decrease the effect they have on vertices,

as shown in Figure 41.

Other modifiers, such as the 'extrude' modifier or the 'delete' modifier, alter the polygon list of an object in some way. Others will alter only the texture coordinates, smooth groups or vertex colours of an object. The term 'modifier' is really an umbrella term for a whole host of nodes that have one thing in common; they alter one or more of the data channels within a mesh object and pass these changes along a chain to another node. In keeping with the procedural design philosophy of the system, modifiers should be able to reconstruct the output data based on internal parameters no matter what the input data.

The implementations of modifier nodes roughly fall into three categories. The first, and simplest, category involves modifiers that alter the values of per-vertex level element such as position, texture coordinates, colours, weights or other coefficients. These modifiers do not require any new elements to be added, or any re-indexing of edges or polygons. The second category includes modifiers such as 'extrude', 'triangulate' and 'subdivide'. These modifiers do not remove any per-vertex level elements but they can add to the vertex level lists. No re-indexing of edges or polygons that are unaffected by the selection is required. Selected edges that are affected by the selection need to be recreated using newly created vertex level information as well as references to the old vertices. The third category includes modifiers such as 'delete' and 'merge'. These modifiers are destructive in that they remove vertex level information and require the polygon list to be re-indexed. Whilst helper functions automate the re-indexing process for such modifiers, the implementation of such modifiers is often more involved than other types of modifier.

3.4.1 The Shrink/Grow/Translate modifier

This modifier adjusts the selected vertices so that they fit inside a bounding cube (see Figure 41 and Figure 42). The cube is defined by two vertices which form its corners and also a matrix which is used to align the cube. On creation, the matrix is the inverse of the world space matrix of the mesh in question and the corners are the local space, axis aligned bounds of the selected region. In this configuration, the modifier has no effect but as soon as the user alters the bounding volume (using either the node panel of the selection or the control widgets of the modifier), the selected vertices are altered.

The modifier creates a matrix based on the difference between volume of the selected incoming vertices and the target bounding volume. This matrix is pre-concatenated with the world space matrix and post-concatenated with the inverse of the world space matrix.

The transformation matrix used by this operator is as follows:

$$m = \begin{bmatrix} (tb_x - ta_x)/(sb_x - sa_x) & 0 & 0 & 0 \\ 0 & (tb_y - ta_y)/(sb_y - sa_y) & 0 & 0 \\ 0 & 0 & (tb_z - ta_z)/(sb_z - sa_z) & 0 \\ -sa_x(sb_x - sa_x) + ta_x & -sa_y(sb_y - sa_y) + ta_y & -sa_z(sb_z - sa_z) + ta_z & 1 \end{bmatrix}$$

where *sa* and *sb* define the source bounding box and *ta* and *tb* define the target bounding box.

3.4.2 The Rotate Modifier

This modifier operates in a similar fashion to the stretch manipulator only it rotates the selected vertices around a point, rather than stretching them.

The rotation matrix is formed from a quaternion value which defines the rotation and a vector defining the centre of rotation.

$$m = \begin{bmatrix} 1 - y^{2y} + z^{2z} & x^{2y} - w^{2z} & x^{2z} + w^{2y} & 0 \\ x^{2y} + w^{2z} & 1 - x^{2x} + z^{2z} & y^{2z} - w^{2x} & 0 \\ x^{2z} - w^{2y} & y^{2z} + w^{2x} & 1 - x^{2x} + y^{2y} & 0 \\ p_x m_{00} + p_y m_{01} + p_z m_{02} - p_z & p_x m_{10} + p_y m_{11} + p_z m_{12} - p_y & p_x m_{20} + p_y m_{21} + p_z m_{22} - p_z & 1 \end{bmatrix}$$

where x, y, z and w are components of the quaternion that defines the rotation and p is the point around which the rotation takes place. The matrix refers to itself, for the sake of brevity, in the final row. However, as these references are not defined in this row, no unexpected results occur. The matrix subscript is defined as m_{00}, m_{10}, m_{20}, m_{30} for the top row.

The rotate manipulator that creates the quaternion value is formed by 4 rings which can be grabbed and rotated. Three of the rings are axis aligned, representing yaw, pitch and roll. The forth always faces the camera and rotates along the axis of view. As the user moves the mouse over a rotation ring, the ring is highlighted (as shown in Figure 43) which is a user interface feature used throughout the system, both in 3D and 2D user interface elements, to indicate that clicking the mouse will have some effect. When the user holds and drags the mouse in a circular motion, a translucent arc appears in that axis indicating how many

degrees of rotation have been completed.

Similar control methods are used in the software packages Maya and Cinema 4D although these do not allow the centre of rotation to be changed when editing the rotation.

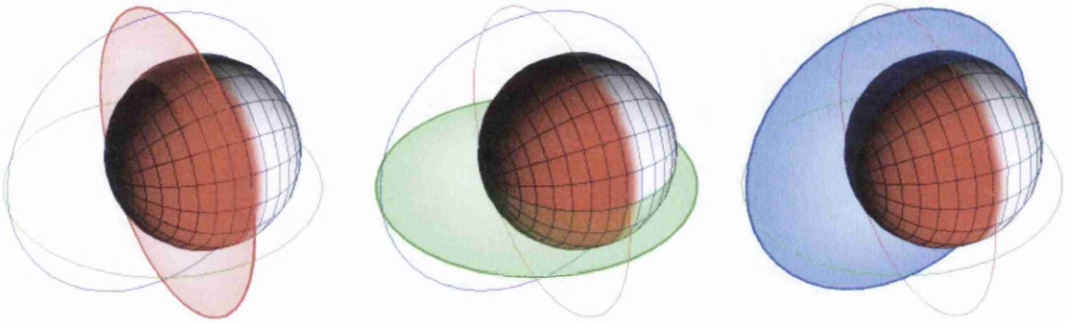


Figure 43. The rotation tool.

3.4.3 The Extrude Modifier

The extrude modifier creates an extrusion at the edge of the selection, as shown in Figure 44. This is akin to splitting the object along the seam formed by the edge of the selection and then 'bridging' this seam back together again with new polygons. The modifier can specify how many steps the bridge can take to get from one side of the seam to the other. The selection channel is also modified, creating a linear, vertex connection based fall off from the previously selected area of polygons to the unselected region. Further operators, such as stretch or rotate can then move the selected vertices around and observe a smooth falloff of the effect.

The modifier works by copying all of the polygons from the input mesh that are not on the edge of the selection, then copying the selection border vertices for n times where n is the number of layers required in the extrusion. The border polygons are then recreated using the new vertices and quadrangles are formed to stitch the unselected portion of the object to the selected portion, using the newly created vertices.

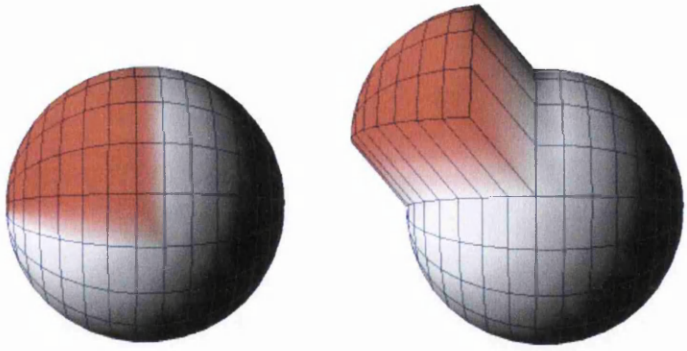


Figure 44. This image shows an extrusion created by selecting a portion of a sphere, creating an extrusion and then translating the extruded section.

3.4.4 The Per-Polygon Extrude Modifier

The polygon extrude modifier operates in a similar manner to the standard extrude modifier. The difference is in how this modifier defines the seam between which new polygons are inserted. In the case of the polygon extrude modifier, the seam is down every polygon edge and invoking this modifier results in the creation of a border around each selected polygon. It is then possible to scale these polygons around the centre of the polygon and translate the along the normal of the polygon which is useful when creating bevels, as shown in Figure 45.

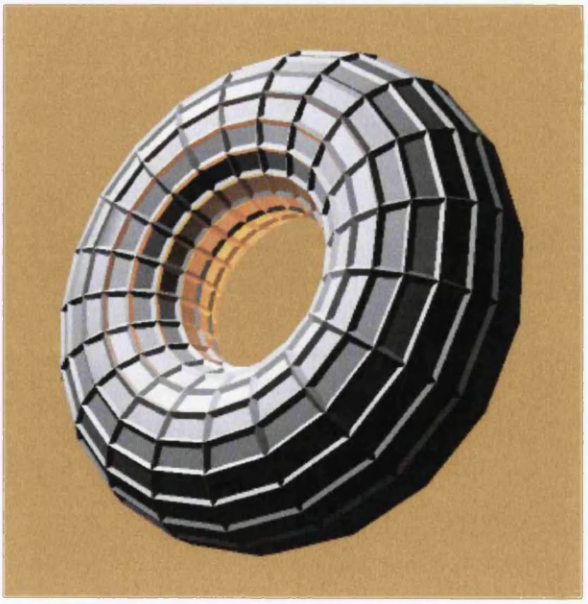


Figure 45. A torus with a per-polygon extrusion modifier applied, creating an attractive beveled appearance.

Since the per-polygon extrude function operates on a polygonal mesh, this modifier does in fact break the paradigm independent methodology since it needs polygons or patches to operate. However, the modifier itself is still procedural and the shapes that result from using this modifier are often useful and interesting.

For a fully generative approach to this modifier, it will be necessary to supply an input shape that can be mapped and tessellated over the surface of the object. These shapes, rather than underlying polygonal representations, can then be used as the basis for an extrusion. An example of a useful application of this modifier is shown in Figure 46 which represents a cog. The flow diagram for this graph is shown in Figure 47.

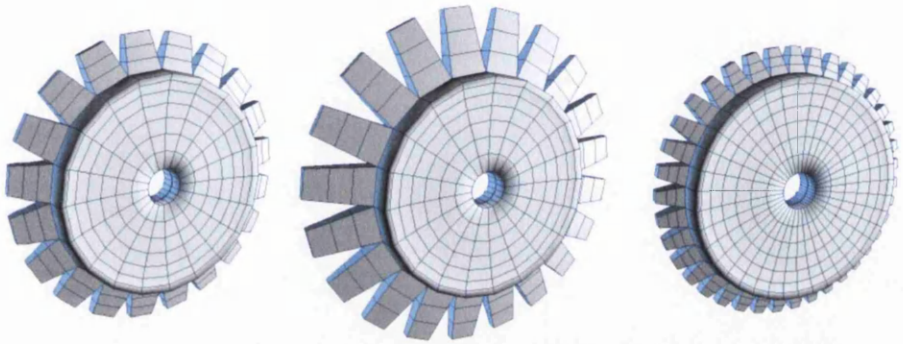


Figure 46. Creative use of the per-polygon extrusion modifier

In Figure 46, we can see an object that started out as a sphere. The central band of polygons was selected and a per-polygon extrusion applied, forming the spokes of the cog shape shown. The sides were then flattened using the stretch modifier and a hole created in the centre by selecting the centre vertices, extruding and stretching towards the centre, deleting the capping polygons and stitching together the selected vertices.

A total of twelve operations (including modifiers and selections) were used to transform the sphere into the cog object shown. Even after all these modifications, it is possible to alter the size of the spokes (centre image) and even the number of segments in the original sphere (and therefore, the number of spokes on the cog) and to then recalculate the final shape at interactive rates.

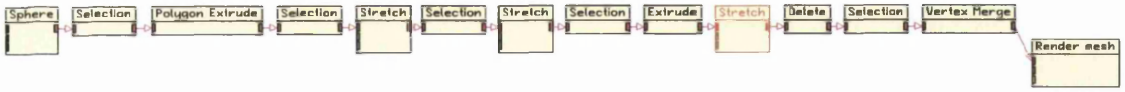


Figure 47. The series of modifiers used to mold a sphere into a cog.

3.4.5 The Smooth Group Assignment Modifier

This modifier assigns a smooth group to any selected polygons in the incoming mesh. The purpose of a smooth group is essentially to define which vertices will share a normal for lighting calculations and which will not. The most common use of this is to create visually sharp edges between polygons that share vertices but are at obtuse angles to each other.

Smooth groups can be automatically assigned based on the angle between two polygons as shown on the left-hand side of Figure 48.

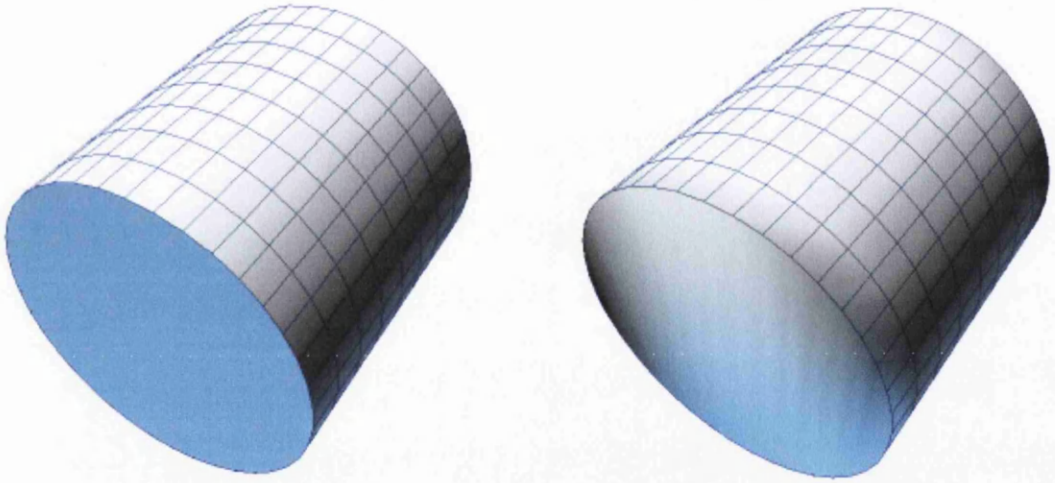


Figure 48. The cylinder on the left has three smooth groups whilst the right cylinder has only one. Notice that the shading on the right hand cylinder that shows shared normals along the sharp edge of the cylinder. The example on the left does not share normals and is shaded in a more aesthetically pleasing way.

3.4.6 The Subdivide Modifier

The subdivide modifier adds detail to the selected polygons from the incoming polygon mesh. The modifier creates a number of translation tables for each per-vertex element which are used when recreating the polygon mesh at a higher detail level.

Translation tables are used extensively by modifiers that alter the topology of the object

in some way. They usually refer to per-vertex elements in the mesh and have as many entries as there are edges in the incoming mesh. The memory for these tables is always taken from an application wide heap which is specially managed for speed-wise efficiency. In the case of the subdivision modifier, the table is used to refer to new vertex elements inserted between edges. The half-edge list of the incoming polygon mesh is traversed and at each selected half-edge, a new vertex is inserted with care being taken to analyse the twin of this half-edge to see if it shares vertex information (half-edge twins always share position information but may have different colour, texture coordinate or weight information).

When building the new polygons, the edges of the selected polygons from the incoming mesh are traversed and new polygons (always quadrangles, see Figure 49) are created around the old vertex (the same as the one from the incoming mesh), the two new vertices to either side and a newly created central vertex.

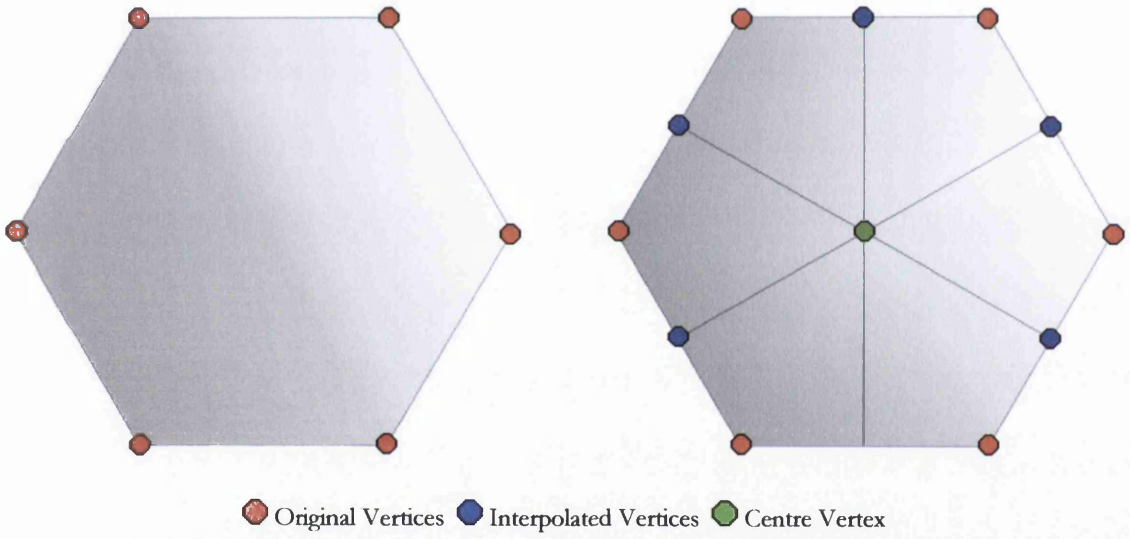


Figure 49. Image showing a subdivided hexagon.

By default, the subdivide modifier interpolates new vertex positions linearly. However, it is also possible to create a smoother mesh by use of subdivision. The method employed in this system is derived from the modified butterfly scheme [ZORN], which derives from the butterfly scheme [NDYN]for smoothing subdivided surfaces. The main difference with the approach taken in this thesis is that the system described here allows for an input mesh containing polygons of arbitrary edge count. The subdivide modifier alters the selection channel by selecting the new subdivided polygons in a manner that matches the original selection. That is, if one face of a cube has been selected and subdivided, the four new

polygons that take the place of the old face will then be selected.

3.4.7 The Triangulate Modifier

The triangulation modifier takes an incoming mesh that contains polygons with an arbitrary number of sides and outputs a mesh containing only triangles as seen in Figure 50 and Figure 51. The triangulation routine used is the same routine that is used by the triangle stripping routine, utilized by the mesh rendering node when creating triangle fans and strips for the output rendering library.

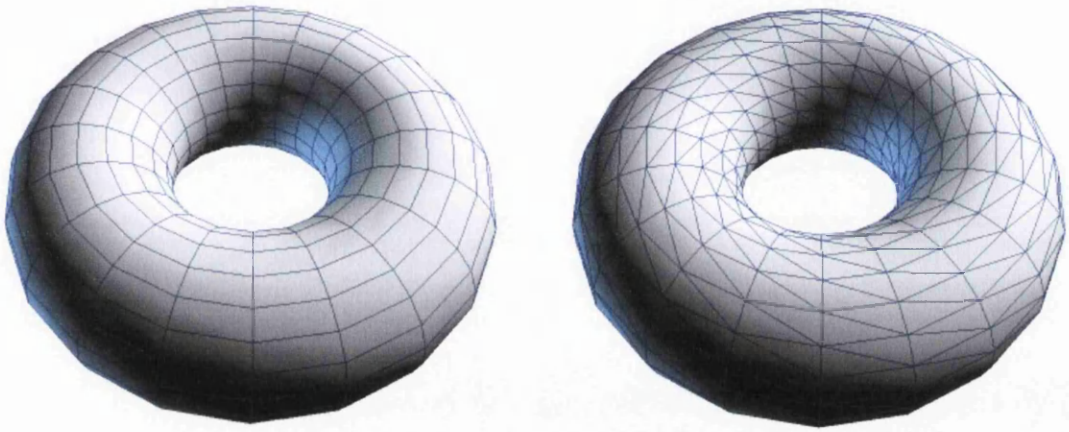


Figure 50. Two images showing the triangulation of a torus.

The triangulation routine can handle concave polygons but does not currently deal with self-intersecting polygons.

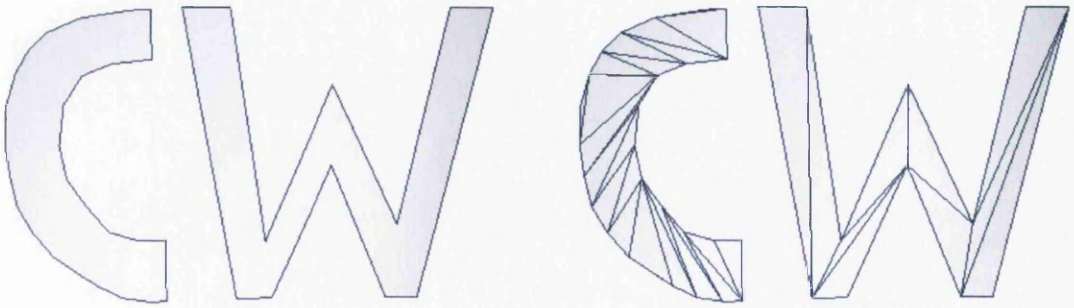


Figure 51. Image showing triangulation of concave polygons.

The triangulate modifier affects the selection channel in a similar manner to the subdivide modifier; The new triangles generated from incoming selected polygons will now be selected and this new selection passed down the modifier chain.

3.4.8 The Detach Modifier

The detach modifier separates the selected polygons from the unselected polygons as shown in Figure 52. In many respects, this modifier is similar to the extrude modifier only it does not perform the 'bridging' operation between the unselected and detached sections of the mesh. In order to detach a region of the mesh, vertices along the edge of the selection must be duplicated and the half-edges of the detached area modified so that they refer to the new vertices. Typically, references to other vertex-level elements remain unchanged but care must be taken to nullify the 'twin' reference of the half-edges along the selection edge.

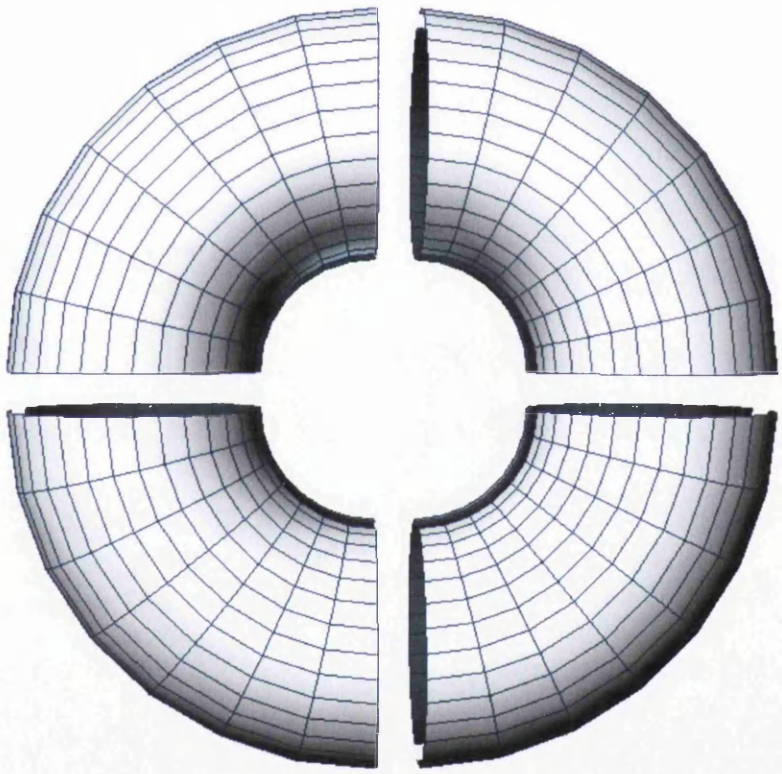


Figure 52. A torus, split down three axis by the detach modifier.

This modifier does not add any new edges or polygons to the mesh; it only adds new vertices which are duplicates of those found along the edge of the selection. Although the edge and polygon elements remain unchanged, the selection channel must be updated to take into account the new vertices that have been added.

3.4.9 The Polygon Copy Modifier

This polygon copy modifier makes a new copy of all the selected polygons and the vertices and edges that these polygons refer to. Making a new copy of the other vertex level elements is optional. The copy modifier first works out translation tables for edge, vertex and polygon level elements and then proceeds to add *n* copies of these objects to the original mesh. The selection fall-off channel is edited by the modifier so that each copy has a different fall off value (see Figure 53), allowing for easy manipulation by geometric modifiers such as rotate and stretch. Only one of the copies is considered selected and is shown in bright red. The other copies are only affected by the selection and are shown fading towards white to indicate the fall-off effect.

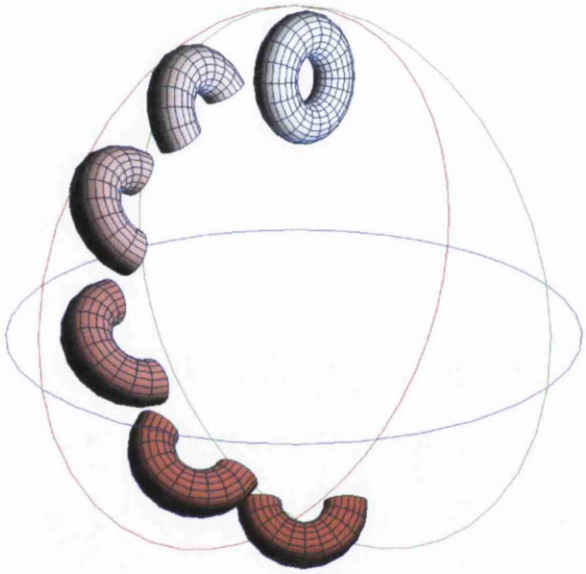


Figure 53. Effect of the rotation modifier applied to a copy of half a torus.

3.4.10 The Delete Modifier

The delete modifier does exactly what one would expect; it deletes the currently selected polygons. This is achieved by copying only the selected vertex level components, edges and polygons and re-referencing them in the output of the modifier. Whilst all selected polygons are removed from consideration, any edges that shared an edge with deleted polygons are now selected (see Figure 54) and this fact can be taken advantage of by any subsequent modifiers.

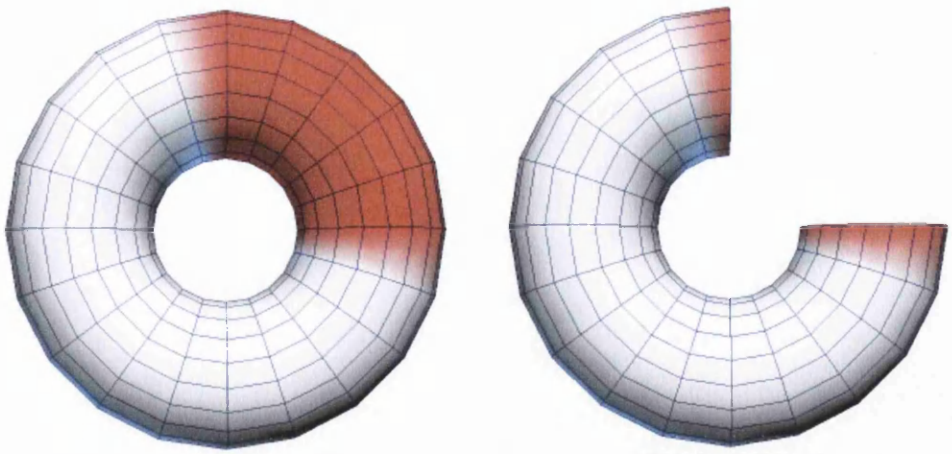


Figure 54. A torus with the top right quadrant removed. Note that the edge is still selected and this selection can be used by subsequent modifiers.

3.4.11 The Material Application Modifier

This modifier simply applies a new material to the selected polygon list. The material node feeds into the material application modifier and this in turn alters the per-polygon material channel of the incoming polygon mesh, as seen in Figure 55. When this list reaches the final rendering node, it is used to sort groups of triangle strips by their assigned material. Since state changes (particularly those involving texture maps) have a relatively high computational cost on current rendering libraries and hardware, it is advisable to sort via material when rendering.

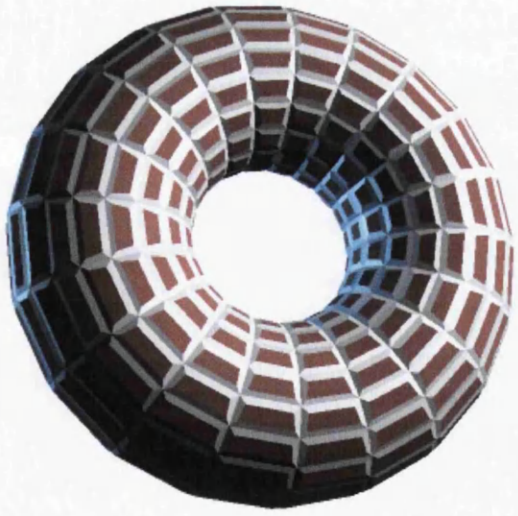


Figure 55. This torus as been modified using the per-polygon extrusion modifier and the resulting selected polygons have had a new material applied.

The material application modifier only affects the per polygon material channel; all other channels remain unchanged by this modifier.

3.4.12 The Vertex Merge Modifier

The vertex merge modifier is used to merge selected vertices together based on a tolerance value. Vertices that fall within this tolerance value will be merged into one vertex, the position of which is determined by the average of all the vertices within the tolerance range.

3.4.13 The Flip Normals Modifier

This modifier reverses the winding order of edges within polygons which has the effect of reversing the polygon normals (see Figure 56), forcing the selected polygons to be lit from the opposite side.

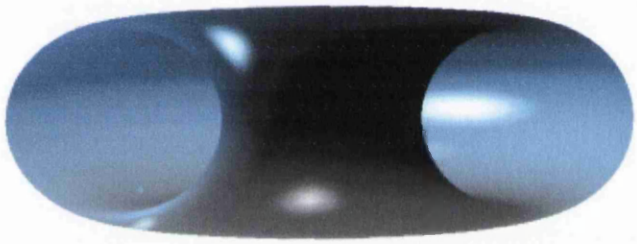


Figure 56. A tours inverted using the 'flip normals' modifier

3.4.14 The Plug Hole Modifier

The plug-hole modifier is a useful tool for closing a mesh that is not solid. The modifier searches through the list of selected half-edges, looking for half edges that do not have a twin. If a twin is not present, this indicates that edge is 'loose' and is not connected to any other polygons and the mesh is therefore not closed. If such an edge is found, the modifier attempts to find a loop of twin less half edges that lead back to the original half edge. One such a loop is found, the modifier plugs the hole with a new polygon (see Figure 57), the edges are flagged as being plugged and the modifier moves on to the next loose half edge. The modifier will not create a new polygon if the loose half edges already form a polygon so if the user tries to perform a plug-hole operation on a shape such as a square (which fits the criteria of having a loop of loose edges), the modifier will have no effect. The modifier doesn't attempt to find co-planar vertices along an edge either; therefore the polygons that are created may not be co-planar. This is acceptable however as the definition of a 'polygon' within the system is deliberately loose and polygons do not have to be co-planar and are

dynamically triangulated when it is necessary to do so.

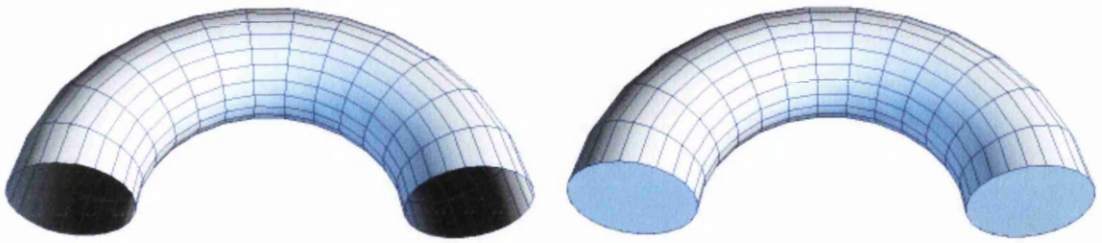


Figure 57. A torus, cut in half will no longer be a solid object. By applying the 'plug-hole' modifier, the two circular holes are closed by single polygons.

3.4.15 The Expand/Contract Modifier (pseudo distance field)

Distance fields are utilized in the system when performing selection; using a field function to define the shape of a selection provides a means to implement volumetric fall-off from selection shapes. The distance field modifier operates on mesh data and provides a simple means to visualize the expansion and contraction of an object through its distance field.

The modifier works by simply moving the vertices of the object through the average of all the normals of the surrounding polygons. This simplistic approach to the problem works as one would expect until the point at which polygons self-intersect (see Figure 58). At this stage, the modifier would have to clip polygons and merge edges in order to create a shape that was the true representation of the original shape with a modified distance field.

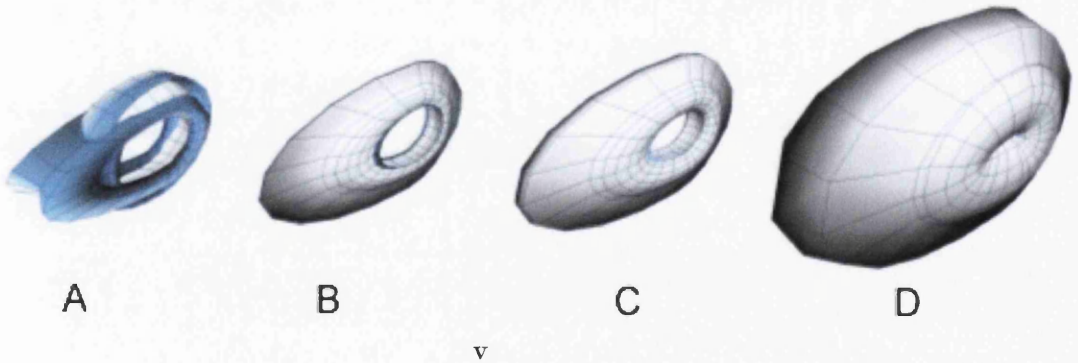


Figure 58. This Image shows the effect of the distance field modifier on a shape. The image labeled 'A' exhibits the erroneous artifacts associated with the modifier. 'B' shows the original shape and the 'C' and 'D' show modified distance fields without artifacts.

3.4.16 The Texture Perturbation Modifier

This modifier operates in a similar manner to the distance field modifier only it modulates the iso-parameter with a texture lookup. The modifier is supplied with a texture input from another node and the texture coordinates within the object are used to perturb vertices along their normal. The effects from this modifier can be quite striking and use of this modifier is a good way to add surface detail to an object quickly (see Figure 59). This modifier also represents a link between procedural texture generation and object geometry, opening up a whole range of new functions for the procedural deformation of shapes.

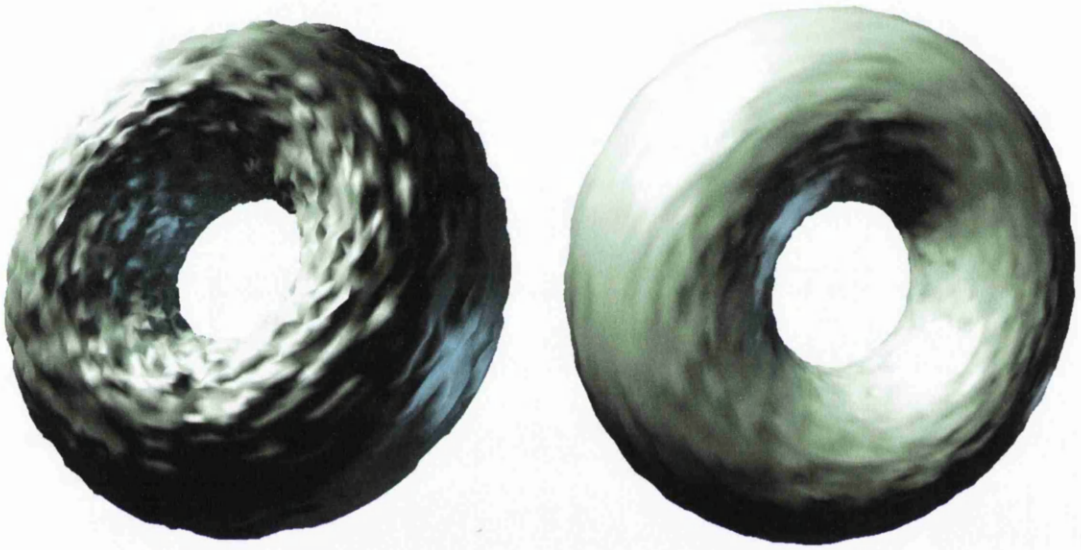


Figure 59. Torii distorted with the texture perturbation modifier taking its input from procedurally generated Perlin noise. This modifier alters the geometry of an object, rather than perturbing surface normals.

3.4.17 Texture Mapping Modifiers

Texture mapping, first proposed by Blinn in 1976 [BLINN1] is a means to add apparent surface detail to an object without adding extra geometry. Texture mapping involves storing a set of texture coordinates which are typically two dimensional point coordinates, commonly referred to as UV coordinates, the space in which they exist being termed UV space. In the system described in this thesis, it is possible for every half edge in an object to refer to a unique UV coordinate or alternatively, every edge could share the same UV coordinate. The list of UV coordinates is not tied to the list of vertices. Were this the case then problems would arise when trying to apply, for example, texture coordinates to faces of a cube (see Figure 60). Unless each vertex has its own texture coordinate, it is impossible to

map each face correctly. The same applies to per-vertex colours and weights; each half-edge has as many references to vertex level elements as there are vertex level elements (it is possible to have an arbitrary number of texture coordinates colour and weight lists for an object).

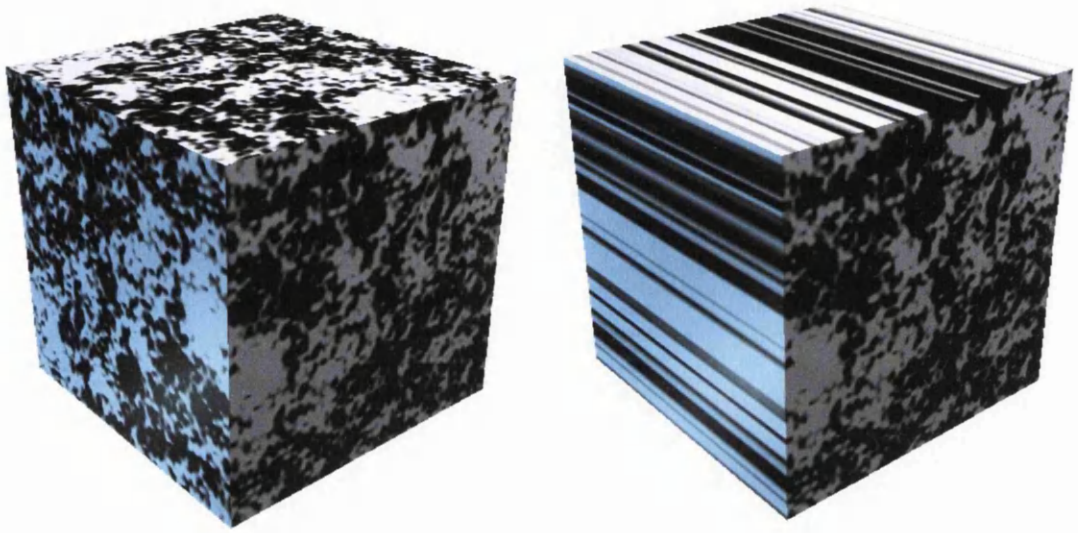


Figure 60. This image shows how using unique texture coordinates, rather than sharing texture coordinates at vertices, affects the final image. The cube on the right shares vertex coordinates, whereas the image on the left has 23 unique texture coordinates.

Multiple sets of texture coordinates are often required if multi-texturing is used. Multi-texturing is a term used to describe the use of multiple textures on a single surface. Most modern hardware supports multi-texturing in some form in a single rendering pass (the latest NVIDIA graphics card at the time of writing, the GeForce FX 5900, supports up to 16 textures in a single pass [NVIDIA]) but when the output library or hardware fails to support many textures in a single pass multi-pass techniques can be used to provide this functionality. At each stage of the multi-texturing process some blending operations are required. It does not make much sense to render a given texture and then overwrite this data with another texture, completely obscuring the texture beneath. Common blends include modulation, where the component colours of each texture are multiplied together and others such as subtractive blending, additive blending and other operations.

Simply applying texture coordinates to an object is not enough to see any visual change. The rendering of an object is controlled by a material node which dictates the manner in

which an object is displayed. Material nodes are responsible for rendering triangle strips, vertex lists and other geometric entities. A material node may or may not use textures, or take advantage of multi-texturing. If the user is to see the effect of applying a texture coordinate modifier node in real-time, the material used by the object must make use of the texture coordinate channel being edited.

The system contains a range of modifiers used to apply texture coordinates to objects. These are mostly based on some intermediate shape which is then used to 'shrink wrap' the texture coordinates onto the underlying mesh.

Texture mapping and shrink wrapping from base forms onto arbitrary topology causes a problem with seams in the texture. This can be illustrated with spherical texture mapping where the seam occurs as the vectors used to generate the UV coordinates rotate fully around the axis of the sphere at the point that is both 360^0 and 0^0 (see Figure 61) Extra texture coordinates need to be generated on this seam to ensure that it is rendered correctly. Problems arise when the underlying geometry does not match the seam; for example, if the seam bisects the centre of a surface, rather than at the edges where the texture coordinates are stored. The solution currently employed by the system the closest match along the edges of the seam, without creating any new geometry. The 'ideal' solution would be to cut the mesh along the seam but this has the unfortunate side effect of necessitating the creation of extra vertices, edges and polygons. Future versions of the texture coordinate application modifiers will include the option to use either of these methods.

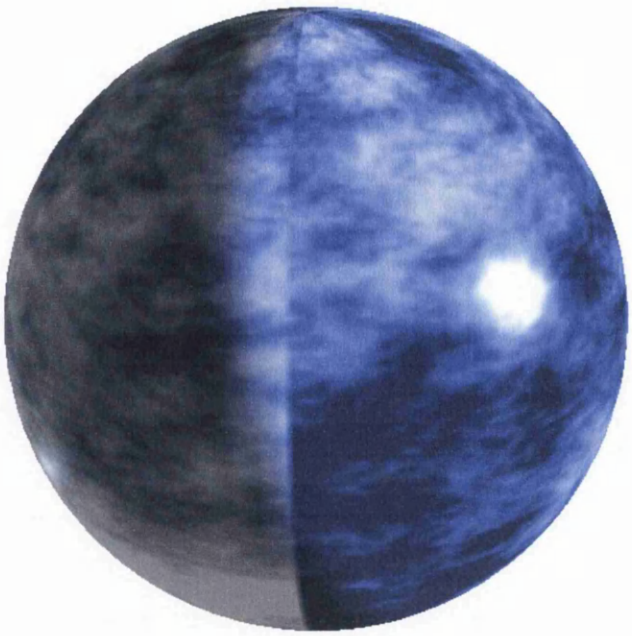


Figure 61. A textured sphere shows a band across one line of longitude; unique texture coordinates are created along this seam to avoid rendering artifacts associated with sharing texture coordinates.

Some forms of texture mapping do not make use of the texture coordinates supplied by the mesh data. Typically these are used when a material makes use of environment mapping, a form of texture mapping used to simulate reflections and other eye position dependent effects. Such texture coordinates are dynamically generated every frame. Typical functions used to determine dynamically generated texture coordinates include cube mapping, spherical environment mapping, cylindrical mapping and hemi-spherical mapping. These functions closely mirror those available in the system but they are distinct in that the underlying rendering libraries and hardware generates these coordinates and it is not usually possible to retrieve the coordinates after they have been generated. Also, rendering libraries have their own solutions to the 'seam' problem which vary from splitting up affected triangles and rendering them several times to avoid seams or performing per texel lookup operations when rasterizing the texture.

3.4.17.1 The UV Plane Assignment Modifier

This modifier falls into a family of modifiers used to assign texture coordinates to objects. The UV plane assignment modifier is the simplest of all of these modifiers; it projects UV coordinates orthographically onto the selected regions of an object from a rectangle orientated in 3D space. This rectangle can be rotated around its axis and can also

be scaled.

The tool used to apply the planar texture modifier (see Figure 62) is similar to the tool used by the stretch manipulator only the controls are limited to a single plane, rather than the stretchable cube used by the stretch modifier. The plane can be rotated by grabbing the cube at the end of the vector that protrudes from the centre of the plane. This intuitive control allows the plane to be rotated with the texture coordinates bellow responding in real-time to the new mapping.

The texture coordinates are generated by taking the vertex position at a given point and applying the matrix generated by the planar manipulator. The matrix involves a scale and offset on the x/y plane, rotated into an arbitrary plane by concatenating this matrix with a rotation matrix. It is also important to note that since this modifier relies on world space coordinates to function, it needs to refer to a copy of the world space matrix of the object as it was at the time the modifier was created.

The planar texture coordinate assignment modifier also contains functions to fit the projection to the geometry at a given angle of projection. This is achieved by projecting the coordinates of an object onto the x/y plane by using the rotation component of the modifier and calculating the maximum and minimum values on the x/y plane. These values are then used to set the scaling and offset properties of the modifier.

Figure 62. Planar texture application. The cyan rectangle is used to scale and position the texture on a selected object. The purple line and box is used to orient the plane in 3D space

3.4.17.2 **The UV Sphere Assignment Modifier**

The sphere texture coordinate assignment modifier maps spherical coordinates onto the surface of a selected object. The spherical coordinates are generated by taking a point on the object and then to project this onto the surface of the sphere used to apply the texture coordinate application. By forming a vector between the centre of this sphere and the vector in question (which may also be rotated in order to align the texture), we produce a vector that can be used to generate a U/V texture coordinate.

$$u = N_x / 2 + 0.5$$
$$v = N_y / 2 + 0.5$$

where u and v are the texture coordinates and N represents the normalized vector formed between the centre of the sphere and the vertex position of the vertex under consideration.

The UV sphere assignment modifier makes use of the same manipulator node that is the rotate manipulator utilizes. This manipulator node outputs a vector for the centre of rotation and a quaternion value representing the rotation and these values are fed into the sphere. Manipulator nodes are often used to control modifier nodes since the implementation of

these visual nodes is rather involved and it would be unnecessary to repeat this code inside many individual modifiers. Manipulator nodes also have the advantage of being able to control several modifiers at the same time; this is a common situation when working with multiple objects.

The sphere texture coordinate modifier, unlike the plane texture coordinate modifier, has to deal with the problems associated with texture seams. The current solution involves identifying the edges that most closely straddle this seam. This is achieved by classifying polygons around the area of the seam as on one side of the seam or the other. Edges between these polygons will be assigned unique UV coordinates, rather than sharing them as with coordinates elsewhere in the mesh. This means that there will be more texture coordinates than vertices on a spherically mapped mesh.

3.4.17.3 **The UV Cylinder Assignment Modifier**

The cylinder texture coordinate assignment modifier uses a cylinder as the intermediate mapping shape. Vertices on the mesh must be classified to see if they should be mapped using coordinates from the caps of the cylinder or the body. This classification stage is initially performed on a per polygon basis which analyses the polygons normal to see which side it 'mostly' faces or in other words, what the major axis of the polygons normal (rotated into the space of the cylindrical projection) is. Major axes can be thought of as the faces of a cube; top, bottom, left, right, front and back. If the major axis is 'top' then the polygon is mapped using a planar projection from above, if it is 'bottom' then the polygon is mapped using a planar projection from below. All other classifications are considered to be facing the body of the cylinder and texture UV coordinates are then generated by taking a vector between the vertex and the closest point on a vector down the centre of the cylinder (x=0,y=1,z=0 in the local space of the cylindrical projection). Seams are created along the edges which are shared by two polygons with different classifications.

The cylinder projection may be rotated, scaled and moved around in real time using intuitive controls shaped in the form of a cylinder with various handles for controlling these parameters. Again, the node that is visible in the editor is a separate node from the actual modifier and feeds its output to one or more cylinder UV.

3.4.17.4 **The UV Cube Assignment Modifier**

The cube texture coordinate modifier uses a similar classification technique as the cylinder assignment modifier only this makes use of all classification vectors. The

classification process caches polygon major axes in a list before traversing every selected edge, creating new texture coordinates as needed. As with all other texture coordinate application modifiers, a seam is created around the selection edge; the unselected texture coordinates are unaffected by the assignment. The cube texture coordinate assignment modifier is essentially six planar assignment modifiers in one, the major difference being that this modifier has to handle more seams and is therefore more processor intensive.

3.5 Common combinations of modifiers

Modifiers are usually used in combination with each other and to facilitate this operation, the system will always re-select areas of the mesh in a useful manner once completing an operation. Many of the basic modifier operations take on new functions when used in combination with other modifiers. Listed below are some common operations that can be performed by using several modifiers in succession.

3.5.1 Twist

Twists can be performed by combining either selection with falloff or any of the extrude modifiers with the rotate modifier (see Figure 63). The vertex fall off allows the rotate modifier to apply rotation based on the scaling factor included in the vertex selection fall-off.

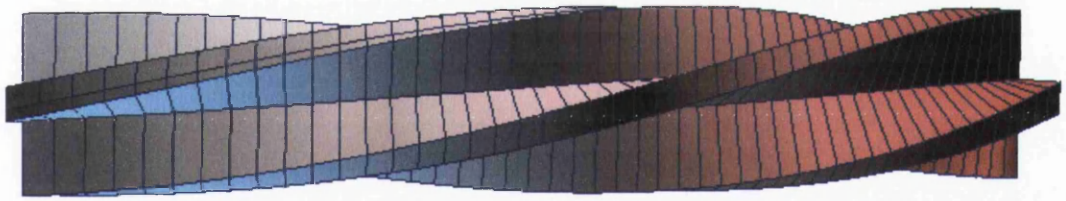


Figure 63. This image illustrates how extrusion or selection fall-off and use of the rotate modifier can be used a flexible twisting operation.

The red areas shown in the image exhibit the most rotation whilst the grey area remains unchanged.

3.5.2 Taper

To taper an object, the stretch or scale modifiers can be used, along with selection falloff, as shown in Figure 64.

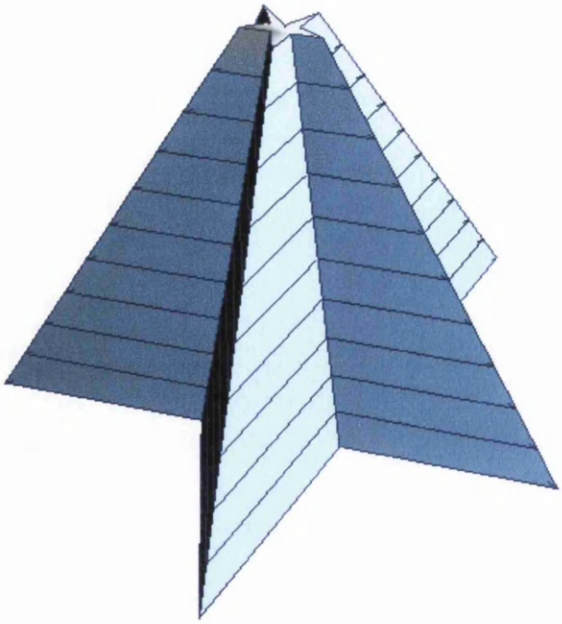


Figure 64. This star shape has been extruded and tapered using the stretch modifier.

3.5.3 Making windows

By using the per-polygon extrude function followed by the delete polygons operator, it is possible to create gaps in the model as shown in Figure 65.

Figure 65. This torus has had the per-polygon extrusion modifier applied, followed by a delete polygon modifier.

3.5.4 Turbine fans

By utilizing the per-polygon extrude modifier followed by a stretch and rotate, it's possible to quickly create turbine fans; the number of blades can be increased by increasing the resolution of the base cylinder and the resolution of the blades themselves can also be altered at any time (see Figure 66).

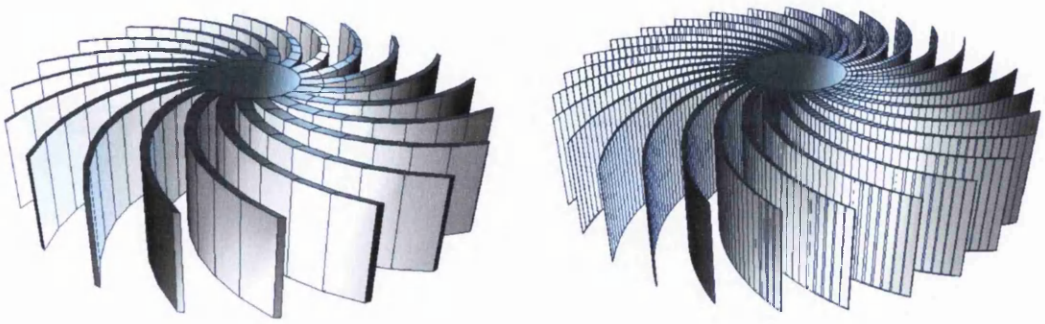


Figure 66. Turbine blades, created using the per-polygon extrusion modifier followed by a rotation modifier. The selection fall-off created by the per-polygon extrusion modifier is used by the rotation modifier to create this twisting effect. In the image on the right, the resolution of the original shape (a cylinder) has been increased, resulting in more turbine blades in the final object.

3.5.5 Procedural planets

Using procedural Perlin noise, spheres and the texture perturbation modifier, creating procedural planetoids is a very simple operation. The surface of the planet and cloud layer can be modified in real time (see Figure 67 and Figure 68).

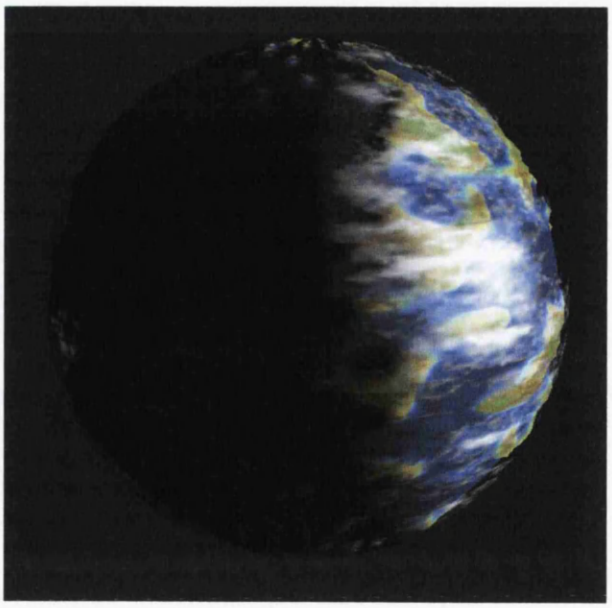


Figure 67. This procedural planet makes use of two Perlin noise textures (one for the continents, one for the clouds) and also

perturbs the surface of the sphere using the luminance values from the landscape Perlin noise texture to provide height values.

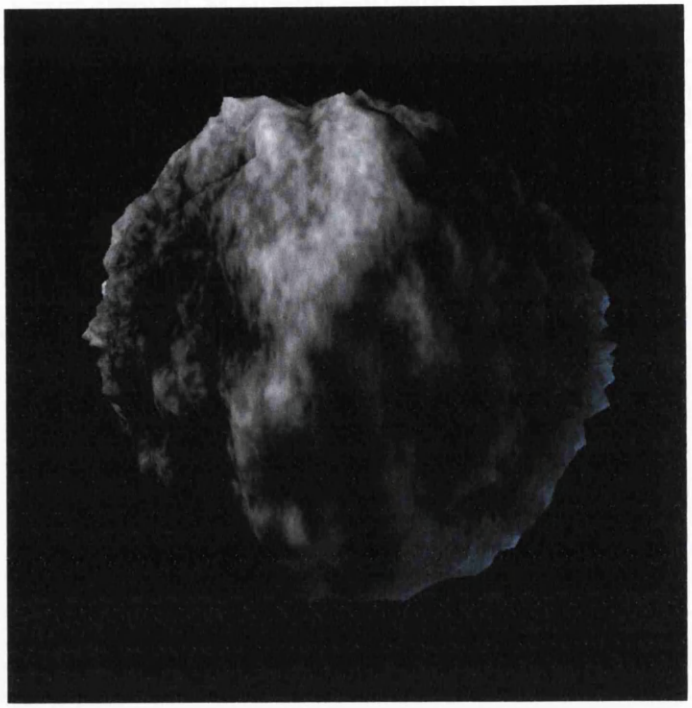


Figure 68. Procedural asteroids are similarly simple to create

3.6 Auxiliary nodes

Auxiliary nodes include those that are not directly part of the geometric data flow. They often augment this data, e.g, in the case of material, texture and animation nodes, but are not considered to be directly part of the process of creating 3D geometry.

3.6.1 Material & texture nodes

In order to display an object, it is necessary to have a material node to describe surface properties. Material nodes are fed into material application modifiers which then affect the per-polygon material channel of a mesh data stream. When this list reaches the node that actually renders the object, they are sorted and assigned a list of triangle strips, wire-frame edges and vertices which can then be used when rendering. All of the functions that actually render data are controlled by the material node which can interpret vertex, edge and polygon data as it sees fit. A 'toon shader' would only render edges without twins or those that straddle forward and back-facing polygons whereas other materials might ignore the wire-frame data altogether.

It is inside material nodes that the most hardware/library specific code exists. Whilst

great care has been taken to avoid dependence on a particular library or hardware configuration in the implementation of the system, material nodes are the most volatile in this respect. In order to take advantage of the latest hardware, material nodes can implement several different means of rendering a desired effect dependent on the best solution for a given hardware/software configuration.

Each material node must also inform the scene of certain capabilities and effects; If a material is translucent, it must indicate this fact to the scene so that translucent sections can be handled effectively by the rendering engine. Likewise, the material must indicate to the scene that it refracts light or reflects objects in the scene. Materials classes must also implement a validation function to check to see if the currently installed rendering hardware and/or software is capable of rendering the material. With such information, the scene graph render engine can make informed decisions regarding the order in which objects within the scene are rendered.

Materials can be implemented however the developer of the material sees fit. They can be as complex or as simple as required (see Figure 69), can aim for compatibility or be exclusive for specific features present in certain video cards. Due to the fast moving nature of real-time 3D graphics hardware, this kind of open ended flexibility regarding materials is important. Tying the whole application to a particular architecture by hard coding specific rendering styles within the system would impede future development. The specification of materials as distinct nodes effectively quarantines them from the rest of the application, resulting in a layer of 'future proofing' for the application as a whole. Most nodes implemented in the system make no calls to the underlying rendering libraries at all; only a very few, specific nodes convert data into a format that is efficient or effective for a particular hardware/software configuration.

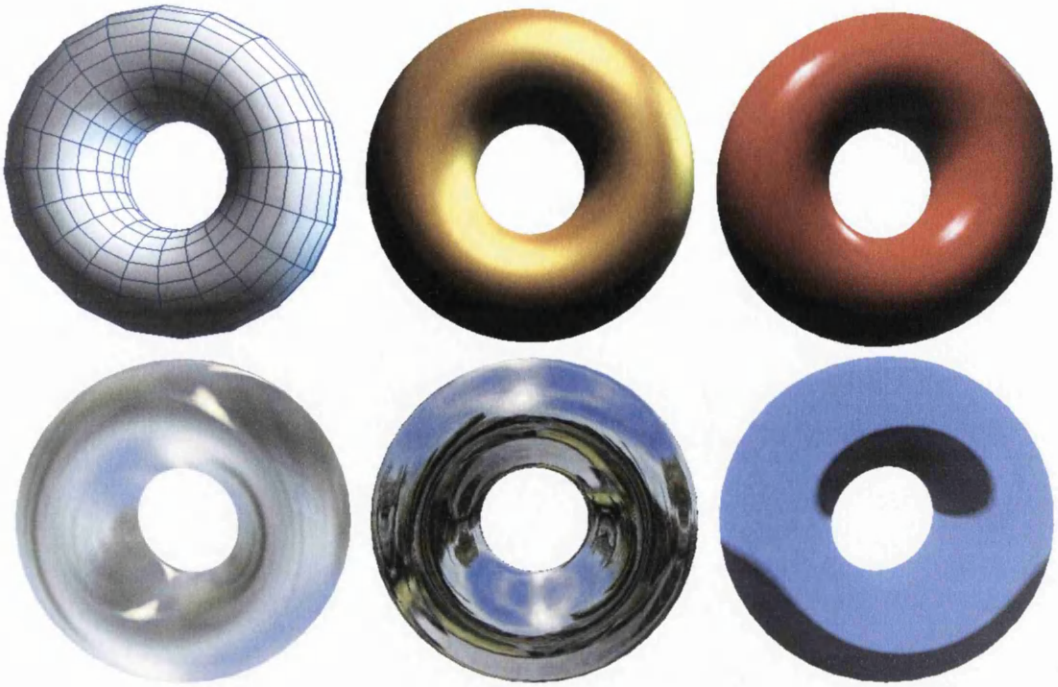


Figure 69. This image shows a number of different materials applied to a base torus primitive. From top left to bottom right they show outlined, metallic, plastic, diffuse and specular environment mapping, 100% specular environment mapping and 'toon' shading.

Texture nodes should not be confused with materials. Textures are used by materials but are themselves distinct nodes which feed data down a node chain in an identical manner to the geometry generation and modification nodes. In keeping with the procedural philosophy of the system, textures can be procedurally generated and altered in the same manner as geometry and a range of nodes is dedicated to the creation and manipulation of 2D data.

Textures can be considered as two dimensional grids of data *or* a function of the form $f(x, y) = c$

where c is a colour value (as in the Perlin noise example in Figure 70). The actual type of this colour value depends on the kind of texture; it could be RGB triple, a floating point value or a single bit. Textures must also implement functions used to render a given rectangle into a cache; If the texture node is actually a stored bitmap cache (similar to the 'data cache' geometry generator type) then this operation is a simple copy. Procedurally generated textures on the other hand, render their procedural data into this cache in a resolution independent manner. The inclusion of both a function to query the colour value at a given

coordinate and the ability to render the contents of the node to an image buffer might seem redundant but due to efficiency constraints, it was deemed appropriate to include a means of rendering a procedural texture directly, rather than sampling points at given coordinates in order to render an image.

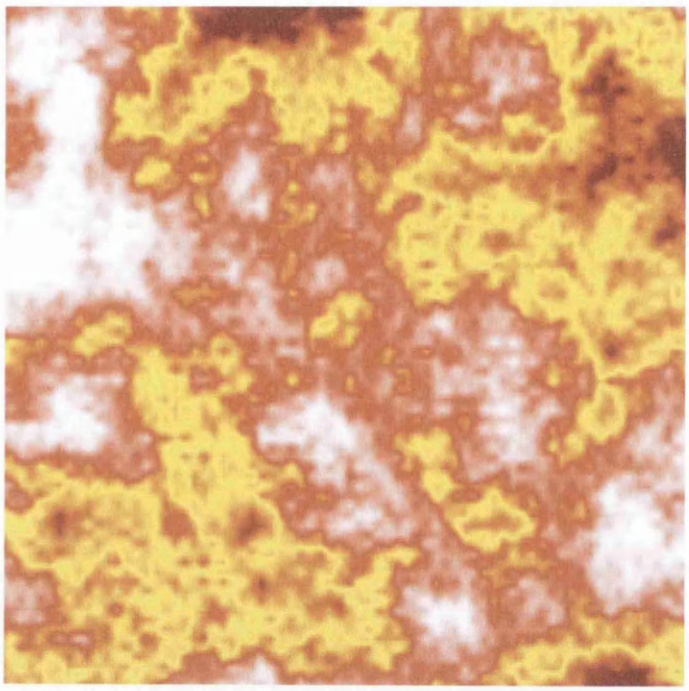


Figure 70. This image shows an example of 'Perlin Noise', as generated by the system.

Chapter 4

IMPLEMENTATION AND EFFICIENCY ISSUES

Whilst the main focus of this thesis is the procedural generation and modification of objects, the framework within which this is achieved is closely intertwined with the method used to achieve this goal. The term used to describe this framework is the *scenegraph* and the method used to achieve parametric object creation is also used in the management of all kinds of data within the system.

A problem faced by all simulation and modelling software is the structure of the scenegraph. It can be argued that the scenegraph *is* the modelling program, or at least the core of it. A poorly designed or implemented scenegraph can seriously hamper further development and extension of a system. Ideally, the scenegraph should be easily extensible and expose the inner workings of the program to 3rd party developers in a clean and orderly fashion. Efficiency is also paramount as this is a real-time application so the system must be able to take advantage of algorithms and data structures that facilitate rapid rendering of large data sets but the system must do this without restricting the ability to edit objects.

A scenegraph will often separate the creation phase of objects from the scenegraph. Some programs (such as Lightwave, [LITW]) even go as far as having a separate program for creating objects and another to arrange the objects in a scene. There is no hard line that can be drawn between editing a single object and editing a scene full of objects and if software is separated down some arbitrary line, duplication of effort will occur for both the programmer of the software and the graphics designer using it.

The scenegraph is implemented with the needs of a 3D modelling program in mind. However, due to the flexible structure of the nodes in the scenegraph, it is also possible to tailor the scenegraph for use in other specialized areas without breaking the connectivity of the graph. Put another way, there are several ways of indexing and referencing nodes within a graph and different applications would create these auxiliary structures independently of the core data relationships in the scenegraph.

4.1 User Interface

4.1.1 Introduction

The user interface used in this application, known as TWIN (Tim's Windows, see Figure 75), is a cross-platform system developed by the author for use in this application and others. It has been compiled and tested on Windows (using DirectX), the PlayStation 2 (a version of which is part of the Xeios [XEIOS1] library for that system) and a legacy version in DOS, using VESA display drivers (Figure 90).

The windowing system has evolved over the years and has provided a foundation for many of the concepts that are used inside the 3D scenegraph. The eventual aim is to integrate the 2D and 3D interface elements into one combined system to aid flexibility and functionality whilst removing duplication of effort. The move to integrate the 2D and 3D interface elements also serves to justify the existence of a custom cross platform interface toolkit; No 2D/3D interface toolkit of the required standard is available at this current time and it would be highly inefficient to attempt to integrate a foreign system into the scenegraph.

4.1.2 Controls within the system

There are three major facets to the user interface which are described here in detail; the **node panels** (see Figure 71), the **3D views** and the **schematic windows** (see Figure 72 and Figure 73).

In the system described, each node created, be it a texture, material, 3D object or any other type, has a 'node panel'. This is a dialogue control that is unique to the node in question and displays editable parameters and possibly some graphical display of the object (such as a 2D view of the texture, a rendered example of a material, a display of colours in a palette or the wave form of a curve or sound wave). Often there are tools available inside these node panels for connecting certain inputs (assigning textures to materials is controlled through a drop down list showing thumbnail views of compatible nodes) and for editing values such as colour inside that node. If these values are actually controlled by an input to this node, such editing tools are 'ghosted', i.e, they appear in a particular visual style that indicates that they cannot be used.

As well as having a unique node panel, some nodes are also visible and editable in the 3D views of the scene. Certain nodes (such as 3D objects) have several modes of display,

depending on the type of view. A view might display the scene as 'intended', that is, with full lighting effects, shadows, and materials or in a manner more conducive to editing. The latter form of view will often expose 3D *gizmos* that can be grabbed and moved around in the 3D view to provide intuitive editing and feedback of 3D shapes. Modifier nodes usually expose some sort of 3D gizmo when selected and shown in an editing window; in a full 3D render these would detract from the display.

A third section of the user interface shows various ordered lists of data. These are collectively termed 'schematic' views and exist in a tabbed panel in the application. The **tree** tab (Figure 72) shows a tree structure of the whole scene, ordered as 'visible nodes' which has various sub headings such as 'lights', 'cameras', 'objects' and special system nodes, 'materials' which shows a list of all available materials, 'palettes' which shows all palettes, 'textures' which shows all the textures nodes and 'miscellaneous' which refers to all nodes not in any other list. The order of these nodes is simply for bookkeeping/ reordering them or placing nodes in different sub folders has no direct impact on node connectivity. The **Graph** tab shows actual node data connectivity and also allows data inputs and outputs in the scenegraph to be connected. This view and the nature of these data connections (see section 4.2.1) are described in greater detail later as it is complex and central to the internal working of the application. This view can be configured to show the whole graph or only a subsection. The **Materials, Textures** and **Palettes** tabs (see Figure 73 and Figure 74) show thumbnail views of all these resources. The main purpose of splitting data into these separate schematic panels is purely ease of use. Third party extensions to the software can also add their own views on the data of the scenegraph to facilitate other forms of viewing, which can be added as extra tabs in the schematic window.

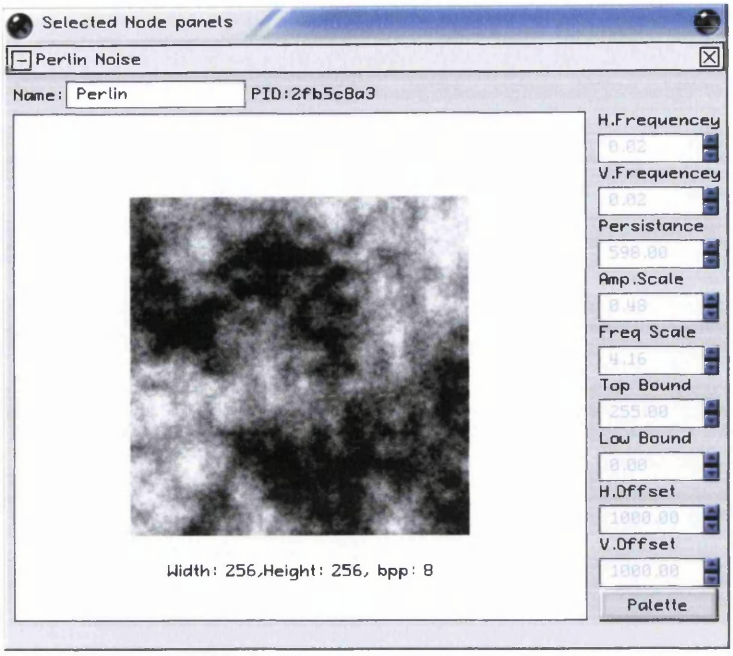


Figure 71. The Perlin Noise editor panel.

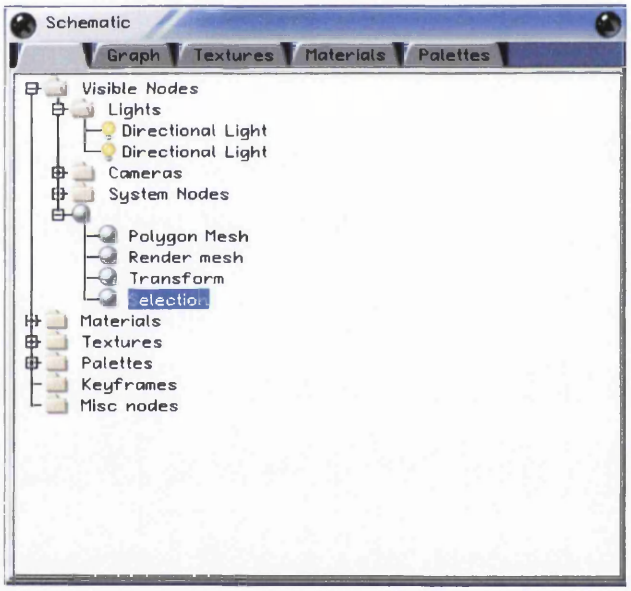


Figure 72. This image shows a tree that represents the scene. Items can be removed and renamed either by using the keyboard or via a context menu. Clicking on an item opens it's editing panel.

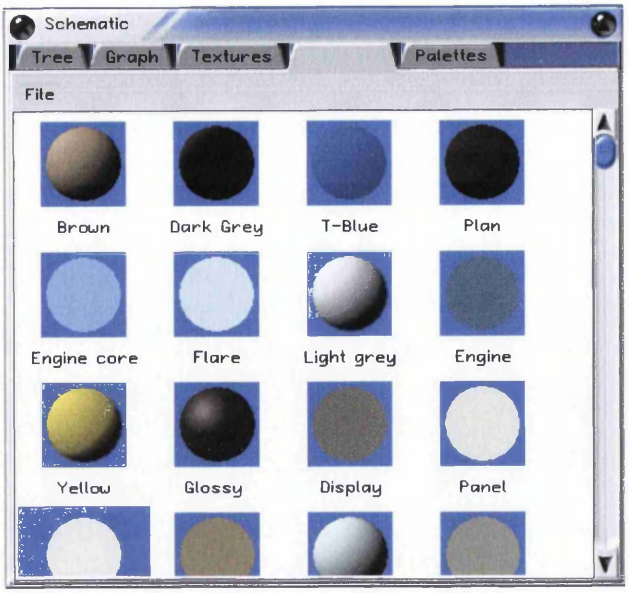


Figure 73 The Material Thumbnail View. This specialized view is able to render thumbnail views of materials. Just as in the tree view, items can be removed, deleted and their editing panels can be accessed by clicking on a material node.

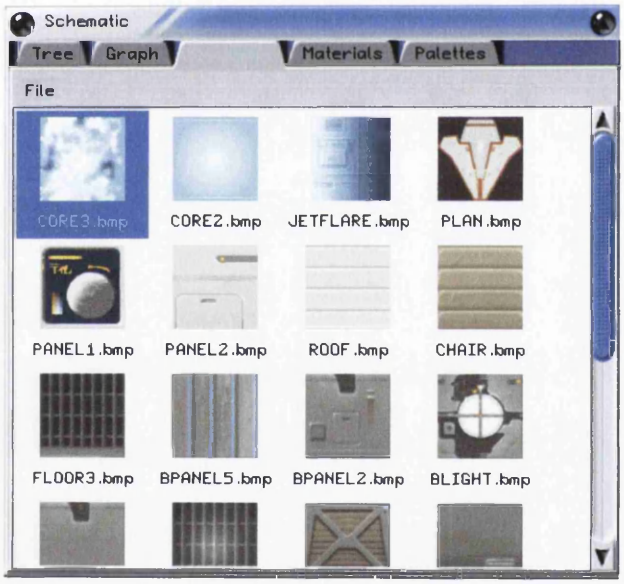


Figure 74. The Texture Thumbnail View. Much like the material view, only this window shows thumbnail views of texture nodes.

4.1.3 Internal Structure and window methods

The user interface is formed of a collection of windows (also known as views or widgets) nested inside container windows. The windowing system has a root window which serves as the main container for all sub windows and widgets. Child windows are always clipped to their parents shape. Windows can be translucent and of any shape although opaque, rectangular windows are more efficient to manage and render using the current

system. Unusually for a windowing system, all the windows work in a global coordinate system; when a parent window is moved, the position of child windows are also updated. The next generation of this windowing system will use more generic local coordinate systems which will include support for matrix based distortions to local coordinate systems.

Each widget has several virtual methods which define the look and behaviour of the widget. Firstly, a window must know how to respond to an event. This is handled by the *handleevent(tevent *event)* method. This method is overridden by most widgets and defines how the widget will respond to keyboard, mouse and command events. Each window must also know how to draw itself and this is handled by the virtual method *'draw()'*. The draw method always draws the current window according to its internal state. Clipping of a window is handled externally by the windowing system but many widgets make use of the current clip state of the rendering engine to perform gross culling of items. The rendering of child windows, which may be of any shape or opacity and in any position within a parent window, is handled by the windowing system, rather than by the draw method of a given window. The draw method is never called directly, if the program wishes to draw an area of a window then it must call one of the TWIN global invalidation methods, such as *wm_invalidate(tview *view, trectlist *rects)* or *wm_invalidate(tview *view, trect *rect)*. The invalidation methods inform the windowing system that a given rectangular area (or list of rectangles) of a window are now invalid and need to be updated. The windowing system collates all of the changes for a given frame and determines the best way to render the invalidated windows. The existence of opaque and non-rectangular windows complicates this task but efficient dirty-rectangle and selective buffering scheme employed by the windowing system performs smoothly. The windowing system keeps a list of rectangles for each window that need to be updated. If a window above the window being updated has a translucent section, or is irregularly shaped, the windowing system redraws that window also. To avoid flicker, this compositing operation is performed in an off-screen buffer before being copied to the display device.

The other important virtual method is the *'changebounds(trect *rect)'* method. This method is called whenever the window changes position or size and is overridden in order to perform custom control over the placement of child windows and the re-arrangement of graphical elements internal to a given window. The default *changebounds* method provides a generic means of controlling child placement by the use of anchor codes. Child windows have a set of bit fields which indicate how each side of their bounding rectangle should conform to their parents bounding rectangle and/or user defined anchor points within the

parent window. These codes are used to dynamically resize windows using the default *changebounds* method. For most cases, the default *changebounds* method suffices but for some cases require this method to be overridden and more complex code based resizing of child windows is performed (one example is with rich-text windows which require child windows to wrap around text).

4.1.4 **Event management**

The user interface is event driven with incoming events translated from the host platform, such as windows, or polled more directly using device specific drivers, as on the PlayStation 2 and DOS versions.

Events take the form of mouse events, keyboard events and command events. Mouse events are created and dispatched to the top level window which then checks if its child windows contain the mouse cursor position. If so, the event is passed on to the child windows until a terminal node containing the mouse cursor is found. If no child windows contain the mouse cursor then the parent window handles the event itself. Mouse events include the following:

Table 4. Mouse events.

Mouse Down	This event indicates that a mouse button has been pressed when the cursor was over the view that received the event. A bit field indicating the state of the mouse buttons is included with the event.
Mouse Up	This event indicates that a mouse button was released when the cursor was over the view that received the event. A bit field indicating the state of the mouse buttons is included with the event.
Mouse Drag	When the user holds down the mouse button and moves the mouse, the drag event is sent to the window below the cursor

Mouse Linger	The mouse linger event is fired when the user doesn't move the mouse for a given number of milliseconds. Some windows respond to this event by creating a 'help bubble' popup window.
Mouse Over	The mouse over event is fired when the user moves the mouse cursor into a window
Mouse Out	The mouse out event is fired when the user moves the mouse out of a window that previously had the mouse cursor over it.

Keyboard events are sent to the currently focused window. The concept of 'focus' is common in user interfaces. A window is often considered focused if the user clicks on it or moves the mouse cursor over it. Both behaviours are supported by TWIN which those that focus themselves with a mouse over being termed 'auto-focusing windows'. Windows usually respond visually to indicate that they have received the current focus. This includes blinking cursors in text boxes, altering the shade or hue of a title bar in the case of windows and other visual cues that indicate that a widget is ready to receive input.

Table 5. Keyboard events.

Key down	This event indicates that a key has been pressed. The key-code is given with the event.
Key Up	This event indicates that a key has been released. The key-code is given with the event.
Key Repeat.	This event indicates that a key is being held down. This event is fired at set intervals for the duration of the key being pressed.

	The key-code is given with the event.

Command events can be sent from any window to any window but typically there are set paths through which such events are passed. Currently, each window has a default command target window pointer. This is the default window to which commands are passed when using the simplest version of the *'sendcommand(UINT32 command)'* method. It is however possible to override this system and send to any other specified window and usually such command event targets include parent or child windows. Command events can include a number of auxiliary data items and the most common form of command event include updating a value from a specific window class such as a slider bar, number box, text edit box and so on.

The data flow model used in the 3D scenegraph would be a far more flexible and well defined model for such interactions. The next iteration of the windowing system, integrated with the 3D scenegraph, will allow a schematic view of an entire application; showing how data flows from different nodes (both visible and invisible; it should be noted that all objects in the windowing system are visible objects, there is no clean support for objects that simply process data but are not visible in the application as there is in the 3D scenegraph) and provide a means to view the overall structure of an application in a visible manner. This future windowing system will provide a valuable tool for creating, debugging and simplifying applications as well as being serializable using the methods currently employed by the system.

The windowing system has an extensive library of in build 'widgets'; specialized windows that perform a specific task. Figure 75 shows a selection of these nodes in the environment of the application. Some of these widgets are container windows, designed to manage sub-windows in a certain way. Roll-down views and tab boxes are examples of these. Other widgets are used to alter a scalar value over a range (scrollbars and number spinner boxes (not shown) are examples of this) whilst others can edit or view text (single line text edit boxes and rich text views are examples of this). Some windows are specific to a given application; the 3D editing window, texture selector and texture viewer are examples of windows designed specifically for the application featured in this thesis.

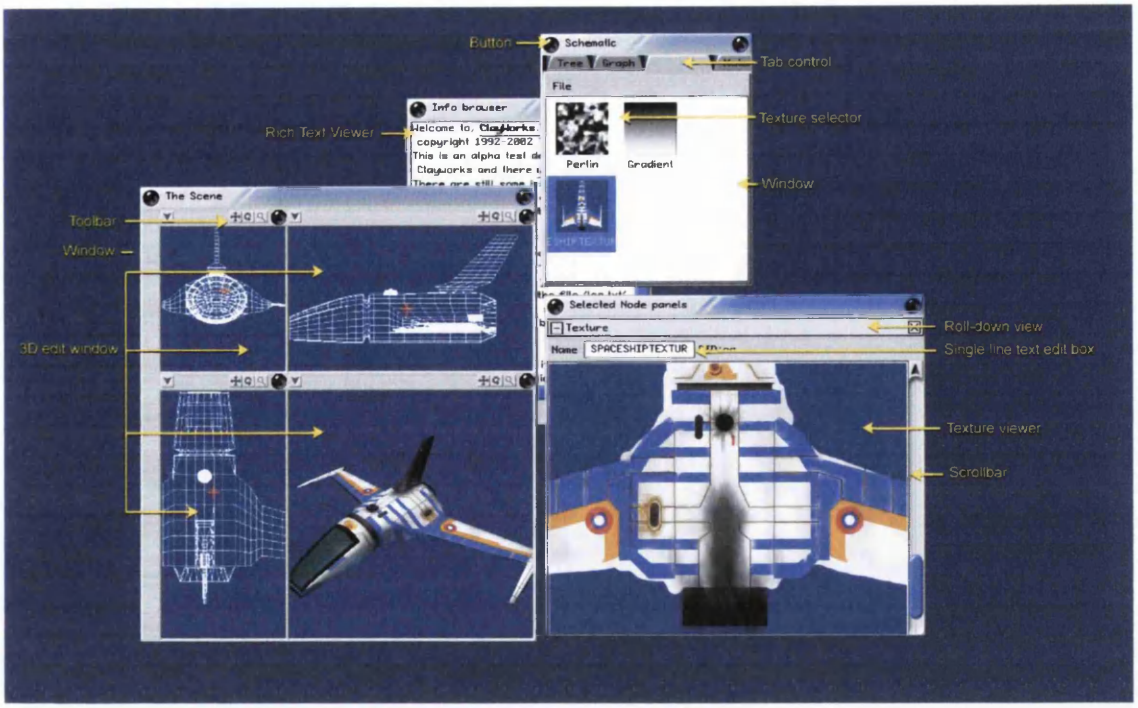


Figure 75. Widgets of the TWIN windowing system.

4.2 Data structures

4.2.1 The Data connectivity graph

The core scenegraph structure is a list of nodes that have various data inputs and outputs, forming a Directed Acyclic Graph or DAG (see Figure 76).

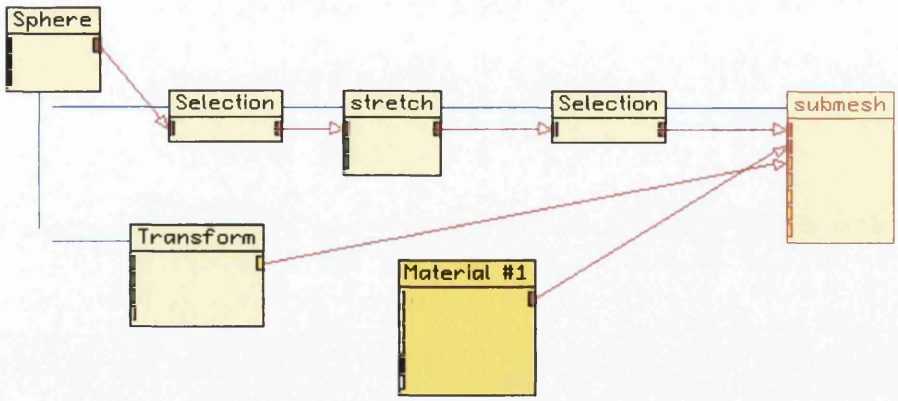


Figure 76. A series of connected nodes as viewed in the node viewer window.

4.3 Data types within the graph

The inputs and outputs can currently have any of the following types:

Table 6. Scenegraph base datatypes.

• **DTvoid**	Void, has no type
• **DTint**	Scalar Integer
• **DTfloat**	Scalar float
• **DTvector**	3 dimensional vector
• **DTpoint**	2 dimensional vector
• **DTimage**	Generic 2D resource
• **DTmatrix**	4x4 matrix
• **DTmaterial**	Generic material resource
• **DTtexture**	Generic texture resource
• **DTcolour**	4 dimensional colour (r,g,b,a)
• **DTpalette**	Generic palette resource
• **DTpolygongeometry**	Generic geometry resource

These types vary greatly in complexity from single scalar values to complex classes that contain many internal data items and methods. This should not be considered as a complete list and new data types can be added dynamically by third party extensions.

The data connectivity between nodes is defined as lists of inputs and outputs within each node and the functionality for this are defined by pure virtual functions inside the root ancestor of all scenegraph nodes, known as **sg_basenode**.

These virtual functions provide methods that are implemented in inherited classes for retrieving the number of inputs and outputs for that node and the type of the input or output and also if a particular input can accept multiple inputs. The scenegraph itself controls connectivity between nodes using a function called '*makelink(...)*' which takes an input node and an output node as input, as well as the two data input/output channels that need to be connected.

All outputs can reference multiple nodes but not all nodes can accept multiple inputs.

4.3.1 Optimization and rendering

When a node is altered in some fashion that will change an output, it must indicate to

the scenegraph that this output has been altered and that the node needs to be updated. This change is then propagated down the data list until there are no more changes, working out the path that this data takes as it does so. Depending on the implementation of the scenegraph, this list of dirty nodes is then 'cleaned' by traversing the path of the altered data and informing each node in the chain that it has received a new input. In the process of updating the graph, a view on the data may need to be changed. Certain nodes are flagged as 'render' nodes that appear in the 3D view of the scene, such as polygonal objects, lights, cameras and other potentially visible elements. These are usually leaf nodes and have standard methods for drawing themselves. The order and manner in which they are drawn often depends on some criteria set by the application. In the case of the system in question, schematic wire frame views may need to be rendered, as well as a fully shaded render or some other, specialized view on the scene. In the case of the former, the style and method in which the objects are rendered will depend on what is required; a wire frame view needs not draw the object fully textured and a 3D textured presentation view would not need to render extraneous 'gizmos' that are required for editing the shapes (such as handles for moving vertices around or rotating a selection, etc).

In addition to the visible 3D elements, the interface itself must be updated when changes are made to the dataset all of this is application dependent and the manner in which an immersive game environment would 'clean' the scenegraph would be different from the method used in the 3D modelling system.

However, in both systems, when an individual node receives a new input (or altered input), it recalculates any internal data that must be changed, indicate to the system that it has been dirtied and that some view on this data must be changed and informs any downstream nodes connected to it that they now have new data.

This behaviour, core to the scenegraph, is encapsulated in a virtual routine named *'updatescene(...)'*. This is where the path of the data is analyzed and updated inside nodes, views are redrawn and user interface elements are updated. The path that changes take through the nodes can also be cached if that change is to be made repeatedly which allows for optimization when refreshing data on the screen. If, for example, a section of a sphere is selected and transformed in some way and this transformation is to be made interactively by the user, the scenegraph has a mechanism by which unaffected portions of the display are saved in an image buffer and only the sections that are to be changed are refreshed. In this

manner, a user can work on a sub-section of a very complicated form without extraneous calculations being performed, at interactive rates. A more convoluted example might involve a colour in a palette being altered using a slider, which then feeds into a palletized (or indexed) texture which, in turn, feeds into a material that referenced by a certain area of an object in the scene. The application realizes that this change is occurring and that only a certain section of the visible data is being affected (in this case, a group of polygons visible in the render window and any other possible views on any of the data items being updated) and renders the scene, minus the sections to be altered, into a buffer and then only renders the sections that are being changed whilst this interactive change is in progress. This becomes slightly more complicated when translucent, refractive or reflective materials are in play but the scenegraph is also aware of how these will be affected and acts accordingly. The system is also aware that some interactive changes are best updated in a staggered manner; for example, if a procedural texture is updated, the most important feedback for the user would be in the node panel view of the texture which shows the texture in 2D form. This display is updated in real time as a priority, whereas the thumbnail view of the texture and the visual changes in the 3D scene might only be updated when the user pauses input. The system can dynamically detect the feedback and check update times against a defined 'frames per second' benchmark to decide how to update the visual display and user interface or it can be forced to update periodically, or update the whole system in real time by setting certain preferences. To recap, the system will always update the most pertinent view of the data immediately and will either display downstream or auxiliary views immediately or when there is a pause in user input, depending on user settings and system performance. No user-editable feature ever shows out-of-date data as a result of staggered updates by the time the user is able to edit that data.

4.3.2 **Spatial Relationship Graphs**

Spatial relationship graphs are application defined hierarchical groupings of visible objects in 3D space. The nature of these graphs is largely dependant on the particular application and scenegraph implementation. The sole purpose of this kind of graph is rendering efficiency.

The form of spatial connectivity used in the modelling application is based on bounding volumes for visible objects and exists as a tree of objects and collection of objects inside certain bounding primitives such as axis aligned bounding boxes and ellipsoids. Certain complex visual nodes also exhibit internal spatial representations to speed up both rendering

and sub-object queries (such as collision detection). The form these complex internal structures take is largely dependent on the type of node and its intended use. A terrain object would have a scenegraph level bounding volume but internally, the node would use a specialized form of storing visual lists and means of dealing with collision from other objects. Typically, a terrain would use a bin-tree structure to store triangles to be dynamically rendered based on the angle and position of the camera. Other methods of storing terrain data, such as that used by the ROAM algorithm [DUCH] could also be utilized.

Static 3D objects that are infrequently edited pose certain problems; A BSP tree coupled with a potentially visible set is ideal for static internal environments but expensive to calculate. To get around this, there are several means of representing rendering data within a visual node dealing with complex geometry. The input to such a node is typically a polygon data stream. For simple (lower polygon) datasets, dynamically creating triangle strips and fans when the incoming data is changed is an acceptable form of storing the data for presentation. However, if this data is static for a period of time or is 'locked' by the user, the application is free to analyze the data more thoroughly to create a structure that is faster to render and query. Even if only a portion of the data is being edited, then the application can use a simpler method for storing the dynamic portion, leaving the static portion in a more efficient format. By utilizing *lazy processing* [KARC1] in a manner similar to the staggered updates in the user interface, the application can work on creating more efficient means of storing data while the user is either inactive or working on a less processor intensive task.

By mixing methods for storing 3D data, it is possible to view and edit large datasets of a virtual environment in real time in a manner that neither degrades performance to an unacceptable level by using inappropriate methods for 3D data storage and also to retain the flexibility that a modelling program requires.

4.3.3 Logical/Functional Connectivity Graphs

Aside from data connectivity and spatial connectivity, the system makes use of a number of methods for referring to nodes in a logical manner sorted into various structures that make referring to nodes of a certain type or logical grouping easier. As with spatial graphs, this is an application feature and largely dependent on what the application requires. In the case of the 3D modelling system, there are many ways to refer to objects in the graph. Textures, Materials, Audio resources, Palettes, Lights, Cameras and other types of nodes with a certain logical grouping are stored in linear lists which make searches and visual user

interface display of these groupings efficient.

Other, user interface controlled, logical groupings include layers (collections of visual nodes which can be toggled for visibility) and organization of objects into user defined folders, visible in the 'tree' schematic view for ease of access. The user defined logical groupings have no functional effect on the workings of the scenegraph, this means of reference is provided to ease the management of complex scenes.

4.3.4 **Procedural object deformation implementation**

The core of the modelling tools in the system revolve around a subset of nodes used for creation, selection, modification, feedback and display of geometric data. These nodes form a chain along which geometric information flows, from creation of the base shape, through modifiers that alter its shape and attributes through to a node which is capable of converting this information into a form amenable to rendering. The final node in the chain is responsible for rendering and handling feedback from the user.

4.3.5 **Internal representation of polygonal geometry.**

The data that passes along this chain is of the DTpolygongeometry type that was mentioned briefly above. Internally, this data type is represented as a class that contains various 'channels' of data. Each channel contains one or more arrays of data that define geometric data, topological connectivity, texture coordinates, per vertex weights, per vertex colours and also selection information. The following table shows a list of the channels that exist within the DTpolygongeometry data type

Table 7. Channels within the DTpolygongeometry data type.

Vertex List	A list of one or more vertices, used to store position information.
Edge List	A list of half-edges. Each half-edge refers to a vertex, its parent polygon, a possible twinned edge winding in the opposite direction and the next half-edge used to form a polygon. Each half-edge also refers to any number of colour, texture

	coordinate and weight values.
Polygon List	A list of references to half-edges that form the start of a polygon loop.
Texture coordinates	Zero or more lists, each containing one or more texture coordinates. For use when rendering materials that employ texture maps.
Weights	Zero or more lists, each containing one or more weight values. These are used when 'skinning' is employed for smooth mesh animation and deformation.
Colours	Zero or more lists, each containing one or more colours. For use when rendering materials that employ per-vertex colours
Smooth groups	A list of integer values containing either one entry or the same amount as the number of polygons. Smooth groups are used for generating per vertex normals. Visibly sharp edges occur when two polygons have different smooth groups.
Material references	A list of references to materials containing either one entry or the same amount as the number of polygons. Material references refer to material nodes and this channel either specifies one material for the entire mesh or per-polygon material references.
Selection Channel	A class that contains bit arrays that define per-element selection.

	define per-element selection.

4.3.6 Half-edge data structure (HEDS)

Whilst each channel type is important, it is the half-edge data structure that defines how polygons and vertex level elements interact. The HE structure was chosen as the fundamental topological data structure due to its simplicity and the high level of connectivity information it contains. As can be seen in Figure 77 the HE data structure contains a reference to its *twin*, the next HE forming a polygon loop, a reference to the polygon that owns the HE and references to any relevant vertex information.

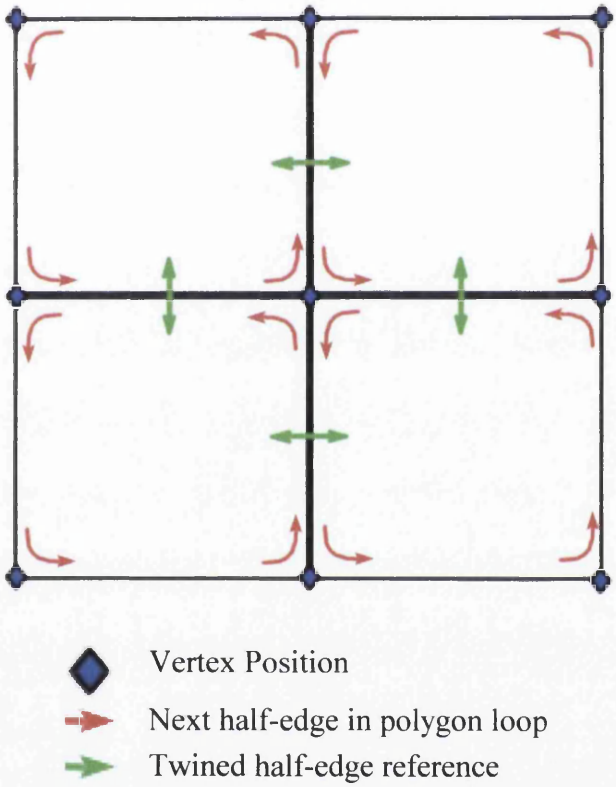


Figure 77. An Abstraction of the Half-Edge data structure.

Each half edge contains references to vertex level components and will always contain a reference to a vertex. Other vertex-level component references are optional and the number of references contained within each half-edge depends on what vertex level channels the mesh contains. If the mesh contains two texture coordinate channels, four weight channels and one colour channel, each half-edge will have seven additional vertex level references aside from the references to the vertex positions themselves.

The list of 'polygons' is really just a list of references to HEs that start a polygon loop. The HEs themselves contain the necessary information to form the polygons.

The advantage of the HEDS is in its brevity and connectivity information. Coupled with vertex references to HEs, it is possible to determine connectivity information from any element, be it vertex, edge or polygon, to any other.

The major disadvantage of the HEDS is that an edge can only have one twin so objects such as the one shown in Figure 78 cannot be represented without duplication. Whilst this is perfectly fine for true solid objects, it does restrict the type of topology that can be created within the system.

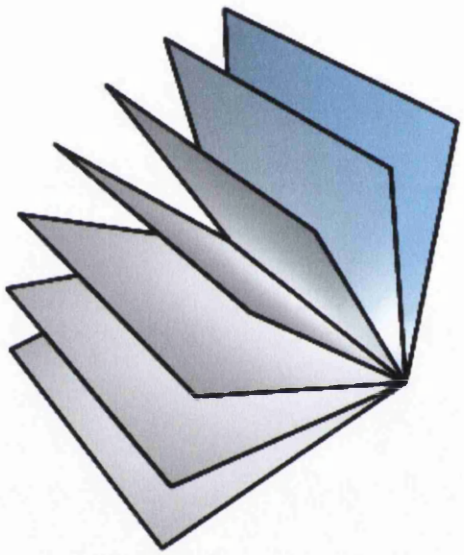


Figure 78. An example of an illegal object using HEDS.

Care must be taken to make sure that incompatible topologies are not created or problems will arise when attempting to query the relationships of edges, vertices and polygons to one another. The method employed by the system to guard against such unwanted topologies is to duplicate vertices if the system attempts to create a polygon containing two vertices that already form a half-edge that has a twin.

4.3.7 The Selection Channel

The selection channel is itself a collection of further lists. This channel stores the selection status of individual elements in the object. The selection lists are stored as *bitarrays*, lists of Boolean values where each value takes up a single bit. The structure of the selection channel follows:

Table 8. Bitarrays within the Selection Channel.

Vertex Selection	Bitarray corresponding to the vertex list. A vertex is considered selected if the bit that represents it is on.
Edge Selection	Bitarray corresponding to the edge list. An edge is considered selected if the bit that represents it is on
Polygon Selection	Bitarray corresponding to the polygon list. A polygon is considered selected if the bit that represents it is on
Affected Vertices	This list refers to all the vertices that are affected by the selection but not necessarily selected themselves.
Affected Edges	This list refers to all the edges that are affected by the selection but not necessarily selected themselves.
Affected Polygons	This list refers to all the polygons that are affected by the selection but not necessarily selected themselves.
Selection Edge	This list refers to the edges in the mesh that are on the boundary between selected elements and unselected elements.
Selection Edge Vertices	This list refers to the vertices in the mesh that are on the boundary between selected elements and unselected elements.
Vertex Weights	This list is an array of floats that define the 'selection weight'. Vertices that are selected have a selection weight of 1 but if falloff is defined, surrounding vertices will have weights ranging from 1 to 0.

The size of the bit arrays that refer to the various elements of the object is determined by the size of the arrays in the mesh. That is, if there are eight vertices in the mesh, eight bits will be used to store the vertex selection, a further eight bits for the affected vertex list and eight floats for the vertex weights. Selections allow fall-off (see Figure 79) which can be

either linear, exponential or via a user defined curve. The fall off region indicates that a modifier will have an effect on the surrounding vertices but to a lesser degree. The fall-off values are stored in an array of weights inside the selection channel.

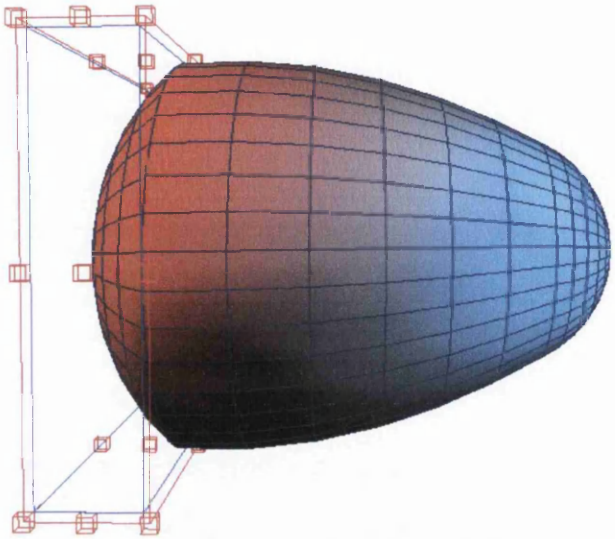


Figure 79. Selection Falloff.

4.3.8 Auxiliary per-vertex data items.

Whilst a mesh must contain vertex information in order to be visible, the inclusion of other per-vertex data items is optional. Textures coordinate, colours and weights can all be referred to by half-edges and there can be as many of these lists as is required. A list of auxiliary per-vertex data items may contain as many items as there are HEs or, if required, a single value that is referenced by all HEs.

Texture coordinates are used by materials that utilize texture mapping. Some materials may have many texture maps and the material node panel for such materials supplies options for selecting which texture coordinate channel will be used for each texture. An example of this are materials that employ light mapping where there is usually a base texture utilizing one set of texture coordinates and a further texture containing pre-calculated lighting information that must utilize another set of texture coordinates.

4.4 Display/feedback nodes

The output of the geometry generator and/or modifier nodes is fed into a node that can take the incoming data and render it to the display. This is referred to as the 'geometry

rendering node'. These nodes are part of the tree of visible nodes and it is this node that displays the geometric data and handles user feedback. All visible nodes are capable of receiving user events and handle these events in a similar manner to a UI element in a 2D user interface. In a 2D interface, visual elements must be aware of a pointing device and a every GUI has a means of handling events from a pointing device such as a mouse or light pen. The 3D scenegraph also responds to events from pointing devices but these are extended to 3D; An event that checks to see if the mouse inside a rectangular window requires a 2D point in a standard GUI but in 3D, this must be extended to a ray. Similarly, a selection rectangle in a 2D system can be represented as a frustum in a 3D system. The pointing device events currently handled by the system are as follows:

Table 9. Events within the 3D scene.

• **SGEVselectionfrustum**	Indicates that a selection frustum has been dragged over the object
• **SGEVmouseoverray**	Indicates that a mouse ray has passed over the object event
• **SGEVmousedoubleclick**	Indicates that the mouse has been double clicked over the object (quickly depressed and release twice in sucession)
• **SGEVmouseclickray**	Indicates that the mouse has been 'clicked' over the object (quickly depressed and released)
• **SGEVmousereleaseray**	Indicates that the mouse has been pressed whilst over the object
• **SGEVmousedownray**	Indicates that the mouse has been released whilst over the object

As can be seen, these events closely correspond to what one would expect from a 2D GUI system. All visible nodes are capable of receiving mouse events such as these and each node will respond differently. When designing the system, instant visual feedback proved to be most helpful to illustrate what should happen when the user clicks on an object using the mouse. To this end, objects will highlight portions of themselves when the mouse passes over to indicate that depressing the mouse will have an effect. Usually, this means that by clicking and dragging, the object will be altered in some way that is obvious from the visual feedback received.

Clicking and dragging outside of an object results in the creation of a selection frustum. The word 'frustum' is loosely defined as either a rectangle or a more complex projected shape, depending on the current mode (similar to rectangular or lasso selection in 2D design

software). This event is then sent to any objects that intersect this frustum and the event is then handled by the relevant objects.

In the case of the geometry rendering node, this event will be interpreted as a sub-object selection. The type of element that is selected is defined by a system-wide filter, the details of which are included in the event data structure. This is simply to allow fine granularity of control over the selection so that the user can select on a vertex, edge, surface or object level. It is conceivable that other sub-object elements such as texture coordinates could be included in this filter as well.

The manner in which the geometry rendering node responds to a selection event is that either creates or appends to a selection modifier. The geometry rendering node analyses the node producing the incoming data stream. If this node is not a selection modifier, a new selection modifier is created using the selection frustum. Otherwise, the selection frustum is appended to the list of frustums already assigned to the selection modifier. The selection modifier then alters or creates selection channels which are fed down the modifier chain.

Other than mouse selection and ray events, nodes can receive a large number of other events that they handle in a similar manner. For example, all nodes accept the **SGEVupdate** event which informs the node that an input has, or several inputs have, been updated. The geometry rendering node also accepts the **SGEVnewmodifier** node which is sent by the system to relevant all selected nodes. This event informs these nodes that a new modifier is to be added and the nodes themselves then ascertain, in the same manner as with a selection modifier, what kind of effect (if any) that modifier will have and can then make a decision to add that modifier to the chain. On creation, modifiers select themselves and the system displays the relevant node panels for the modifiers, allowing instant editing of the modifiers. Selecting the modifiers also has the effect of revealing any editing 'gizmos' that will appear in the 3D views for interactive editing. If several modifiers of the same type are selected, the system may decide to add a *meta-manipulator* node that can control all of the selected modifiers at the same time. One example where this is relevant would be a selection across several different objects and a subsequent stretch modifier. Each modifier has local controls for controlling the stretch but the meta manipulator also has control over the group.

Almost all modifier nodes require a selection channel to be active in order to have any effect. Modifiers work with selected regions of a mesh, derived from the lists inside the selection channel. Aside from the selection modifier itself, topological modifiers typically

update the selection channel as well. An extrusion modifier selects the previously selected polygons but also selects the newly extruded polygons with decreasing linear falloff. This allows subsequent geometric modifiers to affect the new vertices and polygons in an intuitive way. Also, some modifiers such as the shrink/grow selection modifier affect only the selection but require an incoming selection channel to operate.

4.4.1 Optimization of mesh data for specific render targets

The data that enters the geometry rendering node is in a format that is simple to update by geometry modifiers but inefficient to render using the target rendering libraries (OpenGL, DirectX or Software).

In order to facilitate real-time feedback for complex data sets, some sort of optimization is necessary. To begin with, vertex data is split over input channels including position, colour, texture coordinates and weights. These lists must be merged to form a single list in the manner in which the rendering library can process efficiently.

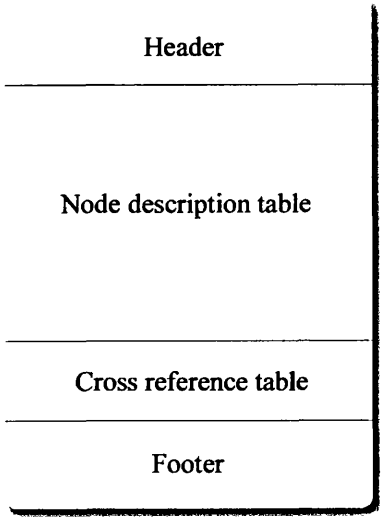
Also, topological data must be rearranged so that it is in an efficient form for rendering. Typically, this involves building triangle fans or strips and creating auxiliary data structures to define spatial relationships between parts of the mesh. The system performs triangle stripping of an arbitrary polygon mesh. The algorithm is greedy and does not generate optimal triangle strips; however, the stripping has to be performed in real-time or near real-time so a trade off between run-time speed and generation time had to be reached.

4.5 Additional application features

Although this thesis concentrates mostly on the working of the procedural generation and modification of shapes, the system described herein is large and has several other features worthy of note.

4.5.1 Serialization of Scenegraph data

The native input/output format for scenes from the system is a binary format containing a header, a node description table, a node cross reference table and a footer.



The file header consists of a file ID double word, and two word values describing the major and minor versions of the file format. The current specification describes version 2.1 of this file format.

The ID double word must be equal to the hexadecimal number '44335743', read as CW3D as an ASCII string.

Table 10. The file format header.

UINT32 id	ID double word, equal to CW3D or '44335743' in hexadecimal
UINT16 majorv	A word value describing the major version number, currently '2'.
UINT16 minorv	A word value describing the minor version number, currently '1'.

The header further continues listing the number of bytes needed to index various elements in the file. This section is called the file metrics block and takes up eight bytes.

UINT8 nodecodesize	This describes how many bytes
UINT8 nodeidsize	The number of bytes needed to store the node IDs for this file. Nodes are given unique IDs when saving and this value describes how many bytes are needed to store these values.
UINT8 maxnodebytesize	The number of bytes needed to store the size in bytes of any node in the file. This value is set to '1' if no node is larger than 256 bytes in length, '2' if no node is larger than 65536 bytes and '4' in all other cases.
UINT16 reserved	Reserved for future use

4.5.2 **Scenegraph & Resource Input/Output**

The import and export of existing graphics data formats is essential for any tool hoping to compete for space in a designer's arsenal. To this end, import and export utilities for several major 3D scenegraph formats and numerous resource formats have also been implemented. This area of functionality is separate from the node/scenegraph structure but still continues the philosophy of extensibility.

Scenegraph data formats suffer from unique problems regarding interoperability. 2D image formats, whilst diverse, are fairly standard in output; the result is usually just a 2D array of colour values. Scenegraph data formats, on the other hand, often contain a bewildering array of features specific to the application that produced them. In order to try and combat the problem of data loss in importing from and exporting to different formats, the system utilizes an approach method for registering and integrating 'foreign' application elements into the application. Import filters, as they are known, are registered at run-time in a similar manner to nodes. On registration, an import filter is able to register specialized

nodes for that particular data format and in so doing, automatically add to the functionality of the application. Whilst this can lead to an application internal version of 'DLL hell' as experienced in certain operating systems (where many versions of a DLL are resident or new DLLs offering very similar functionality are installed, rather than using standard libraries), if used correctly this system greatly increases the flexibility of the application. The general guideline to follow if writing a 3rd party filter is to make sure that if a new class is registered, it does not duplicate functionality found elsewhere in the application. A future goal of this system is to also be able to restrict the application so that the user can only create nodes that can be cleanly exported to a given environment. This streamlines the tool so that it becomes a native development environment for a given scenegraph format. Another exciting possibility is the concept of mappings between nodes from different formats which could conceivably allow for automatic mapping between disparate scenegraph formats with minimal data loss. An application of this might be to take the resource data from a 3D real-time strategy game such as command and conquer and convert it into a 1st person environment such as quake.

4.5.3 **Summary**

In this chapter we have looked at the internal implementation of the software, how data is passed from one node to the next, how events are managed and how the user interface is integrated into the system. There is still a great deal of work to do and some of the lower level aspects need re-working. The user interface replicates a lot of the functionality provided by the 3D scenegraph and auxiliary nodes; future versions will provide closer integration and sharing of interfaces and resources. The 2D and 3D rendering engines need to be reworked for closer integration and for platform independence; currently the 'material' nodes are tied to specific hardware platforms. It's clear that a shader language is required that can be translated into lower level representations (i.e, fixed function hardware pipelines, hardware fragment shaders and software shader implementations for slower but higher quality off-line rendering). We shall look at these possibilities in greater detail in chapter 6.

Chapter 5

GALLERY

This chapter presents some of the models created in the system, along with their file sizes and final output polygon counts.

Measuring polygon counts is not really an appropriate way to quantify the objects created using this system; the real factor that determines file size is the number of operations taken to create the objects and it is that which we must optimize if we are to reduce the storage space required to store the object. Polygons are just the output format of choice; this could just as easily be cardinal patches or a low resolution polygon model coupled with a 'normal map' for the high resolution detail. Table 11shows the resolution of the objects as shown in this chapter, as well as their compressed file size. Note that the helicopter model, which appears in three different resolutions, still takes up the same amount of space when stored.

However, as most other 3D file formats concentrate on the storage and transmission of polygons, it is reasonable to include the polygon counts here for comparison. A system utilizing such high level descriptions of how objects are constructed is able to describe much more in less space. From such descriptions, multiple resolution meshes can be constructed from the same data and, as the data is abstract and descriptive, rather than implicit and non-contextual (as a polygon mesh, or other low-level representation is), the data can be transformed in interesting ways once it is loaded by a system.

The model shown in Figure 80 has 4289 vertices, 14091 polygons and 48993 vertex references. The size on disk using this system is 14.7 kb. Assuming 12 bytes per vertex, and four byte vertex references and four bytes per polygon entry, this model would take up in excess of 413k of disk space using naïve storage, not including the space needed to store materials and other information. In contrast, this method stores materials, transformations and useful modelling information regarding the object in roughly $1/30^{th}$ of the space. Storing the file in a textual format, such as VRML, would take yet more space.

Table 11. File sizes for objects shown in this section.

Object Name	Vertex Count	Half Edge Count	Polygon Count	File size
Royal Albert Hall	2352	7969	1911	4.69 kb
Harley Davidson	12012	49231	13421	14.7 kb
Helicopter hires	2212	8849	2341	4.31 kb
Helicopter midres	8848	35396	9364	4.31 kb
Helicopter lowres	35392	141584	37456	4.31 kb
Lighthouse	30072	143774	44758	13.2 kb
Apple	3486	14020	3528	1.79 kb
Space Plane	2601	9740	2482	2.14 kb
Furry Torus	10800	43200	10800	0.970 kb
Dome	7007	27958	6978	1.16 kb
Compressor	2831	6320	2521	3.97 kb

Figure 80. Harley Davidson Sportster.

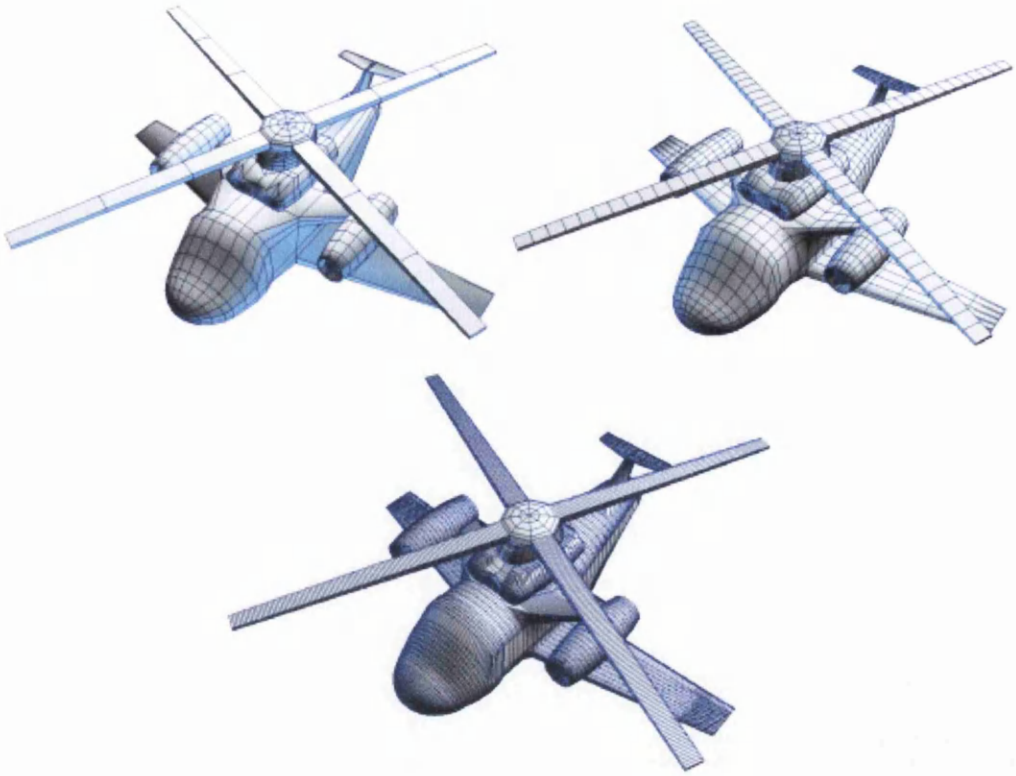


Figure 81. NASA Helicopter model at three different resolutions.

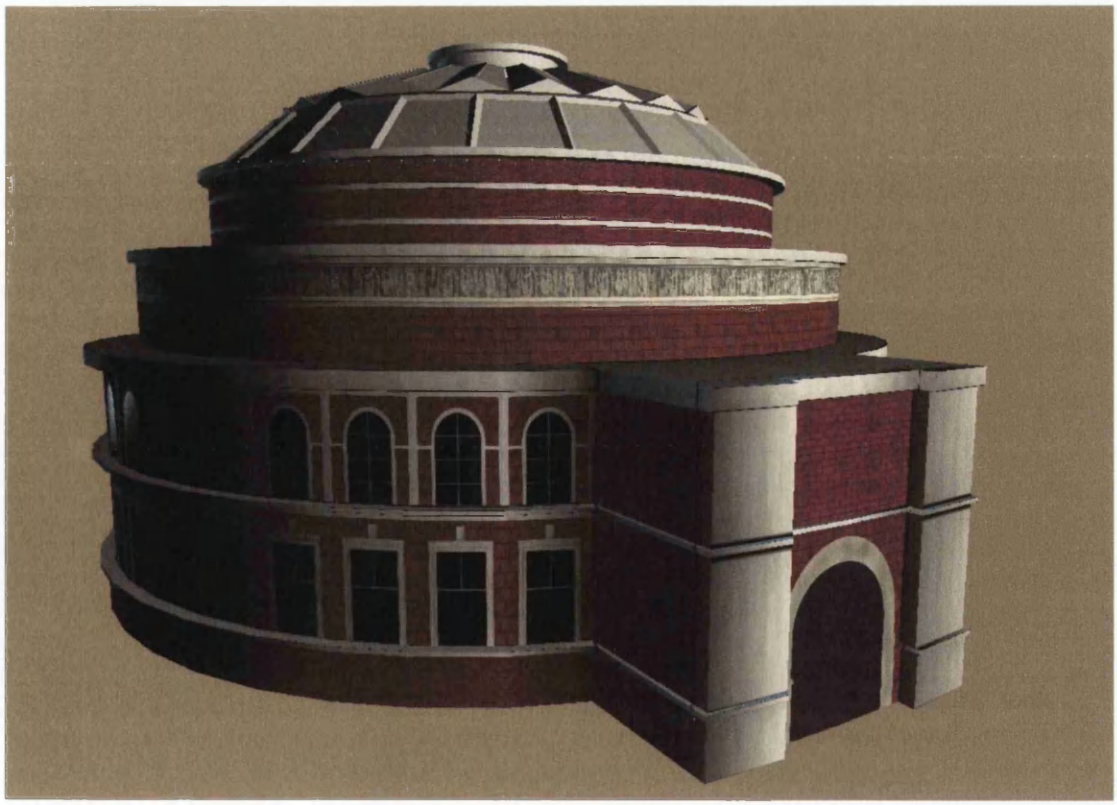


Figure 82. The Royal Albert Hall.

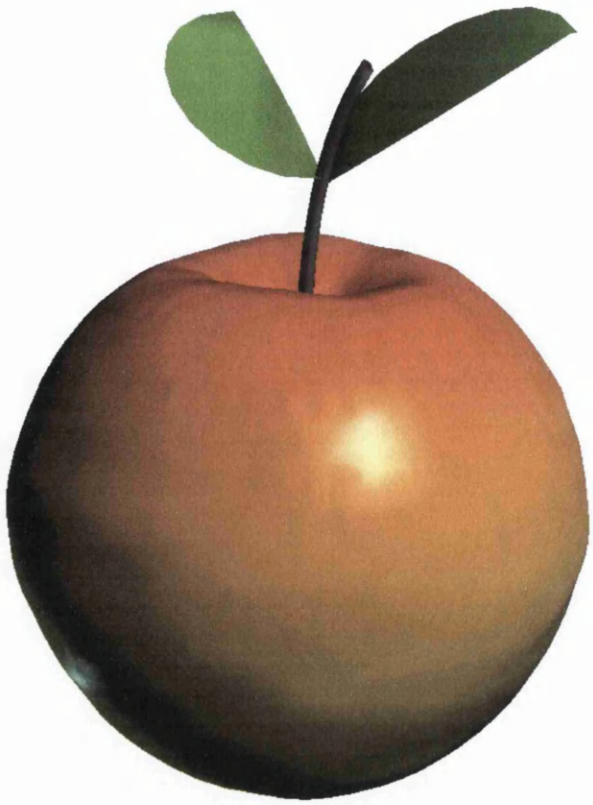


Figure 83. Procedural Apple in 1.79 kilobytes.

Figure 84. Space plane in 2.14 kb

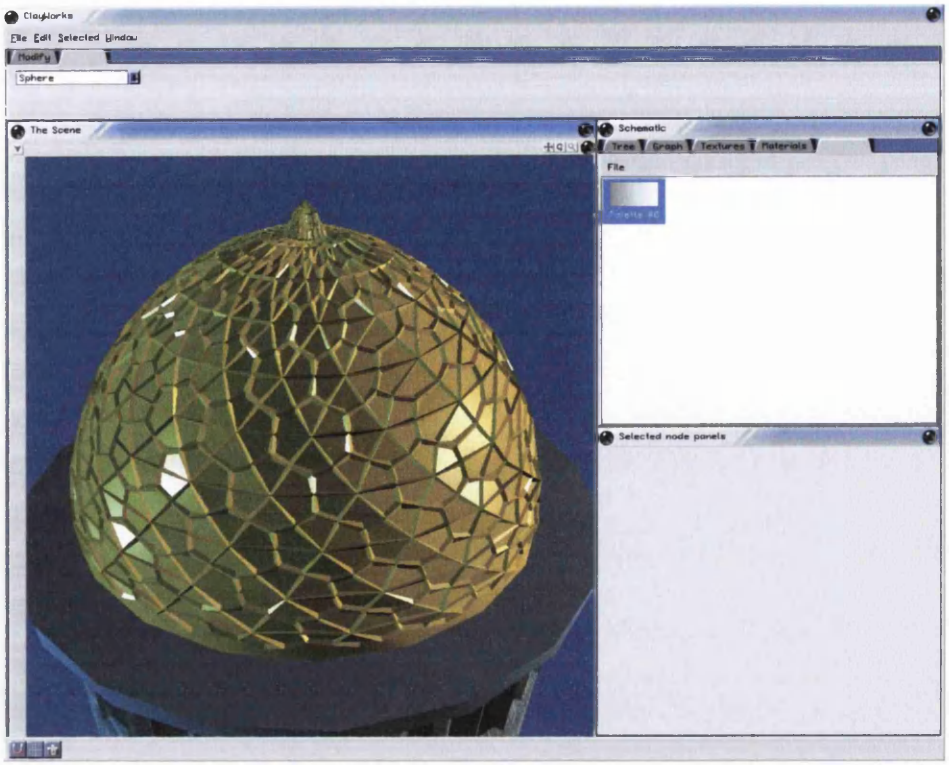


Figure 85. Dome roof, created using triangulate and polygon extrusion modifiers.

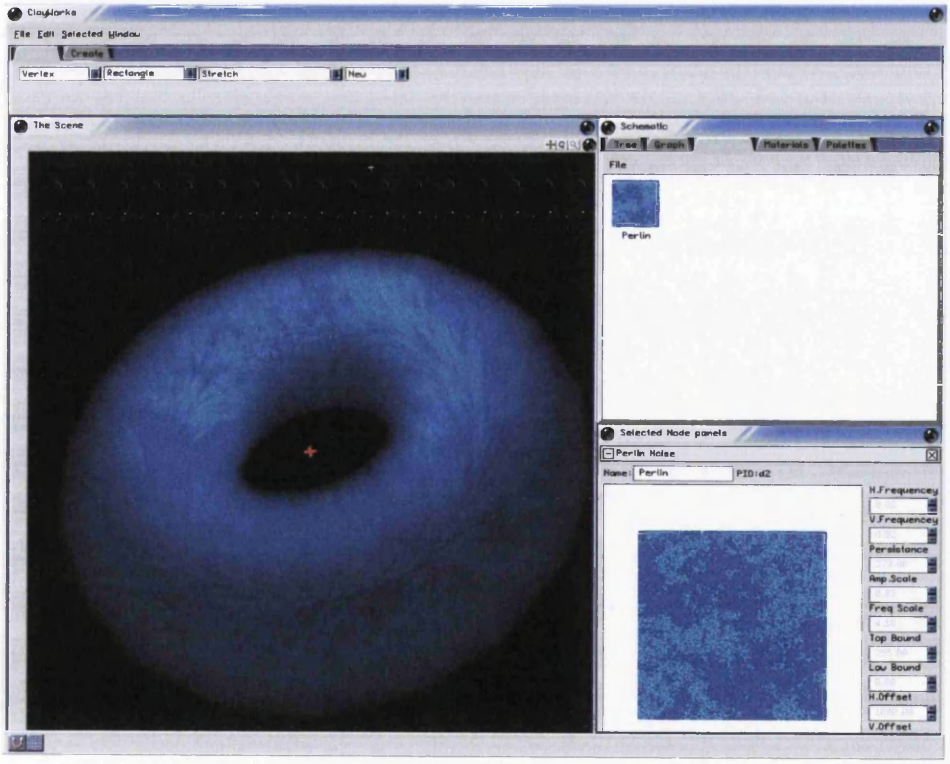


Figure 86. Fur, created using Perlin noise, copy and pseudo distance field modifiers.

Figure 87. Compressor unit.

Figure 88. Lighthouse scene at day time.

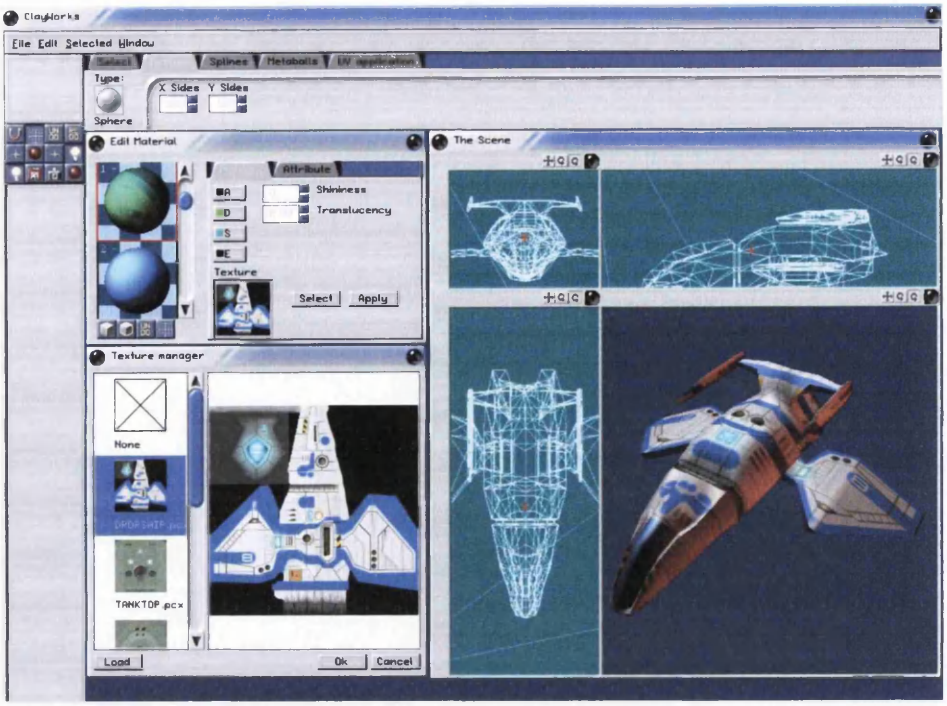


Figure 89. Earlier implementation of software using software rendering engine, in MS Windows.

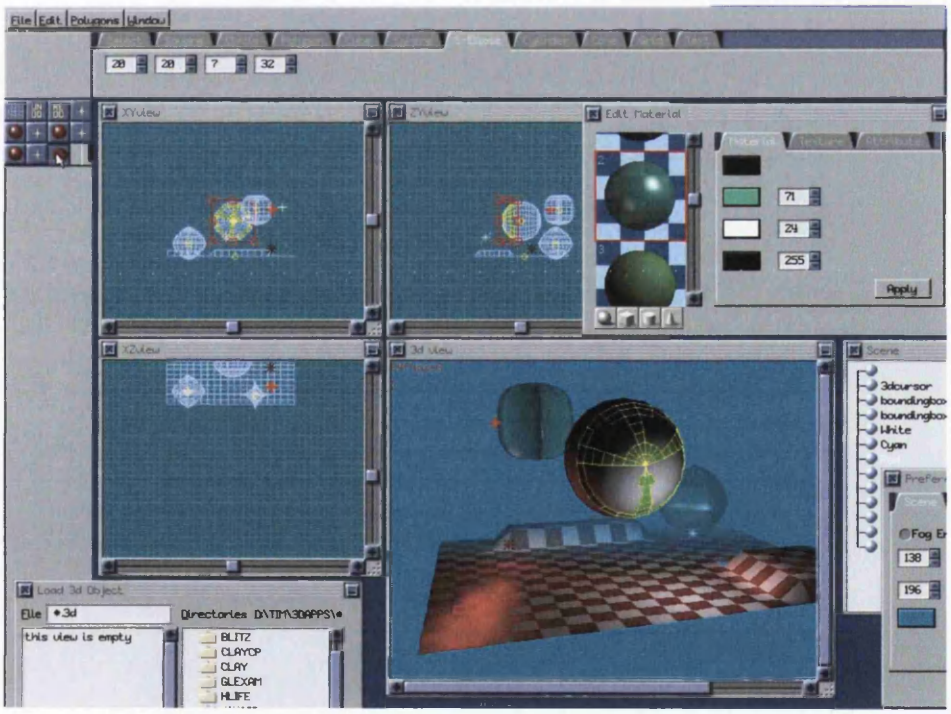


Figure 90. Early implementation of software, using software rendering engine and older interface style, under MSDOS and DOS4G/W for 32 bit support.

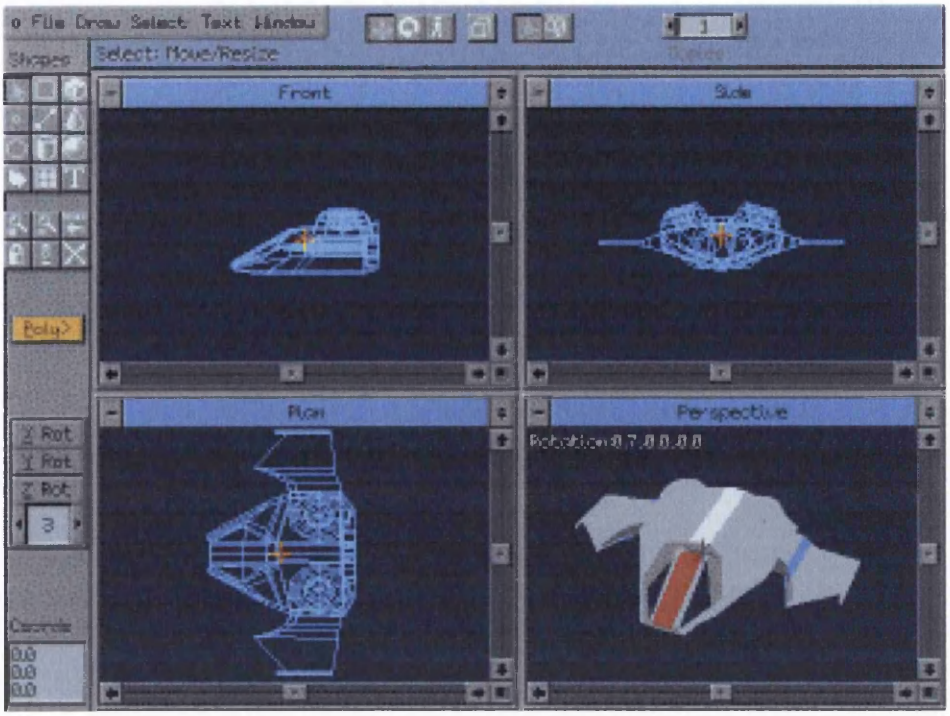


Figure 91. Very early version of a non-procedural 3D modelling package by the author, running in MSDOS.

CONCLUSIONS & FURTHER WORK

6.1 Retrospection

The central topic of this thesis describes a system for defining 3D shapes and deformations in a procedural manner that has all the flexibility and functionality that a user would expect from a modern 3D Modelling system. In addition, the data used to describe a form is at once more compact and more useful than a standard mesh definition consisting of connected points in space. The object description is as much a set of instructions as to how to build a shape as it is an object description and as such, any step along the path to the final shape can be altered at any time, resulting in great flexibility for the designer. As a consequence, high level compression of this chain of instructions is possible (by removing redundant nodes and possible coalescence of others) even before bitwise compression is considered.

We introduced the goals of this thesis in the first chapter, outlining the aim to create a procedural modelling program that would deliver significant savings in both storage and enhanced usability by storing descriptions of how objects are created, rather than explicit, fixed representations of objects.

Chapter two presented a literature review of the diverse fields covered by this thesis. This included procedural modelling, generative modelling and user interaction in 3D.

Chapter three described the function of various nodes in the system, concentrating on those responsible for generation of 3D shapes. These include geometry generators, selection nodes and modifiers.

Chapter four discussed lower level implementation details and efficiency concerns, including how data is transferred from one node to the next, how events are managed and how polygon geometry is optimized for final rendering and interaction nodes.

Chapter five, the gallery, shows visually what has been achieved, along with notes describing the images and their significance.

6.2 Successes and highlights

Although this section highlights many areas of future research, the application described in this thesis fulfills all of the initial goals; namely, to create a non-destructive procedural modelling program that creates terse yet descriptive files whilst being at least as easy to use as non-procedural modelling tools. The fact that there are so many future avenues to explore is, in fact, indicative of this success.

6.2.1 More from less

One of the key advantages of an implicit or procedural approach to 3D modelling is that the resulting data files and both smaller than explicitly expressed files yet are more descriptive. This approach takes procedural modelling into the domain of a useable artistic and design tool by expressing such objects in a way that is both amenable to WYSIWYG editing without sacrificing parametric control. The node model performs excellently as a paradigm for creating, using and storing not only 3D models but also entire visual applications. It is a powerful paradigm, allowing introspection of objects so that the application can optimize output as well as being easier to understand and modify than a scripting language. As presented in this system, it's also non-intrusive, meaning that the user does not have to become embroiled in the node graph if they do not wish to do so; the interface provides expressive visual tools for the vast majority of modelling operations.

6.2.2 Ease of use

Whilst 'ease of use' is largely subjective, the goal of creating a procedural modelling system that is as simple to use as non-procedural modelling programs has, in many respects, been exceeded in that the software is far easier to use than the vast majority of 3D modelling software. This is partly due to design principles and visual feedback and partly because of, rather than despite of, the procedural generative modelling paradigm which allows for non-linear editing in a non-destructive and predictable way.

The application described in this thesis involves the user in the structure of their design by clearly stating the steps used to create it whilst allowing the use of expressive tools. These steps then become tools to further explore and refine a design, rather than a static history of events as seen in countless undo trees.

The system also refrains from relying on creation methods and editing methods that can only be performed by tweaking numeric controls; great care has been taken to create 3D GUI controls for modelling operations that are both intuitive and powerful. The speed with

which 3D shapes can be created is testament to the effectiveness of the user interface of the application presented in this thesis.

6.2.3 **Extensibility**

One of the core design requirements for this system has been that it must be extensible. Again, the node model has made this possible by allowing both the application and the files generated by the application to be seamlessly updated as new node types are added. When a new node is added to the system, it is automatically integrated into the user-interface without needing further programming work; All of the information required by the system to interact with the node is provided within the definition of the node.

6.2.4 **Efficiency**

As real-time 3D graphics evolves, the data requirements of our applications increases. The storage requirements for dense 3D meshes and their auxiliary data have, in modern 3D games, grown to gargantuan proportions. At the time of writing, most new 3D games contain several gigabytes of content data. As mainstream applications gradually become more media-rich, they too may suffer the same fate, putting further strain on system resources. Whilst many may point out that memory and storage space are becoming ever cheaper, this is not an adequate excuse for inelegant waste of resources. The method described in this thesis is an elegant solution to this problem, concentrating on the compression of process, rather than result. It is far easier to optimize the process of recreating an object from a description of how it was made than it is to speed up access from fixed media or over a network; the later are entirely dependent on the hardware configuration of the target system whilst the recreation process can be tailored, on the fly, to suit the specification of a target machine. The system also allows for a degree of flexibility if, for example, a portion of the graph does take longer to calculate than to retrieve from fixed storage; the data can simply be created and cached as needed on the local machine.

6.3 Future enhancements

A possibility for future expansion within the editing system would be the addition of specialized object nodes for different sectors of industry. Whilst the system described in this thesis concentrates on basic geometrical primitives, it would be trivial to add more specialized base primitives that would be of use to specific areas. One obvious example would be the addition of basic mechanical forms such as procedural cogs, levers, pulleys, cams and so on for mechanical design or education. Others might include base primitives

useful for designing the layouts of buildings such as walls, doors and staircases. Specialized objects such as these would make it easier to design complex forms common to certain areas of 3D design. More importantly perhaps, it would abstract the process; a cog would 'know' that it is a cog, rather than a collection of polygons that look like a cog. In this manner, if a cog object receives a collision event from another cog, it would know how to turn based on the gear ratio and in what manner their teeth are meshing, rather than having to rely on a more generic but computationally expensive algorithm.

6.3.1 Application specific scenegraph optimizations

Since nodes are effectively black boxes, we only care about whether a node will produce consistent output for a given input. The actual implementation of a node is largely irrelevant. However, the needs of a 3D Modelling environment do differ slightly from that of a simulation. The former usually requires rich information in its object descriptions that is of use in altering the object whereas speed and performance might be more important for the later. While there is significant overlap, it might be wise to tailor individual versions of the internal object code for development and use. Being able to specify which inputs and outputs are altered more frequently and which are effectively static once an object is created would be of great benefit in optimizing a scene for a specific use. An object used in a game might not need selection information essential to an editing environment but it would perhaps need more information regarding visibility. Likewise, some objects (e.g, a landscape, or a city block) are less dynamic than others (such as an articulated character) and would benefit from complex visibility structures that are rarely computed (BSP trees, PVS trees etc.), whereas these would hinder the latter.

Fortunately, the node structure does not dictate that the internal workings of a node be identical on different systems or that they even be contained within a spatial scenegraph that operates identically. A real-time gaming environment might require additional, or at least different, information for visibility calculations than an editor would; its tailored scenegraph would reflect this, even if the node data connectivity was identical. Likewise, an object node in an editor would need to have certain methods of viewing objects schematically, as a wire frame view or showing other visual information that would be redundant in a game or virtual environment. So, whilst the actual implementation of nodes and additional hierarchical (usually spatial) structures differs depending on the intended use of the graph, the data interconnections remain the defining descriptive structure of a scenegraph.

6.3.2 Animation

Whilst this thesis has described in detail the flow of data necessary to transform a base primitive into a complex and arbitrary 3D form, it also describes a framework that, with further work, could be used to describe a working 3D environment in totality. The system, in its current form, provides a means of sharing data between nodes describing textures, colour palettes, materials and 3D shapes and instances of those 3D shapes in an environment. However, this is just the tip of the iceberg. With further work, it will be possible to describe animation, audio, behavioural and many other forms of data within an environment. The nature of this framework also makes 3rd party additions to the software and even connectivity between the system described here and totally different pieces of software, quite possible. A 'time node' or series of them, fed into nodes describing a curve which could in turn feed into any number of different nodes could perform any conceivable animation and not require a redesign of the basic system in any way. Similarly, these could also be controlled by nodes that could analyze the scene and provide behaviour which also feed into the animation. Other nodes could act as triggers for behaviour which would all work together towards creating a living environment, all sharing data in the well defined manner proposed in this thesis.

6.3.3 Discussion on networking & collaboration

Since an object description embodies the steps taken to create an object, a system such as the one described here would break down the barriers between the development environment of a 3D scene and the 'playback' environment (e.g, game or simulation environment). If desired, this could add a new level of user interaction into online games and simulations or simply as a new, distributed means of creating such environments. Since data connections between nodes are well defined, they are apt to be split across multiple processors or separate computers on a network. Consider a game creation environment where programmers, artists and musicians and product testers are running software compatible with this node model. A texture artist might have control over a texture asset whilst a 3D sculptor is working on an object that utilizes it. They could both have real-time feedback of their changes without overwriting each others work. Similarly, a tester might actually be running a test version of the game, with asset streams coming in from other developers and be able to make suggestions about the game world (say, a misaligned texture or an object upsetting AI path-finding behaviour) and have real-time feedback from the other developers to perfect the game without even restarting it. Such tools, tailored for the styles and habits of a particular development house would be of great benefit in testing and

perfecting of a virtual environment. Issues of ownership and collaboration between programmers and asset creators often disrupt the development of a product. The node/scenegraph model presented in this thesis could become a system flexible enough to solve many of the problems inherent in medium to large groups of developers working together. For example, creation of detailed polygon models, complex textures, fine tuned behavioural algorithms, music and sound effects can take a great deal of time. During this time, the designer of the overall environment (usually referred to as the 'level' or 'game' designer) might be idle and development of the product would be staggered. If the designers could come up with simple placeholders to describe the basic form and behaviour of nodes then the level designer could start to create the basic structure of the environment and see the, even during construction, the refinements of the data assets as each member of the team refines their original form into a final product. This would not require recompilation or even to have all the assets duplicated on the designer's machine, they would simply tap into data streams originating from each developer. This distributed world model would be useful for teams on a local network but extremely useful for geographically separate developers.

6.3.4 Refinements regarding data types

One area that has proved difficult to elegantly solve has been the issue of the data types shared between nodes. Some data types, such as scalar values, colours and vectors present themselves as obvious candidates. However, when it comes to more complex classes, issues of granularity appear. Should a texture data type be a generic texture container for all bitmap resources or should each possible pixel format have its own data type? Should a sound wave be an array of floating point values or an encapsulated class? The most vexing has been the fundamental polygon mesh class. Currently, this class is a container for a vertex array, an array of edges (in the half-edge format), polygonal references to those edges, an arbitrary amount of vertex colour, texture coordinate and weight arrays, the necessary per-edge references to those lists and a selection channel.

6.3.4.1 Current data type implementation

For ease of implementation, these various arrays have been coalesced into a single data type but this might later prove to be restrictive. It would be advantageous to redefine the basic data types into more fundamental, simple, data types and arrays of these types in various dimensions. Modifier nodes would then only receive the data they need which would make internal housekeeping duties inside modifiers more efficient and simpler. This would also make creating a more efficient means of updating and propagating changes through a

graph whilst making modifier nodes more generic (a space warp modifier would then work on any vertex array and would not need to know about polygon meshes at all). The overhead would be in methods to automatically work out how nodes should be wired up (an infrequent operation) and there are also problems in topological data. The edges need to know about the vertex array, or at least how many vertices there are in the list and edges also need individual references to any colours, texture coordinates or weights. Also, a polygon is nothing without and edge list to refer to. Per-polygon smooth group and material references are also meaningless without a polygon list to start with. Also, images could be considered two dimension arrays of colour values, rather than a complex image class but this also raises problems. The current image class is highly flexible and self contained but it would be advantageous to de-couple this and treat it as a simple array of colour values. Such low-level granularity would also allow for simple type casting; a sound wave could be type cast as an array of colour values or even a vertex list if both were considered an n dimensional array of floats. Whilst this might sound undesirable in some situations, it is important to remember that the tools described here is essentially a designers tool. By creating a looser system, users of the program could, by simply reconnecting a few nodes, create new, useful or just plain attractive uses for the tool not previously envisaged. To this end, future versions must exhibit an ideal level of granularity that is at once more flexible and more efficient than the current system. One such method is discussed in the next section.

6.3.4.2 **RTTI (Run Time Type Information)**

RTTI (also known as introspection or reflection) [STROUS] is a term used to describe a system whereby the application has access to the kind of type information usually reserved for the compiler. Some languages (such as Java, C#, SmallTalk and Managed C++) feature RTTI constructs within the system itself but in other languages, such as C++, the programmer must provide his or her own system if RTTI is required. The system already uses a rudimentary system to disclose types at run-time (as discussed above) but this system does not provide the level of flexibility required.

Ideally, such a system would be able to query the types of inputs and outputs in greater detail as well as disclosing the interfaces implemented by a node, the inheritance of classes by a node and a similar level of hierarchical flexibility for the input/output data to and from a node. The RTTI system should also be as transparent as possible to the application programmer, have minimal overhead in terms of storage space and execution speed and allow introspection of data types without creating instances of those types.

An RTTI will allow the application to decide how to connect nodes together and would also allow for greater control over the flow of data, as data-types are known at run-time. RTTI would also allow the system to create node-panels by analyzing the structure of a node, rather than requiring the programmer to create a node panel separately (although the option to do this should still be available). Currently, the implementation of nodes, their corresponding node-panels and also making the application aware of the nodes and their context (for example, the tools to create material nodes and modifier nodes appear in different areas of the application interface) is the most time consuming phase of development. RTTI will ease this process a great deal.

Implementing an RTTI system in a language that does not feature such functionality as standard can be problematic: on the one hand, a RTTI system should make development easier as the application can make more informed choices regarding how to use and place modules, classes or nodes within the system. However, unless great care is taken when designing the RTTI system, it will add additional layers of complexity and degrade performance unacceptably.

C++ features a certain level of compile time processing in the form of the pre-processor and also the generic programming template system, using a programming technique known as 'Template meta-programming', first coined by E. Unruh [UNRH] . Early research into this area has shown that it should be possible to implement a flexible RTTI system that stores type information statically, so as not to incur a great cost per instance of a node or data-type, whilst remaining largely transparent to the application programmer. More research is required but future versions of the application will benefit a great deal from RTTI methods.

6.3.5 **2D vector graphics**

With the advent of powerful 2D vector graphics libraries such as SVG, Flash, MacOS X and various systems from Adobe and others along with more advanced graphics hardware, it seems logical to stop relying solely on bitmapped graphics for our 2D interface and texturing needs. Whilst bitmaps have their place as textures to add complex and arbitrary detail, the scalability and ease of manipulation inherent in vector graphics make their use a logical choice.

The interface of the system described in this thesis relies mostly on bitmapped textures. The possibility of creating a 2D vector graphics library that operates using the principles

described here for the 3D scenegraph creates enticing possibilities. Instead of explicitly coding a user interface element so that it controls the parameters of a certain node, the user interface would link directly to the scenegraph using the kind of node connectivity described in this thesis. Users wishing to customize the program could do by exposing certain areas of the 2D scenegraph and altering them in some way. The user interface would be visible on the graph editor and would work with the 3D scenegraph in a consistent manner. The user interface could then also be used to add a 'Head up Display', or HUD, which would be visible in the 3D view and used for user defined parametric control over elements in the scene.

A vector based user interface would be scalable, portable and far more flexible than a system based solely on bitmapped images. Whilst seen by some as frivolous, transition and animation effects within the interface would be greatly enhanced.

By utilizing the processing power of pixel and vertex shaders, 2D vector objects could also be used as textures or decals on 3D shapes. Such shapes would not exhibit the blurring (in the case of filtered textures) or pixilated effects (in the case of unfiltered textures) associated with texture map data. Such 2D decals would also be able to take advantage of the powerful parametric functionality described in this thesis.

6.3.6 **Implementation of a Shader Language**

At the moment, the materials and the graphics engine are platform specific. There is a need to implement a shader language that can be converted to whatever hardware specific platform we are running on. A high quality software rendering library, able to anti-alias at glyphs at a high level whilst rendering complex shaders is also needed, along side a hardware accelerated implementation of the same functions. There are several high level and low level shader languages available at the time of writing; one of the most famous and long lived is Pixar's RenderMan [RNDM] which has been used to great effect in many animated films by that company and others. Modern graphics hardware is capable of performing shader language instructions per-pixel in real-time and to meet this demand, NVIDIA have released 'Cg' [NVIDCG], ATI have created 'RenderMonkey' [ATIRM] and Microsoft have written 'HLSL' or 'High Level Shader Language' [MSHLSL]. Each of these languages are separate from the low-level byte code representation of shaders (there are also several shader assemblers available) and also separate from the rendering library; the shader language is really a way of providing extreme flexibility when filling polygons.

This will be a great boon for 2D, as well as 3D, graphics as complex shaders break the monotony of gradient and pattern fills that are the mainstay of 2D drawing programs and libraries. As well as updating the 2D and 3D rendering engines, future iterations of the software described in this thesis will feature the use of a shader language to describe polygon fills and material properties. This feature will probably follow the specifications of a lower level shader language to simplify the implementation of a software shader JIT (Just In Time compiler) and so that it can be easily made compatible with the plethora of higher level shader languages.

6.3.7 **Use of compression in the file format**

Currently, no formal attempt at compression has been made within the file format. Despite this, files produced by the system are already highly compact. The node interconnectivity data and also the identifiers used for nodes within the data format are prime targets for Huffman encoding to facilitate compression.

6.3.8 **Interval Analysis for mesh generation**

Whilst the system described in this thesis provides an implicit object definition alongside the streamed mesh data, this implicit object definition is not used as well as it might be. Snyder's work [SNYD] on generative Modelling utilized interval analysis to generate the final shape from an implicit generative description. Future work on the system will include the integration of interval analysis techniques to generate mesh data, rather than the iterative approach we currently employ, which is prone to errors when approximating curves. Utilizing interval analysis will allow us to keep the sharp edges of polyhedral shapes whilst better approximating curved surfaces.

Figure 92. Image of the earth, as rendered by the software.

Appendix A

MODIFIER LISTS

This section contains listing of all the geometry generators and modifiers used to create the objects shown in chapter 5. The modifiers have been limited to their type names only for brevity. Also, only the modifier lists are shown, even if these take other nodes in the graph as input. Each lists starts with a name for the modifier chain which, by default, is the name of the geometry generator type. These names can be changed within the program, as can the given names of each modifier. Adding instance names to nodes in the scenegraph does add to the final file size and is only really necessary as an aid to modelling; final 'builds' of a scene could omit names to further reduce the size of files.

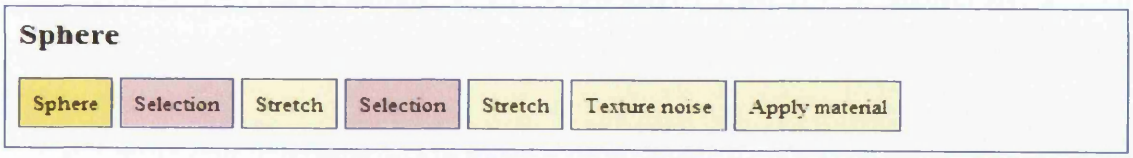
Geometry generator nodes are shown in yellow, selection nodes in pink and modifiers are shown in cream.

7.1 Fluffy Torus Model (Figure 86)

This model shows the simulation of fur using the 'copy' and 'distance field' modifiers.

7.2 Apple Model (Figure 83)

The apple model makes use of Perlin noise to distort the shape and provide the texture. The Perlin noise nodes feed into 'texture noise' and 'apply material'



7.3 Harley Davidson Model (Figure 80)

This complex model took an afternoon to create and is the most complex object shown in the thesis. The model is made up of numerous sub-assemblies, the modifier chains for which are listed here in their entirety.

Inner frame

Polygon | Selection | Polygon Extrude | Selection | Plug hole | Selection | Stretch | Selection

Exhaust 2

Cylinder | Selection | Extrude | Rotate | Extrude | Rotate | Stretch | Extrude | Rotate | Extrude
Stretch | Extrude | Rotate | Extrude | Stretch | Extrude | Stretch | Extrude | Stretch | Extrude
Stretch | Selection

Exhaust 1

Cylinder | Selection | Extrude | Rotate | Extrude | Stretch | Extrude | Rotate | Extrude
Stretch | Extrude | Rotate | Extrude | Stretch | Extrude | Stretch | Selection | Stretch
Selection | Extrude | Stretch | Selection | Stretch | Selection | Stretch | Selection | Stretch
Selection | Stretch | Selection

Engine 2

Polygon | Selection | Polygon Extrude | Polygon Extrude | Selection | Plug hole | Polygon Extrude
Selection | Assign smoothgroup | Selection | Stretch | Selection | Stretch

Cylinder

Cylinder | Selection | Extrude | Stretch | Assign smoothgroup | Selection | Copy | Stretch
Selection

Back spokes

Cylinder | Selection | Extrude | Stretch | Extrude | Stretch | Selection

Back rim

Torus | Selection | Extrude | Stretch | Extrude | Stretch | Extrude | Stretch | Extrude | Stretch
Selection

Fork

Cylinder | Selection | Polygon Extrude | Polygon Extrude | Polygon Extrude | Polygon Extrude
Stretch | Selection | Polygon Extrude | LG | Polygon Extrude | Polygon Extrude | ug
Polygon Extrude | Polygon Extrude | Polygon Extrude | Polygon Extrude | Polygon Extrude
Polygon Extrude | Polygon Extrude | Selection | Extrude | Stretch | Selection | Extrude | Stretch
Delete | Selection | Stretch | Selection | Copy | Flip Normals | Scale | Move | Selection
Stretch | Selection | Extrude | Stretch | Extrude | Stretch | Extrude | Stretch | Extrude
Stretch | Selection | Stretch | Selection | Extrude | Stretch | Extrude | Stretch | Selection
Stretch | Extrude | Stretch | Extrude | Stretch | Selection | Extrude | Stretch | Extrude
Stretch | Selection

Seat

Cylinder | Selection | Stretch | Selection | Stretch | Selection | Stretch | Selection | Extrude
Stretch | Selection | Stretch | Selection | Stretch | Selection | Stretch | Selection | Extrude
Stretch | Selection | Extrude | Stretch | Rotate | Stretch | Selection | Stretch | Selection
Stretch | Selection | Stretch | Selection | Stretch | Selection | Stretch | Selection
Assign smoothgroup | Selection | Rotate | Stretch | Selection

mudgaurd

Circle | Selection | Stretch | Selection | Stretch | Selection | Polygon Extrude | Rotate
Selection | Stretch | Assign smoothgroup | Stretch | Rotate | Stretch | Plug hole | Selection
Stretch | Copy | Stretch | Flip Normals | Stretch | Rotate | Stretch | Selection

Cube

Cube Selection Extrude Stretch Selection Extrude Stretch Selection Extrude
Stretch Extrude Stretch Selection Stretch Extrude Stretch Selection Stretch
Selection Stretch Extrude Stretch Selection Stretch Selection Stretch Selection
Extrude Stretch Extrude Stretch Selection Extrude Stretch Selection Extrude
Stretch Selection Extrude Stretch Extrude Stretch Selection Stretch Extrude
Stretch Selection Stretch Selection Assign smoothgroup Apply material Rotate
Selection Stretch Selection Stretch Selection Stretch Selection Stretch Selection
Extrude Stretch Selection Extrude Stretch Selection Stretch Selection Stretch
Selection Stretch Selection Stretch Selection Stretch Selection Stretch Selection
Stretch Selection Rotate Stretch Selection Copy Stretch Flip Normals Stretch
Selection Stretch Selection

Super Ellipse

Super Ellipse Selection Stretch Selection Apply material Selection

Radiators

Cube Selection Extrude Stretch Selection Assign smoothgroup Extrude Stretch
Selection Stretch Apply material Selection Extrude Stretch Extrude Stretch Extrude
Stretch Selection Rotate Copy Stretch Flip Normals Selection Assign smoothgroup
Selection Stretch Flip Normals Selection Assign smoothgroup Selection

Cube

Cube

Cube

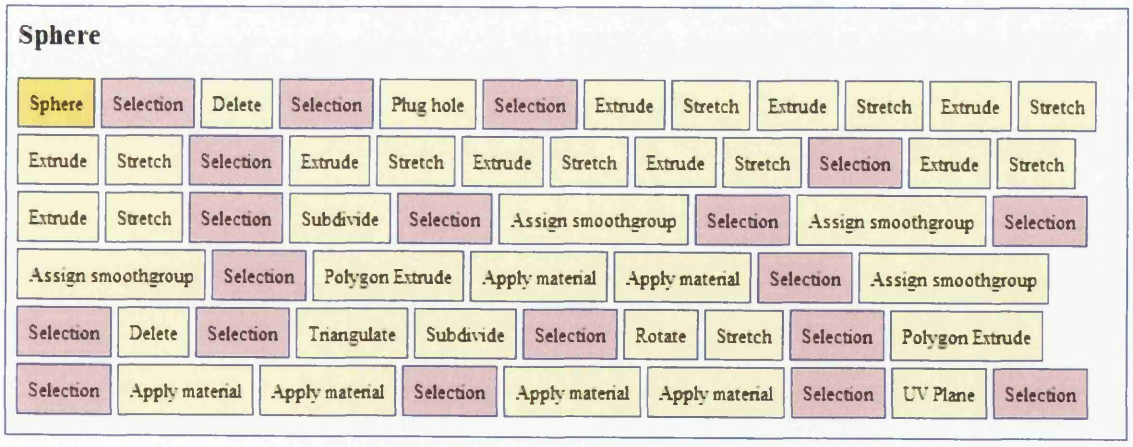
Cube Selection Extrude Stretch Extrude Stretch Selection Stretch Selection

7.4 Albert Hall Model (Figure 82)

This is a model of London's Royal Albert Hall. Although the model isn't accurate to an obsessive degree, it does show how a procedural system can be used to re-create real-world objects and store them in very little space.

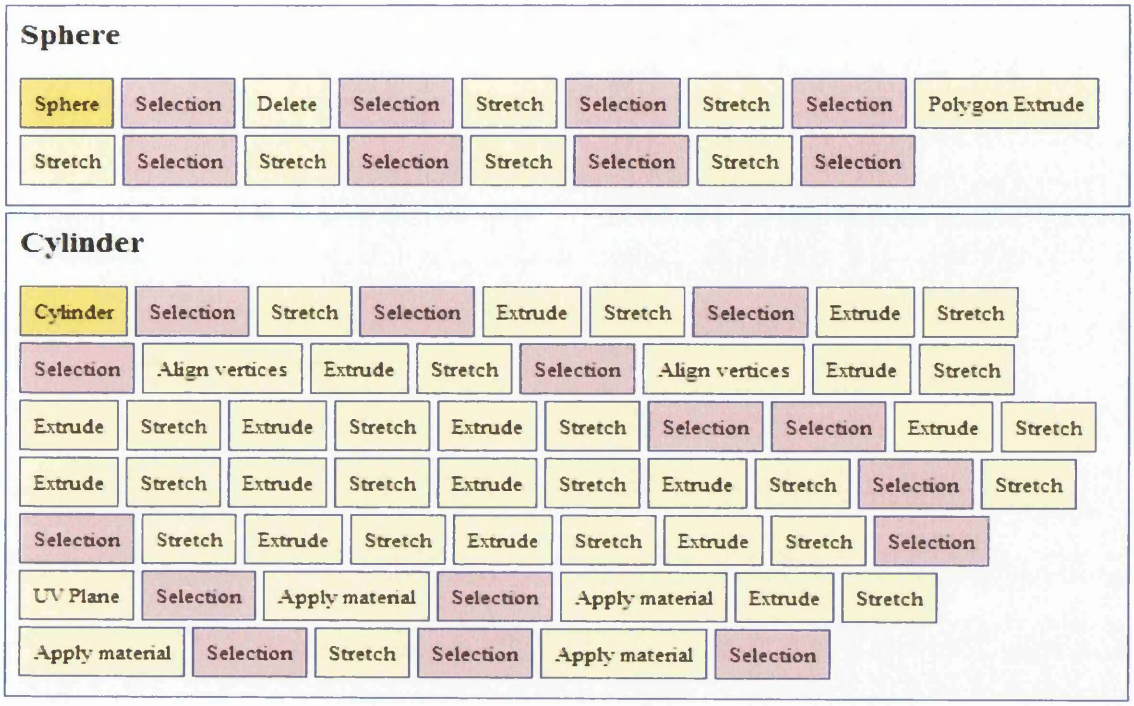
7.5 Dome Model (Figure 85)

This model shows an interesting effect created using the 'triangulate', 'subdivide' and 'per-polygon extrude modifiers'.



7.6 Space Ship Model (Figure 84)

This model is a science-fiction space craft designed with the system for a computer game. The model features partial specularity, achieved with an 8bit mask, to give a slightly worn look.



7.7 Compressor Model (Figure 87)

This model was created for the website of a company in South Wales who manufacture industrial refrigeration units. The final model of one of their units features several instances of this model.

7.8 Light House Scene (Figure 88)

The lighthouse scene is a proto-type for a personal interactive interface project.

Heightfield Grid

Heightfield Grid

tex trans

Heightfield Grid

sky dome

Sphere | Selection | Apply material | Selection | Delete | Selection

Bulb

Cylinder | Selection | Apply material | Copy | Distance field | Selection

lighthouse

Cylinder | Selection | Stretch | Selection | Extrude | Stretch | Extrude | Stretch | Extrude

Stretch | Extrude | Stretch | Extrude | Stretch | Extrude | Stretch | Extrude | Stretch | Extrude

Stretch | Extrude | Stretch | Selection | Apply material | Selection | Assign smoothgroup

Selection | Polygon Extrude | Apply material | Selection | Assign smoothgroup | Selection

Polygon Extrude | Apply material | Apply material | Selection

sea

Heightfield Grid | Selection | Texture noise | Selection | Apply material | Selection

island

Heightfield Grid | Selection | Texture noise | Triangulate | Apply material | Selection

groundplane

Heightfield Grid

7.9 NASA Helicopter (Figure 81)

This is a model of an interesting NASA helicopter model. Only one source photograph from the NASA website were used as reference during its construction. As a result, the rear of the model may not be accurate to the original.

LIST OF FIGURES

LIST OF TABLES

BIBLIOGRAPHY

□ [ANDR]
Anderson, M.G., "Modelling Geomorphological Systems", ed. M.G. Anderson, 1988, New York, *John Wiley & Sons.*

□ [BAR1]
Barr, A.H.,"Superquadratics and Angle Preserving Transformations", *Computer Graphics*, 15(3), pp. 11-23, August 1984. IEEE CG&A

□ [BIER]
Bier, E.A. Skitters and Jacks: Interactive positioning tools. In *Proceedings 1986 ACM Workshop on Interactive 3D Graphics*, pp. 183-196, Chapel Hill, North Carolina, New York October 1986.

□ [BLINN1]
Blinn, J., Newell M.E., "Texture and Reflection in Computer Generated Images", *Communications of the ACM*, Volume 19 Issue 10, pp. 542 – 547, 1976.

□ [CAST]
Castier, B., Martha L.F and Gattas M. "A taxonomy for interactive manipulation and visualization of 3D objects." *Sibgrapi97*, pp. 26-33, 1994. (*In Portuguese*)

□ [CARMACK1]
Carmack, J. "Quake Rendering Engine", *www.idsoftware.com*

□ [CATM]
Catmull, E. and Clark, J. "Recursively generated B-spline surfaces on arbitrary topological surfaces" *Computer-Aided Design* 10(6), pp. 350-355, November 1978.

□ [CHEN]
Chen, M. and Mountford, J. "A study in interactive 3D rotation using 2D control devices". In *Proceedings of ACM Siggraph '88*, pp. 121-129, Addison-Wesley, 1998.

□ [CHOW]
Chow, M. "Optimized geometry compression for real-time rendering". In *Proceedings of IEEE Visualization '97*. pp. 346–354. 1997.

□ [CLAY245]
Lewis, T. "Clayworks version 2.45". http://www.clayworks3D.com/, 1992.

□ [CRIT1]
"RenderWare Game Engine" *http://www.criterion.com*

□ [DISCREET1]
"3D Studio Max" Modelling Software *http://www.discreet.com*

□ [DOBA2]

Dobashi, Y., Nishita, T., Yamashita, H., Okita, T. "Modelling of Clouds from Satellite Images Using Meta-ball", *Proceedings of the 4th Pacific Conference*, pp.53-60, 1998. IEEE Computer Society

□ [DOBA]
Dobashi, Y., Nishita, T., Okita, T. "Animation of Clouds Using Cellular Automation" *Proceedings of CGIM99*, 1999.

□ [DOBA3]
Dobashi, Y., Nishita, T., Yamashita, H., Okita, T. "A simple, Efficient Method for realistic Animation of Clouds". *Proceedings of the 27th annual conference on Computer graphics and interactive techniques,* pp. 19–28, 2000, ISBN:1581132085. ACM Press/Addison-Wesley Publishing Co.

□ [DSE1]
Ebert, D.S., Carlson, W.E., Parent, R.E. "Solid spaces and Inverse Particle Systems for Controlling the Animation of Gases and Fluids", *The Visual Computer*, 10, pp. 471-483, 1990.

□ [DSE2]
Ebert, D.S., "Volumetric Modelling with Implicit Functions: A Cloud is born" *Visual Proceedings of SIGGRAPH '97*, pp. 147, 1997.

□ [DSE3]
Ebert, D.S., "Simulating Nature: From Theory to Application" Course Note #26 of *SIGGRAPH '99*, pp. 5.1-5.52, 1999.

□ [DUCH]
Duchaineauy, M, Wollinshy, M. "ROAMing Terrain: Real-Time Optimally Adapting Meshes", *IEEE Visualization Proceedings,* pp. 81-88, 1997.

□ [EVANS]
Evans, K.B., Tanner, P.P. and Wein, M. "Tablet-based valuators that provide one, two or three degrees of freedom". In *Proceedings of ACM Siggraph '81*, volume 15, pp. 91-97. Addison-Wesley, August 1981.

21 [EMRK]
Maarten J. G. M. van Emmerik, "A direct manipulation technique for specifying 3D object transformations with a 2D input device", *Computer Graphics Forum*, v.9 n.4, pp.355-361, Dec. 1990.

22 [FOST]
Foster, N., Metaxas, D., "Modelling the Motion of a Hot, Turbulent Gas", *Proceedings Of SIGGRAPH* '97, pp. 181-188, 1997.

23 [GMMTK]
Halheiros, M.d.G., Fernandes, F.N, Shin-Ting, W. "MTK: A Direct 3D manipulation Toolkit". *14th Spring Conference on Computer Graphics.*pp. pp. 81-88, 1998. Comenius University, Bratislava

24 [GRIM]
Grim, C, Pugmore, D., Bloomenthal, M. "Visual interfaces for solids Modelling", *Proceedings of the 8th annual ACM symposium on User interface and software technology*, Pittsburgh, Pennsylvania, United States, pp. 51 – 60, 1995.

25 [GARD]
G.Y. Gardner "Visual Simulation of Clouds", *Computer Graphics*, Vol. 19, No 3, pp. 279-303, 1985. ACM Press New York

26 [HOP1]
Hoppe, H. "Progressive meshes". *ACM SIGGRAPH* 96', pp. 99-108, 1996.

27 [IGSH]
Igarashi, T., Matsuoka, S., Tanaka, H. "Teddy: A Sketching Interface for 3D Freefrom Design", *ACM SIGGRAPH* 99', pp. 409-416, 1999.

28 [JARK]
Rossignac, J., "Edgebreaker: Connectivity compression for triangle meshes". *IEEE Trans. Vis. Comput. Graphics 5*, 1 (January–March), pp. 47–61, 1999.

29 [JDKH]
Döllner, J., Hinrichs, K. "Interactive, Animated 3D Widgets", *IEEE Proceedings Computer Graphics International 1998*, pp. 278 - 286, 1998.

30 [KAJI]
Kajiya, J. T., Herzen, B. P. V. "Ray-Tracing Volume Densities", *Computer Graphics*, Vol. 18, No. 3, pp. 165-174, 1984. ACM Press New York

31 [KARC1]
Karczmarczuk, J., "Generating Power of Lazy Semantics", *Theoretical Computer Science 187*, pp. 203-219, 1997. Elsevier Science Publishers Ltd. Essex, UK.

32 [KETT]
Kettner, L. "Theoretical foundations of 3d metaphors". In *Workshop on the Challenges of 3D interaction* at the CHI'94, February 1994.

33 [KIKU]
Kikuchi, T, Muraoka, K., and Chiba, N. "Visual simulation of cumulonimbus clouds", *The Journal of the Institute of Image Electronics and Electronics Engineers of Japan*, Vol. 27, No. 4, pp. 317-326, 1998 (*In Japanese*).

34 [KRUS]
Kruszewski, P. "An algorithm for sculpting trees", *Computers & Graphics*, Volume 23, Issue 5, pp. 739-749, 1999, Elsevier Ltd.

35 [LEWS]
T. Lewis and M. W. Jones, "A System for the Non-Linear Modelling of Deformable Procedural Shapes", In *The Journal of WSCG 12(2)*, pp. 253-260, 2004. [ISSN 1213-6972].

36 [LIND]
Lindenmayer, A. Prusinkiewicz, P., "The algorithmic beauty of plants", *Springer-Verlag New York, Inc.,* NY, 1990, ISBN 387972978.

37 [LITW]
"LightWave Modelling Software" *http://www.newtek.com*

38 [LITH1]
"LithTech game engine." *http://www.touchdownentertainment.com/*

39 [LOVE]
Lovejoy, S. and Mandelbrot, B.B., "Fractal Properties of Rain, and a Fractal Model", pp. 209-232, 1985, Tellus.

40 [LYCH]
Lynch, D.K., "Step Brightness Changes of Distant Mountain Ridges and Their Perception Applied Optics", 30:24, pp. 3508-3513., August 20, 1991.

41 [MAYA1]
"Alias/WaveFront Maya. 3D Modelling Software" *http://www.alias.com*

42 [MAND]
Mandelbrot, B.B., "The Fractal Geometry of Nature", New York, *W. H. Freeman and Co.,* 1982, ISBN 0716711869.

43 [MAN2]
Mandelbrot, B.B., "Fractal landscapes without creases and with rivers", in *The Science of Fractal Images*, H.O. Peitgen and D. Saupe, Editor, Springer-Verlag, New York, pp. 243-260., 1988.

44 [MAXN]
Nelson. M., "Vectorized Procedural Models for Natural Terrain", *Computer Graphics*, 15:3, pp. 317-324, 1981. ACM Press New York

45 [MIYA]
Miyazaki, R., Yoshida, S., Dobashi, Y. Nishita, T. "A Method for Modelling Clouds based on Atmospheric Fluid Dynamics", *Pacific graphics*, 2001. IEEE Computer Society Washington

46 [MILL]
Miller, G.S.P., "The Definition and Rendering of Terrain Maps", *Computer Graphics*, 20:4, pp. 39-48, 1986. ACM Press New York.

47 [MUSG]
Musgrave, F. K, "Method for Realistic Landscape Imaging", *Yale University*, PhD Thesis, 1990.

48 [MSHLSL]
"Microsoft High Level Shader Language."

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnhlsl/html/shaderx2_introductionto.asp

49 [NAGE]
K. Nagel, E. Raschke, "Self-Organizing Criticality in Cloud Formation?" *Physica A*, 182, pp. 519-531, 1992. ACM Press/Addison-Wesley Publishing Co.

50 [NDYN]
N. Dyn, "A butterfly subdivision scheme for surface interpolation with tension control" ACM Transactions on Graphics pp. 160-169, 1990, ACM Press, ISSN:0730-0301

51 [NEYR]
Neyret, F., "Qualitative Simulation of Convective Clouds formation and evolution", *Proceedings of Eurographics Computer Animation and Simulation Workshop '97*, pp. 113-124, 1997.

52 [NISH1]
Nishita, T. Shirai, T. Katsumi, T., Nakamae. E., "Display of The Earth Taking into Account Atmospheric Scattering,"
Proceedings of SIGGRAPH'93, pp. 175-182, 1993.

53 [NISH2]
Nishita, T., Dobashi, Y., Nakamae, E., "Display of Clouds Taking into Account Multiple Anisotropic Scattering and Sky Light", *Proceedings of SIGGRAPH'96*, 1996-8, pp. 379-386.

54 [NISH3]
Nishita, T., Iwasaki, H., Dobashi, Y. and Nakamae, E., "A Modelling and Rendering Method for Snow by Using Meta-balls", *Computer Graphics Forum*, Vol.16, No.3, 1997. Blackwell publishing.

55 [NISHI4]
Nishita, T., Nakamae, H. "Method of Displaying Optical Effects within Water using Accumulation Buffer", *Proceedings of SIGGRAPH'94*, pp.373-380, 1994.

56 [NLSN]
NIELSON, G.M., Olsen, D.R. jr., "Direct Manipulation Techniques for 3D Objects Using 2D Locator Devices", *Proceedings of the 1986 workshop on Interactive 3D graphics*, pp. 175 – 182, 1987.

57 [NVIDIA]
"NVIDIA website", *http://www.nvidia.com*

58 [NVIDCG]
"CG High Level Shading Language", *http://developer.NVIDIA.com/page/cg_main.html*

59 [OPENI1]
"Open Inventor Modelling language" *http://oss.sgi.com/projects/inventor/*

60 [PAMU]

Parish, Y. I. H., and Müller, P., "Procedural Modelling of cities", *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pp.301-308, August 2001.

61 [PERL1]
Perlin, K. "An Image Synthesizer", *Computer Graphics*, Vol. 19 No. 3., 1985. (also in Computer Graphics: Image Synthesis, IEEE Salem, 1988) ACM Press New York

62 [PERL2]
Perlin, K., Velho, L., Ying, L., Biermann, H., "Procedural Shape Synthesis on Subdivision Surfaces", *Proc. Symposium on Computer Graphics and Image Processing (SIBGRAPI)*, 2001.

63 [RAAR]
Ramamoorthi, R. and Arvo, J. "Creating Generative Models from Range Images", in *Computer Graphics* Proceedings, Annual Conference Series, ACM SIGGRAPH `99, pages 195-204, August 1999.

64 [ATIRM]
"RenderMonkey High Level Shader Language."
http://mirror.ati.com/developer/sdk/radeonSDK/html/Tools/RenderMonkey.html

65 [RNDM]
"RenderMan High Level Shader Language" *https://renderman.pixar.com/*

66 [SEDE]
Sederberg, T. and Parry, S., "Free-form deformation of solid geometric models." *Computer Graphics*, 20:151–160, 1986.

67 [SCHM]
Schumm, S.A., "Experimental Fluvial Geomorphology", New York, *John Wiley & Sons,* 1987.

68 [SCHP]
Schpok, J. Simons, J., Ebert, D.S. and Hansen, C "A Real-Time Cloud Modelling, Rendering, and Animation System." *Symposium on Computer Animation,* 2003. Eurographics Association Aire-la-Ville, Switzerland

69 [SGI]
"Silicon Graphics Inc." *http://www.sgi.com*

70 [SGIO]
"Silicon Graphics Inc. Optimizer Software" *http://www.sgi.com*

71 [SGIP]
"Silicon Graphics Inc. Performer Software" *http://www.sgi.com*

72 [SMTH]

Smith, C. "Theory and the Art of Communications Design". *State of the University Press,* 1997.

73 [SNIB]
Snibe, S.S., Herndon, K.P., Robbins, D.C., Brookshire, D., Conner, A.V.D., "Using Deformations to Explore 3D widgets Design." *Proceedings of the 19th annual conference on Computer graphics and interactive techniques,* pp. 351 - 352 , 1992, ISSN:0097-8930.

74 [SNYD]
Snyder, J., "Generative Modelling for Computer Graphics and CAD", *Academic Press,* 1992, ISBN 0-12-654040-3.

75 [SOFTI1]
"SoftImage 3D Modelling Software", *http://www.softimage.com*

76 [STA1]
Stam, J., Fiume, E., "Turbulent Wind Fields for Gaseous Phenomena", *Proceedings of SIGGRAPH '93,* pp.369-376, 1993. ACM Press New York

77 [STA2]
Stam, J. "Stochastic Rendering of Density Fields", *Proceedings of Graphics Interface '94,* pp. 51-58, 1994. ACM Press/Addison-Wesley Publishing.

78 [STA3]
Stam, J. "Stable Fluids", *Proceedings of SIGGRAPH '99,* pp. 121-128.

79 [STRAUS]
Straus, P.S., Carey, R. "An Object-oriented 3D graphics toolkit", In *Proceedings of the 19th annual conference on Computer graphics and interactive techniques,* pp. 341 – 349, 1992, ACM Press New York, ISSN: 00978930.

80 [STROUS]
B. Stroustrup, *The Design and Evolution of C++.* 1994, Addison Wesley, Reading, MA.

81 [SWEENY1]
Sweeny, T. "Unreal game engine", *http://www.epicgames.com/UnrealEngineNews.html*

82 [TSND]
Tesendorfm J. "Simulating Ocean Water", *SIGGRAPH 2001 course notes.* ACM Press New York.

83 [UNRH]
Unruh, E. "Prime number computation," ANSI X3J16-94-0075/ISO WG21-462.

84 [XEIOS1]
Krister, W., Tedder, M., Lewis, T. et.al., "Xeios Rendering Engine." 2000.

85 [ZELE]

Zeleznik, R.C., Herndon K.P., Robbins D.C., Nuang, N., Meyer T., Parker N., Huges, J.F., "An interactive 3D toolkit for Constructing 3D widgets.", *Proceedings of the 20th annual conference on Computer graphics and interactive techniques,* pp. 81 - 84, 1993, Association for Computing Machinery, ISBN:0897916018.

86 [ZORN]
Zorin, D., P. Schröder, and W. Sweldens. "Interpolating Subdivision for Meshes with Arbitrary Topology." Siggraph '96. pp. 189–192. ACM Press New York.