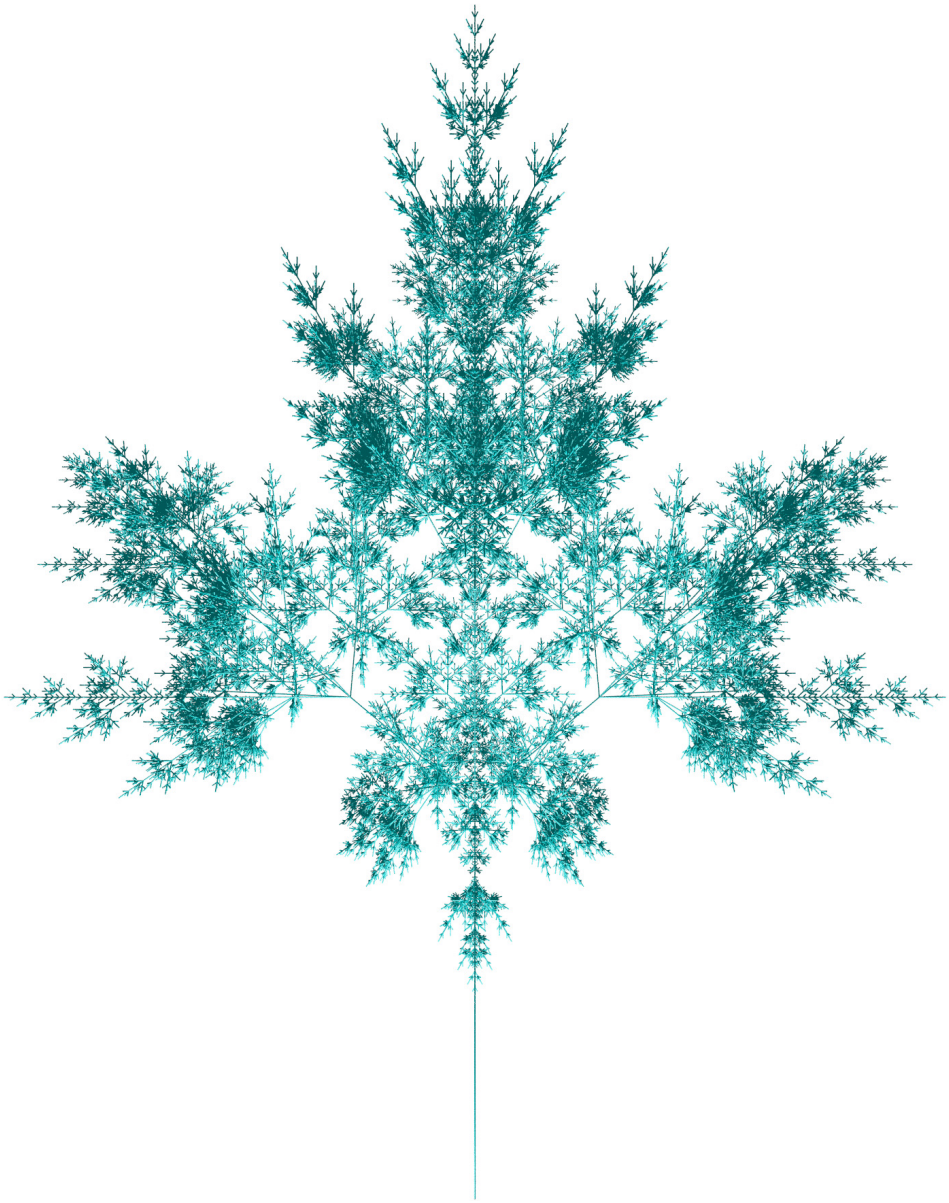
Unifying Procedural Graphics

MSci Individual Project Report

David Birch



Supervisor: Prof. Duncan F Gillies

Second marker: Dr. Andrew Davison

Imperial College London

June 18, 2009

Project presented in fulfillment of the Master in Science (MSci) degree in Joint Mathematics Computing at Imperial College London. The material presented in this project is the authors own, except where it appears with attribution to others.

Abstract

Modern graphics scenes are complex requiring huge volumes of content to create compelling visual effects. This volume increasingly exceeds current content creation, storage and delivery mechanisms.

One solution is procedural or algorithmic graphics which can be executed to generate content on demand. However these algorithms are hard to create - requiring either the artist knowing how to write code or the programmer to be an artist!

A large number of procedural graphics techniques have been developed, each with success in its own domain. Unfortunately each formalism currently has to be implemented in separate environments with no unified system for combining procedural graphics frameworks.

We present an easy to use, highly expressive environment for the creation of procedural graphics which draws together several types of procedural formalisms including LSystems, CSG like trees, math based modelling and graphical pipelines. The system would dovetail well with many other procedural frameworks and would map well to implementation on modern General Purpose GPUs.

Acknowledgements

I would like to thank the following people for their help throughout this project:

- Prof. Duncan Gillies. Without his superb supervision this project would not have been possible.
- Dr. Andrew Davison. For his helpful feedback on the project and its objectives.
- Sébastien Ros of `www.Evaluant.com`. For creating and releasing NCalc [20] [21] as an open source library and for feedback on the changes I made.
- My housemates, friends and family for keeping me sane during my final year.

"*My flesh and my heart may fail,*
but God is the strength of my heart
and my portion forever."
Psalm 73v26

Contents

CHAPTER 1

Introduction

Recent advances in computing mean that the amount of graphical data that modern graphics cards can process outstrips the mechanisms to deliver it to the graphics card. This is true in terms of physical bandwidth constraints and storage media capacities. For example a modern console game now tries to compressed upto 100Gb of data onto an 8.5Gb DVD. The other side to the problem is that content creation is exceedingly expensive with large teams of artists working to produce models and graphical scenes individually.

The major solution to this problem has been the rise of procedurally generated content whereby an algorithm is crafted which generates the geometry required in the scenes. This approach has met with success in many areas including grass and vegetation generation but also in more surprising areas such as real time city generation [31] [32].

However there are two major problems with procedural graphics which have yet to be addressed:

Firstly that each algorithm needs to be hand coded, for example in C, by an experienced programmer who is not by trade an artist, this creates a skills mismatch which if avoided could raise quality, productivity and reduce the cost of producing such content.

Secondly in each area that procedural graphics has been applied to a separate formalism for procedural modelling has arisen, for example Lindenmayer Systems [17] for use in modelling vegetation and Shape Grammars for use in City Modelling. This is unfortunate as if used in combination these techniques could lead to even more exciting graphics for use in high end applications such as the games and movie industries.

In this report we present a new tree based procedural formalism based on the pipelining of simple graphical commands within a tree structure, and then integrate several important procedural formalisms into the language to offer an expressive unified procedural modelling language.

The language is implemented in a user friendly Integrated Development Environment (IDE) which does not require the user to be able to write code but instead to manipulate via TypeSafe drag and drop various intuitive building blocks into a scene tree.

We believe that as language presented is naturally multithreaded in its execution it would map simply to execute on the new generation of General Purpose GPUs which exploit massive parallelism.

In the time available for this project it would have been impossible to create a platform worthy of use in industry so instead we have developed a proof of concept environment. This provides an excellent prototyping environment for procedural graphics and is extensible enough to allow upgrading for use in industry and discuss in detail how the system could be extended further.

1.1 Goals

This project has two major goals:

1. To create an easy to use graphical programming language with a typesafe drag and drop interface which is simple to learn, easy to use and yet very expressive and effective for the purpose of sand boxing procedural algorithms.
2. To bring together several of the most promising procedural algorithms techniques into one unified language environment and to support animation from the get go, something that many previous procedural systems have lacked.

1.2 Contributions

We feel that the key contributions of our software are:

- An easy to use tree based graphical programming language for the investigation and prototyping of pipeline based procedural algorithms for scene generation.
- Simple management of the data amplification problem.
- Non-linear editing of models to allow all design decisions to be changed at any time.
- A system supporting animation from the start in an intuitive integrated manner.
- Multiple methods of tree based execution to suit different graphics problems.
- The unification of several procedural formalisms including:
  - Pipeline based graphics systems such as ClayWorks (section 4.3)
  - An LSystems editor.
  - Tree based build up procedural graphics systems such as Blob Tree (section 4.5)
  - Imperative language constructs such as variable, for loops and if statements.
  - An extensible mathematical expression evaluator embedded throughout the project.
- A graphically manipulatable language via TypeSafe drag and drop user interface
- The whole language is implemented in C# with a user-friendly Integrated Development Environment with the following features:
  - Full Do/Undo/Redo support with full edit history.
  - Save/Load functionality.
  - A drag and drop tool box.
  - Tree highlighting system.
  - Standardised editing system.
  - Validation of user edits to nodes.
  - Visual Debugging interface.
  - End-to-End Test Suite.
- The execution of the language is multi-threaded in a scalable way to take advantage of current and future advantages in mulitcore CPU development.
- We believe the language we describe could be pre-processed and compiled to C code for embedding into PC games giving the benefits of portability, low space overheads and high speed processing.
- We further believe that due to the massively multi-threaded nature of the language it would naturally map onto the latest general purpose graphics cards such as the NVidia Tesla architecture.

CHAPTER 2

Motivation

2.1 The Current State of Affairs

Within this chapter we aim to give an overview of the context into which this project falls, dicussing the current generation of modelling software, its benefits, pitfalls and several techniques which could provide solutions to some of these problems. Current modelling software can be broadly divided into two groups:

2.1.1 Interactive Modellers

These systems rely on complex interfaces with a huge variety of different tools which an artist may use interactively to edit a scene that they slowly build up one piece at a time. These modellers allow users to edit their model interactively in real time through the use of various small tools. These systems are the industry standard with Maya[10] and 3d Studio Max[11] leading the pack. Such systems normally support an entire ecosystem of graphical modelling tools, as each system is extensible through a plugin based tool system. This allows leveraging of existing technology through fully featured third party tools. The use of internal scripting languages such as Python also helps make these systems more extensible and usable for expert users. The primary drawback of these systems is the complexity of the interface which is due to the sheer number of different tools and features available. A non-trivial subset of which must be mastered before a user can become competent enough to produce good models.

2.1.2 Scripted Modellers

The alternative to interactive modellers is scripted modellers, these systems are built in the main of three main components:

1. A scripting language in which the scene is defined.
2. A compiler/interpreter which compiles then executes the scripting language to produce a representation of the scene in terms of graphical primitives such as lines or triangles.
3. A viewer which displays the scene from the primitive list.

Often for convenience the scripting language is written in an Integrated Development Environment (IDE), which also serves as a compiler/interpreter, saving the user time in not having to switch between tools. The IDE will often give the user detailed feedback on any problems with their script. The IDE may also contain a viewer / renderer for the generated scene.

The biggest problem with scripted modellers is the mis-match in skills between the creators of the system (programmers) and the end users of the system (artists).

The artists are not, normally, competent programmers (nor should they be required to be!) and so often struggle to learn and fully leverage the capacities of the scripting language. This has made scripted modellers such as PoV Ray [8] find only limited wide spread usage.

Another problem with scripted modellers is the lack of interactivity. This is especially problematic for the artists using the software as it inhibits their creative process by forcing them to make a change in their script, compile the script; view changes then make another change to the script in an endless cycle until satisfied with the output.

This problem can be illustrated by considering an artist attempting to find the correct position for a 3d model in a scene, they pick x,y,z coordinates they believe to be correct, compile the script and view the output, they then repeat this process several times until satisfied with the positioning. However each iteration of this process may take a substantial amount of time, and since it may well involve switching programs/interfaces during this time the artist may well forget the changes they are attempting to make.

2.2 Current Problems

Current modelling programs have a number of problems which confuse users, waste time and reduce productivity. Both forms of modellers suffer from several or all of the following problems:

2.2.1 Complexity of interface

As mentioned above for scripted modelling systems the primary difficulty is becoming sufficiently familiar with the programming language to exploit it successfully. Whilst interactive modellers avoid this problem, they fall foul of another problem in that because there is a huge variety of tools provided with which to interactively modelling the scene and each tool must be learned individually as it serves a separate function. The artist must then build up skill at each tool individually, then gain proficiency at using them to greater effect in combination. This complexity necessitates a steep learning curve.

2.2.2 Linear design

Within the industry standard 3d modelers Maya[10] and 3d Studio Max [11] the artist must work in a linear fashion, making a series of decisions which eventually become the final output. Should the artist wish to change one of their original decisions they must either undo all work subsequent to that decision, or else attempt to repair or rework their model using correctional tools which have the opposite effect to the tool they originally used, however use of such tools often has adverse effects on the rest of the model they have created so far.

Whilst scripted systems do not suffer so greatly from this problem, it is still clear that a modification to one of the early decisions could require refactoring of the script. This takes considerable time, effort and ingenuity to achieve without unintended side effects. This problem also shows itself in a different manner in both types of system, as can be illustrated in the following problem and example:

2.2.3 Lack of reuse

Another problem with the current generation of interactive modelling software is that they do not allow the user to easily reuse the work they have put into creating a model. For example should the user wish to create a model of a city, they would need to create models of different buildings. The initial process of creating a building, defining a cuboid shape, giving it the required size and position, then adding windows, doors and a roof will be common to every building, and yet the user is forced to laboriously repeat these steps each time they wish to

create a new building with minor variation, or otherwise to copy a standardised base building again and again with no variation.

This issue can be generalised by noting that a user will often use a similar set of tools in a similar order to produce a particular effect. In order to repeatedly use the new effect the user must repetitively repeat the same series of tools that compose the whole every time. This is not only unproductive, but also limits the power of the user to create highly complex scenes.

This problem is also magnified as the user is forced to use a linear design process in that if they decide to modify one of the initial tools used in production of the larger effect they must not only undo all the effects after that point but in order to change every application of this larger effect they must undo all the work back to the first time they carried out this larger effect.

It is useful to note that scripted design suffers less from a lack of reuse as scripts can often be copied and pasted with small modifications quite easily. However it should be noted that most portions of scripts are context dependent and cannot be reused in another model or scene without significant refactoring / rewriting.

2.2.4 Complexity

In most interactive modellers the complexity of a model is a function of the designers skill and the amount of time they put into the design. That is to say that as the model becomes ever more complex it becomes prohibitively complex and time consuming to continue to edit the scene. This issue is to some extent alleviated by the tools provided by a given IDE, but the issue remains that beyond a certain point a complex scene cannot be edited by the continuous interactive application of small tools, some use must be made of scripted or procedural graphics:

2.3 Procedural Graphics

Procedural graphics is a term given to a collection of techniques used within graphical modelling which utilise the construction of algorithmic or mathematical descriptions of the models they create. They may be contrasted with more the traditional design tools which create models by the interactive application of a plethora of small editing tools in a sequential manner. Examples of procedural graphics include L-Systems as described in section 2.6, scripted ray tracers such as POV-Ray as discussed in section 4.2 and other systems discussed within the related work chapter (4). These procedural graphics systems have the following key features:

2.3.1 Similar models

Since procedural graphics utilise algorithmic or mathematical descriptions of the models they create, it is comparitively easy to generate a collection of similar models by tweaking small sections of the algorithm. For example should a forest of trees be required, some randomisation could be introduced into an initial tree model allowing many similar but not identical models to be created. Were the same effect to be attempted using traditional techniques an artist would have to tediously spend large amounts of time designing many different trees from scratch, the effect normally being far less effective.

2.3.2 Complexity of models

As algorithms are extremely flexible and contain a large amount of information in a very concise format, the complexity of models generated through procedural modelling can be far higher than those made through traditional interactive modellers, where the complexity of the model is constrained by the artists time and skill. Also some forms of modelling can only be done via procedural modelling, an example would be that of modelling fractals or other mathematical phenomena.

2.3.3 Small storage requirement

One benefit of procedural modelling is that since the scripts are concise yet powerful they have a very small storage overhead, which is critical in fields such as internet based graphics, and in the games industry where increasingly huge volumes of data are crammed onto a finitely sized CD/DVD. Of course in order to realise this benefit the programs must be able to read, execute and render the procedural descriptions of these scenes.

2.3.4 Lack of Interactivity

One complaint often levelled at procedural modelling is that they are often non-interactive. By this we mean that since the model is described in some language there is a level of indirection between editing this script and viewing the results on screen. This write-view-rewrite methodology can seem restrictive to those used to a more interactive hands on approach to interactive modelling. Unfortunately it is hard to couple the on screen content sufficiently closely with the scripting language that users can edit the script via manipulating the rendered display.

2.3.5 Hard to Write

It must also be remembered that many of the conceptual frameworks within which procedural modelling takes place are non-trivial to comprehend and learn. For example the concepts of L-Systems and Turtle based graphics are somewhat complicated to grasp at first. It is also clear that without a good grasp of the theory of the modelling constructs, the user is unlikely to be able to take full advantage of the complexity the modelling language provides. This gives rise to an often high learning curve within these types of modellers which is often sufficient to put off non-programmers and artists from using them. This is an area we would hope to address within this project.

2.3.6 Rendering

One problem encountered with procedural modelling techniques is that although they create very interesting models it is hard to then reuse these models in everyday applications such as games as they require specialised rendering techniques such as Ray-Tracing. These methods are often too slow to render in real time and can be prohibitively complex in their implementation, especially if specialised scripts must be read and executed in order to be rendered. Exporting from procedural modellers to more standard polygon based rendering formats does go some way to resolve these problems, although fidelity and the storage space benefits of procedural modelling are lost in the process.

2.4 Geometry Generators

A geometry generator is a piece of programmable hardware or a software program which allows for the procedural generation of geometric primitives. The advantage of using a geometry generator is that it avoids having to store great lists of geometric primitives, but instead allows only the initial inputs and the program for the generator to run to be stored. A further advantage is that since there is less data being stored, there will be less bus traffic between the graphics card and other system devices, this is advantageous as this bus can become a bottleneck as other system devices such as hard disks and DVD drives fail to provide the throughput needed to feed the GPU with sufficient textures and other rendering information to fully leverage its power. Use of geometry generators is likely to become increasingly common as the graphics card becomes increasingly user programmable through the use of languages such as NVidia Cuda [28] or the geometry generation stage of the new DirectX 10 graphics API [22].

2.5 Pipelining

> "I expect that by the time of the release of the next generation of consoles, around 2012 when Microsoft comes out with the successor of the Xbox 360 and Sony comes out with the successor of the PlayStation 3, games will be running 100% on based software pipelines"
>
> **Tim Sweeney, creator of the Unreal game engine and CEO of Epic Games**[4]

Pipelining is the technique of taking a series of simple commands and composing them to create a more complex command. It is a powerful technique for managing data flow. Traditionally the graphics pipelines have a number of steps each of which runs on highly specialised hardware. Several parts of the traditional graphics pipeline have become user programmable allowing the creation of powerful graphics algorithms, which must unfortunately be written in a specialised scripting language.

2.5.1 Hardware Convergence

Recent trends within the computer hardware market have seen a convergence between the CPU and GPU. The graphics processor is increasingly becoming general purpose by incorporating an ever wider set of instructions hence it is able to be used for ever more complex applications such as Magnetic Resonance Imaging [12] & Weather Prediction [13]. This generality is increasingly being exploited in computer graphics leaving the traditional hardware based rendering pipeline increasingly obsolete, as more and more parts of it are replaced with customisable code, written in specialised languages running directly upon graphics hardware. This was originally achieved through the use of programmable shader languages such as Renderman [15]. However these languages, whilst introducing flexibility into rendering, were limited both in their functionality and their size. They were also limited by the portion of the graphical pipeline they could act upon or replace. The trend for generality has continued to the point where NVidia's new Tesla architecture graphics cards are now capable of running a language called Cuda - an extension to the C programming language which affords the programmer great leverage over a floating point computation platform vastly superior to most general purpose CPU's [14].

At the same time we have seen the CPU feature set become increasingly similar to that of a GPU. They have an increasing number of processing cores and the addition of SIMD (Single Instruction, Multiple Data) instruction sets such as SSE SSE2 & SSE3 which emulate instructions available on a GPU for dealing with large amounts of vector data. This trend looks set to continue with AMD's forthcoming introduction of a SSE5 instruction set containing native vector compare and test instructions[1].

This trend away from constrictive hardware based graphical pipelines toward more general purpose software based pipelines running on the GPU/CPU provides ample opportunity for research into the best means of writing functional software in which these pipelines may be specified, and controlled. This hardware is also well suited to procedural graphics as it allows large scale general purpose computing which can run custom algorithms for producing sufficient graphical content.

We note also that the increase in the number of threads CPUs and GPU's can run simultaneously has increased markedly this is useful for running pipelines as each can run an instance of the same pipeline and process a portion of the data, or each thread can run a different pipeline allowing more complexity in the scene.

We will now explore the advantages that a customisable rendering pipelines affords:

2.5.2 Advantages

As stated a pipeline is a series of small operations chained together such that data is passed into the first operator, upon which the output of this operator is piped into the input of the next operator in the chain and so on. The advantages of pipelining can be summarised as follows:

1. **Speed of computation:** In most pipelines the operations which are chained together are simple and quick to execute, it is the chaining of multiple small commands together which gives the pipeline its power. The speed of execution of many small operations may outstrip trying to execute a single larger more complex operation which achieves the same effect, especially if they map well to the hardware they run upon. Pipelines lend themselves well to multithreading as one of the following methods can be utilised:

   - **Batch Based:** Each operation in the pipeline is executed by a different thread. This approach is best used if one thread is running on hardware specialised to do a particular task, for example vector operations. It could also be necessary if the operation must operate on all the data passing through in one go. For example when scaling all primitives in a model by the centre of the model.
   - **Stream Based:** Many threads execute the entire pipeline each taking a proportion of the data to be passed through the pipeline. This approach is best used when multiple threads are running on the same hardware. Stream based processing is also useful when the operations on all primitives are independent and thus can then be processed by different processors.

   It is interesting to note that whilst the first approach is taken in hardware pipelines on graphics cards, the second approach is becoming increasingly popular as it leverages the high number of processing cores available on graphics cards, some of which support up to 240 separate processing engines.

2. **Limited storage:** When using pipelining one need only store the program for the pipeline and the input data and not the resulting data which the pipeline produces, this helps combat the explosion in data which occurs when rendering complex graphical scenes. This is known as the *data amplification problem*.

3. **Complexity management** Since there is limited data storage and only the pipeline and the initial inputs to it must be created by the user there is a smaller amount of data and thus complexity which must be stored and maintained by the user. Providing of course that the pipeline can be executed in a short enough space of time to be of use, otherwise it would have to be precomputed and stored in a verbose format negating the space advantages of procedural graphics.

2.6 LSystems

Lindenmayer Systems[17] or LSystems are a simple and powerful form of procedural graphics which have been successfully used within an industrial settings for the modelling of self similar structures such as plants, bushes and trees, as well as the modelling of mathematical structures such as fractals. LSystems consist of the three major elements:

1. An **alphabet**, (normally single letters A,B,C...) which represent single graphical primitives or commands. These characters can be formed into strings or words such as [ABCDEF]

2. A series of rewriting rules or **productions**, which map from a pattern of consecutive characters to a new word. For example "A $\rightarrow$ AB". Upon execution of the rule every occurrence of the left-hand side of the production is replaced with the right-hand side of the production.

3. An initial word upon which the rewriting rules are iteratively applied or *evolved*. This is known as the "axiom".

For example if the production "A $\rightarrow$ BC" was applied to the axiom ABC the result would be BCBC.

The result of evolving the LSystem is rendered in a variety of ways; the most common is through the use of a "Turtle" [16], whereby each character in the alphabet represents a command to an imaginary robot, such as "Pen Down", "Pen Up", "Move forward" or "Turn Left". Of course these commands can be parameterised by colours, angles and distances in pixels, which can again be represented by other words in the language.

The productions in an LSystem need not be simple, for example imagine the production $A*BC \rightarrow ABC$ where $A*$ represents zero or more consecutive A's. Many techniques from *regular expressions* can be used in this pattern matching, indeed LSystems and regular expressions share their genesis in research surrounding formal grammars. It may be noted that if several productions match a given sequence of characters then the LSystem is then probabilistic and may have weighted probabilities attached to each matching production. LSystems of this form are known as *Stochastic* LSystems.

The use of *bracketed* LSystems where "[" and "]" are characters in the alphabet, is quite interesting as they allow branching when given the follwing semantics: Whenever a [is encountered the turtle should save its current state continue executing commands until a] is encountered when the turtle should resume its previous state before continuing to execute any remaining commands.

Figure 2.1: A 2d LSystem generated from Haskell code written as a first year lab exercise at Imperial College. [29]

Thus we see that the power of LSystems for graphics is that they are extremely compact to store and reasonably fast to expand, although it is clear that the linear execution of an LSystem in terms of a turtle is quite slow for large LSystems. LSystems do however produce some amazing graphics, for example the image on the cover of this report.

CHAPTER 3

Our Approach

Within this section we intend to cover the main design decisions of the system we present and to give reasons for and explore alternatives to the design decisions we have taken and features we have decided to implement.

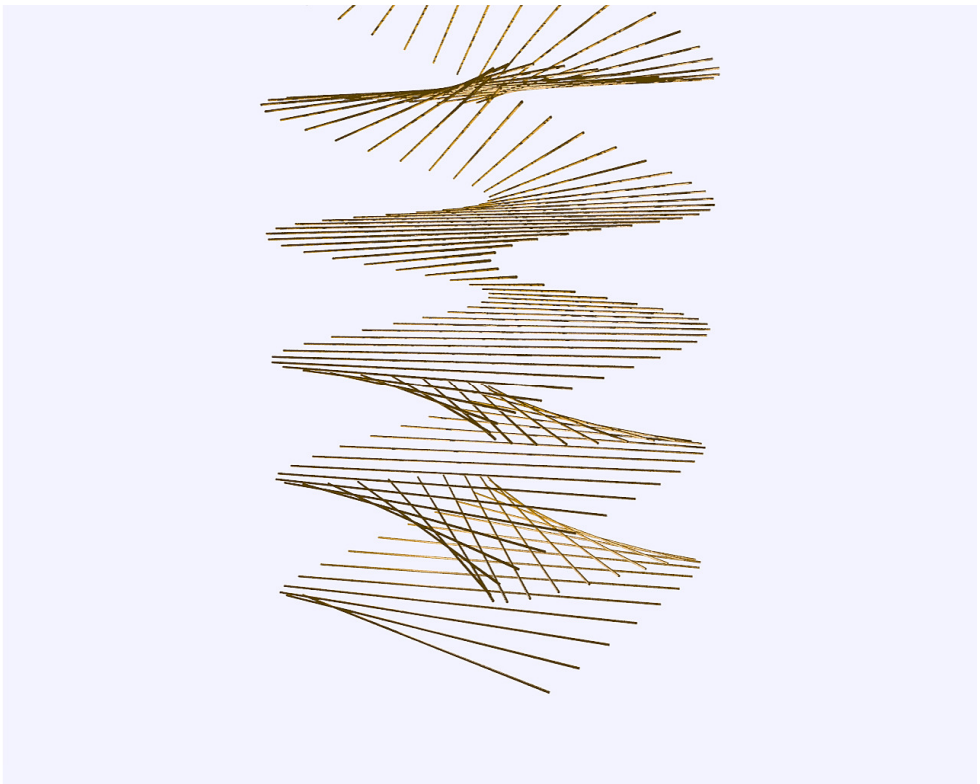


Figure 3.1: A POV-Ray rendering of a helix animation originally created in about six minutes of prototyping in the application.

3.1 Tree Based Execution

In this project we have used a classical tree structure to contain our language. Whilst the use of tree structures in graphical programming is not new, the approach we have taken gives additional power to this construct.

The traditional use of trees within graphics systems (see BlobTree 4.5 and HyperFun 4.4) is to represent a graphical scene by the root node of a tree. The scene is then created by the recursive composition of the evaluated child nodes of the root node. This can be thought of as a "building up" of the graphical scene from the bottom of the tree to the top (root). An illustrated example may be seen in figure 3.3. It is an intuitive and powerful method for building up a single model from smaller, simpler constituent parts.

However as noted in several past systems this "building up" semantics is unable to cope with large models nor, although several optimization schemes have been proposed [5], can this scheme render in real time sufficiently quickly to enable interactive modeling by the system user beyond more than half a dozen tree layers. This method of tree execution is also unable to produce multiple models in order to create a full graphical scene. For these reasons we created a new form of tree based graphical language:

3.2 Pipelined Execution

The approach we offer within this report is to use the branches of a tree to form pipelines through which graphical primitives stored in the leaf nodes of the tree are passed. Pipelines will often have common first sections, branching into different pipelines only when a new branch in the tree is encountered. Thus as various branches in the tree fan out so will the number of pipelines. In this system data may be thought of as flowing from root of the tree down toward the leaves of the tree, the opposite of the system described in the previous paragraph, although the graphical primitives are still stored in the leaf nodes of tree. An example scene tree is shown in figure 3.2, the tree contains a number of modifiers nodes and three groupings of primitives. Primitives with the same parent node will form the input to a pipeline. In the diagram you can see the tree pipelines generated all of which share a common first five nodes allowing for global control of the form. In this case to mirror the scene on the X, Y, $X = Y$ and $X = -Y$ planes. Further rotations are then applied to produce animation in the scene (which produces a star shape which is rotated slowly.)

3.3 Buildup Execution

We also implement the traditional "Buildup" method of execution in our system since it is an extremely good way of creating models for use as a part of graphical scenes and will be familiar to some users. We implement this via the use of a control node which changes the way the subtree it roots is executed. The reason for including these two types of processing is that some parts of a model are best built up from a small number of constructs into larger easier to work with building blocks, these models are then best built up into a scene by use of pipelines.

A Buildup subtree would consist of a tree of modifier nodes with a number of graphical primitives arrayed as the leaves of the tree, which is identical to the layout of a pipelined scene tree. The Buildup tree is executed by forming pipelines which flow from the leaves of the tree back up to the root of the subtree - the opposite of the pipeline based execution which is the default means of execution for the scene trees. The pipelines generated from this formalism will share a common end point but will have different start points. The execution algorithm which we detail in this report is generic enough to be able to execute both semantics for tree execution. This has allowed the benefits of multi-threaded execution to be shared by both formalisms.

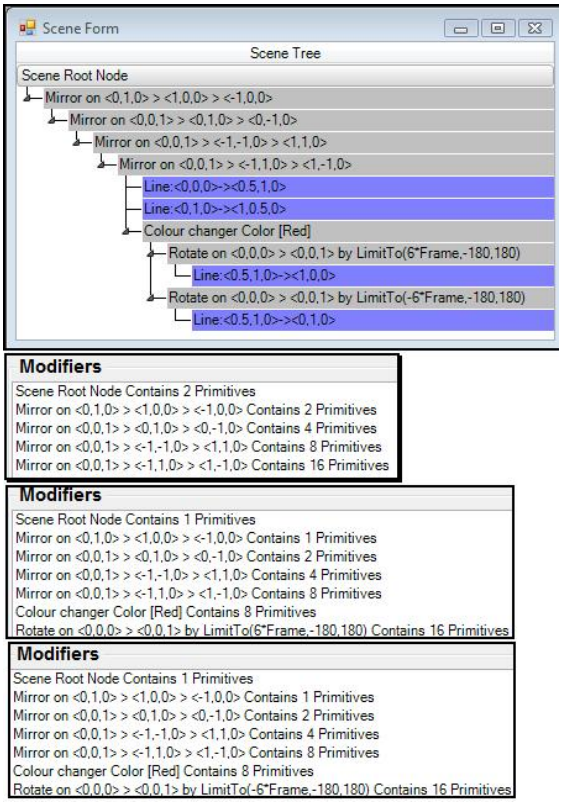


Figure 3.2: A sample scene tree with modifier nodes in gray and graphical primitives at the leaves of the tree in blue. The three pipelines generated by the three groups of primitives are shown also.

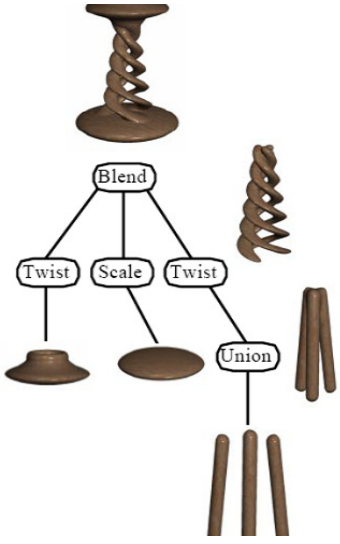


Figure 3.3: A sample tree structure representing a Blob Tree as described in section 4.5 showing the existing approach to tree based modelling with data flowing upward from the leaf nodes. [7]

3.4 Modifier Execution

In addition to the pipelining and "build up" execution, we also implemented stream-based and batch based execution of the pipelines:

In stream based processing each primitive may be processed by every node in the pipeline before the next primitive is processed. This allows multiple processing threads to execute, each processing a portion of the data to be passed through the pipeline thus reducing the execution time to approximately $\frac{1}{\text{number of threads}}$. It should be clear that where within a pipeline there are a number of consecutive stream based nodes then these can be executed as one "group" of nodes, via this multi threaded system.

Batch based execution of a node means that every graphical primitive being passed through the pipeline should be processed by one node before being passed to the next node in the pipeline, although this can prevent multithreading (depending on the nature of the processing being carried out) it does allow more powerful operations to take place, for example finding the longest line being passed through the pipeline. It should be noted that some batch based operations could be multi-threaded by assigning a portion of the work at that node in to each thread.

3.5 Unification

One of the major ideas missing from the related work in this field was any attempt to unify the variety of different procedural formalisms, we have gone someway to tackle this both by the two forms of tree based semantics and also by integrating several other forms of procedural graphics:

Throughout the project we have embedded a fully featured mathematical scripting language allowing the user some interesting possibilities for controlling the generation and positioning of geometry. The user is able to set the rotation of nodes to be $\exp(0.51)$ if they wish. This allows for example the rendering of a harmonic wave on a row of lines. This system is also the driving force behind the built in animation system we use as a "Frame" variable is available throughout the tree to allow the user to animate object by moving them or changing their geometry based on the frame number in the animation.

We have implemented a parameterised LSystems editor which allows the user to generate tree and vegetation like geometry for use in scenes. The LSystems implemented allow the user to use mathematical expressions to set the parameters of the various commands which are used to render the LSystems. The LSystems are fully integrated into the project as a form of graphical primitives and thus can transit any pipelines produced in the normal way, allowing for example the modification of a tree by rotation or translation and so forth.

Unlike any other pipeline based framework we have seen we also implement a number of constructs more normally found in imperative programming languages such as For loops which allow repeating sections of the tree and If tests which allow the truncation of a pipeline. We also implement various filter nodes to filter primitives through the pipelines and nodes to control pipeline flow for example to allow the branching and union of pipelines.

To allow the user to select content to use we support a Tagging system for primitives which allows the selective application of modifier nodes based on the primitives Tag field and a mathematical expression or Tag Test field which is evaluated at application time. Further possibilities for primitive selection are discussed in the Further Work chapter.

Having discussed our approach we now answer some common questions and discuss other design possibilities:

3.6 Why not use a Directed Graph?

Given that we are using pipelines embedded in a tree based structure, we should consider: why if the goal of this project is to investigate inter-related pipelined structures, should a Tree structure and not a directed graph be used as the container of the language?

The justification of the approach we have taken is threefold:

1. The construction of a general directed graph with free form layout is non-intuitive to construct and work with as the user spends alot of time rearranging objects on their workspace to make things look "just right" or to make space to add nodes. On the other hand tree is by its very nature structured and the flow of data through the pipelines from root to leaves is more clearly seen, and much more easily edited. It also hopefully easier to understood and more usable as the user has a clearer mental model of the dataflow and does not have to spend time re-arranging and arbitrary graph layout. We also feel that a tree structure is far more scalable since it is more compact in screen space and does not require the user to manage the layout of a graph structure.

2. Although a tree structure is more limited in structure than a directed graph since for example a tree structure does not contain loops or multiple roots, we have implemented a number of constructs that give the tree structure similar expressive power to a directed graph. The features we have implemented to restore expressive power include:

   - **For Loops** - these allow a list of nodes to be repeated a number of times, thus allowing the tree to contain loops.
   - **Splitter Nodes** - these allow pipelines to share some or all of their data by splitting the list of primitives they collectively contain amongst the pipelines. This is equivalent to a node in a graph having multiple inputs.
   - **Primitive Generators** - these allow primitives to be "injected" at any point in a pipeline, thus allowing multiply rooted trees, especially if used with the "build up" semantics provided.

   Thus the tree structure we present has similar expressive power as a directed graph of various pipeline-able modifier nodes. By retaining the layout of a tree we retain the simple and intuitive mental model of data flowing from top to bottom within the tree. Full details of these nodes together with how they effect data flow can be found in chapter 7.

3. The use of a tree structure provides very useful facilities for managing the data amplification problem experienced when we attempt to generate sufficient geometry to construct a scene detailed enough to fully utilize the features of modern graphics hardware. This is because the branching of the tree is reflected in the branching of the pipelines; this branching allows data to be amplified in an intuitive and easy to manage manner producing highly complex scenes.

3.7 Why not just write a scripting language?

An approach taken by many of the schemes reviewed in the related work section of this report is to implement a scripting language which is written in a standard text editor then parsed, compiled/run and then rendered. Whilst this approach is arguably more efficient for the advanced user and can be more powerful through the unconstrained programming it allows, it does have some notable limitations when applied in the context of this project.

The primary reason we decided not to implement our language as a scripting language is the challenge of representing pipelines and pipeline-able elements in an imperative scripting language, especially one where pipelines may interrelate by sharing a common beginning, merging and splitting or indeed looping a finite number of times. It is hard to imagine a text based formalism which could intuitively and descriptively model such data structures. Were such an approach to be taken there would inevitably be a large and steep learning curve for new users to the system to overcome. This is especially unwarranted considering that graphical systems are often aimed at artists who are not, on the whole, expert programmers. We believe that through the use of a tree based programming formalism we have gained a sufficiently powerful formalism

which more than compensates for forgoing the complete freedom of a text based scripting language provides, without incurring additional overheads and complexity involved in a scripting language.

3.8 Why not implementation as a Plugin?

One common way of implementing new graphical functionality is to implement a plugin, or extension to a major industry standard graphical package such as Maya [10] or 3d Studio Max [11]. Whilst it is clearly an excellent idea to leverage the already existing functionality in such packages, and indeed to extend them to make them even more powerful, it was felt that this approach was inappropriate for this project for the following reasons:

1. The time required to learn the package sufficiently well to be able to effectively implement a plugin was felt to be prohibitive.
2. There is likely to be considerable difficulty in dovetailing with the API's provided within the software with the data structures and features we want to implement.
3. Plugins are often constrained by the choice of language in which they must be implemented in order to be compatible with the software to which they relate.

For these reasons we decided to implement our system as a standalone system, and allow export of resultant graphical scenes into other software packages such as POV-Ray.

CHAPTER 4

Related Work

In this section we will look at several procedural graphical systems relating to the system we present, summarising their key features and relation to this project.

4.1 LinSys3d

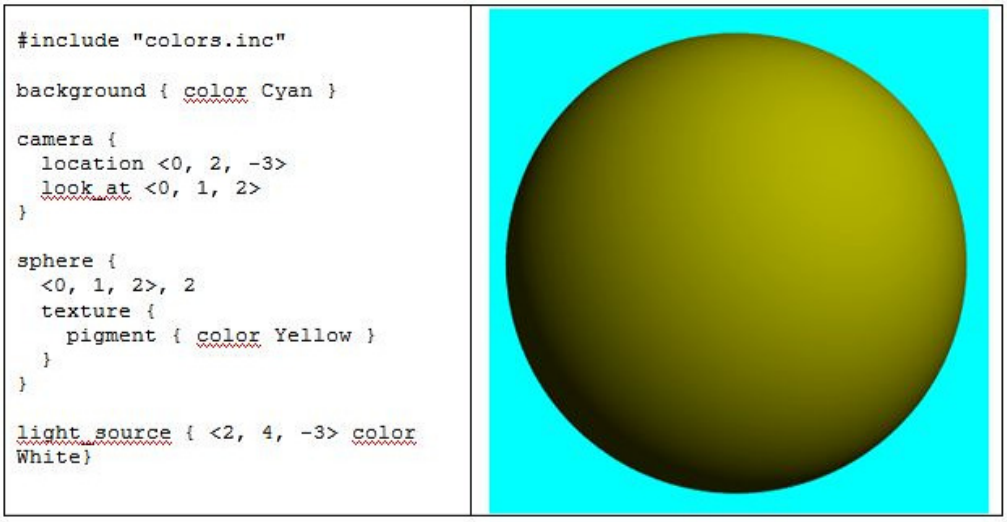
LinSys3d developed by Andrea Esuli[18] is an powerful LSystems editor. LSystems as described in section 2.6 are a very powerful formalism for procedural graphics. They consist of a series of productions to evolve a command string over a number of iterations. This command string or axiom is then translated into commands for a "turtle" which draws out a scene. LinSys3d facilitates the construction of many different types of LSystems, including:

- Stochastic LSystems where productions are given a weighted probability in the case that more than one is applicable in a given situation.
- Context sensitive LSystems whereby the productions specify a string of characters or context to appear to the left and right of the left hand side of the production before it can be applied.
- Parametric LSystems where the alphabet has a number of numeric parameters attached to each letter for use in rendering (for example the length of a line).

LinSys3d provides a structured language definition for writing these LSystems. It also provides means for export and import of 3d models for use within LSystems and in order to use the LSystem generated in other programs. This is useful although it would be more advantageous to be able to execute the LSystem on the fly within a graphics program such as a game or virtual world. One powerful feature of the software is the mapping between the alphabet and 3d objects. Objects in an LSystem can be linked either to set of parameterised 3d models, or to space transformations (eg rotate or translate the turtle position). The final alternative is to use alphabet letters as signals which have no effect on the output but help within the modelling of the system.

Overall LinSys3d is a very useful tool for the production of LSystems, providing very high functionality, although at the cost of a high initial learning curve, the lack of helpful compiler warning messages does not help in this respect especially when trying to write a new LSystem. There is also great scope for improving the user friendliness of scripting LSystems via text highlighting and help prompts so that this effective tool can be used by a wider audience.

4.2 POV-Ray



```
#include "colors.inc"

background { color Cyan }

camera {
  location <0, 2, -3>
  look_at <0, 1, 2>
}

sphere {
  <0, 1, 2>, 2
  texture {
    pigment { color Yellow }
  }
}

light_source { <2, 4, -3> color
White}
```

Figure 4.1: A simple POV-Ray script.[9]

The Persistence Of Vision Ray Tracer[8] or POV-Ray is a well known and very powerful scripted ray tracer. It uses a scene description language to describe mathematical geometry within a scene which is then rendered using the ray tracing technique. During ray tracing each pixel of the screen is represented by a ray which is projected from that screen pixel into the scene, the colour of the pixel is determined by what the ray geometrically intersects with, and what subsequent secondary rays intersect with. Secondary rays are generated when a surface is hit by the original ray, and are sent off in the directions in which light would reflect/refract in such an intersection. Pov Ray's Scene Description language is compact, simple and powerful, although there is a very high learning curve for new users. A simple example is shown in figure 4.1.

We have made strong use of PovRay within this project by allowing the export of the results of execution into a PovRay script for rendering and animation. This proved a straightforward process and was highly useful at the early stages of the project for testing purposes. In latter stages we have been able to produce video animations through using PoVRay and a BMPtoAVI conversion tool.

4.3 Clay Works

The Clay Works[3] modelling system proposed by T. Lewis and M. W. Jones is of direct interest to this project due to the innovative use of procedural modelling. We have adapted several ideas from this project into our software.

Clay Works is a fully interactive procedural modelling language. The language consists of a stictly linear chain of nodes, the first of which must be a graphical primitive such as a sphere, cylinder, toroid or polygon, followed by a chain of modifier nodes such as selection, extrude or stretch. A sample chain of nodes is shown in fig 4.2. This chain of modifiers is then executed from start to finish to produce a representation of the model which is then raytraced or polygonised for rendering. The actual creation history is formed by the interactive application of a number of modelling tools.

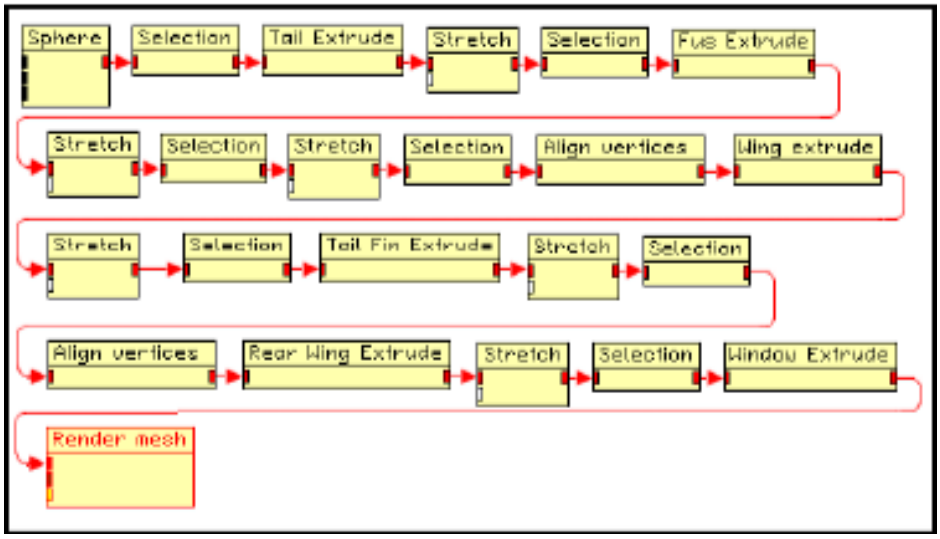


Figure 4.2: A *creation history* from Clay Works [3].

4.3.1 Creation History

This chain of nodes or creation-history, can then be edited parametrically (that is by the setting of node specific parameters) in a non-linear fashion. Such that once the artist has completed a prototype of their model they may return to any previous design decision node and change their choice of value for its parameters. This is a powerful approach and provides huge advantages to artists over having to completely undo and redo large sections of their work to make changes or else use inaccurate correction tools as would be the case in most interactive modellers.

When an early step in the model is changed the geometry that the following modifier nodes act upon will be different. In most systems this would cause the modifiers to have vastly different effects than intended or even not to work at all. However the system avoids this problem by making use of volumetric selection, that is selection as a subset of the scene volume (in xyz space) which the model occupies, the modifiers then take effect within this volume. Thus avoiding specifying the exact topological features upon which the modifier should act, which could be modified or destroyed by editing the effects of previous modifier nodes.

4.3.2 Selection Channels

Practically the selection of a volume within a model is achieved by a selection channel which is passed through the chain of nodes. The channel is modified by selection nodes which allow for use of binary operators (NEW, OR, AND, XOR) acting upon a set of convex hulls specified by the user. These primitives representing the selection channel are then themselves modified by each node in the pipeline, for example being scaled or translated at the same time as the actual geometry. This allows a huge amount of flexibility and makes the selection of geometry for the application of modifiers far less brittle and liable to error.

4.3.3 Multi-resolution procedural modelling

In the creation history shown in figure 4.2, the model is only broken down into a mesh for rendering as the last step of the chain, thus this means that the resolution of the final rendering can be changed independently of the model. As shown in figure 4.3.

The use of a creation history provides a form of procedural modelling allowing a far more compact model description to be saved for permanent storage. This is in part due to the fact that the final polygonisation of the model need not be stored as many thousands of triangles

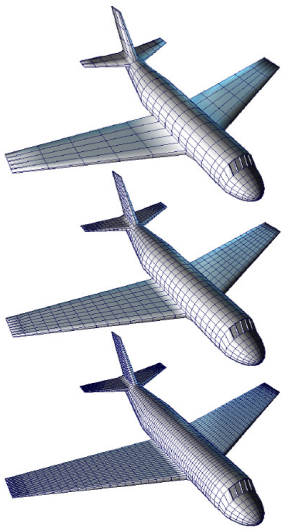


Figure 4.3: Multiresolution models can be achieved simply within ClayWorks by adjusting parameters in the creation-history. From the top down, each plane has 886, 1946 and 5074 vertices; 3576, 7816 and 20328 edges; 906, 1964 and 5092 planes. The model requires 1160 bytes of storage as only the creation history needs to be stored [3]

instead only the concise description of the creation history and the parameters within each node need be stored. The polygonisation of the model can then be computed at runtime.

4.3.4 Limitations

- **Linear creation history & no data amplification** - The creation history within Clay Works must be strictly linear, in that the nodes must form a chain, and not a tree or graph of nodes. This limits the power of the software. As explained in the introduction to our software, the use of a tree of nodes provides the user with great control over the data amplification problem, which is caused when the amount of data within a scene expands exponentially over and above the amount the artist can control and modify successfully in a succinct manner, through the utility of the software packages.

- **No Imperative Constructs** - Coupled with the above, the nodes used within the creation history fall into three categories, graphical primitives, selection nodes and modifiers. There is no facility for control structures such as loops or logical tests (if/case statements) or variables and mathematical expressions to be built into the data structure. This is one of the main foci of our software, as when coupled with other basic constructs from imperative programming languages it provides an extremely easy to use and very powerful procedural graphics environment. This report shows that this is possible and very effective.

- **Lack of Animation** - As mentioned within the further work section of T. Lewis' and M. W. Jones' report, the Clay Works software lacks any facility for animation. This is unfortunate as the parametric nature of the nodes in the creation history could easily be procedurally changed over time, producing some very interesting animations. Since the creation history can be edited in a non-linear fashion animation could be produced in this way very quickly and with very little effort on the part of the user. In the next piece of related software, we describe the idea of modelling in four dimensions - xyz & time may be powerfully extended.

4.4 HyperFun

HyperFun[6] is an interesting project which allows the user to produce models using F-reps or *Functional representations* of geometric shapes. All modelling is done via scripting in the HyperFun language. All items in a given scene are generated from functions in several variables of the form $F(x_1, x_2, x_3, ..., x_n) \geq 0$. This approach is very powerful as it allows the mathematical specification of objects to be used, for example a sphere could be specified as $5^2 - (x^2 + y^2 + z^2) \geq 0$.

The specification of such geometrical objects along with the use of various operators including set theoretic operators (AND,OR,NOT) allow for the creation of very interesting models:

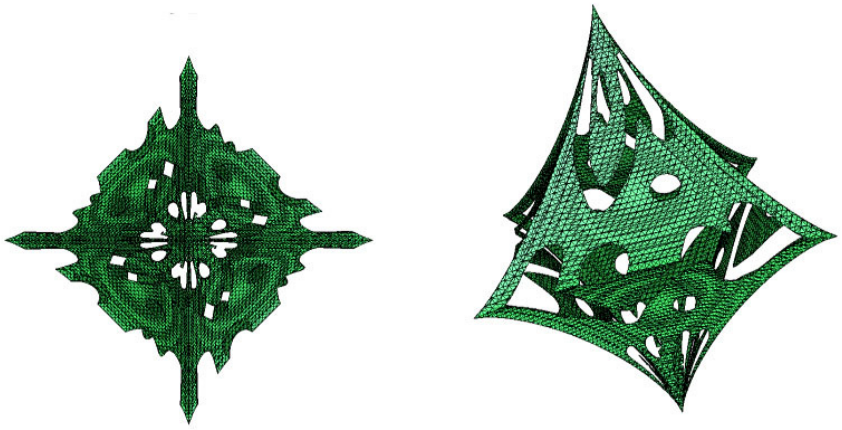


Figure 4.4: An example of the Images produced by HyperFun [6]

4.4.1 Multi-Dimensional Modelling

Since F-reps use multivariate functions used to define its geometric primitives, it is possible to utilise these multiple dimensions for modelling, whereby in addition to the x,y,z ordinates extra dimensions can be used for time so as to provide animation, or as show in figure 4.5 to create a spreadsheet showing similar models with different parameters. This allows the user to review a large number of possible design options and select the best.

4.4.2 Language translation

One of the unique features of HyperFun is that models written in HyperFun can be translated into several other languages, most notably into Java via the "HyperFun to Java" translator. Models may also be polygonised, ray traced or exported to VRML (Virtual Reality Modelling Language) for concise sharing over the internet.

4.4.3 Tree System

The data structure used within HyperFun is very interesting, though not specified through a graphical interface but instead through a scripting language the internal data structure forms a tree structure: (As far as our investigations showed, no direct graphical manipulation of this tree was possible)

Graphical primitives, in this case F-Reps form the leaves of the tree whilst various tools make up the nodes of the tree, such as union, intersection and various other functions. When the model is to be drawn the tree is parsed from the root to the leaves building up metadata on the structure of the tree, whilst the actual model is built up by traversing from the bottom of the tree to the top. For example executing various unions and intersections of child nodes

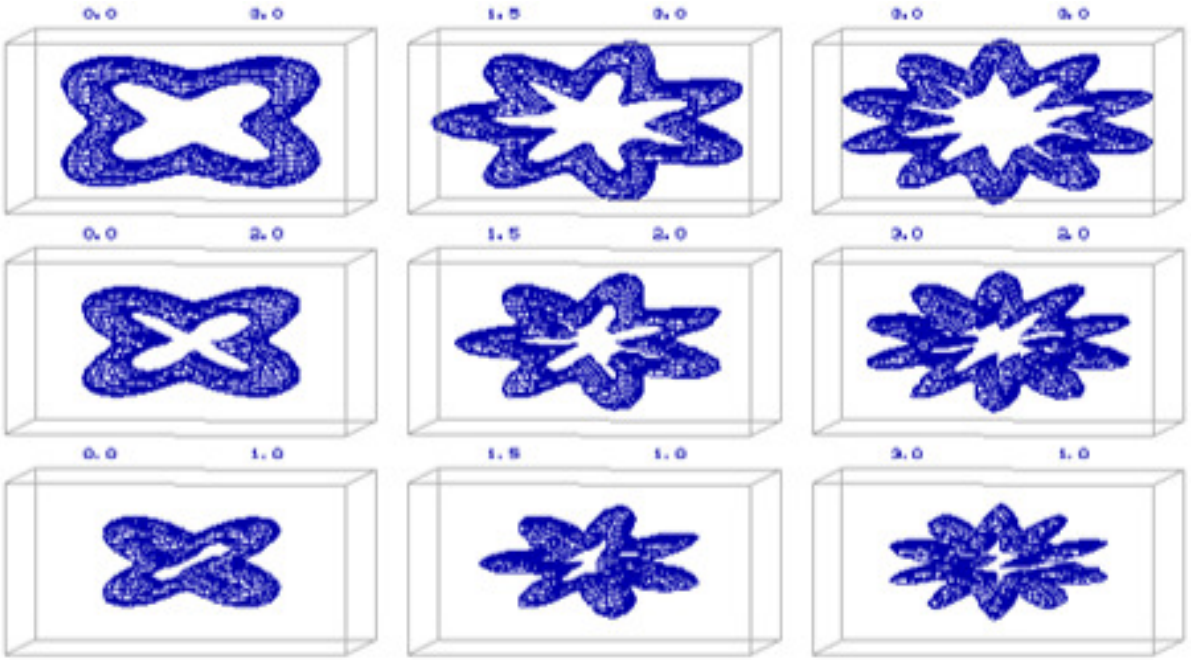


Figure 4.5: An array of images produced using multi-dimensional modelling [6].

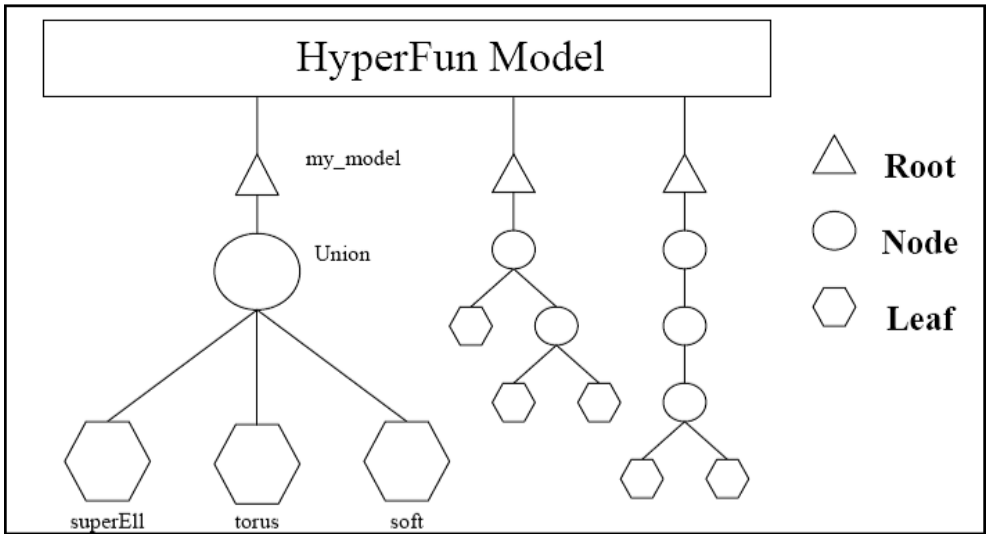


Figure 4.6: A representation of the background yree structure used by HyperFun [6].

until the final model is produced at the root node. This method of execution is identical to the Build Up tree semantics we implement in our system and discuss in section 3.3. This form of semantics is especially good at creating a single model or piece of geometry. However it fails to be able to create a full graphical scene.

4.5 Blob Tree

In "Extending the CSG Tree" [7] Wyvill et al presented a modelling system based on an extended Constructive Solid Geometry tree (a method often used in conjuction with Ray Tracing). A CSG tree is a tree in which the leaf nodes are graphical primitives and branch nodes are composed of boolean operators such as union, difference and intersection. The innovation provided in the system presented was to include additional operators into a tree using an implicit surface modelling system (where objects do not necessarily have well defined surfaces/edges).

The use of warping and blending operators in the tree made possible the creation of some very unusual objects which would be challenging to create in other programs (as shown in figure 4.7).

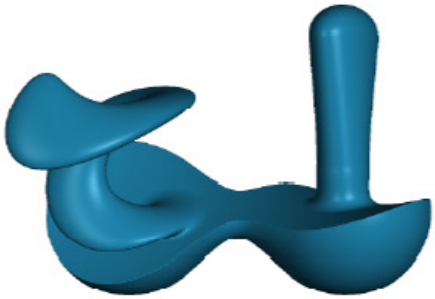


Figure 4.7: A sample model made using Blob Tree [7]

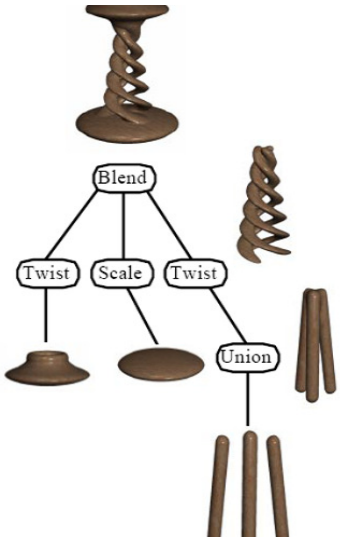


Figure 4.8: The n-ary tree structure produced by this system is termed a "Blob Tree" [7]

Whilst this project is not concerned with implicit surface modelling the Blob tree structure

used within the project is of direct relevance to this project due to the rendering execution of the tree. This is again a form of build up execution where the whole model is represented by the root node, which is formed by a composition of the child nodes of the tree (as shown in figure 4.8). We implement very similar build up execution semantics in the "Buildup" tree semantics presented in section 3.3.

When rendering a Blob Tree structure the tree is evaluated recursively, each node returns a formula in terms of its child nodes or if a leaf node then a function representing the field strength of the object in 3d space is returned. These formulas are then evaluated and the scene is then either polygonised and rendered or rendered via ray tracing. The system makes interesting use of optimisation via space subdivision to minimise the time required for rendering by culling branches of the tree from the recursion which are constant between the change the user is making. The resulting formula is then evaluated at points specified during the ray tracing of the resultant model.

4.6 Sketch-Based Procedural Surface Modelling

The system presented by Schmidt and Singh [5] is an extension of the ideas presented in the "Blob Tree" system described above. It incorporates:

- Additional tools such as soft displacements, sharp creases, holes and handles which increase the usability of the software.
- A sketched based interactive modelling environment for selecting features of the model and editing parameterised nodes.
- The locking of sections of the tree to increase the rendering of the tree during editing, gaining speed advantages by caching results of some branches of the tree.
- A linked copy and paste system within the tree, whereby if one part of the tree is copied to the tree, any changes to the original part will automatically be copied to the new part and vice versa.

These enhancements make the system dramatically more useable and more powerful showing how a procedural system should be put together. The interactive modeller part of this software is a huge step forward as it mixes interactive and scripted modellers, the end effect is a system similar to ClayWorks.

CHAPTER 5

Implementation

In this chapter we detail some of the critical design decisions, their implementation and most importantly they effect they have on the usability and power of the system we present. A full detailing of the language presented is provided in chapter 7

5.1 Development Language

The success or failure of this project was largely dependent on swift development of a language within a professional tree view component giving a user friendly and powerful IDE. Speed of development was critical especially as project weighs in at over 15,000 lines of code. C# was chosen for development partly so that author could learn a new language during this project but also more importantly because was by far the best match for this project since:

- As a .Net language C# runs on the Common Language Runtime (CLR) with Just In Time (JIT) compilation of code. This allows advanced features such as reflection and late binding which was heavily exploited.
- There are a number of commercial grade Treeview components available such as Infralution's Virtual Treeview detailed in section 5.2.
- Heavy use was made of the .Net Threading environment which provides a straight forward way of exploiting multicore processors. For example by sending a delegate (a function pointer) to the ThreadPool maintained by the instance of the CLR running the project one never needs to deal with actual thread primitives, forks or joins. The new .Net Framework 4.0 looks set to improve this still library still further [30].
- Language INtegrated Query (LINQ) was exceedingly useful in managing lists and dictionary's of objects by providing many list processing functions (Filter, Sequence, FindFirst ect) which are normally only found in functional languages. These methods and the custom ones written for this project can be applied to any list in the project saving huge amounts of code and thus speeding up the development process.
- C# integrates well with the Microsoft DirectX graphics API which was used to render animation within the project.
- Event based programming and delegates made several programming tasks far easier and more powerful as well as building in more extensibility than would have otherwise been possible.

Other languages considered were C++ and Java. C++ is clearly the faster language (as it does not use Just In Time compilation) however the lack of advanced object orientated techniques would have made implementation a long and laborious process. For example the reflection capacities of C# were heavily exploited to populate the user toolbox with all valid modifiers, similarly the Serialization libraries saved scores of hours in writing custom save/load functionality.

The advantages of Java are strong as it natively supports write once run on multiple platforms, however it lacks the ease of use for designing GUI forms and powerful a Treeview component. Although it does support the OpenGL graphics libraries its feature matrix does not match up to that provided by C#.

C# and the .Net Framework 3.5 have provided an excellent platform for fast development. Visual Studio 2008 has also provided a good development and debugging environment. I have enjoyed learning C# and found it possible to implement many more features more quickly than I would have expected at the start of the project, for example the LSystems modifier 5.7 was implementated in 800 lines of code and 7 hours by exploiting functional programming methods and LINQ.

5.2 Selection of a Treeview Component

With a tree based programming language the choice of the visual display and representation of the Treeview component is critical. The choice of a Treeview component offering professional grade performance and functionality was key. After much research Infralution Virtual Tree 3.0 was purchased [2]. This .Net component allows: multiple columns, built in printing routines, the embedding of editing components, custom column sorting, rich text support within cells, drag and drop support and alpha blending amongst others features. The main advantage of this component is the data model upon which it is based:

> *"Unlike most other tree controls, Virtual Tree is designed from the ground up to be data driven. Most other tree controls have a basic "unbound" mode in which the tree representation is built manually. If they support data binding, it is typically built on top of this "unbound" mode and simply generates the in-memory representation (consisting of tree nodes) when the Data Source is set. For large Data Sources this approach can be both extremely slow and resource hungry." [2]*

This virtual approach to data binding greatly simplifies the programming process as there is no need to worry about manipulating display nodes, all that the programmer must do is to edit their own internal data structure and through the data binding the Treeview component will update automatically, this proved extremely useful during this project.

5.3 NCalc, A Mathematical Expression Evaluator

Throughout the language we developed almost every possible user input is not simply just an integer or floating point number but instead is actually a mathematical expression. This gives rise to a phenomenal amount of flexibility within the language.

It also means that novice users can simply enter numbers whilst more expert users can craft mathematical expression to give rise to more complex geometry. This hiding of advanced features in plain sight makes for user friendly software since it does not present the novice user with a complex interface, nor does it hide advanced features from advanced users instead it presents the same interface to both as shown in figure 5.1. We feel this principle should be exploited more often in software design!

What makes this possible is the mathematical expressions evaluator which is leveraged. Instead of using a parser generator to create a custom built expression evaluator it was decided to leverage NCalc[20][21], an open source expression evaluator for .Net written by software company evaluant[19]. The library is especially useful as it is fully extensible and allows the

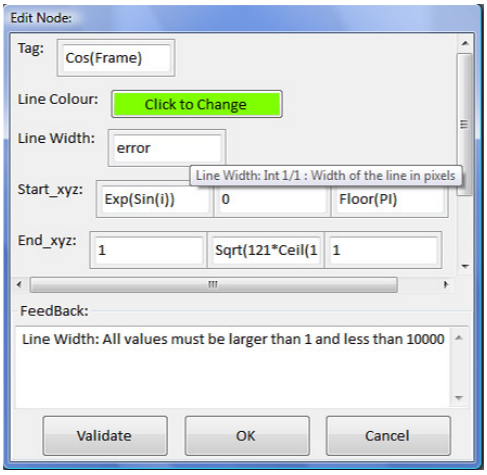


Figure 5.1: an example of an NCalc field in the project [2]

runtime addition of functions to the language without having to recompile or edit the language definition file which generates the parser for the expressions.

This is useful in two ways:

- Firstly it allows additional functions to be added, for example a LimitTo(number,min,max) function was added and is applied to most input in the program to ensure that expressions evaluate to reasonable numbers.
- Secondly it allows the dynamic evaluation of parameters, for example throughout a scene tree one can make reference to the variable (parameter) "Frame" so as to have access to the current frame number for the purposes of animation.

This extensibility is beautifully implemented through the use of delegates (essentially C++ function pointers) representing new functions or parameters which the client code passes to the expression evaluator prior to evaluating a given expression. Alternatively if a parameter or function is still unknown as all evaluation delegates fail to evaluate the given function or parameter then by the use of C# events the client code has a final chance to return a value. For example by hooking an event handler onto the expression evaluators `EvaluateParameterHandler` event we can dynamically handle the parameter "Frame" on any given execution of the scene.

Before integration the NCalc library was upgraded to Visual Studio 2008 and C# 3.5 standards, unit tests were fixed, and the library was refactored to provide a clean point of function addition in other projects. A complete help system was also integrated and the full help listing is shown in figure 5.2 this is also integrated into the project, again this helps the keep the learning curve shallow for new users and provides a useful reference system for experienced users!

The NCalc library forms an integral part of this project since it facilitates animation by way of a "Frame" variable, it also enables For loops and If tests by allowing the loop variables to be passed through the pipeline. It also forms the lynchpin of the Tag system discussed in section 5.6.

Practically these features are implemented via an `Execution_Context` object which is passed through the pipelines along with all the primitives. This `Execution_Context` class contains a custom NCalc evaluation visitor which has been preloaded with a number of functions specific to this project. The context also contains a map between variable names and values, for example `<''Frame''=1, ''i''=2>` this map is updated by various modifiers in the language such as For loops, and Math nodes. The context object also handles all the `EvaluateParameterHandler` events from the evaluation visitor ensuring that expressions are evaluated correctly.

As you can see in figure 5.2 NCalc supports several different data types, including floating point arithmetic and boolean expressions which are used throughout the language. However NCalc also supports a number of other data types such as strings and date/time types which

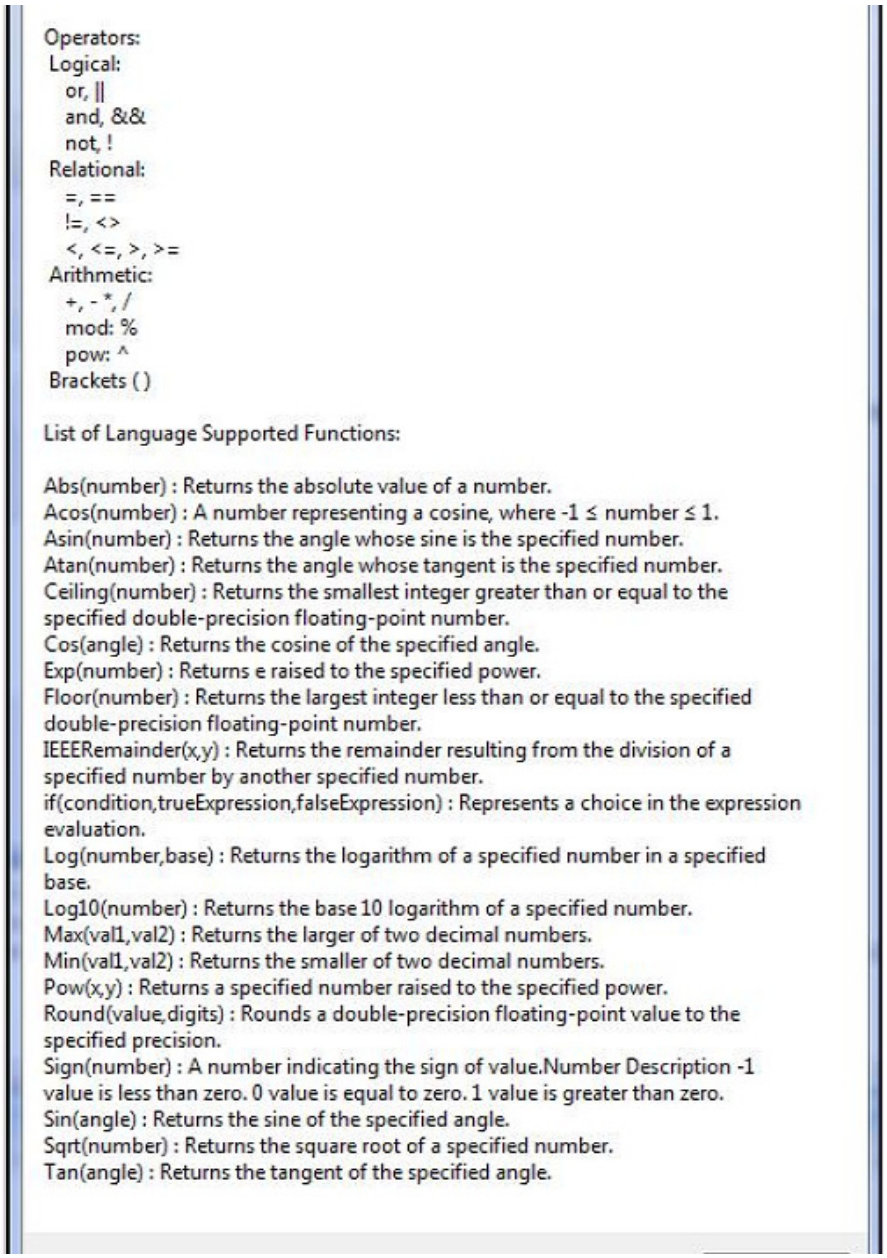


Figure 5.2: An example of NCalc fields in action, new users can simply enter numbers, advanced users can enter mathematical expressions.

are native to C#, this gives rise to a number of possible extensions to the language. For example one could easily imagine the use of string functions to implement a LSystems, or to carry more complex state information and procedural descriptions of the scene through the pipelines in the language. It would also be interesting to put together some form of clock animation which ought to be possible with this functionality.

Overall NCalc has been integral to this project and its extensibility makes it a highly effective platform for expression evaluation. It also gives rise to a great deal of future development possibilities.

5.4 Pipeline Implementation

By far the most complex task in this project was the correct implementation of an execution algorithm for a scene tree. The algorithm has to be sufficiently general as to support several different tree execution semantics and yet allow extension at many different points for the implementation of other procedural formalisms and pipeline flow modifier nodes. The two tree semantics which had to to be supported were the standard tree based build up execution semantics where the data pipelines flow from the leaves to the root node, and the pipeline based execution semantics where data pipelines flow from the root node down to the leaves of the tree where the input primitives for that pipeline are stored.

After correctness the primary goal when implementing this algorithm was make it as multi-threaded as possible. This was to prove that the language is well suited to the new generation of CPU's with their multiple cores and that the language could be executed in a massively multi-threaded environment such as a General Purpose Graphical Processing Units (GPGPU's).

On top of these two execution semantics we add a further classification in that nodes in the pipelines support either stream based or batch based execution. Stream based modifiers can be applied to each primitive independently and individually, where as batch based modifiers need to be executed on all primitives in one group for example when changing the mathematical context. A batch based node acts as a synchronisation point in a multithreaded execution as all threads executing pipelines must finish executing all nodes prior to that node in order that the batch based modifier can be execute upon all data that will be passed to it in one go.

The task of creating this algorithm is split into two distinct subtasks within the `Execution_Controller` which manages execution:

1. **Pipeline Construction:** This phase deals with parsing the tree structure and producing a list of pipelines to be executed along with their `Execution_Groups` which contain a list of input primitives which must be prepared and a mathematical context object.

2. **Pipeline Execution:** This phase deals with the multi-threaded execution of the pipelines whilst obeying the tree semantics (pipeline or build up) and node execution semantics (stream or batch based).

5.4.1 Pipeline Creation

This phase of the algorithm builds a list of `Execution_Groups` - that is a pipeline together with a list of input primitives and a mathematical context. The algorithm to construct the pipeline is recursive as one would expect with a tree structure. Firstly we shall detail the `BuildPipelines` algorithm for pipelining tree semantics which is shown in figure 5.3:

- Firstly the primitive children of the current node are found.
- These primitives are then "Executed" to return the true input primitives (see 5.4.2)
- For every modifier node in the children of the current node that is for each branch in the scene tree we generate a new pipeline via recursively calling this method.
- Once we have a full list of pipelines that will contain the current node we then add this node to the front each of them.

F:\IProject\Code\Project\Project\Execution\ExecuteController.cs 1

```
138 /// <summary>
139 /// Single threaded method to build up all the pipelines which will be required within this execution.
140 /// </summary>
141 /// <param name="root">The root node of the scene tree.</param>
142 /// <param name="collector">return final primitives to this collector. </param>
143 /// <param name="isroot">Is the node passed the root node of the scene? </param>
144 /// <param name="BuildupExec">Should we be doing build up exec? </param>
145 /// <returns>A list of execution groups.</returns>
146 public static List<Execution_Group> BuildPipelines(IExecutable root, IPrimitiveCollector collector, bool isroot, bool BuildupExec, Debug_Execution debug_execution, Dictionary<string, float> math_paramMap, Debug_Container debug_container) {
147
148     List<Execution_Group> execGroups = new List<Execution_Group>();
149     INode rootNode = (INode) root;
150
151     // get primitives
152     List<IPrimitive> primitives = rootNode.Children.AsList().CastFromTo<INode, IPrimitive>();
153
154     if (primitives.Count> 0) {
155         // deal with advanced primitives:
156         _getPrimitiveChildren(ref primitives,math_paramMap,debug_container);
157
158         // if we have primitives then a pipeline ends here so
159         // create a new pipeline ending here
160         execGroups.Add(
161                 new Execution_Group(
162                         new PrimitiveCollector(primitives), collector, debug_execution));
163     }
164
165     if (!isroot && BuildupExec) { // if build up then add before recursing
166         _addNodetoPipelines(root, isroot, execGroups);
167     }
168
169     // for each of the IExecutable nodes (branches)
170         // recurse
171
172     List<IExecutable> executables = GetExecutableChildren(rootNode);
173
174     foreach (IExecutable execNode in executables) {
175         execGroups.AddRange(
176             BuildPipelines(execNode, collector, false,
177             BuildupExec, debug_execution, math_paramMap, debug_container));
178     }
179
180     // then collect resulting list of Exec groups
181     // and add this node to them
182     // if pipelining add after recursion
183
184     if (!BuildupExec || (isroot && BuildupExec)) { // deal with build up and root nodes
185         _addNodetoPipelines(root, isroot, execGroups);
186     }
187
188     return execGroups;
189 }
```

Figure 5.3: The algorithm for construction `Execution_Groups`

Thus we see that the tree is parsed downwards and the pipelines are built in reverse from their last nodes to their first nodes (ie flowing up the tree). The reason for this two phase pipeline creation is not immediately obvious, however in order to ensure synchronisation before batch based nodes a list of pipelines to be synchronised must be constructed and shared. This is done by embedding each modifier node in the pipeline within an `ExecNode` object which keeps a list of the pipelines which must signal that they have finished executing all nodes before the `ExecNode` before the batch based modifier contained within the `ExecNode` can be executed. This list is built up by the `addNodetoPipelines` method.

We note in passing that the pipeline is stored as a list of `ExecNodes`, with generic extension methods applied to give it similar functionality to a stack but with the full accessibility of a list. The first node of the pipeline to be executed is stored at the end of the list (i.e. the top of the stack). This enables the pipeline to be created quickly as the add method of a list is faster than inserting a node at the front of a list.

The algorithm is currently only single threaded this due to the nature of the recursion from the root as all pipelines will eventually have the first few nodes added to them. However it would be possible to thread the algorithm such that each recursion (branch of the scene tree) is done via a different thread via a work splitting algorithm. However the overhead due to the inherent synchronisation of adding the same `ExecNode` to each pipeline may well leave this algorithm bereft of benefit beneficial parallelism.

5.4.2 Primitives

Whilst at first glance it may seem simple to extract the primitive nodes from the children of a modifier node, there is some inherent complexity due to the addition of advanced primitives within the language. For a full list of advanced primitives see section 7.2. The code to find all primitives is executed once per pipeline and is called whenever a group of primitives are found as the leaves of scene tree. Primitives are grouped if they all share the same parent node. Each such group of primitives marks the input of a new pipeline and their parent node marks the last node of their pipeline and is the first node to be added to the pipeline when it is being constructed.

Advanced primitives include Primitive Groups which allow primitives to be grouped within a primitive tree (a sub tree of the scene tree). The Primitive Groups must be extracted from the list of primitives and the primitives that are hidden within them must be extracted. Clearly this process is recursive and must be applied until no further Primitive Groups are found.

The remaining advanced primitives would be nodes which generate primitives, for example the LSystems node which must be executed to produce a list of new primitives. These nodes are extracted after the primitive groups are dealt with and are executed. There is the possibility to execute these LSystems in parallel, however it is unlikely that more than one LSystem will be found in the same input collection for a single pipeline so this is not currently implemented.

The final type of advanced primitive currently implemented is the `BuildUpNode` which denotes that the subtree which it is the root of actually represents a new sub-execution. This sub-execution must be carried out with the Build Up execution semantics described in the next section. After its execution the resulting primitives are returned and added to the list of input primitives for the pipeline being constructed.

5.4.3 Buildup Nodes

When a `BuildUpNode` is encountered denoting a change in execution semantics we must begin a new sub-execution. This sub-execution is started by a call back to the `Execution_Controller` class which in turn calls the `BuildPipelines` method described above.

This method however runs somewhat differently when the `BuildupExec` flag is turned on. When build up execution is carried out we wish to build pipelines which run from the leaf nodes to the root node (ie back to the build `BuildUpNode` which started the sub-execution). This is the reverse way to the normal pipelining execution. To achieve this we simply add the node to the pipeline before recursing instead of afterwards.

Special care must be taken of the root node when using build up execution since this node should not appear in the pipelines since it is already being executed in the construction of the pipelines which fall below it.

The pipelines generated are then executed in an identical manner to those generated with the pipelining semantics. This shows the generality and extensibility of the pipeline formalism we implement.

5.4.4 Execution

The result of creating pipelines is to produce a list of `Execution_Group`s which contain a pipeline (a list of `ExecNodes`), a list of input primitives and an `Execution_Context` which contains the mathematical context used to evaluate mathematical expressions. The execution of these `Execution_Group`s needs to be multi-threaded to take advantage of todays multi-core CPU's.

In order to execute a scene the list of `Execution_Groups` must be split into a number of workloads each of which can be executed by a thread. A `Workload` is a list of `Execution_Groups` together with the number of stages of that pipeline which should be executed and the `Execution_Context` for this execution. This splitting of work is managed by an `Workload_Controller`, we actually wrote two control algorithms for this, the one we eventually decided upon was more reliable and is as follows:

- For each `Execution_Group` currently able to execute (ie not a batched node blocked waiting for another pipeline to complete)
- Group the `Execution_Groups` into lists according to their first node.
- Find the length of common pipeline which each list of `Execution_Groups` shares so that it can be executed together, ensuring that we do not include any batch based nodes.
- Wrap each of the lists into a `Workload` item to be executed by a thread. The length of the workload is the number of common first nodes amongst the pipelines in the Execution Groups.

Each workload item is dispatched to the .Net threadpool which will spawn a number of threads proportional to the number of processors to process the workloads. To enable synchronisation at the end of execution and to ensure that it is possible to know exactly when the execution finishes before rendering, the `Workload_Controller` keeps track of the number of workloads it has issued to the ThreadPool. Upon completion of a workload this number is decremented and when it reaches zero (with no blocked Execution Groups) the `Workload_Controller` will signal completion of the execution.

Alternatively it would also be possible to thread the list of pipelines sharing the same section of pipeline individually, the algorithm then being as follows:

- For each `Execution_Group` currently able to execute (ie not a batched node blocked waiting for another pipeline to complete):
- If the next node to execute in the pipeline is not batch based then create a workload item executing the `Execution_Group` until the next batch based node.
- For all pipelines beginning with a batch based node - group the `Execution_Groups` into lists according to their first node.
- Find the length of common pipeline which each list of `Execution_Groups` shares so that it can be executed together (this creates a larger workload instead of just executing one batch based node).
- Wrap each of the lists into a `Workload` item to be implemented by a thread. The length of the workload is the number of common first nodes amongst the pipelines in the `Execution_Groups`.

This would introduce further parallelism, however it is also likely to be going to far on a CPU, since the overheads of threading although reduced via thread reuse in the .Net threadpool are still likely to be too larger faction of the total work done, thus we have used the original algorithm.

Each of these workloads is then executed via the algorithm show in figure 5.4:

Firstly the state object passed to the worker thread is unpackaged as a workload object, then the number of pipeline steps (modifier nodes) specified in the workload object are executed. Once these nodes are executed. We then check if the pipeline has been completed if so we return the output of the pipeline to the return site which collects the final primitives of the scene. If not then we split up the pipelines according to any branches in the tree and return the `Execution_Groups` back to the `Workload_Controller` so that more workload objects can be created and passed to the thread pool for execution. On completion of the **Workload** we signal back to the `Workload_Controller` so that it can ensure that it signals execution completion at the correct time.

There are a large number of subtleties in these algorithms which are not discussed here, for example management of the mathematical execution context is quite difficult as one must ensure that all pipelines get the correct context and that when primitives are evaluated they get the mathematical context generated from the Animation Controller and the Scene node. We refer the interested reader to the codebase which is well documented and the HTML documentation generated from it.

5.5 Rendering

Once the scene tree has been executed the resulting list of primitives needs to be rendered. We provide two methods of achieving this both of which support animation:

1. Firstly we provide a built in DirectX renderer. This renders the primitives through a standard graphics pipeline. It renders quickly and allows fly through navigation via user control. We have actually had to introduce delays in execution to ensure that an entire animation is not displayed within one second. We used a push data mechanism to move data to the rendering controller. This ensures that all primitives have been prepared for rendering before the DirectX renderer tries to access the data. The effect is similar to using a Z-buffer.

2. Secondly we provide a method of exporting the list of primitives into POV-Ray scripts. This is done simply by translating every primitive into its POV-Ray representation. Although more concise scripts could be generated by exploiting the structure of the tree (for example by translating translate node in the tree to equivalent POV-Ray code) the current method suffices well. Each successive frame of an animation is written to a separate script file which is numbered sequentially. Finally an animation script is written which when executed in POV-Ray results in the whole animation being rendered.

It would be quite possible to add additional methods of output including export to industry standard file formats. The execution engine provides an event which can be handled by various classes including loggers and the two renderer's mentioned above.

Animation can be rendered to video in POV-Ray as each script generates a bitmap file of the rendering a set of which can be strung together into a .avi video file using an open source *EasyBMPtoAVI* tool [23]. There is also support for creating videos using DirectX rendering via the same mechanism.

5.6 Tagging

To allow the user to selectively apply modifier nodes we have implemented a system of primitive tagging. Each primitive has a "Tag" field associated with it and every modifier node has a "Tag

F:\IProject\Code\Project\Project\Execution\Executor.cs 1

```
14 /// <summary>
15 /// execute a workload - this code is multithreaded
16 /// </summary>
17 /// <param name="threadContext">Workload object to execute</param>
18 public static void Execute(object threadContext) {
19     Workload workload = threadContext as Workload;
20     if (workload == null) {
21         Logger.ERROR("Thread was passed an invalid workload");
22     }
23
24     // do work (execute workload.length steps for each exec group)
25
26     Pipeline pipeline = workload.Exec_Groups.First().Pipeline; // pipeline is identical for
       all Exec_Groups
27
28     while (workload.Length > 0 && pipeline.Length>0 ) {
29         pipeline.Head.ExecuteGroups(workload.Exec_Groups, workload.Context);
30         workload.Length--; // logic nodes mean this wont just go down by one
31     }
32
33     // check for end of pipeline & if so send output to return site.
34
35     List<Execution_Group> finshedGroups = workload.Exec_Groups.Filter(g => !g.Pipeline.
       IsEmpty()); // extract empty pipelines  - remove if ¬¬IsEmpty
36     foreach (Execution_Group group in finshedGroups) {
37             group.ReturnOutput();
38     }
39
40     // check for branching
41
42         // split on first node
43
44         Dictionary<IExecutable, List<Execution_Group>> groups = new Dictionary<IExecutable,
       List<Execution_Group>>();
45         foreach (Execution_Group group in workload.Exec_Groups) {
46             groups.AddMapofArrays(group.Pipeline.Head, new List<Execution_Group>() { group
        });
47         }
48
49         // return more work
50         foreach (KeyValuePair<IExecutable, List<Execution_Group>> group in groups) {
51             workload.Controller.AddandExecJobs(group.Value, (Execution_Context) workload.
       Context.Clone());
52         }
53
54     // finish
55     workload.Controller.FinishTask();
56 }
```

Figure 5.4: The algorithm for execution of `workloads` (sections of pipeline).

Test Expression" based upon a "Tag" variable. The semantics of the Tag Test vary between modifiers however the most common usage is to apply the modifier to a primitive only if the test evaluates to true.

As noted the Tag Test is actually a mathematical expression and thus can become quite complex including the use of boolean operators and even If statements to choose which expression to evaluate! Accordingly the primitives also have a mathematical expression to represent their initial tag. This tag expression is evaluated at the same time as the primitive is evaluated, that is when the primitive is processed by the root node at the start of every pipeline.

This semantic tagging allows the user to group primitives and to apply modifiers to exactly the right primitives and is a very useful construct. We also provide a "Tag Changer" node which if a Tag Test expression evaluates to true will replace the primitive's tag with a new tag which is found by evaluating another mathematical expression possibly based upon the current tag of the primitive. This tagging of primitives complements an idea called selection channels discussed in the further work section 9.

5.7 LSystems

To demonstrate that the LSystems formalism described in section 2.6 works well within our framework we implemented an LSystems primitive. The options this node takes are described in 7.3, in this section we give an overview of the type of LSystem we have implemented and its power.

The LSystems are evaluated using Turtles (that is drawing agents) which parse the evolved LSystems. Our turtle moves around 3d space drawing lines as he goes. The commands which our Turtle comprehends are:

- **Pen Control**: Pen Down, Pen Up represented by the characters **"+"** and **"−"** respectively. These allow the turtle to move around without creating new primitives.
- **Line Creation**: Forwards **"F[amount]"** this command tells the turtle to move in the direction he is facing by the specified amount (which may be negative causing him to move backwards). If the pen is down then he will create a new Line primitive beginning at his old position and ending at his new position.
- **Rotations**: rotate around the X,Y or Z coordinate axes represented by the letters **"X[degrees]"**, **"Y[degrees]"** and **"Z[degrees]"** respectively. These commands rotate the turtles direction vector around the given axis by the given number of degrees (values may be negative).
- **Bracketing**: These commands allow the turtle to warp back to a position he was originally in. The position and direction of the turtle are stored every time a **"{"** is encountered and the turtle warps back to the last saved state every time a **"}"** is encountered. This mechanism allows the branching of LSystems as shown in figure 5.5.

The LSystems we implement is a *parameterised* LSystem - each alphabet letter has a parameter attached to it: **"letter[parameter]"**. To evolve the system the user specifies a list of productions. These are given in two parts, first a pattern to match for example `"[ab]+B[b]-A[ba]"` one should note that every time a letter is mattched the user must specify a new variable for that input, these variables do not effect the matching process but are instead used in the production rules. An example production rule could be `"B[ab+b]+A[b-ba]-B[ab*ba]"` which is used to replaced the matched pattern. The new parameters of each of the letters is actually a mathematical expression which may involve any of the variables in the mathematical context and the parameter variables created by the matched patterns. This is a novel and highly useful feature.

A description of how to write an LSystem is found in section 7.3

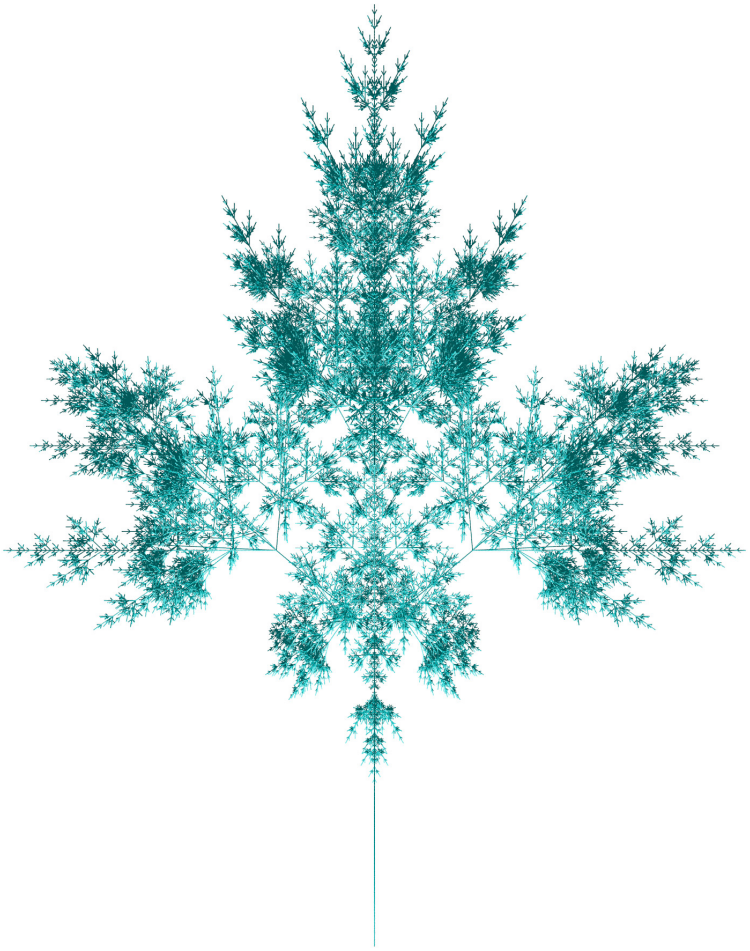


Figure 5.5: An example LSystem, original system from [29] extended to 3d by the authors.

CHAPTER 6

IDE Implementation

In order to bridge the gap between the skills of the artist and the programmer we must ensure that the development environment is easy to use as possible. This is especially the case in a graphically manipulated language, if the language is hard to edit then the system no matter how powerful will not be successful. To this end the language has its own Integrated Development Environment (IDE), with the following features to ensure the user experience is as good as possible:

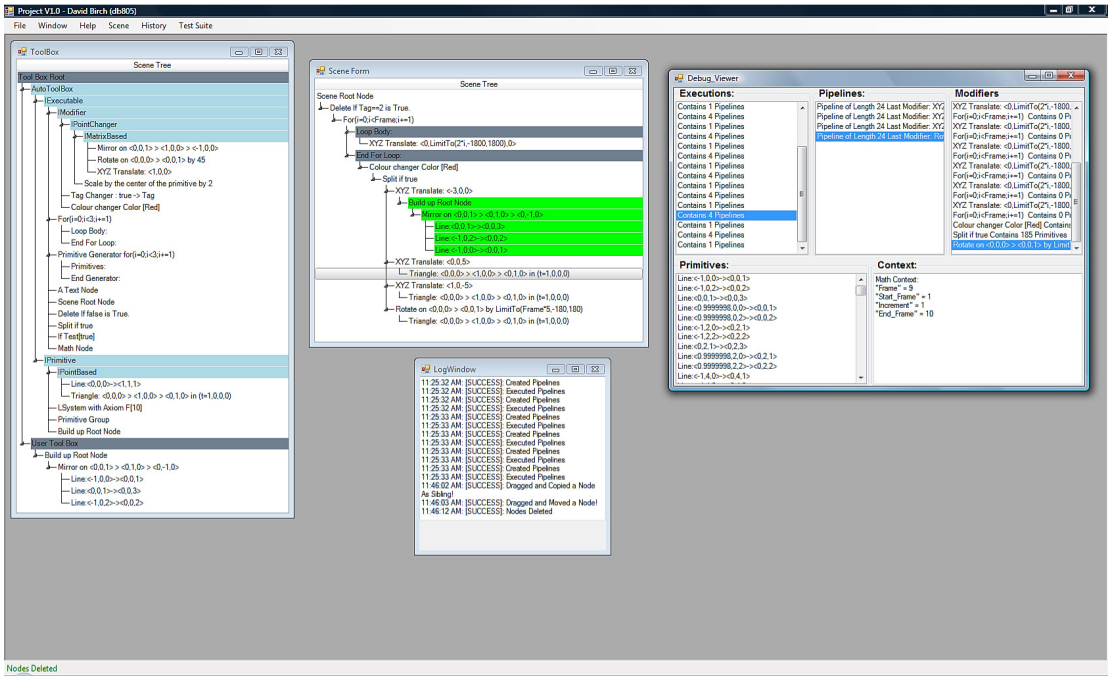


Figure 6.1: The interface of the project as discussed in this chapter.

- **Save/Load Functionality** - We exploit the C# Serialization technology to save and load all non-volatile data corresponding to a scene. This technique allows built in libraries to

parse each object in memory and then automatically create a bitstream representing the class which can then be written to disk. Thankfully the library also provides deserialisation methods! Although this is perhaps not the most space efficient method it has served well the purposes of this project especially as saving a new class was as simple as following a few simple rules and tagging the class as `[Serializable]`.

- **Do/Undo/Redo Command System** - Any good IDE should allow the user to undo mistakes, we allow this with a twist - as explained in section 6.1.
- **Full log system** - This is a project wide logging system which provides detailed messages to the user, with five levels of importance from *debug* through to *success* and *failure* messages. All messages are timestamped and logged to a log file and to a log form in the users workspace, as well as the latest message being displayed on the status bar for ease of viewing.
- **Highlighting System** - This is a user editable set of highlighters which allows the user to colour in certain areas of the scene tree in different colours to aid their understanding of it. For example all primitives could be coloured in green and all modifiers in blue. Alternatively the two different semantics for tree execution could be coloured in different colours to avoid confusion.
- **End to End Test Suite** - This was implemented as a way to ensure the integrity of the code base. The tests work by storing the output of executing a scene which has been user checked for correctness. Then when the test is run the scene is loaded and executed again, the output files are then compared for correctness with those originally generated. For each execution (i.e. for each frame) we store both a textual representation of each pipeline with its input primitives and also the final set of generated primitives.
- **Type Safe Drag and Drop** - Since the language is implemented in a tree structure, the primary way of editing the structure is via the dragging and dropping of tree nodes and subtrees within the users scene tree. The drag and drop system is made type safe by ensuring that nodes can only be dropped in legal positions thus avoiding the need for compilation or constraints checking before execution. We detail this mechanism in section 6.2.
- **Tool Box** - In common with the drag and drop interface we provide the user with a standardised toolbox which contains all language elements. We also provide the user with the ability to rearrange this toolbox and indeed to store their own favorite "snippets" of scene tree. This allows the user to be able to store more complex geometry generators and composite modifiers. This aids reuse and speed of scene development.
- **Standardised Input Forms** - In any large development environment the users is likely to be presented with a plethora of dialog boxes. To avoid the confusion that this causes we have implemented a unified means for generating forms with a standardised style. Indeed there are only 9 types of input a user can be required to make, all presented in a standardised form with clear validation techniques, tooltip help messages and error reporting. This system is detailed in section 6.3.
- **Visual Debugging** - As detailed in section 6.4 we provide the user with a means of debugging any problems with their scene trees by showing step by step exactly how the scene was generated.

6.1 Command History

As in any good Integrated Development Environment we provide a do/undo/redo system. This system is achieved by wrapping every action the user makes in a command object which supports an execute method to carry out the users actions. The object also supports a method for

creating an Undo Command which undoes the action the command object carries out. This undo command should then be able to generate an undo command which will return the original command!

We store all command objects in a specialised linked list which allows an undo/redo history to be maintained. We also provide further links within this list to allow the storing of all commands ever executed. For example if a user edits a node then undoes that editing then creates a new node. All three commands (do edit, undo edit and create node) will be stored. Though if the user were to repeatedly press undo they would only ever be able to undo the create node command. This extra storage allows for an interesting possibility in that because the entire history of the scene is stored it would be possible to render every step of the creation process of a final scene - including all actions that the artist did, undid and did again. The effect would be similar to an instruction book for a model plane, it would also allow the artist to review what they could change in their work process. Of course if these extra storage overhead becomes too large we provide a mechanism to clean the command history of all commands which have been done and undone that is commands which can never be accessed by the user through undo/redo actions.

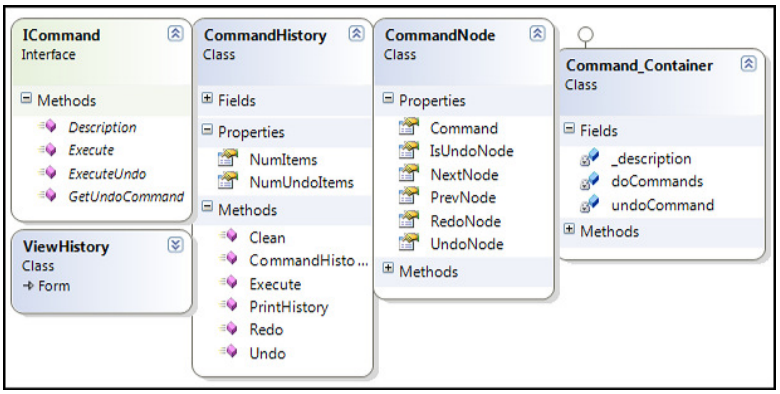


Figure 6.2: The Command stack of classes which facilitate do/undo/redo functionality.

The Undo/Do class stack is shown in figure 6.2. Several commands implemented are actually the composition of a few more basic commands, for this we provide a command container to present the composite command to the user as one command. We actually only implement 10 different commands: Three commands for adding and deleting nodes: Add node as child, Add node as sibling and Delete node. Six commands for dragging and dropping nodes: Moving a node as sibling and as child and splicing it between where its dropped and that nodes parent, along with copy versions of these for use when moving between the toolbox and a scene (when extra state information must be stored as the source of the drag could become unavailable). The final command is a user edit command which is described in section 6.3.

6.2 TypeSafe Drag and Drop

As the language implemented is embedded in a professional tree structure we decided to make the primary means of manipulating the language to be the *dragging* and *dropping* of language elements. The toolbox we provide allows users to drag and drop new elements into the tree and also to store tree fragments for later reuse. We feel this means of editing is intuitive and easy to use as there are no special tools to use, every node in the toolbox is preconfigured to its default values which the user can then edit by double clicking the node in the tree using the UserEditable System discussed in section 6.3.

However one major problem of drag and drop systems is that they allow any arrangement and ordering of nodes in a tree with no regard to whether or not a particular arrangement is legal and will execute correctly. The standard way of resolving this problem is to provide some

form of compilation or constraint checker which is run over the tree structure before execution to ensure the legality of the tree (for example to check that primitives do actually appear only at the leafs of the tree) and then provide a number of error messages back to the user. However this process of tree checking and error messages can become very frustrating especially if the user is new to the language or the error messages are unclear.

A far superior solution to this problem is to ensure that no illegal configurations of nodes can arise in the first place. To this end we implemented a TypeSafe drag and drop system that ensures that nodes being dragged can only be dropped in legal places within a scene tree. This is done by ensuring that every node in the tree implements the `IDragDropable` interface show in figure 6.3 :

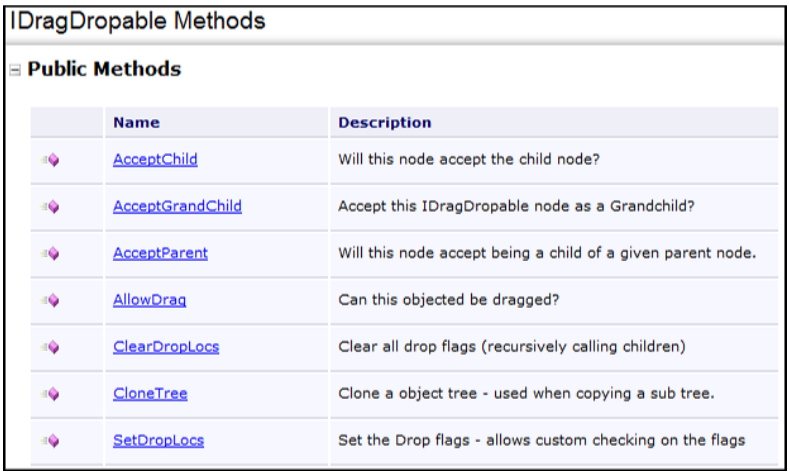


Figure 6.3: The methods of the `IDragDropable` interface which all nodes in a tree must implement to ensure they meet the typesafe drag drop requirements. Screenshot from Doc-O-Matic generated project documentation.[24]

When a user attempts to drag a node, the node is queries via the `AllowDrag` method to see if it can be dragged. For example the root node of a tree cannot be dragged. Once being dragged the node will move across a tree component, as soon as this is detected a Tree Visitor class accesses every node in the tree and tests whether or not it would accept the node being dragged via the `ClearDropLocs` and `AcceptChild` methods which may in turn call the `AcceptGrandChild` and `AcceptParent` methods depending on the node type. The result of running the vistor class through the tree is that each node has a `DropFlags` object which records whether or not the node can be dropped as a child of the node or as a previous or next sibling in the tree. This information is then used by the highlighter stack of classes to render a graphical representation of the legal drag sites to the user, and the tree component enforces these constraints ensuring that only legal trees are built. An example of this process is shown in figure 6.4.

Overall this is an extremely useful method of ensuring the integrity of the tree, and although it is sometimes complex to cover all cases for the `AcceptChild` method (especially for composite nodes such as For Loops) it is far more than worthwhile in terms of improving user experience and ensuring that the scene tree is at all times coherent and legal. Especially as it avoids having to write a tree constraints checker or to compile the language.

6.3 User Editable System

One of the goals of this project was to create a language which was much easier to use than previous procedural modeling systems. To this end a good Graphical User Interface (GUI) needed to be created. Thankfully the use of C#, its Rapid Application Development (RAD) environment and visual component library made this possible and fairly painless.

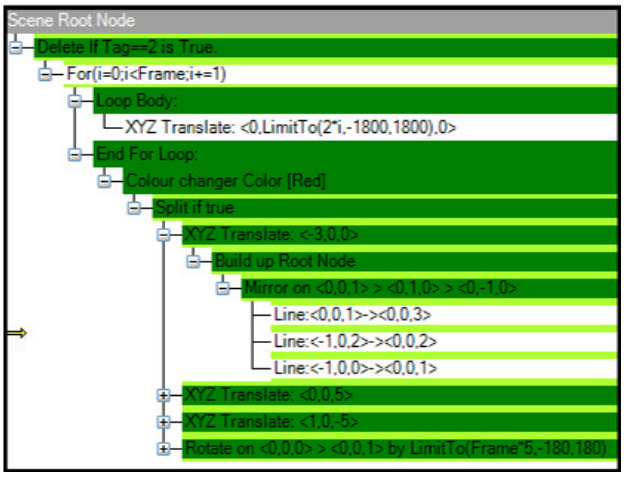


Figure 6.4: An example of Type Safe Drag and Drop - a Mirror modifier node is being dragged from the tool box: green shows where the node can be dropped as a child, light green shows where the node can be dropped as a sibling (above or below). The small yellow arrow shows where the cursor is (the node being dragged is also shown by the cursor but does not appear in screenshots).

However the number of visual forms which would need to be hand crafted is very large (around 25) which would exceedingly time consuming, error prone and dull to create individually. To avoid this a form generating system was created. The class diagram for this system can be seen in figure 6.5.

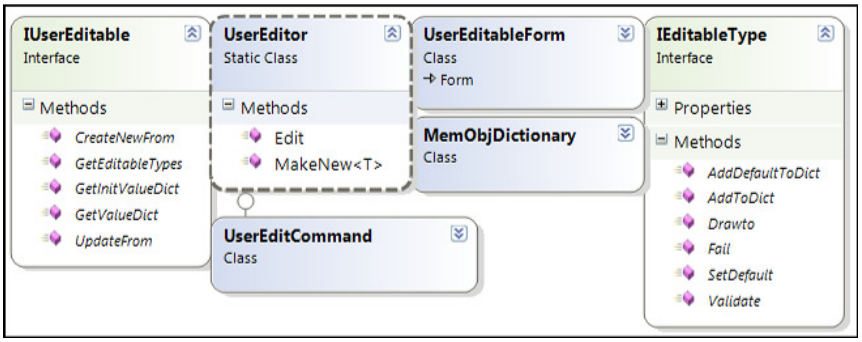


Figure 6.5: The user editable system which allows any class implementing `IUserEditable` to have a visual form autogenerated for it using a `MemObjDictionary` to store all editable data which is edited by a number of `IEditableType`'s which represent different editors on screen such as a colour picker or edit box. The system integrates well with the Do/Undo system 6.1.

Any class requiring a user editable form should implement the `IUserEditable` interface which provides methods to get the current and default states of the object. This information is stored in a custom built serializable dictionary object which identifies each object by a string name. The class must also provide a list of editors which implement `IEditableType` which are used to edit the state of the object. For example if a colour is added to the objects state dictionary with the text "Line_Colour" then an `Editable_Colour` class with data source "Line_Colour" should be added to the editors list which will add a colour picker to the form generated.

Each editor has a number of validation routines built in by default, for example the `Editable_Int` editable type has validators to ensure that the text entered is not empty and does actually parse to an integer value. Custom validation logic can also be added by providing a custom C# del-

egate to validate the input. All user input is validated before changes can be applied and any error messages are shown to the user in the validation box show in figure 6.6. To help the user all editable types also take a description which is displayed as a tool tip when the user hovers over a particular input box.

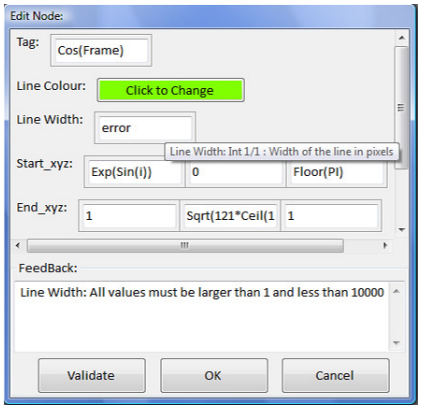


Figure 6.6: An example autogenerated form using the *user editable system*. Note the tool tip help messages, the validation routines and the various NCalc expressions.

Once the user has finished editing the class and has fulfilled all data validation routines, a dictionary of the new state is created and a `UserEditCommand` is created to represent the action of editing the class. This command is then passed to the scenes `CommandHistory` class for execution. This command contains a copy of the old values of the class as well as the new in order to support do/undo/redo automatically across all editable classes in the project.

When the command is executed the class is passed the dictionary containing its new state, from which it should update itself. The system also provides the facility to create new instance of a class from a given state dictionary. When the command is undone the original state dictionary is passed to the class so it can revert to its previous state.

Overall this system has been exceedingly powerful and has saved many scores of hours which would otherwise have had to have been spent building 25 windows forms each with their own validation and do/undo functionality.

For completeness we list the nine `IEditableTypes` which form the components of each user editable form:

- `Editable_Bool` - provides a checkbox to edit a boolean value.
- `Editable_Colour` - provides a button of the specified colour, on clicking the button the user is presented with a colour picker dialog.
- `Editable_Float` - provides an array of text boxes in which the user can enter an array of floating point values, provides min and max validation.
- `Editable_Int` - provides an array of text boxes in which the user can enter an array of integer values, provides min and max validation and ensures the value is indeed an integer.
- `Editable_Selection` - provides a drop down box to allow a user to choose from a limited number of options, the current option is selected by default.
- `Editable_String` - provides a text box allowing the user to enter a piece of text.
- `Editable_String_Array` - provides an array of text boxes in which the user can enter an array of strings.
- `Editable_Expr_Array` - provides the user with an array of text boxes which allow the user to enter an array of mathematical expressions using the NCalc expression language 5.3.

All expressions are validated to ensure they can be parsed - they are not checked to ensure they can be validated as the values of some parameters will be unknown.

- `Editable_Var_Array` - provides the user with a way to edit a mathematical context. A $2N$ matrix of edit boxes is displayed. The first column accepts strings which represent the variable names, the second columns contain mathematical expressions which are the variables initial state. Some variable names may be locked - to ensure they are not changed and validation is done on every row. Blank rows are ignored as the user may enter an arbitrary number of variables.

6.4 Visual Debugging

During the development of this project it became very apparent that it would be extremely useful to have a visual display of what pipelines were actually generated and the primitives and execution context which were the output of each stage of each pipeline. To facilitate this a Visual Debugging option was added to the animation node 7.6 and the form shown in figure 6.7 was created:

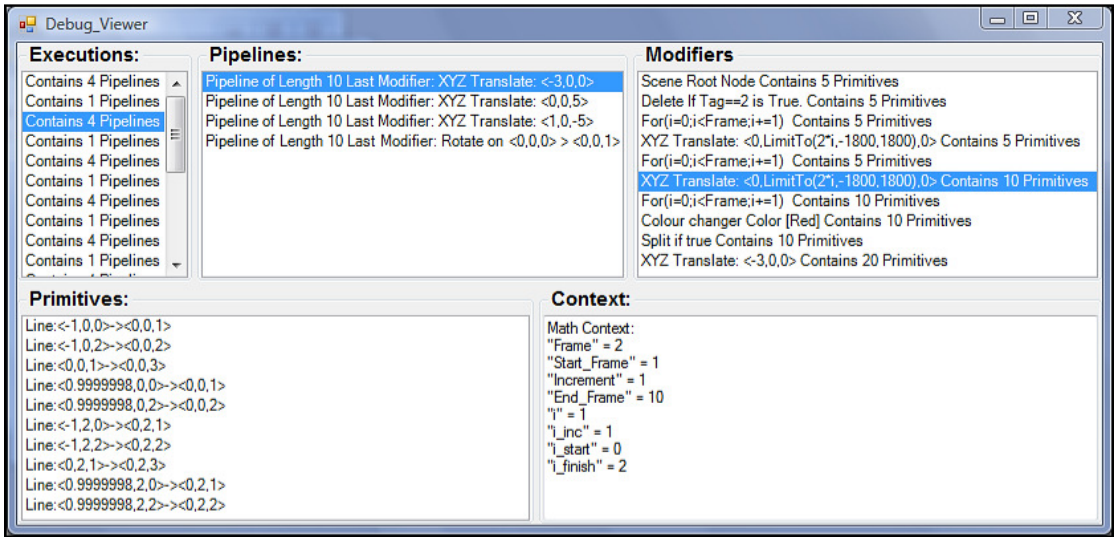


Figure 6.7: The debug form allows the quick navigation of all executions, pipelines, modifiers and primitives generated during an animation. The modifiers and primitives are user editable in the normal way to allow access to full information. The `Execution_Context` is also displayed.

The debug data is collected by overriding the method for removing a modifier from a pipeline once that modifier has been executed, whereupon a snapshot of the pipeline state is taken. The pipeline shown includes the result of a For loop node which repeatedly adds the loop body and the for loop to the pipeline as described in section 7.7.4. The execution list includes all sub-executions - for example the execution of "build up" sub trees. The modifiers and primitives are editable by double clicking to allow access to full information. The `Execution_Context` is also captured and displayed which is useful for working out in which iteration a primitive is generated. This has proved a very useful tool!

CHAPTER 7

Language Definition

In this section we give an overview of the features within the language. This could be seen as a detailed user manual for the language. We will work through every node available to the user in the Toolbox as shown in figure 7.1.

We note that one of the prerequisites of advanced features of the language is a good grasp of the power of the mathematical scripting language implemented as this underpins many of the features we will now explore. A detailed overview and full function listing can be found in section 5.3.

The language implemented in the work consists of a variety of modifiers and primitives which appear in toolbox. Users build up their scene by dragging and dropping modifiers and primitives from the toolbox into their scene tree.

7.1 Primitives

As this project is only a proof of concept only a few primitives have been implemented, namely lines and triangle. However additional primitives such as spheres and meshes could be implemented quite easily. Primitives form the leaves of the tree structure of each scene and are passed through a pipeline consisting of the path from the root node down to where the primitive resides in the tree.

In fig 7.2 we can see the editor form for a line which is auto generated when a user wants to edit a line. The fields listed are:

- **Tag** - This is a numerical expression - either a number or a mathematical expression based on the available variables. The usefulness of the Tag system is described in section 5.6.
- **Line Colour** - This represents the colour of the line and is editable via a colour picker form opened when the button is clicked.
- **Two Points** in 3d space. The X,Y and Z coordinates of each point is actually a mathematical expression, though it is limited to reasonable bounds (± 3600) so that the lines can be generated based upon the mathematical context for example a line may move based upon the current frame number.
- **Time and spartial dimensions** there are integers which identify the point in multiple dimensions, due to time limitations and the method of animation used; these fields are somewhat redundant.

A triangle has a very similar editing system though with an extra Point.

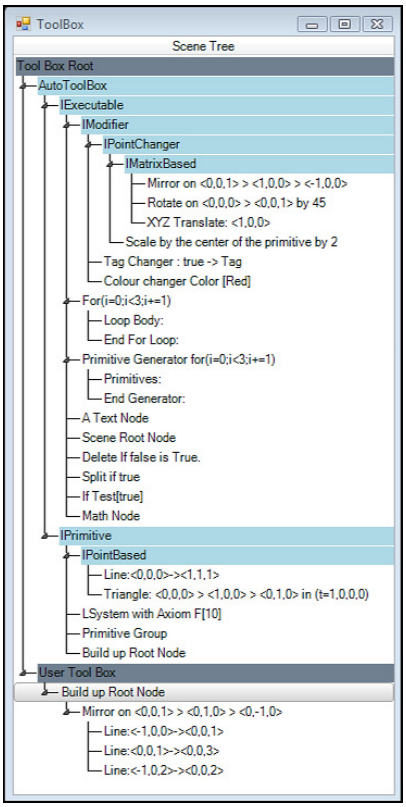


Figure 7.1: The full toolbox provided to the user, all nodes are preconfigured to their default values and are thus ready to use. Note the “User Tool Box” which allows the user to store their favorite snippets of scene tree for later reuse.

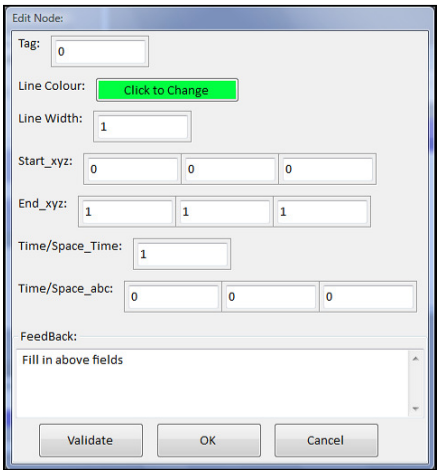


Figure 7.2: The editor for a line showing its various fields.

7.2 Advanced Primitives

Along with the basic primitives available a number of more high level constructs are provided to help users construct scenes. The first of these is a primitive group node which allows primitives to be grouped under a new node in the tree. This helps to give semantic groupings to large number of primitives. Each primitive group node has an editable text field allowing groupings to be named "Arm primitives" or "Wheel triangles". Two other means of generating primitives are provided:

7.2.1 Primitive Generator

This is a node which allows the insertion of primitives halfway through a pipeline, in the example below a line is inserted between the mirror node and the colour changer node. As seen in fig 7.3 node has two static children (which can't be edited, deleted or moved), the first of which contains a list of primitives to be inserted, the second of which provides the place to add on nodes which follow the primitive generator in various pipelines. The primitive list may contain Primitive Groups or Buildup Nodes as well as just plain primitives. The primitive generator also has similar functionality to the for loop node discussed in 7.7.4. This allows multiple instances of a given primitive to be created, simply by evaluating the mathematical expressions representing the points within the primitives with different values for the loop variable provided in the Primitive Generator. Overall this is a useful functionality and allows for a pipeline to have fresh input at multiple stages as show in fig 7.4.

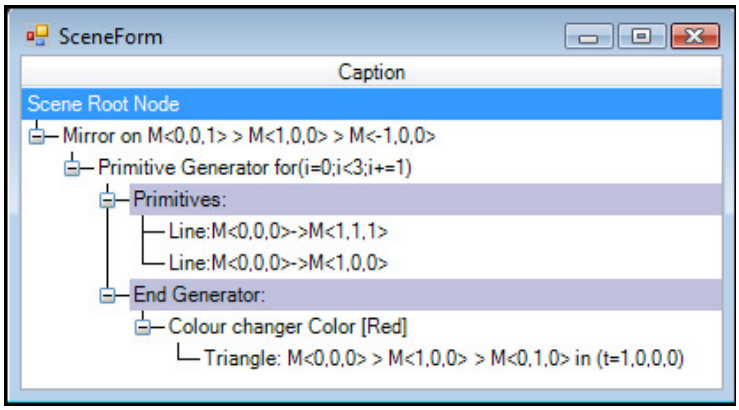


Figure 7.3: An example of the Primitive Generator in use. It allows primitives to be inserted in the pipeline after the Mirror node but before the Colour Changer node.

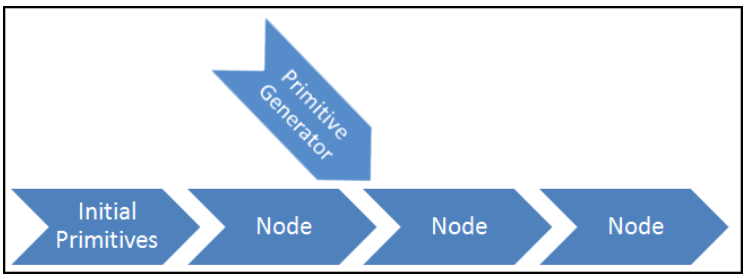


Figure 7.4: Data flow for a pipeline including a Primitive Generator, allow multiple sites of input.

7.2.2 Buildup Execution

As described in section 3.3 we utilise two different semantics for the tree structure one of which "Builds up" larger primitives from smaller ones, and the other is the downward pipelined execution which is the default. The Build up execution semantics are implemented via the insertion of a BuildUp node which swaps from the pipelined execution to the build up execution. The build up execution works by reversing the normal pipeline building mechanism so that all pipelines flow from the leaf nodes up to the root Buildup Node. The primary advantage of including build up semantics is that it allows the easy creation of more complex models from a set of primitives. The more complex models can then be used in other models, for example in creating a column of stick men.

Of course under the build up nodes multiple pipelines may lie, all of which have the possibility of using the same modifiers as in a normal pipeline, though of course their execution will flow back to the build up node. It is also possible to embed other buildup nodes within a buildup node. This is because these buildup nodes are treated as separate executions which are carried out prior to the execution of the execution for which the buildup node is a primitive. The buildup node contains a set of mathematical expressions (similar to a math node 7.5) which allows for variables to be defined only within the sub tree (and indeed the sub execution) of the build up node. Fig 7.5 shows a build up node in use, the short pipeline containing a mirror node ensures that the stick man is symmetric and has two arms and two legs (when only one of each is actually user defined). The stick man could then be used as a whole within the scene.

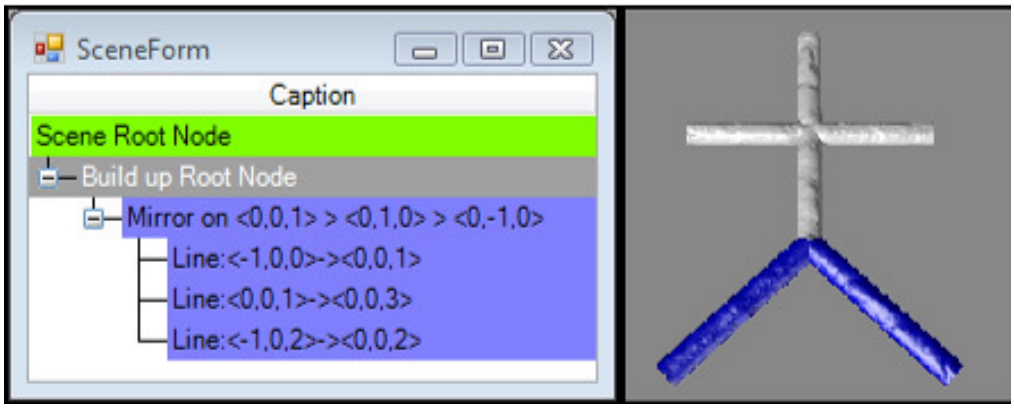


Figure 7.5: A scene making use of a Buildup Node to create a more complex primitive by using a mirror to make the stick man symmetric.

One possible extension of this semantics is the introduction of Constructive Solid Geometry (CSG), which allows the construction of interesting geometry by forming a shape by the union of all shapes defined by its child nodes. These child nodes can for example be the intersection of two shapes, the difference between two shapes or the binary AND of two shapes. Providing the correct data types to support representation of such geometry it would not be hard to add this functionality, as long as some way of rendering it is implemented such as polygonising or possibly mapping directly to PoVRay scripts which support CSG natively.

7.3 LSystems

The LSystem node is an advanced primitive, a full description of the type of LSystem we implemented can be found in 5.7. We now discuss how the user manipulates the LSystem node which can be seen in figure 7.6

The first option the user is presented with is the Axiom of the LSystem, the initial string to which all productions are applied. This string must be able to be correctly parsed. The user is

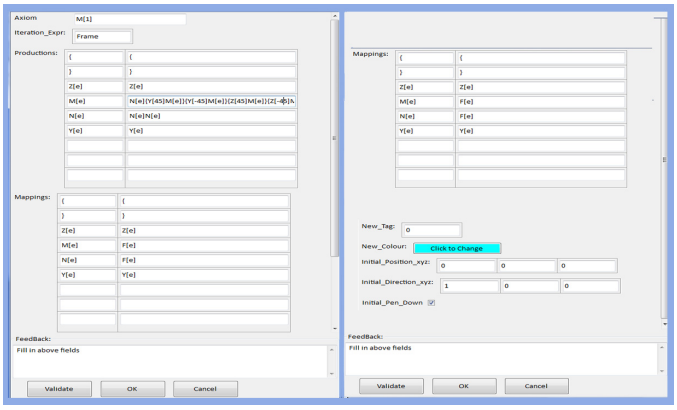


Figure 7.6: The Edit form of an LSystem. The output of this LSystem is shown in figure 5.5.

also given an expression to represent the number of types the LSystem should be evolved - that is how many times the productions should be applied, this must evaluate to an integer.

Next the productions are entered into a string array, these must be entered in pairs (pattern,production). The pattern must contain one letter turtle commands ("{", "}", "+" or "-") or letters and variable names for their parameters for example "A[a]", letters and variable names need not be single letters for example "trunk[height]branch[length]".

When a system is evolved the Axiom is parsed left to right and the longest matching production is used to replace the string matched with the production. All letters in an Axiom MUST match with productions. If more than one production matches a pattern then the longest is used - two identical patterns may not currently be used.

Once the system is evolved the correct number of times a final evolution takes place using the productions specified in the Mappings table. These must be entered according the same rules as for productions. The purpose of the mappings is to transform user letters (eg "trunk[height]") into commands the turtle can understand for example "F[height]", again the parameters in the patterns become variables which can be used in expressions specifying the parameters of letters in the productions.

The Tag of the primitives that the LSystem generates must be specified here, it is also possible to enter an expression which is evaluated prior to the execution of the LSystem. The colour of the LSystem is also specified.

The final set of user editable fields relate to the initial state of the turtle, its initial position, direction and whether or not its pen should start down or up. The position and direction can be mathematical expressions.

7.4 Basic Modifiers

The bulk of the language consists of a series of modifier nodes which act upon the basic primitives defined earlier. Some of these are quite simple such as the Colour Changer or Translate nodes. Modifier nodes form the nodes in each pipeline regardless of the semantics of execution being used to build that pipeline from the tree structure.

As explained in the section on the Tagging system exploited throughout the language each of the basic and matrix based modifiers has a "Tag Test" field which is normally a mathematical expression based on the primitives "Tag" variable which evaluates to true or false and indicates whether or not the modifier should be applied to that given primitive. By default the Tag Test field is set to "true" so that it is applied to all nodes.

The simplest Modifier node is the Colour Changer node which changes the colour of all primitives passed through it which meet the specified Tag Test. The colour is user defined at design time via a colour picker field.

A Scale by Center of Primitive node is also implemented which unsurprisingly scales each basic primitive by its center (ie the point represented by the average of all points which make up the primitive - two for a line and three for a triangle).

To complement the Tag System a Tag Changer Node is implemented which sets each primitives Tag to the result of evaluating a given expression which may include the primitives old tag. In keeping with the Tag system the tag changer also has a Tag Test Expression which allows the user to set which primitives have their tag changed. For example a user could enter If "Tag Test: Tag $>$ 3" then "New Tag: Tag $*$ 2". This is useful for managing semantic groupings of primitives.

7.4.1 Matrix Based Modifiers

A major class of modifiers are those based upon matrices. If one exploits Homogeneous co-ordinates (that is representing a point in 3d by a point in 4d), then a 4 by 4 matrix can be constructed to represent various transforms in 3d space. These matrices are such that multiplying the matrix by the 4d vector representing the point in 3d has the same effect as the desired transform. These matrices are known as transformation matrices and can be constructed for the following transformations for which we have created a modifier node:

- **Rotations** The Rotation Modifier rotates a primitive by a given number of degrees around an arbitrary line defined by two points in 3d space. As expected the number of degrees and the definitions of the two points are all actually mathematical expressions which give rise to some interesting possibilities for example rotating an object based upon the current frame number.
- **Translations** The Translation Modifier translates a primitive by a given amount in 3d space. Again all user input is actually a mathematical expression.
- **Mirroring** Although the math is complex it is possible to define a reflection on a 2d plane in 3d space as a transformation matrix. In the Mirror Node we ask the user for three points in 3d space (which must NOT be co-linear) and then reflect the primitives points on the plane through all three points. If the points are co-linear then there are an infinite number of planes through them and the modifier will collapse to the identity matrix (normally!). Again all three points can be comprised of mathematical expressions though care should be taken to ensure the points do not end up co-linear!

All of these modifiers have the Tag Test attribute to find out which primitives to act upon. They also a boolean value called "Keep Originals" which does exactly what it says on the tin, and is quite useful especially with the mirror node.

7.5 Mathematical Context

A Math node is an opportunity for the user to edit the mathematical context which is passed through the pipeline. This context is comprised a mapping between variable names and their floating point values. The Math node allows existing variables to be redefined and new variables to be defined.

It should be noted that variables are re-evaluated in a top down approach, this has the advantage that variables defined later in the node can use the NEW values of existing variables or the newly defined variable which appear earlier in the node. The user should also note that three new variable definitions are allowed per editing of the node. If you wish to add more variables simply add 3 then save your changes before editing the node again whereupon another three variable definition boxes will appear.

Math Nodes are an example of batch based execution in that all primitives must have been processed by all previous stages in the pipeline before they are all processed as group by the math node. This is done not because the Math node needs to act upon all the primitives in one go (it doesn't) but because we must ensure that no Execution groups proceed to the node after

the math node before the context has been changed for all Execution Groups, otherwise we may end up with inconsistent mathematical contexts being passed through the various pipelines. An Execution Group is a pipeline together with its group of primitives and a an execution context which contains the mathematical context.

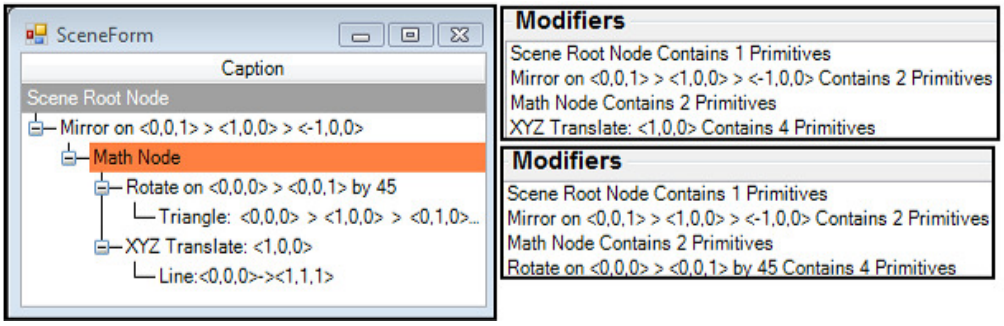


Figure 7.7: An example scene using a Math Node and the two pipelines it generates, the first tree nodes of each are identical and hence will be executed as one pipeline section before diverging.

If you like a Batch based node acts as a red traffic light to all Execution groups so that execution in all pipelines containing the batch based node must finish before the node is executed. Thus the execution of the first two nodes of each of the pipelines show in figure 7.7 must be executed before the Math node is executed. This avoids corruption of the execution context which contains the math context, especially when there are multiple Execution Contexts (pipelines) with the same next node. Figure 7.8 shows the dataflow and the point which all executions must reach in order for the math node to execute.

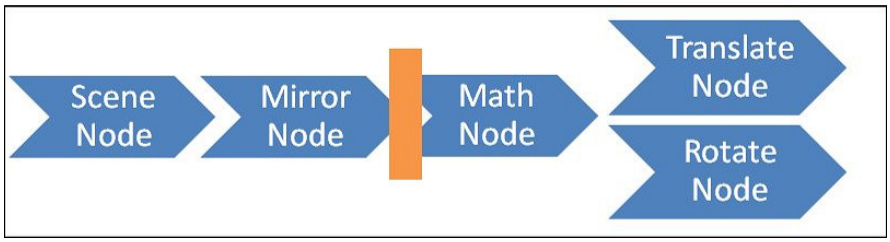


Figure 7.8: The data flow for the scene shown in 7.7

7.6 Animation

In contrast to much of the software discussed in the Related Work section 4 our software supports animation natively. This is done by exploiting the mathematical context which allows a Frame variable to be passed through the scene tree. This is done via an animation node which supports three mathematical variables, “Start_Frame”, “Increment” and “End_Frame” all of which are arbitrary mathematical expressions though they should all evaluate to (ideally consecutive) integers. As you would expect Start_Frame is the initial frame number. The frame number is then incremented by the increment and a frame is generated, this is repeated until the frame number equals or is larger than the End_Frame expression. A standard set up would be to have Start_Frame $= 1$ Increment $= 1$ and End_Frame $= 5$ which would produce frames $\{1, 2, 3, 4, 5\}$.

The Animation Node also allows for additional variables to be defined, for example the user may want to define the size of the image so could enter variables for X_Min,X_Max,Y_Min,Y_Man,Z_Min and Z_Max which can then be used in the scene to set the size of geometry. A further use could defining a variable for Level of Detail such that if level of detail is high (3 say) then extra geometry is generated which would not have been generated if Level of Detail was lower (1 say).

There are many other possibilities for example the provision of random seeds which would be very useful for the creation of different scenes for each level of a computer game.

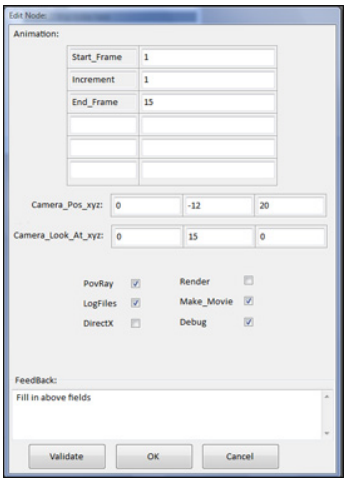


Figure 7.9: Editing the animation parameters before a scene is rendered.

7.6.1 Rendering Options

As seen in figure 7.9 the animation node also has several options of the rendering of the scene. These options are as follows:

- **PovRay** This option takes the output of executing the scene and generates PovRay scripts from it (one for each frame) so that the scene may be rendered using the PovRay Ray tracer. More information on PovRay is detailed in section 4.2
- **LogFiles** This option is primarily for debug purposes it outputs all pipelines built from the scene to text file, as well as logging the final set of primitives generated to text file. It is this set of files which is compared when running the end to end Test Suite built into the software.
- **DirectX** This option renders the scene / animation to a built in renderer based on Microsoft's managed DirectX Framework [22]. This rendering option is far faster than the ray-traced PovRay rendering.
- **Render** This option turns on the rendering of each frame individually as appose to just generating the scripts without rendering them or just rendering all the frames in one animation.
- **Make_Movie** This options ensures that a bitmap file is created of every frame rendered. These bitmap frames are then stitched together using a movie maker tool called EasyBMPtoAVI [23]. This functionality does not appear in any of the related software we reviewed.
- **Debug** This is a highly useful feature which creates detailed information about what the execution engine is actually doing. It lists the execution for each frame (and any sub executions from BuildUp nodes) as well as the list of pipelines which comprised that execution. For each stage in each pipeline the set of primitives it produced is stored so that users can track where certain primitives were created. Each modifier and each primitive is editable in the normal way so as to allow the user to have full access to the state of the primitive or modifier. Also the mathematical context at each stage in each pipeline is stored to allow users to debug any problems they have with their mathematical scripts. The storage of the pipelines allows the user to easily track the data flow through

the program. Of course gathering so much information does slow down the execution and should not be used if the user does not require debugging!

7.7 Pipeline Flow Nodes

In order to make working with pipelines more straightforward and to enable inter pipeline communication we provide a number of nodes which modify pipeline flow. These can be divided into two categories those that act upon primitives (Filter and Splitter Nodes) and those that act upon the actual pipelines (If and For loop nodes) :

7.7.1 Filter Node

This node provides similar functionality to the If Node 7.7.3 however it functions upon primitives. It contains a user editable Tag Test expression which evaluates to *true* or *false* for each primitive in the pipeline. The actual functionality of the node is deterred by a user editable check box "Delete_On_Match" which has the following semantics:

- If "Delete_On_Match" is *checked* then any primitives which make the Tag Test evaluate to *true* are deleted. All other Primitives are allowed to continue down the pipeline unaffected.
- If "Delete_On_Match" is *unchecked* then any primitives which make the Tag Test evaluate to *false* are deleted. This means that primitives are only kept (or allowed to continue through the pipeline) if the meet the Tag Test.

Both semantics are provided to aid expressibility. The caption of then node within the tree changes depending on the semantics to read "Delete If *expression* is true" if "Delete_On_Match" is *checked* and "Keep if *expression* is true" if it is unchecked. We trust this semantics is not confusing but allows the user more expressibility and makes the tree far more readable.

7.7.2 Splitter Node

The Splitter Node also acts upon the primitives in a pipeline. It allows primitives to be shared between the pipelines which include the splitter node. The Node contains a Tag Test expression which is user editable. The intuition behind this node is that any primitives in any one of the pipelines which include this node should be copied so that they appear in all the pipelines which include this node. An example use of the splitter node is show in figure 7.10. The data flow diagram in figure 7.11 should help us to understand what this node does. The three pipelines are executed independently until reaching the batch based Splitter node at the orange bar. The splitter node is then executed by compiling a list of all primitives in each of the three pipelines which meet the user specified Tag Test. Once generated a copy of these nodes is then added to the primitive list of each pipeline and execution continues as normal. The Splitter node is useful for exchanging primitives between pipelines, although it does incur the cost of a synchronisation point between a number of different pipelines which would be executed by different threads.

7.7.3 If Node

The purpose of the If Node is to provide a stop gate in the pipeline. The If Node contains a user editable mathematical expression which should evaluate to a boolean. If this this expression evaluates to *true* then the node does nothing and execution continues to flow down the pipeline as normal. However if the expression evaluates to *false* then the pipeline is truncated and no further nodes are executed. In this case the user is able to choose what happens to the primitives already generated via a user editable check box named "Kill_Primitives". The primitives generated so far can either be discarded or taken as part of the final scene being generated.

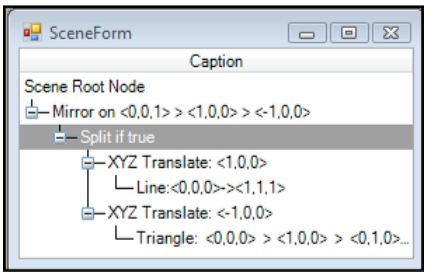


Figure 7.10: An example Scene Tree with a splitter node - all primitives reaching the splitter node will be added to each of the pipelines finishing with the translate node.

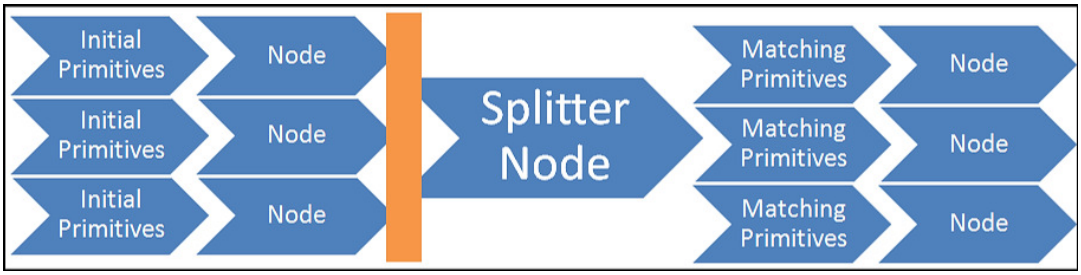


Figure 7.11: The Data flow between three pipelines each containing the same Splitter Node.

7.7.4 For Loops

For loops are another feature more normally found in imperative programming language. We implement for loops as a means of succinctly increasingly the length of pipelines. It allows the user to repeatedly apply a linear series of nodes with a varying variable. For example in figure 7.12 we repeatedly apply a rotation to any primitives within the pipeline. Since both the original and the rotated primitives are passed onto the next node we end up with an arc of repeated primitives. The For Loop provides the user with the ability to rename the loop variable to make a scene more understandable and to allow the possibility of nested for loops. The Node also provides three editable expressions for "Start_Value", "Increment" and "Finish_Value", each of these can be an arbitrary mathematical expression.

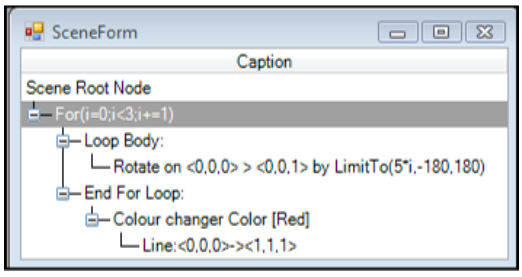


Figure 7.12: An example Scene Tree with a for loop which repeatedly rotates any primitives passed through the pipeline. Note the loop bounds $i = 0; i < 3; i+ = 1$ which control the number of iterations. The two child nodes of the for loop are Static nodes 7.8. All loop bounds and indeed the name of the loop variable are user editable with the loop bounds being arbitrary mathematical expressions.

When the For Loop is executed it adds a copy of all the modifiers listed under its Loop Body node to the pipeline together with a copy of itself. It then adds the loop variable and bounds to the mathematical context. When the loop body has been executed and the For Loop node is executed again the process is repeated provided that loop variable has not exceeded the loop

bounds. The loop body is added again and the mathematical context is updated to reflect the new value of the iteration variable. When the For Loop Node is executed for the last time the loop variable and bounds are removed from the mathematical context. Since the For Loop node modifies the mathematical context it is a Batch Based node (see section 3.4). Figure 7.13 shows the result of executing a For Loop on the pipeline that actually gets executed.

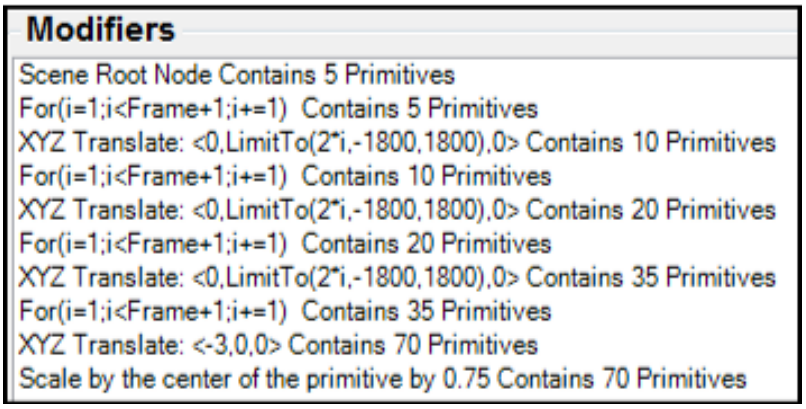


Figure 7.13: The Debug representation of a pipeline which uses a for loop to translate a set of primitives mulitple times. After every iteration of the loop the for loop is added again so that it can update the mathematical context with the new value of the loop variable (and test for the end of the loop!).

7.8 Miscellaneous Nodes

Finally within the language there are a number of utility nodes which have little effect on the scene but are integral to the working of the language. These nodes are:

- **Text Node** This node has no effect on any primitives passed through it and may be placed anywhere in the scene tree. It simply allows the labellings of various branches of the trees via a user editable caption.

- **Static Node** This node is similar to the text node however it is used in the creation of the more advanced nodes such as For loops 7.7.4. These have two static nodes, the first of which contains a list of modifiers (child nodes) which form the loop body. The second static node provides the point at which the rest of the pipeline is attached. Static nodes are not user editable, nor can they be dragged and dropped or deleted since they are part of their parent modifier and can be moved and deleted only with that modifier.

- **Root Node** This node is the root node of every scene as such it has a number of different implementation details. However form the user perspective it carries out a similar function to a Math node allowing the mathematical context to be set up at the beginning of the execution of the scene. This node is also not able to be deleted or dragged and dropped for obvious reasons!

The code base is sufficiently extensible that new nodes can be implemented fairly quickly. For example at the end of the project the Filter node 7.7.1 was implemented in around 90 minutes. Care should be taken in any future work to ensure that all nodes work well with each other and that the user is not presented with such a plethora of nodes that they becoming overwhelmed and confused. Of course classifying nodes into levels of difficulty would be one way around this, since the basic modifiers (translate, mirror and rotate) are quite straightforward and can be learned quickly.

Should the reader wish to find out more about the use or execution of one of the nodes presented we refer to the source code which is quite well documented thanks to C#'s XML documentation system. The Test Suite built into the application should also contain at least one functioning example of each node being used and together with the debug system is a good way to get an intuitive handle on how the node executes, especially as one can step through the execution of a pipeline using the debug system. The user edit system also provides pop up hints or tool tips on each field which is user editable.

CHAPTER 8

Evaluation

As stated in our introduction this project had two major goals:

1. To create an easy to use graphical programming language with a typesafe drag and drop interface which is simple to learn, easy to use and yet very expressive and effective for the purpose of sand boxing procedural algorithms.
2. To bring together several of the most promising procedural algorithms techniques into one unified language environment and to support animation from the get go, something that many previous procedural systems have lacked.

At times during this project it felt like these goals were mutually exclusive, as implementing powerful procedural formalisms was challenge enough without trying to make them simple to use. As an example the introduction of LSystems was an interesting challenge to implement especially with the bracketing and mathematical expression parameter system. To then make this system as straight forward for the user to use as possible added another layer of complexity.

Thankfully Visual Studio 2008 and C# 3.5 made the task of meeting both project goals somewhat simpler by allowing the creation of a user editable system for automatically creating a windows form for the editing of the class. This system generates the form directly from class and integrates well with the command history system we implemented to allow the user to have full do/undo/redo functionality.

We feel we have achieved both of the goals of this projects, although there is much further work that could be put into these areas. Firstly however we should apply the same criteria to our solution as to the modellers reviewed in this report:

8.1 Problems & Solutions

In the motivation chapter (2) we identified a number of problems with the current generation of interactive and scripted or procedural modellers. It is important that we evaluate our own software with the same criteria as we applied to that software:

The primary problem identified with interactive modellers is the complexity of the interface which is off putting to new users and complex for advanced users primarily because of the plethora of tools each of which the user must understand and learn to apply separately. Scripted modellers fell foul of a similar problem since before the user can make use of the modeller they must spend a great deal of time learning a custom language which is often a non-trivial task.

The interface to our project is hopefully more straightforward. By utilising a graphically manipulated tree structure to contain our language the user does not have to build up skill at applying tools interactively in 3d. By the use of a TypeSafe drag and drop system we ensure that

the user can only create legal trees which avoids a lot of the complexities of scripted modellers. Finally most of the modifier nodes are quite simple to use e.g. translate by amount in x, y, and z directions. This is especially true as a uniform layout of editing forms is implemented via a form auto-generation library we implemented. This ensures uniform layout with robust help messages and validation routines as well as descriptions of every field the user must enter. Should additional help be required for more complex composite nodes such as For loops there is much documentation in this report and examples are provided via the end-to-end test suite we implemented, however we feel that most of the nodes are quite intuitive - something we would like to keep in further modifications to this system.

Perhaps the least comprehensible part of the system is LSystems, a formalism which must be learned before it can be used. Unfortunately we can see no way of making this system more user friendly, although we do implement it in the standard user editable form system.

Another problem with most modelling systems is that scenes must be created in linear fashion by a succession of user decisions. Should they wish to undo one of the earlier decisions all subsequent decisions must be undone. This is a real problem with interactive modellers and not quite such a problem with scripted modellers. Our procedural framework allows the user to edit any of the nodes they have created so far in any order with without having to undo their work - this saves the user much time and increases productivity.

Coupled with this problem is that of a lack of reuse. Interactive modellers do not allow the user to reuse a set of actions at a later point, unless by complex and brittle macro systems. Scripted modellers also do not allow such reuse except by reusing the outlines of various scripts. Our Toolbox system allows the user to save sections of scene tree which they would like to reuse simply by dragging them into the toolbox. The user is thus able to save a larger modifier such as mirroring on all coordinate axes or a certain For loop. Of course the TypeSafe drag and drop system ensures that when the snippet of tree is reused the use is actually legal. This system aids productivity and is a unique feature amongst modelling software.

Finally the primary problem with interactive modellers is that the complexity of a scene is governed by the skill of the artist and the time available. The solution to this is procedural modelling which our system exploits heavily allowing the creation of complex scenes through the algorithms specified in the pipelines.

8.2 Ease of Use

As discussed above the first goal of this project was to create an easy to use procedural modelling language. We implemented a large number of features which make the software as easy to use as possible, these include:

- A TypeSafe drag and drop system which ensures that the user can only build legal scene trees.
- A save/load system.
- A do/undo/redo system which enables users to undo any mistakes.
- A toolbox system which allows the user to reuse snippets of scene trees.
- A highlighter system which allows the user to highlight the features they are editing.
- A visual debugging system which shows the user exactly what the execution engine is doing.
- A full end-to-end test suite is provided firstly to validate the software and also to provide a full demonstration of the power of the software.

We have found the system exceedingly easy to work with during this project, being able to demo new ideas for scenes to demonstrate functionality in under ten minutes. We also invited a number of technical and non-technical to use the project and complete a simple tutorial after a

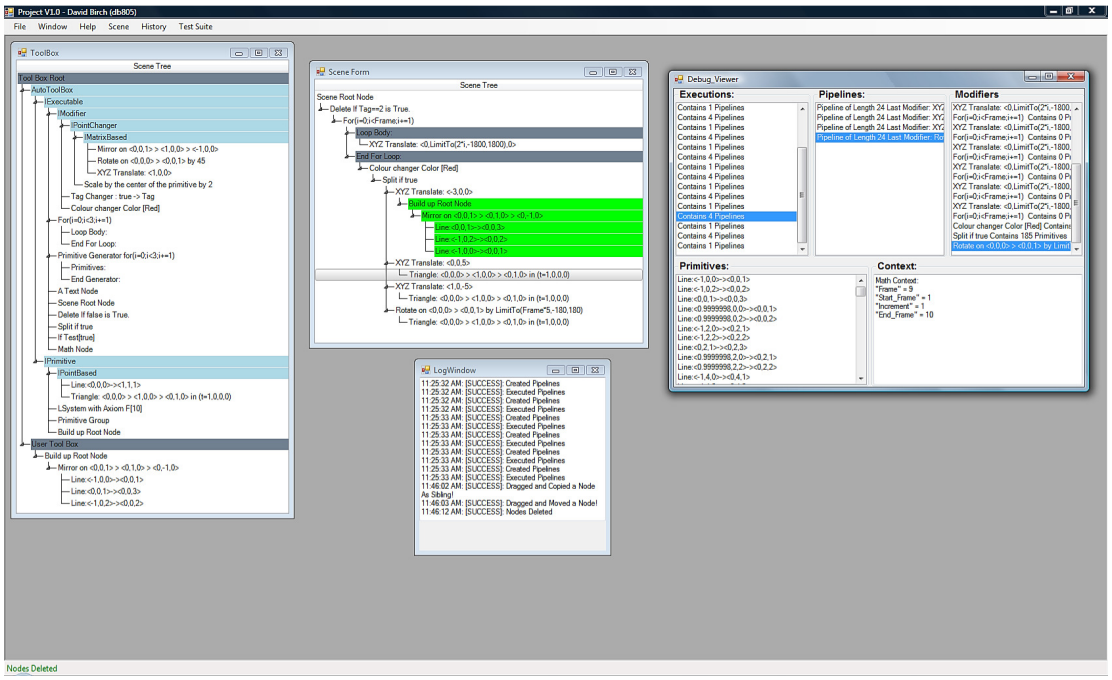


Figure 8.1: The interface of the project as discussed in chapter 6.

brief introduction to the software. Whilst the sample size was too small for statistical analysis many useful conclusions and observations were shown.

Users generally found the interface straightforward to use with the typesafe drag and drop system consistently being rated excellent and the concept of a toolbox was understood by all. Some users struggled with the concept of hierarchies in a tree structure - however once explained the concept was very quickly grasped and presented no further problems

The user edit system for producing uniform input forms was well received and users were able to apply what they learned in editing their first node in editing all subsequent nodes. This helped to greatly reduce the learning curve of the software, especially as the alternative would be custom edit forms for each type of node. Similarly the do/undo system was automatically grasped via the normal keyboard shortcuts (even though the users were not told there were any!).

A number of issues with the interface were discovered, many users wanted to use the delete button to delete nodes which is not currently implemented (the delete button is on the pop-up menu), similarly several users tried to press enter in an input box to close the input form. These two useful observations will be implemented into the project.

One user compared the software to industrial CAD software which was said to be more complex although the interactive modelling environment such software provides is a huge advantage over the current system and thus should be one of the first pieces of further work to be implemented.

Another user was able to edit the mathematical scripting language well, although some frustration was experienced with the case sensitivity of the language (something which could be fixed quite easily).

In summary the software would require some user training before competent use. This training should be quite quick as all users seemed to pick up the interface of the software very quickly, although the underlying concepts of pipelines took a little longer to grasp and would

thus form the bulk of any tutorial or training. The learning curve for this software is likely to be far smaller than that of scripted modellers, although lacking an interactive element is probably slightly steeper than some interactive modellers.

8.3 Unification & Utility

The second goal of this project was to unify a number of procedural frameworks so that when utilised together they would produce novel results. We have brought together a number of procedural frameworks including two forms of tree based languages, LSystems, mathematical scripting and some constructs from imperative programming languages.

All these formalisms are embedded into the pipeline based scene trees which the user composes. These are executed by the creation of multiple pipelines consisting of a number of modifier nodes. These pipelines run from the root of the scene tree down to a group of graphical primitive nodes residing at the leaves of the tree. Pipelines will often share common first sections allowing control over the entire scene before specialising to effect control over a subset of the scene which is passed through that pipeline. The modifier nodes in the pipeline act upon the primitives by for example translating or mirroring them. Certain nodes must be executed on all their input in one go whilst others can execute on each primitive independently (Batch and Stream based execution). These two execution semantics are exploited to allow multi-threaded execution of the pipelines.

The number of formalisms implemented gives rise to some very interesting possibilities, for example the LSystems formalism we have adopted allows mathematical scripting to be inserted into the parameters of the productions used to evolve the Axiom. These parameters can be based on variables being based through the pipelines of the scene tree, which can be modified by math nodes, For loops or If nodes. The latter nodes being constructs more normally found in imperative programming, in our case they modify the flow of the pipeline by either truncating it or by repeating a number of nodes.

We also allow the user to embed the LSystem generator into a Buildup subtree which executes with execution semantics that represent a model by the root of the Buildup tree and form the model by the recursive composition of the child nodes. This is similar to systems presented in Blob Tree (see section 4.5) or of Computational Solid Geometry. It is also similar to the main pipeline based system but built and executed in reverse (flowing from leaves to root and not root to leaves is as normal).

Another common usage scenario is the use of a Buildup node is to create a complex model which is then sent through a number of pipelines to create an entire scene. For example a buildup tree and an LSystem could be used to create an oak tree model, a pipeline could then be used to produce an entire forest scene.

The deep embedding of animation within the language is also unusual in procedural modelling, we utilise the mathematical framework which is deeply integrated in the project to provide a Frame variable which users can exploit to provide animation. As a proof of concept we implemented a sine wave animation based upon the frame variable.

We believe these usage scenarios and the close integration developed between the various formalisms shows that we have been able to unify a number of formalisms together in a workable way. In the further work chapter we give a number of possible extensions to the software, including exploiting massive parallelism on modern General Purpose GPU's. We also present a number of further procedural formalisms which would integrate well with the current system.

8.4 Performance

The final critical metric by which graphics software must be evaluated is that of performance. This is somewhat hard to quantify as the time taken to render a frame of animation is almost totally dependent on the actual pipeline being use and the types of node it contains. A full performance study looking at the speed of each node and the result of various pipelines is

Execution (ms)	Pipeline Constuction (ms)
39	11
38	11
37	11
38	11
38	12
36	10
41	11
41	12
37	11
38	10

Table 8.1: Timing for rendering 10 frames of the sine wave animation

unwarranted as this software has not been profiled nor fully optimised. However to give an idea of performance we will follow a few examples. All executions occurred on a Quad core Intel Core two duo Q6600 running at 2.4Ghz with 2Gb of ram running windows vista 32bit.

Firstly we consider a small application which renders 2 sine waves each consisting of 60 primitives. The pipeline construction phase produces two pipelines each of which produces two sine waves. Each pipeline is long and linear at 67 steps which perhaps explains the slow execution times shown in table 8.1. The average time for pipeline construction is 11 milliseconds. whilst the average execution time is 38.3 milliseconds. So the total average runtime per frame is 49.3ms which is 20.28 frames per second, a little slow perhaps but given the length of the pipeline it is not bad.

A larger example can be seen in table 8.2 which renders 4 columns of marching stick men, each column column contains 1000 or so primitives. Each column is rendered by a different pipeline of length 36 and thus by a different thread. A separate Buildup execution is carried out to construct the stick man. The total average execution time is 297.6ms or about 3.3 frames a second which is quite slow probably due to the threading overhead. The Buildup execution is far quicker achieving 67 frames a second were it to be executed alone.

Finally we ran a larger example rendering an orchard scene populated by LSystem trees. The final image in the scene consisted of some 131,036 primitives based on translating an LSystem with 5000 primitives. The execution of the final frame was single threaded and quite slow taking around 4 seconds to generate the LSystem and 10 seconds to execute a pipeline of length 19 which increased the amount of data by some 26 times! It was gratifying to see that execution algorithm was able to stand up to such a test and with suitable multi-threading and some optimisation of the scene tree could be made much faster.

In running these tests we noticed that quite often the first run was up to 9x as slow as the median run,this is likely to be due to the .Net runtime (the Common Language Runtime or CLR) just in time compiling the code. During the results presented above we have ignored the first 2 frames to avoid these data points. It was however interesting to note that the rendering times were very constant and that the garbage collector did not appear to interrupt the program.

It is clear that this language does not currently possess enough performance for integration into modern games engines. There are a number of reasons for this, firstly that the codebase has not yet been profiled and performance enhanced, and secondly the choice of language C# is not known for its speed of execution since it runs in a virtual machine. However this said, the application is more than fast enough for the purpose of sandboxing procedural algorithms quickly and efficiently without the user having to write masses of low level code. The ideal usage scenario would be for an artist to spend time in the program quickly and efficiently creating procedural graphics algorithms and tweaking them through the easy to use interface before handing the scene tree on to a programmer to implement in low level code. As discussed in the further work section it seems quite feasible that automatic translation to C code could achieved which would greatly increase the speed of execution.

Execution (ms)	Pipeline Constuction (ms)	
263	30	
8	7	308
243	52	
10	7	312
223	32	
9	6	270
317	30	
9	6	362
241	50	
9	7	307
241	32	
8	6	287
242	30	
8	6	286
229	37	
8	5	279
239	36	
9	6	290
222	38	
7	8	275

Table 8.2: Timing for rendering 10 frames of a marching column animation. The short executions are the execution of a Buildup sub-execution. Total frame execution times are given in the third column

CHAPTER 9

Future Work

As with any large software engineering project there are a large number of features which did not get implemented. In this section we detail a number of ideas, features and modifications which could be made to the language to have it meet its goals.

9.1 Additional Primitives

In order to make the language more expressive a key step would be the inclusion of more primitive data types. For example the addition of various 3d geometric shapes would be useful, spheres, cuboids and so forth should be included. This would be fairly straightforward as long as a simple means of representation as a number of 3d points could be found. A means of rendering the output in both PoVRay and DirectX would also need to be found, the latter would probably involve some form of polygonisation which is less than trivial to implement.

Most modelling software exports models to a number of common model representation formats such as .obj or Microsoft's .X file format. It would be useful for reasons of interoperability to allow the user to import such models (and perhaps export to these formats). Once imported these models could then be acted upon by pipelines in the standard way either via decomposing the model to a triangle list or introducing a new data type to store it. Care would have to be taken to allow the new primitives to be textured and to ensure that the texturing is maintained intact after the application of each modifier. Similarly the vertex and face normals used for rendering would need to be recomputed at some point in the execution.

Currently the language does not support any curved geometry other than that composed of a number of lines/triangles. It would be useful to add curved surface support for example by using splines for curves and NURBS (Non-Uniform-Rational-Basis-Splines) [25] for curved planes. The latter is an industry standard way of representing and exchanging curved surfaces. To edit these a specific editor would have to be built and/or importing of NURBS and splines would need to be implemented. A new set of modifiers to act upon such geometry could then be included to allow more interesting effects, however many of the current modifiers would also work on NURBS and splines by modifying their control points. Implementation of these primitives would be time consuming. However there is an open source .Net library provided by Rhino a company which makes professional NURBS editing software which should allow easy reading and writing of their .3DM file format for NURBS.

9.2 Additional Data Types

Within the tree the only form of data allowed within the pipeline is a set of graphical primitives. It would be very interesting to add two further types of data:

- Adding 2d graphics would be interesting. If one could generate a 2d texture for example from an LSystem then the texture could be passed through the pipeline and applied to a number of primitives. It would also be interesting to implement a number of 2d image transforms such as blur and sharpen to act upon the 2d textures. There has been some interesting work on such procedural textures and it would seem a logical objective to try to integrate such work into this project [33].

- Most graphics scenes have large numbers of lighting sources, to this end primitives representing lighting sources with their various parameter. Should be introduced. They could for example be represented as a line with one end being the light position and the other the direction in which it shines. The light could then be passed through the pipeline in the normal way. This would be useful for example in building a street scene with lampposts. Other parameters for colour, and intensity should also be added.

9.3 LSystems Extensions

There are a plethora of types of LSystems in existence we have implemented the most common and most user friendly system. There are a large number of extensions which could be made to the LSystems implementation in this project including:

- Within a branching LSystems it would be possible to split work between threads at each branch. So that when a branch occurs (a"[") one thread takes the branch and the other skips on ahead to the return site (a "]"). Efficient implementation of this would require a good work splitting algorithm and also an efficient data structure that stores forward links between each "[" and "]" so that skipping through the axiom does not constitute such an overhead as to make the scheme unworkable.

- The implementation of *Stochastic* LSystems which allow multiple productions to match a given pattern. The actual production to apply is then randomly chosen based on the weights attached to each rule. This allows randomness to be introduced into the LSystem to more faithfully represent real plants. This could be implemented in our system with each weight actually being an expression to be evaluated allowing the weights of the rules to vary based upon the parameters of the of the pattern matched. Extending this idea another useful way of selecting rules would be to provide an "Apply If" expression which would denote whether or not the production should be applied even if it does match the axiom. It would also be interesting to be able to allow the expressions in each production to have a variable for the current iteration number, this could prove useful for changing a systems behavior as it evolves.

- Incorporating some of the ideas from LinSys3d discussed in section 4.1 would also be interesting, for example the mapping of alphabet letters directly onto primitives could be highly useful especially if it was possible map it onto a set of primitives. One could extend this idea still further by mapping other alphabet letters directly onto modifier nodes such that the LSystem is no longer rendered by a Turtle but instead generates a standard pipeline which could be executed in the normal way. Given the exponential nature of LSystems it may be wise to only use this idea on short LSystems! This idea would also require extending the number of parameters that the LSystem supports per letter (at the moment this is only 1).

Once again we see that the procedural framework of LSystems fits very well into the project and if extended could make the system even more powerful.

9.4 Additional Modifiers

A good part of the expressibility of this language is made up by the number of modifier nodes we implement. It seems worthwhile to attempt to implement several more to give the user more power. Though care would need to be taken not to implement such a plethora of nodes that the user becomes lost and confused.

Examples of new modifiers that could be included when 3d primitives are introduced would include extruding an area of a model, twisting a model or stretching a section of model. It aught also to be possible to remove sections of models. The difficulty in implementing some of these is that they require quite complex data storage types to cope with these transforms.

There is also scope for implementing more exotic modifiers which act upon sets of primitives. For example joining sets of lines into sets of triangles or vise versa. This would provide some interesting effects useful for special effects in games.

9.5 Tree Structure Modification

One limitation on the structure of the current language is that composite nodes such as for loops can only have a list and not a tree of "body nodes". This means that a For loop can only loop through a linear section of a graph and not a tree. This is unfortunate as it limits the expressibility of the language. Much time was devoted to generalising the execution algorithm to fix this however time eventually ran out trying to ensure that batch based nodes would not be fired too early. Another week or so modifying the algorithm would bring it to completion.

Another highly inivative idea for making the tree structure as easy as possible to use is found in Sketch Based modelling a system discussed in section 4.6. In this software the idea of "linked copy and paste" is implemented such that the user can copy one part of a scene tree to another area of the same scene tree in a manner that links them. After this any edit to either of the two copies is automatically reflected in the other copy. Of course this idea can be extended to more than two copies. This idea seems worthwhile to implement as it eases the amount of tree structure that the user has to maintain which should help productivity.

The addition of case based filter node would be a useful addition. It should be implement as a "Case Node" which has a number of Filter nodes as its children such that all primitives passing the case node flow into one or more of the pipelines starting with a child filter node. Currently this effect can be achieved by use of a splitter node and then a set of filter nodes, but it could be more efficiently implemented as a new modifier.

Whilst a For loop node is implemented it would also prove useful to implement a While loop which has similar functionality with the difference that the loop is executed until a boolean condition, which would actually be a mathematical expression, evaluates to false. This would give more power than a purely numeric loop test given by the For loop (though each of the "Loop_Start", "Increment" and "Loop_Finish" are mathematical expressions evaluated prior to the first execution of the for loop.). There are several more possible structural modifications of this ilk that could be implemented, hopefully the pipeline construction and execution algorithms are now sufficiently robust and extensible that no further modification of them should be required so implementation should be quite quick.

9.6 Selection Channels

One of the most interesting ideas from the ClayWorks modelling system discussed in section 4.3 is the idea of selection channels. The idea allows the user to select geometry not by selecting individual lines or triangles but instead by selecting a volume of space. The selection is made by the application of boolean operators (NEW, OR, AND, XOR) acting upon a set of convex hulls specified by the user (spheres or cuboids). These primitives representing the selection channel are then themselves modified by each node in the pipeline, for example being scaled or translated at the same time as the actual geometry. This allows a huge amount of flexibility

and makes the selection of geometry for the application of modifiers far less brittle and liable to error.

This method of selection fits exceedingly well with procedural graphics and would work very well in the system we have presented. For example it would be possible to implement selection channels by having a tree structure similar to the Buildup semantics consisting of convex hulls and boolean operators which could be represented easily in the scene tree especially if 3d primitives and modifiers were implemented. The selection channels could then be operated on via a specalised modifier nodes which should include operations to take two selection channels and apply a boolean operator such as AND or OR. The convex hulls would be graphical primitives standard to the language and could be set apart by reserving a block of tags for different selection channels.

This method of selection is very complimentary with the Tag System we have implemented which selects primitives according to a user specified tag. This tag can also be modified as the primitive travels through a pipeline. The selection channel system allows selection based on volumetric selection where as the tag system allows the user to semantically group primitives. The interplay between the two types of selection could be interesting, for example selecting all triangles tagged 2 in a certain convex hull could be used to select all pieces of smashed glass from a window in a certain area.

Thus we see that selection channels fit our procedural modelling formalism perfectly as they avoid the brittleness of primitive selections which would be lost as soon as the underlying model is modified by changing an earlier node in the pipeline. It would be necessary to refit all modifiers implemented so that they not only have a "Tag Test Expression" but also "Selection Channel" to allow selective application to primitives in the pipeline, but this would be straight forwards by the user editable system implemented (see section 6.3).

9.7 Computational Solid Geometry

Computational Solid Geometry is an established technique for creating interesting geometry via the use of a tree structure. The tree is comprised of primitive data types on the leaves of the tree and the remainder of the tree is comprised of boolean operators which compose the lower layers of the tree. This has identical execution semantics to the Buildup mechanisms presented earlier. The tree based nature of the representation makes it an excellent candidate for adding to the language, and would enable the addition of another procedural formalism. Care would need to be taken in creating an expressive enough data structure to hold the results of the application of multiple boolean operators which may give rise to highly non-uniform geometry.

9.8 Shape Grammar

A Shape Grammar was a concept first presented by G Stiny & J Gips [27] in 1971. It is a system similar to LSystems discussed in section 2.6. A Shape Grammar consists of a series of identification, transformation pairs, which first take a subset of all available graphical primitives in a scene and then transform or replace them by one or more new primitives. This simple concept is quite hard to implement in normal scripting languages, although it has meet with great success in the City Engine from Procedural Incorporated [31] [32] which allows the procedural generation of entire cities.

We believe this powerful technique could be integrated very well into our system. This could on a basic level be achieved as follows:

- The identification phase could be implemented by a test for the type of the primitive being tested. Then further tests could be made via the Tagging system and if implemented selection channels. Further a mathematical expression based upon the x, y and z ordinates of each of the points in the primitive could be evaluated at run time to test if the primitive meets the correct criteria. If necessary the expression evaluator could be provided with additional functions which act upon points in 3d, for example by find the angle between

two lines or finding the length of a line, these would be easy to add as the expression language is fully extensible.

- The transformation phase could be implemented by a subtree of modifiers and primitives. These could be provided with a mathematical context which would include the x,y and z ordinates of each of the points in the primitive. This would enable the subtree to create new primitives based upon the original identified primitive.

A shape grammar maps well to multithreaded execution (where as LSystems because of their linear evolution and turtle based rendering does not). This is because each identification and transform subtree could execute in different threads via a work splitting algorithm. Of course a subtree of sufficient size could also be split for via different threads via the standard execution algorithm.

In summary a Shape Grammar is an extremely useful way of writing procedural graphics and as it integrates naturally into the system we have developed it should be amongst the first new features to be implemented if possible.

9.9 Interactive Modelling

Probably the biggest improvement to useablity that could be made to the system would be the inclusion of some level of interactive modelling. The advantage of this is that it would avoid the user having to create primitives by entering coordinates, instead they could simply draw out a line or triangle in 3d. It may also be possible to add certain modifiers to the scene tree by use of various tools in 3d. One could imagine the user selecting a given pipeline from the tree and rendering it up to a given node in the pipeline. The user could then transform the displayed primitives, these transforms could then be inserted directly into the pipeline after the end of the selected portion as new modifier nodes.

This approach could probably be generalised to work on groups of pipelines providing they share a common beginning portion of pipeline and that all pipelines that branch from that end point onwards are exactly the ones selected by the user, a new modifier could be inserted at the end of this shared pipeline. If there is no such pipeline segment which is shared by exactly the user selected nodes then this approach would fail.

Further interactive display and modification of the scene tree and its output are possible, for example by displaying only nodes with a certain tag. Interactive modelling is clearly a highly productive way of making modifications to a scene tree however it would require very fast execution and rendering in order to be truly interactive, especially for large scene trees.

9.10 Optimisation

In order to facilitate interactive modelling it is probable that the execution of scenes would need to be optimised. This is especially true when running animation since at the moment each frame of the animation is built from scratch with no reuse of data from the previous scene. This shares some similarity with the way that graphics pipelines operate, however there would appear to be much scope for optimisation of the execution process.

Firstly if the pipelines are unchanged between executions for example when just the Frame parameter changes or the user edits a primitive. Then the pipeline data structure need not be thrown away and could instead be reused from the previous execution. Similarly if certain pipelines are known not to have changed between executions then their output could be retained or cached instead of being recomputed. Care would need to be taken with regard to changes in pipeline structure brought about by the changing of the mathematical context, especially with regards to For loops carrying out “Frame” number of iterations.

One highly interesting direction for optimisation would be looking at implementing a form of Active Semantic Caching [26] whereby not only is data cached but also a description of the query/algorithm (in our case a pipeline) which was used to generate it is cached. This allows

exact matching of requests to stored data. The "Active" part of the cache is that if the cached data is very similar to a given request except perhaps for an extra node in the pipeline then the cache controller can apply only that last stage of the pipeline to the cached data and return the this as the result.

Active Semantic Caching should map very well to our language as it is pipelined and made of the composition of a number of simple modifier nodes acting upon very regular data. We believe that this form of optimisation would provide extremely good results if well implemented speeding up execution and facilitating interactive modelling.

9.11 Aggressive Threading

One of the ways to improve performance still further would be investigate the amount of parallelism which is as yet unexploited. Most of the execution algorithm is already threaded except for the pipeline construction algorithm which may contain some parallelism, although the nature of pipeline construction may preclude this.

Currently the sections pipeline shared between multiple pipelines are currently executed by a single thread to avoid having workloads too small to be worth threading. In some scene trees this is probably the wrong decision as the amount of work for one thread becomes large and should be split. An investigation into automatically deciding how much work there is in a pipeline section and then whether or not to split it could prove quite fruitful.

Similarly all modifiers themselves are single threaded as threading them seems unworthwhile, though when dealing with tens of thousands of primitives the workload should certainly be split and threaded. Algorithms to support this should be developed.

One of the ways in which this kind of aggressive threading could be supported would be via the forthcoming .Net/C# 4.0 release [30] which promises builtin support for far finer-grained parallelism.

9.12 NVidia Cuda and Compilation to C

Given the parallelism which we have already exploited in the project and the scope for introducing yet more, it seems worthwhile to look at more parallel devices than the current or even next generation of desktop CPU's.

NVidia's recent Tesla architecture[28] is a graphics compute engine incorporating massive parallelism to an unheard of degree. The latest generation of graphics cards now support up to $30,000$ threads. Each thread is able to run a different code path, with the best performance being achieved when groups of threads run identical code. The Tesla platform is able to use the huge number of threads to hide memory latency and produce stunning amounts of compute power, eclipsing that available for the CPU.

With this in mind it is interesting to consider whether or not our language could be ported to run upon this piece of graphics hardware. Currently the major problem is that the graphics card will only run one *Kernel* or code block across the whole device. Although speculation it seems quite likely that this constraint will be removed in the near future, which would give rise to some interesting possibilities:

When a scene tree is executed it would be possible to store the set of full pipelines created for a particular frame. The data stored would be similar to that given in the debug system 6.4. These pipelines could then be translated into C code and then possibly transformed into Cuda code (an extension of C which runs on the Tesla architecture). Care would have course have to be taken in translating LSystems and the mathematical expression language which is so deeply embedded within the project, though both seem feasible propositions. This process could either be done as a one off event before a graphics application was compiled or could possibly be done at runtime on the CPU.

With the code to generated to run each pipeline node and the pipeline metadata detailing the order of the nodes within the pipelines and their respective input it should be possible to

work out a strategy to run the pipelines on the graphics card.

Every pipeline could be broken down into its constituent nodes. For each primitive being processed by each node a new thread could could be spawned to apply that node. With so many different nodes we see why the ability to run multiple kernels is required. With so many primitives and pipelines it would be quite easily possible to make use of the huge number of threads available, though to achieve the best throughput it may be worth making each thread process more than one primitive.

One possible draw back is that when new primitives are generated by a node in the pipeline additional threads would have to be started. This makes the execution of the algorithm somewhat serial. The situation could possibly be improved by breaking down each pipeline into a series of sections between the synchronisation points caused by batched based nodes. One could then execute that stretch of pipeline from start to finish for the input nodes before spawning a new pipeline for each set of new nodes primitives executing from the node at which they were generated to the end of the pipeline. Alternatively one could imagine extending the metadata to record at which point new primitives are generated and the primitives themselves, though this would negate some of the benefits of procedural generation by adding to the amount of data requiring storage.

One performance sensitive issue with the Tesla architecture is the use of branch instructions which slow down the execution dramatically. Most nodes within the pipelines do not contain branches, especially as all composite modifiers (If nodes and For loops) would already have been dealt with by precompiling the code and meta data. This would leave only the Filter and Splitter nodes which we recall are batch based and thus must be executed by themselves and so would comprise only a small fraction of the total amount of computation completed.

Overall this scheme would mean that the user could take full advantage of the procedural nature of the language by only having to store the code, pipeline metadata and the initial pipeline input data of the graphical scene. This would be especially useful as the final primitives would reside on graphics card memory and not have to be transfered there through the bottlenecked of the PCI-bus or from disk.

CHAPTER 10

Conclusions

> "I expect that by the time of the release of the next generation of consoles, around 2012 when Microsoft comes out with the successor of the Xbox 360 and Sony comes out with the successor of the PlayStation 3, games will be running 100% on based software pipelines"
>
> **Tim Sweeney, creator of the Unreal game engine and CEO of Epic Games**[4]

In conclusion we believe that the new generation of graphics hardware will increasingly require new approaches to producing graphical content, this will undoubtedly require artists to rely on procedurally generated content. However the fragmentation of the various procedural frameworks is a huge handycap to their adoption in industry. Similarly the programming skills required to use these frameworks is another major problem which must be addressed.

In this project we have presented a new approach to procedural graphics by presenting a system based on the composition of simple modifier nodes to form a series of software pipelines embedded into a tree structure. The language we developed contains a number of geometric primitives such as lines and triangles, and a larger array of modifier nodes which act upon them (rotate, translate, mirror and colour changer nodes are implemented amongst others).

The language we present successfully unifies a number of procedural formalisms with scope to integrate many more as described in the Further Work chapter (9). This allows the user to take advantage of many techniques which are well suited for particular tasks in the same familiar environment. It also allows the user to exploit these formalisms in concert which allows huge flexibility and power not available in other procedural modellers.

The first two formalisms are implemented by exploiting two different tree execution semantics, the default pipelining and the more traditional buildup semantics which work similarly to Computational Solid Geometry with data flowing up toward the root node. The same multithreaded execution algorithm is exploited for both semantics. We also support at every point in the project an extensible mathematical expression evaluator which allows the user to input most data as a mathematical expression providing huge functionality. Since numeric values can be inputted instead of expressions we do this without presenting the user with a complex interface which is a unique feature.

We have also implemented a number of constructs from imperative programming languages such as If tests and For loops as well as allowing variables to transit the various execution pipelines. The addition of flow modification nodes which allows pipelines to intersect and split apart is also highly usable and gives the language similar power to a free-form directed graph.

The inclusion of an LSystem execution engine shows that many different procedural formalisms can be integrated into the language we have developed, of course there are a plethora of ways each of the formalisms already integrated could be extended and there are many other ideas and formalisms which could be integrated, most notably a Shape Grammar [27].

Of course any new language is only as good as its user interface, this is something we spent a good deal of time on. Most prominently we implement a TypeSafe drag and drop system for editing the tree via a user editable toolbox. This system ensures that only legal scene trees can be constructed thus preventing much user frustration by not having a compilation / constraints checking stage of execution with the vague error messages it could entail. We also implement more standard features of an IDE such as save/load and do/undo/redo functionality. Finally we provide a standardised node editing system which auto-generates a standard style form for each node in the tree along with custom validation routines which full and useful help messages. More details of this interface can be found in chapter 6.

Unlike much of the related work in this field we support animation as an integral part of our system producing an array of content in real time through two different rendering engines. Finally the execution algorithm for the language and its various semantics is multi-threaded. This allows the user to fully exploit the current and future generations of CPU's, this is something not addressed in related work (see chapter 4). The multi-threaded nature of the language and its execution gives rise to an extremely interesting possibility - that the language could be compiled to C code and then to NVidia Cuda code to run on the latest generation of General Purpose GPU's (GPGPU's) which support tens of thousands of threads. This is a reasonable objective and could be achieved by exploiting far finer-grained parallelism than currently implemented, for example by having one thread for each primitive transiting a pipeline. The advantages of producing such a mechanism are manifold and would include exploiting the 100 fold increase in compute power that the GPGPU offers. Also the final graphical scene would be produced directly in graphics memory ready for rendering - avoiding clogging up the CPU and system buses with data.

In conclusion this has been a very exciting project to undertake and has produced some interesting graphical animations which would have been challenging to producing other programs. Learning C# has been a fascinating experience and has allowed the production of some very interesting systems which greatly increased the speed of development. The system is much easier to use than much of the related work in this field and is the first to unify a number of procedural formalisms. There are multitude of directions for further work either by extending the currently implemented formalisms with the latest work in their domains or by integrating a number of new procedural frameworks which would harmonise well with the current execution engine and already integrated formalisms. It would be fascinating to see how this framework could be developed in the future.

The end.

Bibliography

[1] http://developer.amd.com/cpu/SSE5/Pages/default.aspx as at 24/08/2008

[2] http://www.infralution.com/virtualtree.html as at 15/10/08

[3] "A System for the Non-Linear Modelling of Deformable Procedural Shapes" Tim Lewis and Mark W. Jones 2004 ,Twelfth International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (Winter School on Computer Graphics) pg253–260

[4] TG Daily interview http://www.tgdaily.com/content/view/36410/118/ as at 24/08/2008

[5] "Sketch-Based Procedural Surface Modeling and Compositing Using Surface Trees" Ryan Schmidt and Karan Singh 2008 Computer Graphics Forum, vol 27 pg321-330

[6] "HyperFun project: a framework for collaborative multidimensional F-rep modeling" Adzhiev et al.

[7] "Extending the CSG Tree" by Brian Wyvill, Andrew Guy & Eric Galin. Computer Graphics Forum, Vol. 18, No. 2. (1999), pp. 149-158.

[8] "Persistence of Vision Raytracer" www.povray.org as at 24/10/2008

[9] http://www.csb.yale.edu/userguides/graphics/povray/demo.pov.html as at 7/11/2008

[10] Autodesk Maya - http://www.autodesk.com/maya

[11] Autodesk 3ds Max - http://www.autodesk.com/3dsmax

[12] "How GPUs Can Improve the Quality of Magnetic Resonance Imaging" by Sam Stone et al in Proceedings of the 5th International Conference on Computing Frontiers May 5-7 2008 http://doi.acm.org/10.1145/1366230.1366276

[13] "GPU Acceleration of Numerical Weather Prediction" by John Michalakes & Manish Vachharajani Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium.

[14] Nvidia Cuda portal - http://www.nvidia.com/cuda

[15] Renderman Portal & Tutorials http://www.fundza.com/ as at 7/10/2008

[16] Przemyslaw Prusinkiewicz: Graphical applications of L-systems. Proceedings of Graphics Interface '86, pp. 247-253.

[17] Lindenmayer, A. [1968]: Mathematical models for cellular interaction in development, Parts I and ll. Journal of Theoretical Biology 18, pp. 280-315

[18] LinSys3d from http://www.esuli.it/

[19] Evaluant - a software development company, developers of NCalc http://www.evaluant.com

[20] NCalc - open source Mathematical Expression Evaluator available from http://ncalc.codeplex.com/

[21] An overview of NCalc, online article by developer Sebastien Ros http://www.codeproject.com/KB/recipes/sota_expression_evaluator.aspx

[22] Microsoft DirectX developer center: http://msdn.microsoft.com/en-us/directx/default.aspx

[23] EasyBMPtoAVI an open source command line movie making tool from http://easybmptoavi.sourceforge.net/

[24] Doc-O-Matic used to create full documentation available in project repository. www.doc-o-matic.com

[25] Article on Non-Uniform-Rational-Basis-Splines by Rhino3d a company providing software to edit NURBS http://www.rhino3d.com/nurbs.htm as at 10/06/09

[26] "Active semantic caching to optimize multidimensional data analysis in parallel and distributed environments" Henrique Andradea, Tahsin Kurcc, Alan Sussmanb and Joel Saltz doi:10.1016/j.physletb.2003.10.071

[27] Stiny G, Gips J, 1972, "Shape Grammars and the Generative Specification of Painting and Sculpture" The Best Computer Papers of 1971: Auerbach, Philadelphia 125-135 online at http://www.shapegrammar.org/ifip/ifip1.html as at 10/06/09

[28] "NVIDIA Tesla: A Unified Graphics and Computing Architecture" Lindholm, E.; Nickolls, J.; Oberman, S.; Montrym, J. Micro, IEEE Volume 28, Issue 2, March-April 2008 Page(s):39 - 55 Digital Object Identifier 10.1109/MM.2008.31

[29] First year laboratory script "LSystems" at Imperial College London http://www.doc.ic.ac.uk/lab/cs1/GiveToStudents/Haskell_LSystems.pdf

[30] Microsoft .Net framework 4.0 beta release information http://msdn.microsoft.com/en-gb/netframework/dd441784.aspx as at 12/06/09

[31] City Engine from Procedural Incorporated http://www.procedural.com/

[32] "Procedural Modeling of Cities" Yoav I H Parish & Pascal Mller siggraph 2001 available online at http://www.vision.ee.ethz.ch/~pmueller/documents/procedural_modeling_of_cities__siggraph2001.pdf

[33] "Shade trees" Robert L. Cook (Computer Division, Lucasfilm Ltd.) ACM SIGGRAPH Computer Graphics Volume 18 , Issue 3 (July 1984).