Name: George Christov Rassovsky
Student ID: i7258984
Course: Msc CAVE (NCCA)
Module: Animation Software Development
Assignment: L-Systems
Paper: Report on “L-System Visualiser”

January 31, 2014

Abstract

The aim of the report is to give a background to the topic and a brief description of the different types of L-Systems, followed by the main design of the project and a plan on how the problem would be resolved as well as the implementation itself. The research includes all well known L-System types. The project proposed the production of a flexible piece of Software which could be used for the generic creation and interpretation of L-Systems. The outcome of the software development resulted in the 'L-System Visualiser'. The report gives a detailed evaluation of the finished product, drawing useful conclusions from the learning process.

Introduction

Introduced by the Hungarian Biologist Aristid Lindenmayer in 1968, Lindenmayer-Systems, also known as L-Systems, as they will be referred to in this document, are a type of formal recursive grammar. In it's most basic form a L-System is simply a string of symbols which is derived by an initial axiom expanded through a set of rules for a given number of iterations. A simple example can be seen in *Figure 1*.

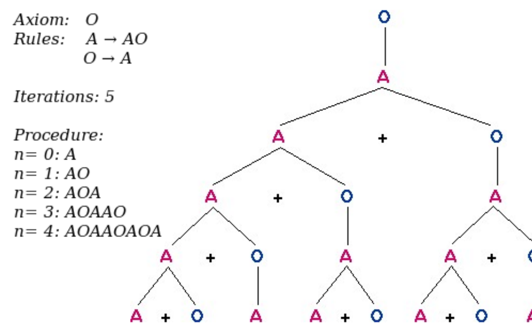

Figure 1: This figure depicts the first five iterations of a simple Fibonacci sequence (D0)L-System. In the tree to the right we can see how the symbols are being replaced every iteration, following the rules of the L-System.

The work on L-Systems is a derivative of fields such as developmental biology and fractal geometry. Morphogenesis, the development of patterns and structural features in organisms (The Tickle Trunk 2011), which deals with the shapes of tissues, organs and entire organisms (Science Daily 2013), is one of the main inspirations behind the invention of the L-System. Alan Turing and his last paper The Chemical Basis of Morphogenesis (Turing 1952), could also be mentioned amongst the contributors to the field of work leading up to the formal language.

Related Work

L-Systems were initially used, by their inventor, for modelling the growth process of plant development and for describing plant cells behaviour. Aristid Lindenmayer's publication, The algorithmic beauty of plants (Lindenmayer 1990) is one of the most detailed sources on L-Systems, applicable for this project. Since then, they have been utilised in the application of diverse fractal based systems. From visualizing the morphology of a variety of organisms to the creation of procedural computer generated imagery and geometry. Most modern AAA Games use L-Systems in one way or another somewhere in the flow of their pipeline. 'Elder Scrolls IV: Oblivion' *Figure 2*, 'Halo 4', 'Game of Thrones' and 'Fable: The Journey' are only some titles which stand under the common denominator of using 'SpeedTree' (Mobygames 2013). Having mentioned this, 'SpeedTree' is only one of the many 3D vegetation generators with wind physics algorithms which heavily uses L-Systems in it's core. Games are not the only place where L- Systems are being put into use. Film VFX and Animations are other fields which use software, integrated with L-System algorithms, in various ways.

Figure 2: A forest scene from the PC Game, 'Elder Scrolls IV: Oblivion' (2006).

L-Systems are incorporated into most modern 3d Software packages, such as: Houdini, Maya, 3DsMax, Blender and many others (only as plug-ins in some cases). As an example, in *Figure 3*, we can see a screenshot from the Houdini L-System interface used for the creation of tree-like structures.

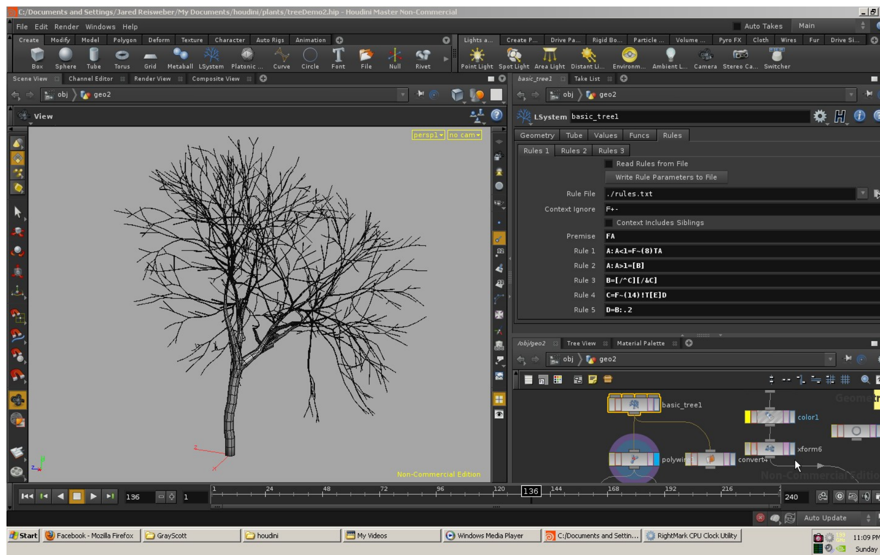

Figure 3: A screenshot of a L-System in Houdini.

Technical

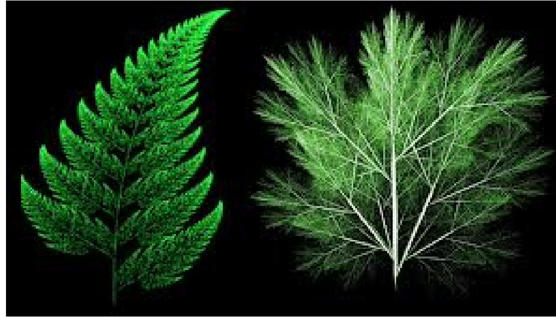

Figure 4: Fractal plants created using L-Systems.

Parameters of an L-System

A typical L-System has a simple set of parameters. What is done with those parameters and how they get interpreted later, is what produces the stunning results, as those seen in *Figure 2* and *Figure 4*. In basic terms, each L-System must have the following: an axiom, a set of rules and a number of iterations.

Axiom – In classic terms, the axiom is the initial string with which the L-System commences. It is not derived but rather is a given constant. It never changes and it is used only for the first iteration of the system. In the case of a recursive L-System, each string derivation would become the axiom for the next call.

Rules – It is the set of rules that expand the axiom, and every string in every following iteration. They must contain the definition of each variable symbol in the system, excluding the constant symbols, which are just copied on to the next string. There is a wide range of rule types. Some of them would be mentioned in this paper. Iterations – This is the number of times the system is to perform the replacing algorithm on it's current state, based upon the rules of the system. The iterations number must be a positive integer, e.g. A Natural number as the system cannot make a recursive call a negative or zero amount of times.

Types of L-Systems

There are two main branches of L-Systems, namely 'Deterministic' opposed to 'Stochastic' and 'Context-free' opposed to 'Context-sensitive' ones (Procedural Composition Tutorial).

Deterministic vs Stochastic – As the name suggests, a Deterministic L-System would have only one rule for each symbol in the string that is being

processed, thus being deterministic in it's choice of definition for that symbol. Stochastic or Non-deterministic L-Systems, on the other hand, could have more than one rule (or definition, to be more precise) for each symbol. Then one definition would be chosen out of the list of all of the definitions for this symbol. This could be done in a pure random fashion, where all the definitions have the same probability of being picked. Another more complex but flexible way could be specified, were each separate definition has a value associated with it, which would then act as a weight or percentage in it's probability rate during the random selection when expanding the symbol (Lindenmayer 1990).

Context-free vs Context-sensitive – Context-free L-Systems only focus on the current symbol which is being processed, and on the rule(s) it might have. Context-sensitive L-Systems, however, keep track of previous and following symbols. Specific patterns could be similarly set as separate rules themselves. If the pattern is spotted around the currently processed symbol, the "context-sensitive" rule is expanded, having priority over any context-free rule related with the current symbol (Lindenmayer 1990).

These four variations of L-Systems are the building blocks of the most popular types of L-Systems. They are the following (Where the "D" denotes "deterministic", and the number denotes whether it is a context-free "0", a one-sided context-sensitive "1", or a two-sided context-sensitive "2" L-System.):
D0L-System. Deterministic, Context-free L-System. (The simplest class of L-systems, as shown in *Figure 1.*)

- D1L-System. Deterministic, one-sided context-sensitive L-system.
- D2L-System. Deterministic, two-sided context-sensitive L-system.
- 0L-System. Stochastic, context-free L-system.
- 1L-System. Stochastic, one-sided context-sensitive L-system.
- 2L-System. Stochastic, two-sided context-sensitive L-system.

Some other types which are not mentioned above could be parametric or timed L-Systems, where the grammar would vary based upon time or some customised behaviour. If we consider every variation as a separate type then there could be an infinite amount of L-System types (Procedural Composition Tutorial).

The Program incorporates all these L-Systems in a single package. This makes the Program much more generic and adds a crucial amount of realism when it comes to the visualisation of natural phenomenon or nature itself. As there are no two plants which are exact copies of each other, neither would there be a forest with identical trees. Mimicking those attributes is the key to realistic representation and simulation of the real world.

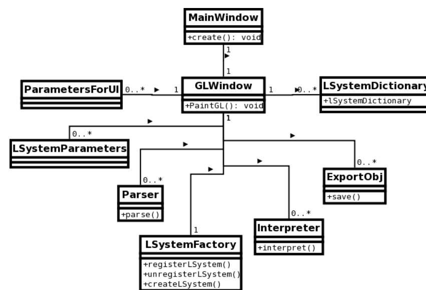

Figure 5: A basic class diagram, describing the layout of the program, with only the most relevant members presented.

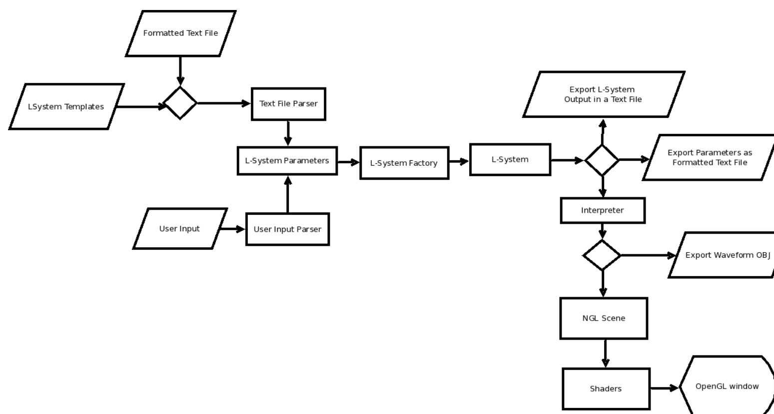

Figure 6: The flow of the program with regards to the final implementation.

Class Diagrams

The Program Flow

The program is logically split into two main sections. Part one focuses on the creation of L-Systems, while the target of the second part is the interpretation of L-Systems, whether that is output to a file or rendering to a OpenGL window in real-time.

This project, only considers the six most renowned types of L-Systems, which were derived from the two main categories spoken of earlier. The program is created to support flexible usage and is built with a generic interface. Namely an “Extensible Factory” pattern is used for the creation of the different L-System types and Interpreters. This method leaves room for the assignment of new types of L-Systems and Interpreters by the user or simply extending the programs interface. The L-System parameters (the axiom and the rules) are externally derived in two ways, either as input from a text file with a specific format,

or via user input within the GUI restrictions. Either way the program would parse the input and store it as a generic data structure of L-System parameters. There could be an unlimited amount of different structures of L-System parameters in the program as they would be stored as objects. Each one of these sets of parameters could then be used for the generation of L-Systems from the six previously mentioned types, at a user specified iteration level. This would then allow the user to compare the result of every type of L-System begotten from the same set of parameters.

The L-System could then be passed on to the interpreter of choice. Afterwards the interpreter reads the generated string, symbol by symbol and a set of actions is performed, based upon the interpreter's representation of each symbol. The interpreter goes through the whole string doing the calculations for each symbol according to it's definition and generates a set of points and indices in 3D space. Those then would get passed on to the OpenGL visualiser class of the NGL Graphics Library where after being processed they get further passed onto the Shaders, which on their behalf do the visualisation computation each frame, drawing the derived geometry in the OpenGL window.

The user can then recreate the visualisation of the L-System at any of the computed levels, as all the L- Systems' build() functions have the option of returning a vector of strings, where each string is a separate iteration level in the system.

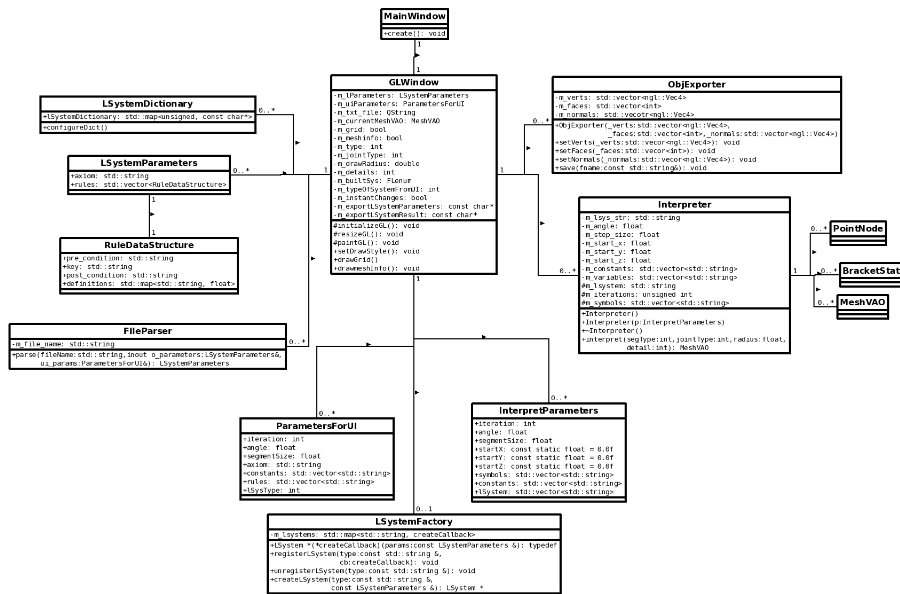

Figure 7: The main UML detailed diagram of the project.

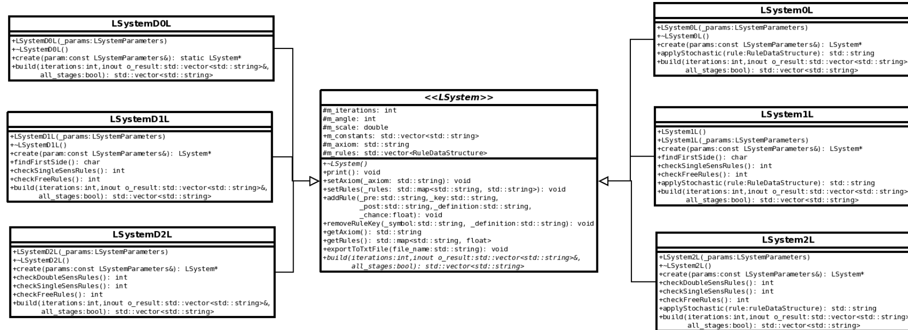

Figure 8: A UML diagram depicting the different types of LSystems inheriting from the abstract LSystem class.

Design Justification

A brief description of the reasoning behind certain decisions in the initially proposed design of the project, as seen in *Figures 7 and 8*.

Overall Design

The Program implements the Extensible Factory pattern as this allows for a more generic, cleaner and easier-to-use interface. The style of the extensible factory is based on the example in the book API design for C++ by Martin Reddy (Reddy 2011). This allows an eventual extension of the functionality and the addition of more types of L-Systems, and the continuation of the development. This prospect is the main reason for using the Extensible Factory over the standard Factory method. The piece of software is very suitable for this design pattern as there is a L-System abstract base class which is the parent to every type of L-System. This type of inheritance, on the other hand was deduced from the fact that most types of L-Systems share common functions between each other, differing only in the build function and in certain other helper-functions.

The Factory class is created using a singleton, thus allowing for only one instance of itself ever existing. As it can be seen in the main detailed diagram in *Figure 7*, the L-System Factory can combine any type of L-System with any already created structure of parameters, by registering them together. This allows for a very flexible behaviour of the system.

The simple UML diagram, *Figure 5*, describes the basic class arrangement. As seen in *Figure 7*, the LSystemParameters data structure is required for the creation of a L-System. Each instance of an L- System could only be registered with one set of parameters, but one set of parameters could be registered to a number of different instances of types of L-Systems. The LSystemParameters data structure itself is derived either through parsing input from a formatted file or through the user input extracted from the UI. If the user fails to provide

data in either of the specified ways, the parameters proceed with their default constructor. This is done instead of having a default constructor for each L-System class as the L-System factory design, that is used, does not permit an L-System to be registered without the association of a parameter data structure to it. This however is restricted with the help of the user interface and thus the program cannot proceed until the user specifies a valid L-System parameter file or a template from those provided.

Parameters and rules

The parameters and rules themselves are defined as structs for they are complex structures of diverse data, which has to be more general for the sake of flexibility between the different types of L-Systems. Rules also have the capability of storing pre and post-conditions, if any, as well as the actual rule key, followed by a map of the definitions and weights for that specific key. The std::map is chosen over any other data structure, as in a stochastic L-Systems, there could be more than one definition for a single symbol. That is where the weight value comes in hand as it allows the user to specify what is the probability of each definition.

A weighted system is used, instead of actual percentage rate, for the ease of the user. All the weights are being calculated upon parsing and are then stored as a weighted sum, which adds up to a total of one. This ensures more stability, as they are recalculated on the fly only in the case of adding new rules after the file parsing.

The reason the definitions are stored as the key of the map, the weights being the second value respectively and not vice versa is because there cannot be two same definitions for a single symbol, but there could be more than one definition with the same weight, for a single symbol. If no weight is presented for a certain definition, the program assumes the value of 1.0, which in the case of no other definitions would be translated to 100%. A weight of 0.0, on the other hand would result in 0% influence, thus making the definition non-active in the context of a stochastic system.

Parsing

The parser parses data from a customly formatted text file. It has the task of populating a LSystemParameters structure. For optimisation it gets its parameters passed in by reference.

Formatted Text Files

The formatted text file mentioned above is as simple as possible and yet incorporates all the features that any conventional type of L-System could have. This allows for quite a lot of flexibility and human-error in the formatting of the text file while still retrieving the correct information. The parser allows for comments which help aid the user to what the desired format looks like. The greatest feature of the formatted text file is that it is absolutely generic, so the user does not need to specify the type of L-System he would build with

it. The parser would parse all the data accordingly and then the instance of the L-System would use only the information that is applicable for it, according to it's type. Once an L-System is created it allows for the export of a text file, which would export the L-System parameters described in the same type of formatted text file which could then be read in by the program straight away. The exported file will include any changes done to the L-System, so if the user has altered the axiom or has added new rules, via the provided functions, those changes would be present in the export. This functionality is added in case the user wants to recreate the same, or similar L-System in the future.

L-System export

The provision is made for the functionality of exporting already built L-System strings. The exported output could then be used in other software interpreters. This functionality provides another middle ground between the produced piece of software and any other implementation. This is more of a pipline precausion, that two different L-System parsers or visualisers can speak to each other dispite having different parsers, on the basis of a resultant L-System string.

L-System builds and Interpreters

Once a certain L-System has been built, the output could be used in the provided interpreter. There is no limit to the number of interpreters, as they could have very different tasks, from creating points in 3D space to exporting a height map. There are an infinite variation of interpreters that could be written.

Meshing

The project was extended with the ability of producing an actual mesh. It produces a very crude mesh but never-the-less it could be exported and integrated into a 3D software package, as seen in *Figure X*.

Figure 9: A simple tree-like structure created in L-System Visualiser, exported as a OBJ file and imported into a simple scene in Maya.

The nature of L-Systems is such that it not possible to have a watertight

mesh in every scenario. Some L-System patterns are overlapping, some plant-like structures have their 'branches' crossing, etc. All these unknowns restrict the usage of L-Systems in many areas such as physics engines, and makes them in many cases not 100% visually correct. This undesired results however, could be limited using appropriate parameters. The problem of inconsistent meshing remains when joining segments one with another and especially at the joints where more segments connect to one.

The Joints – This topic does not give room for great discussion as there is no easy solution apart from implementing implicit surfaces. That is why this project has spheres, with the complexity and scale of the segment, for joints. Sphere joints do not solve the problem with mesh discontinuity, it is mainly implemented for aesthetic reasons. A much better solution would consist in the procedural creation of correct joints matching all attaching segments, without holes and discontinuities. This is the concept of the circular joints which were conceived in this project, but which run out of the scope of this particular project. They definitely remain as a case of further research and potentially implementation.

Outcome

The Code

The "L-System Visualiser" is a small software program written in C++ using NGL graphics library for the OpenGL content and Qt for the GUI components. It follows the NCCA coding convensions and uses Doxygen commenting style.

The final implementation takes advantage of the "Extensible Factory Pattern" for creation and registration of L-Systems according to their types as derived from the parameters passed on to the program. The same design pattern was initially used for Interpreters of the L-System. But it turned out to be rather cumbersome and was removed, following the KISS principle, hence the single Interpreter class, which takes care of all possible interpretations, since there could be only one at a time anyway.

The input stream parser, which was initially proposed, also did not make it to the final product as a result of it being redundant due to the GUI developed with Qt.

For the stochastic systems, boost random engine called "mersenne twister" was implemented at first but was later removed for the same reason. It was replaced with the standard library's random function. Even though theoretically the boost random is much better in many ways, it required much more additional code and was much more computationally expensive than the std random function, which was satisfactory for the purposes of the system.

The Program

The program itself is constructed as a standalone application, relying on the a current version of NGL. The main focus of the project was implementing all foundational types of L-Systems. And then graphically interpreting them in 3D space using OpenGL primitives such as points lines and triangels. The latter is represented under the form of cylinders which are created on the fly and passed onto the shaders, as VAO's, for drawing. Using this method, a mesh is created which can be seen in the OpenGL viewport. This way of rendering on the GPU allows for high-polygonal meshes to be displayed on screen that would otherwise be impossible on the CPU. In *Figure X* an example of such a mesh is displayed. This is a 3D Hilbert Curve at 6 iterations with a cylinder detail of 12, which is to say: made out of 12 slices (each consisting of 2 triangles). This gives a grand total of just over 76 million triangles rendered on screen in real-time and still keeping the viewport reasonably interactive. This is a impressive figure considering that the code has not been optimised in terms of rendering.

A created mesh can then be exported as an OBJ file to a user specified location. By using the universal .obj file format within minutes those meshes could be loaded into a major 3D software package such as Maya and rendered with appropriate settings, as seen in *Figure X*.

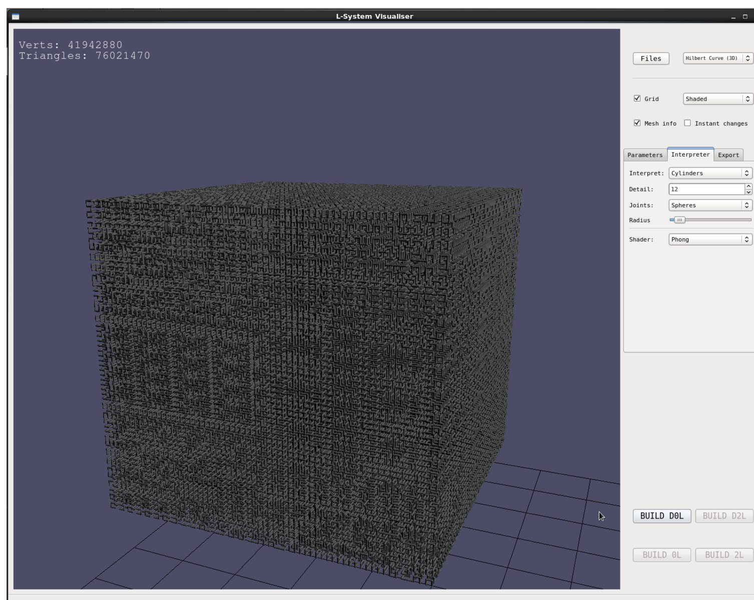

Figure 10: A 3D Hilbert Curve at 6 iterations with a cylinder detail of 12, resulting in a mesh of over 76 million triangles, rendered in real-time.

The program allows for two other types of exports, which are that of the parameters (e.g angle, iterations, constants, axiom, rules, etc.) and that of the resulting L-system string. Both of them are in the form of a text file which is formatted in a manner compatible with the file input of the L-System Visualiser. Those exported files could be used directly into the same system or used as reference on other systems which do not recognize the exact formatting of the files.

The GUI

The user iterface is created entirely with qtcreator under QApplication. For the GUI some of the main rules of intuitive UI were followed. The UI design was led by some of Nielsen's principles, such as, "Recognition rather than recall" - Jacob Nielsen (1995). The aim when creating the UI, in the limited timespan, was to create an intuitive, easy-to-use extension of the program which would allow for the user to learn how to use the program in a much quicker way. The UI is used for some other good reasons too, such as, restricting the users input abilities to reasonable values and thus preventing some potential pitfalls of the exponentially growing complexity of L-Systems. At the same time, the goal was that all the major features of the program would be extracted and displayed for fast user interaction and usage. As a result tabs were used to separate the main components of a L-system and at the same time larger, always-visible buttons were used for the main features connected with the creation of a new L-System,

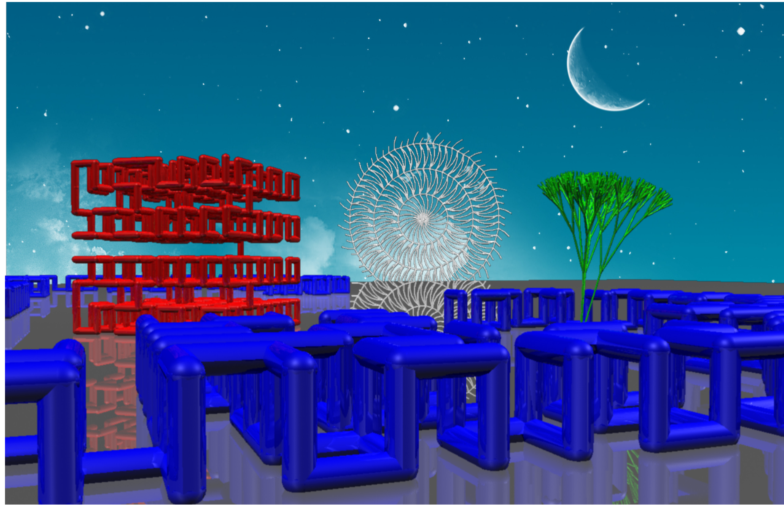

Figure 11: A few meshes created with and exported from L-System Visualiser, which were later imported into Maya and rendered in a scene.

such as loading a file/template and building a recognised system. This type of layout provides a steep learning curve when using the software for the first time. However some basic knowledge of L-Systems is recommended in order for the user to be capable of proper interaction with the system, due to the nature of the topic itself.

Encountered Issues

Many issues were encountered along the way to final release. The final version of the program though, does not vastly differ from the originally proposed design and implementation. There were no major changes to the product due to unmanagable tasks, however time was the main issue. Even though the product would be considered finalised, there are many other features which could be introduced, potentially as a further work.

Milestones – Some of the major milestones of the project were respectivelly: Parsing correcly from a file; Building the six major types of L-Systems, spoken of earlier, from the parsed parameters; Interpreting the L-Systems within a OpenGL context; Implementing 2D, Bracketed-systems, 3D, 3D bracketed-systems; Creating and interpreting with geometry; Implementing sphere joints and working on the concept of 'watertight' joints; Creating the UI and exposing all necessary functionality.

Known issues – The current version of the software is far from perfect. There are many small problems which were left unresolved. Mainly due to the lack of time.

- There are some inconsistencies between the UI and keyboard shortcuts for various tasks as for example switching between wireframe and solid mode. The fix to this would be a matter of connecting the two ways of manipulating the draw mode with a common attribute.
- One other problem with the UI is the double key press which is registered once the displayed mesh becomes heavy and a lag occurs. This introduces an unwanted additional key press which would increment the value twice instead on once.
- Another known issue is a small triangle puncture in the north pole of the sphere model. It would be fixed once the correct index is passed, which wold close this last triangle and thus the sphere.
- There are three registered pairs of shaders used in the current version of the program. Two of them are used for the visualisation of the L-System, namely, the "ColourShader" and the "Phong". The colour shader is used to pass colour per vertex using the VAO. As this is not done correcly for the cylinder representation, when using the colour shader with the cylinders, colours would appear wrong. The colour shader works perfectly for now only with the line and point interpretation. This is not a hard problem to fix, but there were other more vital things that had to be done in the allocated time span of the project.
- The so called "watertight" joints, which currently are under a beta tag, also would not apear correct as they are left in the program only for reference and proof of concept.

- Other more key issues which were spotted in the code would be const correctness and the program's performace were some memory leaks can be observed after a vast and heavy exploitation.

Conclusion

The project fulfilled all major goals it set off to do. Some of the main ones were namely, delivering a L-System generator comprising all basic types of L-Systems with ability for further extensions. The interpretation into 2d and 3d visualisation was also a fundamental milestone. The ability for exporting meshes in the .obj file format was also one of the key features promised in the initial report. All those delivarables were implemented and delivered with the final product.

One feature which was mentioned but was never executed was the creation of a heightmap by the system. This was not looked into in great depth as it inhabits a different branch of L-System usage which was not focused upon in the limited timespan. This still remains as a further extension which could be worked upon in the future.

On the other hand the project delivered some additional features which were not necessarily planned in the initial report. One of them is the cylindrical representation of the L-System, which allows for the creation of a solid mesh out of the interpreted positions. This was further expanded by the implementation of sphereical joints and the concept of circular (watertight) joints, which were mentioned in the above sections.

Future Work

There is a vast oportunity for extensions and further work on the software. Some suggestions are mentioned below.

Universalising the parser – A standardised XML format could be used for parsing which would make the file input more robust.

Shader writing – There is always room for shader writting. Even though that wouldn't necessarily be of the greatest importance for this specific project, at that stage, it is certainly a place where time could be invested for the creation of a better looking product.

Extending the L-System Factory – This would be one of the key further work topics. There is a whole universe of possibilities opening up with the implementation of Parametric L-Systems. As well as any of the other more advanced systems such as DT0L, T0L, etc.

Interpretation modifiations – The interpreter could be extended along with the extension of the L-System Factory. In the case of parameteric L-Systems being implemented a parameter could mean the drawing of different geometrical primitives, such as patches, spheres, curves, etc. This would allow for the creation of complex natural shapes, such as those seen in *Figures X, X and X.*

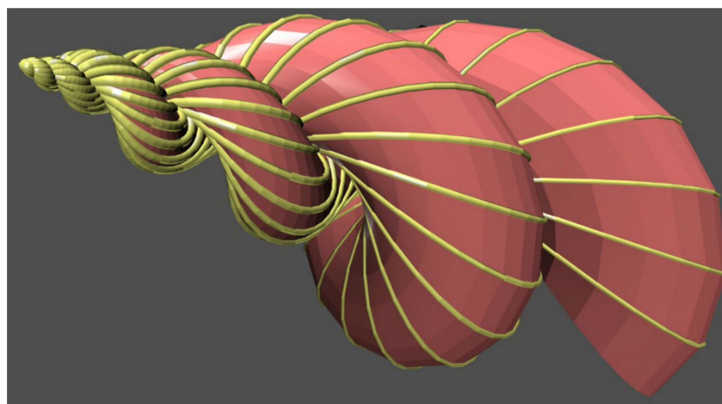

Figure 12: A snail-like mesh created with a simple parametric L-System and interpreted with some additional geometeric primitives.

Watertight meshes – One of the main issues with with most L-Systems out there is that they produce meshes which cannot really be used in many places, due to the fact that many of them are not water tight. One of the key

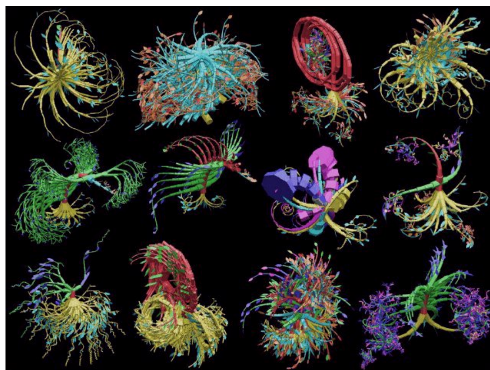

Figure 13: Parametric L-System were used for the creation of those plant-like structures.

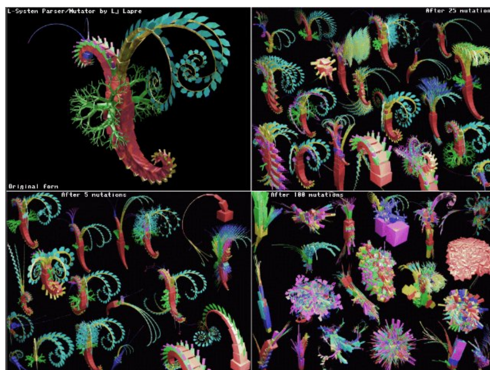

Figure 14: An array of different mutations.

problems is the joints in between segments. Implicit surfaces could be used to create watertight joints. Another method was started for this project which takes account of the parent and children branches and constructs a joint after creating all the segments. This method unfortunately could not be completed before the deadline and thus is left under a BETA tag in the final release.

Working on the UI – There is always more to be desired from the UI, both in terms of extentions and in simplicity. This category is open to many changes and improvements.

Optimisation – A key step would be optimisation, both on the parsing and interpretation of the L-Systems but most importantly on the actual visualisation in OpenGL. There are many types of algorithms that could be imlpemented and even many fixes to the way the code workes which would increase the performance of the program.

References

[1] The Tickle Trunk, 2011. Lindenmayer Systems [online]. *The Tickle Trunk*. Available from: http://www.cgjennings.ca/toybox/lsystems/ [Accessed 14 Nov 2013].

[2] Science Daily - Science Reference, 2013. *Morphogenesis* [online]. Science Daily. Available from: http://www.sciencedaily.com/articles/m/morphogenesis.htm [Accessed 14 Nov 2013].

[3] Turing, A. M., 1952. *The Chemical Basis of Morphogenesis.* Philosophical Transaction. Royal Society of London.

[4] Prusinkiewicz, P. and Lindenmayer, A., 1990. *The Algorithmic Beauty of Plants.* New York: Springer-Verlag.

[5] Mobygames, 2013. *Middleware: SpeedTree* [online]. Mobygames. Available from: http://www.mobygames.com/game-group/middleware-speedtree [Accessed 15 Nov 2013]

[6] Worrall, D., 1997. *Procedural Composition Tutorial: L-Systems* [online]. Procedural Composition Tutorial. Available from: http://www.avatar.com.au/courses/Lsystems/ [Accessed 14 Nov 2013].

[7] Reddy, M., 2011. *API Design for C++.* Amsterdam: Morgan Kaufmann.

[8] Nielsen Norman Group, 2014. 10 Usability Heuristics for User Interface Design [online]. Jacob Nielsen. Available from: http://www.nngroup.com/articles/ten-usability-heuristics/ [Accessed 29 Nov 2014].

[9]