Transportation Network Generation Framework

Masters Thesis

Peter R. Dodds
MSc Cave 2009–10
Bournemouth Universtiy

August 20, 2010

Contents

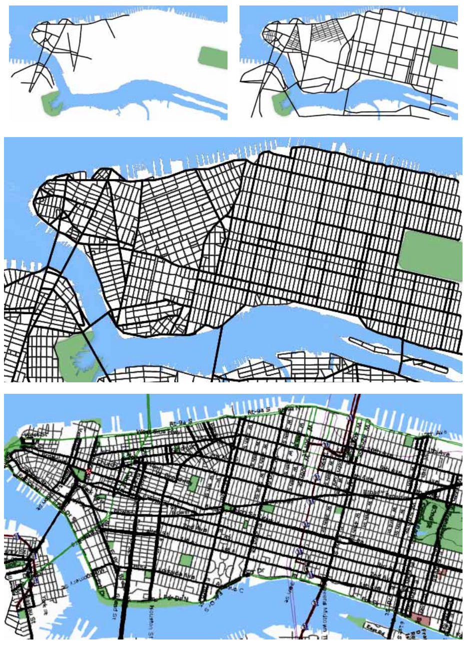


Figure 1: CityEngine street creation system applied to Manhattan. Top row: The network after 28 and 142 steps. Middle: The final roadmap. Lower: A real map of Manhattan's streets for comparison. (Parish and Müller, 2001)

1 Introduction

In the computer graphic and games industry, there is a demand for ever increasing complexity, detail and realism in the work produced. Traditionally this would require increasing the number of artists to meet demand. This method does not scale though, and new approaches must be taken to generate content (Kelly and Mccabe, 2006).

The generation of urban landscapes is one of these large asset creation problems, and is often solved by using procedural techniques. The fundamental component of this generation, is the creation of a road, or transportation network to build a city upon.

One modern approach is to generate in a single pass, an entire road network. A downfall of these batch method though, is that it doesn't fully model the historical growth of town and is more suited to the generation of post- industrial cities which follow a grid pattern (Kelly and Mccabe, 2006).

A clear example of this can be seen in an example given by Parish and Müller (2001) of the output of *CityEngine*. The system was used to generate the road layout of Manhattan, New York (Fig. 1) and in most respects, it created an accurate facsimile of the road layout. One of the obvious failures is the absence of Broadway, the original Indian "Wickquasgeck" trail (Shorto, 2004) that ran the length of Manhattan Island. In Parish and Müller (2001), this cannot be foreseen due to the lack of any city 'growth' and is therefore missing.

In this project I aimed to try and develop a C++ map generation framework that could be used to do advanced repeated road generation. This framework would be intended to overcome some of the limitations of *CityEngine* (Parish and Müller, 2001) by enabling an iterative growth over time.

2 Previous Work

The generation of road networks, and urban environments in general, is a difficult task due to the complexity and variation in the patterns that appear. It has, in past research, been approached in using several different methods; basic grid layouts, similar to modern planned American cities; Lindenmayer system, or L-system, generator; agent based simulation; and template based generation, which applies a desired road pattern onto geographic information (Kelly and Mccabe, 2006). Of these methods, the L-system and agent based methods present the most natural road networks.

Agent based simulation can be used to simulate the behaviour of developers, planning authorities and road builders. This can generate 'evolving' road networks that change over time (Kelly and Mccabe, 2006).

The other method, using L-systems, has been used by the city generation application CityEngine (Parish and Müller, 2001). Using this method, in a single pass, the city application is able to create realistic city layouts as demonstrated in Figure 1. As discussed in Section 1, although it is on the surface very accurate, it is missing some of the nuances which may be better generated using an agent based model.

3 Technical Background

3.1 L-Systems

L-systems or *Lindenmayer* systems is a rewriting system that operates on structured strings (Měch and Prusinkiewicz, 1996). They were first introduced in 1968 as a theoretical framework to study multicellular organisms (Prusinkiewicz and Lindenmayer, 2004). A visual example of an L-system rewrite can be seen in Figure 2.

An L-system string is made up of symbols with arguments called *modules* and square brackets which denoting branching structure. A simulation is initiated with an *axiom*, an initial string, which then has *rewriting rules* or *productions* applied to it. If the *context*, the modules surrounding elements, and any *conditions* of the rewriting rule are satisfied, the current module, or *predecessor* is replace with a *successor* string (Měch and Prusinkiewicz, 1996). The successor string may contain more than one module, branching elements or a single termination character. The termination character, ε, symbolises a removal of the module. Figure 3 demonstrates a simple L-system with some of these features.

Měch and Prusinkiewicz (1996) describes a further extension of the L-system to enable the bi-directional communication of plants with their environment. Called *Open L-system*, the design adds additional *communication modules*. These modules, taking the form '$?E(x_1, \ldots, x_m)$', receive and transmit environmental information. This information is stored within the modules arguments '$x_1, \ldots, x_m$'.

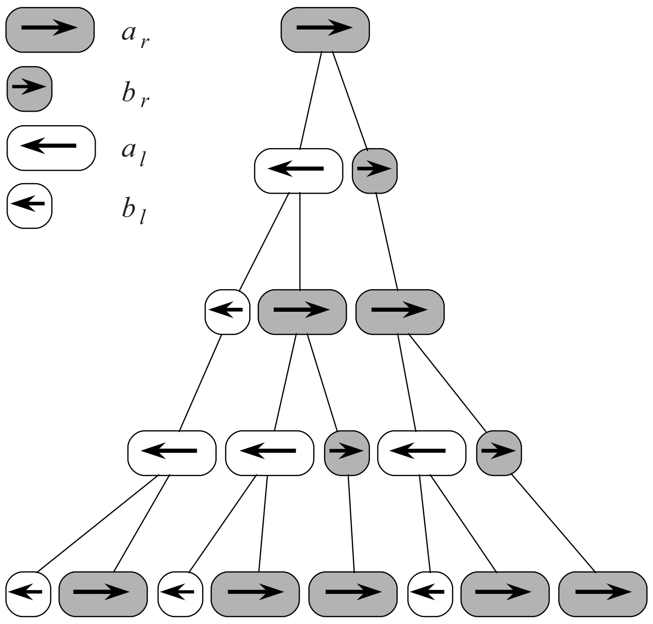


Figure 2: Development of a filament (Anabaena catenula) from Prusinkiewicz and Lindenmayer (2004).

$$\begin{aligned}
\omega :&\quad A(1)B(3)A(5)\\
p_1 :&\quad A(x) \rightarrow A(x+1) : 0.4\\
p_2 :&\quad A(x) \rightarrow B(x-1) : 0.6\\
p_3 :&\quad A(x) < B(y) > A(z) : y < 4 \rightarrow B(x+z)[A(y)] : 0.6
\end{aligned}$$

$$A(1)B(3)A(5) \Rightarrow A(2)B(6)[A(3)]B(4)$$

Figure 3: Example L-system with first derivation step (Měch and Prusinkiewicz, 1996).

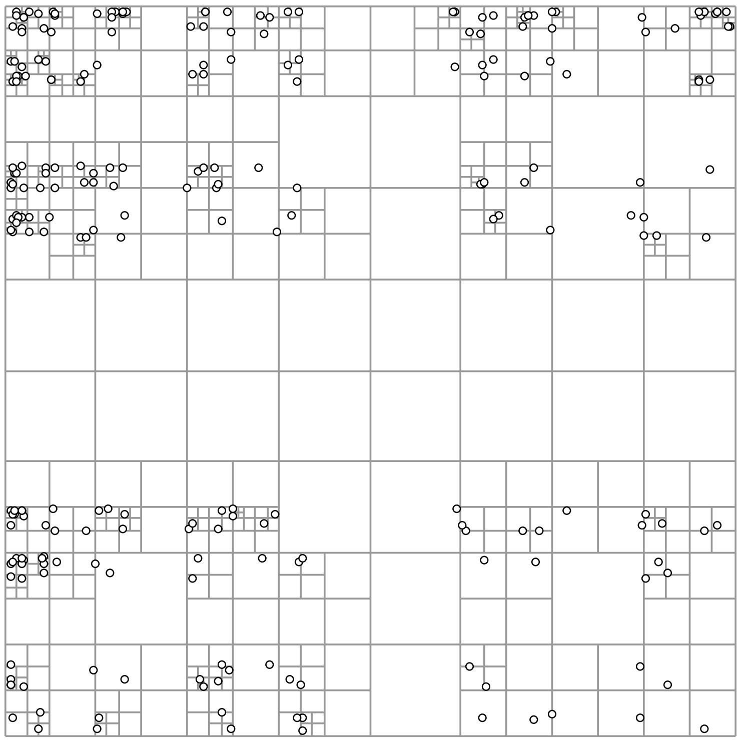


Figure 4: An example quad-tree storing two-dimensional points (Goodrich and Sun, 2005)

The Open L-system enables environmental feedback creating very realistic plant structures. Parish and Müller (2001) also uses this system, applying it to streets, to create natural road networks.

3.2 Quad-trees

A quad-tree is a tree data structure in which each node has four branches and can be used as multidimensional database (Weisstein, 2010). This can be achieved, when inserting points into a two-dimensional space, by splitting an area into four pieces, and then dividing each of those pieces into four pieces, until an inserted point is stored in an empty branch (Ferraris, 2000) (see Fig. 4).

The advantage of sorting data into quadrants comes when searching the database. When looking for the nearest neighbour, with only a simple array of data, the program must iterate through the entire data set comparing the distances. With a quad-tree, the search is limited to the local quadrants that surround each node, drastically reducing search times.

3.3 XML

Extensible Markup Language, abbreviated *XML*, is a ASCII file format intended to describe the behaviour of programs (Bray et al., 2008). XML is made up of elements defined by markup *tags* which can optionally contain character strings

```
1  <?xml version="1.0"?>
2  <conversation>
3     <greeting>Hello, world!</greeting>
4     <pause/>
5  </conversation>
```

Figure 5: An example XML file. Line (1) is the XML declaration describing the version of XML used. (2) starts a *conversation* element with its tag. (3) contains a *greeting* element containing a character string. (4) is an element with no content. (5) closes the *conversation* element.

with markup (see Fig. 5). A tag is identified by a pair of '`<`' and '`>`' surrounding a single name. Each tag is opened in this way, and then closed, this can be done by added a '`/`' character at the end of the tag, or by inserting another *closing* tag. The closing tag contains the same name as the opening tag preceded by the '`/`'. The text and markup between the opening and closing tags it the element's content.

A tag can also have additional attributes, these are appended within the opening tag. For example, '`<object name="apple">`' contains an additional '*name*' attribute with the value '*apple*'.

3.4 Shapefile

A basic geometric data format developed by ESRI (1998), it is used for geographic vector based data in the form of points, poly-lines and polygons. Stored as in a binary format, each shapefile can only contain one type of data, for example, only poly-lines (ESRI, 1998).

3.5 Wavefront OBJ

The *Object file* or *Obj* format, developed originally by Alias|Wavefront (1995b), is an open, human readable and editable, static three-dimensional geometry format. It is widely supported and easy to develop interpreters for.

Object files support points, lines, polygons, curves, surfaces, object names and grouping. In conjunction with a *Material* or *Mtl* file (Alias|Wavefront, 1995a), they can support surface shading information, though this hasn't been used within this project.

The Object file stores data on single lines, using reference character sets at the beginning of the line to identity the data type (see Fig. 6). Focusing on basic polygon shape data, vertices are stored as groups of four real numbers $[x, y, z, w]$. These points are then collected into faces, lines or point sets, each of which require the indexes of the vertices within the set. The indexes can either be absolute, referencing the vertex using its sequential ID, or relative to the current line using negative indices (Alias|Wavefront, 1995b).

4 Approach

In this project it is intended that a set of low level C++ classes and methods be developed to generate transportation networks that develop over time.

```
1 # A triangle
2
3 o 1
4
5 v 1.5 -25 0.2
6 v 24 3.2 0.11
7 v -98 21.45 0
8
9 f 2 1 3
```

Figure 6: An example *Object file.* Line (5–7) contain the vertex data and line (9) has the face information, references the vertex indices.

The networks will be stored as two dimension graph-network stored within a managed container. The network's generation will use Lindenmayer systems, or L-systems, to generate natural paths that change over time.

For the L-system parsing and processing, a C library will be developed. It will use procedural methods to generate an L-systems string's next derivation.

The transportation networks, described internally as graph-networks, will be made up of inter-connected two-dimensional *nodes.* Each of these nodes contains type information and unique attributes, such as vertical height and capacity. The node networks can be used to describe roads, rivers, airports or any interlinked, two-dimensional transport network.

In addition to the network, additional location specific information, such as terrain, buildings or population density, will be stored as polygon *zones* or rasterised data similar to a bitmap image. This data will directly influence and be influenced by the transportation network.

Networks, zones and other two-dimensional information will be stored within a *map*, A map describes a geographical region and any local information required for network generation.

Global information, such as the date, will be stored in a *world.* Maps must be associated with a world to do generation. At each generation step, the world's internal date will increment by one time step. The world's global information will be optionally time-dependent, meaning global factors influencing generation will change on each step.

The network will be initially loaded – or generated using L-systems – into a map containing terrain information, which is itself attached to a world. Once loaded, the user will be able to step forward in time. As time progresses, the network will change depending on map and world factors specified at start-up.

Subsequent derivations of the original map will be generated by applying agent based path-finding and small L-system generation to alter the network.

An important goal in the development of this project is the pursuit of clear, hight-quality, maintainable code. This will allow the continued development of the framework and the reusability of the code associated with it in the future.

```
1 | # A simple l-system
2 |
3 | a A(1)B(3)A(5)
4 |
5 | r A(x) ~ A(x+1) : 0.4
6 | r A(x) ~ B(x-1) : 0.6
7 | r A(x) < B(y) > A(z) : y < 4 ~ B(x+z)[A(y)] : 0.6
```

Figure 7: Example configuration file for L-system described in Figure 3. Line (1) is a comment; (3) an axiom; (5–7) are rules.

5 Framework

The framework attempts to implement many of the aspects described in Section 4, in some cases by utilising third-party libraries, in others by implementing separate libraries in a more appropriate language as with *Lsys*, the L-system parser. The framework offers a wrapper, set of tools and data structure to allow an application to interact with these libraries and generate transport networks.

5.1 L-Systems

For the Lindenmayer system parsing, it was decided – after being unable to find a suitable third-party library – to develop a separate parsing library, *Lsys*. It was developed in C for two reason; simplicity and speed. In comparison to C++, C has a much more narrow set of features, it contains within its standard library all of the character array parsing required to parse a L-system string and required none of the object-orientation within C++. Secondly, it may be easier, in future work, to improve the efficacy of the C code and get slight speed benefits. Although possibly only marginally faster, when generating L-system derivations repeatedly throughout a map generation, these speed benefits may add up.

The main functionality of Lsys is to take in a character array string and return the next derivative string. This derivative is generated by stepping through the predecessor string, comparing each module to a list of rules and, when a match is found, replacing the predecessor module with a successor set of modules.

Lsys uses configuration files, with the `.lsy` extension, to store an L-system's definition, its rules. These files contain the rules to describe the system and optional initial axioms in a simple to read ASCII format. Each line is started with a single character denoting the information on that line – `a` for axiom or `r` for rules (Figure 7). For axioms, after the identification character and a dividing space, the full l-system string is defined. Rules are defined as described in Figure 8, with each component separated by the corresponding – or similar – Open L-system (Měch and Prusinkiewicz, 1996) as described in Section 3.1.

It was the intention, when developing the file format to mimic L-system syntax described in Prusinkiewicz and Lindenmayer (2004), Měch and Prusinkiewicz (1996) and Parish and Müller (2001), but some symbols have had to be replaced to match them with simpler ASCII characters. The replaced symbols are 'ω' became '`a`', '$\rightarrow$' became '`~`', and 'ε' became '`&`'. White-space, other than the

```
r [ lc < ] pred [ > rc ] [ : cond ] ~ succ [ : prob ]
```

Figure 8: Rule format in configuration file, `[` x `]` are optional elements. `lc` and `rc` are the left and right context; `pred` is the predecessor and `succ` is the successor, both of which are required; `cond` is the condition and `prob` is the probability of the rule being applied (Měch and Prusinkiewicz, 1996).

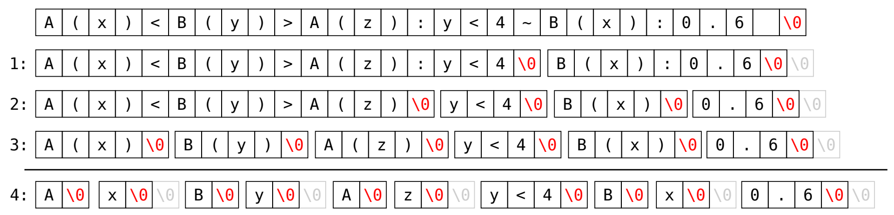

A (x) < B (y) > A (z) : y < 4 ~ B (x) : 0 . 6 \0
1: A (x) < B (y) > A (z) : y < 4 \0 B (x) : 0 . 6 \0 \0
2: A (x) < B (y) > A (z) \0 y < 4 \0 B (x) \0 0 . 6 \0 \0
3: A (x) \0 B (y) \0 A (z) \0 y < 4 \0 B (x) \0 0 . 6 \0 \0
4: A \0 x \0 \0 B \0 y \0 \0 A \0 z \0 \0 y < 4 \0 B \0 x \0 \0 0 . 6 \0 \0

Figure 9: Splitting of the rule string "`A(x)<B(y)>A(z):y<4 B(x):0.6`". *Step 1* trims trailing spaces and splits left and right side by '`~`'. *Step 2* splits the condition and probability by '`:`'. *Step 3* splits the left context, strict predecessor and right context by '`<`' and '`>`'. *Step 4* splits each module into name and argument by '`(`' and trims '`)`'.

first space after the identification character, are ignored.

A modules, for example '`A(1,2.4)`', have a defined syntax. Names – the characters before '`(`' – must be a single capital letter, ie. `A`–`Z`, or one of '`/\+-^%`'. The name can be optionally prefix with '`?`' or '`@`'. Following the name, the module can have a set of up to four arguments enclosed within '`(`' and '`)`'. The arguments are limited within Lsys to four arguments. This is defined by a compile-time constant which can be changed if required.

5.1.1 Parsing

Splitting a string into elements – components of the L-system string – and comparing these elements to rules requires right and left contexts so Lsys must navigate through the L-system string, storing the element in front and the one behind. For this reason, the string is navigated with the method described in Figure 10, passing pointers to each element back from the right context to the predecessor and then to the left context.

When an element, and its right and left context are found to match a rule, the arguments of the element must be matched up to the rule's argument variable names. This is by creating an array of the arguments in the left context, predecessor and left context, in order. Then creating a corresponding array of the rules argument names, as a key. This key is then used to substitute the arguments into the condition, probability and successor, if an argument name is found.

Once the arguments are substituted in, the condition and probability components are evaluated (see Section 5.1.2), and a successor string is produced to replace the predecessor element.

```
char *LCON, *PRED, *RCON, *BUFF

// get first element
BUFF = nextElement
RCON = malloc strlen(BUFF) of sizeof(char)
strcpy BUFF to RCON

for each element:
    LCON = PRED // make current element previous
    PRED = RCON // make next element current

    // get next element
    BUFF = nextElement
    RCON = malloc strlen(BUFF) of sizeof(char)
    strcpy BUFF to RCON

    // compare element to rules ...

    free LCON

free PRED and RCON
```

Figure 10: L-system string navigation loop. For a given string made up of L-system *elements*, loop over each value while storing the next and previous value.

5.1.2 Eval

When processing a rule's successor arguments, condition or probability, the numeric value of these elements must be calculated. Stored internally as character arrays, they must be converted to floating point number.

Lsys implements a custom string to float conversion function to replace the standard C function `atof`. The problem with `atof` is that it is not possible to do reliable error checking with its output. When `atof` fails to convert a string to a float, it returns a float of value `0`. This of course cannot be used as an absolute sign of failure since it would also return `0` when the string "`0.00`" is passed to it. The Lsys string to float converter uses `atof`, but when exactly zero is returned, it does a basic check to test if the string starts with a zero character. This is not a robust method of error checking, but catches most of `atof`'s errors.

It is also possible for those same parts of the rule to contain mathematics and logical operators (see rules in Figure 7 for examples). Before evaluation of the element, it is tested to find any of these operators within the string. The possible maths operations include the basic operators, `+`, `-`, `\` and `*`, plus boolean comparison operators, `==`, `!=`, `<`, `>`, `<=` and `>=`. In the case of the comparison operators, when applied they return a floating point value of zero for *false*, or one for *true*.

The evaluation is done within Lsys by parsing the argument string from left to right, splitting it by the first operator found and comparing the last value calculated, the operation's left-hand side, to the next string, the right hand side. Once the right hand side has been evaluated and the operation applied, the result becomes the left hand side of the next operation, if there is one.

The evaluation of the right hand side, and the evaluation of the first left-hand side value, does not need to split the number from the characters in the string because of the behaviour of `atof`. When converted from string to floating point number, any characters after the last numeric character are ignored. This allows the operator evaluation to skip trimming the input.

5.1.3 Probability

When calculating the probability of a rule replacement occurring, Lsys must generate a pseudo-random number. For this, the standard C `rand` function is used. Although not a high-quality pseudo-random number generator, `rand` is able to produce enough variation.

The `rand` and `srand` functions are wrapped in Lsys functions though, the random function differs from `rand`. Rather than producing an integer value between zero and `RAND_MAX`, Lsys generates a uniform deviant from the output of `rand` similar to the method described by Walker (2007)

5.1.4 Communication modules

The bi-direction communication features of Open L-system (Měch and Prusinkiewicz, 1996) – which were to be incorporated into the framework – have not been put within Lsys. These components require too much interaction between the application and the L-system, and so have been left for the framework to implement.

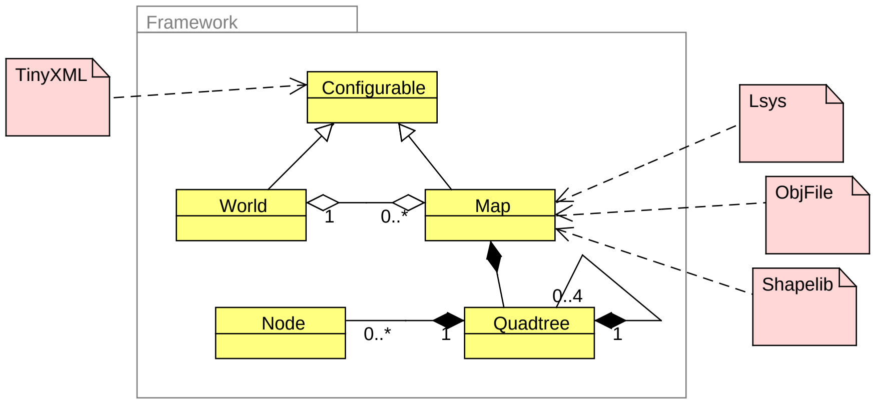


Figure 11: The framework class structure, demonstrating the relationship between each component. Also shown are the libraries included within the framework.

5.2 Vector Maths

Vector maths performed by the framework is done using the *CGMath* library (Dodds, 2010). This library – still in development – provides vector, matrix and quaternion maths functions optimised for two- and three-dimensional computer graphics work. The framework specifically uses the `cgm::Vector2` to do two-dimensional vector calculations.

5.3 Map Graph Network

As described in Section 4, the transportation network is an inter-connected graph-network of two-dimensional points, or *nodes*. The network is defined by the connections between these nodes, their positions and the additional attributes associated with them.

Within the framework, these nodes are stored inside of a managed container. The managed container is intended to be a quad-tree similar to the type described in Section 3.2. This container is implemented in the framework as `QuadTree`, a template class.

Initially it was looked into using a third-party C++ quad-tree container, but there appeared to be no standard method. For this reason, it was decided to develop a quad-tree container, modelling it on the standard template library's container classes, with the intention of creating a fully reusable class. The use of a templates allows the class to be used to contain any data type distributed over a two-dimensional space.

Currently, `QuadTree` only implements the features of a quad-tree partially. All data is currently stored in the root quad-tree in an internal `std::set`. This offers no benefit over other container types other than the ability to later replace the quad-tree's behaviour without damaging code that relies on it.

To navigate the quad-tree template class, an iterator class was developed.

Derived from `std::iterator`, it is a forward iterator meaning that it can be incremented but not decremented. This iterator is, in its current implementation, a wrapper to the `std::set<T>::iterator` that can be used to navigate the internal data of `QuadTree`.

The current iterator is the basis of an intended ‘fast’ iterator that does an unsorted traversal of the quad-tree. The second type was intended to be a ‘slow’ sorted traversal that iterated out from the starting node through its closes neighbours. Although slower for each set – iterator would need to calculate the next node in comparison to the already navigated nodes – in situations in which the iterator is looking for the closest node with matching a specific condition, then on finding the node the iteration can cease.

As it is currently implemented, the layers, zoning and other surface data described in Section 4 have not been incorporated.

5.4 Configurable Classes

The local and global data, intended to be used by the framework to seed network generation, is stored in the `Map` and `World` classes respectively. These classes load in configuration data at runtime and must be able to dynamically create, access and save environment attributes. To accomplish this, both classes are based on a dynamic configuration class, `Configurable`.

The configurable works as a wrapper to the XML editing library, *TinyXML* (Thomason, 2010). Inside of `Configurable` is an XML document (see Section 3.3) containing all of the properties. Within the class are protected methods to set and get properties as well as public load and save methods.

`Map` and `World` are derived from `Configurable`, and so inherit its functions. When a file is loaded by one of these configurable classes, the internal load function calls the XML parser. In some cases, the configurable class may need to parse custom data from the configuration file. For this reason, `Configurable` contains a pure-virtual function `ParseConfig`, which must be implemented by the derived class. This also has the positive side effect of preventing creation of empty `Configurable` class.

The configurable class requires that each of the XML files loaded has a ‘Header’ element (see Fig. 12). This can contain meta-data about the file but importantly specifies the `Configurable` file type. A configurable class’s file type is denoted by a unique string for each derived class, by default its value is “cug::Configurable”. The class `Configurable` uses this to assert that the correct file type is being loaded into each class.

5.5 Input Data

The *Map* class is able to load in external data set to use as network data. This is to enable the use of real world data to increase the realism and detail of the simulation.

The primary cartographic format used is *Shapefile* (ESRI, 1998), which is described in Section 3.4. It is a well known format used often for mapping data (Dhulipudi, 2008), which allows the use of accurate geographic data. The files are loaded into *Map* using Shapelib (Warmerdam, 2010).

The other format loaded is the Wavefront OBJ (Alias|Wavefront, 1995b) (see Section 3.5). These offer better a format for an end user to create content

```
 1  <?xml version="1.0" standalone="no" ?>
 2  <Header>
 3      <FileType>cug::World</FileType>
 4      <Date>
 5          <Created>Tue Jun 29 09:44:48 BST 2010</Created>
 6          <Modified>Tue Jun 29 09:44:48 BST 2010</Modified>
 7      </Date>
 8      <Name>DemoMap</Name>
 9      <Author>Peter Dodds</Author>
10      <Copyright>None</Copyright>
11      <CugVersion>0.1</CugVersion>
12  </Header>
13  <!-- data -->
```

Figure 12: An example configurable class header. Line (3) contains the `World` file type string. The file would continue after line (13) with the world's configuration data.

by hand. Initially it was intended to use a third-party library to load an OBJ, but since the format was almost universally used to save polygon models the available formats didn't read line or point sets.

The C library ObjFile, developed for the framework, reads in the Object file format. It works by looping through a `.obj` file twice – first time to count the count the number of instances of each data type; second time to read the data – and stores it within a structure.

For the creation of test files, the graphics application Maya (Autodesk, 2007) was used. To save the data out of Maya, the graphics package being used, initially it was thought possible to use the supplied ObjExport plugin. After testing though it was clear that it didn't fully support the Wavefront OBJ specifications. For this reason, a MEL script was used to export lines and points in the OBJ format.

In both Shapefiles and Wavefront OBJs, lines are stored as points with a maximum of two connections. This mean, when loading them into a multi-link graph network of points, the nodes must be checked for overlap. When a node is added to the map, it is compared to its nearest neighbouring node. If they are in the same position, no new node is created, and the equal node is returned. This comparison, in the current version of the framework with the absence of a working quad-tree, requires the node to be compared with all existing nodes.

5.6 Exceptions

Within the framework, C++ exceptions are used to do run-time error handling. These are custom error types, which are thrown when abnormal or erroneous behaviour occurs. It is then up to the user to catch these exception errors or let the application exit, if appropriate.

There is a base exception type, derived from `std::runtime_error`, which each class within the framework derives a nested custom exception class, eg. `cug::Configurable::Error`. Each error, when generated, is passed a *type* along with a message in the from of a `std::string`. The *type* is used to differentiate between different errors generated by that class. A list of these types is defined within the classes header and is extended from its parent error class. The message is passed – if the exception is not handled – to `std::runtime_error`, which displays it to screen when the program exits.

5.7 Render

The rendering of L-systems and maps have been kept separate from the framework, and left up to the application using the framework to decide and process. In this project though, to demonstrate the generation, a demo renderer was included that used Cairo (Worth and Packard, 2007), a vector renderer, and GTK+ (GNOME Foundation, 2009), a cross platform window. Some examples of the rendering output from Cairo and the framework can be seen in Figures 13–16.

To render a map, the rendering loop must iterate over the graph network within `Map`, and draw each individual node. If the connections between nodes are to be drawn efficiently, the application must also store each node it has already drawn. This 'closed' list can then by used to check if the link between the currently being drawn node and the linked node is to be drawn. If the linked node is in the 'closed' list, its links have already been drawn and so the link dose not need to be drawn again.

5.8 Code Testing

All code was tested with valgrind for memory leaks and errors. As stated in Section 4, code quality and efficiency was an important goal during development. The continued development and re-usability of the of the code will be influenced by its quality.

6 Results

The current version of the framework only scratches the surface of the originally proposed idea in Section 1. It is currently able to generate street-like node networks, but these do not model most of the behaviours described in Parish and Müller (2001).

The first parts implemented were the node graph-network and the asset loading. Shapefile, with use of the third part libraries, were loaded into the framework relatively effectively (see Fig. 13). Once the Wavefront OBJ loader and Maya export were implemented, those too were incorporated into the framework. In both cases, when the data was loaded into the framework, the node container, `QuadTree`, started to show some of the expected limitations. On large map files, the comparison between each node took several seconds to perform when over 3000 points had been loaded.

The implementation of *Lsys*, the Lindenmayer system parser, developed rapidly once started, quickly being able to demonstrate large L-system string parsing. Early renders of Lsys-only networks were able to create natural appearing, two-dimensional plants (see Fig. 14). It was also possible to simulate road-like structures, as demonstrated in Figure 15, but without checking for overlapping, it is limited in its appearance.

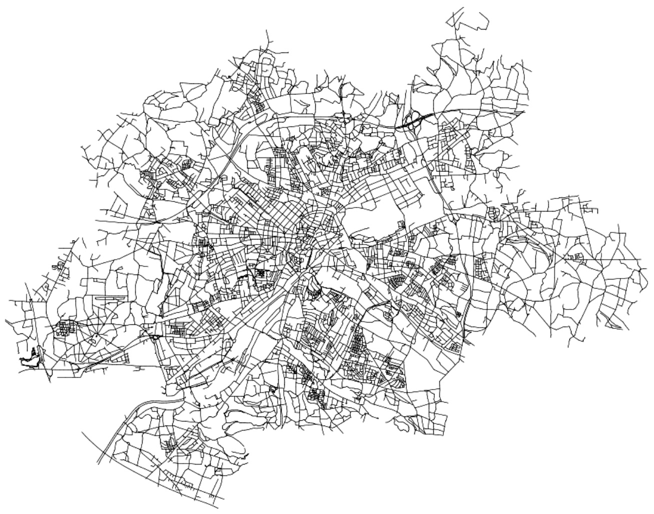
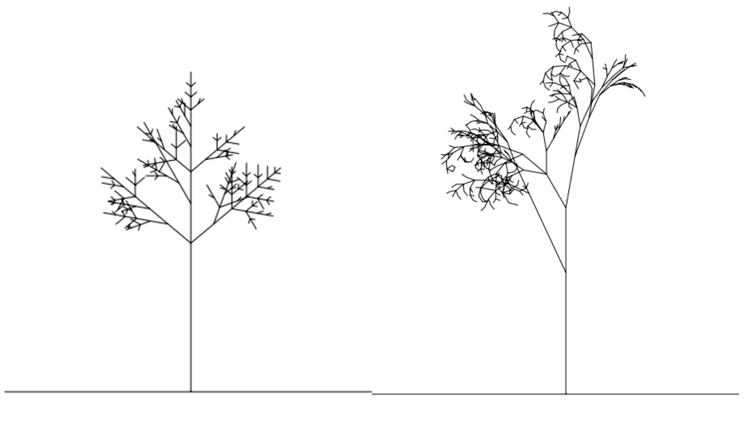


Figure 13: Loaded Shapefile, from Dhulipudi (2008), into the framework



(a) Basic system (b) More complex system

Figure 14: Simple L-systems developed using Lsys

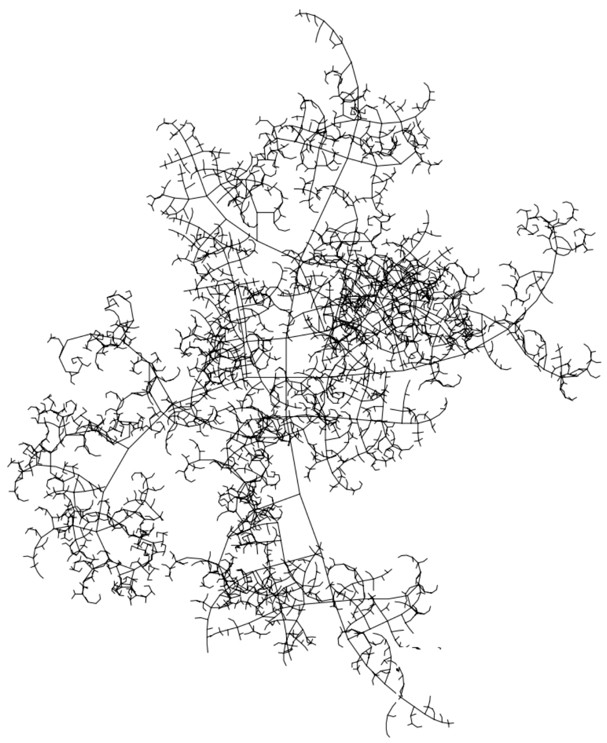
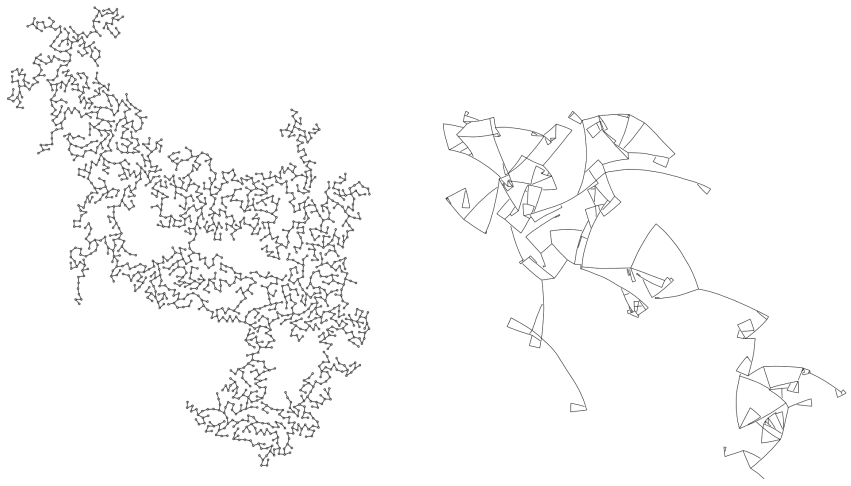


Figure 15: A L-system generated by Lsys.



(a) No overlapping (b) Straight roads

Figure 16: L-systems created in the framework.

When Lsys was integrated into the framework, so that the generated strings created new nodes, attached to the network, it was possible to do inter-node 'collision' testing. When each node was placed, it was tested against the existing nodes to check it was a minimum distance away (see Fig. 16a). As it is implemented, this is a hard-code procedure that will not scale.

Many of the current behaviour of the framework is hard-coded, similar to the 'collision' avoidance. In addition, through the development of the code, many missing components were identified.

7 Future Work

In future work, it is planned to address the missing functionally of the framework. This will be done by reassessing the approach being taken, and redefining some of the goals.

The first goal of any future development would be to recreate the road generation system described in the CityEngine paper (Parish and Müller, 2001). From there, the framework could begin implementing some original goals – described in Section 4 – that have yet to be achieved, ultimately aiming to produce the time depended city engine originally proposed.

7.1 Lsys

Though already very robust, the L-system parser still could be significantly improved by implementing the full L-system syntax. This would include expanding the `.lsy` file format to process defines, arrays, comments, macros and replacement rules. These missing features can be seen in Figure 17.

One method of extending the L-system format to include missing features would be to re-implement the file parsing code to use *lex* and *yacc*. This would increase the libraries future flexibility and maintainability by replacing difficult to manage string parsing code with a common set of string parsing languages.

```
 1  #define $\alpha_0$ 0.5             /* none ASCII character */
 2  #define Len 0.2                    /* multi character */
 3  #define N 3                        /* array length */
 4  Define: { array
 5  Req[N] = {0.1, 0.4, 0.05},         /* array of size N */
 6  Del[N−1] = {30, 60}                /* array of size N−1 */
 7  }
 8  #define Prob$_R$($x$)                ($0.12 + x \times 0.42$)
 9
10  $\omega$ : A(1,1)
11  $p_1$ : A(dir, x) : (x < N)&&(x>=0) {h = dir/Req[x];} → B(h) : Prob$_R$(x)
```

Figure 17: An example of missing L-system grammar. Line (1–8) are *defines*, (4–7) *arrays* and (8) a *macro*. Line (11) shows the defines in use and a *replacement rule* between '{' and '}'.

An obvious cost to moving the L-system parser to use lex and yacc would be researching the tools and then the re-write time. In the long run though, this may become an insignificant cost and potentially yield extra code performance benefits.

String parsing using lex and yacc could also be incorporated into the L-system's internal evaluation function described in Section 5.1.2. Currently, only the basic mathematical operations, `+`, `-`, \ and `*`, and the comparison operations are supported. These also only do simple left-to-right processing – preventing the use of brackets or long operations – and string to number conversion using a wrapper to the C `atof` function.

Other improvements to be looked at – which might or might not be resolved by moving to a better parser – include the review of function string buffers. In the case of the splitting of module's arguments, the module is limited to a maximum of *four* arguments by a compile time definition. This can be changed by recompiling the library but a more practical solution would be to dynamically resize the argument buffer to cope with varying input. This sort of solution can be applied to other similar string buffer problems.

Finally, in its current state, *Lsys* does not do adequate error checking. For example, if a rule definition contains a module with a different number of arguments to the module it is being compared to, undefined behaviour may occur. This may end in the program crashing or returning incorrect next derivation and possibly later causing the program to crash. This kind of error must obviously be caught or handled appropriately.

7.2 Quad-Tree

In the quad-tree container, as it is currently defined, points are not sorted by their positions into the correct quadrant. This fundamental aspect of the `QuadTree` class must be implemented to take advantage of the spacial sorting.

At its current point of development, the quad-tree also has no method of navigating nested data. To do this, the `QuadTree` iterator must be fully implemented to traverse the tree's leaves. The iterator currently implemented is a wrapper to the `std::set<T>::iterator`, the underlying quad-tree leaf data iterator. This acts like a fast, unsorted iterator, bound to the contents of a single quad-tree node. It needs to be fully implemented and for the second, sorted iterator to be added.

Currently, without the advantages of the spacial sorted quad-tree and the sorted iterator, the framework still responds reasonably quickly on a modern computer. As more data is passed to the quad-tree, and each node is compared for duplication, the comparisons take longer, significantly slowing the application down.

7.3 Layers and Terrain Data

A fundamental missing feature is the ability to layer multiple networks within a map. This is an important feature to organise and split the data into different types. For example, a road network and a rail network may interact, but they interact differently than with their own nodes. Dividing it into a separate layer organise the data and increases the simulations efficiency by reducing the number of node comparisons.

The additional data within each node, intended to store height data amongst other things, is also not implemented in this version. Specifically, the missing height data is a crucial component of the road generation. Without it there it cannot model real-world terrain interaction.

As well as the height information of each point, the map terrain information is missing. In future versions this will be loaded from a data source – such as Ordinance Survey (2010) which provides spot height data in the DXF format (Autodesk, 2010) – into a surface grid stored, similarly to the transport network, in a graph network of nodes. This would then be used to calculate the height at any point in the terrain limits. If raster data was loaded instead of point heights, this data will be converted to point data.

Zoning will also be included in future versions of the framework. Described with two-dimensional polygons, zones will be used to distinguish area values such as population density, buildings and bodies of water.

7.4 Output Format

In future versions of the framework, it will be important to consider creating a map format to store all of the internal network data and fully describe a map. The formats currently used to load data – Shapefile and Wavefront OBJ – would not be suitable since they are unable to store the additional attributes each node will contain. A possible format for these map files might be using the XML file format, similar to the configurable classes.

Once the map is developing maps over time, as originally proposed, the map file format will need to be able to store each of these steps. It could work in a similar manner to a renderer, saving a complete snapshot at each time-step, though this would start rapidly start creating an impractical number of files. An alternative method would be to store only the differences between, or *delta* of, each time-step.

8 Conclusions

As described in Section 7, development of the city generation framework has only scratched the surface and has still many fundamental aspects to be implemented. At its current stage, it has yet to demonstrate the content generation of CityEngine (Parish and Müller, 2001).

The slow development of the city generation framework can be traced back to an initial misplaced emphasis on fully implementation of asset loading and node structure. This delayed the development of the L-system parser, which was only started over half-way through the project.

Had the focus of the project early on been firstly on a simple L-system parser and subsequently on implementing the environmental feedback described in (Měch and Prusinkiewicz, 1996), by finishing the project a reasonable facsimile of CityEngine's road generation could have been achieved or even built upon.

Overall, although the progress of the project has been significantly slower than originally hoped, the research and code developed has created a strong base to continue developing the urban landscape generator.

It is the intention to continue the development of the ObjLoader and Lsys libraries to the point of being able to be released. In conjunction with this, the development of a growing city simulation framework will also continue.

References

Alias|Wavefront (1995a). Mtl material format (lightwave, obj) [file format specification], `http://local.wasp.uwa.edu.au/~pbourke/dataformats/mtl/` [Accessed August 17, 2010].

Alias|Wavefront (1995b). Object files (.obj) [file format specification], `http://local.wasp.uwa.edu.au/~pbourke/dataformats/obj/` [Accessed August 17, 2010].

Autodesk (2007). Maya 2008, version 9.0 [software].

Autodesk (2010). Dxf reference [file format specification], `http://usa.autodesk.com/adsk/servlet/item?siteID=123112&id=12272454&linkID=10809853` [Accessed August 17, 2010].

Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E. and Yergeau, F. (2008). Extensible markup language (xml) 1.0 (fifth edition).

Dhulipudi, D. P. (2008). Rendering shapefile in opengl, `http://www.codeproject.com/KB/openGL/RenderSHP.aspx` [Accessed August 19, 2010].

Dodds, P. (2010). Cgmath [programming library], `http://github.com/m0tive/cgmath` [Accessed August 19, 2010].

ESRI (1998). Esri shapefile technical description [file format specification], `http://www.esri.com/library/whitepapers/pdfs/shapefile.pdf` [Accessed August 19, 2010].

Ferraris, J. (2000). Quadtrees, `http://www.gamedev.net/reference/programming/features/quadtrees/` [Accessed August 19, 2010].

GNOME Foundation (2009). Gtk+, version 2.16.1 [programming library], `http://www.gtk.org/` [Accessed August 19, 2010].

Goodrich, M. T. and Sun, J. Z. (2005). The skip quadtree: a simple dynamic data structure for multidimensional data, *In Proc. 21st ACM Symposium on Computational Geometry*, ACM, pp. 296–305.

Hampshire, N. (2009). *Dynamic animation and re-modelling of l-systems*, Master's thesis, Bournemouth University, Bournemouth, UK. Available at `http://nccastaff.bournemouth.ac.uk/jmacey/MastersProjects/MSc09/Hampshire/index.html`.

Kelly, G. and Mccabe, H. (2006). A survey of procedural techniques for city generation, *ITB Journal* (4): 87–130.

Měch, R. and Prusinkiewicz, P. (1996). Visual models of plants interacting with their environment, *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, pp. 397–410.

Ordinance Survey (2010). Os opendata supply, `https://www.ordnancesurvey.co.uk/opendatadownload/products.html` [Accessed August 17, 2010].

Parish, Y. I. H. and Müller, P. (2001). Procedural modeling of cities, *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, pp. 301–308.

Prusinkiewicz, P. and Lindenmayer, A. (2004). *The Algorithmic Beauty of Plants*, Springer-Verlag New York, Inc., New York, NY, USA. Available at.

Shorto, R. (2004). *The Island at the Center of the World*, Doubleday, p. 60.

Thomason, L. (2010). Tinyxml [programming library], `http://www.grinninglizard.com/tinyxml/` [Accessed August 18, 2010].

Walker, J. (2007). Using rand(), `http://www.eternallyconfuzzled.com/arts/jsw_art_rand.aspx` [Accessed August 18, 2010].

Warmerdam, F. (2010). Shapefile c library, version 1.3.0b2 [programming library], `http://shapelib.maptools.org/` [Accessed August 19, 2010].

Weisstein, E. W. (2010). Quadtrees – from woldfram mathworld, `http://mathworld.wolfram.com/Quadtree.html` [Accessed August 19, 2010].

Worth, C. and Packard, K. (2007). Cairo, version 1.8.10 [programming library], `http://cairographics.org/` [Accessed August 19, 2010].

A Appendix

A.1 Source Code Guide

Following is a guide to the source code supplied with this project.

framework/include/cug The framework header files:

- globals.hpp – Constant, macros and base exception class `Error` definitions,
- node.hpp – `Node` class declaration,
- quadtree.hpp – `QuadTree` and `QuadTree::Iterator` template class declarations and definitions,
- configurable.hpp – `Configurable` class declaration,
- map.hpp – `Map` class declaration,
- work.hpp – `World` class declaration.

framework/src The framework source files:

- node.cpp – `Node` definitions,
- configurable.cpp – `Configurable` definitions,
- map.cpp – `Map` definitions,
- world.cpp – `World` definitions.
- main.cpp – Demo application and executable entry point.

framework/lsys/include/lsys The L-system parser, Lsys, header files:

- lsys.h – Lsys structure, constants and macro definitions, and function declarations.

framework/lsys/src Lsys's source files:

- lsys.c – Lsys file opening and closing functions and random number generator,
- parser.c – Lsys string parsing and rewriting functions.

framework/objfile/include/objfile OBJ loader, ObjFile, header files:

- objfile.h – ObjFile library data types definitions and function declarations.

framework/objfile/src ObjFile header files:

- objfile.h – ObjFile function definitions.

framework/cgmath CGMath library files.

framework/tinyxml TinyXML library files.

framework/shapelib ShapeLib library files.

mel MEL scripts

- objexport.mel – Wavefront OBJ MEL export script to save out lines and points.