

City Architecture Generation

Tom Kelly



A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of
Master of Engineering in the Faculty of Engineering
August 2006 | CSMENG-06

Declaration

This dissertation is submitted to the University of Bristol in accordance with the requirements of the degree of Master of Engineering in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.

Tom Kelly, August 2006

Summary

In this thesis I present a historical city generation tool named *Sity*. It allows video game artists to design the style of the buildings and then use *Sity* to create an area of architecture.

The games industry is worth £13 billion worldwide per year, and this market has doubled in the last 6 years. Annually game studios spend £3.6 billion creating content for games. *Sity* is a tool to allow studios to control these spiralling budgets by automating the content development process. As Don Mattrick of EA games stated:

“building a game engine is easy; creating content is hard”

The previously used alternatives consist of the labour intensive and expensive approach of creating content in-house or risking loss of artist control and quality to content creation. Game development studios are well funded, highly technical companies that are always looking for a competitive edge over their competition.

The attractions of *Sity* include¹:

- Solving the impending content crisis caused by the increased detailed required for next generation console titles.
- Allowing more draft environments to be evaluated in the same time allowing the consideration of different layouts.
- Reducing art personnel budgets.
- Giving the consumer a better experience with larger and more consistently detailed environments.

Technically I show how the Voronoi diagram and the camp skeleton algorithms complement each other in generating a city. They are combined with some simple inputs and directed by an L grammar.

The program provides a non-programming user with an interface for creating city geometry. It provides a straight forward presentation of the myriad of options available to control *Sity*, using a GUI that is generated on the fly from the internal data structures.



Summary of Work:

- Developed a weighted straight skeleton algorithm that allows negative weights and bevelling (a camp skeleton), and explored the computational complexities of such a structure (page 25)
- Developed an implementation the Voronoi algorithm (page 31)
- Created a set of techniques for approximating a historical city (page 36)
- Written 25,000 lines of Java including use of the JMonkey² scene-graph engine (available from <http://www.cs.bris.ac.uk/home/tk1748/sity>)
- Developed a GUI interface to the city generator, incorporating runtime interface generation and a 3D preview (see user guide in Appendix 2, page 55)

Contents

Contents.....	5
Background and Research.....	8
Commercial Viability.....	8
Expansion.....	9
Research & Considered Approaches.....	9
A Description of Sity's Approach.....	16
Technical Aspects of Sity.....	19
On Skeletons.....	19
On Convex Skeletons.....	20
On Concave Skeletons.....	21
On Camp Skeletons	25
Voronoi Diagram.....	31
Sity Grammar Overview.....	36
Design and Implementation.....	39
Tool chain choices.....	39
Engineering Overview.....	41
Waterfall System.....	42
GUI Design & Generation.....	46
Sheets, Sheaves and Sity's internal Polygon System.....	47
External 3D Generation.....	49
Results.....	50
Future Work.....	50
Business Steps.....	50
Future Work on Sity.....	51
Future Work on Theory.....	53
Appendices.....	55
Appendix 1: Miscellaneous.....	55
Appendix 2: User Manual.....	55
Getting Sity.....	55
Running Sity.....	55
Introduction.....	56
Outputting a City via MEL.....	60
Outputting a city as an .obj file.....	61
Saving and loading a set of waterfalls.....	61
About the Waterfalls.....	61
Appendix 3: Jordan Curve Algorithm.....	64
Appendix 4: Index of Figures and Tables.....	64

<u>Appendix 5: Voronoi Bisector calculations.....</u>	<u>66</u>
<u>Appendix 7: Ear clipping algorithm.....</u>	<u>70</u>
<u>Appendix 8: Union Algorithm.....</u>	<u>71</u>
<u>Appendix 9: Results.....</u>	<u>73</u>
<u>Road Maps.....</u>	<u>73</u>
<u>Generated Cities.....</u>	<u>74</u>

Dedicated to Green & Black's

Background and Research

Commercial Viability

As video games become more complex the resources required to generate content for these video games is increasing. Games developers are facing rising bills for content generation, and risk having to rush work to create the quantity of content required or quality and lose artistic control by outsourcing the work.

Sity is a tool that offers a solution to this problem – procedural geometry. By using a computer to generate large fictional cityscapes, artists are able to spend their time defining the details of the city and then having arbitrarily sized cities created automatically.

The speed of graphics hardware is growing exponentially, guided by Moore's Law. A typical game in 1993, such as ID Software's *Doom*³, may have had three thousand polygons in the game, by 2000 with games such as Ion Storm's *DeusEx*⁴ this had become fifty thousand and a typical game today (mid 2006) such as Ubisoft's⁵ *Ghost Recon: Advanced Warfighter* may have three million polygons in an environment. This growth shows no sign of slowing and its effect can be seen in the typical make up of a game developer shown in Figure 1.

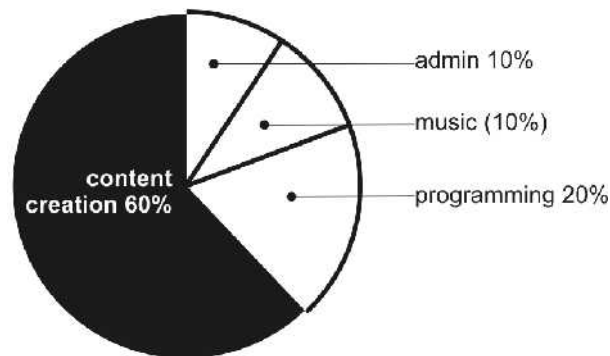


Figure 1 DTI: Typical Development Talent Split⁶

Content creation personnel make up the majority of game development staff and budget, and as the complexity of video games' content continues to rise this is becoming a serious problem for next generation console developers. Finding more intelligent ways to use these content creation staff is essential to creating affordable yet rich and detailed environments. *Sity* is a tool for these content artists to do just that.

Sity as a company offers huge potential:

- There are theoretical models showing that the technologies are viable, which have not been turned into successful competing products,
- There are several engines for the creation of city geometry, but none have been successful commercially (such as Binary World's *Descensor Engine*⁷) or only been applied to other fields (Such as Pascal Mueller's academic project, which has only been used to produce content for films)

- There are already procedural content companies creating other content for digital worlds. For example IDV and their flagship product SpeedTree⁸.

Expansion

After the development of *Sity*, there is the possibility of many spin off products. *Sity* is a tool that generates procedural architecture designed to be copied into a game environment (at game “compile time”). With some optimisation the technologies developed could be used to generate cityscapes at game runtime - every time the user turns a corner a new block of houses awaits them, the same game will never be played twice.

There are also many other potential areas of expansion – from procedural cars to humans. A video game where the ‘bad guy’ changes every time you start a new game offers exceptional value to the consumer.

Research & Considered Approaches

There is a large body of work relating to automated architecture and cityscape generation. Some of the broader topics in this scope include:

- Automated construction of a city from collected data is examined by Takase et al⁹. By combining laser scans, existing 2D maps and aerial photographs the system can digitally reproduce whole cities – drastically cutting down the time required. Manual touching up of complex buildings is required, although the final results are very detailed and accurate.
- Parker and Leach¹⁰ describe the data structures required to endlessly produce 'pseudo infinite' cities. By assigning each building a seed that is used to generate the building an endless city can be generated and explored. This random approach is similar to that I take in this paper.
- Rapid procedural modelling of architecture is tackled by Birch et al¹¹. By understanding some of the common features of architecture a building modelling tool can be created that is much quicker and easier to use than a more general purpose tool. For example opposite walls can be kept parallel and features such as doors can be selected as an entity and moved to a different location, as the program knows this feature belongs on a wall.

There is a substantial body of academic work relating specifically to the automated generation of fictional architecture. The topics covered are quite varied; researchers have approached this problem and various sub problems with a multitude of techniques and varying levels of success. Below I discuss the different aspects of the challenge in context with the research.

In one of the first attempts using a machine learning centric approach was taken by Mitchell¹². He describes the huge potential search space for designing buildings in real world architecture setting. Many parameters - from local building codes, to the quantity of light a window would produce must be encoded and solved to produce a technically correct building. Trickier is the problem of designing an algorithm with a sense of aesthetic beauty, which complicates this search space again. For example we may desire that windows across

a building are different sizes in odd and even columns. Formulating a problem that is sufficiently constrained, without over constraining the problem is a tough problem itself, before solving this massive logic problem is considered.

Mitchell suggests a range of possibilities, from using a computer to design the buildings and humans to verify them to the opposite as possibilities for making use of computers in architecture.

'The logic of architecture' is an earlier book by Mitchell¹³. He describes the kind of first order logic predicates that may be used to describe architecture. Throughout this lengthy volume many approaches for describing architecture in logical form are given. None of these are concise enough to implement and the NP-complete nature of inferring facts from logic problems make this approach a little unrealistic.

My initial response to the problem is to construct an encoding for a building and a city and then to then use heuristic guided search in this cavernous space to find a realistic city. This could be done using a first order set of predicates and assumptions. After reading Mitchell's paper and making a rough estimate of the number of variables that would have to be combined to design a house in one location in a city (30 degrees of freedom) I came to conclusion that searching this space would be too computationally taxing, even if an automated evaluation routine could be created. This is mostly because we would be searching in an arbitrary 3D search space with many complex and interdependent limits. To construct one building would be time consuming, to construct the thousands required to create a city would be beyond the technology available to me.

The rest of the materials I have reviewed have taken a much less theoretical approach. Instead of search, a popular approach is to procedurally create buildings. The speed of finding a solution is much improved, at the cost of realism. Yap's¹⁴ stochastic approach to modelling a virtual Manhattan is typical. A simple grid is imposed in the shape of Manhattan Island. This is then split up into the archetypal grid pattern with definitions for how big the buildings should be in each block. The simplistic buildings are grown stochastically to a distribution around the specified size. From a distance this creates a reasonably interesting view of down-town New York. It is very limited to skyscrapers and American 'block' cities. But if an overview of the LA area were required it may be an appropriate tool to use.

Here I noted that the Yap's concept of a stochastically applied distribution when determining the height of a building may also be extended to many different parameters of a building from the floor plan to the number of panes of glass in the window.

In the very enlightening book *How Buildings Learn*¹⁵, Bond describes the way houses age, and how their appearance changes over time. This was very interesting to me as it supported observations that I had made about buildings. Buildings that have been used for a while end up looking very different to how they were built.

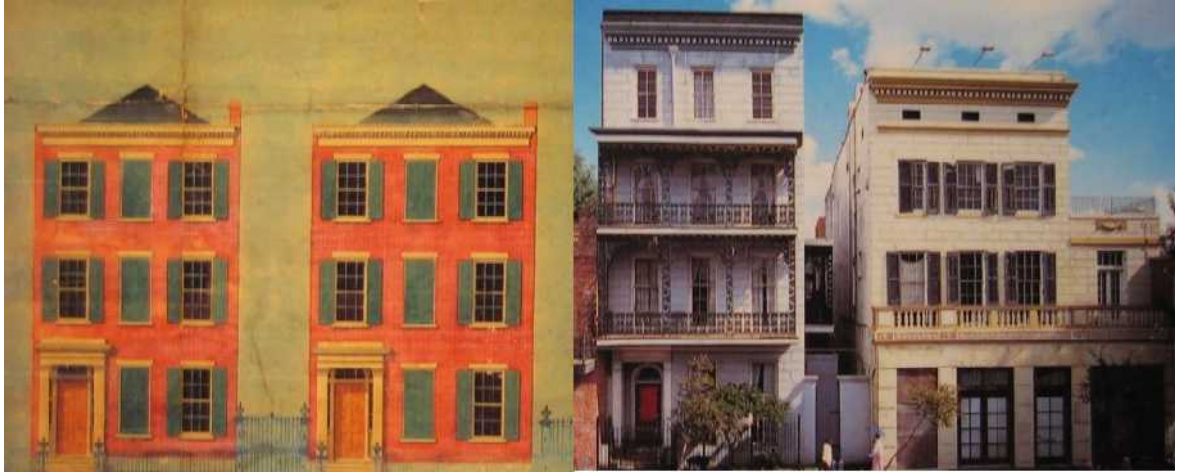


Figure 2- *How Buildings Learn*, front cover, left: houses New Orleans 1857, right same houses 1993

In old cities each house in an initially identical row will have had many different modifications made to it by different owners through the years. Some houses will have had similar alterations to do with current trends (such as PVC windows), others will have unique changes by generations of rich or poor owners (such as telescopes and flag poles).

Bond describes numerous examples of how the floor plan of a house can over time grow. There is also an interesting description of the effect of wealth on house development. Rich home owners are more likely to make changes to their properties. Poorer home owners will not alter the appearance of their houses, and this may preserve many historic features, such as has happened in the older parts of New Orleans (Figure 2).

The end result is that the old buildings seen in European cities are a complex mess of features with underlying patterns. There are more likely to be correlations between buildings close to each other. Such structures are built to similar plans and similar materials and modifications that are made to one house are likely to be applied to other local houses. For examples when conservatories became popular they were added to old and new houses alike. Because of historic building protection laws uniform appearance is not correlated with the age of the property.

This huge variety provides a challenge for city generation. On a positive note for my project it means that irregular features are expected, and from a gross overview outside of the house there is very little order to these patterns.

There is a huge body of related material dedicated to land use modelling. This field is particularly well advanced as it has useful applications in town and regional planning. Mackett¹⁶ describes his iterative microsimulator model, *MASTER*, that partitions cities into zones, and models the composition of families and households to predict zone popularity and growth year on year. These models are frequently used by professional planning consultancies such as MVA¹⁷, and have rigorous economic and mathematical underpinnings. These systems are intended to model the effects of changing parts of a city's infrastructure but do not (yet) deal with house locations or road locations. Instead they deal with the make up of each abstract house, which only uses a zone as its area, and transport costs between these zones. To take this iterative concept to the level of detail that would be able to generate geometry would incur the exponential penalties discussed above by Mitchell.

Another detailed land use model is introduced by Letchner and Watson¹⁸ that works with a rigid American zone system (industrial, commercial or residential) and also generates an associated road map. By using an agent based approach to tertiary (smallest road) road system creation a network is extended that services as much land as possible. The resulting city maps do appear resemble 'western culture within the last century'.

An alternative to using a search agent to construct a road system, is to use a self sensitive Lindenmayer(L)-system as impressively shown by Parish and Müller¹⁹. An L-system uses a simple set of production rules, repeatedly applied to create a fractal-like structure. A system is defined by a set of variables, a set of constants and set of production rules. In the following example X may represent “a window”, Y represent a drainpipe and +,- represent different quantities of space between the items.

Variables: X,Y

Constants: +,-

Start: X

Production Rules: $(X \rightarrow XY)$ $(Y \rightarrow -X+Y)$

This produces the successive generations:

Iteration	String produces
0	X
1	XY
2	XY-X+Y
3	XY-X+Y-XY+-X+Y

Parish and Müller construct larger 'motorway' roads are constructed between areas with high population density. Between these a grammar is used and recursively applied to construct the smaller roads. A modification on the usual L-system is to let the roads be self sensitive, meaning that:

- if the terminal roads of the L system are close to another road they will attempt to move towards it,
- the terminals roads will prefer to join with an existing junction to form an intersection,
- the terminal roads will then attempt to connect with existing roads to form T junctions.

The roads are also sensitive to height and the L grammar used is modified to produce circular, random or Manhattan-esque road maps.

Parish and Müller then go on to subdivide each 'block' between the streets into plots and then to grow a building on it, again using an L-system. Level Of Detail (LOD) groups are used to store multiple representations of group in order to allow very large cities to be computed and rendered, without producing unnecessary detail. Onto these buildings repeating textures are devised to give the impression of doors and windows. The location of doors and windows is specified by performing a binary AND operation between horizontal and vertical binary patterns, so where they intersect is designated as a location for a window or door. This system is used commercially and has produced some very impressive results, although the buildings often appear block-based and too square and regular for an ancient city.

Another possible extension to the L-system is hinted at by Yap's stochastic approach to modelling Manhattan. By pseudo-randomly or stochastically choosing the production rule to use in an L-system it is possible to introduce additional variation. In this case each production rule in a given context is assigned a probability and the computer makes a stochastic choice as to which to use.

There is also a significant body of work related to architecture generation on single buildings. Most of the approaches use a hierarchical organisation of subcomponent parts to define the block first, then the land the building occupies (plot) then the walls and roof, followed by the windows, doors and other adornments. A typical example of this is given by Wonka et al²⁰. He develops a system based on a modified L system using a 'split grammar'. The grammar is very tightly controlled with application rules to ensure that the type of architecture is consistent, for example a window will have the same type of frame on all sides. As it is a L-grammar it is context sensitive and chooses appropriate designs for the size of the feature and the building material. A split grammar allows for the replacement of items of geometry (such as the cuboid representing the top floor of a building) for different sized items of geometry (a triangular prism for the roof). This approach creates a massive range of possible buildings, from those with central arches to cylindrical skyscrapers, but relies on an equally detailed language, grammar and additional rules as to where the different elements of the grammar may be applied. A serious limit to the popular L system is highlighted – that buildings cannot 'grow into another', for example one house being knocked through into another. While we saw Parish and Müller using a self sensitive L system to overcome this with a simple road network, doing the same with the architecture itself would be very challenging.

Wonka et al. go on to another feature that is interesting in buildings – that of repeated symmetry/patterns. For example features such as window size and spacing are likely to be kept symmetrical throughout a floor of a building. The same types of windows are also more likely to be found on different floors of the same building. There will almost certainly be a door on the ground floor. This lead me to the concept that repeated symmetry is a building block of architecture, but also that knowing where to break these symmetries is critical, for example a house should not create an overhang onto a motorway just to be symmetrical. Even the locations where the symmetries are broken will often follow a set repeated pattern. For example this can be observed in addition of satellite dishes to houses, while not every house will have them a sizeable minority will. They will all face in similar directions and be on the same side of people's houses in order to get the

best signal. However properties next to other large buildings that would obstruct the signal, or listed properties where the use of dishes is controlled follow different patterns. This argument follows on from the earlier points about the distribution of uniform appearance of properties. The probability or frequency of these symmetries does vary over a large range, from 'certain' for having a door somewhere in the building to 'a minority' for satellite dish ownership.

A less applied approach to building design is taken by Finkenzeller and Schmitt²¹ who describe a logical way of defining a floor plan so that the individual geometric components. By using a simple algorithm to combine regular polygons into an ordered wall plan, and then basing higher storeys around this plan detailed buildings may be constructed without a formal L system. By retaining the geometric information the underlying geometries can be retained and used later to decide upon façades that are shared between conceptually similar components, such as the same wall on both sides of a window. As well as producing some very realistic floor plans the system is the nicely hierarchical (building, floor-plan, line-strip wall, subwall, door or window, edge refinement, frame). The system also interfaces with Alias' Maya MEL scripting language to allow quick renders of scenes suitable, a feature suitable for their anticipated audience of virtual environment manufacturers.

An even more detailed approach to this problem is taken by Legakis et al. in devising a cellular encoding mechanism for applying details to a wall. By identifying walls and the location of holes in them, it is possible to select areas that are window frames or wall edges. Cells around these components can then be marked and geometry generated for them automatically. Typically these cells are converted into bricks and mortar, to create a tessellated stone wall. An important point raised is the difference between abstract geometry and real geometry. Abstract geometry will typically be converted into polygons to render, but in doing so will lose much information about where edges exist. This lost information is expensive to retrieve from a polygon mesh and is valuable in detecting edges. By storing the abstract representation Legakis' implementation is able to detail with complexities such as curved walls and pillars with ease. It also allows the details of the bricks to be consistent around the edges of walls where standard 2D texture derivations typically fail.

While exploring the possibilities for roof generation I came across a paper from Franz Aurenhammer²² explaining the issues generating a typical hatched roof from an arbitrary floor plan. This problem, finding the 'straight skeleton', is more complex than it initially appears. The straight skeleton is a mathematical structure that is defined by the shrinking of the base polygon. I describe the details later in this work. The best solution found runs sub-quadratic time to produce. While Eppstein and Erickson²³ describe one such fast approach, and describe the result of the process as a tree, the details of the algorithm are best described by Felkel and Stepan²⁴, who go into some detail about the complexity of its construction.

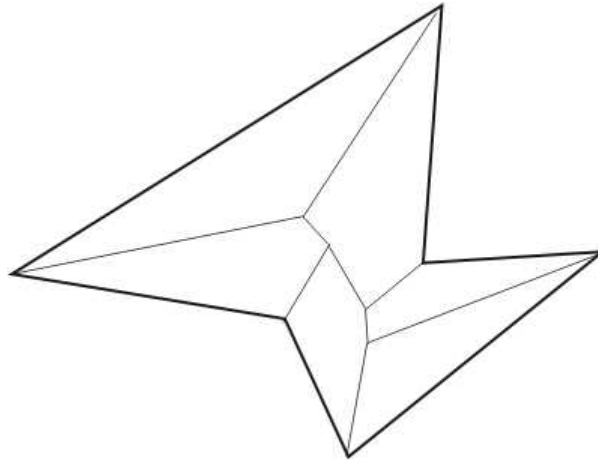


Figure 3 – a straight skeleton (not-bold) of a polygon (bold)

Laycock and Day's paper²⁵ investigates how to use the straight skeleton to produce different types of roof edges, from Mansard to Gable. By adjusting the result of the straight skeleton, after it has run, it is possible to produce a wide range of roof styles. My observation of the skeleton revealed several modifications that might make the skeleton a powerful abstraction tool:

- A more interesting skeleton may be constructed if each edge is assigned different slope. This could be used to create gabled ends of roofs without Laycock and Day's messy alteration after the skeleton algorithm has run.
- The skeleton can be truncated at any height during its construction, leaving a 'bevelled' object, this may be used as a wall if the edges slope is near-vertical or for constructing a window or door frame.
- The slope of a side may be changed at any time to create say a vertical wall, followed by an overhang, followed by a roof.
- It should be possible to construct skeletons from non-planar polygon. For example from a roof line, that has been adjusted due to additional windows.

This modified straight skeleton, or *camp skeleton* (for its similarity to a ridge tent with sides of different slopes) is used constantly in architectural design and seems to be an excellent component of a grammar. It can be used to create both a wall and roof of a house, as well as many other features it forms a useful abstraction.

I discovered another useful algorithm on the data visualisation course at Bristol. The Voronoi²⁶ diagram was introduced as the dual of the Delaunay triangulation for the purpose structuring unstructured data.

As shown in Figure 4, a Voronoi diagram is constructed from a collection of points on a plane, inside a master-polygon. A polygon is drawn around each point, inside the master polygon, so that for all possible locations inside the polygon they are nearer the bounded point than any other. It occurred to me that the map of polygons produced resembled an aerial view of city streets. This had been noted before by Glass, Morkel

and Bangay²⁷ only as recently as January this year. Glass et al. used a regular square grid of points with some quantity of noise added to produce the course grid structure in African informal settlements. The results were impressive, but I was surprised to see them switch to an L-System inspired by Parish and Müller for the more detailed streets. I believe that by combining different types of point generator, such as rectangular, circular and spiral, a very detailed and intricate road map could be created, a simple example with random points is shown in Figure 4. The Voronoi would not have the dead ends and symmetries of the L-system, unless they were added in at a later stage

Figure 4 - Comparison of Voronoi diagram and a Bristol street map

Some useful features of the Voronoi diagram include:

- Complete delegation – all the space defined in the initial polygon is accounted for by one sub-polygon or another
- Concave and convex shapes allowed, while a little more complex to implement the concave version of the algorithm allows intricate shapes to be tessellated
- Simple input – it is relatively simple to generate the input, as only points are required
- It may be possible to weight the Voronoi, to give some points a larger space than others, to allow more variation in the patterns produced.

A Description of Sity's Approach

Sity combines some of the above ideas into a product designed to generate areas of under a kilometer square of a city in a believable manner. As simulating the processes that lead to a city's growth was too complex for the resources I have, the city is constructed using a stochastic L-system.

The Voronoi and camp skeleton algorithms complement each other as production rules in the grammar. The skeleton creates very rigid 3D geometry, with interesting exceptions that appeal to the eye as man made. The 2D Voronoi is able to clean up these anomalies into something that is suitable for additional computation.

For example the Voronoi can create city blocks, while the straight skeleton can shrink these in to create roads in the spaces between; the skeleton can identify the areas of ground that may be a wall and the Voronoi can split the wall up to allow a door in the wall of a specified size.

The camp skeleton can be used to generate both the walls and the rooves of the buildings, based on Voronoi output. The Voronoi input will be dots within the containing polygon or a specified pattern. By using a stochastic L-system additional variety can be introduced to the grammar without letting the grammar become too large and unusable.

The anticipated use of *Sity* in game development:

1. A base grammar is imported
2. Changes to the grammar are edited by the game artists
3. The grammar is evaluated and the output commented upon
4. (optional) improvements to the grammar are applied
5. The grammar is evaluated again and output placed in a standard game environment development tool, such as AutoDesk's Maya²⁹.
6. Additional features that are not available in the grammar are added in
7. The game development workflow continues.

An important business aspect of *Sity* is that it is a program intended for use by the game artists. This is critical as these artists need to take leading roles in content generation. Working by proxy through a programmer is not acceptable, as it will diminish artistic input to the process. Part of *Sity's* appeal is that it allows quicker draft turn around times, and waiting for a programmer to convert concepts into will reduce this benefit.

For these reasons *Sity* will present itself to the user as comprehensive GUI that gives a preview of the city generated, and allows the complex grammar to be built up in a guided manner in an integrated environment.

This manner in which this grammar is presented was designed to be familiar to game artists. Using the mathematical form of the L-system as outlined above would not only be difficult to interpret into three dimensions, but be difficult for someone without a mathematical background to interpret. For these reasons it was decided to have an input system similar to Maya's *Hypergraph* (Figure 5) as this is an often understood concept. Extensive use of graphics, colour and topology are characteristics of this input method. This system provides much more information than an L-system would require, such as multiple input points, so a subset has been chosen for *Sity's* GUI.

Sity also interfaces with *Maya* and a common 3d file format output (Lightwave .obj format), to ensure a game developer's work-flow is as uninterrupted as possible.

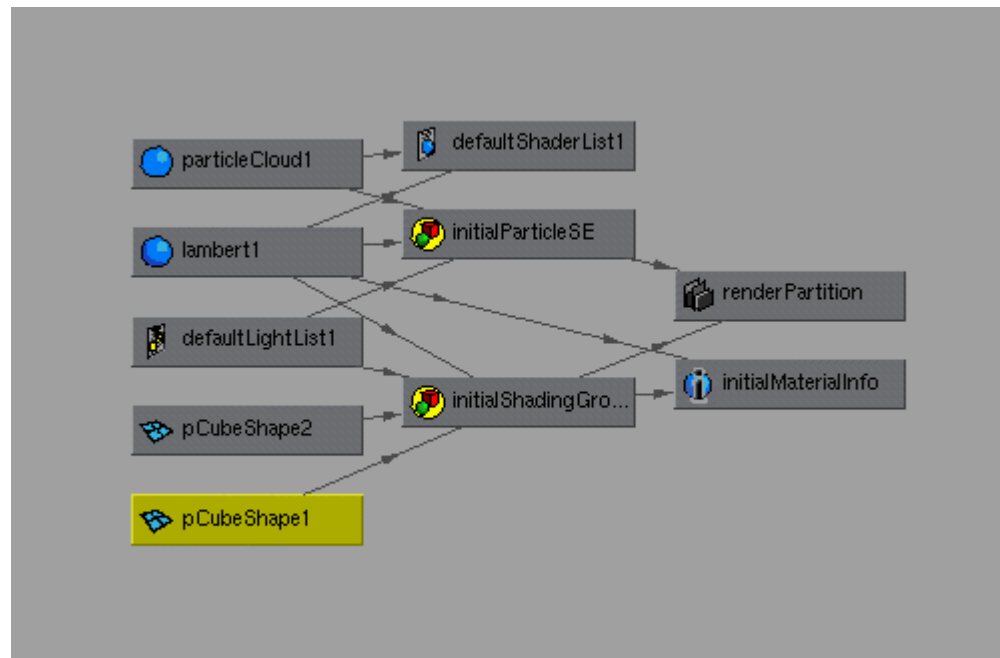


Figure 5 – Autodesk's Maya Hypergraph

Technical Aspects of *Sity*

This section considers the theory used in *Sity*. I start by giving details of the two algorithms introduced above, and move on to showing how they can be combined into an effective grammar.

On Skeletons

The straight skeleton algorithm given by Felkel24 was implemented, revised and updated to be able to generate camp skeletons. Here I describe that algorithm & details of its implementation, and how it grew to create the more interesting geometry given by the camp skeleton. I expected this to be a relatively easy implementation, but should have noted the lack of any documentation on creating a skeletons before I started. There were numerous hold ups and complications that suggests that creating a skeleton of this form as a project unto itself.

A straight skeleton is an operation on a simple polygon. A simple polygon in a mathematical sense is a 2D shape in which no lines cross. The straight skeleton creates the ‘roof’ corners suitable for this polygon. It defines a volume bounded by the input polygon and planes defined by the edges of the polygon that slope inwards. Another way of thinking of it is the result of shrinking in the vertices of the polygon, as shown in Figure 6. Note that the skeleton is a 2D structure, but map be easily converted to a 3D form.

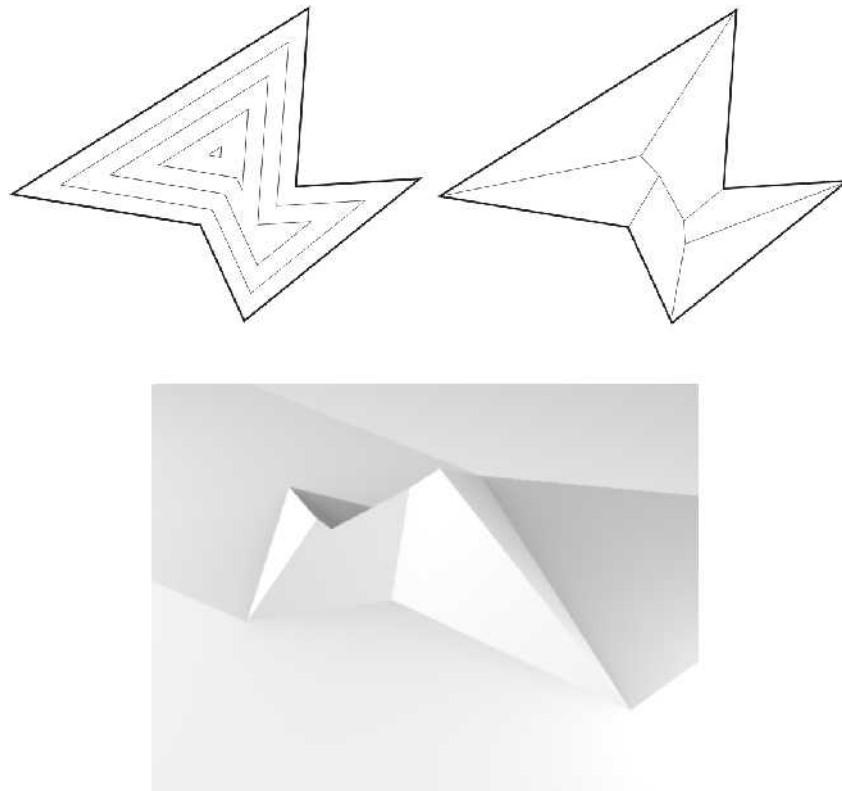


Figure 6 – Top left, the idea of shrinking a polygon(bold) to give the straight skeleton (top right).
Skeleton extrapolated to 3D, bottom.

Here it is interesting to observe that we are constructing a tree. Each vertex generates a bisector (roof edge) and these merge together, removing sides as we progress higher up the skeleton. Branches merge as we come towards the top of the skeleton. While not strictly true when dealing with concave shapes (a twig, may lead to two branches), this hierarchy is an important feature when manipulating the skeleton.

A naïve way to create the skeleton is by shrinking in the vertices several times. This is relatively inefficient because for each iteration shrinking, each vertex must be moved in and then checked against all other lines in the polygon to check it did not cross them. To get accurate results large numbers of iterations would be necessary and hence make the procedure very costly. Felkel's method runs in $O(nm + n \log n)$ time, where n is the total number vertices and m is the number of these that are concave. The modifications for the camp skeleton mean my implementation runs in $O(n^2)$ time.

On Convex Skeletons

Firstly I will describe the algorithm for unweighted convex simple polygons. The method I used, detailed by Felkel relies on the concept of finding collisions between the bisectors of each vertex. Each vertex is stored in a doubly linked list, along with pointers to the previous edges and the next edge, see Figure 7i.

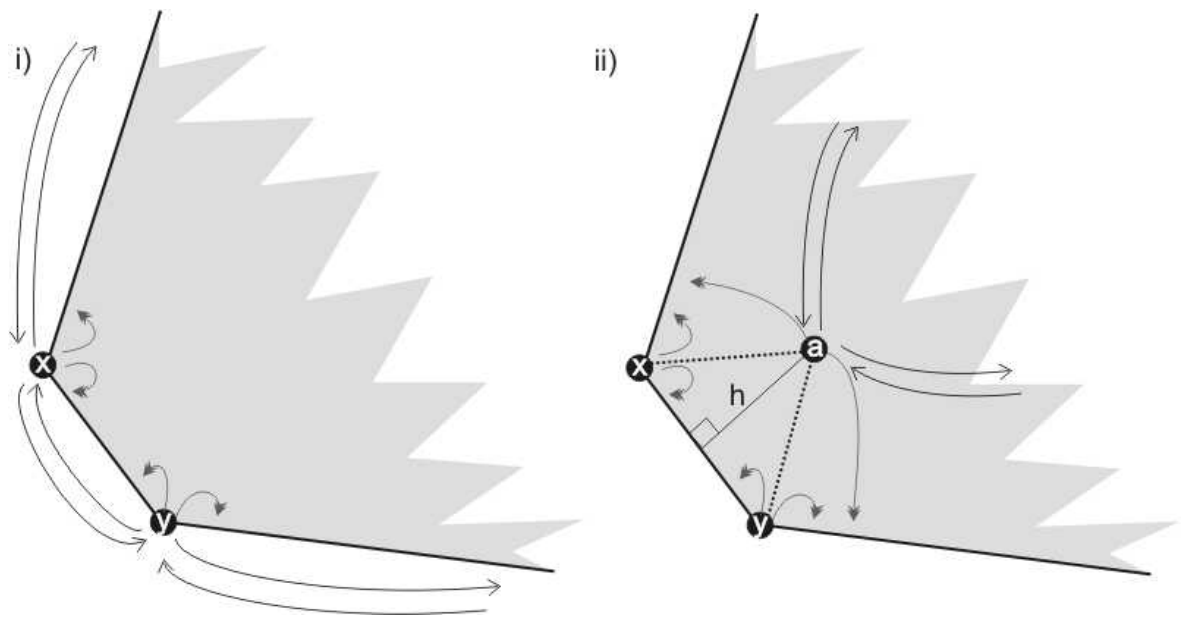


Figure 7 – i) concave pointer layout, ii) changes that result after a split event, edge pointers are double arrows next and previous pointers in the active list are single arrows

At any time this linked list of vertices is the 'active' working set representing the height up the side of the roof that we have reached. All active points are stored in a hash set as well as the doubly linked list.

The bisector direction is defined by next and previous edges. This can be thought of as the midway line between the adjoining lines; a full description is given later. In a concave shape a vertex's bisector may collide with either of the bisectors of the vertexes on either side. This type of simple collision is known as an edge event.

Edge events are calculated in turn for each pair of vertices' bisectors and stored in a priority queue according to the perpendicular distance to that point's edge. This can be seen in Figure 7ii, where the lowest collision is between the bisectors of x and y at point a and height h . When an edge event is processed the two colliding vertices (x & y) are removed from the active list and hash set, and replaced by a single new vertex (a) at the collision point. This new vertex is assigned the first vertex's previous edge and the second vertex's next edge. The new vertex's bisector is set accordingly, and collisions with neighbouring vertices in the active list are found and inserted into the priority queue. Sides are added between the previous vertices and the new collision point (xa & ya , Figure 7 ii, broken lines).

When an edge event is removed from the priority queue it is discarded if either of the two colliding points it refers to is no longer in the active list, having been removed by a collision with the other side.

We rank the edge events by their perpendicular height to either of their adjoining edges. Both these heights as they mark a point of collision between two faces, this is explained later. This distance is proportional to the height in the 3D shape, for non-weighted skeletons.

It is important to note that at this point we have finished dealing with the edge between the first and second points (Figure 7, x and y), it has been enclosed by sides forming a planar surface (x,y,a) and removed from the active list.

We continue in this manner, removing the lowest event from the priority queue, until there are only two vertices left in the active list, then add in the final enclosing side.

It is interesting to note that this stage of Felkel's algorithm is a clever hack. As the two remaining points in the active list point to each other, they also share edges, but with reversed order. Their bisectors therefore point precisely towards each other, and would naturally collide in the final stage, leaving the active list empty. However detecting such a collision in all degenerate cases is very tricky due to the coincident nature of the bisectors; Felkel's method neatly sidesteps this issue.

Felkel suggests keeping lists of which vertices are in which loops using a List of Active Vertices (LAV), stored in a Set of List of Active Vertices (SLAV). I found this method unnecessary and, in places, inaccurate when dealing with camp skeletons. I found it sufficient and simpler to keep a hash set of all vertices, and for each point store a pointer to the next and previous vertices. This improves the robustness for the camp skeleton as this procedure may merge LAVs between different SLAVs. This also makes it much quicker to determine if a collision still has both its colliding points, as it is a constant time look up, rather than a LAV traversal.

On Concave Skeletons

Felkel's next refinement was to show how this scheme could be extended to skeletons with concave bases. This introduces a new type of collision, the split event, which is triggered in the same way as a edge event –

as being the lowest height event in the priority queue of both event types. This occurs when a concave point's bisector impacts against an opposite face, shown in Figure 8. The bottom volume in this Figure shows the state of the active list as the event happens and demonstrates why the list is broken into two. The top solid shows the final state. It can be observed that these two lists then continue to act independently and have no further affect on one another.

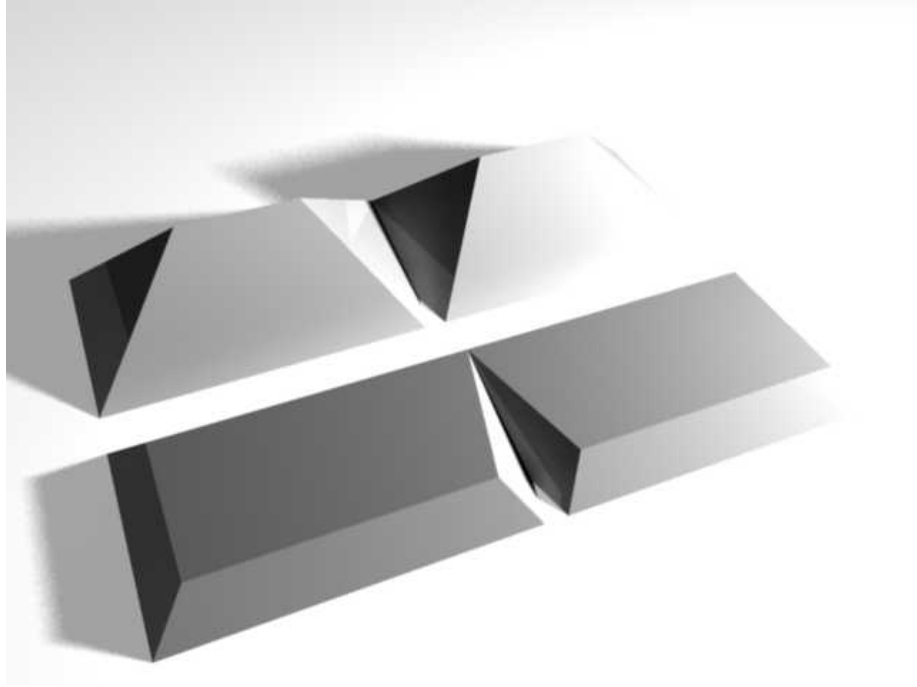


Figure 8 – a split event

This effect is achieved by manipulating the active list as shown in Figure 9.

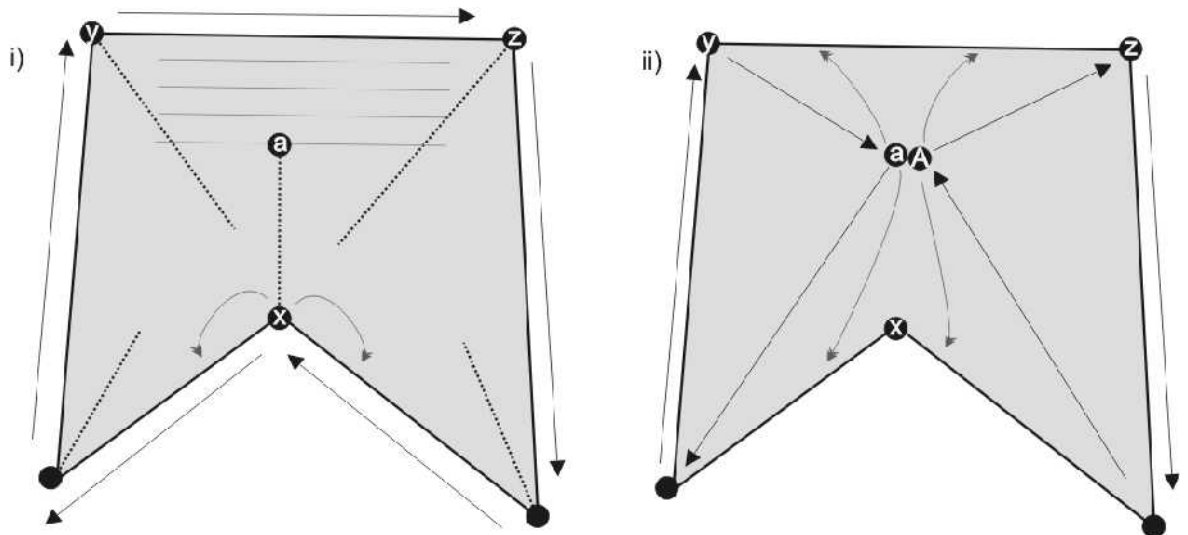


Figure 9 – active list changes on finding a split event of a against edge yx

Figure 9 i) shows a detected split event at a . The bisector of x has intersected with the quad defined by the bisectors of points y and z as well as the line yz . Note that split event can only occur on reflex angles. The second part of the figure, ii), shows the modifications to the active list that take place as part of this event:

- Firstly two new points at the same location are created, a & A .

- Each of these will be linked into one half of the active list. So A's previous vertex is set to x's previous, and A's next becomes the last vertex in the base edge of quad collided against, z.
 - Conversely, a's previous vertex is set to x's next vertex and a's previous vertex is set to the first in the edge of the quad.
 - Vertex A's next edge is set to the bottom edge of the collided quad, and its previous edge set to x's previous edge
 - Vertex a's next edge is set to vertex's x's next edge, and a's previous edge become the base of the quad.
- This is rather a long winded way of describing the events depicted above. This technique proves quite robust and deals correctly with collisions of weighted edges in the camp skeleton between separate polygons for negatively weighted edges.

There are however two interesting caveats to dealing with this split event. When a split event occurs we need to find the opposite edge. This may have changed because of a previous split event. For example after a previous split event of a against edge se in Figure 10, the split event b must be handled correctly.

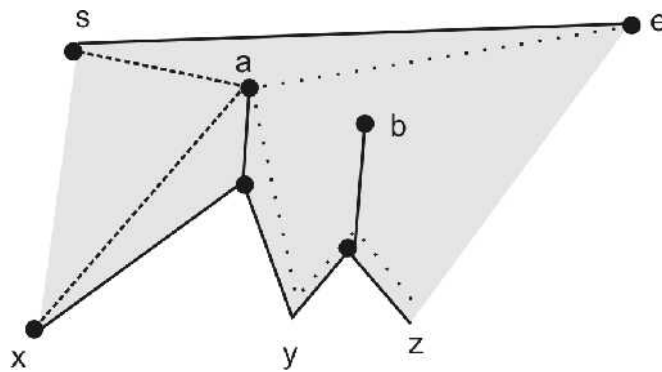


Figure 10 – finding the opposite side in a split event

Because the split event b was found against edge se, we identify (via search) point a as the owner in the dashed set of points of line se. Because edge se may be on another circuit of points (if there is a hole in the shape) this is done via a brute force search of all points, finding ones that have s...e as the next edge. In a camp skeleton implementation Felkel's LAV concept is inappropriate, because we still need to search all the points for possible collisions, as we may be colliding between different LAVs.

The next obstacle is that both s and a have se as their next edge. To resolve which one is correct we can check that whether the collision point b lies inside the bisectors of s..a..x for s or e..a..y for a. In this case a is the correct point on the circuit, and two new circuits are created a..b..y and e..b..z. Some care must be taken because

- if edge se has a negative weighting in the camp skeleton then b will lie outside of the area
- if se has zero weighting it b will lie on s..e, which requires careful handelling to ensure s and a is not chosen

The reason for this choice of edges is illustrated in Figure 11.

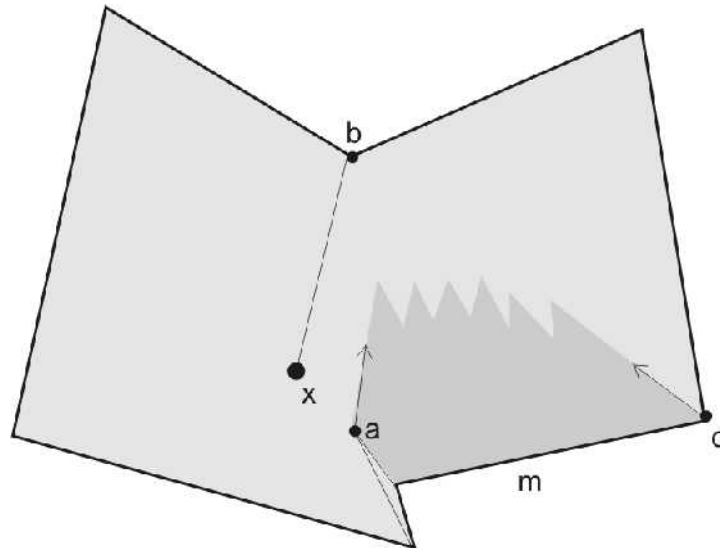


Figure 11 – careful split event clarification is required

Say point b was able to locate a split event against side m at location x. The concave shape allows this to fall within m's defining points' bisectors.

But suppose before that event point a is formed by a edge event with the new bisector being shown by the arrow from a. Allowing c to split against m would be invalid as the edge defined by m is now bounded by a (and guided by its bisector) and the previously found collision point is invalid.

For this reason we must check split events occur inside of the shape defined by the line ac and the bisectors of a and c. Noting that this shape may have infinite area if the bisectors diverge, as they may in a weighted straight skeleton.

One final type of event on concave shapes is the multiple split-event, this event is not described by Felkel, but Eppstein and Erickson²³ discuss it. This occurs when two would be split events happen at the same location, splitting against one of each other's defining planes, for example in as shown in Figure 12.

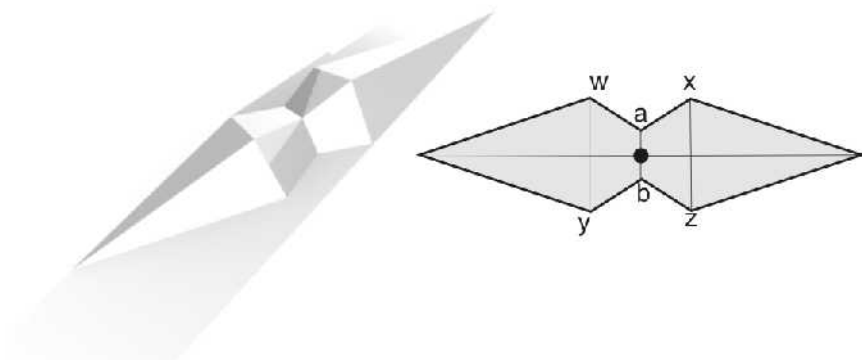


Figure 12 this shape features a double split event

The point marked by a circle is the first collision event to happen, and is a borderline double split event as if a 's bisector were slightly to either side it would become a single split event. This is a typical problematic borderline case. It can be detected here by seeing that the bisector of a performs a split operation with either edge yb or bz , while the bisector of vertex b is in a similar situation with edge wb and ax . Both these split events should be registered at the same location and combined. In this situation it is necessary to split the active list into as many partitions as there are simultaneous split events, here we would (as described above) split the active list into two partitions $[\dots w, \text{collision}, y, \dots]$ and $[\dots z, \text{collision}, \dots, x]$ Unfortunately although this feature was adopted several times in *Sity* it was removed because of some knock on problems it caused.

Finally holes can be defined in the skeleton simply by adding another list of input points, moving in the opposite direction. *Sity* uses the clockwise convention to define a polygon, so holes are defined counter clockwise. Because of this reversed direction the bisectors are treated correctly – a hole that in itself is convex, has all its vertices are treated as concave vertices that generate split events, see Figure 13. Vertex a is treated as reflex angle, and so is able to generate split event x .

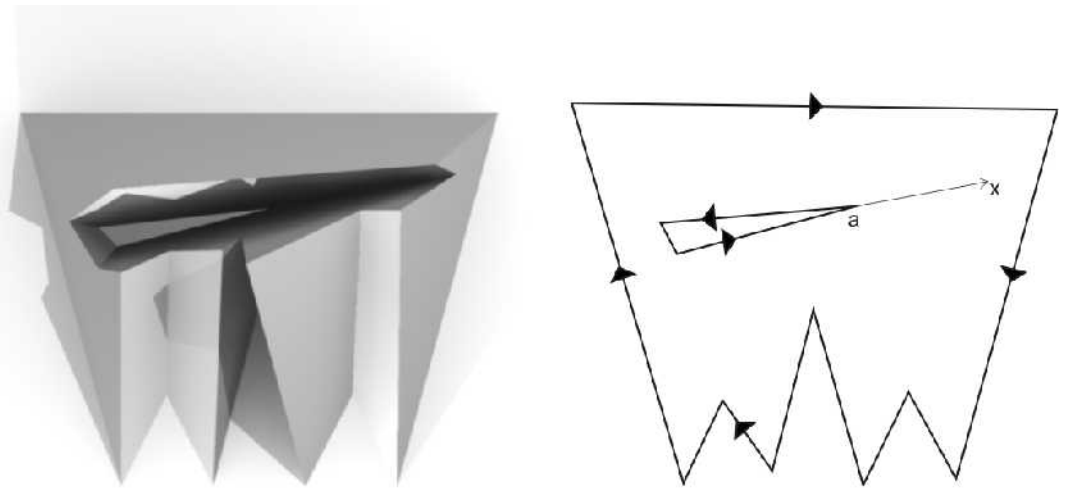


Figure 13 – a hole in weighted straight skeleton

On Camp Skeletons

This section details my alterations to the straight skeleton. My first feature was to allow different sides of the polygon to have different speeds. I found one reference to this structure using positive only weights²², but is not discussed in relation to having Felkel's polygons with holes, or multiple polygons. These different speeds evaluate to different slopes in the finished roof. They work out as gradients, so a speed of 0 indicates a vertical wall, a speed of 1 indicates a 45 degree slope and a flat wall is given by an infinite speed. I thought this feature would be interesting do to the potential of the straight skeleton to easily create roof-eave like structures.

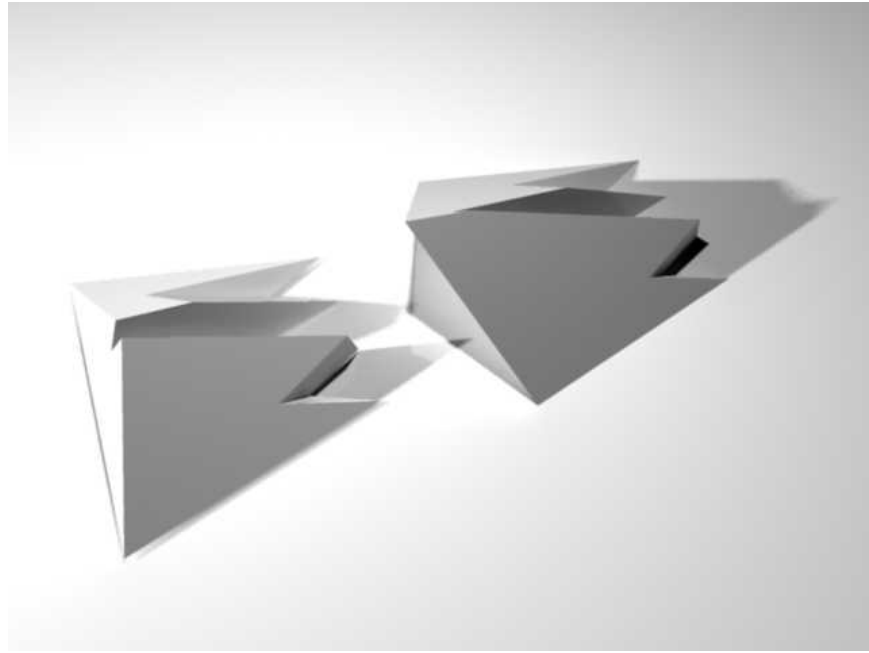


Figure 14 two similar polygons with varying weights, right: the left most face has a negative weight

This extension of the weighted concept to a negative weight was obviously not what Eppstein and Erickson were referring to in their paper²². They claim that convex angles (interior angles of less than 180 degrees) can never cause split events, and hence come to a in $O(nm+n\log n)$ time complexity. Where n is the total number vertices and m is the number of these that are concave. Figure 15 demonstrates one such case in which two concave polygons collide when negative weights are allowed. Note that this is shown as two polygons for the sake of clarity; both polygons could be connected by a thin strip around the sides that wouldn't interfere with the split event.

Table 1 shows which edges now need to be checked for collisions for a given point; on the balance of these my camp skeleton algorithm will run in $O(n^2)$ time. There are optimisations¹ that would return a better for intricate skeletons with several input polygons, but for the usage the skeleton finds in this project they are not necessary to implement.

¹ Based on a tree specifying whether one loop lies inside or outside another. This may will give complexities in the order of $O(f(n,m)+n\log n)$.

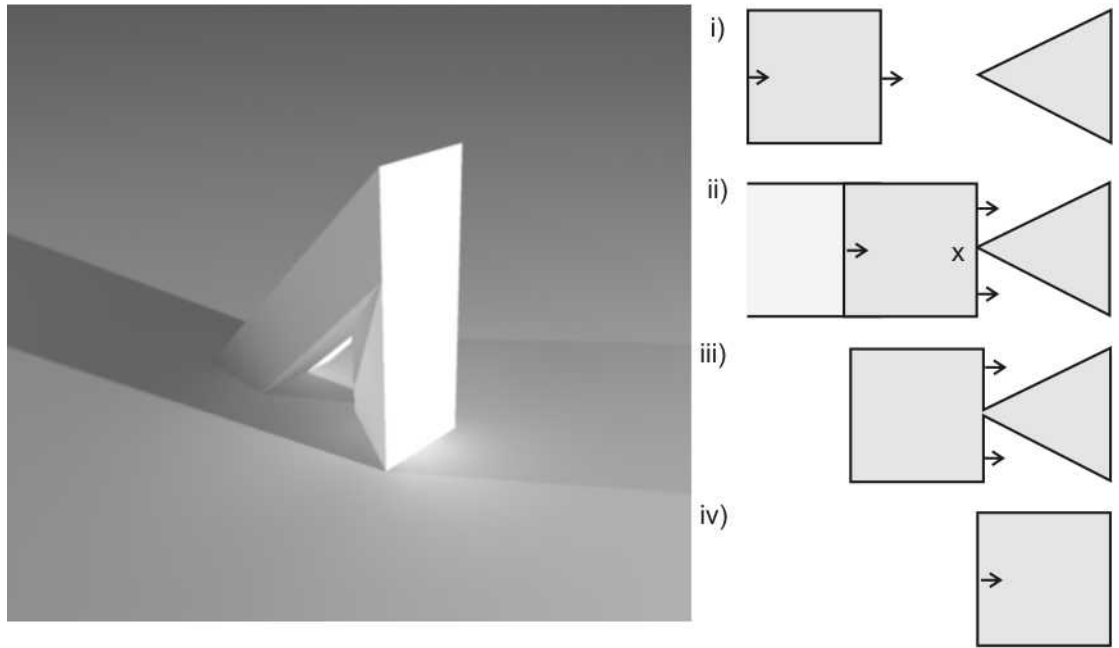


Figure 15 – a split event on a convex angle

		interior angle	
		convex	concave
compare side speeds	2 positive speeds	check collisions with adjoining sides	check collisions against edges in same loop
	1 positive speeds		
	0 positive speeds	check against all other edges	

Table 1 – The required checks when finding collision on camp skeleton bisectors. Grey area shows interior bisectors, those considered by previous authors.

To evaluate the weights correctly we need to introduce a length value to our directional bisector, the length of the bisector represents the height of an adjoining wall of gradient one after a travelling one unit of distance.

The calculation of the bisector is shown in Figure 16. Note that this is easily adapted to the case where bisectors do not meet by moving the one of the bisectors to the location of the other.

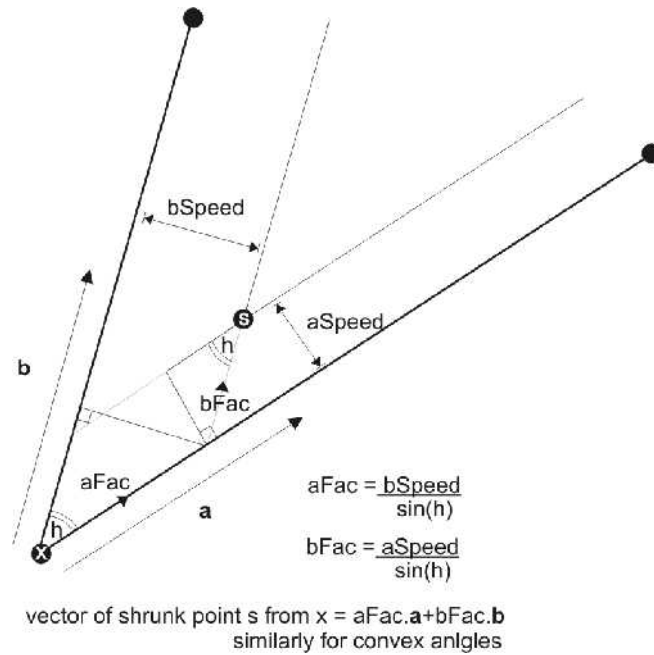


Figure 16 – Calculation of bisector vector

To calculate the intersection of this bisection with opposite quads to calculate split events I use the Jordan curve algorithm modified to allow borderline intersections, the details of this algorithm are given in Appendix 3. It is interesting to note that depending on the weights these quads may not be finite, and need rounding to finite values to perform sensible calculations on them. The modifications to allow borderline cases are important when dealing with regular shapes as a bisector must not ‘escape’ between two supposedly touching planes.

Care must be taken when dealing with straight angles (180 degrees). If the edges are weighted this produces geometric impossibility in calculating the bisectors. The weighted surfaces do not intersect to form a line. A solution is found in the limiting case, when the angle is very small we are given a very fast bisector going in the direction of one of the adjoining corners. A satisfactory conclusion is therefore found by removing straight angles from the input polygon before calculation, as this emulates the instant merger of the bisector with the next or previous point (which one has no effect). However this will lead to a sudden snapping between one edge speed and the other when angles are close to 180 degrees.

A further simple modification to the skeleton procedure is an optional bevel command. After reaching any specified height it is possible to examine the active list, and modify it to create a ‘flat-top’ to the shape’.



Figure 17 – the shape in Figure 6, with a bevelled top

There are two steps to achieving this bevelling of the skeleton during its construction. We specify a height at which to bevel and before the next event higher than this height we stop the operation and perform the following two tasks on the skeleton:

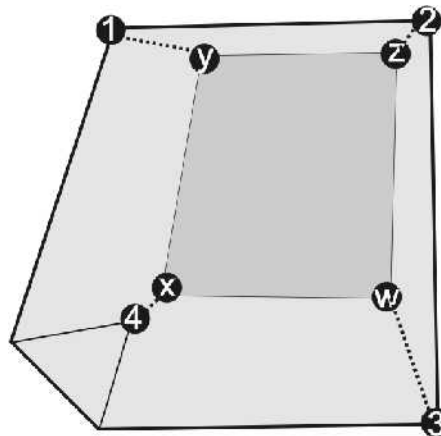


Figure 18 – bevel procedure

1. Sides (Figure 18, bold dashed lines) are added for all points in the active list (1,2,3 & 4) in the direction of that point's current bisection (3-w, z-2, y-1 & x-4). The length of the line is such that it ends at the specified bevelled height, given the direction and steepness of the points bisector. All previous points (1..4) are then removed from the active list and each is replaced by the point at the end of the side it has just created (w..z).
2. All points in the active list are now the 'flat top' (Figure 18, dark grey) of the bevel. Each pair of points (wx, xy, yz & zw) are added as a side to the skeleton and added to a list of points that specify the flat top of the skeleton.

Note that we do not bevel at all if the skeleton has no points in the active list at the bevel height requested, as this would be above the top of the skeleton.

At the start of the above description of the straight skeleton I gave an unconvincing complexity argument for not using the edge shrinking algorithm, the real reason is that if we had shrunk the vertices it would also be impossible to bevel a skeleton midway through, without storing the state after the last collision before evaluating the next collision. If we were creating vertical walls with zero slope there would be no such collision (or it would be infinite) and would take many iterations to find. This is the main reason the iterative approach was not used.

I have not yet discussed what constitutes a ‘side’ when adding them into the skeleton as the algorithm progresses. Felkel generates panels, or roof surfaces by treating the skeleton as a topology and traversing the network; starting on each of the bisectors of the original shape and always choosing next edge with the smallest the inside angle. This is sufficient for straight skeletons, but for camp skeletons a more devious scheme is required, as the panels may move outside the original shape and so have an anti-clockwise projected direction. I considered using the same scheme in 3d, finding three points and then projecting the other points onto that surface to determine the minimum inside angle; this seems unnecessarily contorted, involving multiple 3d operations. My solution, illustrated in Figure 19, is to give each edge of the original polygon a set of points, stored as a hash set. Figure 19 i, shows the points in the sets for input polygon (bold) side a (as solid points) and b (as hollow points). Each time a point is removed from the active list it is added to both the sets of both edges it has pointers to. The sides formed by the bevelled top are only put into the one set of the input polygon edge they are associated with.

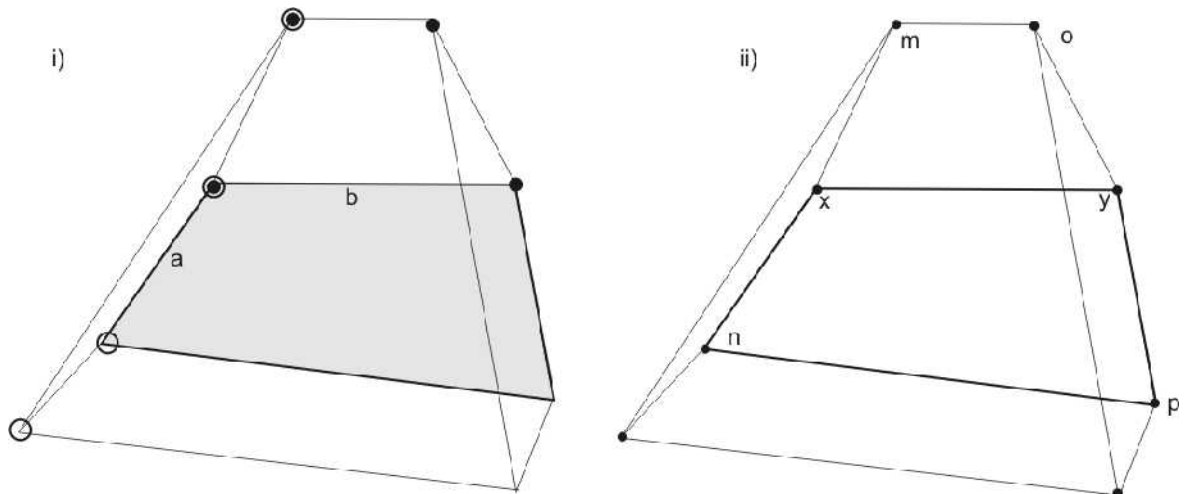


Figure 19 – point traversal scheme

As well as these point sets I build a network topology as shown in Figure 19 ii. This is implemented as a hash map of points indexing a list of points, giving the possible destinations from any one point. For example under the key of point x, the hash map would store the value a list of m,n & y, and under the point y it would contain the list o,x&p. All points are entered in both directions to create the map, as the points are provided in a fairly arbitrary order, especially if several split events have occurred during the skeleton’s construction.

To produce the planar polygon sides of the skeleton we start from the first point in every input edge, for example point y in Figure 19, and choose the point that is listed in the hash map’s entry for y, that is also in

the relevant edge's (b) hash set and wasn't the previous point. This continues through the points y,o,m,x (note anti-clockwise order due to upside down polygon) until we find the start point again.

When converting the 2D topology to a 3D roof structure the heights of each point are taken to be some fixed factor multiplied by the height that the collision that created the corner occurred at.

Voronoi Diagram

The next algorithm implemented was a Voronoi tessellation routine. This is a very robust structure for the partitioning of an area given a set of point inside that area, see Figure 4. For each area or *cell* in the structure any location is closer to the point that specifies the cell than any other point. For concave shapes the definition of *closest to* is minimum distance that does not cross the boundary of the shape.

One of the fastest methods to generate a Voronoi diagram is using Fortune's method³⁰. After my extended experiences with the camp skeleton I decided to go for a higher time complexity, simpler to implement, algorithm detailed from a website³¹. The outline was taken from here and the subsequent details worked out. Each input point is added in turn, with an area or cell of its own. The algorithm cuts space for the new point from the existing points until all points have been added.

The initial boundary and a starting point are passed to the algorithm first. The entirety of the boundary is associated with this first point forming a cell, as well as being marked as an edge. For each additional point we then find the bisector (a different meaning to the word bisector than used in describing the skeleton!) line that defines which locations that are closer to one point than the other. This line runs through a point halfway between the input points and runs at 90 degrees to them. This line is followed until either reaches another cell or the boundary. If it reaches the boundary it follows it in a clockwise direction until it reaches a new cell or the start. If the bisector ends in a new cell that cell is split.

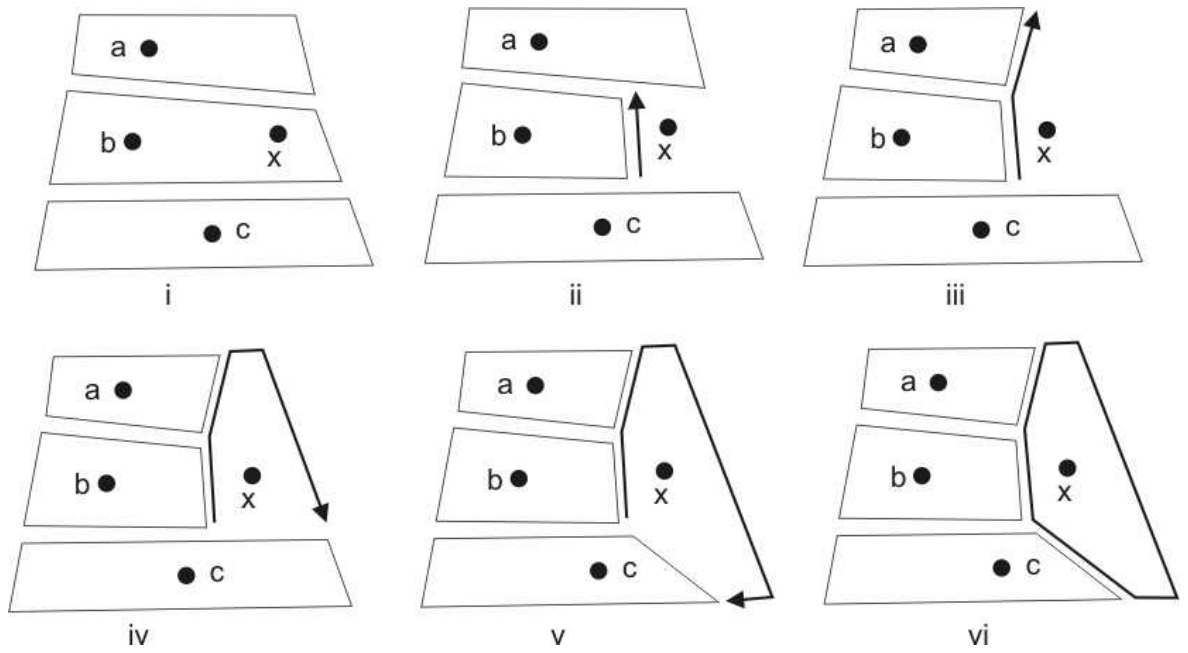


Figure 20 – Voronoi calculation

This is shown in Figure 20, where we are inserting x into a set of cells (abc) that have already been calculated. Initially x is in b's cell so b's size is truncated (Figure 20 ii), at the end of the bisector we find a, so a is the next to be truncated (iii). We then follow the edge of the shape(iv) until will find a new cell c. c's bisector is added, and we return to following the edge(v) until we are back at the start (vi). The path traced forms the cell around x. The important fact to note here is that we need to find the bisector of c and then traverse the edge until back at the start. To explore this issue examine Figure 21.

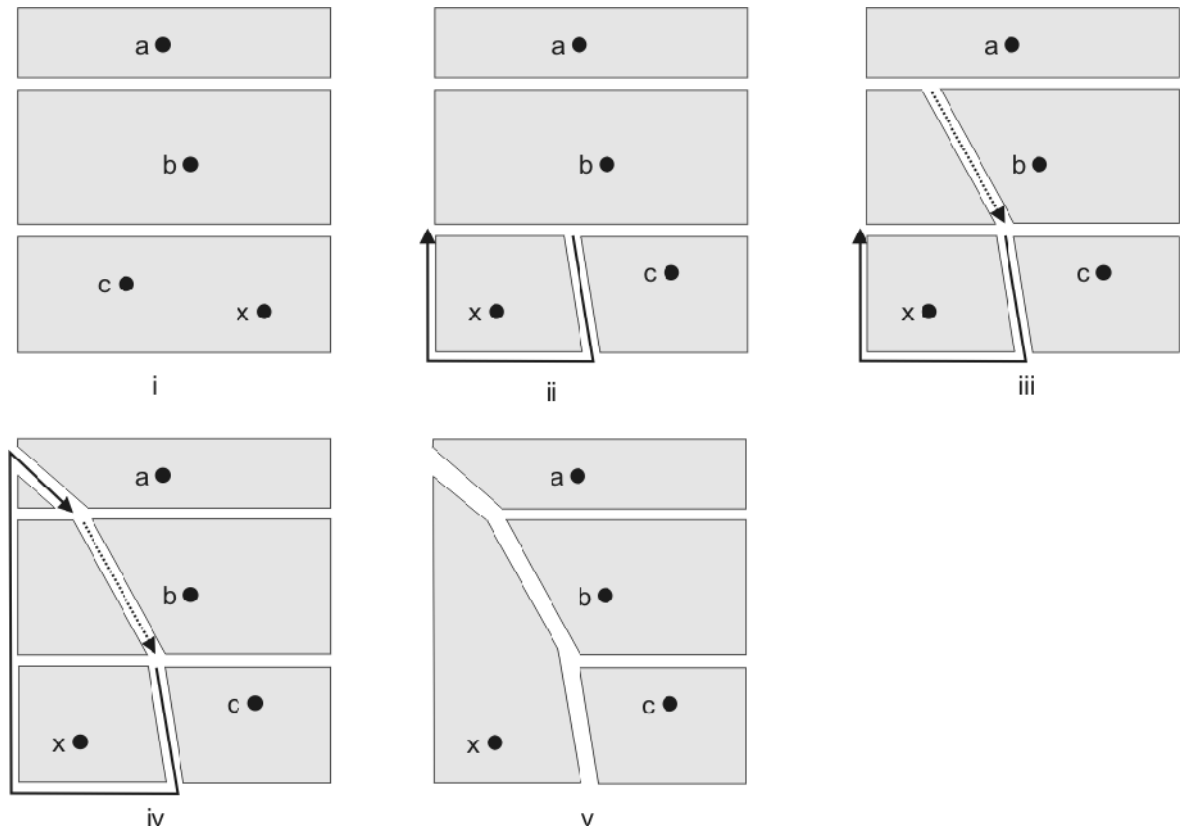


Figure 21 – the need to remember where we should go.

After encountering cell b in Figure 21 ii we need to store the bisector until we get back there (iv). In a similar method to the generation of the edge polygons of the straight skeleton we must therefore keep a two sets of edge hash maps. One defines the boundary in a clockwise direction, and the other specifies bisectors in both directions and subsequent shapes. In the above example not only do we need to know where the bisector of b is, we may also need to know that after it we move into shape c.

A consequence of using hash tables in this manner is that we have to be careful to ensure that the bisectors end at the same location as the next section starts. In Figure 21 we see that that start of the x...c bisector is in the same position as the end of x..b's. As all the calculations take place in double precision and the bisectors are created using the parametric form of a line this is almost certainly not the case. Close points are therefore merged into single points, with a small tolerance. Care has to be taken in degenerate cases where other points are this close, for example when tessellating regularly positioned points.

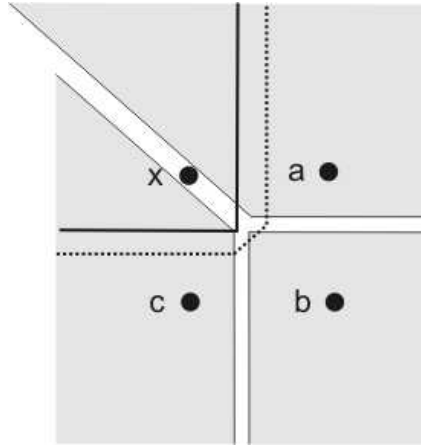


Figure 22 – corner disambiguation

Border line cases occur with regularly spaced points (such as occur when laying out street blocks). The path of the section is effectively non-deterministic due to the floating point arithmetic and rounding errors involved. In Figure 22, inserting x into abc, on a grid layout, the route to be followed may be given by the dashed or solid lines, depending on rounding errors.

Using the merging technique described above we already ‘snap’ close intersections to the start or end of their edges. Following one of these snaps, it is then necessary to choose the right most edge to follow to the next shape, as we are traversing in a clockwise direction

More formally the algorithm is, for each point to be added:

- Find the cell that the new point falls into
- Find the bisector of the two points
- Current point is set to the start of the bisector, while not back at the start:
 1. Split the cell by the bisector
 2. Add any split outside edges back into the hash of outside edges
 3. Add the split into the hash of bisectors and following cells
 4. From the start point trace the outside of the shape by following the bisectors first, and then outside edges using the hash maps.

The calculation of the bisector is detailed in Appendix 5.

The next challenge I faced was to create Voronoi tessellations from concave input polygons. This finds use when trying to break up a concave polygon, such as a wall, in order to create a door. It complicates the calculation of the bisector as shown in Figure 23.

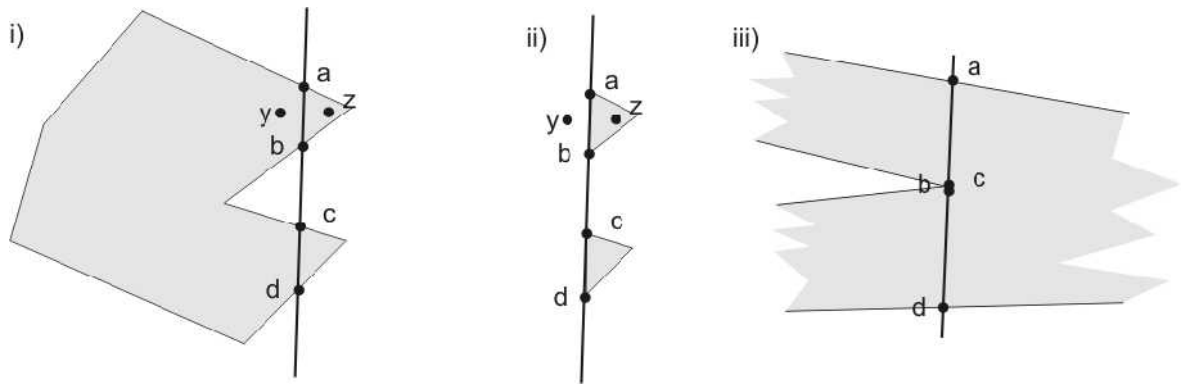


Figure 23 – finding the bisector in convex shapes

When more than one pair of intersections take place we use the point that defines the bisector to determine which bisector is correct. In Figure 23i) a..d is the bisector of points z and y. The routine described in Appendix 5 will return the parametric points a,b,c,d and we must determine that the correct bisector is ab. To do this we simply decompose the polygon according to the bisector (Figure 23 ii), then use a Jordan curve algorithm (Appendix 3) to find which of the sections the new point falls into.

Inevitably we have to deal with degenerate cases. In the situation given in Figure 23 iii) we will receive the intersections a and d. Depending on rounding errors we may detect none, one, or more of the intersections b and c. A similar case occurs when the bisector starts or ends at the corner of a shape - we will get two entry or exit parameters. To overcome this problem we order the points by the parametric term in their formulae, and using the fact that they are entry or exit points. Entry or exit point classification is based on the cross product of the bisector and the direction of the line defining the polygon, then:

- a series of min values becomes the lowest (parameter) min value
- a series of max values becomes the highest max value

This is proof against the previous case. However I recognise that degenerate polygons could be constructed where line segments could be missed if point entry/exit locations are reported as just a exit point or vice versa. Mostly this algorithm relies on receiving both events b and c in Figure 23 iii).

Apart from these changes, the algorithm is identical to the concave and can work even when the input points are at the vertices of the input polygon (Figure 24).

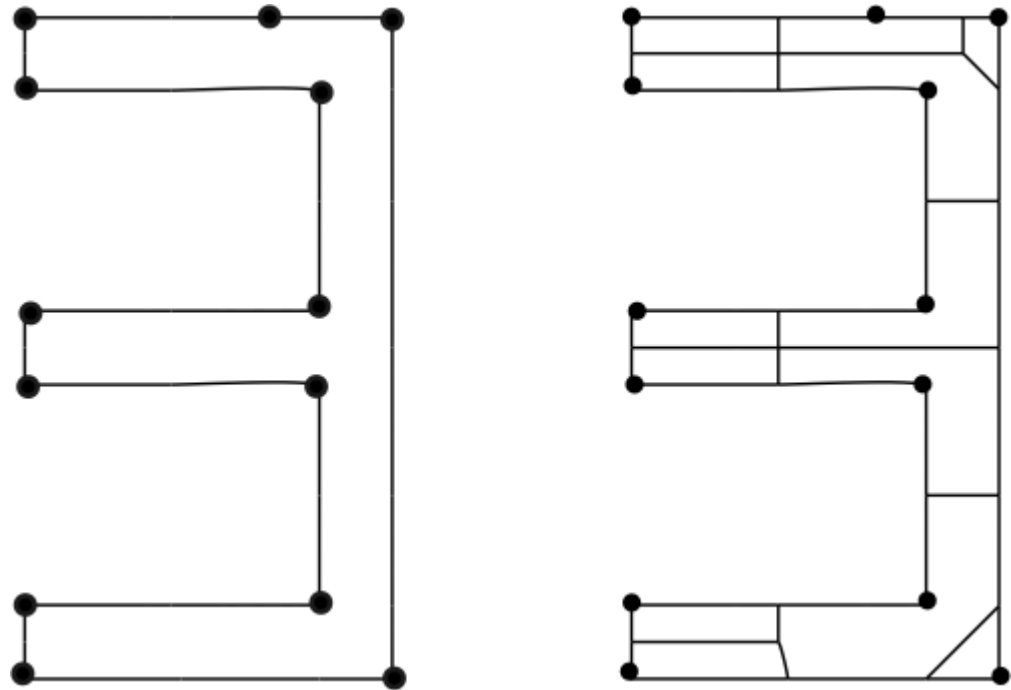


Figure 24 Concave voronoi tessellation; left: input points and boundary, right output geometry

Sity Grammar Overview

In this section I give an overview of how the Voronoi, the camp skeleton and some other algorithms can be used to produce the geometry of a city. For a more detailed guide to the grammar refer to the user guide, Appendix 2 (page 55).

To overview to the process –

- A set of input points define the centres of city blocks within the city; these are shrunk using the skeleton in to leave room for the roads.
- A set of points within the block are used by the Voronoi create the individual plots for buildings.
- The plot is shrunk to a house using the camp skeleton and some of the remaining space may be used for a wall.
- The house is grown upwards and then inwards as a camp skeleton with changing edge weights.

The shrinking process using the skeleton is the result of taking the input polygon, assigning weights to its edges and generating the skeleton bevelled after a certain distance. The flat top of this bevel is then projected back onto the original polygon, giving a ‘shrunk’ version of itself.

Many of the parameters for this process are specified by a set of Gaussian distributed values, which are guided by a large set of input variables at each level of the grammar. As this is a stochastic L system there can be a multitude of branches defining different sets of variables, leading to a large variety of output shapes.

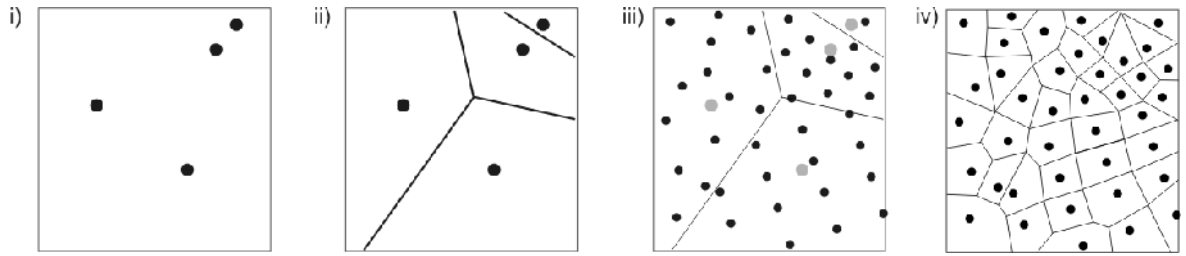


Figure 25 – block generation

Figure 25 illustrates the process of generating blocks. An input rectangle defines the outer bounds of our city. The size of this is not stochastically generated, and like all sizes here is measured in meters. Typical output sizes are 200m square up to 1Km square.

I divide the city into neighbourhoods. Each of these will have a unique layout and type of building to simulate a city built up over the ages. We do this using a ‘dot generator’ as in Figure 25 i, this creates a predetermined number of points inside the city limits. The Voronoi then operates on this to create the neighbourhoods (ii). Each of these neighbourhoods selects a dot generator from one of the following

- Grid based, specifying width and height. The angle is determined randomly.
- Circular, specifying radius interval and width of block. This uses random staggering on each concentric circle to ensure that there isn’t a uncommon repeating pattern or moiré effect in the choices. The centre of the pattern is chosen inside the squared bounds of the neighbourhood.
- Random, specifying only the number of points.

All of the above have a scatter parameter to add noise to the points. Each neighbourhood then returns a set of points within its bounds, and these are combined into one large list of points (iii). The Voronoi then operates on these to create a road layout. At this point there may be some small cells that are unsuitable for building a block upon. These are merged (using a modified version of the union algorithm specified in Appendix 8) into neighbouring cells, until there are no blocks below a certain area. At this point there is an option to randomly merge blocks into their neighbours to create a more disrupted city network.

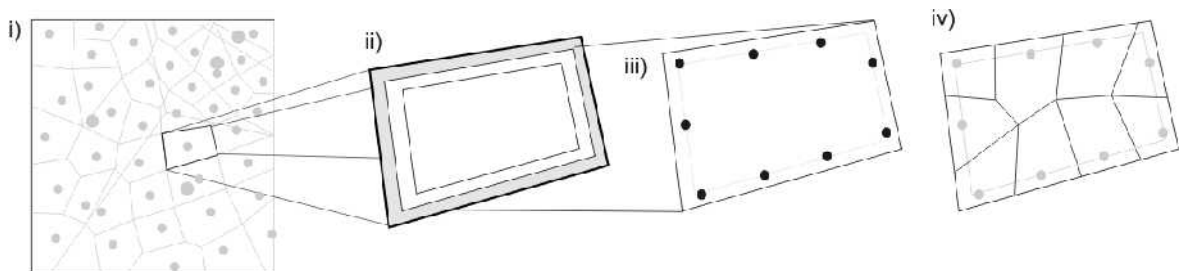


Figure 26 – block to plot generation

Figure 26 describes how a block is firstly shrunk in by a stochastic quantity to create a ‘road’ around itself (ii, shaded). Then this is shrunk in again to create a guideline for the plot point generators. A desired quantity of street frontage is specified by each block, the circumference of guideline is then split as closely as possible to

give this quantity of frontage. After this some optional noise to the frontage is added to give more variation. As shown in iv, the Voronoi operates on these output points and the block input shape to create house plots.

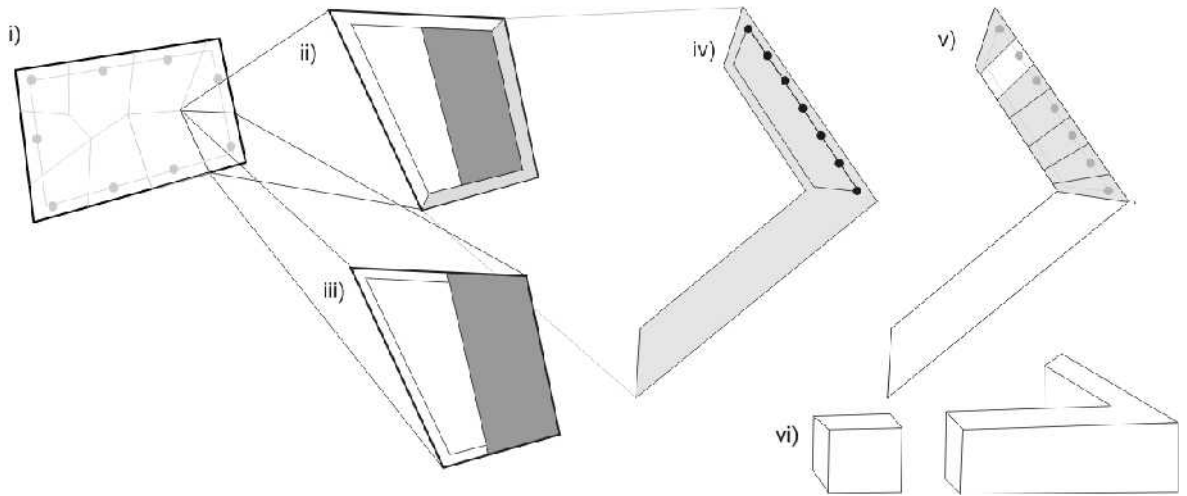


Figure 27 – shrinking the boundaries to create a wall

As Figure 26 shows, in order for there to be a fence around the house there are two options, if there is no space at the side of the house (iii), we skip straight to generating the building floor plan and add on a back garden later. Otherwise we shrink in the sides (ii) of the plot to create front and back walls, and create the floor plan from this point.

To robustly generate a fairly rectangular floor plan from an input polygon (Figure 27ii, iii dark shading) I devised a method of using one of the side output polygons of a bevelled camp skeleton. This was achieved by giving all the sides have a weight of zero except the front side. This was given a weight proportional to the desired depth of the house. This polygon has the property that the base and top lines are parallel, so creates an excellent front and back to a house.

Adding a fence at the front of a house looked strange unless there was a gap in it. For this reason the front fence is split up, using similar method described above for the city block division technique- by placing points for a Voronoi along a bevelled straight skeleton. One of these cells is then removed to form a hole in the wall. This is shown in Figure 27 iv. These are then merged together(v, vi) using a union algorithm (Appendix 8) to form the entire front wall.

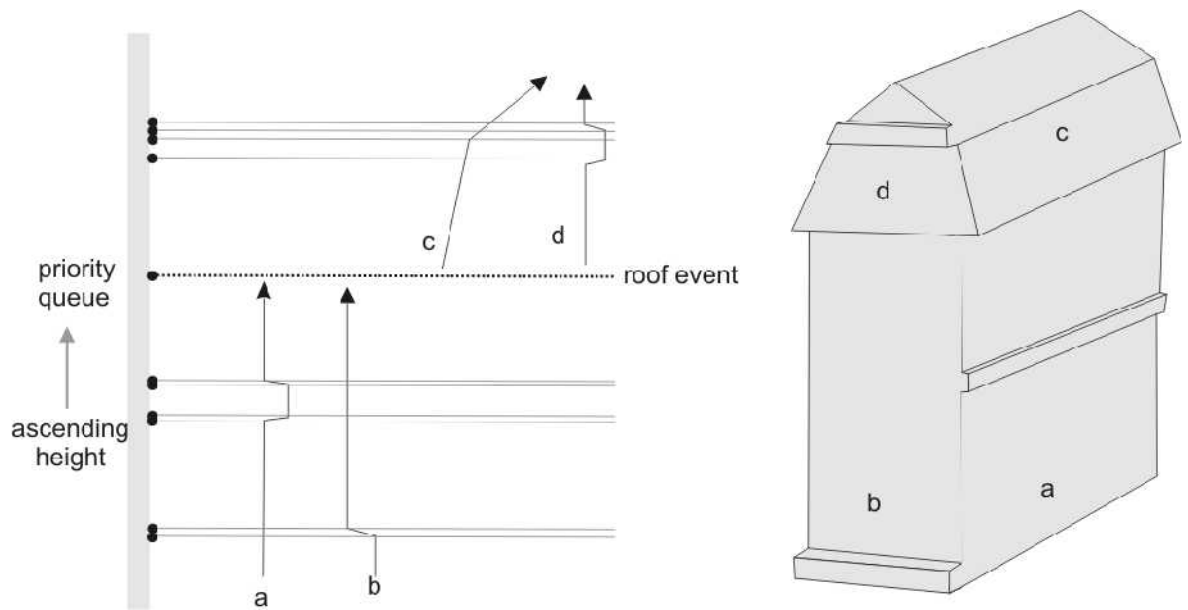


Figure 28 – generating walls and a roof using the camp skeleton

The walls and the roof of the houses are generated via repeated application of the camp skeleton (Figure 28). Each wall's appearance is specified by a decorator (abc&d), which specifies the heights at which the weight of the wall should change. These events are stored in a priority queue by height, with the lowest event or events happening first. The polygon is then input to the camp skeleton, and the lowest weight change event is applied. The skeleton is then constructed and bevelled at the height of the next event. At this next level the weights are changed again, and so on until all events are processed. After the last event is processed the skeleton is left to grow indefinitely, so care has to be made certain that the roof is finite and closes in around the building.

Roof events also exist in this queue. When a roof event occurs, all the following events are removed and new roof decorators (Figure 28: c,d) are assigned to each edge of the roof. An optional overhang before the roof is created from a bevelled camp skeleton and then scaled horizontally by a negative value.

Design and Implementation

Tool chain choices

Sity was written in Java 1.5, mostly as it is the language that I have the most experience using. It also allows me to work seamlessly on the departments Linux computers, my Windows desktop and my Apple laptop. This will also be of benefit to my customers as they will not be constrained to the platform used, it was also the easiest way to match the many different platforms that *Maya* is available for. Java also makes it trivial to inspect objects or classes at runtime, a feature that has allowed me automate GUI creation. The existence of several solid cross platform graphics library for the language made Java my final choice.

Serious thought was given to two other programming languages apart from Java:

- C++, by using C++ and Maya's native API it would have been possible to fully integrate a solution into Maya. C++ has many large and de facto graphics libraries. After careful consideration this was abandoned as due to an obscure technical reason – Maya running on a OS X (the Macintosh operating system) is unable to un-load plug ins. This made developing my laptop an unfeasible choice.
- OCaml is ML based language; I was impressed by its ease of use and efficient native code compilation. The Meta-OCaml implementation also provides very good reflection like code, only with fewer restrictions than Java. OCaml was abandoned because I had little experience in using it, knew no one who had completed a similar project in the language, and the GUI libraries were still being finalised.

I used the Eclipse IDE for similar cross-platform reasons. I had been introduced to it a few weeks previously, and being impressed as to how consistently it presented itself across a range of platforms. Eclipse includes impressive tools for automatic code generation (such as creating getters setters and constructs from a set of fields) and many refactoring options that helped keep the development process fluid.

Sity displays the grammar/'waterfall' construction window in 3D as well as a 3D preview of the city. To generate these 3D sections there were various choices described in the table below. Some of these made extensive use of the scenegraph abstraction. This is a 3D display interface that provides an API that simply allows adding objects and lights to a scene. This is in contrast to a low level approach where individual polygons and transform matrices for each object and movement must be specified.

Option	Pros	Cons
JOGL ³² – Java OpenGL Bindings	<ul style="list-style-type: none"> • Fast – direct bindings to native OpenGL • Complete control • Active community 	<ul style="list-style-type: none"> • Have to write a lot of custom code as nothing is provided • I would have to refresh myself on the intricacies of OpenGL
Java3D ³³ – Sun's now open source scenegraph based engine for Java	<ul style="list-style-type: none"> • Simpler to use than JOGL • Developed in Java's spirit of complete compatibility with all platforms 	<ul style="list-style-type: none"> • Not optimised for games • A lot of unnecessary calls to the API, for example to set permission that can now be automated. • Slower than any of the other options due to compatibility issues. • Inactive community
JMonkey – Another scenegraph based engine for java	<ul style="list-style-type: none"> • Really simple to use • Fast game development time (e.g. key bindings and first-person camera already set up) • Better frame rates than Java3d • Active community 	<ul style="list-style-type: none"> • Still under heavy development • Large dependency tree – LWJGL³⁴ and JOGL are both required
Custom scenegraph	<ul style="list-style-type: none"> • Complete control • Easy to use once it is created? 	<ul style="list-style-type: none"> • Many hours of additional work and debugging required

Table 2

On the balance of the evidence I opted for the JMonkey engine because it was the easiest tool for me to pick up and use out of the box, and was optimised (at development and runtime) for game creation. The developers seem active, and although the engine undergoes constant improvements, several commercial games have been created using the engine, so the API is fairly stable. The dependence on other libraries – LWJGL and JOGL was a concern, but both these projects are again stable and in active commercial use.

To show how *Sity* would work in the real world it was important to show the results in an industry standard package. Because Autodesk's *Maya* was the tool in use in the department, it was the only such tool that I had access too. The open source 3D editor *Blender*³⁵ was a contender, but not having been used in the commercial games arena, wouldn't present the functionality that would make *Sity* attractive to customers. *Maya* also has a well developed scripting language, MEL (Maya Embedded Language) that is able to communicate other programs over a standard port.

The original plan was to use MEL to output the generated data *Maya*. However as the complexity of the generated cities increased through the project the performance of MEL was drawn into question. An alternative of exporting the data to the Wavefront .obj format was tested and proved very successful. However the MEL interface, through its robustness, has saved much time by allowing graphical debugging of complex algorithms. By outputting the location of points and pointers over time the MEL output has highlighted many bugs faster a standard debugger. During algorithms such as the Voronoi it was able to create an animation of the executing algorithm, making it very easy to see potential problems.

A CVS stored changes to the source code and documentation at a different site to the one I was working at. I tried to make it standard practise to submit the system to CVS at the end of every day and tag the repository every time a major feature was finished.

A blog³⁶ was also kept of development progress. This became a very useful tool in assessing my rate of progress, in order to predict which features would be finished inside the allocated time frame. It also contains an interesting view into the way projects evolve, initially there were many grand schemes that became more realistic as time moved on. A minority of features were completed and then left alone for a long time, but the majority of sections of the project were revisited over time to correct or apply bug fixes too.

Engineering Overview

This section gives a brief overview of the system layout. The following chapter reviews some of the more interesting design details. *Sity* uses the Java concept of packages or modules to partition code, as shown in Figure 29, with each package representing a specific portion of the system.

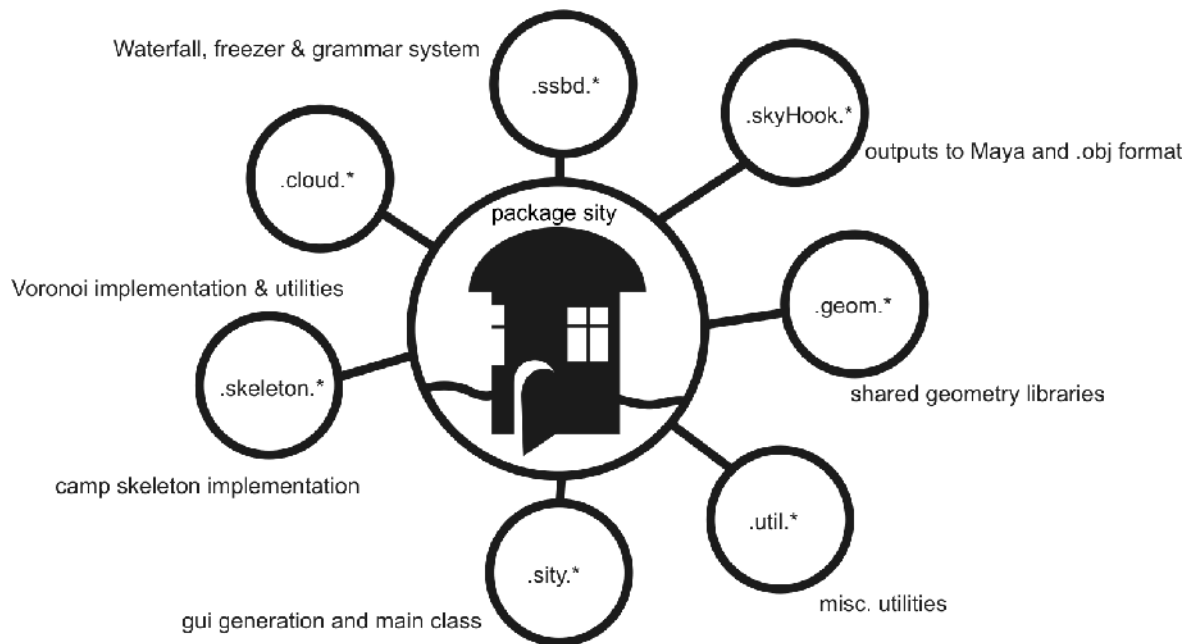


Figure 29 – Sity Package level overview

Overall I have tried to take many suggestions from the extreme programming paradigm, including:

- Extensive test libraries have been developed, over 4,000 lines of test code have been written
- The tests document how the libraries I have written work. This proved especially valuable if coming back to work on code that was written a month ago.
- Keeping a working product to hand, and using many incremental steps in its development to spot problems early. This also made it apparent that progress was being made when the work got bogged down.
- I limited my working week to 40 hours, from experience I knew that I was useless at anything other than simple repetitive tasks after 6 hours of programming.
- Use of metaphor.

Waterfall System

The waterfall and freezer system is a metaphor for the grammar construction mechanism. This system had several tasks assigned to it:

- To allow the expression of a grammar
- To allow the user to edit the grammar
- To evaluate the grammar
- Allow grammars to be saved and recovered from disk

To meet all these requirements the system had to be quite complex. After several rounds of design I came up with two distinct concepts –

- waterfalls, grammar building blocks that the user creates, manipulates and saves/loads to disk
- freezers, ‘frozen waterfalls’, the objects that are created to do the work as the grammar is evaluated.

This separation of definition and operation was necessary as there was a gulf between the things you could do with either. For example fields in a waterfall class determine a stochastic range for a value to fall in, while fields in a freezer are there numbers themselves.

There is however a strong connection between the two and in an attempt to close this gap I used a system of assumed naming of a freezer's waterfall. If a freezer was known as `plot`, the associated freezer would be `FREEZER_plot`. The capitals were intended to mirror the Java enumerated types convention that things that are constants are written in capitals. In hindsight having to write out names in capitals made for hard to write, ugly looking code. This name scheme allows the waterfall to find the corresponding freezer using reflection on object creation.

As a grammar element of an L-system a waterfall has an input, some outputs and a set of parameters. The inputs and outputs became known as 'plugs' as the terminology is in heavy use in the Maya API. If a waterfall is not a leaf node it will reference other waterfalls through its output plugs. The user uses a 2D GUI to connect up a set of waterfalls, all connected to one omnipresent 'root' node. This process creates a tree-like directed (sometimes) cyclic graph. As *Sity* uses a stochastic L-system each output may lead to several different inputs with a variety of probabilities.

When the user asks to evaluate the grammar we must traverse the graph of waterfalls creating geometry, but encroaching on this design space is the requirement that the freezer be given a relatively simple, elegant and easy to debug interface as much programming work takes place here.

A freezer has values specific to itself that are generated each time a freezer of this type is created, this may include for example, the exact height of a house given the mean and standard deviation defined in the accompanying waterfall. It must also be able to call downstream freezers and inform them of their input state. These two data sources – from the waterfall and from the grammatical parent are hard to present in an easy to use interface to freezer. It is very important that the interface to a freezer is clean and extendable as this will be used by future programmers extending *Sity*'s capabilities.

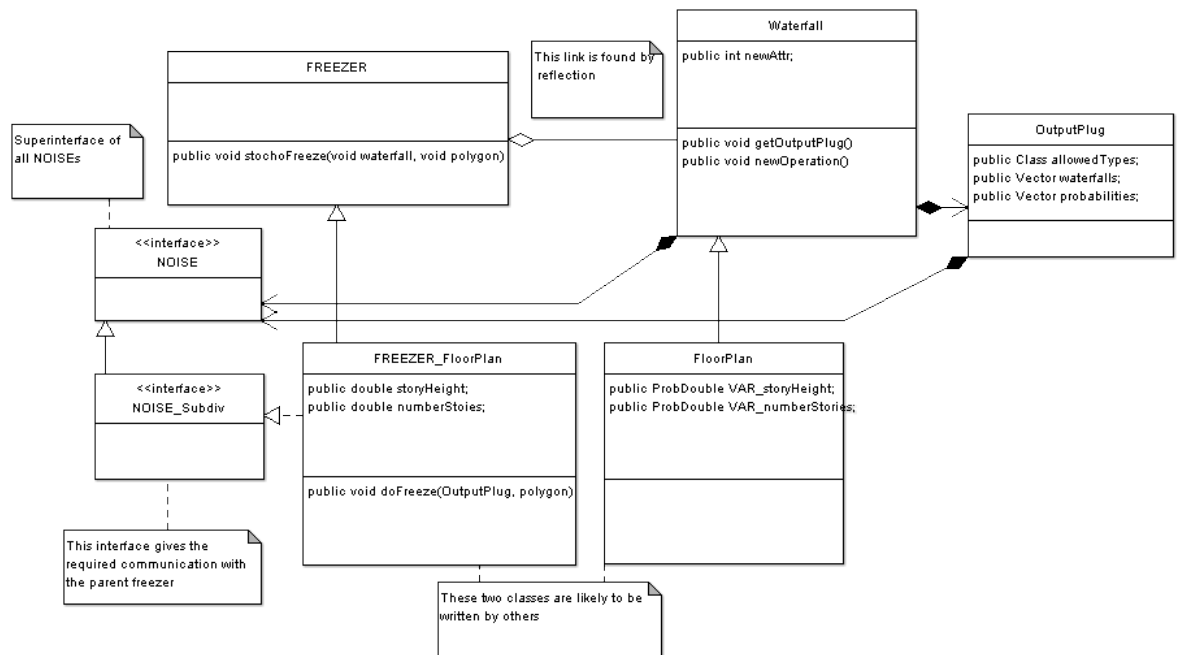


Figure 30 – freezer and waterfall inheritance system

Eventually I decided on an interface and reflection based approach show in Figure 30. Each waterfall and freezer is assigned a ‘noise’. This is a Java interface definition and this mechanism gives the extensibility to deal with a range of different interfaces between freezers. For example a dot generator waterfall passes a list of points, while a decorator defines a series of heights and weights. These noises are implemented by the freezer, but found automatically by the waterfall (by examining the interfaces of the found freezer at runtime). The waterfall needs to know its noise so that it can only be connected to output plugs with the same noise.

When a freezer asks for the next freezer of a certain type (via a freezers stochoFreeze() function), a routine finds the probabilities of all waterfalls specified from the given plug in current freezers waterfall and stochastically chooses the next waterfall. This is then instantiated as a freezer using the pointer from the waterfall object to the desired freezer class. The next stage is to instance to input variables to the specified freezer, these are also stochastic and the parameters are taken from the waterfall. There are a large variety of input variables (integers, Boolean, lists etc...) and as it is so important to give the programmer a simple interface these are copied automatically into local fields in the freezer using reflection. This is based on string matching of the parameters sets specified in the waterfall to those in the freezer:

```

private void setLocalFields()
{
    try
    {
        // find all variables
        Field[] myFields = getClass().getFields();
        // hashtables of variable names ordered by types
        LookUp lookup[] = { LUInt, LUDouble, LUBoolean, LURest };
        for (LookUp look : lookup)
            for (Field f : myFields)
            {
                // is the field in one of our hashsets
                Object l = look.find(f.getName());
            }
    }
}
  
```

```

        if (l != null)
        {
            // if so assign it...
            f.set(this, l);
        }
    }
}
catch (IllegalAccessException e)...

```

This technique requires that the programmer copies the variable names between the waterfall and the freezer correctly as Java does not have the ability to create fields in classes at runtime. Perhaps if large scale development of *Sity* ever took place an IDE tool to create waterfalls and freezers could remove this weak link. The effort in this naming issue is paid back by not having to call a function to return an instanced value. For example a typical waterfall/freezer pair would be:

FREEZER_Floorplan.java (freezer)	FloorPlan.java (waterfall)
<pre> public class FREEZER_FloorPlan extends FREEZER<FloorPlan> implements NOISE_Subdiv { // these variables are automatically copied from waterfall VAR_name -> public double storyHeight; public int numberStories; public FREEZER_FloorPlan(FloorPlan w, Random r) {super(w,r);} // this is called by the grammatical parent's stochoFreeze(FloorPlan..) command public void doFreeze(Sheaf in, List<FREEZER> parent) { roof =(NOISE_SheafChange) stochoFreeze (waterfall.onTop,Parameters.NULL_SHEAF); ... (NOISE_Panel)stochoFreeze (waterfall.front, Parameters.NULL_SHEAF); } } </pre>	<pre> public class FloorPlan extends Waterfall { // these are the inputs to stochastically defined variables (min, max, deviation, mean) public ProbDouble VAR_storyHeight = new ProbDouble (0.1,100,0.5,2); public String DEF_storyHeight = "how tall is one story"; public ProbInt VAR_numberStories = new ProbInt(1,Integer.MAX_VALUE, 2, 4); public String DEF_numberStories = "number of stories above this floorplan"; public SluiceManual front = new SluiceManual(NOISE_Panel.class,"Front of house appearance", this); public SluiceManual side = new SluiceManual(NOISE_Panel.class,"Side of house appearance", this); public SluiceManual back = new SluiceManual(NOISE_Panel.class,"Back of house appearance", this); public SluiceManual onTop = new SluiceManual(NOISE_SheafChange.class,"Roof or shape change of floor plan",this); public FloorPlan(Waterfall parent) {super(parent);} } </pre>

Table 3 – freezer and waterfall code comparison

The use of Java generics in the above freezer is simply a bug catching technique; by changing the generic type of the freezer we can find which output plugs are available in the waterfall at compile time. It would have been ideal to use the generic type (<FloorPlan>) as an indicator of which freezer is associated with which waterfall, but generics information is removed at compile time and so not available to run time reflection methods.

A further consideration here was the fact that reflection is being used recursively, and this obscures the error stack trace when classes are instanced. Errors are caught and re-thrown to the internal error handling system, which can dump the true cause stack to the command line and, depending on whether the program is being debugged, or not, can bring the program to a halt.

To make *Sity* easy to debug it is important that, although the output is essentially random, the same output is created each time the program is run with the same inputs. For this reason the root waterfall contains a random seed, and this is used to seed new instance of the `Random()` object. By passing this random object to all called methods and making it available by default to all freezers through their common parent class we can ensure that the output of grammar evaluations is deterministic to us. Several bugs in this system were introduced by the Java virtual machine (JVM) in combination with my IDE. Because the IDE keeps a copy of the JVM in memory, one optimisation is that frequently used shared objects are reused, for example `HashSets`. This can mean that between different runs of the program the output from `HashSet.get()` is not the same. The problem is solved with the `LinkedHashMap` and `LinkedHashSet`, as these guarantee traversal order.

GUI Design & Generation

The aim of *Sity*'s GUI is to present grammar creation and evaluation in a manner similar to that given by *Maya*'s Hypergraph. To this end the grammar is constructed graphically, by positioning waterfalls on the screen and dragging connections from the outputs to the inputs. As described above, the process for copying data from the waterfall to the freezer is automatic, leaving the waterfall as a very bare simple class. This is useful as it enables the waterfalls not only to be serialized to disk, but also generate their own GUI elements from the fields have been placed in the Waterfall class.

The waterfall object, as written by the programmer contains all the information for *Sity* to generate a graphic representation of the waterfall and present the user with a set of editable parameters (Figure 31). The waterfall keeps track of the output plugs as they are created, allowing the creation of a graphic representation of the plug with the desired number of output plugs

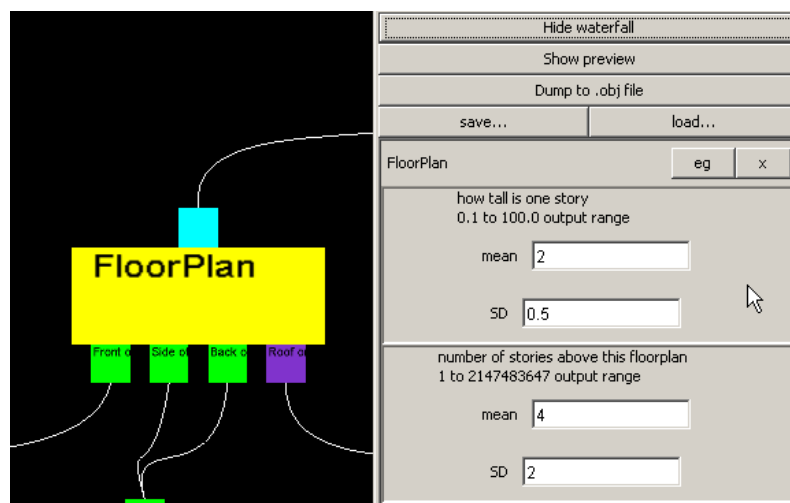


Figure 31 – the graphical element (left) and parameters (right) generated from the source in Table 3

To generate the parameters several levels of reflection are used:

- For each parameter object, say a `ProbDouble()` instance, containing the min, max and standard deviation variables, a panel is created that will contain the variables. The objects to use are marked by letting their names start with the string “VAR_”.
- We search the freezer for an accompanying string that describes the property, these have the same name as the variable themselves, but start with the string “DEF_”, if one isn’t found we assign an empty string.
- The string description of the `ProbDouble` is added to the panel, followed by a description defined by `ProbDouble.toString()`, which will typically describe the output values, typically a range of numbers.
- So that the min and max values remain hidden and unchangeable (so that some values, such as house height cannot become negative) these values in `ProbDouble` are stored in private fields.
- The public fields in the object are examined using reflection, so for `ProbDouble` this would yield two doubles – mean and standard deviation. We then create a new *generator object* specified by the class of the field (double) and the string “_GUI”. This set of classes (`double_GUI`, `int_GUI`, `List_GUI` etc....) were written by me to display and allow values of these types to be changed. These are then added with appropriate descriptions to the panel.
- These generator objects are passed the `Object` and the `Field` of the initial parameter (`ProbDouble`) to allow them to change the field in that object when their values are changed.

By allow re-use of these generator objects much time is saved in having to define the layout, appearance and method for value updates from each waterfall individually. Major sections of this system’s implementation appear in Appendix 6.

Sheets, Sheaves and Sity’s internal Polygon System

As the grammar is being evaluated it is expected to output geometry. This is performed by specifying individual polygons to be output. These polygons can also be passed as input to a downstream freezer.

Because of the planar nature of the basic camp skeleton and Voronoi algorithms discussed I decided to implement a 3D to 2D abstraction, see Figure 32. Each 2D shape is made from a list of 2D points (`FlatPoint` objects), each of these lists is known as a Sheet. To describe complex shapes with holes, or two sections (such as after shrinking with the straight skeleton) these are grouped in another object, the Sheaf, that contains a multiple Sheets and a transform. This transform supplies the additional data to move a 2D polygon into 3D space.

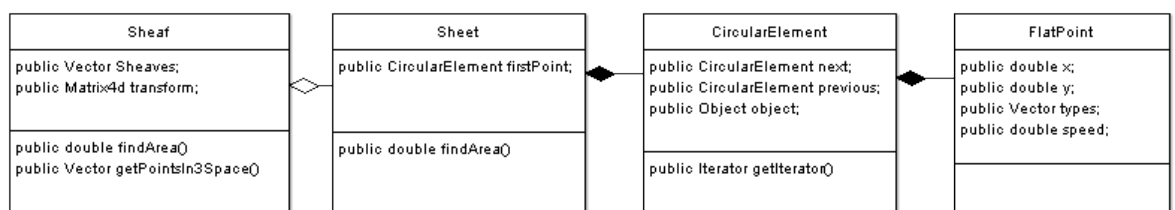


Figure 32 – Sheaf Sheet and FlatPoint abstractions

There is often a need to add metadata to parts of a polygon, for example to mark it as the top edge of bevelled camp skeleton, and so the back of a house, or to mark the front of a plot as being on the street. This is done by added one of several tags to the FlatPoint object.

To make this system easy to use a SheetBuilder class takes points specified in the parents 2D space, with an additional height variable and converts them into another Sheaf. This process is illustrated in Figure 33.

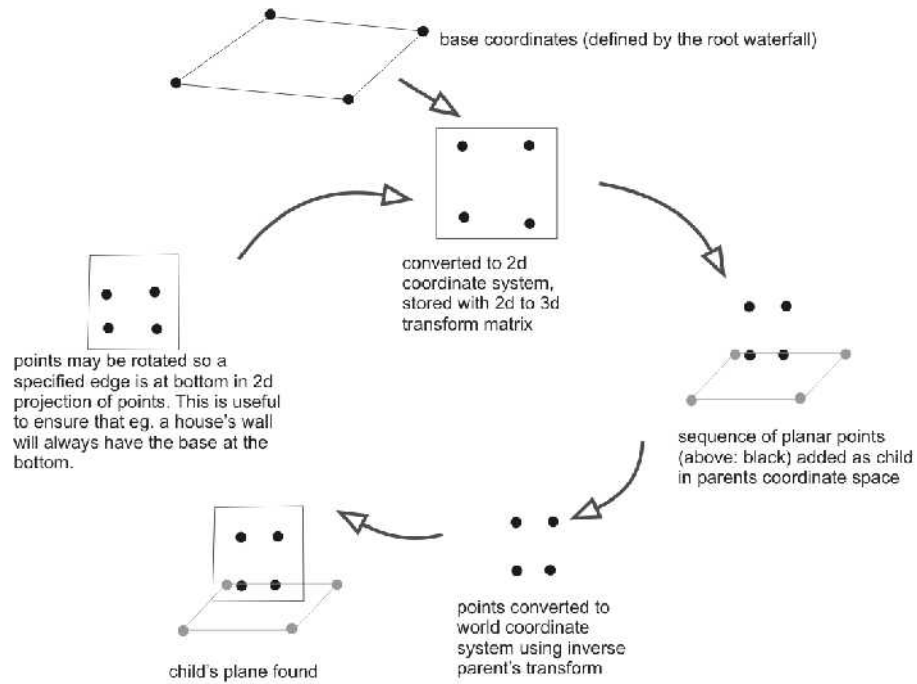


Figure 33 – SheetBuilder use

There is a potential problem with finding the plane of a set of points, it may point the wrong way. This is because there is no way to determine the normal correctly without examining the entire loop, so the initial normal is based on the cross product of the first two points. One way to determine the correct plane direction is by finding the area of the polygon relative to the assumed direction and reversing the normal if the area is negative, as explained in Figure 34.

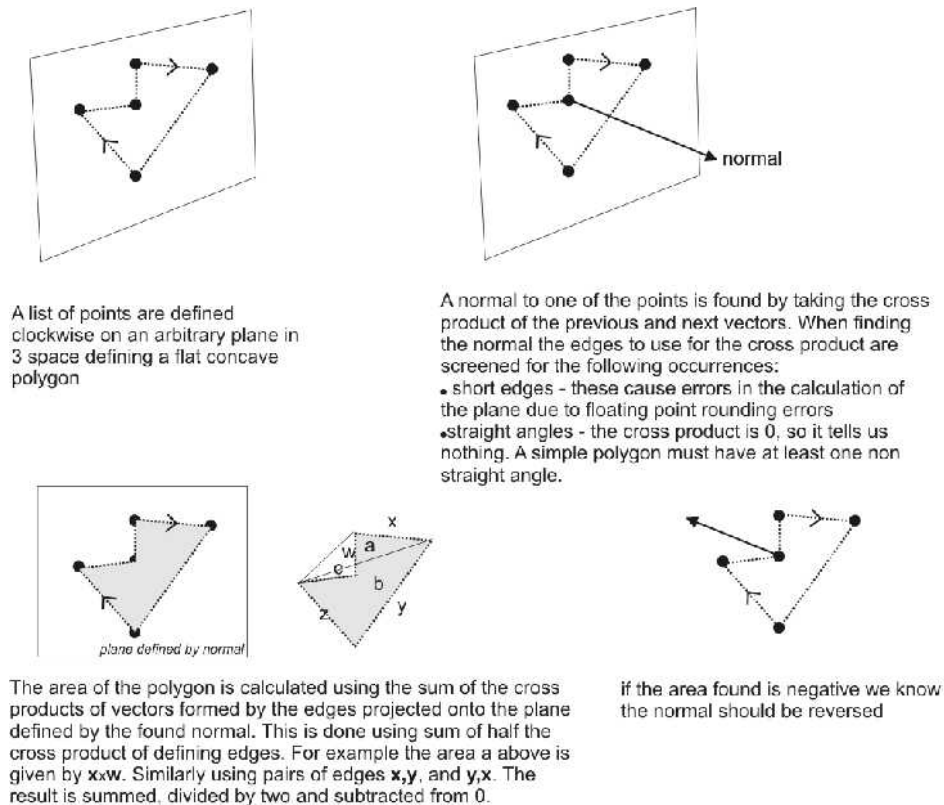


Figure 34 – Selecting the correct normal

External 3D Generation

Sity offers three geometry output options:

1. JMonkey preview window – useful for instant output.
2. Maya MEL, for slow output immediately to Maya.
3. WaveFront's .obj format – fast & optimised output suitable for many 3D packages.

To support these different options *Sity* lets the freezers access a class implementing the 'Anchor' interface. This is a static reference, using the singleton design pattern that allows lets the freezers not pass Anchor references back and forth. An anchor provides the functionality of letting a Sheaf object to be output in the required format. This system is very extensible, just writing another Anchor class is all that is required for a new type of output. This was proven the case when the MEL interface was shown to be too slow for large outputs and an .obj exporter was written in an hour.

Many of these output options could only deal with polygons without holes, such as the obj format. The solution to this was a triangulation routine, which is packaged as a *triangle Iterator* given a Sheaf. An ear clipping algorithm was implemented to this end, the details of which are given in Appendix 6.

One disadvantage of the freezer model of grammar traversal is that each surface does not have a reference to the surfaces on either side of it. This is largely a limitation of the L-grammar and the grammars design. For

this reason the MEL output of each face is output with its own set of vertices. For example a cube would have 24 vertices. A significant improvement in speed was introduced by caching all of the points in a hash set, to ensure that duplicates are identified and referenced from the correct place. This optimisation means that a cube would now only have 8 shared vertices, and is used in the obj file format export Anchor. This gives a significant speed increase when performing operations (such as rendering) this output file.

Results

The output from *Sity* is able to imitate a variety of buildings over huge areas. In real terms it is limited to areas of less than a kilometer square due to memory constraints. Generating areas this large takes in the order of fifteen minutes to complete on a modern (2006) box. Groups of ten or so blocks can be created in under a second.

The stochastic system is very effective in creating a mixture of buildings. Each of these is always different from their neighbours, but form identifiable trends within neighbourhoods. These trends can be anything from the types of buildings to the buildings heights.

One noticeable shortcoming of the Voronoi for the road map is the lack of dead ends or cul-de-sacs in the street maps generated.

I have asked several different people, each familiar with Maya, to try and building a grammar using *Sity* and they have all had reasonable success with a little prompting. After the first couple of examples they were able to create *Sity* grammars unaided. This is a very pleasing endorsement of the GUI design.

Appendix 9: Results on page 73 gives some output from *Sity* rendered in *Maya*.

Future Work

Business Steps

Sity a convincing prototype of what can achieved with this technology. It serves its purpose as a proof of concept, and although it is a distance from creating the level of detailed required for modern games it can be used to prove several things to potential investors:

- easy of use for non-technical staff
- integrity of the algorithms
- capacity to create ‘endless’ quantity of consistently detailed geometry
- effective interface with current industry tools

The first thing to do is protect the intellectual property generated so far (if it hadn’t been submitted for publication) and request patents and copyrights on the unique aspects of *Sity*. Then the job begins to get *Sity* out into the real world by showing it venture capitalist and games publishers to bring in the initial £70,000 of

seed funding that the next stage of the business plan demands. On receiving this funding the business plan would be enacted and a Limited company would be registered, some premises found and an addition developer hired

Future Work on Sity

While I started this project with many grand schemes as to what I wanted *Sity* to be, many where not practical in the time allowed. Several of these ideas are quite well developed.

The most obvious of these is the creation of doors and windows, other concepts to explore could include:

- Allowing the city to lie on a hill, as although the output is fairly realistic for a flat city like Cambridge, *Sity*'s output is not typical of cities such as Bristol.
- Letting the city grow in a non-rectangular shape around natural features such as hills or rivers.
- A total rewrite of the algorithms to deal with curves, so that curved streets could be accommodated. Using NURBS would be the obvious way forward here. Writing a camp skeleton algorithm to deal with these would prove challenging.
- A feature that is conspicuous by its absence from work published so far is the generation of gutter systems. By marking the edge of roofs and suitable locations on the walls as they are generated a search algorithm could try and connect all roof gutters to the floor. This would quickly add a unique marketing bullet point to *Sity*.
- The serialized form of *Sity*'s saved files is subject to change between different Java virtual machines. Ideally it should save out to a XML format, to allow saved files to work even when a JVM change or input definition is changed in a waterfall class.

There are various extra features that could be emulated with the camp skeleton. Figure 35i demonstrates that by adding a disappearing small (6) extension to a roof profile and assigning it suitable decorators it is possible to make dormer windows with little effort. Assigning a zero weighted edges to edges a, b, & a's counterpart (obscured) lets the window grow away from the roof as the camp skeleton progresses.

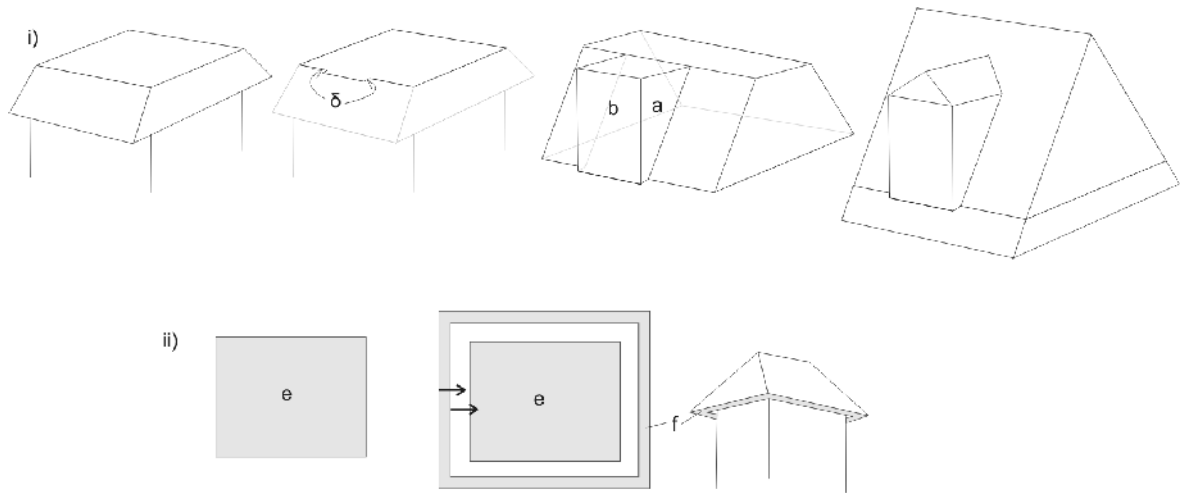


Figure 35 – Camp skeleton progressions

Figure 35 ii) demonstrates that by using a double, bevelled straight skeleton it is possible to create a ‘gutter’ (f). By assigning the outside edges of the gutter a positive weight and the inside edges a negative one, it is possible to create houses with more realistic eaves.

Figure 36 i) demonstrates a concept for creating bay windows using the Voronoi. In the figure hollow circles represent Voronoi cells that are merged, and solid circles represent cells that are calculated but not merged. By setting one point back a little it is possible to create the shape that resembles a bay window. For a straight shape, such as exists on a buttress leaving the point in line would suffice. Using the Voronoi in this way is appealing as it does not make the floor plan any larger, but rather shrinks it in.

Figure 36 ii) shows that once a hole has been cut into a roof and assigned zero speeds, perhaps using the Voronoi technique described in i, the camp skeleton will build around it (ii centre). This gives a well formed chimney. This concept of Voronoi space division may also be applied to create more interesting floor plans, such as L and T shaped buildings by selectively merging Voronoi cells that have been laid out on a floor plan.

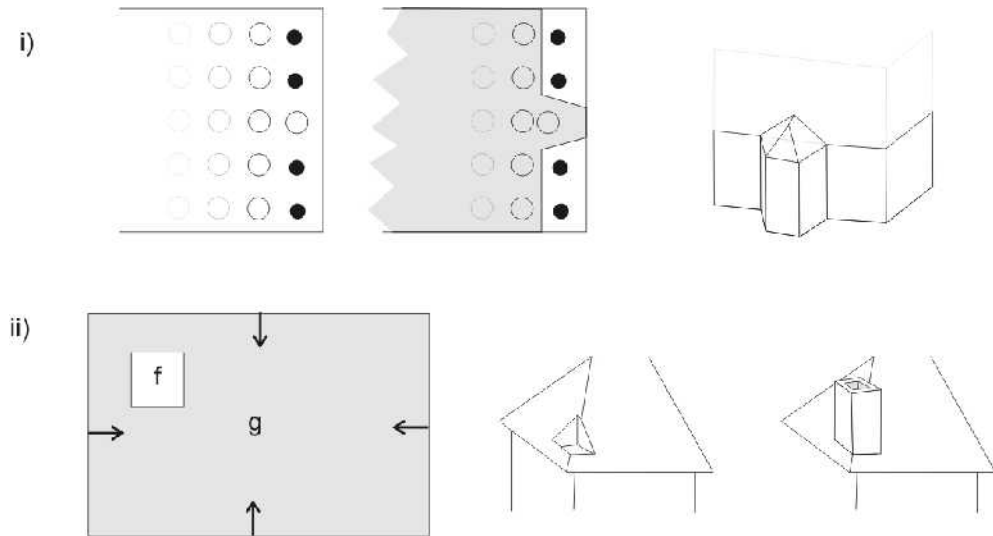


Figure 36 – Voronoi based space reduction and zero weighted holes for chimneys

Modifying the Voronoi to allow each point to have a weight would increase the variety of space divisions possible.

A final area of exploration is the statistical distributions of housing features. *Sity* explored one of these distributions – the Gaussian, successfully, but by changing the ‘random’ choices in the grammar evaluation some other possibilities reveal themselves:

- Geographic – Making a certain choice more likely because it is closer to a certain location. The distribution for this may be this instant, or of a certain falloff for example a probability of a gabled roof could be defined as $\max(n/\text{radius}^2, 1)$.
- To create the appearance of more building sets, each decision for the next choice could be set the first time it is evaluated, and only reset once the waterfall tree traversal returns to a pre defined level. For example all houses will have identical rooves until we move to the next street. Taking this one step further we could extend the probabilistic idea introduced in this paper to this to have the type of roof being used reset with a 10% probability every time we create a new house, 70% when we create a new block and 100% when we create a new neighbourhood. This could be implemented quite easily and would create some of the similarities within differences that were noted in my research.

Future Work on Theory

The camp skeleton presented should be able to run on a non-planar input polygon or set of polygons. This would allow new and interesting roof structures to be created.

Both the major algorithms presented in previous chapters have parallels in 3D, which could be implemented and used a grammar evaluation.

A 3D straight skeleton may be shown to exist, by taking a polyhedron as input, and projecting parallel extensions to each plane into 3D. Finding the collision of these planes would define the skeleton lines. Care would have to be taken in cases where 4 faces meet at one corner. The concept seems to scale well and skeletons may well exist in higher dimensions also.

The concept of a 3D Voronoi diagram exists³⁷ and is the equivalent in 3D. A set of points in 3 space divided into polyhedra with a point at its centre such that for any location in such a polyhedra the nearest point is the one inside its bounds. A 3D Voronoi would have been an ideal tool for specifying ownership of areas in *Sity*. *Sity* suffers from the fact that houses that grow outward from their plots may collide with neighbouring houses. If each plot was assigned a volume to grow into such problems would be avoided. The 3D Voronoi may provide such a volume.

A camp 3D skeleton may be an idea tool for creating the interiors of houses. By shrinking in the exterior walls by different by different amounts, building interiors may be created. Furthermore by subdividing the space using a 3D Voronoi separate floors, and floor plans, could be generated procedurally.

Appendices

Appendix 1: Miscellaneous

The renders in this paper were prepared using the *Sity*-MEL-Maya interface and rendered using Mental Ray. The Voronoi output was generated by dumping the geometry to Maya, creating a screenshot, and using a tracing program to convert the outline to a suitable form to match the other graphics in this report. Other graphics were created using an off the shelf vector application and screen captures.

Appendix 2: User Manual

A working overview of the *Sity* program is given here, for more detail about the mechanics of the options presented it is recommended that the relevant chapter (page 36) is consulted.

Getting Sity

Before you begin you'll need Java 1.5 (<http://java.sun.com>) and Java3d (<http://java.sun.com/products/java-media/3D/>).

Sity is available for download from:

<http://cs.bris.ac.uk/home/tk1748/sity.html>

and downloading the *sity.zip* file, extract the contents to a directory.

Running Sity

You can then execute *sity* with the following command from within the directory:

```
java -Djava.library.path=./lib/ -jar sity.jar -port 2424
```

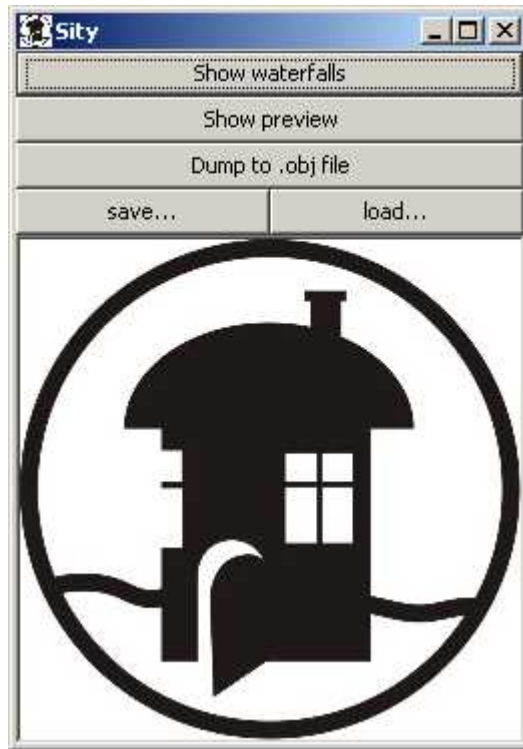
If you experience the Java error “`java.lang.OutOfMemoryError Java Heap Space`”, you can assign a larger quantity by adding the following argument to the above command.

```
-Xmx [memory size in bytes]
```

(for example `-Xmx 1000000000` to assign a gigabyte of memory)

If you get the error `Exception in thread "main" java.lang.NoClassDefFoundError: javax.vecmath.Matrix4d...`, you have forgotten to install Java3d (see above), or maybe need to specify its location in the Kava classpath.

Once *Sity* is running seen you should see the utility window:



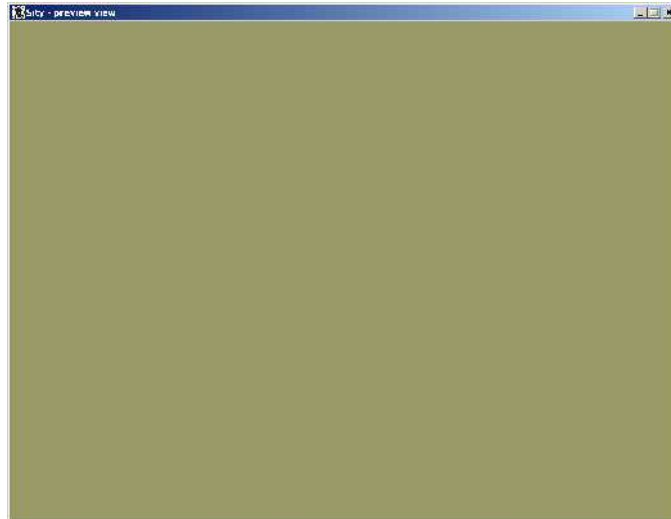
It may look slightly different depending on your operating system.

This window remains open as long as *Sity* is running. The logo will be replaced by options when operations are performed. You can quit *Sity* at any time via the usual operating system methods, or by pressing the Q key.

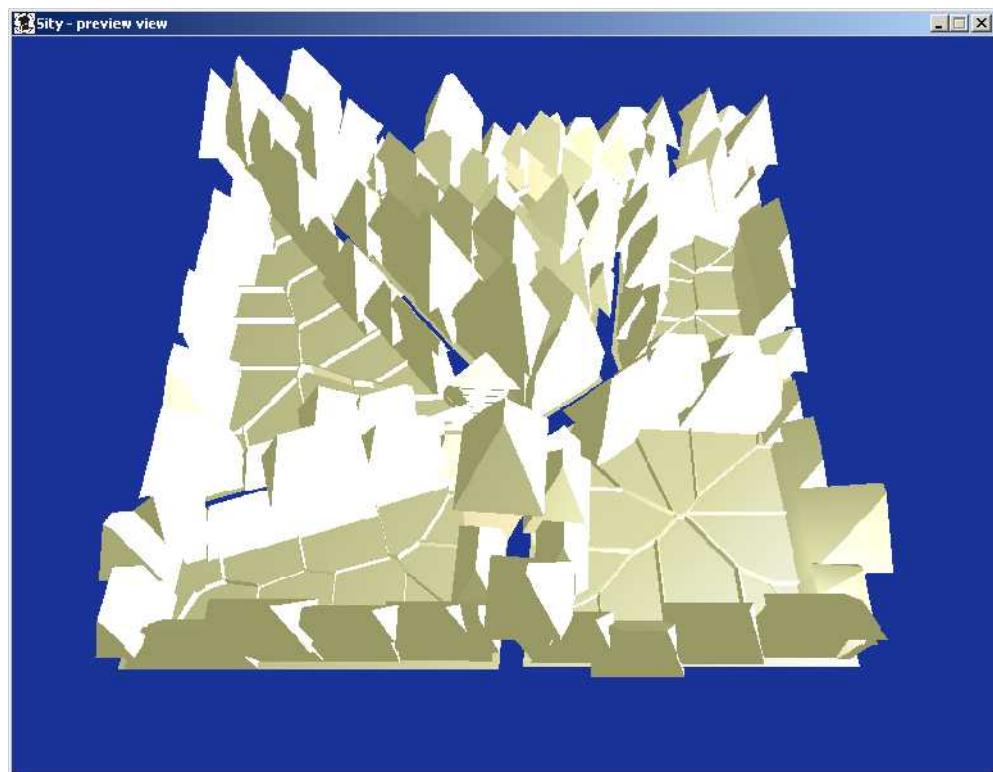
Introduction

Sity creates cities from a set of waterfalls that you connect together. Learning what each waterfall does, and which options they have takes a little time to learn, but is well worth it.

To create your first *Sity*, click the "Show preview" & the preview window appears, after a couple of seconds a city will also appear.

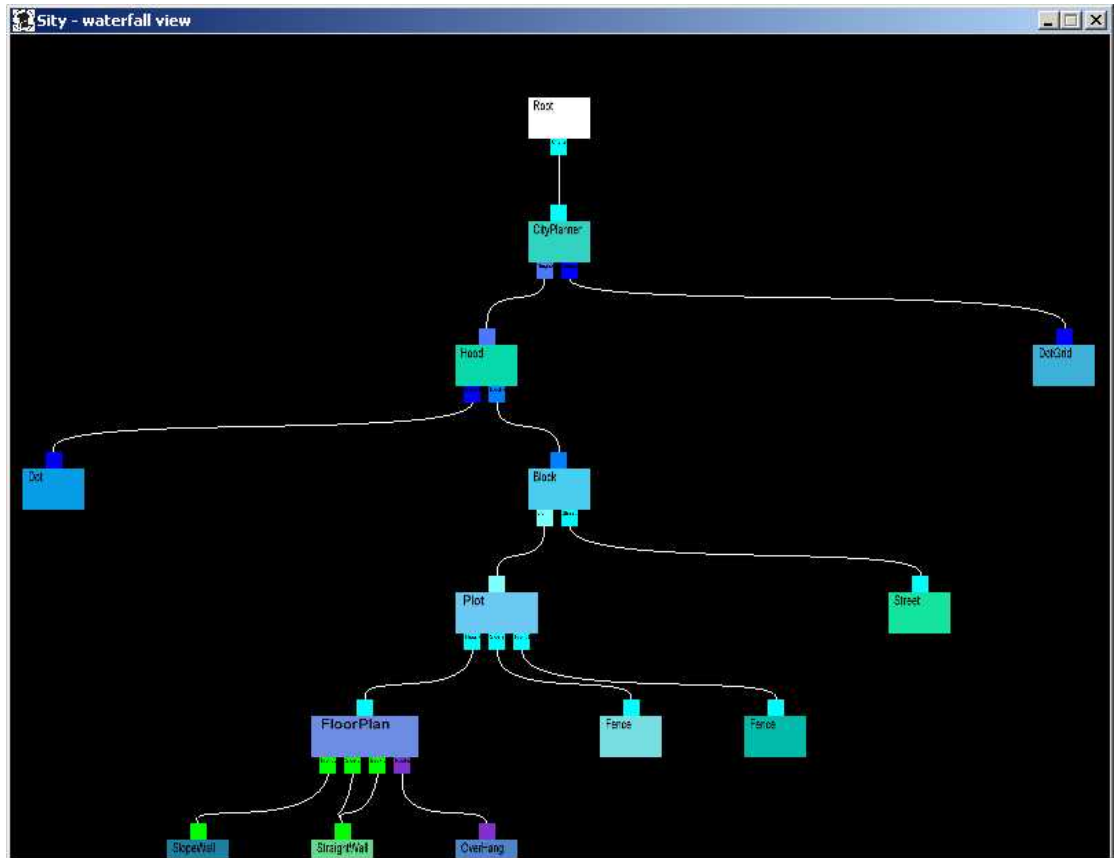


After clicking in this window you can navigate the city using the arrow keys and AD to slide left and right and WS to move forwards and backwards.



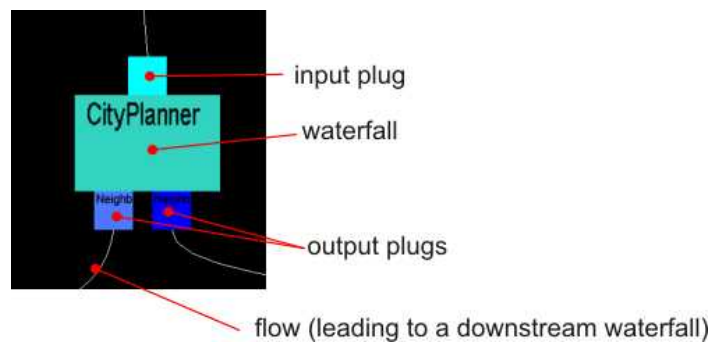
From the preview window you can ask for the current city to be sent to Maya by pressing the return key, and can generate a different city by pressing the space bar. More details about these event to follow.

Returning to the utility window clicking the “Show waterfalls” will display the waterfall graph:



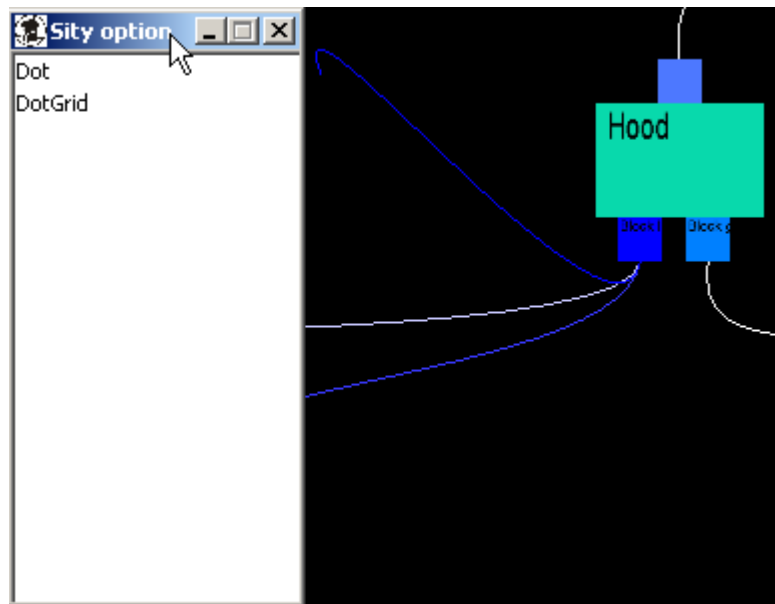
This structure determines how the output city appears.

The waterfall has the following elements:

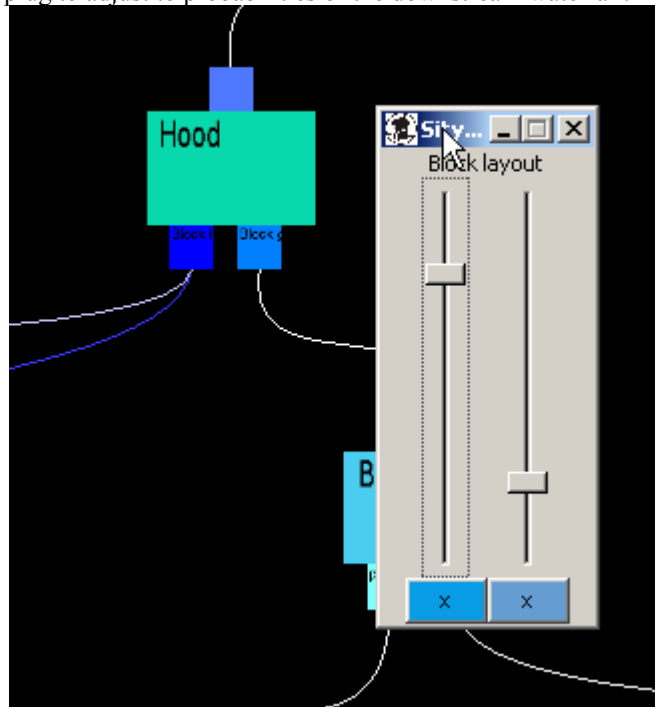


The controls in the waterfall view are

- Click and drag on the black background – move around the view
- Scroll in or out with the mouse, zoom the view
- The R key will attempt to automatically lay out the waterfalls
- Click on a waterfall or input plug to select it (note that the utility panel displays the options relevant to the waterfall)
- Click and drag from an output plug to an input plug to create a new link
- Click and drag from an output plug to a blank space to create a new waterfall, you will be presented with a menu of compatible waterfalls:

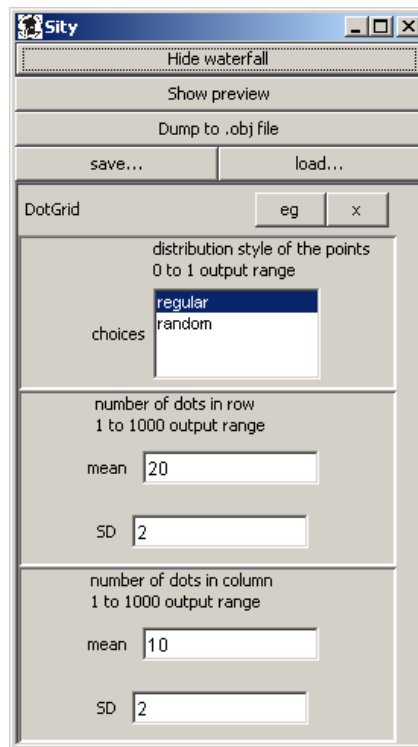


- Click on an output plug to adjust to probabilities of the downstream waterfall:



Each of the sliders controls the probability of each of the downstream waterfalls. By hovering the mouse pointer over the sliders, the tool tips will inform you of what each connection leads too, at the same time that flow will become highlighted. To remove a flow click the cross (x) button at the under the sliders. To adjust the probability of the downstream waterfall, move the slider up or down. The higher a slider the more likely the waterfall is to be chosen.

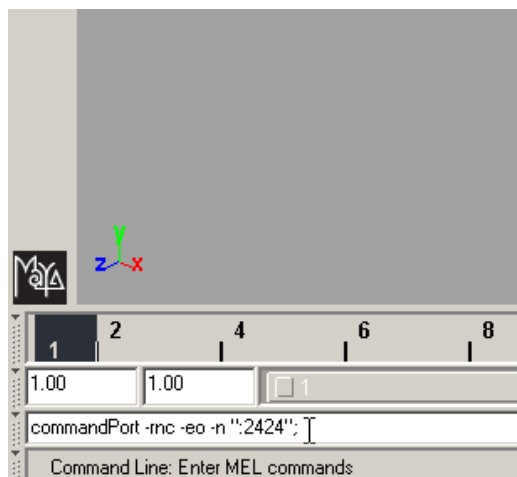
By connecting up different waterfalls in different ways a variety of cities can be created. Another way of changing the appearance is through the parameters that appear in the utility window when you click on a waterfall. A description of the waterfalls and the parameters appears at the end of this appendix.



Outputting a City via MEL

As describe above, pressing the return key in the preview window will attempt to output the current sity to Maya via MEL over the local port 2424. However Maya must first be told to listen to that port. To do this open Maya, and in the MEL command box enter

```
commandPort -rnc -eo -n ":2424";
```



Once you are finished, the port should be closed again as Maya does not check who is writing to that port and poses a security risk. Close the port using:

```
commandPort -cl -n ":2424";
```

Once the port is open, upon pushing return in the preview window the data is sent to Maya, the Maya will flicker and may stop responding during this time. Once this has stopped, the city will have been transferred to Maya.

Outputting a city as an .obj file

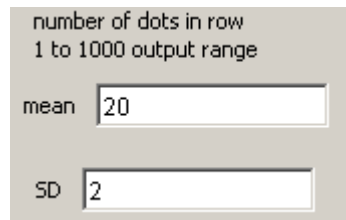
It is also possible to output a city as a .obj format file, this may be faster, produce the same result and produce less vertices than the MEL method above. Click the 'Dump to .obj file' option on the utility panel. You can then open the city in a range of 3D packages.

Saving and loading a set of waterfalls

The save... and load... buttons will save and load the tree of waterfalls to disk. Note that only those connected to the root node are saved and recovered.

About the Waterfalls

When asked to make a city, *Sity* works its way through the waterfalls, starting at the root and one of the branches of the following output plugs. At a later time while creating the city, *Sity* may return and create that element again, at that time it may take a different route if there is more than one waterfall connected to the downstream plug. The probability of which waterfall is chosen is set by clicking on the output plug (explained above). You can change the way a waterfall acts by changing its parameters. Often these contain several choices for one variable, such as block width:



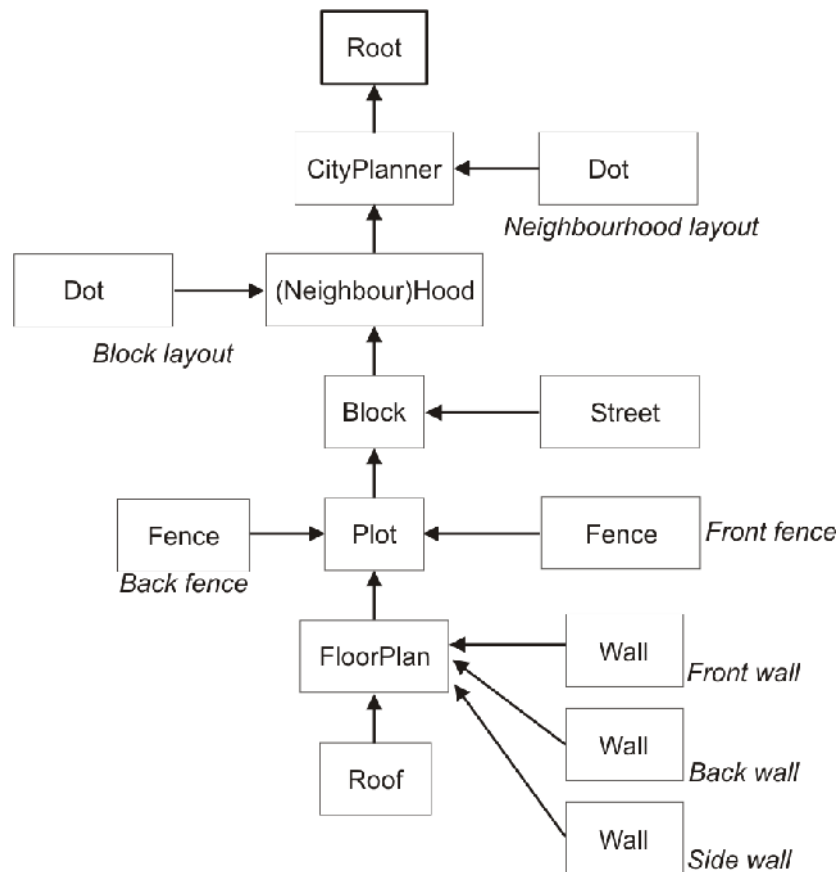
number of dots in row
1 to 1000 output range

mean 20

SD 2

The mean specifies the most likely value (approximately meters like all values in *sity*) and the SD or standard deviation specified how far from this number the value may be. So for many different, chaotically sized blocks you would set the SD to the same value as the mean. For all waterfalls to have the mean value set the SD to zero. The values are truncated to be in the limit specified (here a 1m to 1000m output range), to ensure there are no invalid sizes. You cannot change these ranges.

On the following pages are a summary cascade of waterfalls (a *grammar*) of a *Sity* city followed by a table detailing the different components properties.



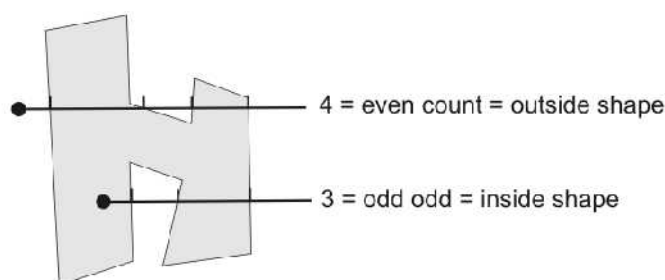
Root	At the top of the waterfall is the Root waterfall node, this cannot be deleted like the other waterfalls.	Root random – the random seed that generates a city, changing this will mean that you get another random city. This value is changed by pressing space in the preview window Input height & width – how big an area city operates in.
City Planner	City wide choices. The neighbourhood space generator uses a set of dots to set the centres of the neighbourhoods.	Merge area – what is the smallest block allowed? Probability of Blocks - how ‘bombed out’ this city is, this causes more or less blocks to be merged
Hood (Neighbourhood)	Defines a block and dot generator for a given neighbourhood. Typically an entire hood will have streets in a set pattern	-
Dot	Generates dots in a given style. These dots define the centres of the blocks used in the neighbourhood	Square – generates rectangular blocks Spiral – generates spiral cities Circular – generates ‘spoke’ cities Noise – how much variety there are in the defined blocks
Block	The island of land bounded by	Small plots merged – how small must a plot be

	streets	before it is merged into its neighbours.
Street	Specifies a street width for the block	Width – specifies width
Plot	Specifies how a plot is broken up into fences and floor plans	<p>Street space – what is the ideal street frontage for each house onto the street</p> <p>Side Space – how much space a house has between it and its neighbours. If this value is zero the wall will only be created at the back</p> <p>Front space – how much additional space to add at the front of the house</p> <p>Side Space – how much space to add at the sides and back of the house</p> <p>Depth – how deep is the house</p> <p>Front Wall – chance of wall at front</p> <p>Garden Wall – chance of wall at back</p> <p>Gap size – how big a gap in the front wall to leave for people to walk through.</p>
Fence	Specifies a fence or wall	Height – specifies the height of a fence
FloorPlan	Controls the walls and height of a house. The output plugs define what each wall on the house will look like. Some wall appearances can be added to each other to create complex shapes.	<p>Story height – how tall one story in the house is</p> <p>Number of Stories – the number of stories in this house</p>
Overhang	Specifies the roof should hang away from the wall	<p>Front/sides/back – distance the overhang should go in these directions.</p> <p>Drop – how far the overhang comes down over the top of the house</p>
StraightWall	A simple straight wall that rises vertically	-
SlopeWall	A sloping wall with optional decoration	<p>Slope – gradient of wall – 0 is vertical, 1 is 45 degrees.</p> <p>Lump height – height of wall decoration</p> <p>Lump depth – extrusion caused by decoration</p>
SlopeRoof	A simple sloping roof	<p>Slope – gradient of wall – 0 is vertical, 1 is 45 degrees.</p> <p>Minimum steepness – the lower limit of how the random element can change the slope</p>

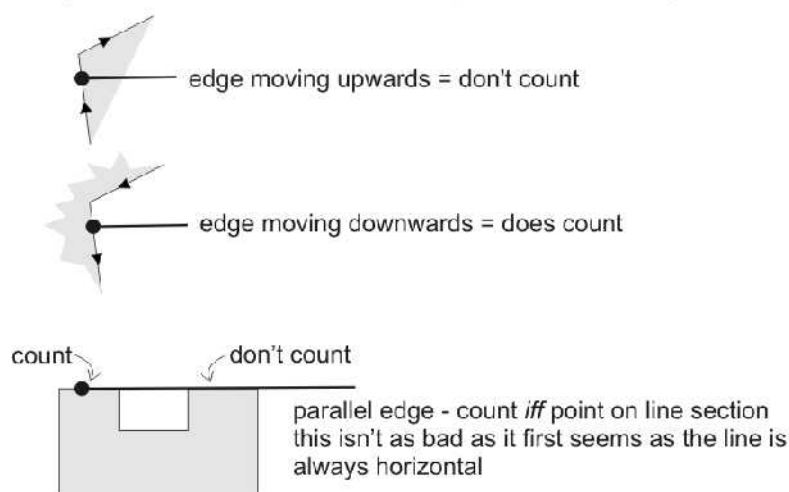
Appendix 3: Jordan Curve Algorithm

The basic Jordan curve algorithm is defined on the web page http://www.siggraph.org/education/materials/HyperGraph/raytrace/raypolygon_intersection.htm, and uses a projections of the polygon in question onto a 2d plane to reduce the problem to a 2d one, then it counts the number of crossing of a line from the collision point to infinity, of the 2d polygon. I implemented the Jordan curve code in *Sity* myself in order to make the modifications below that consistently allow intersections on edge points:

Jordan curve - count number of line crossings:



my modifications for borderline cases; to include boundary in clockwise defined polygons:

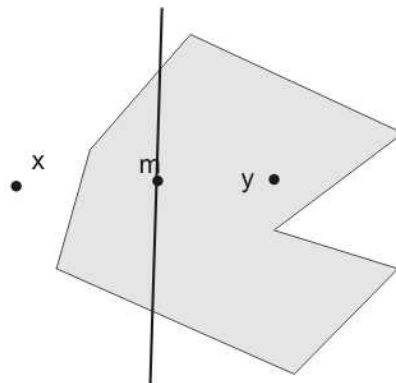


Appendix 4: Index of Figures and Tables

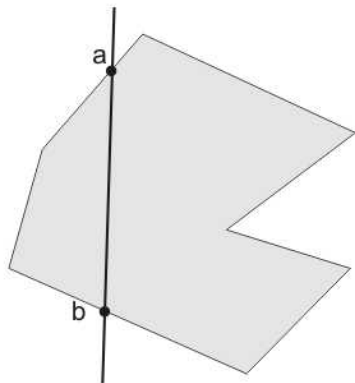
Figure 1 DTI: Typical Development Talent Split.....	8
Figure 2- How Buildings Learn, front cover, left: houses New Orleans 1857, right same houses 1993.....	11
Figure 3 – a straight skeleton (not-bold) of a polygon (bold).....	15
Figure 4 - Comparison of Voronoi diagram and a Bristol street map.....	16
Figure 5 – Autodesk’s Maya Hypergraph.....	18
Figure 6 – Top left, the idea of shrinking a polygon(bold) to give the straight skeleton (top right). Skeleton extrapolated to 3D, bottom.....	19
Figure 7 – i) concave pointer layout, ii) changes that result after a split event, edge pointers are double arrows next and previous pointers in the active list are single arrows.....	20
Figure 8 – a split event.....	22

Figure 9 – active list changes on finding a split event of a against edge yx.....	22
Figure 10 – finding the opposite side in a split event.....	23
Figure 11 – careful split event clarification is required.....	24
Figure 12 this shape features a double split event.....	24
Figure 13 – a hole in weighted straight skeleton.....	25
Figure 14 two similar polygons with varying weights, right: the left most face has a negative weight.....	26
Figure 15 – a split event on a convex angle.....	27
Figure 16 – Calculation of bisector vector.....	28
Figure 17 – the shape in Figure 6, with a bevelled top.....	29
Figure 18 – bevel procedure.....	29
Figure 19 – point traversal scheme.....	30
Figure 20 – Voronoi calculation.....	32
Figure 21 – the need to remember where we should go.....	33
Figure 22 – corner disambiguation.....	34
Figure 23 – finding the bisector in convex shapes.....	35
Figure 24 Concave voronoi tesslation; left: input points and boundary, right output geometry.....	36
Figure 25 – block generation.....	37
Figure 26 – block to plot generation.....	37
Figure 27 – shrinking the boundaries to create a wall.....	38
Figure 28 – generating walls and a roof using the camp skeleton.....	39
Figure 29 – Sity Package level overview.....	42
Figure 30 – freezer and waterfall inheritance system.....	44
Figure 31 – the graphical element (left) and parameters (right) generated from the source in Table 3.....	46
Figure 32 – Sheaf Sheet and FlatPoint abstractions.....	47
Figure 33 – SheetBuilder use.....	48
Figure 34 – Selecting the correct normal.....	49
Figure 35 – Camp skeleton progressions.....	52
Figure 36 – Voronoi based space reduction and zero weighted holes for chimneys.....	53
Figure 37 – Sity’s Roadmaps (one) and Real World Roadmaps.....	73
Figure 38 – Sity’s Roadmaps (two).....	74
Table 1 – The required checks when finding collision on camp skeleton bisectors. Grey area shows interior bisectors, those considered by previous authors.....	27
Table 2.....	40
Table 3 – freezer and waterfall code comparison.....	45

Appendix 5: Voronoi Bisector calculations



To find the bisector of the points x,y inside the cell 'owned' by y we find the midpoint m, then rotate the vector xy by 90 degrees and move it so it runs through m



edge start point = s2
 edge direction vector = d2
 bisector start point (anywhere on line) = s1
 bisector direction vector = d1
 parametric coordinate of bisector =

$$\frac{d2.x*(s2.y-s1.y)+d2.y*s1.x-s2.x}{d2.x*d1.y- d2.y*d1.x}$$

 similarly we check that the parametric coordinate of the edge is in the range 0..1, this is as above but with the pairs of variables d1,d2 and s1,s2 swapped

parametric line formulae modified from: <http://astronomy.swin.edu.au/~pbourke/geometry/lineline2d/>

The calculations used to find the parametric form on the line are shown below. The two lines (the bisector and the edge) are defined by start points and directions, x1b, y1b as the start points and x1 as the direction. This is very similar to the work Cyrus and Beck³⁸.

$$\textcircled{1} x_1b + t_1x_1 = x_2b + t_2x_2$$

$$\textcircled{2} y_1b + t_1y_1 = y_2b + t_2y_2$$

solve for t_2 !

$$\text{from } \textcircled{1} t_1 = \frac{x_2b + t_2x_2 - x_1b}{x_1}$$

substituting into $\textcircled{2}$

$$y_1b + \frac{y_1}{x_1}(x_2b + t_2x_2 - x_1b) = y_2b + t_2y_2$$

$$y_1b + \frac{y_1}{x_1}(x_2b - x_1b) = y_2b + t_2y_2 - \frac{y_1t_2x_2}{x_1}$$

$$y_1b - y_2b + \frac{y_1}{x_1}(x_2b - x_1b) = t_2(y_2 - \frac{y_1x_2}{x_1})$$

$$\frac{x_1(y_1b - y_2b) + y_1(x_2b - x_1b)}{x_1y_2 - y_1x_2} = t_2$$

• $x_1b = \text{baseX}(\text{start line } x, \text{ coord})$
 \downarrow $x_1 = \text{directionX}(\text{direction along } x \text{ from } x_1b)$
 ...
 dir 1 know!

Appendix 6: Automated Code Generation source

The GUIser class is called for each public field in the object to be display, a description is also passed in:

```
/**
 * Reflection based class that creates an
 * editor for another class (that has
 * only elemental fields) for some
 * definition of elemental
 *
 * @author people
 *
 */
public class GUIser {

    Object object;

    String desc;

    /**
     * Object we are operating on and its
     * description
     *
     * @param in
     * @param d
     */
    public GUIser(Object in, String d) {
        object = in;
        desc = d;
    }

    /**
     * does the hard work
     *
     * @return
     */
    public Component getGUI(){
        JPanel out = new JPanel();
        out.setToolTipText(desc);
        out.setLayout(new BoxLayout(out,
        BoxLayout.PAGE_AXIS));
        out.setBorder(new
        BevelBorder(BevelBorder.LOWERED));
        Field[] fields =
        object.getClass().getFields();

        JLabel jl = new JLabel(desc);
        out.setToolTipText(desc);
        jl.setToolTipText(desc);
        out.add(new JLabel(desc));

        out.add(new JLabel(object.toString()));

        for (Field f : fields) {
            String s = desc;
            s = s.concat(object.toString());
            out.add(getComponent(f, object, desc));
            out.add(Box.createHorizontalGlue());
        }
        return out;
    }

    /**
     * A 'switch' statment of primitive types
     * that returns a gui editor for that
     * type
     *
     * @param f
     * @param toUse
     * @return
     */
    private Component getComponent(Field f,
    Object toUse, String desc) {
        JPanel out = new JPanel();
        out.add(new JLabel(f.getName()));

        try {
            Object val = f.get(toUse);
            try {
                // System.err.println("looking for
                >" + f.getType().getSimpleName() + "_GUI<");
                Mojo m = new Mojo("util." +
                f.getType().getSimpleName() + "_GUI", val,
                new ObjectWrapper(toUse), f);
                out.add((Component) m.run("makeGUI"));
            } catch (ClassNotFoundException e) {
                out.add(new JLabel(" Field type " +
                f.getType().getSimpleName()
                + " not editable yet!"));
            }
        } catch (IllegalAccessException e) {
            sity.Parameters.fatalErrorSD("trying to
            set field " + f.getName());
        }
        return out;
    }
}
```

```
}
```

The Mojo class is just a wrapper for Java's reflection procedures:

```
/**
 * Wrapper for reflection, mostly from
 *
 * http://java.sun.com/docs/books/tutorial/re
 * flect/object/invoke.html
 *
 * @author people
 *
 * @param <T>
 */
public class Mojo<T> {
    T o = null;

    public Mojo(Class c, Object... arguments)
    {
        setup(c, arguments);
    }

    public Mojo(String s, Object...
arguments) throws ClassNotFoundException {
        Class c = Class.forName(s);
        setup(c, arguments);
    }

    private void setup(Class c, Object...
arguments) {

        try {
            Class[] ca = new
Class[arguments.length];
            for (int i = 0; i < arguments.length; i
++)
                ca[i] = arguments[i].getClass();
            Constructor constructor =
c.getConstructor(ca);
            o = (T) createObject(constructor,
arguments);
        } catch (NoSuchMethodException e) {
            fatalErrorSD(e.toString());
        }
    }

    public T get() {
        return o;
    }

    public Object run(String name, Object...
arguments) {
        Class c = String.class;

        Class[] ca = new
Class[arguments.length];
        for (int i = 0; i < arguments.length; i+
+)
            ca[i] = arguments[i].getClass();

        Class[] parameterTypes = new Class[]
{ String.class };
        Method method;
        try {
            method = o.getClass().getMethod(name,
ca);
            return method.invoke(o, arguments);
        } catch (NoSuchMethodException e) {
            fatalErrorSD(e.toString());
        } catch (IllegalAccessException e) {
            fatalErrorSD(e.toString());
        } catch (InvocationTargetException e) {
            e.getTargetException().printStackTrace(
);
            fatalErrorSD(e.toString());
        }
        return null;
    }

    public static Object
createObject(Constructor constructor,
Object[] arguments) {
        Object object = null;

        try {
            object =
constructor.newInstance(arguments);
            return object;
        } catch (InstantiationException e) {
            fatalErrorSD(e.toString());
        } catch (IllegalAccessException e) {
            fatalErrorSD(e.toString());
        } catch (IllegalArgumentException e) {
            fatalErrorSD(e.toString());
        } catch (InvocationTargetException e) {
            fatalErrorSD(e.toString());
        }
        return object;
    }

    public static void set(Field f, Object
source, Object value) {
        try {
            f.set(source, value);
            // } catch (NoSuchFieldException e) {
            // System.out.println(e);
        } catch (IllegalAccessException e) {
            fatalErrorSD(e.toString());
        }
    }
}
```

Finally as an example of a generator class, double_GUI:

```
public class double_GUI implements
PropertyChangeListener {
    private double v;

    Object object;

    Field field;

    public double_GUI(Double value,
ObjectWrapper source, Field field) {
        v = value;
        this.object = source.getObject();
        this.field = field;
    }

    public Component makeGUI() {
        JPanel out = new JPanel();

        JFormattedTextField amountField = new
JFormattedTextField(NumberFormat
.getNumberInstance());
        amountField.setValue(v);
        amountField.setColumns(10);
        amountField.addPropertyChangeListener("v
alue", this);
        out.add(amountField);
        return out;
    }

    public void
propertyChange(PropertyChangeEvent arg0) {
        System.err.println("Change to "
+
arg0.getNewValue().getClass().getSimpleName(
));
    }
}
```

```

Object o = arg0.getNewValue();
double d;
if (o instanceof Long) {
    d = Double.parseDouble(((Long)
o).toString());
} else if (o instanceof Double) {
    d = (Double) o;
} else {

```

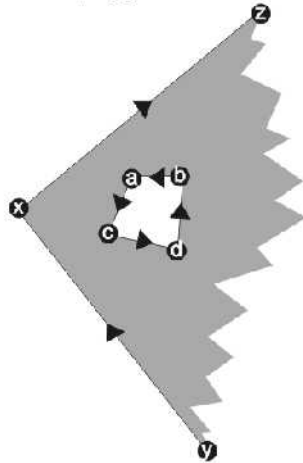
```

d = -1;
Parameters.fatalErrorSD("Property
change events returned something
strange");
}
Mojo.set(field, object, (Double) d);
}
}

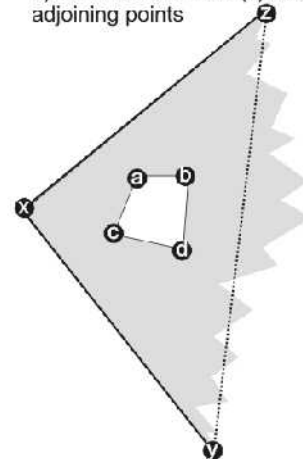
```

Appendix 7: Ear clipping algorithm

1) concave polygon with holes defined counterclockwise

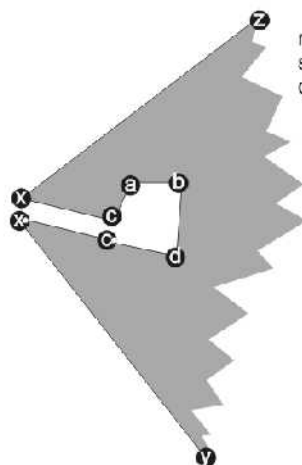


2) start with left most (x) and find if it makes an ear with adjoining points

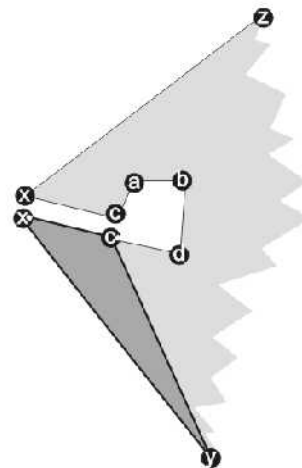


not an ear - a,b,c&d inside triangle xyz

3) order abcd by distance from zy line -d,b,a,c, and find furthest - c duplicate x and c, and split polygon down line



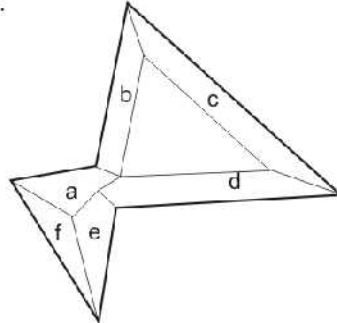
note that x, x' are in the same position, as are c & c'.



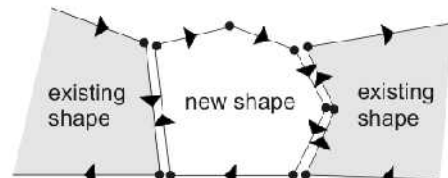
4) take left most point, x' and test it for being an ear - it is! return x'c'y and remove x' from the graph

Appendix 8: Union Algorithm

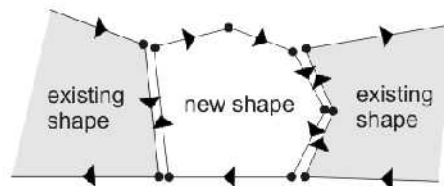
The aim of this algorithm is to union resulting edges from a bevel routine into a single (planar) surface. In the below we wish to have the shape represented by the union of shapes a through e (but not f), as these are the sides of the building on which we desire a feature (say a fence or roof). This is also used to combine the cells from voronoi output, in places such as walls.



The point sequences are stored as a doubly linked list. A hash table stores a pointer from each point to its position in the linked list. Initially it stores all the points in the existing shape, but not those in the new shape.



In the above contrived example there are two existing polygons (or one single shape wrapped all the way around). The final 'panel' eg d in the above is about to be added in a clockwise direction. First we collect all the points into a doubly linked list, so all points have next and previous pointers. Note that the points close by share the same location but are shown separately for clarity in the diagram

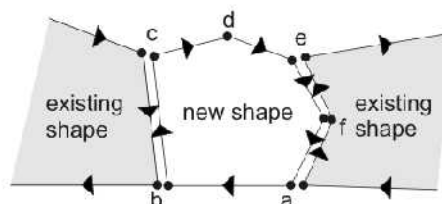


Corners are then marked as leaving a shape (a,c below), entering a shape (b,e), following a shapes border(f) or doing nothing (d). This is implemented via bit masks.

If the hash does not contain this point it is added to the hash, and tagged as a no change point.

if the line leaving the point in the new shape has the same destination as the line entering the point in the existing shape then we are either following a border or entering.

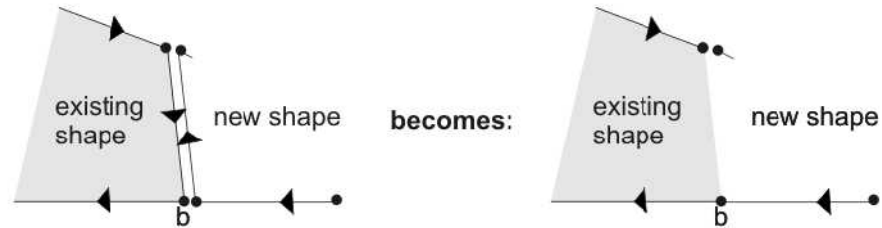
if the line entering the point in the new shape has the same origin as the line leaving the point's destination in the existing shape then we are either following a border or leaving.



After all points have been tagged, they are then operated upon as follows depending on the tag:

points that are doing nothing are added to the hash

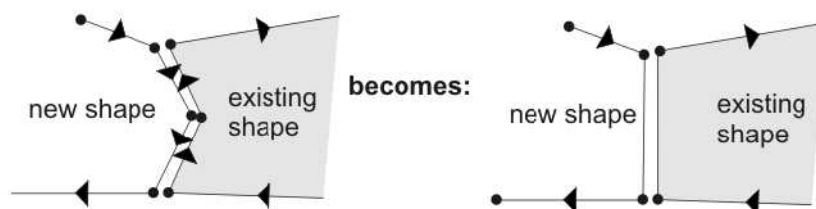
pointer changes when entering: (note that the old pointers are removed, or set to null). If a subsequent operation finds a null pointer, the operation is ignored.



when leaving a shape:



when following an edge (note: the point is then removed from the hash)



In this method each extra shape is unioned with those (if any) already added. Once done we are left with a list of pointers consisting, each with an entry in the hash table defining one or more loops (in either direction, there may be holes) of points.

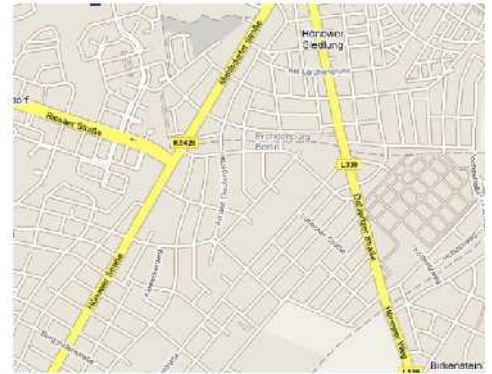
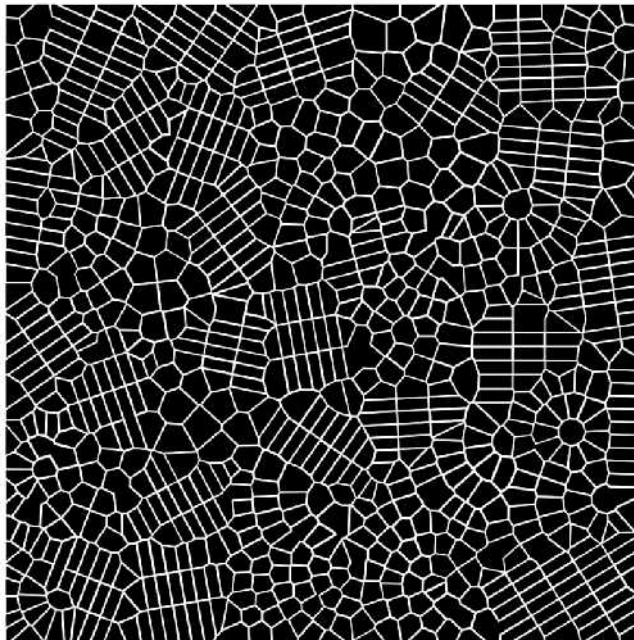
We then repeatedly take the first point in the hashtable, following all the points in any loop it defines, removing them from the hashtable, until the table is empty. Each loop is output as a sheaf, it is important for the first point to be on a non-hole defining loop as the sheaf must have the correct normal. I manage this when traversing a straight skeleton by always starting on an point that is input to the skeleton procedure (on the original sheaf);

The algorithm runs in $O(n)$ time and space where n is the number of points entered.

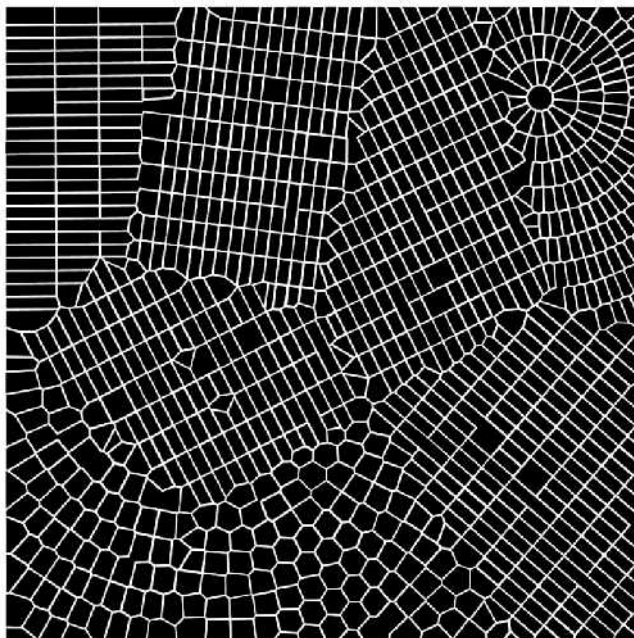
Appendix 9: Results

Road Maps

Figures Figure 37 and Figure 38 show some sample output of *Sity*'s block generator with some real life maps for comparison.

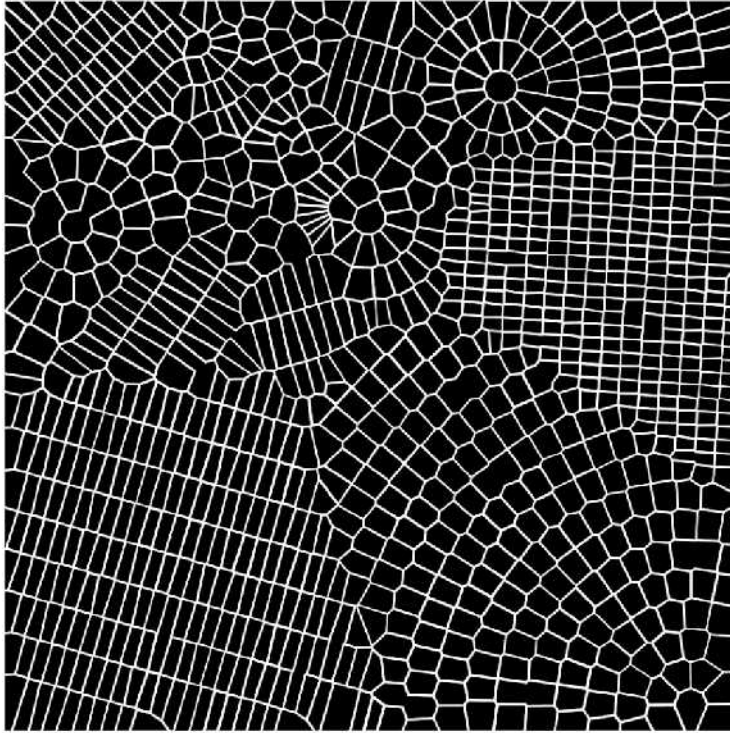


Sity output, left, and Berlin roadmap above. Notice the different neighbourhoods of roads with associated road patterns, and the manner in which these areas merge.

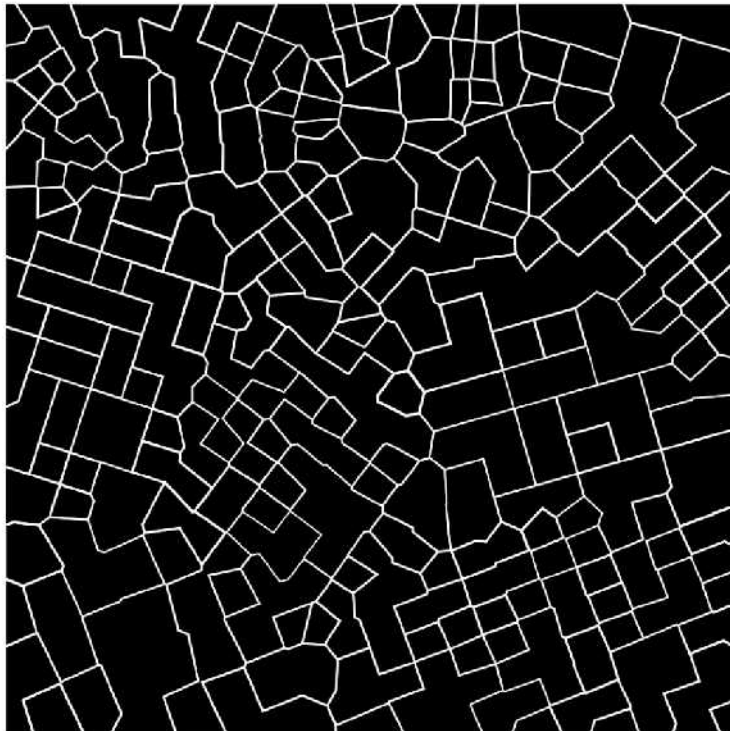


Paris, above, has many circular sections of road. These are replicated by Sity. Sity is successful in modelling these areas, but fails to create larger roads between neighbourhoods

Figure 37 – *Sity*'s Roadmaps (one) and Real World Roadmaps³⁹



Using a spiral input to define the neighbourhoods leads to more variation in the 'middle' of a city. Small blocks are merged to create ones that are large enough to create houses upon.



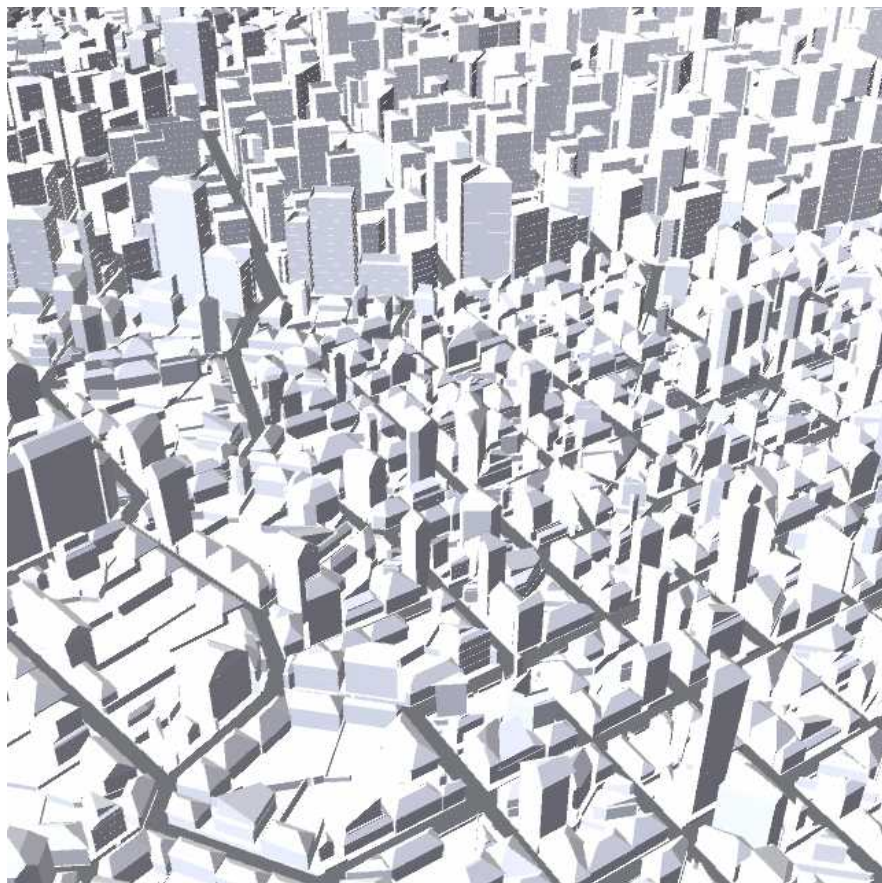
By specifying a high threshold area for merging blocks, the random method used can lead to some very strange cities.

Figure 38 – *Sity*'s Roadmaps (two)

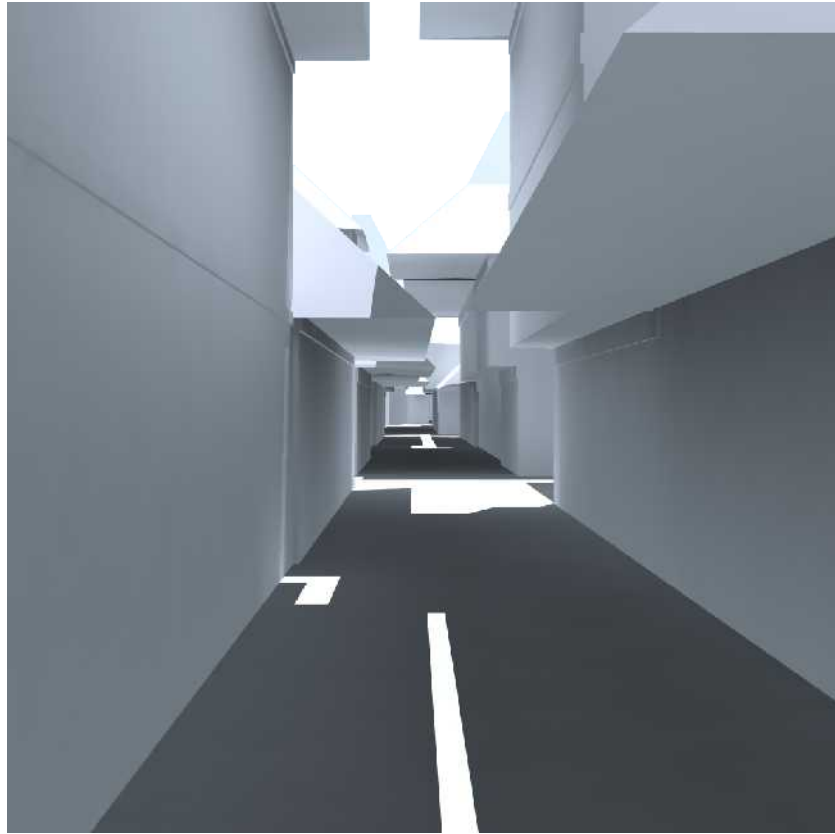
Generated Cities



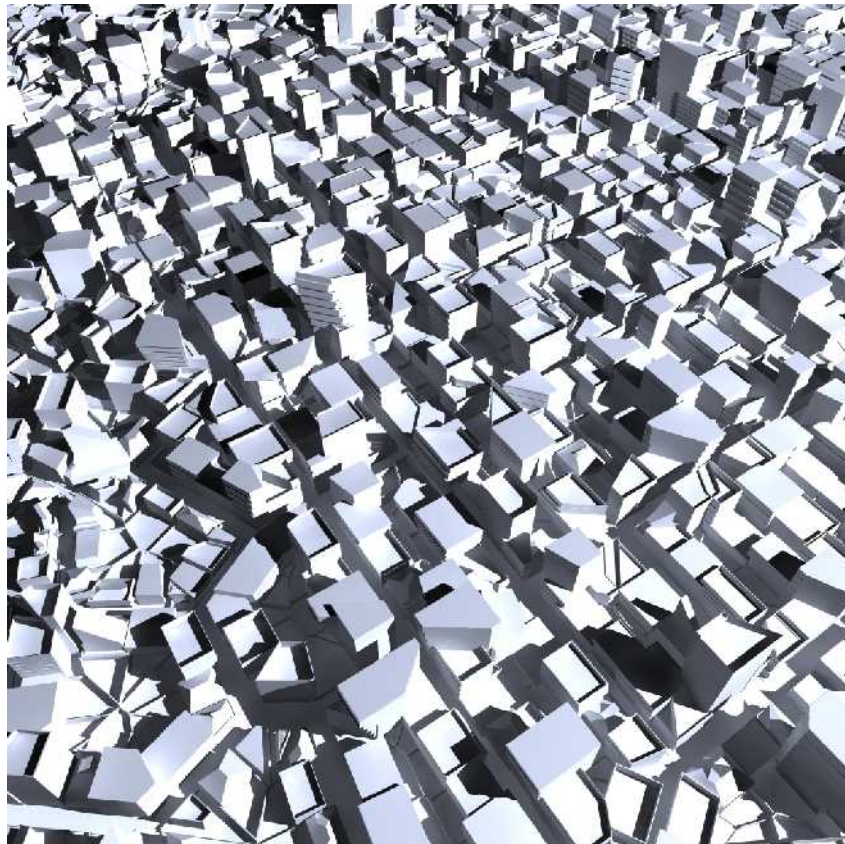
i)



ii)



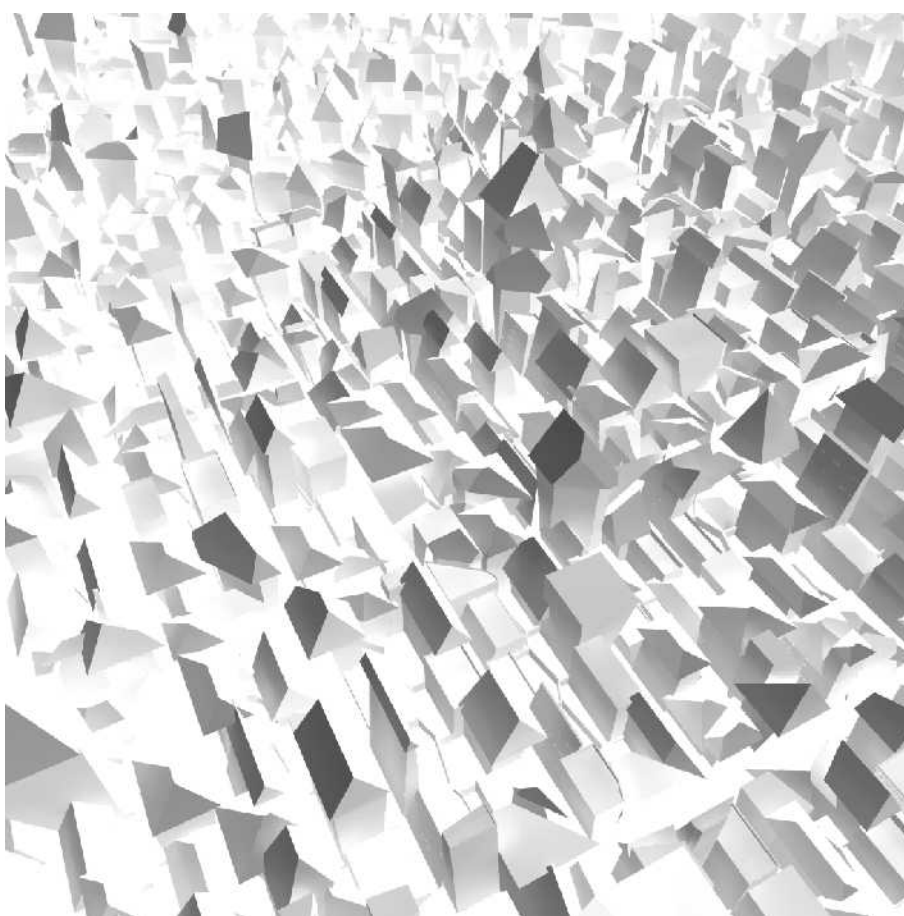
iii)



iv)



v)



vi)

i, ii) A city from different angles. There are many possible roof structures, with different combinations of gradients. The context grammar makes it simple to create a city with different looking buildings on different street layouts. Given regular input the Voronoi produces very lifelike designs with 3 or more streets often meeting at the same location. It is up to the artist designing the grammar to ensure that buildings are lifelike, for example large skyscrapers often have flat tops. Using the Gaussian to control features such as building heights leads to occasional extreme features such as very tall buildings.

iii) A negative weighting on the camp skeleton is used to generate eaves and building details.

iv) A city showing geometry around different block generators colliding. By cross linking the grammar it is possible to use building descriptions from one neighbourhood in another. Here two neighbourhoods share a definition of a skyscraper, one has family houses and one doesn't. This render also demonstrates different sized streets.

v) Roof sections. By changing the gradients used it is possible to present a large number of different types of roofs. By changing the probabilities for each production rule in the grammar I have made a flat roof less likely.

vi) These sections of city take about 50Mb on disk.

Bibliography

- ¹ Sity Business Plan by Tom Kelly (Author), submitted as part of the MEng qualification to Bristol computer Science, June 2006
- ² JMonkey Engine, scenegraph tool for Java by Mark Powell and Joshua Slack <http://www.jmonkeyengine.com/>
- ³ <http://www.idsoftware.com/> viewed 07 August 2006
- ⁴ <http://www.eidosinteractive.com/games/info.html?gmid=50> viewed 07 August 2006
- ⁵ <http://www.ghostrecon.com/uk/ghostrecon3/gameinfos.php> viewed 07 August 2006
- ⁶ “Exuberant youth to sustainable maturity”, Department of Trade and Industry, Spectrum, page 31
- ⁷ <http://www.binaryworlds.com/products.html> viewed 07 August 2006
- ⁸ <http://www.speedtree.com> viewed 08 August 2006
- ⁹ Automatic Generation of 3D city models and related applications - Y. Takase a, N. Sho, A. Sone, K. Shimiya, International Archives of Photogrammetry, Remote Sensing and Spatial Information Sci-ences, Vol. XXXIV-5/W10, 2003
- ¹⁰ Real-time Procedural Generation of 'Pseudo Infinite' Cities- Stefan Greuter, Jeremy Parker Nigel Stewart, Geoff Leach, GRAPHITE, 2003
- ¹¹ Rapid Procedural-Modelling of Architectural Structures -Birch, P.J., Browne, S.P., Jennings, V.J., Day, A.M. & Arnold, D.B. Virtual Reality, Archaeology, and Cultural Heritage (VAST2001 Proceedings): 187-196, 2001
- ¹² The automated generation of architectural form – Mitchell, WJ, Proceedings of the 8th workshop on Design automation, p 193-207, 1971
- ¹³ The Logic of Architecture William J. Mitchell, Massachusetts Institute of Technology 1944
- ¹⁴ A different Manhattan project: Automatic statistical model generation - Yap, C., B, Iermann, H., H & Ertzman, A., L, T Tech. rep., Courant Institute, New York University. <http://www.cs.nyu.edu/visual/> 2001.
- ¹⁵ How buildings learn – Stewart Brand 1994, Viking
- ¹⁶ Micro-Analytical Simulation of Transport, Employment and Residence – Mackett, R.L., Transport and Road Research Laboratory, 1990
- ¹⁷ Martin and Voorhees Associates, <http://www.mvaconsultancy.com/> viewed 07 August 2006
- ¹⁸ Procedural City Modeling - Lechner, T. et al.,1st Midwestern Graphics 2003.
- ¹⁹ . Procedural modelling of cities - Parish, Y. I. H. &Mueller, P.,Proceedings of ACM SIGGRAPH 2001, ACM Press/ ACM SIGGRAPH, New York, 301–308, 2001
- ²⁰ Instant Architecture – Wonka, P. et al, Siggraph 2003
- ²¹ Feature-based decomposition of façades – Finkenzeller D., Bender, J., Schmitt, A. Proc. Virtual Concept, Biarritz, 2005
- ²² Straight skeletons for general polygonal figures in the plane. - Aurenhammer F, etal, Proc. 2nd Ann. International Computing and Combinatorics Conf. COCOON'96, pg117-126, 1996.
- ²³ Raising roofs, crashing cycles, and playing pool - Eppstein, D. Discrete & Computational Geometry 22(4):569-592, 1999
- ²⁴ Straight Skeleton Implementation - Felkel, P. , Obdrzalek, S. 14thSpring Conference on Computer Graphics, April 23 -25, Budmerice, Slovakia, 1998
- ²⁵Automatically generating roof models from building footprints – Laycock RG, Day AM. - Proceedings of WSCG, Poster Presentation, 2003
- ²⁶ Nouvelles applications des parametres continus a la theorie des formes quadratiques. Voronoi, G.F. 136, p.67, 1909

- ²⁷ Duplicating road patterns in south african informal settlements using procedural techniques – Glass, K. Morkel, C. Shaun, D, Proceedings of the 4th international conference on Computer graphics, virtual reality, visualisation and interaction in Africa, 2006
- ²⁸ <http://maps.google.co.uk>, © Teleatlas. Viewed 07 August 2006 Page available from: <http://tinyurl.com/emvfb>
- ²⁹ <http://www.autodesk.com/alias> viewed 07 August 2006
- ³⁰ A Sweepline Algorithm for Voronoi diagrams – Forutne, S. in Algorithmica, Volume 2, Number 1 Pages 153 – 174, 1987
- ³¹ http://www.cip.ifi.lmu.de/~viermetz/cg/Voronoi_Algorithms.html viewed 09 August 2006
- ³² <https://jogl.dev.java.net/> viewed 08 August 2006
- ³³ [http:// java.sun.com/products/java-media/3D/](http://java.sun.com/products/java-media/3D/) viewed 08 August 2006
- ³⁴ <http://lwjgl.org> viewed 08 August 2006
- ³⁵ [http:// www.blender.org](http://www.blender.org) viewed 08 August 2006
- ³⁶ personal blog, not submitted as part of thesis: <http://twak.blogspot.com>
- ³⁷ Percolation and conduction on the 3D Voronoi and regular networks: a second case study in topological disorder – Jarauld, G. et al, Journal of Physics C: solid state Physics, 1984
- ³⁸ Generalized Two- and Three-Dimensional Clipping – Cyrus, M. Beck, J. Computers and Graphics, 3, pp. 23-28 1978
- ³⁹ <http://maps.google.com> viewed 21 August 2006