PROCEDURAL CITY DETAILING

Alex Mole

BSc (hons) Computer Games Technology

2004

School of Computing and Advanced Technologies

University of Abertay Dundee

Table of Contents

Declaration of Originality and Permission to Copy

UNIVERSITY
of
ABERTAY DUNDEE

Author	Alexander Mole
Title	Procedural City Generation
Qualification	BSc(hons) Computer Games Technology
Year of Submission	2004

I certify that the above mentioned report is my own original piece of work.

Signature ....

Permission is hereby granted to copy the whole or any part of the above mentioned report.

Signature ....

Date ....

Abstract

This project discusses a highly flexible method for generating city geometry from very low-detail input. Given a set of simple polyhedra (typically cuboids), the presented system procedurally generates geometry for architectural features such as windows, arranges the features on the walls, generates a roof for the building, and applies textures to them all. The system is structured around a core database of simple components (including executable objects, geometry and materials) that are loaded at run-time. The system is capable of generating hugely varied output from a small data-set by randomly combining appropriate objects from the database, resulting in a number of combinations that grows exponentially with each data module added. The project shows that such a system can be used to accelerate the process of creating virtual cities, targeting those used in game environments specifically.

1 Introduction

City-based games are currently very popular: each week, several of the best-selling games are city-based (ELSPA). As with any type of game environment, believable cities require a huge amount of detail, in terms of their buildings, roads and inhabitants. Games companies must invest a large amount of time and resources into designing, creating and testing these cities.

Because of the large costs involved, developers may be unable or unwilling to design particularly large and immersive cities. For the same reason, they may not be able to create the desired levels of detail in their cities, even if the game engine is capable of processing it. Additionally, due to game play elements such as physics of cars, certain areas of the environment may not be fun to play in. Such areas may be left unchanged due to the time involved with re-planning and reworking them.

A significant proportion of the effort involved in creating a city is spent on the buildings. Each one must be modelled and textured individually. Larger cities involve many hundreds of buildings, and this represents a significant amount of work.

In general however, buildings are very procedural entities. Most buildings have a relatively small set of architectural features (such as windows and doors) which are arranged according to straightforward rules (*e.g.* “doors must be at the bottom of walls”). The textures on buildings are also often simply tiled, with texture coordinates being directly related to the size of the wall.

1.1 Research question

The question this project sets out to address is:

“What is an appropriate strategy for producing a flexible system to procedurally add believable detail to virtual cities for use in computer games?”

1.2 Roadmap

Section 3 of this report discusses existing work in this field. Section 4 presents the design of the system. Section 5 examines the output of the system, and discusses pertinent design issues. Finally, section 6 discusses the success of the system and future work that could possibly be undertaken to extend and improve upon the techniques presented here.

The output generated by the system can be seen in appendix A, in terms of the manually created input and the generated output.

2 Literature review

2.1 Existing city creation systems

The conventional method for creating virtual cities is to model each building using a general-purpose editor such as 3D Studio Max (Discreet). This approach provides very fine control over the output, but is unable to take into account the similarities (and hence duplicated work) between buildings. Numerous approaches have been devised to reduce this workload.

When constructing virtual versions of real cities, as in The Getaway (Team Soho 2003), it is possible to make use of photogrammetric methods. Such methods rely on reconstructing geometry based on the analysis of photographs. There are two major branches in this field. The first involves analysing aerial photographs to find the outline of buildings and selecting the closest matching building from a library of standard models (Weidner 1995). The second requires each building to be photographed from a number of different angles. The building's outline is then found from this, and used to create the 3D geometry (Braun 1995). Both of these methods suffer accuracy problems as lighting affects the appearance of the buildings. Additionally, Weidner's method requires a comprehensive building model library to have been created before using the system. However, this photographic approach is not applicable to the majority of city-based computer games, as these tend to need custom cities designed for them.

Birch et al (2001) created a set of tools to aid in the construction of city geometry. These tools treated buildings and city scenery as the basic primitives and included comprehensive support for modifying these structures. Building models were created by arranging cuboids representing rooms and floors, and adding doors and roofs to them. These building models were then arranged within the city scene. The system had support for different levels of detail to improve rendering performance. It also featured a tool to generate a library of windows, stored as simple hierarchies of “panels”. This system succeeded in reducing the workload associated with producing city geometry. The creation of custom tools to edit cities resulted in a system where cities were both easy to model, and where subsequent modification of these buildings was well supported. However, this implementation was limited in terms of flexibility and extensibility, and as windows were chosen from a library, the city could exhibit some unwanted similarities between buildings.

Chen (1999) described a system for generating city geometry from building outline and height information. Textures and features (window and door models) were tiled across the faces of the buildings. The system selected the features from a library and the software supported several different types of window. The tiled features resulted in output that was aesthetically believable.

Paris (2001) created a system that generates cities using only terrain information as input. It used L-systems to generate street plans and to place and generate buildings. The buildings were textured by assembling images procedurally. This approach is capable of producing convincing cities with very little user input. It is however inappropriate for games applications, as it does not allow the required level of control over the output.

For most games applications, it seems that some form of procedural generation method is the most appropriate[1]. This provides good flexibility without the requirement for extensive real-world reference material. Two main approaches have been taken in the systems described above: either to operate directly on geometry and to apply detailing to that (Chen 1999, Paris 2001), or to design a new tool set to operate on buildings and architectural features as primitive entities, and to provide sophisticated operations for these primitives (Birch et al. 2001). The latter is clearly the most useful approach in the long-term, as it will make refining the city trivial, as well as providing interactive feedback about the city's appearance during the construction process. However, a large amount

1 For games whose levels are based upon real cities, it may be more expedient to make use of a photogrammetric method.

would need to be invested in the creation of such a system, in terms of the design and implementation of the tools, as well as the user training inherently required by any new editor. The former method has the considerable advantage of being able to fit into the existing work flow model. In addition, technologies developed as part of this system could later be incorporated into custom tool set solutions.

All of the procedural methods detailed above rely on large libraries of models. This is undesirable for two main reasons: all of this data must be created manually (which will be a time-consuming task), and there is likely to be a high level of repetition of features between buildings. This project examines a far more free-form and flexible approach, with the aim of addressing both of these issues.

The method used to describe windows in the system presented by Birch et al (2001) could be extended to a more general case with more levels in the hierarchy. This would be capable of describing a far greater number of structures, including a wider variety of windows, as well as doors and other architectural features. By using such a system coupled with random numbers, realistic features could be generated uniquely for each building.

2.2 Implementation considerations

In order to achieve the desired level of flexibility, the core of the system will need to be independent of the code that actually implements the detailing. Siddiqui (2003) discusses a design pattern involving orthogonal interfaces to classes, and the use of class factories to supply the system with class instances that can be defined outside of the core system. However, this limits the implementation to a single level of hierarchy, and as such is inappropriate for the complex structure required.

A more suitable solution could involve a self-referential structure with an implicit class factory between each internal link. Such a system could be implemented using a database containing executable code objects (Agrawal and Gehani 1989, Zaiane 1998) and data (such as materials) coupled with a pseudo-random selection mechanism to choose between specific objects. This database could be created at run-time, with code being loaded from self-registering modules. These executable modules would contain procedures for generating geometry, and would implicitly define the structure of the database.

The Python language (Python) would be an ideal platform for such a system, as it provides excellent support for introspection and late-binding. These are both features that would make the implementation of a system with run-time loading of executable modules simpler. In addition, Python code is easy to write and maintain, and as it uses garbage collection, the complex memory allocation issues inherent in such a system will not be an issue.

3 System Design and Implementation

3.1 Design Considerations

3.1.1 Quality of output

The primary requirement of this system was that the output had to be usable as a game environment. If the cities generated by the system were not of sufficient quality, the system would have been of little practical use. To this end, buildings needed to look believable, and their styles had to be appropriate for the game. The appearance of a given building needed to be consistent across itself. Ideally however, the system would produce no two buildings that looked identical, as this would reflect the architectural diversity of the real world.

3.1.2 Flexibility

The system needed to be capable of generating a wide range of styles of cities. The requirements in terms of city styles would vary enormously from game to game, so if the system could only generate a limited set of cities, it would be of little benefit. Furthermore, most city-based games feature areas from different neighbourhoods of a city (e.g. residential, industrial and office districts).

The system also needed to support easy extension and customisation. Games such as *Grand Theft Auto 3* (DMA Design Limited 2001), feature several different cities. Each has its own style, but there are a lot of shared features. Ideally the system had to be able to share data between cities as much as possible.

3.1.3 Efficiency of output

The generated geometry had to be tailored for rendering in real-time. It needed to be appropriately simple, with a minimum of unnecessary vertices, and in a format suitable for rendering APIs (*i.e.* triangles rather than general polygons, with texture information specified using texture coordinates). Additionally, texture usage had to be kept within reasonable limits to minimise or prevent the need for texture swapping at run-time.

3.1.4 Speed of execution

The system was intended to be an off-line process. In practice, a system such as this will not be run regularly, and it is feasible that, for large jobs (*e.g.* entire cities), the system would be run overnight. Because of this, the speed of execution is not the primary concern. However, the system would be of far greater use if it could be used interactively.

3.2 System Design

3.2.1 Overview

A heavily data-driven approach was taken in the design of the system. With the requirement for such flexibility, more conventional development strategies could have resulted in a very large code-base that would have been time-consuming to maintain. By treating much of the content as data, many dependencies were removed. To this end, much of the code was implemented as data modules (“plug-ins”) and was maintained externally to the core of the system.

Appendix B illustrates the operation of the program in the form of flowcharts.

3.2.1.1 Initialisation

The system reads the designer's input in the form of a scene full of blocks (see appendix A for examples). Each block represents the location, size and shape of a building. This scene can be created rapidly, and the block-based layout will allow the designer a reasonable insight into the appearance of the final level[2]. The system then loads a data set into a **`Registry`** database. This set includes materials and objects, as well as executable code objects. The **`Registry`** is common across all buildings.

Each of the blocks in the scene is then decomposed into a set of general polygonal faces. This is done by analysing the triangles in each block and finding those that are coplanar and joined by two vertices. Once a general polygon has been obtained, it it simplified if possible, by removing redundant vertices such as vertex *B* in Figure 1 (see section 4.1.1 for further discussion on this). Once simplified, the faces are classified according to their normals: faces whose normals point largely "up" (y-component is large and positive) are interpreted as roofs, those pointing "sideways" (y-component is small) as walls, and those pointing "down" (y-component is large and negative) as floors. These sets of polygons are then stored in a **`SceneTree`** object.

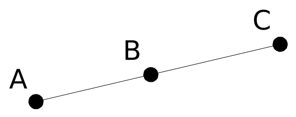


Figure 1: Vertex B is redundant

3.2.1.2 City Generation

Once the data has been loaded, the system iterates through the buildings in the **`SceneTree`**. For each one, a **`Style`** object is chosen from the **`Registry`** at random. The **`Style`** obtains an object in the **`Registry`** (by default "*/featureset*") and uses it to generate a **`FeatureSet`** (a collection of architectural features, materials and a roof generator function). The features within this set are generated procedurally by the code objects in the **`Registry`** (see section 3.2.2 for a description of how this is done).

Once a **`FeatureSet`** has been generated, the system iterates through the walls of the building and generates geometry to replace them. The style object arranges features in the first quadrant of the x-y plane, an area the same size and shape as the original wall. Once the features have been

2 In addition, this simple geometry could potentially serve as collision meshes.

placed, the empty space between them is filled with triangles (see section D.2 for a discussion of the implementation). The generated geometry is then transformed so that it occupies the same position as the original wall.

The system then calls the **FeatureSet**'s roof generation function to create geometry for the building's roof. The function generates geometry in the first quadrant of the x-z plane, in an area the same size and shape as the building's roof. Once this has been generated, the geometry is transformed to occupy the same space as the original roof polygon.

The generated geometry is then grouped into a single object and added to the output scene.

3.2.2 Feature Generation

One of the most important areas of the system is the procedural generation of architectural features. One of the biggest flaws of the existing city generation systems (see section 2.1) is the repetition throughout the city caused by the re-use of features between buildings. The generation of unique features for each building greatly increases the credibility of the cities.

Features are generated by a hierarchy of simple functions. Each is allotted an area in the first quadrant of the x-y plane to fill, and it can either create geometry to fill it, partition the space and call other functions to fill the space, or both. Figure 2 shows an example of how a window could be created using this technique, and section C.2 shows some Python modules that can be used as part of the hierarchy in Figure 2.

The dimensions of the features are then chosen for the particular building and style. In addition, the sizes of various parts of the generated geometry (*e.g.* the thickness of the window frames) is chosen at random for a given **FeatureSet**. Because of these factors, there is scope for a huge amount of variety amongst the generated features.

There may be a number of different possible functions at any level of the hierarchy. For instance, while Figure 2 shows a window divided into three

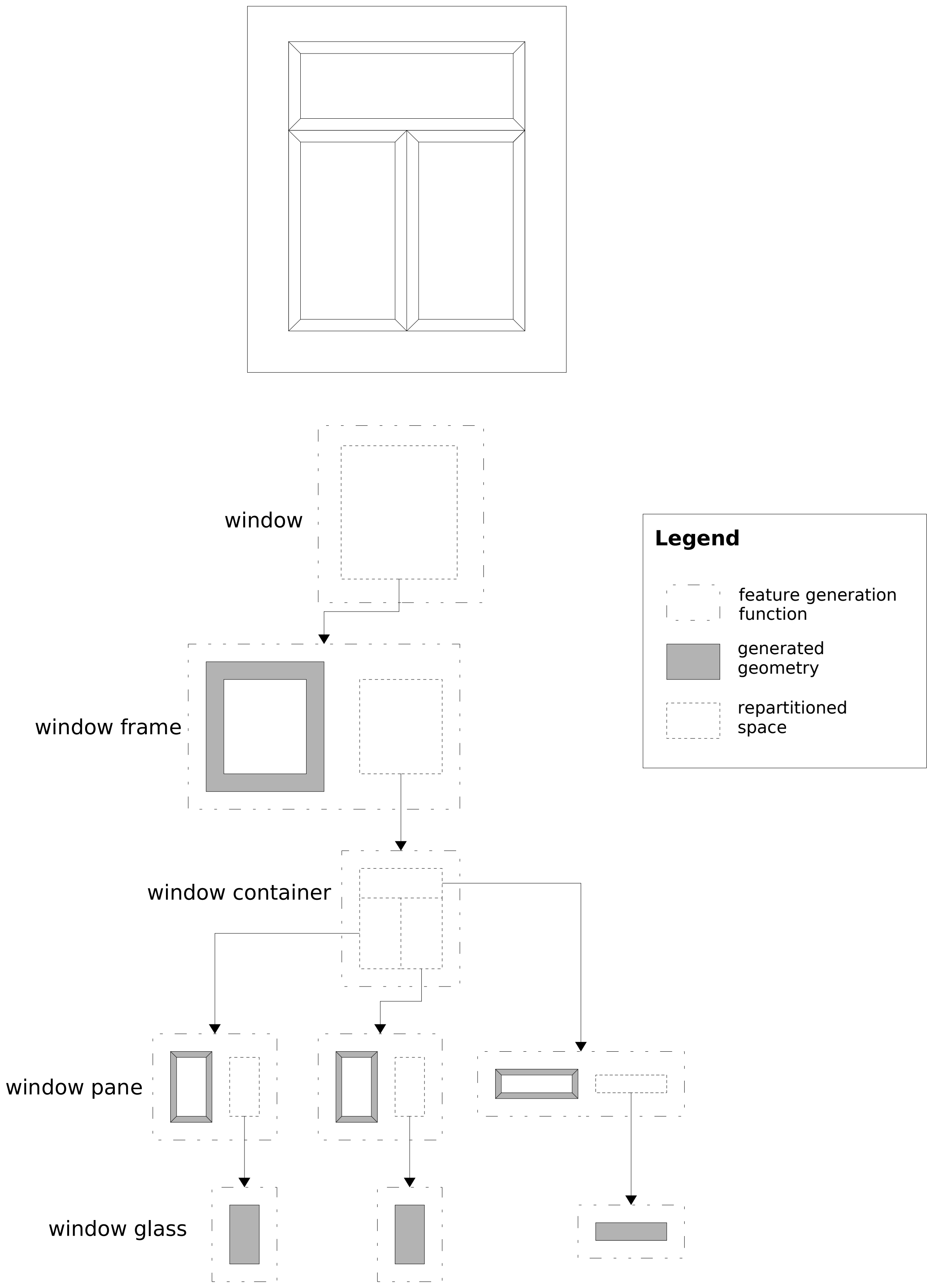


Figure 2: The feature generation process

panes, there may exist other “*window container*” functions which split the window into a different number of sections, or do not split it at all. A function is selected at random from the appropriate objects registered in the database. This significantly increases the number of different possible hierarchies, and hence increases the number of different features that the system can generate.

Care is taken to insure that by default, the same feature generation function is chosen for all branches of the tree at a given level. This ensures the features are consistent across themselves (*e.g.* in Figure 2, the three window panes must be generated in the same style). Section 3.2.4.2 discusses the implementation of this.

3.2.2.1 Texturing the features

In conventional level design, architectural features are usually modelled as separate objects from the main building, with a custom texture created for each. The same features are generally repeated on several buildings across the city, and the same texture could be used on different but similar geometry. Hence each feature texture is shared amongst many buildings.

However this type of approach is impractical when procedurally generating the features to this degree: there is an extremely large number of possible features, most of which would not be able to share textures. In order to overcome this, each feature is composed of several objects with different standard textures. For instance, in Figure 2, there would most probably be three objects, one for the window frame, one for the panes and one for the glass. When the features are applied to a building, the objects are merged with the others using the same textures.

It is likely that this approach is more efficient for rendering than the conventional model, as fewer textures will be used overall, so texture swapping should be reduced. In addition, the texture set would consume less video RAM, allowing an increase in the quality or quantity of the textures used.

3.2.3 Roof Generation

The system takes the general polygon produced by aggregating the triangles whose normals point “up”, and generates roof geometry to fill this polygon. The implementation passes the polygon outline to a function that returns geometrical data. This simple interface allows a high degree of flexibility for the roof generation. For instance, roof generation functions are able to add stock scenery objects (such as air conditioning vents) and to procedurally generate features of their own. Section C.3 shows the Python source for an example roof generation function.

3.2.4 Implementation

3.2.4.1 The Registry

The core of the system is a database of objects known as the **`Registry`**. This database is organised like a file-system, but rather than file data, its nodes are Python objects. The “path” to an object is its type (e.g. '*/materials/brick*' or '*/features/window/frames*'), and all objects of the same type have the same interface (this is required for the system to work). In addition to storing an object, each node may have metadata[3] (Day 2001) associated with it. This metadata is used both for storing extra information about objects, and for accessing and changing parameters to objects.

The registry is populated at run-time with objects loaded from file. Many of these objects are instances of custom classes defined in “data” code modules. These objects implement a standard set of methods which inform the core system of their desired path in the registry as well as any metadata they have.

One special type of object metadata is the **socket**. Due to the data-driven design of the system, the core code is required to know as little as possible about the structure of the registry (in order to achieve the desired level of flexibility). Therefore the objects in the registry have to “know” how to use each other. For instance, architecture style classes “know” what kinds of material should be used for walls and where the

3 Metadata is data about data.

materials are stored in the **`Registry`**. These types of connections are implemented using sockets. In this case, the **`Style`** may have a socket called "*matwall*" of type "*/materials/walls*". For each generation set, an appropriate object is selected for the socket, and this is used to texture the walls. There are two types of socket: global and local. While most sockets can be treated entirely independently, certain cases require co-ordination across the building (this particularly applies to the selection of certain materials[4]). Global sockets are resolved once for each building, insuring visual consistency.

3.2.4.2 Random numbers

The system makes heavy use of pseudo-random numbers. They are used to select items from the registry, as well as to choose dimensions for procedural geometry (*e.g.* the thickness of a window frame). Python supplies a high quality pseudo-random number generator, and this is used to supply the numbers.

However, it is important that certain results can be repeated. For instance, for any building, each time a given feature generation function's socket is resolved, it must return the same object. If this was not the case, there would be little consistency across the building and the visual quality would be very low.

In order to address this, the system makes use of a persistent random number generator class, a new instance of which is created for each building. This returns values for named requests. If a request for a random number with a particular name hasn't been made before, a value is generated and stored. Any subsequent calls for that name will return the original value.

3.2.4.3 Style classes

Once a building object has been decomposed into faces, a **`Style`** class is selected from the registry. The **`Style`** class describes a particular architectural style (for instance, there may be a class for buildings with

4 For instance, if a roof has a wall around it, it must have the same material as the walls of the building.

shop signs between the ground and first floors, and another class for warehouse-style buildings with very large doors at one end). As with any object stored in the registry, **`Style`** classes can have sockets.

The system incorporates a comprehensive base **`Style`** class, which implements a large amount of functionality, allowing for thin derived classes for individual architectural styles. This standard base class implements *geometry tiling* (Chen 1999), dividing each wall into regular grid cells and calling a virtual method to select a feature to fill each cell with (derived **`Style`** classes override this method to realise their own architectural style). However, any class exposing a small number of interface methods can be used as a **`Style`** class, so different building generation techniques could be used without the need to alter the core system.

The standard **`Style`** class requests the creation of its set of features from a special object stored in the **`Registry`**, "*/featureset*". This object is the standard hook into the rest of the **`Registry`**'s structure, effectively translating standard feature names (such as "*window*") into **`Registry`** types (such as "*/features/windows*"), allowing more complete decoupling of code and data (and hence flexibility). Derived **`Style`** classes request features (and their sizes), materials and a roof function from the "*/featureset*" object. This then calls the relevant feature generation functions to create geometry for these features.

Section C.1 shows the Python code for an example **`Style`** class derived from the standard base class.

3.2.4.4 Wall generation

The system detects walls as rectangular faces parallel to the y-axis. It then divides the wall into equally-sized cells. The size of each cell is calculated to be the smallest possible that can fit the largest feature while maintaining an integral number of cells in each direction. Each cell then has a feature selected for it by the building's **`Style`** class.

Each feature has associated "padding" and "justification" values (see

Figure 3). The padding value specifies the minimum distances from the four edges of the feature's bounding box and the justification specifies where in a grid cell the feature should be anchored to. These two pieces of information provide a simple but flexible way of describing positions of features within cells that will scale to cells of any size.

This approach makes the creation of **`Style`** classes trivial; a subclass need only implement a virtual method to select a feature for a given cell. The method is passed the cell's coordinates (in cell-space), so that it can, for instance, handle cells on the ground floor differently (*e.g*. by placing a door). Section 4.1.2 discusses this approach and its limitations.

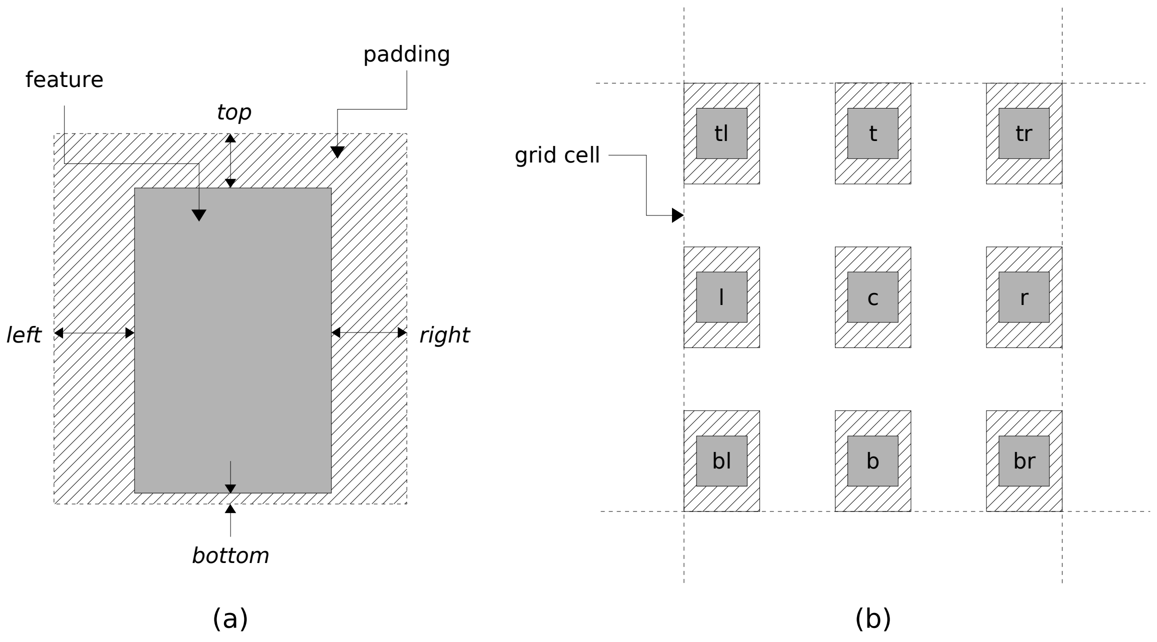


Figure 3: Feature padding and justification. (a) shows how a feature's padding is described, and (b) shows where the various justifications anchor to (for clarity , the features are artificially small).

4 Results

Appendix A shows examples of the cities produced by the system.

These buildings could plausibly be used within computer game environments. There are no misplaced features, and the style of the city is consistent without being repetitive.

The input data is easy to create. As it closely relates to the output, the designer has good interactive feedback allowing them to accurately plan the environment. As the process of generating the city from this data is so computationally cheap, the designer can also quickly identify areas needing refinement and modify them accordingly.

The system takes approximately 1.5 seconds to generate each building. This speed is within acceptable limits: individual buildings and small areas can be regenerated without disrupting work flow at all, and larger cities involving several hundreds of buildings can be processed in under an hour.

The core system is very flexible. Entirely different styles of city could be generated without modifying the core system in any way. This is due to the data-driven approach used; the system offloads a small amount of extra knowledge onto its data, and the result is that by changing the data, its operation can be drastically altered.

Due to the random manner in which the system combines the functions in the feature generation hierarchy, coupled with the use of random numbers to decide dimensions of specific details, a small set of feature generation functions can produce an almost unlimited number of different features. This was the intention of the design, and it has been very successful: e.g. by adding a new “window container” (typically a trivial task), a whole new class of windows is generated. However, it does also lead to a lack of control over the output, which means that the system is likely to occasionally generate nonsensical geometry. Section 4.1.3 discusses this issue in more detail. The problem can be overcome to some extent by careful arrangement of the Registry, but the problem is inherent in a system solely reliant on randomness to select children for each node in the hierarchy. Ideally, some form of feedback is required, so that when certain feature generation functions are selected, the system adjusts the probabilities of other functions being chosen later. One possible approach to this is presented in section D.1.

4.1 Discussion

4.1.1 Issues involved with geometry generation

In any system that procedurally generates geometry, certain issues inevitably arise.

4.1.1.1 “T-junctions”

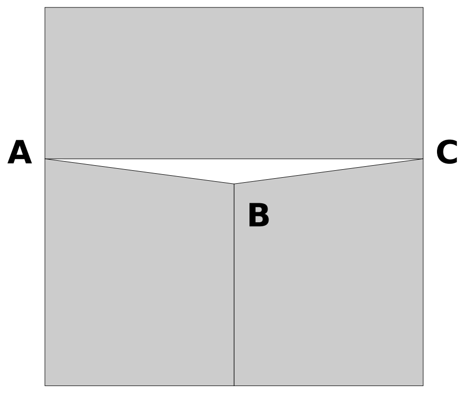


Figure 4: An exaggerated T-junction. Typically ABC would be near-linear.

When generating geometry that must meet at some edge *AC*, it is possible that the geometry on one side passes through an intermediate vertex *B* (see Figure 4). Due to errors in numerical precision, it is likely that *B* will not lie exactly on the line *AC*. This structure can lead to visual artefacts (Akenine-Möller and Haines 2002, p.448). The solution to this problem is to divide the edge *AC* and the adjoining geometry so that it goes via vertex B. This involves dividing a quad into three triangles, or a triangle into two triangles for each T-junction along an edge, so a substantial amount of geometry may be added by this process.

It is possible to detect and correct these cases after geometry has been generated. However, the processing load involved is high, as each edge must be tested against every other edge in the world. An alternative solution would be to mandate that each geometry-producing element in the system constructs appropriately bridged geometry. This would represent a significant increase in programming effort, but means that the extra geometry would be created by the code with the most “understanding” of it (which could improve areas such as texture coordinate calculation). In addition, by producing the extra geometry at this stage, it is possible that the solution will be more efficient (in terms of number of triangles) than the output of a general algorithm.

4.1.1.2 Texturing

The system assigns texture coordinates for vertices based on the dimensions of each face. However, it assumes that the texture should repeat once per world unit (this commonly represents one metre or one yard). This is an oversimplification: textures may tile over any arbitrary distance. Indeed, tiling a texture once per metre leads to unpleasant visual artefacts.

To overcome this, each material should be defined as having world-space dimensions, and the system should provide a standard method for geometry generation functions to map between the world-space and this appropriately scaled texture-space.

4.1.2 Strategies for feature placement

The approach to placing features on a wall involved dividing the wall into a number of equally sized cells and treating each cell as a discrete case. However, buildings in both the real world and in games typically have more sophisticated algorithms. For instance, it is common to have ledges running the width of the building. While it is possible to write a **`Style`** class which can apply extra geometry across a range of cells[5], it will be limited.

An alternative approach to the problem would be to hierarchically divide

5 The system's “data-ny-res” dataset includes such a **`Style`**, and ledges can be seen in the pictures in Appendix A.

the face in a similar manner to the one in which features are generated. The concepts of floors could be used to partition each wall vertically, and each floor could be partitioned into cells into which geometry such as windows could be placed. This method would allow for floors to be of different heights, and for more interesting structures to be generated across the wall. The existing repeatable random number generator (see section 3.2.4.2) would automatically allow floors of the same type to be laid out in the same manner, providing the necessary coherence across the wall.

An extension of this concept would be to treat all the walls as part of the same object (as opposed to treating each wall entirely independently as is the case in the current system). The walls could be considered to be a level higher in the hierarchy than the floors, allowing for the same level of coherence between walls as would exist between floors.

4.1.3 Lack of control

Because the system uses randomness in as many places as possible to increase the variety in the output, it is possible that it will combine inappropriate objects (this is discussed further in section 5.1). One example of this is illustrated here.

Given the function hierarchy presented in Figure 2, one can imagine a "*window pane*" function which is undecorated and simply passes through to a "*window glass*" function. This could result in a window that is subdivided into three sections, but with no obvious geometry to separate them, as shown in Figure 5 (a).

This has two negative effects. It creates extra, unnecessary geometry, which may have an adverse impact on rendering speed (if this error is common across the city). Additionally, it causes the visual artefact shown in Figure 5 (b).

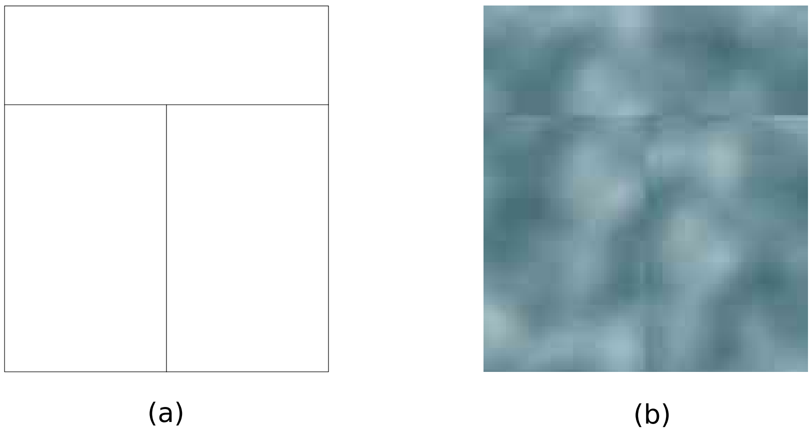


Figure 5: Badly chosen feature generation functions. (a) shows the geometry and (b) shows the texturing artifact.

To combat this, the **`Registry`** needs to be organised carefully. Listing 1 shows a **`Registry`** arrangement which has a type called "*/features/window/pane/nontrivial*", which stores only window panes that have some form of border. Any "*window container*" that partitions the window uses this type instead of "*/features/window/pane*".

Note that if a request is made for an object of type "*/features/window/pane*", the contents of the "*nontrivial*" sub-type will be included in the selection set.

Section D.1 presents a better potential solution to this problem.

4.1.4 Architecture description language

Rather than using executable Python objects to describe architectural styles, it would be possible to design a mini-language. Such an approach is often adopted for problems with a relatively small domain such as this, and has several real advantages over using a general programming language (Raymond 2004, p.183).

For instance, it would be possible to use the language to define a set of rules to decide upon which feature should be used to fill a given grid cell on a wall. The language could support complex objects such as features as primitives, and could supply operators for common operations such as geometry transformation and random number generation.

However, such a language would need to be very complex in order for the language to be able to produce useful output without restricting the possible set of operations. A substantial amount of design work would be required for such a language, and the benefits over using a general programming language with a comprehensive library of helper functions would be minimal. Additionally, with any new language, there is an associated time penalty as each developer must learn to use it.

5 Conclusions and Recommendations

5.1 Conclusions

This project presented a system which takes an input in the form of a city laid out with simple blocks, and generates an output where each block has been replaced by a building of the same size, with a roof and architectural features, and which has been textured. The architectural features are procedurally generated, so each building has its own set.

In order to maximise flexibility, the system was designed to have a clear distinction between the core system and the data. Much of the data was actually executable Python code[6] which is loaded at run-time and is stored into a database. Through the use of strict interfaces, the code objects in the database form a sophisticated data structure composed of simple elements, each of which is easy to write and maintain, and the sum of which is capable of producing an almost unlimited amount of different geometry. For each feature, a hierarchy of objects (including executable ones) is constructed by randomly selecting appropriate elements from the database (see Figure 2 for an illustration of this).

The principle issue with the system is one of control. With such an emphasis placed on random numbers for selection, the database must be organised very carefully to avoid inappropriate combinations of elements. Section D.1 presents a possible solution to this, but it is perhaps an issue inherent in the hierarchical arrangement of nodes used in the database. It is possible that a more descriptive form of database organisation, with selection based on queries rather than “paths” (Reiser 2001), could alleviate a substantial amount of the control problem, but the database technology required for such a solution is non-trivial.

At a higher level, the cities produced are of a useful standard, but the system does not produce buildings that can practically be used for any more than scenery. The buildings are not aesthetically exciting, merely believable. This is perhaps inherent in the problem to a degree;

6 In terms of number of lines of code, approximately 20% of the python source is classified as data.

architecture is an art form. In order to produce beautiful buildings, a great deal of factors must be taken into account, including the purpose of a building, its surroundings and the history of the area and of architecture, as well as more nebulous aesthetic concepts.

5.2 Future Work

A number of refinements to the current system are presented in Appendix D. These concern aspects of the current system that testing have shown to be lacking (see discussion in section 4.1). This section presents additional, more substantial changes which would require significant reworking of the system.

5.2.1 Levels of Detail

One of the areas of real potential for a procedural city creation system is the generation of multiple levels of detail (Chen 1999, Birch et al. 2001). Contemporary games make heavy use of this technique, as it can dramatically increase the amount of objects on screen, while maintaining a viable rendering load (Akenine-Möller and Haines 2002, pp 389-401).

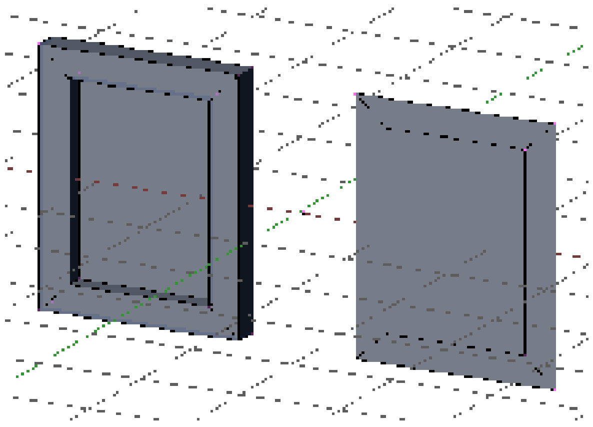


Figure 6: Simplifying geometry by removing depth

Rather than generate a single geometry set for a given feature, the system could generate a number of sets at different levels of detail (henceforth: LOD). For instance, a method of reducing geometry for a window would be to remove any depth from it (see Figure 6). An even lower LOD could be created by rendering the feature to a texture and replacing each instance of the feature with single quad with that texture.

5.2.2 Improved editing support

The system generates geometry for buildings once. All of the parameters for the appearance are selected randomly, used to generate the geometry, and then discarded. This means that there is no way for an artist to adjust some small parameter of the building; if the building is unsatisfactory, it must be regenerated.

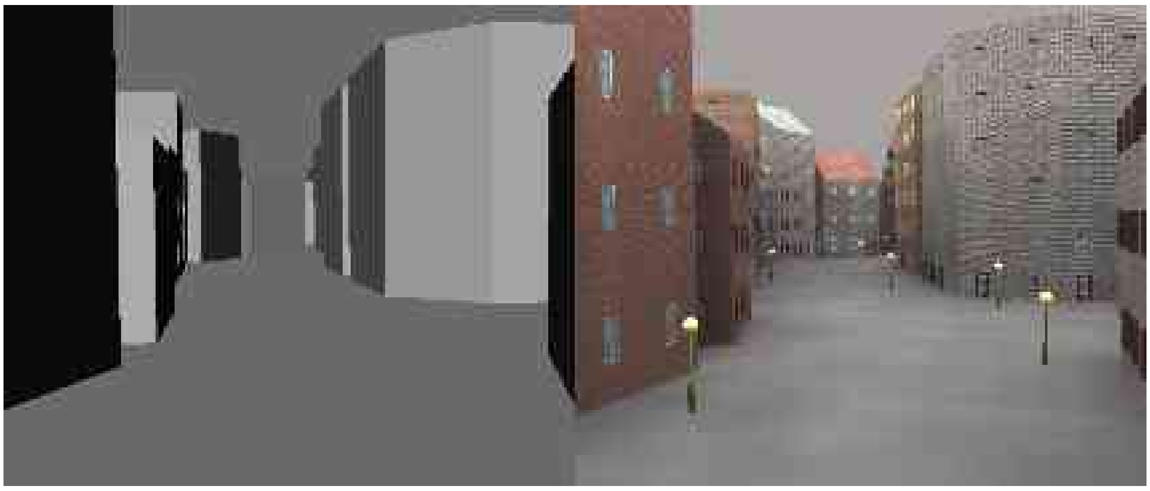
A far better approach in terms of work flow would be for the system to work in two phases: the first would randomly generate the parameter sets for each building (including choosing feature generation functions, **`Style`** classes, and materials), and the second could at any time convert the set into geometry. This would allow the second phase to be separated altogether and embedded within a level editor program, along with an interface to allow the artist to override any of the randomly generated parameters and see its effects interactively. Such a system would allow the maximum control over the output without losing the benefits of large-scale automation, and would represent a significant tool for game environment creation.

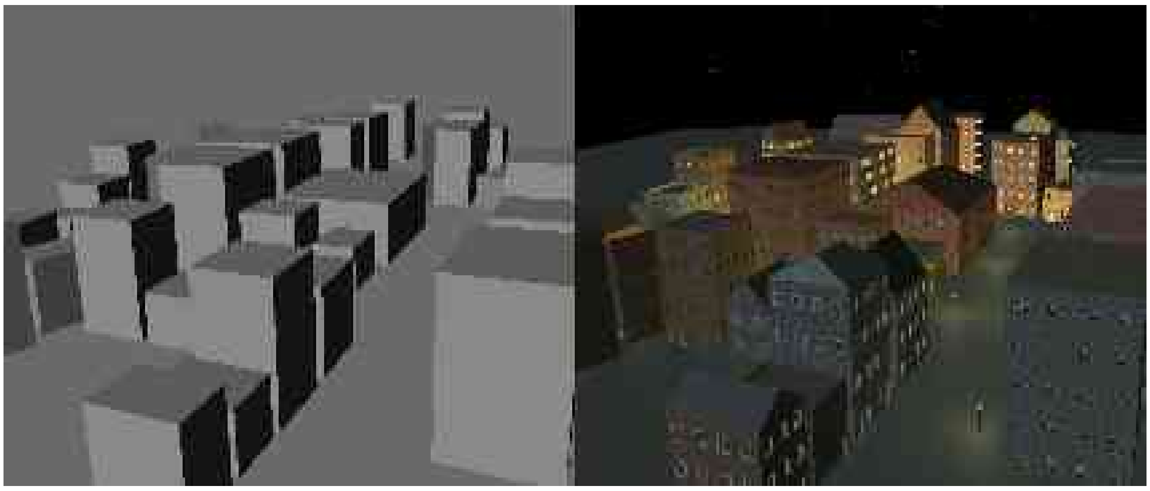
Appendices

A Example Output

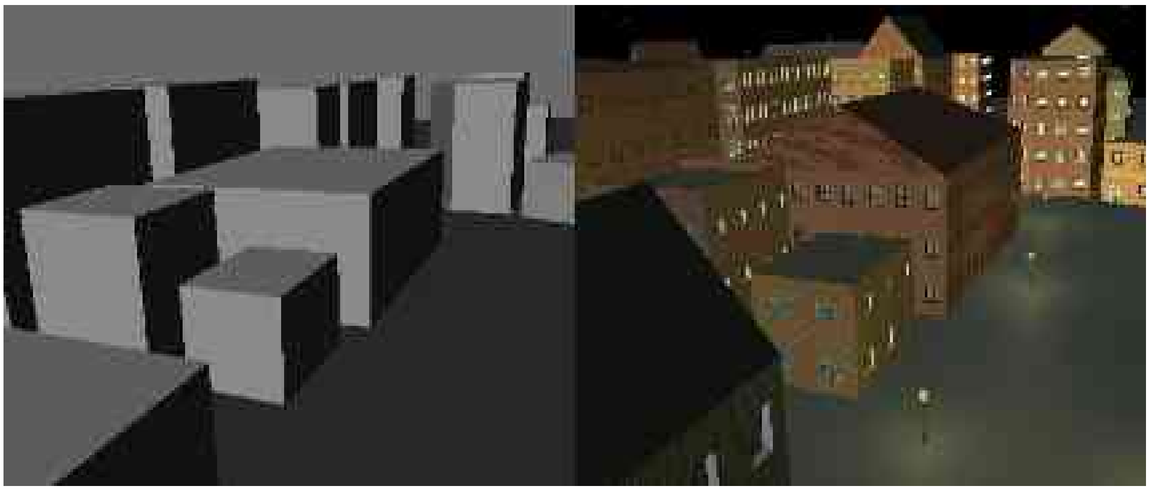
This appendix shows rendered images of the buildings generated by the system. The images on the left are the low-detail input that has been created by hand, and the images on the right show the system's output[7].

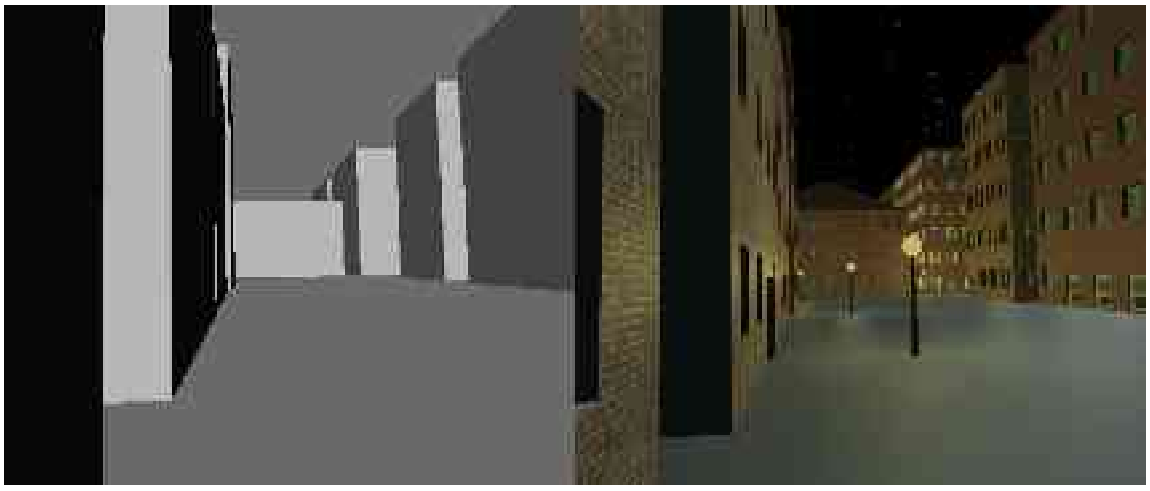
Listing 1 is a listing of the contents of the **`Registry`** that was used to generate these buildings. Note how few objects are required to generate such a varied output.





7 Note that the streetlamp models and the lighting were added manually afterwards.





```
/featureset
/features/door/frames/box3
/features/door/frames/simple-recessed
/features/door/objects/simple
/features/doors/basic
/features/ledges/basic
/features/window/containers/basic
/features/window/containers/four-sections
/features/window/containers/three-sections
/features/window/frames/box
/features/window/frames/simple-recessed
/features/window/glass/basic
/features/window/panes/basic
/features/window/panes/nontrivial/bevel-frame
/features/window/panes/nontrivial/thin-frame
/features/windows/basic
/materials/basic/dark-brown
/materials/basic/white
/materials/doors/plain
/materials/glass/0
/materials/glass/1
/materials/ledge/0
/materials/roof/flat/0
/materials/roof/flat/1
/materials/roof/tiled/0
/materials/roof/tiled/1
/materials/roof/tiled/2
/materials/roof/tiled/3
/materials/roof/tiled/4
/materials/roof/tiled/5
/materials/roof/tiled/6
/materials/roof/tiled/7
/materials/roof/tiled/8
/materials/wall/brick/0
/materials/wall/brick/1
/materials/wall/brick/2
/materials/wall/brick/3
/objects/decoration/roof/flat/aircon-vent
/objects/decoration/roof/flat/skylight0
/roofs/apex-simple
/roofs/apex-sloped
/roofs/flat-simple
/roofs/flat-walled
/styles/basic-ledge
/styles/multi-door
/styles/multi-door-2
/styles/simple
```

Listing 1: The contents of the **`Registry`** *that was used to generate the above buildings.*

B Diagrams

B.1 Process Flowcharts

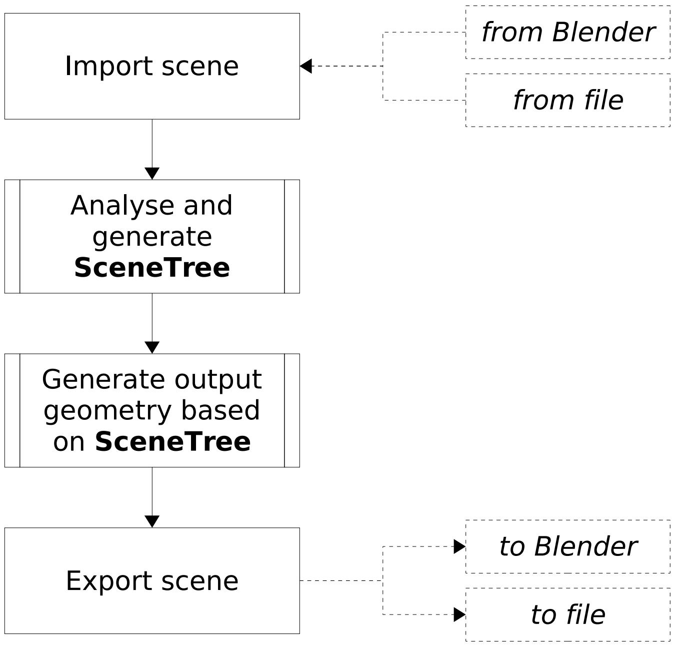

Import scene
from Blender
from file
Analyse and generate **SceneTree**
Generate output geometry based on **SceneTree**
Export scene
to Blender
to file

Figure 7: System overview

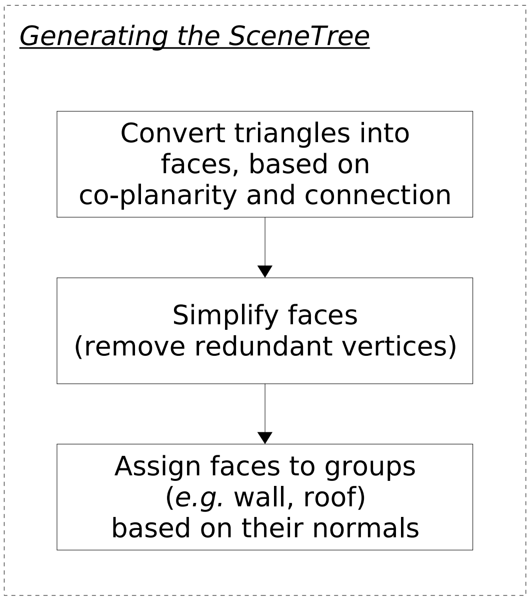

Generating the SceneTree
Convert triangles into faces, based on co-planarity and connection
Simplify faces (remove redundant vertices)
Assign faces to groups (*e.g.* wall, roof) based on their normals

Figure 8: Analysing geometry and generating the SceneTree

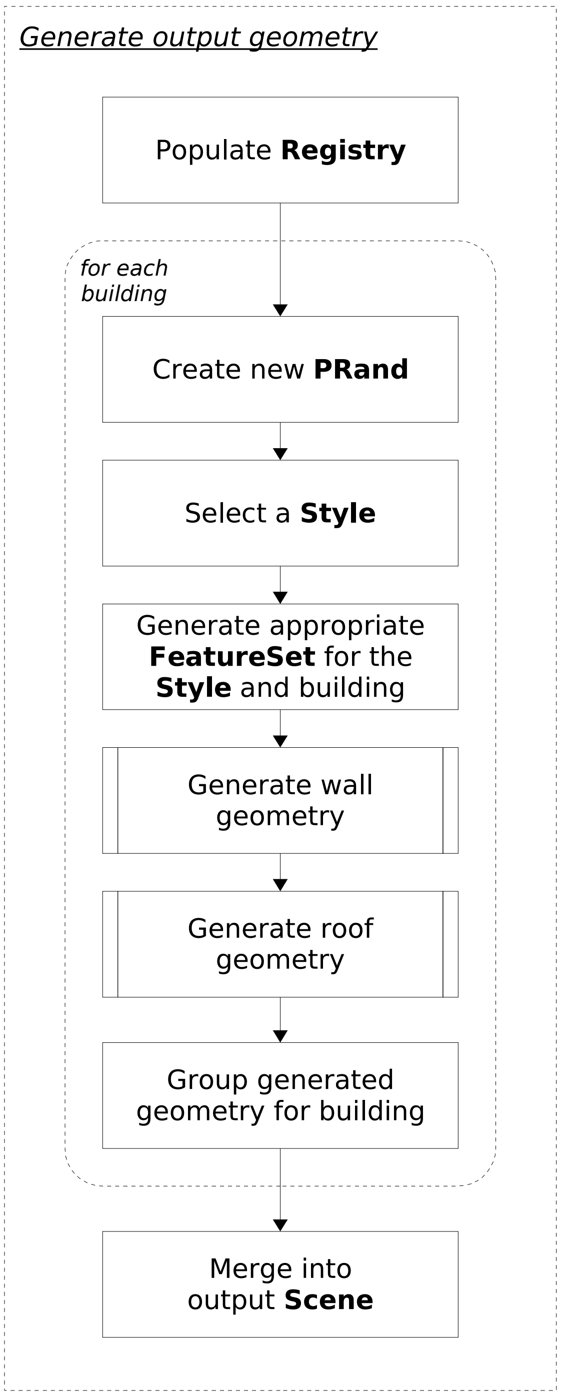


Figure 9: Generating the output geometry

C Example Code

C.1 Example Style Class

```python
import Architecture, Architecture.Style
import random


class SimpleStyle (Architecture.Style.Style):
    "Simple style with doors on the front and back"

    def __init__ (self):
        Architecture.Style.Style.__init__ (self)
        self._regname = 'simple'

    def _getFeatureList (self):
        win = (random.gauss (1.3,0.2), random.gauss (1.6,0.1))
        door = (random.gauss (1.2,0.2), random.gauss (2.1,0.2))
        return [ ('window','window',win), ('door','door',door),
                 ('roof','roof', None) ]

    def _startFace (self, facetype):
        if facetype == Architecture.MISCFACE:
            # no doors
            self.doorcol = -1
        else:
            self.doorcol = random.randrange (self.ncells[0])

    def _fillCell (self, row, col, cellbase, facetype):
        if row==0 and col==self.doorcol:
            self.addFeature ('door', cellbase)
        else:
            self.addFeature ('window', cellbase)


def Register ():
    return [ SimpleStyle() ]
```

Python module defining a simple building style class that has a single door on the front and a single door on the back of the building, Note that the system is unable to distinguish between these types of face, and a random decision is used to choose the "type" of a face.

C.2 Example Feature Generation Functions

These are examples of the Python code needed to create features. These examples are all used to generate windows; Figure 4 illustrates where each of these classes lie in the window hierarchy.

C.2.1 Top-level window class

```
from Architecture import FeatGen, Feature
import random


class BasicWindowFunc (FeatGen.Func):
    "Basic window type- just passes straight through to frame"

    def __init__ (self):
        self._regpath = '/features/windows'
        self._regname = 'basic'
        self._regsockets = [ ('frame','/features/window/frames') ]


    def generateContent (self, size, registry, prand, level):
        obj = self.socketGenContent ('frame', size, registry,
                                                        prand, level)
        hpad = size[0] * random.uniform (0.4,0.6)
        vpad = size[1] * random.uniform (0.4,0.6)
        feat = Feature.New ('window', obj, 'c', (vpad,hpad))
        return feat


def Register ():
    return [ BasicWindowFunc() ]
```

Python module for generating basic windows. This code passes directly through to its single child, a window frame function. Note that as this is a top-level feature generation function, it returns a Feature object with associated padding and justification information instead of the more common Object or MetaObject.

C.2.2 Window container class

```
import Object, MetaObject
from Architecture import FeatGen
from Maths.Vector3D import Vec3


class FourSectionsWinContFunc (FeatGen.Func):
    "Divides the window vertically and horizontally into a 2x2 grid"

    def __init__ (self):
        self._regpath = '/features/window/containers'
        self._regname = 'four-sections'
        self._regsockets =
     [ ('pane','/features/window/panes/nontrivial') ]


    def generateContent (self, size, registry, prand, level):

        w,h = size
        hw = w / 2.0
        hh = h / 2.0
        psize = (hw,hh)

        bl = self.socketGenContent ('pane', psize, registry,
                                               prand, level)
        br = self.socketGenContent ('pane', psize, registry,
                                               prand, level, (hw,0,0))
        tl = self.socketGenContent ('pane', psize, registry,
                                               prand, level, (0,hh,0))
        tr = self.socketGenContent ('pane', psize, registry,
                                               prand, level, (hw,hh,0))

        all = MetaObject.New ()
        all.addObjectData (bl)
        all.addObjectData (br)
        all.addObjectData (tl)
        all.addObjectData (tr)

        return all


def Register ():
    return [ FourSectionsWinContFunc() ]
```

Python module for partitioning a window into four sections. This class just repartitions the space and calls children to create the geometry.

C.2.3 Window pane class

```
import MetaObject, Object
from Architecture import FeatGen
from Maths.Vector3D import Vec3


class BevelFramePaneFunc (FeatGen.Func):
    "Pane with a thin beveled border around it"

    def __init__ (self):
        self._regpath = '/features/window/panes/nontrivial'
        self._regname = 'bevel-frame'
        self._regsockets = [ ('glass','/features/window/glass'),
                             ('mat', '/materials/basic')            ]


    def generateContent (self, size, registry, prand, level):

        w,h = size
        t = prand.val ('paneframe-thickness', 0.02, 0.06)
        d = prand.val ('paneframe-depth', -0.02, -0.06)

        # generate some verts
        ov = [ Vec3(0,0,0), Vec3(0,h,0), Vec3(w,h,0), Vec3(w,0,0) ]
        iv = [ Vec3(t,t,d), Vec3(t,h-t,d),
                                   Vec3(w-t,h-t,d), Vec3(w-t,t,d) ]

        # ...and some texcoords
        T = lambda v: (v.x,v.y)
        ot = map (T, ov)
        it = map (T, iv)

        frame = Object.New ()

        robj = self.chooseSocketValue ('mat', registry, prand)
        frame.material = robj.getObject ()

        self._genQuadStrip (frame, ov, iv, ot, it)

        gsize = (w-2*t,h-2*t)
        glass = self.socketGenContent ('glass', gsize, registry,
                                                prand, level, (t,t,d))

        all = MetaObject.New ()
        all.addObjectData (frame)
        all.addObjectData (glass)

        return all


def Register ():
    return [ BevelFramePaneFunc() ]
```

Python module to create a window pane with a bevelled recessed border. Note that this creates geometry for the border and then combines it with that created by its child window glass function.

C.3 Example Roof Generation Function

```
from Architecture import Roof
import math, operator, random
import Object, MetaObject
from Maths.Vector3D import Vec3
import Maths.Util


class Simple (Roof.Func):
    "Totally flat roof"

    def __init__ (self):
        self._regpath = '/roofs'
        self._regname = 'flat-simple'
        self._regsockets = [ ('mat', '/materials/roof/flat'),
                             ('decor', '/objects/decoration/roof/flat')
                           ]

    def __genDecor (self, face, registry, prand):
        “Place any décor and return it”

        i = random.randrange (len (face.verts))
        ni = (i+1) % len (face.verts)

        e = face.verts[ni] - face.verts[i]
        n = e.cross (face.normal)
        n.normalise()
        el = e.length ()

        obj = self.chooseSocketValue ('decor',
                                      registry, prand).getObject ()

        if el < obj.getWidth():
            # we can't add decor here, it's too small
            return Object()

        s = random.uniform (0.2, 0.8)
        t = random.uniform (-3.5, -1.5)

        pos = s*e + t*n + face.verts[i]
        yrot = math.atan2 (n.x, n.z)

        xform = Maths.Util.MakeXForm (pos, yrot)

        decor = obj.clone ()
        decor.applyTransformFunc (xform)

        return décor
```

```
def generateContent (self, face, registry, prand):

        o = Object.New ('roof')
        robj = self.chooseSocketValue ('mat', registry, prand)
        o.material = robj.getObject ()

        verts = face.verts
        centroid = reduce (operator.add, verts) / float(len (verts))

        for i in range (len (verts)):
            ni = (i+1) % len(verts)
            t0 = (verts[i].x - centroid.x, verts[i].z - centroid.z)
            t1 = (verts[ni].x - centroid.x, verts[ni].z - centroid.z)
            o.addTri ((verts[i], verts[ni], centroid), (t0, t1, (0,0)))

        roof = MetaObject.New ()
        roof.addObjectData (o)

        if random.choice ( (True, False, False) ):
            decor = self.__genDecor (face, registry, prand)
            roof.addObjectData (decor)

        return roof


def Register ():
    return [ Simple() ]
```

Python module to generate totally flat roofs (this will only work for buildings where the roof outline is a convex polygon). Note how the `__genDecor` *method is used to add manually created scenery objects to the roof.*

D Refinements to the System

This appendix presents ideas that could be used to enhance the current system.

D.1 Finer control over object selection

As discussed in section 4.1.3, a major failing of the system is its lack of precise control over the creation of the feature generation hierarchies. This section presents a potential solution to the problem: a system allowing the user to write a simple program written in a custom language that would be able to adjust probabilities of certain selections being made in reaction to other selections being made.

The Selection System

By default, the value returned for a socket will be a random choice between all of the possible objects (so if a socket was of type "*/features/window/frame*", any object registered with that type could be used). However, it will often be necessary to restrict the choice of objects, or to change the frequencies with which each will be used. To this end, the user will be able to define a list of possible children for any object, with an optional relative weighting value.

The system will support *triggers:* code executed when certain conditions are met (Agrawal and Gehani 1989). These will allow for weightings to be changed under certain conditions. This will be useful in situations where seemingly unconnected objects (such as "*/features/window*" and "*/features/door/window"*) need to influence each other in order to keep various features looking similar. The triggers will be local to a single feature set generation; before each generation is started, a copy will be made of the selection state and only this copy will be modified. There will be a watchdog overseeing the trigger system to prevent infinite loops occurring[8].

8 This will most probably take the form of ensuring that any trigger is activated a maximum of one time for any generation.

D.1.1 Parser

The parser will read the program for controlling the function selections. The default selection state will be valid and complete before the user's input is read[9]. The contents of the file will only be used to modify this. The control file will be strictly checked against the registry as it is parsed. If there are any inconsistencies or missing data, the user will be notified and processing will stop[10].

Wild cards ('*') can be used to refer to all objects of a given type. A wild card can be used as an object, in that it can have *pseudo-sockets*. If a socket of a wild card is used, the socket with that of any of the matching objects will be modified. Note that no warnings are generated for missing sockets, unless no objects possess the named socket (i.e. if "*/features/door/frame/*:sockets/foo*" is modified, no warning will be generated as long as one object of type "*/features/door/frame*" has a socket called "*foo*". The modification is simply ignored for the rest of the objects). This will allow the shorthand to be useful without being overly restrictive.

Figure 10 shows an example of the control file syntax. Lines beginning with '#' are comments.

9 The purpose of the control file is to fine-tune the system, rather than simply to make it work. The registration process should result in a complete and working system.

10 The entire detailing process may take several hours, and it is important that every attempt is made beforehand to ensure that the system does not halt partway through because of a small input error.

```
# limit the choice of sub-functions for the “frame” socket of the “basic” door type to be
# either “box” or “side-window”
weight /features/doors/basic:sockets/frame => box side-window

# make the “left” socket of the “split-v” window container be 40% likely to be the “sash”
# function, and 60% likely to be the “simple” function.
# note that the path to these functions need not be specified as it is implied by the
# registered type of the “left” socket
weight /features/window/container/split-v:sockets/left => sash:2 simple:3

# if the “side-window” door frame function is used,re-weight the window probabilities to
# make the window style more like the door style
on /features/door/frame/side-window do
{
       # make window containers more likely to be “interesting”
       weight /features/windows/basic:sockets/container => split-v:5 basic:3

       # make fancy window types more likely to appear
       weight /featureset:sockets/window => oriel-bay:2 basic:1
}

# most triggers will fire on multiple functions, all of which are mentioned within the trigger
# this is because certain sets of functions will tend to be associated with each other
on /features/door/frame/plain or /features/window/frame/plain do
{
       weight /features/door/*:sockets/frame => plain
       weight /features/window/*:sockets/frame => plain
}
```

Figure 10: Example selection control program

D.1.2 Compiled state representation

In order to represent the selection control state more efficiently, the parser will store the state in a compiled format, merged with the default selection data[11]. This format will maintain weighting data and a graph of triggers. The state information will be in the same form as the registry, but the only data stored for each object will be the values for the sockets.

D.1.3 Per-generation state representation

This is copied from the compiled state at the beginning of each

11 The default data set will provide equal weightings for each function object, and provide no triggers,

generation. This will ensure that changes made in a generation are kept local.

D.1.4 Runtime functionality

The selection state will receive requests for objects from the feature set generation code. For this, the system will need to consult the selection state to obtain weighting lists, as well as referring to the registry itself for descriptor/constraint lookups.

The runtime system will handle triggers. As triggers and function trees are inter-dependant, it is likely that a late-firing trigger will invalidate a function tree that had been generated earlier. This situation will be handled, possibly by checking existing trees for validity every time a trigger fires, and recalculating sections of trees that have become invalid. The watchdog system will guarantee that a solution will be reached (even if it not entirely valid).

D.2 “Ear clipping”

The algorithm used to generate the flat geometry between features for the walls is a simple one: as the wall is divided into rectangular cells, four rectangles are generated between the cell boundaries and the bounding box of the feature, as in Figure 11.

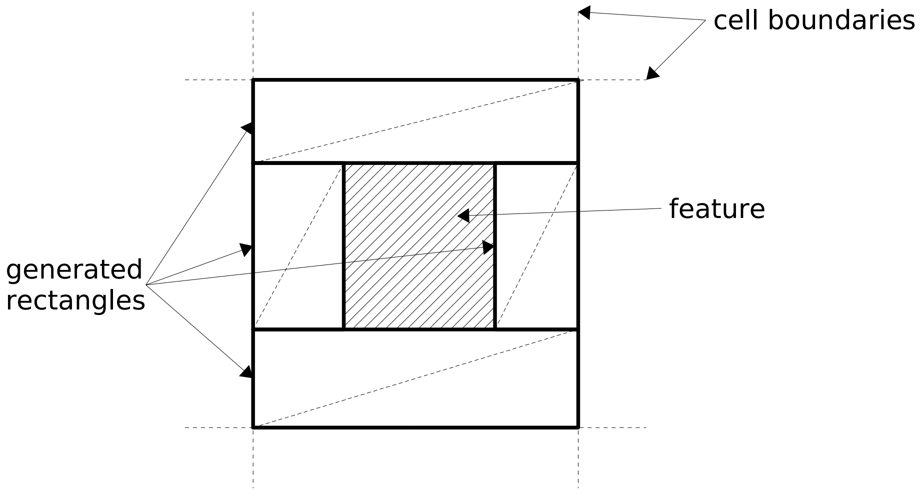


Figure 11: Wall generation, showing triangulation

This implementation is a temporary measure that worked sufficiently well as to never have been replaced. It does however generate more triangles than are necessary, is limited to rectangular features, and introduces *T-junctions* (see section 4.1.1.1).

The planned implementation involved constructing a simple polygon by “subtracting” the outermost contours of the features from the original wall polygon. This polygon is then triangulated using the *Ear Clipping* algorithm (Eberly, 2002). Figure 12 shows the process. This implementation addresses all the above issues, producing a tessellation with the minimum number of triangles that can be generated without introducing *T-junctions*.

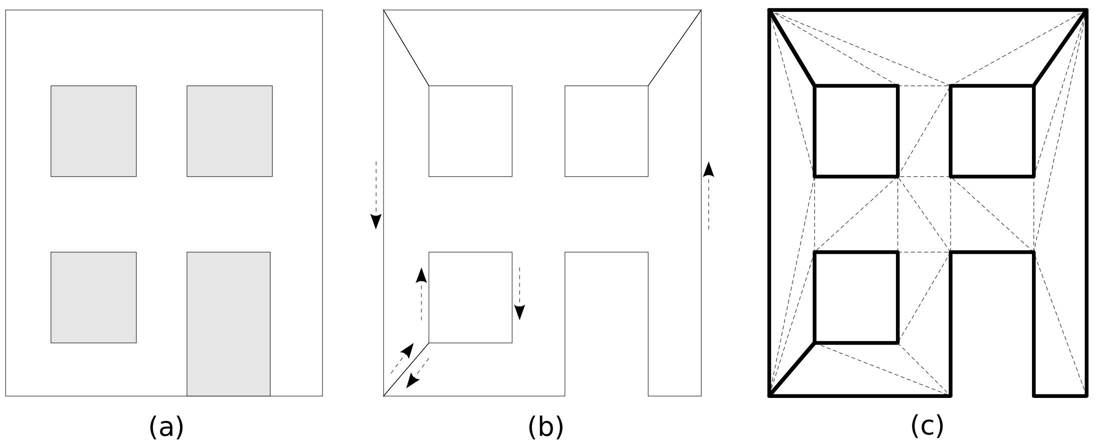


Figure 12: Using ear-clipping to generate wall data. (a) shows the initial wall with features, (b) shows the simple polygon created by "subtracting" the features' outermost contours from the wall polygon (with winding directions indicated by the arrows) and (c) shows a possible triangulation of the simple polygon using ear-clipping.

References

- Agrawal, R., Gehani, N., 1989. *Ode (Object Database and Environment): The Language and the Data Model*, in Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data. Association for Computing Machinery, June 1989, pp. 36-45.
- Akenine-Möller, T., Haines, E., 2002. *Real-time Rendering.* 2nd ed. Massachusetts: A K Peters.
- Birch, P.J. et al., 2001. Rapid Procedural-Modelling of Architectural Structures. *Proceedings of the 2001 conference on Virtual reality, archaeology, and cultural heritage*. New York: ACM Press.
- Braun, C. et al., 1995. Models for Photogrammetric Building Reconstruction. *Computer & Graphics*. 19(1): pp.109-118.
- Chen, M., 1999. *Generation of Three-Dimensional Geometry for Night Illumination and Urban Visualization.* [online] MIT. Available from: World Wide Web: http://gfx.lcs.mit.edu/~maxchen/AUP.pdf [Accessed 6 Oct 2003]
- Day, M., 2001. *Metadata in a nutshell.* [online] UKOLN. Available from: World Wide Web: http://www.ukoln.ac.uk/metadata/publications/nutshell/ [Accessed 11 Dec 2003]
- Discreet [online]. Available from: World Wide Web: http://www.discreet.com [Accessed 25 Mar 2004]
- DMA Design Limited. 2001. *Grand Theft Auto 3*. [Disk]. Sony Playstation 2. Rockstar North.
- Eberly, D., 2002. *Triangulation by Ear Clipping.* [online] Available from: World Wide Web: http://www.magic-software.com/Documentation/TriangulationByEarClipping.pdf [Accessed 21 Mar 2004]
- ELSPA. *Weekly chart summary.* [online] Available from: World Wide Web: http://www.elspa.com/about/charts/charts.asp [Accessed 21 Mar 2004]
- Parish, Y., 2001. *Procedural Modelling of Cities*. [online] SIGGRAPH 2001. Available from: World Wide Web: http://www.centralpictures.com/ce/tp/paper.pdf [Accessed 7 Oct 2003]
- Python [online]. Available from: World Wide Web: http://www.python.org [Accessed 26 Mar 2004]

- Raymond, E.S., 2004. *The Art of UNIX Programming.* Massachusetts: Addison Wesley.
- Reiser, H., 2001. *The Naming System Venture*. [online] Available from: World Wide Web: http://www.namesys.com/whitepaper.html [Accessed 29 Apr 2004]
- Siddiqui, A., 2003. *The Bridge Design Pattern*. [online]. University of Scranton. Available from: World Wide Web: http://www.cs.uofs.edu/~bi/2003f-html/se516/bridge.htm [Accessed 26 Mar 2004]
- Team Soho. 2003. *The Getaway.* [Disk]. Sony Playstation 2. Sony Computer Entertainment.
- Weidner, U., Förstner, W, 1995. Towards Automatic Building Reconstruction from High Resolution Digital Elevation Models. *ISPRS Journal.* 50(4): pp.38-49.
- Zaiane, O., 1998. *New DB Applications*. [online]. Simon Fraser University. Available from: World Wide Web: http://www.cs.sfu.ca/CC/354/zaiane/material/notes/Chapter8/node2.html [Accessed 26 Mar 2004]