A caching approach to real-time procedural generation of cities from GIS data

Brian Cullen
Trinity College Dublin
cullenb4@cs.tcd.ie

Carol O’Sullivan
Trinity College Dublin
Carol.OSullivan@cs.tcd.ie

ABSTRACT

This paper presents a method for real-time generation of detailed procedural cities. Buildings are generated as needed from real GIS data, using modern techniques that can generate realistic content and without having a huge impact on the rendering system. The system uses a client-server approach allowing multiple clients to generate any part of the city the user wishes without requiring the full data-set, or any pre-generated models. The paper introduces the use of object oriented shape grammars to reduce redundant code and presents a parallel cache to allow real-time generation of detailed cities.

Keywords: Procedural Modelling, GIS Data, Buildings, Cities, Real-Time Rendering.

1 INTRODUCTION

Procedural modelling of urban environments has become an important topic in computer graphics. With the ever increasing demand for larger and more realistic content in games and movies, the time and cost to model urban content by hand is becoming unfeasible. Apart from the entertainment industry, large urban models are also desired for urban planning applications and emergency response training.

We present a client-server system capable of generating huge cities of any size without requiring the client to download large 3d geometrical data sets. Our main contributions are as follows:

1. We propose the use of object oriented shape grammars to combat redundancies when creating buildings with multiple different styles.

2. We introduce a multi-state parallel cache that procedurally generates the city’s geometry before it becomes visible. We will demonstrate frame-rate improvements over a system that simply generates buildings as they are needed.

While many cache based approaches have been proposed for rendering large terrains, the use of such techniques has not been explored for procedural generation of urban models. Numerous problems occur as rendering the buildings takes much less time than generating them. We aim to tackle this problem with a simple solution that can be used with existing techniques for terrain paging.

After an overview of our system and how we can utilise GIS data to model real cities (Section 3), we then introduce the idea of object oriented shape grammars (Section 4) demonstrating how they can be used to make simple changes to a building without creating redundant code. In Sections 5 and 6 we present our cache based system that can generate huge cities in real-time with interactive frame-rates and evaluate it. Example code of object oriented shape grammars is listed in the Appendix for the interested reader.

2 RELATED WORK

This section will review current techniques for the procedural generation of 3d building models. We will mainly review systems that employ production systems as they have been the most successful at generating realistic content. Other approaches based on stochastic texture synthesis ideas are touched upon briefly.

Detailed architectural models can be created using production systems (a set of symbols that are iteratively replaced according to a well defined grammar) but require a modeller to manually write rules. Their strength lies in the ability to provide detailed descriptions and yet randomness in a structured way.

Parish et al. [24] introduced the idea of using L-Systems [26] to model architectural content. L-Systems are production systems that use the parallel replacement of symbols in a string to simulate a growth process. L-Systems have previously achieved a lot of success in modelling trees and plants [27, 23], but have limitations in modelling buildings (since a building structure is more spatially constrained and does not reflect a growth process).

Stiny pioneered the idea of shape grammars [33, 31, 32] which can be used for generating complex shapes within a given spatial area. Shape grammars have been used for the construction and analysis of architectural designs [5, 8, 34, 12]. However, Stiny’s original shape grammar operates on sets of labelled points and lines.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
and is difficult to implement on a machine because of
the number of transformations that must be searched
before a rule can be selected and applied.

Wonka et al. [37] modify the idea of shape grammars
to better represent building facades. They use a split
grammar in which building facade is derived using a
sequence of split and repeat commands to subdivide a
planar shape.

Müller et al. [21] expand on this idea by develop-
ing the CGA shape grammar. This grammar includes
environmental parameters that allow a shape (a part of
a derived facade) to query if it is occluded by some-
thing else in the city, thereby aiding the placement of
windows and doors. CGA shape is continually being
improved and has even been used to reconstruct arche-
ological sites [22] and is used in commercial products
like CityEngine [1].

Recently Kracklau et al. [13] presented a new gen-
eralised language based on Python. They can create pow-
erful descriptions by passing non-terminals as param-
ters, thus enabling abstract templates to be defined.

Shape grammars alone are not sufficient to gener-
ate realistic roofs on buildings. Laycock et al. [14] de-
demonstrate a technique to generate roof models in dif-
f erent styles from a building footprint. They modify
the straight skeleton algorithm proposed in [7] to gen-
der different roof types. Soon [30] describes an algo-
 rithm capable of modelling roofs common to east Asian
buildings, like temples and pagodas.

A completely different approach to production gram-
mars takes concepts from texture synthesis and applies
them to 3D models. Texture synthesis traditionally ex-
trapolates image data by incrementally adding bits of
the image that best match a small neighbourhood. This
can produce very convincing results [35, 6, 15].

Merrell and Manoch [18, 19, 20] present a method
that takes an example model as input and can produce
larger models that resemble it. Output models are still
very random and lack the fine control that production
systems provide. Synthesis based approaches to gener-
ating new models are still very slow and are thus not
applicable for interactive applications.

Layout generation concerns the automatic layout of
roads and placement of urban content that is crucial for
generating an entire city. Urban planning applications
require the possibility to view changes to city layouts
and to see the effect a proposed road network would
have on traffic congestion. Using procedural tech-
niques, such changes can be made interactively which
is a great improvement over manual systems.

Parish et al. [24] introduce the use of L-Systems to
grow road networks in a similar way to branches on a
tree. This was one of the corner-stone papers in the area
of procedural cities. However, it is difficult to fine tune
the results because the variables do not give enough
control over the road layout.

Chen et al. [4] introduce the use of tensor fields to
guide road network generation. The user edits the ten-
sor fields using interactive techniques discussed in [38].
Users can then interactively edit individual roads in a
quick and easy manner.

Aliaga et al. [3] take a different approach to recon-
figuring road networks. Using vector data of roads they
form a graph to represent road intersections and parcels
of land. Then, using k-means clustering [17], user-
deformed parcels are replaced with similar parcels from
elsewhere in the city. In [2] they improve on this sys-
tem to allow the synthesis of completely new areas of
the city. Cities with different road structures can then
be blended together.

Grueter et al. [9] use a lazy generation technique to
construct a potentially infinitely large city. Build-
ings are constructed when they are visible in the view
frustum. The system seeds a random number generator
based on the building’s coordinates, thereby allowing
each building to maintain a persistent style. Whelan
et al. [36] present a system that allows real-time in-
teraction in modifying roads and tweaking parameters.
The user provides a height map and lays the roads, af-
after which the system automatically places buildings and
other details. The buildings are simple extrusions with
texture and bump maps. Recently Haegler et al. [10]
presented a system capable of generating detailed cities
in real-time by carrying out procedural generation on the
GPU.

Cache based techniques have been used extensively
in real-time rendering. Paging is a popular technique
for rendering large terrains [28, 16, 39]. Slater et al.
[29] present a caching system that exploits temporal co-
herency to accelerate view culling. Akenine-Möller et
al. [11] discuss many modern real-time rendering tech-
niques including level of detail, batch processing and
imposters.

3 SYSTEM OVERVIEW

In this section we present a system that can produce
large detailed virtual cities in real-time using GIS data.
Previous approaches discussed in Section 2 focus on ei-
ther pre-generating large cities or are limited to simple
grid layouts and building geometry with random styles.
The system presented continuously updates the city by
streaming GIS data from a server along with style de-
scriptions for every building, without interrupting the
rendering system.

Urban GIS is preprocessed and stored in a database
along with style descriptions for every building for
quick referencing. This preprocessing step is explained
in section 3.1. Style sheets that control the facade gen-
eration are loaded at run-time and are stored in a hash-
table on the client’s system. The geometry cache up-
dates itself based on the camera’s position in the en-
vironment, downloading the surrounding environment
data from the GIS database. This includes the position and shape of building footprints and style parameters (such as texture id, height and style id) used for generating the buildings. This allows persistent generation of the city. The cache controls what geometry is procedurally generated based on its distance from the camera. Meshes for the roads and buildings are then batched together for efficient rendering and sent to the render system. This process is described in detail in Section 5.

3.1 Data Extraction from GIS

The GIS data recorded contains detailed urban planning information, which is stored in different semantic layers that make it easy to access the building layouts. However, since the data is simply represented by a set of poly-lines, it is necessary to determine which lines belong to the same buildings. Figure 1 illustrates this process. The following algorithm describes how to extract the building layouts:

1. Create a graph representing all the vertices and edges.
2. Start at the bottom left node which contains two or more edges.
3. Follow the least interior angle edges until the starting node is reached again, thus creating a cycle.
4. Decrement the degree of every node along the cycle.
5. Repeat from Step 2 until no nodes with a degree greater than one remain.

A similar approach was taken by Pina et al. [25], however, individual buildings are extracted as opposed to urban blocks. The extracted building footprints are then loaded into a database for quick referencing by the system. A similar technique is used to extract the roads and insert the road network graph into a database.

4 BUILDING GENERATION

Buildings are procedurally generated using split grammar rules based on [21]. The rules compose of subdiv, repeat, insert, extrude, detrude and comp commands, which can subdivide and decompose shapes into new ones.

Comp

Breaks a shape down into the lower dimensional shapes it is composed of. For example, a building is broken down into its composing facades;

Subdiv

Subdivides a shape along a given axis;

Repeat

Subdivides a planar shape several times to fit many new shapes of a given width;

Insert

Replaces a planar shape with an external model;

Extrude

Extrudes a planar shape, thereby creating a new volumetric shape;

Detrude

Detrudes a planar shape, thereby creating a new volumetric shape.

Combinations of these simple commands can produce complex architectural geometry, while building roofs are generated using the approach described in [14]. The rules are specified using a script with a parameterised L-System style syntax:

\[ \text{Pred} : \text{Exp} \sim \text{Command}(\text{params})\{\text{Successor}\} : \text{Prob} \]

If the Boolean Expression evaluates to true then Command is carried out on the shape with ID Predecessor and the resulting output shapes are given the ID Successor. Multiple rules can be specified for the same Predecessor and one is chosen at random based on its Probability value. This allows some variability among generated shapes.

A simple compiler was built to parse the scripts at runtime and generate a hash table of C++ function objects. This allows the script to be applied extremely quickly to new buildings but also allows parameters to be changed at runtime.

4.1 Object Oriented Buildings

The production system presented in [21] contains a lot of redundant code between different building scripts. It is very cumbersome to rewrite entire building specifications just to make a specific change.

We propose the use of object oriented buildings as a solution to this problem. Figure 2 illustrates this idea. Buildings inherit everything from more abstract styles and only respecify certain aspects of the style. This is achieved by encapsulating semantically relevant production rules in labelled blocks. Each block is given a list of variables that can be changed at runtime or respecified by a child style. Code listings to generate the buildings in Figure 2 can be found in the Appendix. Buildings also inherit their parents’ elements (i.e., 3D models that are imported and used to replace certain terminal symbols) and can add or remove from their parents’ element set. We allow multiple meshes to be specified, corresponding to different levels of detail for the rendering system. Meshes are swapped with different level of detail meshes depending on their distance to the camera. In this implementation of the system the Ogre rendering engine was utilised to manage level of detail swapping and rendering of the scene. The use of object oriented building styles can simplify the writing of new styles and can link building styles together in a meaningful way.
Figure 1: Extracting building footprints from GIS data (left). Layer containing buildings is first chosen by the user (middle), while buildings are then extracted by finding loops in the data (right).

Figure 2: Building2 inherits from Building1, specifying how windowsills should be added. Building3 also inherits from Building1, adding a ledge to each floor. Code listings are provided in the Appendix.

5 REAL-TIME GENERATION

In this section we present our process for generating procedural cities in real-time.

5.1 Parallel Geometry Cache

In order to maintain a constant and high frame rate, building generation should not interrupt the rendering system. We achieve this by introducing a multi-state cache that stores geometry that is currently being generated. The system is based on the idea of paging geometry for rendering large terrains. The world is split into a regular grid as illustrated in Figure 4. The data in the cache has the following three states:

State 1 Geometry descriptions are downloaded from the database and the area is procedurally generated. (Outer white area in Figure 4).

State 2 Meshes are constructed and sent to the graphics card but are not yet rendered (Middle blue area in Figure 4).

State 3 Meshes currently being rendered (Inner green area in Figure 4).

Depending on the camera motion, grid squares that are likely to become visible in the near future are loaded. Geometry descriptions are downloaded from the GIS database, procedurally generated and inserted into the cache. This is done in a separate thread from the rendering system. Only squares that are close to the camera are rendered. If a square is not yet generated, the rendering thread will put it on the end of a queue and try to retrieve the next square.

5.2 Parallel Building Generation

With the trend in computing power drifting towards multi-processor architectures, it is desirable to take advantage of parallel computation. It is possible to procedurally generate multiple buildings at the same time by utilizing parallel processing techniques. Algorithm 3 presents a simple algorithm that can speed up building generation on multiprocessor systems.

```
while NewPage = getPageFromQueue()
    NumBldPerThd = NewPage.NumBlds/NumProc
    for x = 0 to numProcessors - 1
        Thread[x] = ForkThread()
        Thread[x].MemoryPool = new MemoryPool
        Thread[x].BuildingList = distBuildings(NumBldPerThd)
        Thread[x].GenerateBuildings()
        NewPage.setBuildingMeshes(Thread[x])
    end for
    SynchroniseThreads()
end while
NewPage.BatchMeshes()
```

Algorithm 3: Algorithm for procedurally generating buildings in parallel.

Each thread maintains a memory pool that is reused for every building it generates, which reduces memory allocation bottlenecks. Threads must synchronise before writing to the cache so that buildings can be batched together for fast rendering.

6 RESULTS

To test the system, we conducted two separate benchmark, which were performed on a machine with the following specifications:

- CPU: Intel(R) Core(TM)2 Duo CPU E8500 @ 3.16GHZ
- RAM: 4GB
- GPU: NVIDIA GeForce 9800GT

First, a frame rate analysis of the system was taken while the camera was moved between two preset points,
Figure 4: As the camera moves towards the geometry, new pages must be loaded. The pages are organised into a queue and processed in order of their distance to the camera. The buildings within each page should be shared among parallel executing threads.

both with and without the parallel geometry cache (Section 5.1). The results are given in Figure 5. While the camera travelled a distance of 800m in the scene, exactly 2,038 buildings were created. This had a significant effect on the frame rate of the system without a parallel cache. The sudden drops in frame rate correspond with new geometry pages being loaded and cause a jerk in the camera motion. In the system with the parallel cache there is much less jerking when pages are loaded and the overall frame rate stays within acceptable levels.

The second experiment performed was a multi-threaded processing benchmark. Four pages were generated consisting of 10, 100, 1000 and 10,000 buildings respectively. Processing time was logged for each of the pages with building generation distributed over different number of threads. The average results over ten repetitions are shown in Figure 6. A configuration with two threads running in parallel yielded the best performance on the dual core machine. Running the experiment with more threads than processors led to worse results because of the overhead of thread switching. However, this result suggests better performance could be achieved with a greater number of processing cores. Better results were obtained using larger page sizes with 10,000 buildings leading to a 27.48% increase in performance (We suspect that this is due the initial memory pool allocation assigned to each thread). Table 1 shows the number of buildings generated per second for the 10,000 building page test. Each building was set to be strictly the same shape, contained an average of 980 vertices and required 610 shape operations to generate.

Figure 7 demonstrates the type of architecture and scale of the city generated in the tests.

Figure 5: A comparison of results with and without the cache described in Section 5.1. The system with the cache has a much higher frame rate and less jerky movements of the camera. There was an average of 1,922 buildings in the scene at any time with 2,038 buildings created and destroyed over the distance.

Figure 6: Time in seconds to generate buildings with different levels of multithreading. On the dual core machine two threads yielded the best performance.

<table>
<thead>
<tr>
<th></th>
<th>Bld/Sec</th>
<th>Ops/Sec</th>
<th>Percent Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Thread</td>
<td>255.56</td>
<td>15,586.34</td>
<td></td>
</tr>
<tr>
<td>2 Threads</td>
<td>325.78</td>
<td>19,868.86</td>
<td>27.48%</td>
</tr>
<tr>
<td>3 Threads</td>
<td>277.47</td>
<td>16,922.73</td>
<td>8.57%</td>
</tr>
</tbody>
</table>

Table 1: Benchmark of multi-threaded processing on the 10,000 building data set. Results are shown for the number of buildings generated per second, the number of shape operations (discussed in Section 4) performed per second and the percentage performance boost over a single threaded configuration.
7 CONCLUSION

We have presented a system that can generate large virtual cities with detailed buildings in real-time. The system can be run over a network while allowing multiple clients with only one data set. We introduced the idea of object oriented building styles that can help reduce code redundancies and make it easier to specify multiple building styles. We also presented a set of benchmarking statistics calculated with different configurations of the system. The results showed that our parallel cache offers superior performance to that of a system that simply generates the buildings as they are needed. We also showed a performance benefit when utilising parallel generation on multi-core processors.

Regarding limitations, currently the system only generates buildings within a single page in parallel. The results from our experiment suggest that improved performance could be achieved by generating sets of pages in parallel, thus handling more buildings per thread and requiring less thread synchronisation. Rendering of the system could improve by implementing occlusion culling and better LOD techniques. In this implementation, different level of details are provided for a building’s elements but not the shape of the building itself.

A SHAPE GRAMMAR SYNTAX

In this appendix we present the syntax of our object oriented shape grammar.

The listings correspond to the buildings shown in Figure 2. Semantically relevant production rules can be combined into meaningful blocks. Each block can have its own list of variables that may be changed at runtime. A child class inherits everything from its parents and may redefine a block of rules and its variables. In addition to defining a block of production rules, a class can also define a set of building elements (Listing 9). These elements correspond to terminal symbols in the production system, which should be replaced with external models. A series of meshes can be given to each element specifying a different level of detail. In our system, the distance at which to change a mesh is the same for each element and is specified by the cache system. As with the production rules, probabilities are given for the replacement of terminal symbols with 3D meshes.

### Listing 8: Listing for simple building

class Building1 : ElementPack
{
    Footprint {
        FOOTPRINT
        extrude ( BUILDING_HEIGHT ) BuildingVol : 1
        BuildingVol ~
        comp ( "facades" ) FACADE : 1
    }
    Facade {
        var GroundFloorHeight 1
        FACADE : H > ( GroundFloorHeight + 1 )
        subdiv ( "Y" , GroundFloorHeight , 1 )
        [ GROUND.Floor | UPPER.Floors ] : 1
    }
    Ground.Floor {
        var EntranceWidth 0.75
        var DoorDepth 0.1
        GROUND.Floor : W > ( EntranceWidth + 1 )
        subdiv ( "X" , EntranceWidth , 1 )
        [ FLOOR | EntrancePanel | WALL ] : 0.3
        GROUND.Floor : W > ( EntranceWidth + 1 )
        subdiv ( "X" , EntranceWidth , 1 )
        [ FLOOR | EntrancePanel | FLOOR ] : 0.44
        GROUND.Floor : W > ( EntranceWidth + 1 )
        subdiv ( "X" , EntrancePanel , 1 )
        [ WALL | EntrancePanel | FLOOR ] : 0.3
        subdiv ( "Y" , 0.02 , EntrancePanel , 1 )
        [ WALL | Entrance ]
        Entrance ~ extrude ( DoorDepth )
        [ DOOR | WALL ] : 1
    }
    Upper.Floors {
        var FloorHeight 1.0
        UPPER.Floors ~ repeat ( "Y" , FloorHeight ) { FLOOR } : 1
    }
    Floor {
        var TileWidth 1.1
        FLOOR ~ repeat ( "X" , TileWidth ) { TILE } : 1
    }
    Tile {
        var WindowDepth 0.1
        var WindowWidth 0.75
        var WindowHeight 0.5
        TILE ~ subdiv ( "X" , 1 , WindowWidth , 1 )
        [ WALL | Tile | WALL ] : 1
        Tile ~ subdiv ( "Y" , 1 , WindowHeight , 1 )
        [ WALL | WindowPlane | WALL ] : 1
        WindowPlane ~ extrude ( WindowDepth )
        [ WINDOW | WALL ] : 1
    }
}

Figure 7: Output of the system
\begin{verbatim}
class ElementPack
{
    Elements {
        WINDOW:
            "window1LOD1 . mesh" "window1LOD2 . mesh" : 0.5
            "window2LOD1 . mesh" "window2LOD2 . mesh" : 0.5
        DOOR:
            "door1 . mesh" : 0.2
            "door2LOD1 . mesh" "door2LOD2 . mesh" : 0.8
        LEDGE:
            "windowLedge1LOD1 . mesh" "windowLedge1LOD2 . mesh" :
            }
}
\end{verbatim}

Listing 9: Listing for elements

\begin{verbatim}
class Building2 : Building1 {
    Tile {
        var LedgeHeight 0.075
        var WLedgeHeight (WindowHeight+LedgeHeight)
        TILE ~ subdiv("X",1r,WindowWidth,1r) (WALL | TileC | WALL) ; 1
        TileC ~ subdiv("Y",1r, WLedgeHeight,1r) (WALL | WindowPlane | WALL) ; 1
        WindowPlane ~ detach (WindowDepth)
            (WindowPlaneInner | WALL) ; 1
        WindowPlaneInner ~ subdiv("Y",LedgeHeight,1r)
            (LEDGE | WINDOW) ; 1
    }
}
\end{verbatim}

Listing 10: Building2 inherits everything from Building1 but
specifies how windowsills should be added.

\begin{verbatim}
class Building3 : Building1 {
    floor {
        var TileWidth 1
        var LedgeHeight 0.075
        FLOOR ~ subdiv("Y", LedgeHeight,1r)
            (LEDGE | FloorU) ; 1
        FloorU ~ repeat("X", TileWidth) (TILE) ; 1
    }
}
\end{verbatim}

Listing 11: Building3 inherits everything from Building1
adding a ledge to each floor.

REFERENCES

[1] Procedural inc. - 3D modeling software for urban
    N. Andrysco. Interactive reconfiguration of urban
    47, 2008.
    active example-based urban layout synthesis.
    E. Zhang. Interactive procedural street modeling.
tool for customizing Álvaro Siza’s mass houses at
Malagueira. PhD thesis, MIT School of Architec-
Texture synthesis from photographs. CGF: Euro-
    implementation. In SCCG: Spring Conference on
[8] U. Flemming. More than the sum of parts: the
    grammar of queen anne houses. Environment and
    Planning B: Planning and Design, 14(3):323–
Real-time procedural generation of ‘pseudo inﬁ-
nite’ cities. In GRAPHITE: Computer Graphics
    and Interactive Techniques, page 87–ff, New
    York, NY, USA, 2003. ACM.
[10] S. Haegler, P. Wonka, S. M. Arisona, L. V. Gool,
    and P. Müller. Grammar-based encoding of fa-
    cades. CGF: Eurographics, 29(4):1479–1487,
    2010.
    programmable culling unit. ACM Trans. Graph.,
[12] H. Koning and J. Eizenberg. The language of the
    prairie: Frank lloyd wright’s prairie houses. En-
    vironment and Planning B: Planning and Design,
ized use of Non-Terminal symbols for procedural
    modeling. CGF: Eurographics (to appear 2010),
    2010.
    erating roof models from building footprints. In
[15] S. Lefebvre and H. Hoppe. Appearance-space tex-
    ture synthesis. In ACM Trans. Graph., pages 541–
    548, Boston, Massachusetts, 2006. ACM.
    patches for GPU-based terrain rendering. Vis.
[17] J. B. MacQueen. Some methods for classifica-
    tion and analysis of multivariate observations.
    In 5th Berkeley Symposium on Mathematical Statis-
    I3D: Symposium on Interactive 3D Graphics and
    Games, page 105–112, New York, NY, USA,
    2007. ACM.


