GRAPHITE 2003

Real-time Procedural Generation of 'Pseudo Infinite' Cities

Stefan Greuter, Jeremy Parker
stefan.greuter@gmx.de,jeremy.parker@rmit.edu.au
Centre of Animation and
Interactive Media

Nigel Stewart, Geoff Leach
nigels@nigels.com, gl@cs.rmit.edu.au
School of Computer Science and
Information Technology

RMIT University, Melbourne, Victoria, Australia

Abstract

We present an approach to procedural generation of 'pseudo infinite' virtual cities in real-time. The cities contain geometrically varied buildings that are generated as needed. The building generation parameters are created by a pseudo random number generator, seeded with an integer derived from the building's position. The varied building geometries are extruded from a set of floor plans. The floor plans for each building are created by combining randomly generated polygons in an iterative process. A display list caching and frustum filling approach manages the generation of buildings and the use of system resources. This approach has been implemented on commodity PC hardware, resulting in interactive frame rates.

CR Categories: I.3.5 [Computational Geometry and Object Modelling]: Geometric algorithms—; E.1 [Data Structures]: Lists, Trees; G.3 [Probability and Statistics]: Random number generation; I.3.7 [Three-Dimensional Graphics and Realism]: Virtual reality

Keywords: real-time, procedural generation, LRU caching, view frustum filling, architecture

1 Introduction

This work is motivated by the desire to generate visually interesting virtual cities with highly diverse and complex buildings that are composed of simpler elements. A generated virtual city can be freely explored from a first person perspective. It is self-contained and neither imports nor exports geometry.

A procedural generation approach is used to create these virtual cities. Pseudo random numbers in combination with algorithms generate a variety of buildings and streets that create the impression of a city. To increase diversity, individual buildings are generated on the fly as they are encountered by the user. As a result the city expands to an extent that would require a 'lifetime' to explore, which we term pseudo infinite.

Procedural generation techniques are widely used in computer graphics to model systems of high complexity. Many of these techniques target the generation of natural phenomena in high complexity and detail to achieve realistic results. Procedural generation can be computationally intensive and is not commonly used in real-time systems to generate entire virtual worlds. However, advancements in processing speed and graphics hardware make it now possible to procedurally generate three-dimensional models in real-time on commodity hardware.

Procedural geometry generated on the fly cannot be preprocessed for performance by the same methods used for static geometry. To achieve stable frame rates and smooth, coherent navigation our approach uses view frustum filling and cached display lists. View frustum filling determines which geometry to draw on the screen. Display list caching manages the generation of the three-dimensional content and frees resources no longer in use.

Real-time procedural generation has great potential for applications in education, architecture, simulation, entertainment or the playful pursuit of interesting imagery. Virtual worlds that are procedurally generated have desirable characteristics such as high degree of visual variety, flexibility and pseudo infinite extent.

This paper is divided into six sections: Section 2 provides an overview of related work. Section 3 discusses procedural generation of floor plans and buildings as well as techniques such as view frustum filling and hashing to generate and display virtual cities in real-time. Section 4 provides an overview of OpenGL display lists and how they are used for geometry caching. Section 5 discusses the test results of experiments regarding the performance of city generation, building generation and LRU (least recently used) caching. Concluding remarks and areas of further work follow in Section 6.

2 Related Work

Procedural generation has a long tradition in the field of computer graphics. Techniques include noise [Perlin 1985], fractals [Mandelbrot 1977], L-systems [Prusinkiewicz et al. 1990] and shape grammars [Stiny 1975]. These techniques are common components of systems that generate entities such as clouds [Pallister 2000], trees [Oppenheimer 1986] and other natural phenomena.

Macri and Pallister [Macri and Pallister 2000] describe the procedural generation of a three-dimensional landscape with trees and clouds, all based on Perlin noise [Perlin 1985]. Their prototype demonstrates a procedurally generated terrain with trees and two dimensional cloud layer. The terrain can be freely explored on the horizontal plane in real-time and expands around the user's point of view.

A system called 'CityEngine' [Parish and Mueller 2001] uses extended L-systems to generate entire city models. The system uses a hierarchical set of rules to generate street patterns and buildings. The 'CityEngine' does not support the generation of building geometries in real-time, but facilitates VRML export functionality.

'The Other Manhattan Project' [Yap et al. 2001] describes tools for automatic generation of 'Manhattan like' cities based on statistical parameters.

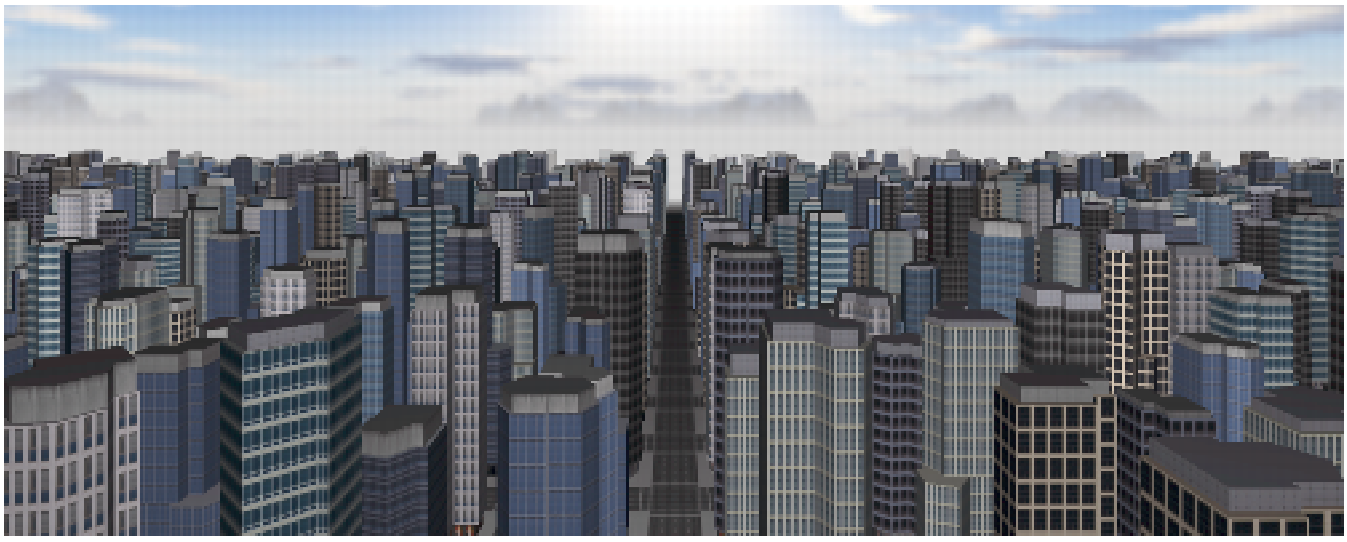


Figure 1: Real-time procedural virtual city

Semi-automated systems to reconstruct the interior of buildings from two dimensional architectural floor plans have been proposed [So et al. 1998]. Common tasks of wall extrusion, object mapping, and ceiling and floor reconstruction are automated, but still rely on user input. The end result is a three-dimensional model that can be exported to VRML.

Lecky-Thompson [Lecky-Thompson 2001] explains the principles of seeded random number generation in relation to procedural content generation of 'infinite' computer game worlds. The principles are discussed in terms of two-dimensional examples.

Alexander [Alexander et al. 1977] describes construction patterns for the methodical creation of interior and exterior design of cities, buildings, streets and gardens in various levels of detail. Although these patterns are not organized in a format that can be directly utilized by computer software, they do provide a useful guideline to identify significant parameters that govern the visual appearance of objects and structures.

3 Procedural City Generation

We present a system that generates pseudo infinite virtual cities which can be interactively explored from a first person perspective. An example of one of our cities is given in Figure 1. All geometrical components of the city are generated as they are encountered by the user. The shape of a building is determined by its location. If the user returns to a particular location the same buildings will be present. Only buildings and streets which surround the viewpoint are generated and stored in memory. Accordingly, buildings that drop out of the viewing range are deleted and the memory reclaimed. As a result, the amount of information stored in memory remains roughly constant, even though the virtual city has no apparent boundaries and can be explored to a pseudo infinite extent. A similar approach for landscapes has been outlined by Maurice Danaher [Danaher 2002].

3.1 View Frustum Filling

Real-time 3D applications often use view frustum clipping algorithms to constrain rendering to geometry visible from a particular viewpoint. In the context here the problem is formulated differently. Our aim is to fill the view frustum with procedural geometry rather than cull hidden, existing geometry.

We use the term *view frustum filling* to describe the restriction of procedural generation to parts of the virtual world located within the camera's view. In our example of a virtual city, view frustum filling is implemented to determine the visibility of virtual world objects before generation.

The approach to view frustum filling we have used is to divide the terrain into square cells on a 2D grid. Each cell represents a proxy for its procedurally generated content. The cells are arranged in square loops around the camera's position located at the center. Cells are tested for potential visibility before their content is generated and drawn. Each cell in our virtual city contains either buildings or streets.

The potential visibility of a cell is determined by the angle between the cell and viewing direction, as well as the distance from the camera. In our implementation only the content of cells located within a 120° viewing angle and a distance of $loops \times cellsize$ are considered visible. Figure 2 shows the visible cells in the viewing area from a bird's eye view.

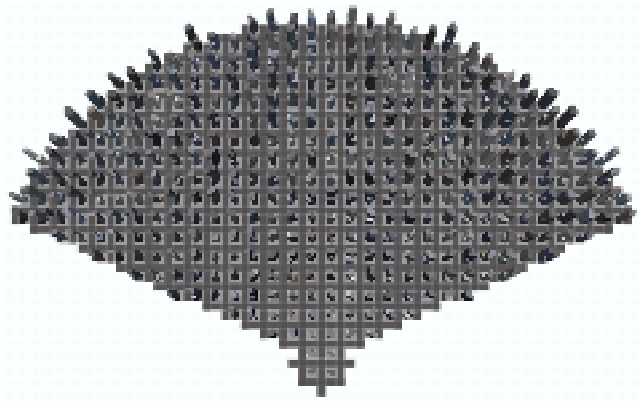


Figure 2: View frustum filling

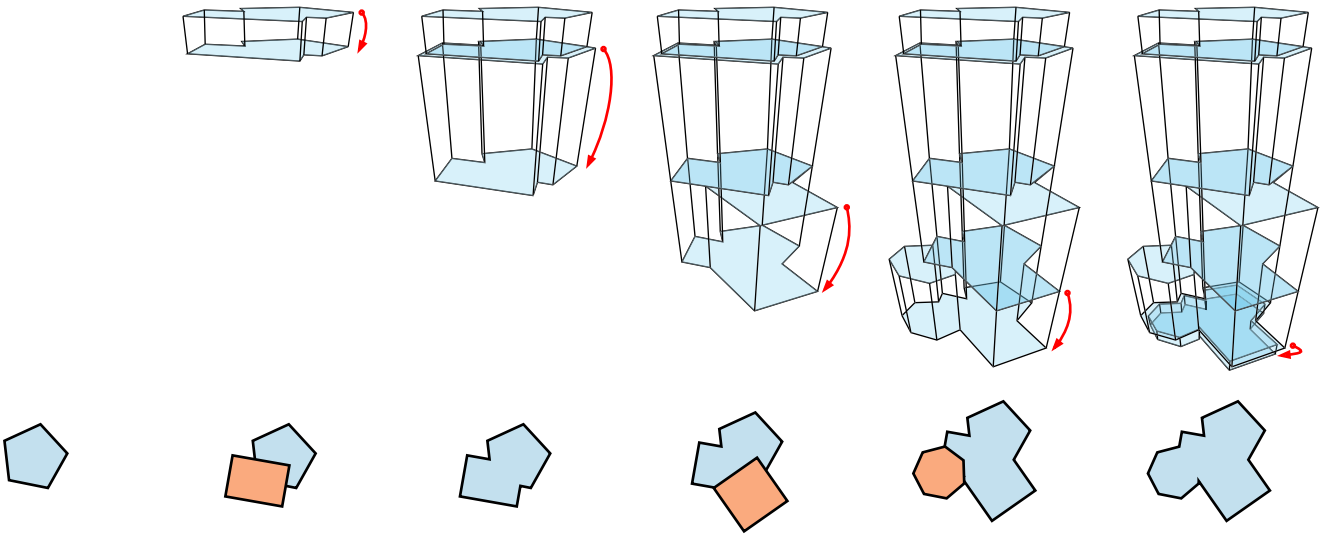


Figure 3: Floor plan generation: (a) generated source primitive (b) generated temporary primitive (orange) with center translated to randomly selected vertex in source with extruded top floor (c) merged temporary and source primitive with extruded building section (d-e) another two iterations (f) finished floor plan with complete building

3.2 Hashing

The form and appearance of each building is determined by a single 32 bit pseudo random number generator (PRNG) seed. The random number sequence determines building properties such as width, height and number of floors.

Similar initial sequences of random numbers for similar seeds have been observed with the random number generator we are using. Similar sequences of numbers can result in recognisably similar buildings. We avoid the generation of similar buildings by using a hash function to convert each cell position into a seed.

For hashing we use Thomas Wang's 32 bit Mix Function [Wang 2000] which is fast and provides a good distribution of seed values. The function is based on a sequence of bitwise operations and returns a 32 bit integer value for any 'key' as follows.

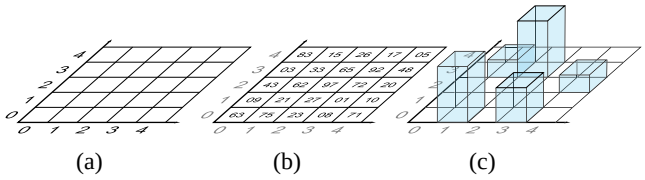
```
unsigned int hash(int key)
{
  key += ~(key << 15);
  key ^=  (key >> 10);
  key +=  (key << 3);
  key ^=  (key >> 6);
  key += ~(key << 11);
  key ^=  (key >> 16);
  return key;
}
```

The x and z coordinates of a cell are hashed with a global *citySeed* to determine a 32 bit integer seed value for each building.

```
seed = hash(x^hash(z^citySeed));
```

Figure 4(a) illustrates the cell coordinates and Figure 4(b) the correspondingly generated seed values. The resulting cell seed is used in the pseudo random number generator and determines the properties for the cell's building as illustrated in Figure 4(c) in a 'feedforward' process [Lecky-Thompson 2001].

The 32 bit integer for x and z limits the extent of the city to 2^{32} cells in length and width. Cells in our city are 25 meters in width and length. To travel in a straight line from one end of the city to the other at a speed of two blocks per second (about 180 km/h) would take approximately 68 years — a human life time.



(a) (b) (c)

Figure 4: From integer grid to individual buildings: a) 2D grid (b) hashed seeds (c) procedural buildings

3.3 Pseudo Random Number Generation

Pseudo random number generators are an important component of procedural systems. PRNGs are used as an integral part of our algorithms which generate floor plans and buildings.

PRNGs produce a sequence of 'random' numbers given an initial seed value. When initialized with the same seed, identical sequences of numbers are produced. In the context of procedurally generated cities the regeneration of the same sequence of numbers is important. Buildings generated for a particular cell are always the same, maintaining the coherence of the city.

The quality of the numbers produced by the PRNG is not critical in our context, since only a few random numbers are used to generate each procedural object. We use Park and Miller's [Press et al. 1992] linear congruential random number generator. This random number generator has limitations but is portable, fast and has a reasonably long period: 2.1×10^9.

3.4 Floor Plan Generation

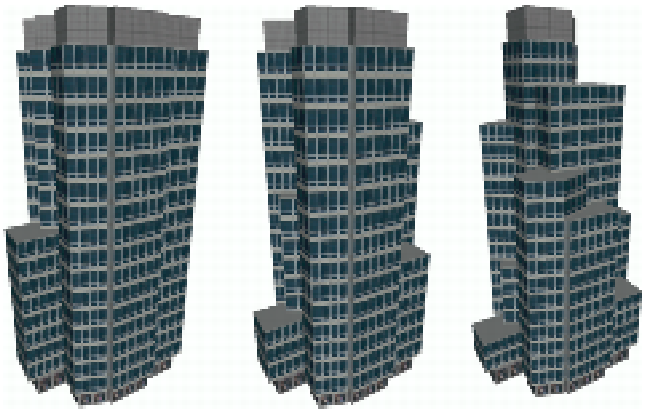
Floor plans are two-dimensional polygons. Each floor plan consists of randomly selected and merged regular polygons and rectangles. Floor plans are generated by an iterative process, which is based on the building's master seed that seeds the floor plan's PRNG. Floor plans are generated for each extrusion step from the top level to the ground level. The first iteration generates a random polygon which serves as the first floor plan, as shown in Figure 3(a). Figures 3(b)–3(d) show subsequent iterations, where a new floor plan is created by generating a new random polygon that is combined

in a union operation with the floor plan polygon from the previous iteration. The resulting floor plan polygon serves as the source polygon in the next iteration. Each resulting floor plan is stored in a `std::vector` of polygons. All floor plans are scaled to fit into a unit square and surface normals are calculated. We used the 'General Polygon Clipping Library' [Murta 2000] to iteratively extend the floor plans with random polygon primitives.

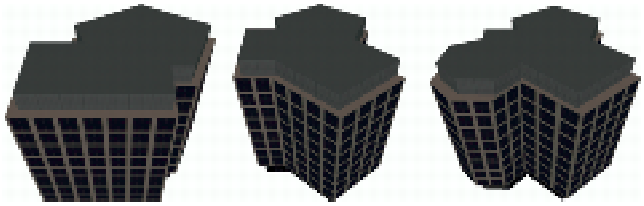
Pseudo-code for the algorithm is given below:

Algorithm 1 Floor plan generation

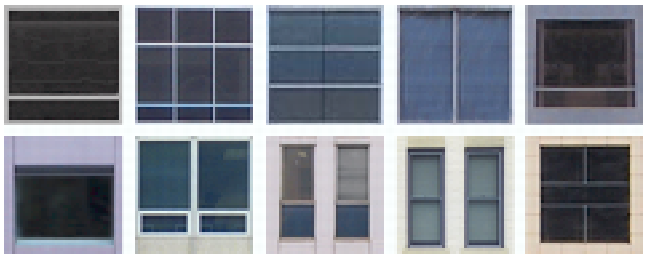
```
src ← random polygon
for every building iteration do
    tmp ← random polygon
    rotate tmp randomly about y axis
    translate tmp to random vertex in src
    src ← src union tmp
```



(a) Building steps: 1, 4 and 8 steps



(b) Floor plan iterations: 2, 3 and 5 iterations (with no steps)



(c) Building textures

Figure 5: Building parameters

3.5 Building Generation

The outer shape of a building is subdivided into a number of steps each of which consist of extruded floor plans. The two dimensional floor plans are transformed to fit the length and width of a building. As shown in Figure 5(a) floor plans are vertically extruded by a random height to resemble the various steps of a building.

The extrusion of each building section starts with the top floor. In Figure 3(b), the top floor is extruded from the second floor plan iteration and its floor height is limited to a maximum of two stories. Each consecutive building section is of random height and is attached to the bottom of the preceding section. The extruded floor plan increases in number of iterations and size, while the building increases in height, as illustrated in Figures 3(b)–3(f).

The actual extrusion process is realised by adding the missing height information to the two dimensional floor plan vertices. Following a counterclockwise order, all floor plan vertex coordinates are translated by a random height along the y-axis in the negative direction, as shown in Figure 6.

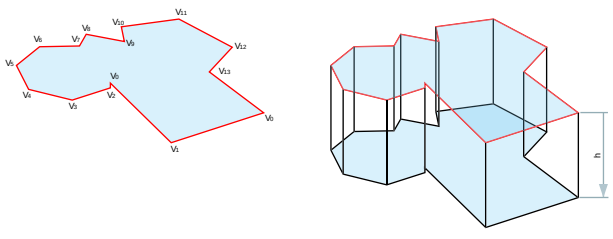
As part of the extrusion process, texture mapping coordinates are generated. Mapping coordinates define, by values of s and t, how textures are applied on the surface of the building. To arrive at an integer for s, we round the result of the horizontal facade length and divide by a random building window length between 2 and 4 meters. The integer for t is derived from the rounded quotient of the section length and the floor height. This process is repeated for each side of the building. Ten different textures, displayed in Figure 5(c), of single window styles are used to achieve the virtual city's look and feel. Texture coordinates for the roof of the building are taken from the normalized x and z coordinates of the floor plan iterations and multiplied by the amount of repetitions.

The top and bottom of each building has the shape of the corresponding floor plan, as illustrated in Figure 5(b). The building's top and bottom floor plans may be convex or concave. As OpenGL requires convex polygons the OpenGL Utility Library 1.2 (GLU) [Chin et al. 1998] tessellation algorithm is used to close each end.

4 Display List Caching

Procedurally generated geometry needs to be stored for use in subsequent frames in order to maximise real-time interactivity. Our approach is based on the use of OpenGL display lists as an intermediate storage that can be redrawn without the need to regenerate geometry procedurally.

Display list caching is based on the following assumptions: it is substantially faster to render a procedural shape than to actually generate it; geometry drawn in the current frame is likely to be drawn in subsequent frames (temporal coherence); the granularity of procedural geometry is fine enough to allow procedural generation interleaved with display; there is sufficient memory to store procedural geometry for reuse in subsequent frames.



(a) (b)

Figure 6: Floor plan extrusion: (a) floor plan (b) extruded floor plan

We implemented an OpenGL display list cache with 'least recently used' (LRU) replacement policy for the cache management in our pseudo infinite city. The cache is responsible for maintaining a set of recently used buildings, reusing older display lists for new buildings, compiling new display lists, and deleting old display lists as necessary.

Performance aspects of display list caching is examined experimentally in Section 5.2.

4.1 OpenGL Display Lists

An OpenGL display list [Woo et al. 1999] captures a sequence of OpenGL instructions. The same sequence of instructions are performed whenever the display list is used. Depending on the OpenGL implementation, calling a display list is usually more efficient than issuing the same instruction stream to OpenGL. However, display lists cannot be modified once compiled — so they may prove less suitable for dynamic geometry which continuously changes over time.

Display lists have several advantages in the context of procedurally generated geometry. Nearly all of the available OpenGL functionality is captured without the need for complicated data structures. This allows ease and flexibility of immediate mode procedural generation and rendering for the programmer, combined with the performance of compiled display lists in subsequent frames.

The performance advantage using display lists varies between OpenGL implementations, with the details of display list compilation hidden from the programmer.

4.2 LRU Cache

The total memory requirement and generation time is very high in a pseudo infinite procedural world. During navigation only a fraction of the world is visible, and it is only this part that needs to be generated. So-called 'lazy evaluation' allows procedural generation of geometry only as needed. Making use of a cache with a least recently used (LRU) replacement policy ensures that OpenGL display list resources are used efficiently, with the reuse of display lists least likely to be used in subsequent frames.

An LRU cache performs three main tasks: determining if a particular item is in the cache; inserting a new item into the cache; and determining the least recently used item in the cache. These queries need to be efficiently handled and not present a performance bottleneck for a real-time procedural world.

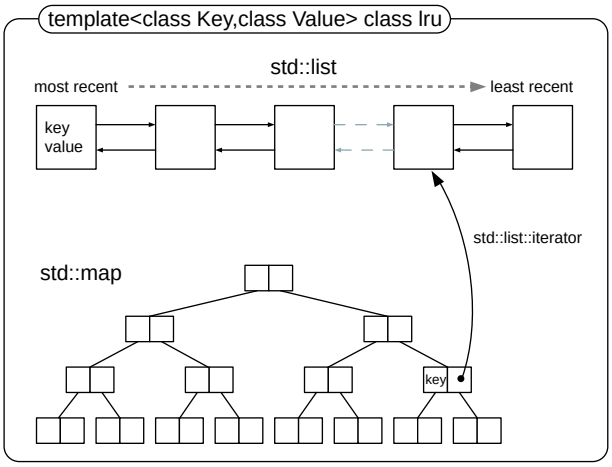


Figure 7: LRU cache using a list and balanced tree

4.3 LRU Cache Implementation

We implemented our `lru` container in C++ using a doubly linked list (`std::list`) container and a balanced tree (`std::map`) container from the C++ standard library as illustrated in Figure 7. These two data structures in combination allow tracking of the order of access and efficient queries.

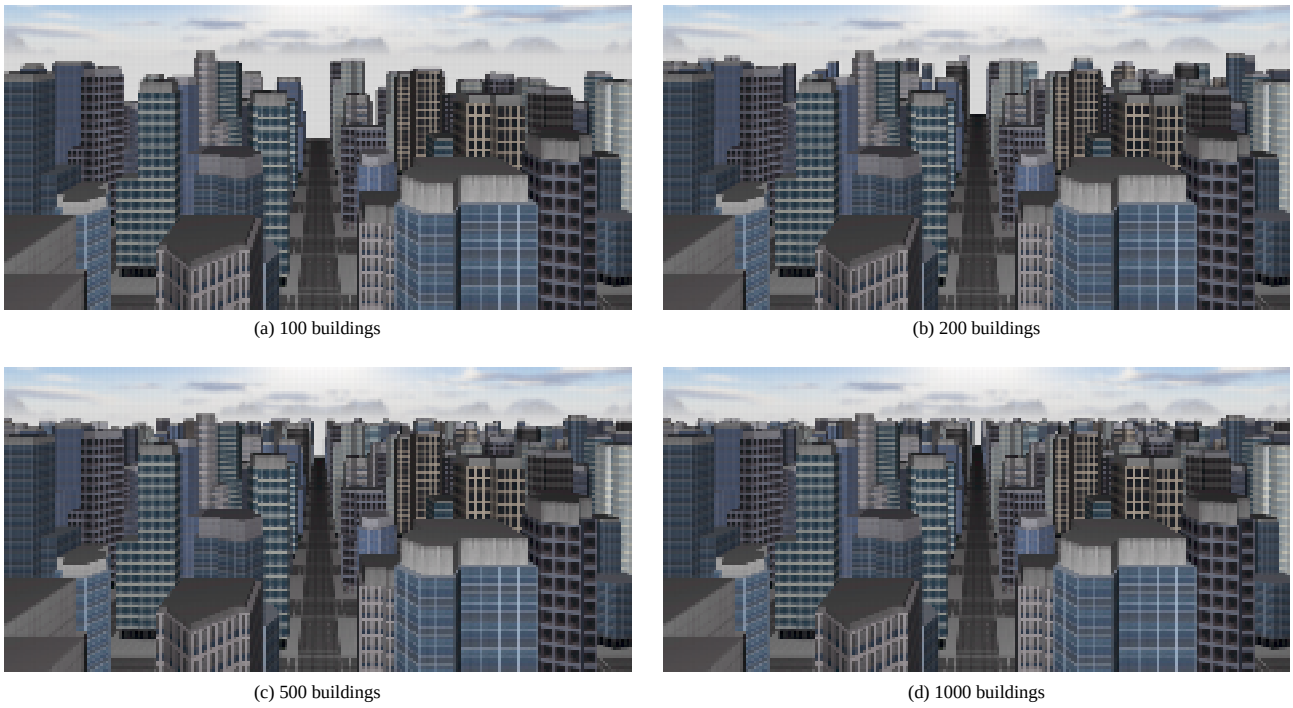
The list is sorted by order of access — items are moved to the front whenever they are queried, ensuring that less recently used items drift towards the end of the list. The `std::map` provides a fast index into the linked list. Querying a balanced tree requires $O(log\ n)$ time, whereas $O(n)$ time is required for a linked list.

The `lru` container stores pairs of *keys* and *values*. The *key* is the integer identifier of the procedural building and is used for querying the cache. The *value* is a collection of data including the OpenGL display list identifier and a time stamp corresponding to the time of last access. The `std::map` is used to look up list entries based on the *key*. Our C++ `lru` container is templated in order to support arbitrary *keys* and *values* for flexibility:

```
template<class Key,class Value> class lru
{
   typedef std::list<std::pair<Key,Value>>  List;
   typedef std::map<Key,List::iterator>     Index;

public:
   const uint32  size()  const;
   const Value  &front() const; // Most recent
   const Value  &back()  const; // Least recent
   const Value   pop_front();   // Most recent
   const Value   pop_back();    // Least recent

   Value *find(const Key &);
   Value &insert(const Key &,const Value &);
   Value &insert(const Key &);  // Recycle LRU item

private:
   List    list;
   Index   index;
};
```

Algorithm 2 Display list cache draw

while time stamp of LRU item $>$ maxAge **do**
  remove LRU item
  delete display list
query cache for requested item
if item already exists **then**
  move item to front of list
  update time stamp
  draw display list
else
  $a \leftarrow$ age of LRU item $>$ minAge
  $b \leftarrow$ age of LRU item $>$ 1 AND cache capacity exceeded
  if a OR b **then**
    reuse LRU item
    update time stamp
    compile and draw display list
  else
    insert new item
    update time stamp
    compile and draw new display list



(a) 100 buildings

(b) 200 buildings

(c) 500 buildings

(d) 1000 buildings

Figure 8: Procedural City Performance

4.4 Display List Cache Implementation

The display list caching algorithm is described in Algorithm 2. The behavior of the cache is configured via three parameters: the minimum age of an item before it is reused (*minAge*); the maximum age of an item before it is removed (*maxAge*); and the capacity of the cache (*capacity*).

Each item in the cache is time stamped with the number of the frame in which it was last used. Items 'age' when they are not being used and are available for reuse once they reach *minAge*. Items may be reused earlier if the cache capacity is exceeded. Items that reach *maxAge* are always removed. Note that the capacity of the cache is not strictly enforced, the cache will always retain items from the previous frame whether or not the desired capacity is maintained. We have used the following settings for our virtual city: *minAge* is 100 frames, *maxAge* is 500 frames, and *capacity* is 1000 items.

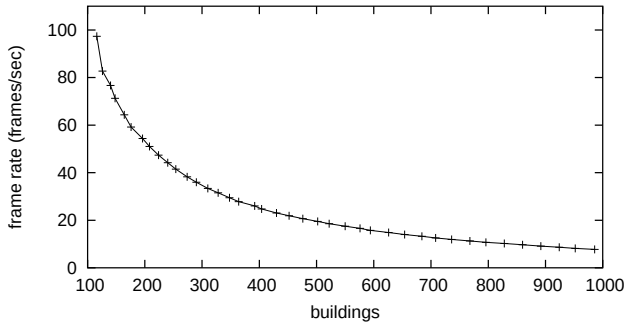


Figure 9: City performance

5 Performance Results

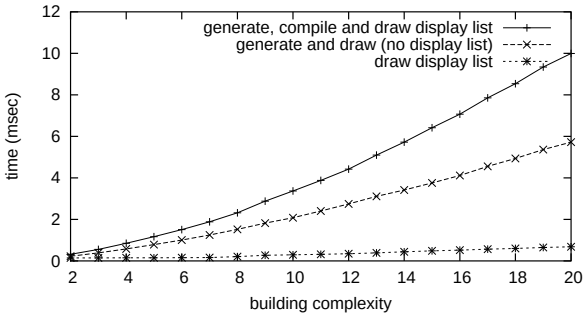
This section presents performance results of the virtual city, building generation algorithm and LRU cache. The city prototype is programmed in C++ using the standard C++ template library, OpenGL [Woo et al. 1999] and GLUT [Kilgard 1996]. The experimental platform is a 2 GHz Intel Pentium 4, 512 MB RAM, RedHat Linux 8.0 and NVIDIA GeForce 4 graphics hardware. The implementation is also portable to other UNIX platforms and Windows. Experiments were conducted using an 800x600 pixel OpenGL window in 32 bit color.

5.1 Procedural City Generation

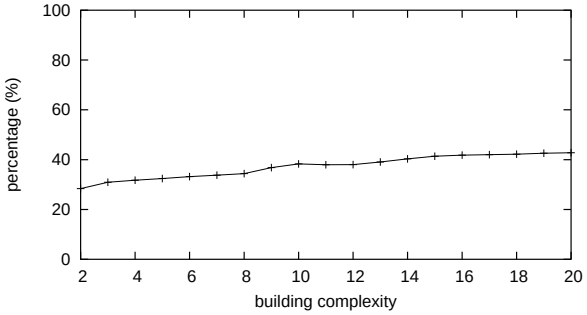
Frame rate is a common indicator of the overall performance of a real-time graphical application. Figure 9 shows the performance in frames per second with respect to the number of buildings displayed. The frame rate varies inversely with the number of buildings. Figure 8 illustrates the visual difference between a view of the city with 100, 200, 500 and 1000 visible buildings. A city with 200 buildings can be displayed at 60 frames per second, whereas a city with 1000 buildings can be displayed at only 5 frames per second.

5.2 Procedural Building Generation

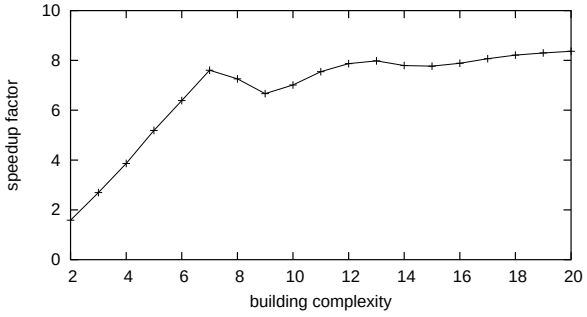
Display list compilation adds additional overhead to a building's generation time. We used a range of 'building complexities' to measure the time for building generation, display list compilation, and drawing. 'Building complexity' combines the degree of floor plan complexity with the maximum number of extrusion steps and serves as indicator for the highest polygon count of a building (worst case). A 'building complexity' of 10, for example, describes a building generated from a floor plan with 10 iterations and 11 extrusion steps. We used a sample population of one thousand random buildings for each measurement and a range of 'building complexities' from 2 to 20. All buildings were positioned to fill the viewport and the tests conducted with the depth buffer disabled.



(a) Generation, display list compilation and drawing



(b) Display list compilation overhead



(c) Display list drawing speedup factor

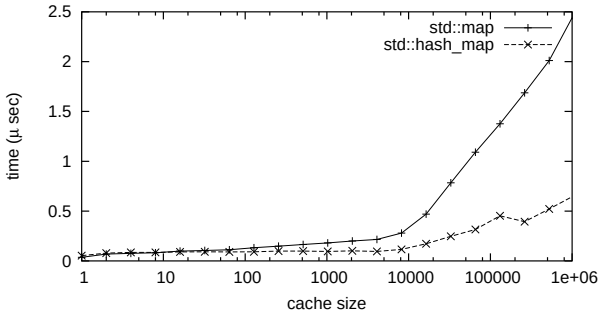
Figure 10: Procedural building generation and drawing

Figure 10(a) gives a comparison of time for building generation, display list compilation, and drawing from a pre-compiled display list for various 'building complexities'. The display list compilation time increases slightly with growing 'building complexity' but remains substantially less than the generation time. Figure 10(b) shows the display list compilation overhead in relation to the total display list compilation and drawing time. The overhead grows slowly with building complexity to around 40% for a high complexity building.

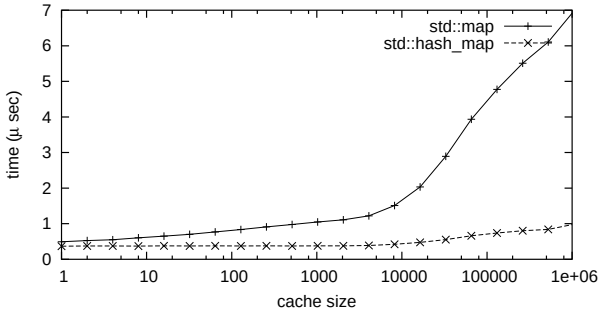
Buildings are typically redrawn hundreds of times without being regenerated. The longer display list compilation time is therefore more than compensated by a speedup in subsequent frames. The result of using display lists is plotted in Figure 10(c) and indicates speedups of up to eight, depending on the complexity of the building.

5.3 LRU Performance Characteristics

We conducted experiments to measure the overhead of cache management. In this experiment, the CPU time required by the `lru` container was measured in two separate scenarios: the time taken to look up an item already cached (cache hit) and the time taken to determine the LRU item and reuse it (cache miss). We measured these times for a range of cache sizes up to one million. We timed two different index data structures: `std::map`, and the hash-table based `std::hash_map` included with GCC version 3.2.



(a) Display list cache hit



(b) Display list cache miss

Figure 11: LRU performance results

Figures 11(a) and 11(b) show the time required by a cache query is several orders of magnitude less than the time required to display the actual item (Section 5.2). Depending on the frame coherence of the path through the world, cache misses are relatively infrequent and procedural building generation and display dominates.

These results show that the `std::hash_map` container has a particular performance advantage over `std::map` for extremely large caches. Both containers scale up to around ten thousand items gracefully. Overall, the caching scheme imposes little CPU load in the context of a pseudo infinite city. Use of a `std::hash_map` is preferable to a `std::map`, but neither can be expected to have a noticeable impact on the final frame rate.

6 Conclusion

An approach for procedural generation of pseudo infinite virtual cities, which achieve interactive frame rates on consumer level hardware, has been presented. A virtual city is generated in real-time on the fly using only a single integer seed as input. A display list cache with LRU (least recently used) replacement policy minimises procedural regeneration of buildings to make efficient use of memory and maintain a stable frame rate.

Experimental results indicate that LRU caching overhead is insignificant, particularly for buildings of high complexity. The longer compilation time is compensated by increased rendering performance of cached buildings in subsequent frames. When a building is first generated around 40% of processing time is spent on display list generation. In subsequent frames a speedup of up to

Figure 12: Street level panoramic view

eight is observed.

Real-time procedural generation poses further technical and artistic opportunities for exploration in the context of education, architecture, simulation, entertainment and art. Software aspects such as display list caching, hashing and random number generation provide a general purpose platform for seemingly unlimited and varied infinite worlds.

6.1 Future Directions

A complex structure like a city poses many interesting problems that we have not addressed. Further extensions could lead to more realistic results and better performance.

Virtual cities are ideal candidates for occlusion culling techniques. Our frustum filling algorithm could be extended to perform occlusion-based prioritisation. Buildings need not be drawn if they are occluded by nearer ones. The temporal coherence of the city could be used to accelerate computation in subsequent frames.

Large cities are often composed of a multitude of visually diverse and interesting buildings and places. We generated our virtual city with just one building type, an office skyscraper, and 10 different window textures. Many more building types such as houses, factories and schools, as well as areas such as park lands, industry and highways would enhance the realism and intricacy for applications in art, simulation and entertainment.

6.2 Acknowledgments

This research is partially funded by a scholarship from the German Academic Exchange Service (DAAD) and an International Postgraduate Research Scholarship (IPRS) from the Australian government and RMIT University.

References

ALEXANDER, C., ISHIKAWA, S., AND SILVERSTEIN, M. 1977. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press.

CHIN, N., FRAZIER, C., HO, P., LIU, Z., AND P.SMITH, K. 1998. *GLU The OpenGL Graphics System Utility Library*. Silicon Graphics Inc. http://www.opengl.org/developers/documentation/glx.html.

DANAHER, M. 2002. Dynamic landscape generation using page management. In *the 10-th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision '2002 - WSCG 2002*, 135–138.

KILGARD, M. J. 1996. *OpenGL Utility Toolkit Programming Interface API*. Silicon Graphics Inc. http://www.opengl.org/developers/documentation/glut.html.

LECKY-THOMPSON, G. W. 2001. *Infinite Game Universe: Mathematical Techniques*. Charles River Media.

MACRI, D., AND PALLISTER, K. 2000. Procedural 3D content generation. Tech. rep., Intel Developer Service. http://developer.intel.com.

MANDELBROT, B. 1977. *Fractale: Form, Chance and Dimension*. W.H. Freeman and Co.

MURTA, A. 2000. A general polygon clipping library. Tech. rep., University of Manchester. http://www.cs.man.ac.uk/aig/staff/alan/software/gpc.html.

OPPENHEIMER, P. E. 1986. Real time design and animation of fractal plants and trees. In *Computer Graphics (Proceedings of ACM SIGGRAPH 86)*, ACM, 55–64.

PALLISTER, K. 2000. Generating procedural clouds in real time on 3D hardware. Tech. rep., Intel Developer Service. http://developer.intel.com.

PARISH, Y. I. H., AND MUELLER, P. 2001. Procedural modelling of cities. In *Proceedings of ACM SIGGRAPH 2001*, ACM Press / ACM SIGGRAPH, New York, 301–308.

PERLIN, K. 1985. An image synthesizer. In *Computer Graphics (Proceedings of ACM SIGGRAPH 85)*, ACM, 287–296.

PRESS, W. H., TEUKOLSKY, S. A., VETTERLING, W. T., AND FLANNERY, B. P. 1992. *Numerical Recipes in C (2nd Ed)*. Cambridge University Press.

PRUSINKIEWICZ, P., LINDENMAYER, A., HANAN, J. S., FRACCHIA, F. D., FOWLER, D. R., DE BOER, M. J., AND MERCER, L. 1990. *The Algorithmic Beauty of Plants*. Springer-Verlag.

SO, C., BACIU, G., AND SUN, H. 1998. Reconstruction of 3D virtual buildings from 2D architectural floor plans. In *Proceedings of the ACM symposium on Virtual reality software and technology 1998*, ACM Press, New York, 17–23.

STINY, G. 1975. *Pictorial and Formal Aspects of Shape and Shape Grammars*. Birkhauser.

WANG, T. 2000. Integer hash function. Tech. rep., HP Enterprise Java Lab. http://www.concentric.net/∼Ttwang/tech/inthash.htm.

WOO, M., NEIDER, J., DAVIS, T., AND SHREINER, D. 1999. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*. Addison-Wesley.

YAP, C., BIERMANN, H., HERTZMAN, A., LI, C., PAO, H., AND PAXIA, T. 2001. A different manhattan project: Automatic statistical model generation. Tech. rep., Courant Institute, New York University. http://www.cs.nyu.edu/visual/.