Proceedings of the Third International Symposium on AI & Games, Daniela M. Romano and David C. Moffat (Eds.),
at the AISB 2010 convention, 29 March – 1 April 2010, De Montfort University, Leicester, UK

Procedural Content Generation and Level Design for Computer Games

Simon Cooper [1], Prof. Abdennour El Rhalibi [1], Prof. Madjid Merabti [1] and Jon Wetherall [2]

Abstract. Procedural methods show great potential but are an underused solution to manual content creation. Limitations to these methods include the lack of control of the output due to its random nature and the absence of integrated solutions, although more recent publications increasingly address these issues. This paper describes procedural methods applied to terrain modelling, including the variable realism of their output, performance and control users can exercise over the procedure. In this preliminary paper, we focus on the techniques and processes we used to create a web based tool in Java that quickly creates and exports a variety of stylized 3D computer game levels using procedural content generation with random and user selected constraints.

1 INTRODUCTION

Computer game development has moved on a long way in the last 20-30 years. As gaming machines become more and more powerful the average development time has significantly increased.

A large portion of this time is used to create sprawling levels with highly detailed assets, requiring game development teams to have over a hundred full-time people, including not only dozens of programmers but also equal numbers of artists and level designers. These assets also require large amounts of storage space on optical media, with read breaks in gameplay to stream new content in to the game engine or an installation to a local mass storage drive with greater memory access to counter this delay.

A way to reduce these issues would be to procedurally create assets programmatically or creatively reuse a small number of predefined assets and using recursive algorithms, Artificial Intelligence, rule systems, Iterated Function Systems, noise generation, random or pseudo-random processes and shape transformations in a manner such as to avoid obvious pattern repetition.

Our goal is to create a system that can procedurally generate content for a game level, using either random constraints or those set by a user before the generator is started. The final level should then be exportable to a generic file format, which could be loaded in at a later time, even in another game engine or programming language. The software we develop will be interoperable to be distributed on a variety of platforms including modern web browsers and multiple operating systems and system architectures; including Google Android in the future. Once completed the application will allow a game designer to set up parameters and initial constraints for procedural world generation and then define agent AI rules. The agent AIs (think of them as robots) will then adjust the game world, place pickups, objects, and setup game play using Rules imposed by the game designers.

The remaining chapters are as follows: In section 2 we discuss related work. In section 3 the techniques to generate the environment in particular terrain, plants and water. In section 4 we discuss possible algorithms to build road networks using L-systems or A* pathfinding. In section 5 we present the techniques based on shape grammars, developed to design city environment with procedurally generated buildings, textures and arranged primitives. In section 6 we briefly introduce the implementation with Homura (the engine we have used to develop the applications), the interface and the performance of the application. In section 7 we conclude the paper and highlight future work to be undertaken.

2 RELATED WORK

Other researchers have tackled procedurally generated game environments with varying degrees of success. The most well known and most significant research has been carried out on the CityEngine [1], a system that is capable of producing realistic and detailed models. The generation algorithms are inspired by the modelling of natural phenomena in string grammars [2] and L-systems are used to construct road networks and buildings [1]. Procedural building generation has focused on the application of grammars to describe structure, like the Shape Grammars original proposed by [3]. These have been applied in various guises to the construction of building geometry [4].

In [5], the authors define a specific grammar to characterise building structure. This system defines rules to operate on shapes and can be used to design a range of detailed buildings in several architectural styles. Recently this approach has been extended to employ imaging techniques to aid in the acquisition of generation rules from existing building façades [6; 7].

Other approaches include the application of intelligent agents [8], real-time frustum filling [9] and template based generation [10]. In [11; 8] the authors propose to simulate the evolution of cities by modelling land use and evolving a city usage map over time that can be used later to create a cityscape. Real-time city generation has also been attempted.
In [9; 11] the authors implement a city generation system that fills the view frustum rapidly with buildings of various shape combinations but their placement is restricted to a road network consisting of a regular grid. In [10] the authors propose the application of templates that encapsulate patterns such as raster radial to generate road networks.

Compared to these techniques our approach introduces several novel aspects including: interoperable

[1] School of Computing & Mathematical Sciences, Liverpool John Moores University, Byrom Street, Liverpool, L3 3AF, UK. Email: s.cooper@2003.ljmu.ac.uk, {a.elrhalibi, m.merabti}@ljmu.ac.uk.
[2] Onteca, Toxteth TV, 37-45 Windsor Street, Liverpool, L8 1XE, UK. Email: jon@onteca.com.

solution development allowing deployment across many platforms including web browsers, multiple OS and the Google Android platform; instant in-game generation, rendering and recreation of 3D tile based and organic environments; agent AI Bots to terraform the environment after the generation stage and according to other user-set constraints, and structural constraints induced by the game world; and full control of the environment generation by the user, as well as semi random generation options.

3 ENVIRONMENT

To generate a natural environment procedurally we use a variety of techniques, which are broken down into the sections presented below. These sections are the basic parts of a natural world and provide the building methods for land mass/terrain, seas, rivers & streams, and trees/plants.

3.1 Terrain & Midpoint displacement fractals

The terrain is created first, and determines where everything else can be placed, as every other part must be built up on top of it.

To create a random terrain we will use the midpoint displacement algorithm to create fractal height maps. Four corner points are chosen from the map grid and assigned random colour values between 0 and 1. The square is then subdivided equally into four equal squares and the midpoints of each edge are given a colour value equal to the average value of the two adjacent points (Figure 1). The centre point then gets its colour value from all four corners averaged with the addition of a random offset value, which is scaled by the size of the grid and a roughness variable.

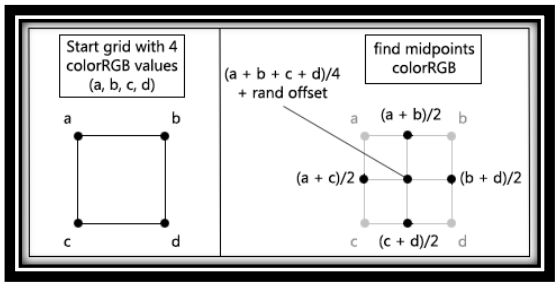

Figure 1 Midpoint Displacement Algorithm

The higher this roughness variable the more dramatic the peaks and troughs will be. This process is repeated to divide recursively until the smallest divisor/unit size is reached, which in this case is 1 pixel. This height map is then output as a black and white image with a grid size of 128x128. These numbers/pixel colours correspond to the terrain height at point [x, y] (or [x, z] in our 3d game world). The colour values are a percentage of the maximum height of land, with black representing 0% height - the lowest points and white representing 100% height - the high points.

3.2 Water

We split water creation up into the following three parts:

- The general flat water level created by the sea.
- The edge detection between sea and land creating the shore line/shallows.
- The generation of rivers and streams, which need to be routed around the map in a natural way.

3.2.1 Sea and Shoreline

For the sea we create a water level plane; the generated height map for the land mass can determine which parts of the map are above water level and which are submerged. The sea level depends on the range of terrain heights and a water level variable between 0 and 5, which divides the range of terrain heights in to five with zero being no water, five being flooded and the values in between setting the level at 1-4 fifths of the max height.

Once the land and sea are created we can add a shoreline of lighter water with an animated tide texture. To implement this we use textured quads placed just above the sea quad. An algorithm scans across each tile in the height map grid that is equal or less than sea level, performing a four-way check to see the surrounding tiles are above sea level. If one direction passes the tile is added to the checked list and not tested further. If the unchecked list has remaining tiles they are checked off one by one until it is empty. Next for each tile in the checked list a new quad will be created at the same coordinates with the direction of the shore tiles tide pattern determined by the sides that are touching land.

3.2.2 Rivers & streams

Rivers can be harder to create dynamically as their path heavily depends on the terrain. We can start with the assumptions that:

- Rivers will flow into a larger body of water (in our case the sea or in-land pools created by creators in our terrain) or off the screen if there are none present.
- Rivers will flow downhill due to gravity usually taking the easiest path.

Using these rules we can generate a random start and end point for the river, either on one edge of the map, in the sea or in a lake pool. Next we can set an A* pathfinding algorithm to compute the route between these two points, using a custom ‘cost heuristic’, similar to a possible technique for road generation (see section 4.2 for more detail). The heuristic will look for the most cost effective path, with steps taken downhill being cheaper.

3.3 Trees & plants

To add more features to our map we add trees and plants. The problem is defining where to place them in a pattern that looks natural. If the trees and plants are scattered using ‘n’ randomly generated [x, z] positions within our grid the results can look “neither random nor natural”.[12] Due to the way trees and plants grow: from seeds falling; to growing and taking in nutrients, water, CO2, light and competing with other plants, their scattering pattern can look more complicated. A simple model for this pattern would include each plant having an exclusion zone around it that no other plants can occupy. Plants and trees can be arranged in a grid with an equal distance between them before a small random offset is applied to scatter them. This offset must also be small enough to stop two trees overlapping. More variation can be added later by randomly rotating and scaling the trees

(including their exclusion zones), as some trees will be more fully-grown than others.

4 ROADS

Once the land is created it can be populated with an urban city structure. The first main step is to divide up the level into lots (smaller areas that can be used to place buildings and other man made areas such as parks or town squares) by forming a road network. There are two main structural layouts for the roads, which are dependent on the area being in an urban or rural setting.

For a big city the gird plan road structure is most fitting, but in rural areas the streets need to be non-uniform (smaller, more winding and with further separation). As the land and natural features are determined first, the height map from terrain generation can be used to control where the roads can and cannot be built. Features like water and hill gradients can be factored in to exclude roads or define points for bridges and tunnels.

There are two useful techniques we could use to generate the roads: L-systems and A* Pathfinding. These could be used individually to give different results or be used together in passes to create roads more intelligently.

4.1 L-Systems

An L-system or Lindenmayer system is a parallel rewriting system using string rewrites, where each symbol is interoperated as a rule step. They can be used to generate self-similar fractals. An L-system contains: variables, constants, a start string (*start*, *axiom* or *initiator*) and rules. The original L-system was used to model the growth of algae and can be seen in Figure 2 below.

This L-system contains:
Variables: A B
Constants: none
Start: A
Rules: (A → AB)
(B → A)

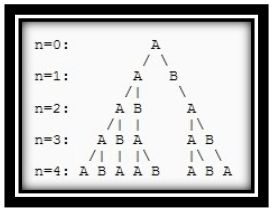

Figure 2 Original L-System for modelling the growth of Algae

We use this system to create our roads starting with a small line drawn in the initial forward direction. As the steps for 'n' increase more lines will be added and for each one a random number check is done with a small chance of returning true. If the test returns true up to three branches will be created in the north, east and west directions from the original point at around 90 degrees to create a block pattern (this number could be changed, or have a random offset to give different road patterns).

To create a rural non-linear road system the distance for each line drawn every 'n' could be shortened and the maximum number of branches reduced from 3 to 2. Also as mentioned previously the chance of the angle changing could be increased and the angle value varied to create a more curved or 'dog-leg' shaped road.

To make the road creation more dynamic we overlay the height map from the terrain whilst building the road [13]. Areas that have water could be impassable or require a bridge and the gradient can be checked to stop roads having too steep an incline and force them to spiral uphill gradually. We also add a population density map, which can control where main roads are build and the density of the roads. These rules can all be added to the drawing process 'A' mentioned above and alter the return true statement for adding new branches.

Our L-system is structured in the following way (Figure 3):

Variables: X A
Constants: + −
Start: X
Rules: (X → [[-AX] +AX] + AX)
Angle: 90°
X = road growth control variable
'[' = Push
']' = Pop
'-' = Turn Left Angle°
'+' = Turn Right Angle°
'A' = if random check true draw line of distance 'D' in forward direction.

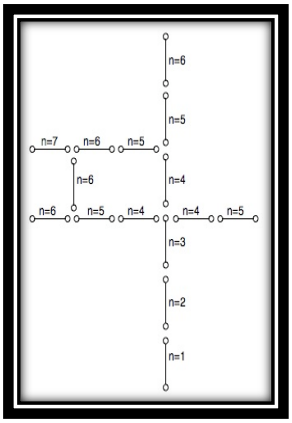

Figure 3 L-system for grid block system roads

4.2 A* algorithm and heuristic pathfinding

Pathfinding is used to navigate from one point to another whilst avoiding any obstacles in the way by finding a suitable route around them. A popular algorithm for this is A*, which can run in real time on a reasonable sized map. As our roads are generated in an editor a slight delay is not as big an issue as it would be in other cases such as an enemy AI in a shooter game. We can scatter points around the map then connect them using pathfinding to create the roads along suitable routes [14; 15].

A heuristic developed by multiple authors for the game 'Open Transport Tycoon' devised a logical cost heuristic for a road-building pathfinder, which also includes provisions for turns, slopes, bridges and tunnels [16].

5 BUILDINGS

When the roads are placed the shape of the lots between them can determine where the buildings can go.

5.1 Shape grammars

Shape grammars are used to provide a computational approach to the generation of designs. In architecture, building structures have many repeating shapes and patterns that can be described using a rule. Shape grammars consist of:

- Shapes – the structures building blocks.
- Spatial Relations – how these shapes are positioned relative to one another.
- Rules – to describe how and where new shapes are added.
- Rule Labels – give a set orientation marker to the rule.
- Transformations – translation, rotation and scale of the shapes.
- Derivation – repeat the rule for n steps to create multiple designs.

Figure 4 shows a labelled rule for a rectangle. For each step a new rectangle is added with a clockwise rotation of 90° and aligned to the original shapes narrow edge (labelled with a dot). The pattern repeats for 3 steps, forming a closed square shape with a gap in the centre.

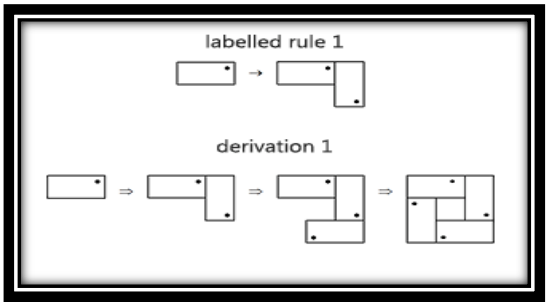

Figure 4 Labelled Rule Example 1 [17]

Figure 5 shows that with exactly the same shapes that changing the label dot, thus changing the rule, can produce a different structure. This time the new rectangle is rotated 90° in an anticlockwise direction and aligned with the short marked edge.

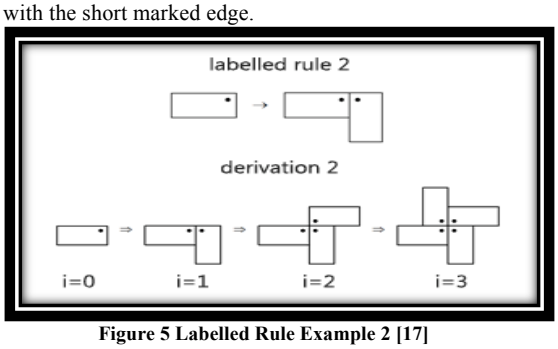

Figure 5 Labelled Rule Example 2 [17]

Using these examples with randomly sized boxes can produce some interesting tower block shapes for the city buildings. The number of steps n can be also set randomly, so just by using the two rules above 8 different structures can be made and when combined with the randomly sized boxes this creates hundreds of different buildings.

To create cylindrical office buildings we consider the technique used in [18]. A circle is generated in 10° slices, with a random chance of skipping ahead 90°, creating a bigger slice and a flat edge. This process continues with up to three 90° skips, until the total rotation is a full 360° circle. The circle can then be extruded upwards with a random height value and capped to create a cylindrical office building.

We experimented with shape grammars to also create smaller houses and other buildings such as a church. Figure 6 below shows two basic house shapes (A & B) created in 3DS Max and loaded into our game engine as a shared mesh which can be cloned.

Using some labelled rules and a combination of these two shapes (along with a thin box for a chimney) many more complicated houses can be built (as depicted in C, D, E, F, G & H). For example the house 'D' is a combination of 'B' and 'A' rotated 90° clockwise. House 'H' is more complicated using translation, rotation and scaling on four instances of house 'A' along with the originally positioned house 'B' and two chimney boxes.

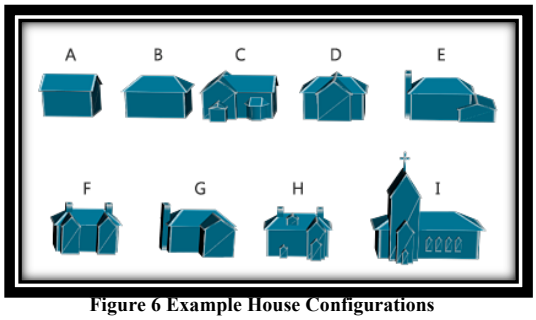

Figure 6 Example House Configurations

5.2 Procedural Textures

The textures used in the scene are generated in two main ways for the land, trees, water and buildings:

- Existing textures produced before hand by an artist in the PNG format are dynamically chosen and scaled according to the object and a set of rules.
- The texture is created procedurally using AWT image graphics and saved as a texture state, which then can be treated as the predesigned textures above.

To create the textures for our buildings we use the latter option to create a series of windows in different sizes, colours and designs. To make the process easier to start off with, we use a procedure similar to the one described in [19], where buildings are dark to mask the missing detail and defined by lit windows, which gives a night time city effect. A 512x512 texture is reserved in memory and each pixel is coloured in several stages:

- First determine the number of windows across and down and what their size is when equally dividing up the texture.
- Set a base colour for the window that is dark (unlit), white or orange (lit). These colours also have a random offset to give a small variation in tone.
- There are two settings for the distribution of the lit windows to simulate either residential tower blocks or offices. The residential buildings have a random scattering; the lit and unlit windows are selected using the math library's next random integer function, with the probability set to roughly 1:3 lit to unlit and an equal chance for white or orange to be chosen for the lit windows. The offices are lit in random blocks consisting of up to a whole floor of windows as companies usually occupy a whole floor or several units and some are open later than others.
- Next for each pixel row the selected base colour is modified by subtracting a small value each time to create a light to dark gradient. This gives the windows more depth and matches the light sources, which are usually at the top of a room.
- To create more variation a per-pixel random colour noise is added.
- Now adding black lines or pixels can create different window shapes/designs and for smaller houses a separate texture can be created with fewer windows spread out over a wider space.

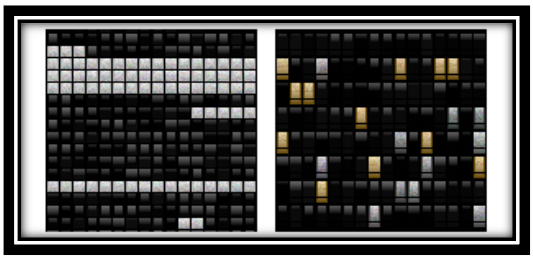

Figure 7 Procedural Window Textures

To create more variety we consider an assortment of window shapes and sizes, especially for the smaller houses textures. Houses have less windows and light, with a more varied array of designs including the bay styled windows.

6 IMPLEMENTATION

6.1 Homura Game engine

We developed our test applications using the Homura Game Engine, another research project we are currently undertaking. The Homura project's development framework provides an Open Source API for the creation of Java and Open GL based hardware-accelerated 3D games applications, which support cross platform, cross-browser deployment using Java Web Start (JWS) and Next-generation Applet technologies. The framework bundles together several example applications and technical demos, which demonstrate how to implement common games functionality in your application; An application template, which acts as a great starting point for development your own Homura related games; The APIs of both Homura and the key open-source projects it builds upon including the Java Monkey Engine scene graph API, jME Physics Library, MD5 Model Importer, GBUI User Interface Libraries and many more; External Tools for the creation of Font Assets, Particle Effects and Levels for the games [20].

We have implemented a user interface using swing. Some controls are provided to the user to constrain elements of the level generation, including the option to turn off some generated features in a level or completely randomise the generation for a two-click level design.

Figure 8 Swing User Interface Layouts

Some example settings that would be useful are:

- Terrain Height – flat/large peaks.
- Terrain Roughness – flat/many peaks & troughs.
- Water Height – no water/some water/flood.
- Tree Density – no trees/ many trees.
- Texture theme – Grass, Desert, Snowy
- Rebuild Level – Rebuild geometry using the above settings.
- Show Tweening Animations – on/off.
- Buildings – on/off
- Population Density - City/Rural
- Roads, Bridges, Tunnels – on/off toggle each.

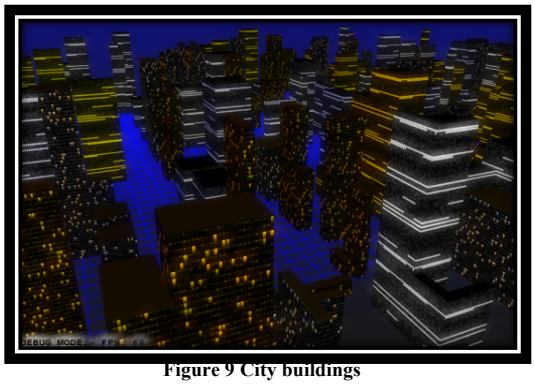

Figure 9 City buildings

6.2 Optimising geometry

To optimise the creation of our geometry we use shared meshes. There are many objects in the scene that are copies of each other and shared mesh allow us to share that data between multiple nodes. "A provided TriMesh is used as the model for this node. This allows the user to place multiple copies of the same object throughout the scene without having to duplicate data." [21] Each copy node has unique translations, rotations, scales and render-states applied to them. Another useful feature to include is object pooling. We can initialise a store of objects in a pool when the game starts for objects like new buildings, trees and textures; so that when one is created it can be quickly taken from this pool.

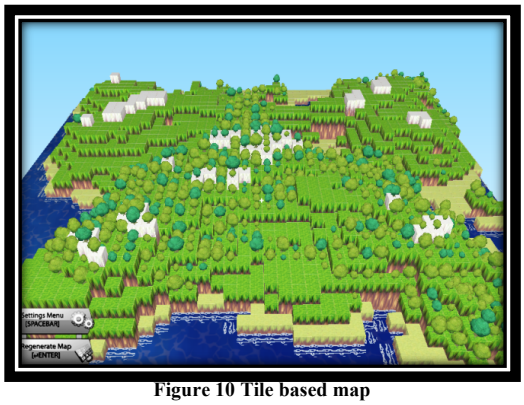

Figure 10 Tile based map

6.3 Performance evaluation

After completing a test application that runs a tile based map generator we had compelling results. The actual generation of a new level only takes milliseconds, so a tweening animation was added to show how the level is built up in layers, similar to a scan line renderer. The variety of the generated levels is noticeable with different types of water levels, landscapes & tree scattering. At this point in time the buildings are created in a separate application, which will be merged with the environmental one when it is nearer completion. The frame rate runs steadily at around 30fps on a regular specification computer, and much higher through the WebStart version, but deters proportionally with the size of the map being requested. An optimisation for running these maps in a game environment would be to overlay a better scene management system such as a Binary Space

Partition Tree, Octree or Clipmap [22; 23]. The other aspects we have considered are:

- Novelty: contains element of randomness and unpredictability.
- Structure: is not merely random noise, but contains larger structures.
- Interest: has a combination of randomness and structure that players find engaging.
- Speed: can be quickly generated.
- Controllability: can be generated according to a set of natural designer-centric parameters.

The program is still under development and a full usability evaluation will be performed once it is completed, to analyse users' reaction to and satisfaction with the tools.

7 CONCLUSIONS & FUTURE WORK

This paper describes procedural methods applied to terrain modelling, including variable realism of their output, performance and control users can exercise over the procedure.

In this preliminary paper, we focus on the techniques and processes we used to create a web based tool in Java that quickly creates and exports a variety of stylized 3d computer game levels using procedural content generation with random and user selected constraints

In future work we will produce with the game company Onteca, a demonstrator running on a commercial platform. The system will allow a game designer to set up parameters for procedural world generation, initial constraints and then define agent AI rules. The agent AIs (think of them as robots) will then adjust the game world, place pickups, objects, and setup game play using Rules imposed by the game designers. This demonstrator will allow us to explore whether 'fun' can be automated.

8 REFERENCES

1. D. Bekins, D. A. 2005. "Build-by-number: Rearranging the real world to visualize novel architectural spaces". In IEEE Visualization, 143– 150.
2. Greuter, S., Parker, J., Stewart, N., AND Leach, G. 2003. "Real-time procedural generation of 'pseudo infinite' cities". In Proceedings of Graphite 2003, ACM Press, New York, USA.
3. Lechner, T., Watson, B., Wilensky, U., AND Felsen, M. 2003. "Procedural city modelling". Tech. rep., Northwestern University, November.
4. Lechner, T., Watson, B., Ren, P., Wilensky, U., Tisue, S., and Felsen, M. 2004. "Procedural modelling of land use in cities". Tech. rep., Northwestern University, August.
5. Muller, P., Wonka, P., Haegler, S., Ulmer, A., and Gool, L. V. 2006. "Procedural modelling of buildings". In Proceedings of ACM SIGGRAPH 2006, ACMPress, New York, NY, USA, 614–623.
6. Muller, P., Zeng, G., Wonka, P., AND Gool, L. V. 2007. "Image-based procedural modelling of facades". In SIGGRAPH '07: ACM SIGGRAPH 2007 papers, ACM Press, New York, NY, USA, 85.
7. Parish, Y. I. H., AND Muller, P. 2001. "Procedural modelling of cities". In Proceedings of ACM SIGGRAPH 2001, ACMPress, New York, NY, USA, 301–308.
8. Prunsinkiewicz, P., and Lindenmayer, A. 1990. "The Algorithmic Beauty of Plants" (The Virtual Laboratory). Springer-Verlag, New York, New York, NY, USA.
9. Stiny, G. 1980. "Introduction to shape and shape grammars". Environment and Planning B: Planning and Design 7, 343–351.
10. Sun, J., Yu, X., Baciu, G., and Green, M. 2002. "Template-based generation of road networks for virtual city modelling". In VRST-02, ACMPress, New York, NY, USA, H. Sun and Q. Peng, Eds., 33–40.
11. Wonka, P., Wimmer, M., Sillion, F., AND Ribarsky, W. 2003. "Instant Architecture". In Proceedings of ACM SIGGRAPH2003, ACM Press, New York, NY, USA, J. Hodgins and J. C. Hart, Eds., vol. 22(3), 669–677.
12. Mick West. (2008, August) Gamasutra. [Online]. http://www.gamasutra.com/view/feature/1648/random_scattering_creating_.php
13. Lintfordpickle. (2009, August) Britonia Game Blog. [Online]. http://britonia-game.com/?p=28
14. Patrick Lester. (2005, July) gamedev.net. [Online]. http://www.gamedev.net/reference/programming/features/astar/
15. Kevin Glass. Coke & Code. [Online]. http://www.cokeandcode.com/pathfinding
16. Contribs. Various. (2009, August) Open TTD Wiki. [Online]. http://wiki.openttd.org/AI:RoadPathfinder
17. Terry Knight. (2002, March) MIT Open Courseware. [Online]. http://ocw.mit.edu/NR/rdonlyres/Architecture/4-184Architectural-Design-Workshops--Computational-Design-for-HousingSpring2002/285C5722-AAFC-4DA7-A744-49F31BFE069D/0/fixlecture2slides_final.pdf
18. Shamus Young. (2009, April) Twenty Sided Tale. [Online]. http://www.shamusyoung.com/twentysidedtale/?p=2968
19. Shamus Young. (2009, April) Twenty Sided. [Online]. http://www.shamusyoung.com/twentysidedtale/?p=2954
20. LJMU CMP Research Team. (2009, Dec.) Homura Games Framework. [Online]. http://java.cms.livjm.ac.uk/homura/about.php
21. Mark Powell. JME Java Docs. [Online]. http://www.jmonkeyengine.com/doc/com/jme/scene/SharedMesh.html
22. Joshua Slack. (2009, November) renanse.com. [Online]. http://blog.renanse.com/2009/11/terrain-example.html
23. Christopher C. Tanner, Christopher J. Migdal, and Michael T. Jones, "The Clipmap: A Virtual Mipmap," in Proceedings of the 25th annual conference on Computer graphics and interactive techniques, NY, 1998, pp. 151-158.