

# Procedural City Generation Tool



A Project Presented to the Faculty of The Guildhall  
at Southern Methodist University

By Kevin Wu

B.S., University of California, Irvine, 2006

In Partial Fulfillment of the Requirements for a Masters of Interactive  
Technology in Digital Game Development with a Specialization in Software  
Development

12/12/08

To the Graduate Faculty:

I am submitting herewith a project written by Kevin Wu entitled “Procedural City Generation Tool.” I recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Interactive Technology in Digital Game Development, with Specialization in Software development.

---

Jeff Wofford, Supervisor

I have read this Project  
and recommend its acceptance:

---

Gary Brubaker, Advisor

Accepted for the Faculty:

---

Dr. Peter Raad, Executive Director  
The Guildhall at SMU

## ACKNOWLEDGEMENTS

I would like to thank my parents and my girlfriend for their constant support, Professor Jeff Wofford, Professor Gary Brubaker, and Professor Wouter van Oortmerssen for their teaching and guidance, and cohort 9 programmers for their feedback.

Kevin Wu

M.I.T., The Guildhall at SMU, 2008

Procedural City Generation Tool

Supervisor: Jeff Wofford

Master of Interactive Technology degree conferred December 12, 2008

Thesis / Project completed December 12, 2008

There is no argument that a designer can create a level that is more interesting than a level created by an algorithm. There is also no argument that algorithms can perform certain tasks much more efficiently than people can. As game scope expands, the asset creation becomes ever more time consuming. Procedural content generation allows artists and level designers spend more time improving and polishing important parts of the game and lets algorithm generate areas of lesser importance. The objective for this project is to incorporate procedural city generation with traditional level design tools to create a system that can generate random city efficiently yet still be customizable. The first component of this project is a procedural city generator that allows the user to specify the road patterns and their influence on the layout of city blocks and buildings. The second component of this project is a set of tools that allow the user to modify the procedurally generated city.

# Table of Contents

List of Figures .....	vi
List of Tables .....	vii
Nomenclature .....	viii
Chapter 1: Introduction .....	1
Chapter 2: Field Review .....	3
2.1 Procedural Content in the Industry .....	3
2.2 Procedural City Generation Methods.....	4
2.2.1 Road Generation .....	5
2.2.2 Building Generation.....	6
Chapter 3: Methodology .....	8
3.1 Quadtree.....	8
3.2 Procedural Tool.....	10
3.2.1 Road Generation .....	10
3.2.4 Block and Building Generation .....	15
3.3 Editing Tool .....	17
3.3.1 Moving Control Points.....	17
3.3.2 Add Road .....	19
3.3.3 Remove Road.....	19
3.3.4 Edit Block .....	21
3.3.5 Save and Load.....	22
Chapter 4: Results and Analysis .....	24
4.1 File Size and Efficiency .....	24
4.3 Randomness .....	25
4.4 Customizability and Realism .....	26
Chapter 5: Conclusion.....	28
5.1 Future Work .....	28
References.....	31

# List of Figures

Figure	Page
3.1 Example quadtree.....	9
3.2 Raster pattern and radial pattern .....	11
3.3 Blending road patterns .....	13
3.4 Boundary check .....	14
3.5 Intersection check .....	14
3.6 Creating block.....	15
3.7 Moving control points.....	18
3.8 Add road.....	20
3.9 Remove road .....	21
4.1 Randomness .....	26

# List of Tables

Table	Page
3.1 Save file format.....	22
3.2 Change size .....	23
4.1 Profiled Data .....	25

# Nomenclature

**Quadtree** – a data structure that recursively divide the 2D space into four quadrants

**Map** – a 2D array of data or a texture

**Distribution map** – map used to determine where a road pattern goes

**Boundary map** – map used to determine if a specific location is legal or not

# Chapter 1: Introduction

As video game and PC hardware becomes more powerful, the size of games increases proportionally to match it. Game worlds expand in size, and art assets increase in detail. Similarly, project schedules and costs go up, motivating game developers to adopt procedural content generation. Rather than having artists and designers hand-craft multiple variations of an asset, a system is created that takes simple parameters and generates assets on demand. For example, many developers use *SpeedTree* (Interactive Data Visualization, 2002), a software package that generates virtual foliage and forests for use in games. Procedural generation has been used to produce many types of assets ranging from textures to landscape terrains, but one is desperately needed for cityscape.

Procedural city generation can greatly benefit simulation games such as the *SimCity* (Maxis Software, 1989) series and RTS games where building placement is part of the game, and for games that require an extremely high level of polish and human touch, it can still be used to complement existing tools to make the production process easier. For city-based sandbox games such as *Assassin's Creed* (Ubisoft Montreal, 2007) and *Grand Theft Auto IV* (Rockstar North, 2008), having level designers block out each building and place it in the level to form a city is a tedious and time-consuming task. To make things worse, many locations won't even be seen by the players since people mostly hang around important quest hubs and destinations. There is no argument that a hand-crafted asset could be higher quality than something generated by an algorithm, but there's no reason why the two methods could not be combined. Ideally, areas of lower interest can be generated procedurally, and developers can spend more of their time on polishing the key locations to make those even better. The same applies to not just sandbox games but any game that takes place in a city. First-person shooters and third-person shooters often feature city stages. Street racing games contains tracks all around the city. Games of any genre, even flight simulators, can benefit from such a system.

Realistic city layout can be procedurally generated. The objective for this project is to incorporate procedural city generation with traditional level design tools to create a system that can generate random cities efficiently while allowing them to remain customizable. The

system can generate a realistic city from scratch using simple parameters, but it also features tools that can modify it every step of the way. If the user is not satisfied with the street layout or building configuration generated by the system, he will be able to add, delete, or modify them as he sees fit.

# Chapter 2: Field Review

## 2.1 Procedural Content in the Industry

Most games that use procedural content generation use it to generate levels. Though not the first to implement it, the *Diablo* series (Blizzard Entertainment, 1997, 2000) is probably the most popular game to feature procedural levels. Each level is composed of a tile set that is pre-made by designers offline. For example, the jungle level features a collection of tiles with grass and trees, and the desert level is composed of sand and dunes tiles. Each time a player enters a level, tiles are selected from the level set and rearranged randomly while matching the edges of each tile to make sure there is no discontinuity. Dynamically putting together the level in this way increases a game's replay value. Every time a player enters the same level, the experience varies because the map is always different. The disadvantage of generating levels this way is that even with a large number of tiles, players learn to distinguish each tile and may be able to tell where one tile ends and the next one starts. This diminishes the player's sense of immersion into the game world. Also, the randomness of the layout can actually make a level less finely-tuned in terms of gameplay.

According to its developers, the game *Spore* (Maxis Software, 2008) procedurally generates almost all of its contents. The game follows the evolution of the player's creature from a single cell organism to a planet conquering race, and everything along the way will be generated procedurally. For example, during the creature phase, players get to design their own creature. If the player gave his creature four legs, a gallop animation will be generated, and the creature will run like a horse. However, if the player decides to create a three-legged monster, the game will determine how a creature would walk with three legs and generate the animation accordingly. In fact, everything about the creatures will be procedurally generated from only a couple kilobytes of information. "Think of it as sharing the DNA template of a creature while the game, like a womb, builds the 'phenotypes' of the animal, which represent a few megabytes of texturing, animation, etc", says Will Wright (Wikipedia, Development of *Spore*).

Another game that is completely generated procedurally is *.kkrieger* (.theprodukkt, 2004), a first person shooter with a 97,280-byte executable. Everything in the game is procedurally generated during load time including the models, textures, and animations. Although *.kkrieger* uses procedural generation, it does not randomize the output: the produced content is always identical. In this case, procedural generation is used as a form of compression rather than dynamic content. This compression mechanism is highly effective. If procedural generation were not used, the game would be between 200 and 300 megabytes.

Also created by .theprodukkt, *.debris* is a demo featuring a procedurally generated city. Upon execution, the 177 kilobytes program generates models and textures that rival most games today along with over seven minutes of animation. The demo also features lighting, shadows, and other advanced visual effects. Similar to *.kkrieger*, procedural generation is used purely as a form of compression.

## 2.2 Procedural City Generation Methods

Procedurally generating a city has not been done in games, but it has been done in academia and research for years because of its usefulness in city planning and reconstruction. All the algorithms can generally be divided into two categories. The first type of algorithm takes actual photos of city streets and buildings and extracts the street layout and building structures from them. The second type takes geometric and visual parameters and generates a city entirely from scratch. Both types have application in the industry. For games that require accurate replication of an entire city, such as racing games, the first method is ideal. However, in most games, having the level completely mirror reality is not desired. This project is devoted to research into the second type of algorithm.

Academic research into this area has led to the conclusion that generating a city can be divided into two steps. The first step generates the street layout, and the second step populates the resulting layout with buildings. The following sections discuss resources on the respective topics. Papers that cover both road and building generation will be covered in each section.

## 2.2.1 Road Generation

“Procedural Modeling of Cities” (Parish and Müller 2001) describes a system called CityEngine which procedurally generates an entire city from the streets to each individual buildings. The system takes as its input a series of textures that describes the city. For example, a water and vegetation map tells the algorithm where not to generate roads, and a population density map that influences the road to go towards crowded areas. A map of street pattern influences the street layout. A New York style raster pattern creates streets that form a grid, and a Paris style radial pattern encourages roads to form rings. To generate the roads, the CityEngine adopts the L-system, a parallel string rewriting system that traditionally was used to model the growth of plants and organisms in biology. Together with a set of global and local constraints, the system is able to grow a street network. The global constraint orients the road to grow towards high populated area, and to follow the dominant street patterns of the area. The local constraint checks the road against illegal areas such as water and search for nearby intersections. If an intersection is found, the road is automatically snapped to it. If the local constraint cannot be satisfied, the road segment is deleted. Once the road generation is complete, a simple algorithm divides each block into lots, and a building is generated on top of each lot. The building generation strategy will be covered in the next section.

In “Template-Based Generation of Road Networks for Virtual City Modeling” (Sun et al. 2002), the authors proposed a simpler version of CityEngine’s road generation system. Rather than using the L-system which is not very extensible, the authors use road templates, which essentially serve the same function as CityEngine’s road patterns. The template could be raster, radial, or population based. Raster and radial is identical to CityEngine’s road pattern. Population based template searches the input population map for peaks and create a Voronoi diagram from those points, and the edges of the resulting Voronoi diagram becomes the roads. Rather than growing the entire street layout all at once, the algorithm generates the highways first then streets. The highways are generated along the template but checked against boundaries. If it extends into illegal zones, the border of the zone is found, and the highway is modified to run along the border. Once the highway generation is complete, streets are generated using the raster template on the regions bound by highways. The

algorithm, although simple, does not achieve the same result as the CityEngine. Roadmap generated with this algorithm seems less organic.

Though not directly related to procedural generation of city, “Modelling virtual cities dedicated to behavioral animation” (Thomas and Donikian 2000) offers some interesting ideas that can be applied to game levels in general. As roads are placed in the world, intersections are detected and placed in a topological graph. Bounding boxes are created that uses a routing matrix to direct entities inside the box where they should go. This is done for each intersecting lanes of the two roads since lanes of the same road can go in opposite directions. On top of roads, traffic signals and road signs can be placed in the world that further modifies entities’ behaviors. When entities such as pedestrians and cars are placed in the world, they move around according to the traffic rules. Essentially, the behaviors of entities are baked into the map.

As previously mentioned, procedural content can be generated offline, during load time, or dynamically on the fly. Even though the intended system is meant to be used offline, the methods described in “Real-time Procedural Generation of ‘Pseudo Infinite’ Cities” (Greuter et al. 2003) are still interesting and worth mentioning. To make real-time generation possible, the article uses view frustum filling, a technique that works in the opposite manner as view frustum culling. Normally when an entity is about to be drawn, its bounding volume is checked against the view frustum for intersection. If the volume intersects the frustum then it is in the scene and needs to be rendered. Rather than checking each building against the frustum to see if it is inside, only buildings inside the frustum is generated. The entire world is divided into a two-dimensional grid. Grids that are inside the frustum are hashed to a value that is used to seed a pseudo-random-number generator, which is in turn used to generate the building. As the frustum moves, new grid enters the frustum, and the generated building is inserted into a list. Buildings built on top of grids that are no longer visible are destroyed, so everything in the list is always visible. The building generation will be covered in the next section.

### **2.2.2 Building Generation**

In “Procedural Modeling of Cities”, Parish and Müller generates the buildings of the city by using a second L-system that modifies the shape of the buildings. Each building

starts out as a bounding box, and through iterations of the L-system, more and more details are generated. Finally, the facades are divided both horizontally and vertically to generate a grid structure that allow textures such as windows to be aligned by floors and/or columns. An added bonus of using this system is the free level of detail. The initial bounding box is the lowest LOD, and each iteration of the L-system increases the LOD by one level.

In “Real-time Procedural Generation of ‘Pseudo Infinite’ Cities”, a random geometric shape is added to the floor plan, and the floor plan is extruded by a random amount. The same steps are repeated multiple times, and a building with towers of different height is generated. The building is then textured with windows to make it more realistic. The algorithm is simple, but the generated buildings are only suitable for small office buildings and skyscrapers and look rather generic.

In “Instant Architecture” (Wonka et al. 2003), the authors proposed a better way to general procedural buildings than the L-system used by Parish and Müller’s CityEngine. They authors argued that L-system is not a naturally fit for this application since it is meant to be used for growth in open spaces, and buildings are bounded by the bounding boxes. They proposed a new grammar they called split grammar that operates on the geometry itself rather than strings like the L-system. The split grammar operates similar to how CityEngine divides a façade into grids for texturing, but rather than splitting it for texture, it splits the 3D geometric shapes to generate details. The grammar can split a façade into smaller regions, and each region is replaced by mesh of the same dimension but more detail such as windows. The split grammar system does offer more flexibility and diversity over Parish and Müller’s L-system, but to generate interesting and complex buildings requires a large number of grammars to be written.

A simpler form of split grammar named wall grammar is proposed in “Wall Grammar For Building Generation” (Larive and Gaildrat 2006). Rather than working with 3D geometries, wall grammar manipulates planes instead. Each side of a building is a different unit that can be manipulated separately. Simple rules such as dividing a wall and extruding sub regions makes generating details such as windows very easily, and the article also briefly explained roof generation. Though not as diverse as split grammar, the wall grammar achieved its purpose of making building generation simpler.

# Chapter 3: Methodology

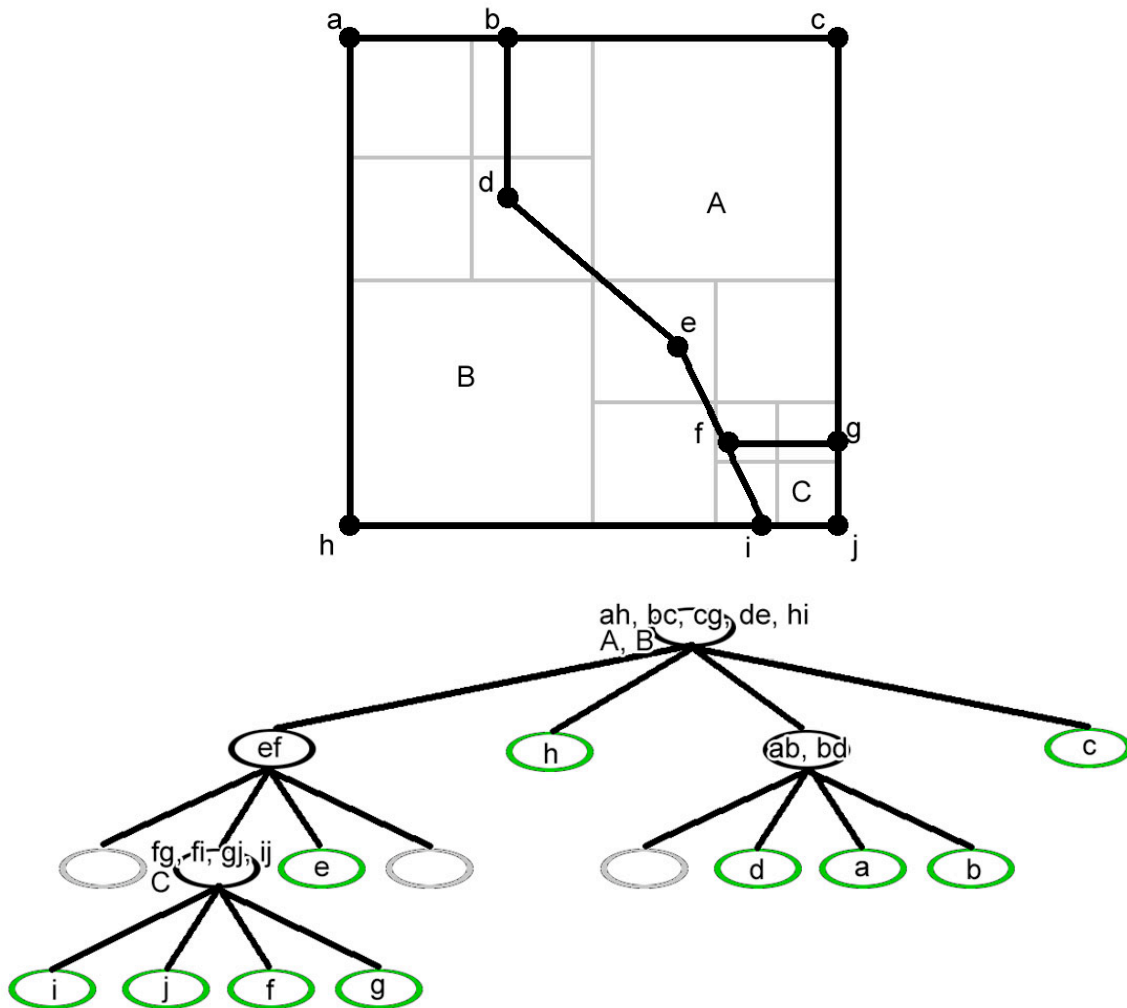
As stated previously, the purpose of this project is to produce a tool that combines procedural content generation method with traditional level design tool to create a system for creating cities. Therefore, the project can be divided into the procedural component and the editing tools. The procedural component will take as input some initial parameters from the user and create the city layout including the streets, blocks, and buildings. Once the city is generated, the user can then use the set of tools available to modify the procedurally generated city. A dynamic quadtree is used to hold all the data for the city. It must interface with both the procedural and the editing tool, therefore it is described first.

## 3.1 Quadtree

The resulting city is represented by a series of control points, road segments, and blocks. A control points is a point that specifies a location in the 2D space. A road segment connects two of these control points together, and a series of road segments link together to form a complete road. Finally, Polygon regions bounded by roads are blocks. Each control point has a location and a list of road segments that use the control point. Each road segment knows the two control points that make up the segment as well as the blocks on either side of itself.

The quadtree stores all control points on the leaves. All road segments and blocks are stored on the lowest common parent shared by all control points making up the road segment or the block. During the road generation stage, when the control points are inserted into the quadtree, the quadtree will subdivide until each control point in the quadtree is in a leaf. When a road segment is inserted into the quadtree, starting at the root of the quadtree, the algorithm checks whether the two control points making up the segment are in the same quadrant. If they are not, then the current node is the lowest common parent of the control points, and the road segment is inserted into the current node's list of roads. If the control points do belong to the same quadrant, then the same procedure is performed on the child belonging to that quadrant. Each intermediate node also contains a list of blocks, and the same algorithm is used for inserting blocks into the quadtree when they are created.

When a control point is moved during the editing stage, the leaf containing the point is rearranged in the quadtree to the correct location. The quadtree will further subdivide when necessary, and a collapse is performed on the former parent node if the rearrangement causes it to have only one child. When a control point is deleted during the editing stage, the leaf containing it is removed from the tree, and a collapse is performed when necessary.



**Figure 3.1** An example city and the corresponding quadtree. All the points are stored in the green nodes which are the leaves. Road segments and blocks are stored in the intermediate nodes.

## 3.2 Procedural Tool

### 3.2.1 Road Generation

The road generation system uses a modified version of the L-system described in “Procedural Modeling of Cities” (Parish and Müller 2001). The system recursively loops through a list of modules that each contains road information and modifies them using global goals and local constraints. Each module contains a point that represents the starting point of a road segment, a vector that represents the direction the road is heading, the length of the segment, and the width. Given a module, the global goal looks at the direction the road is currently growing and generates an ideal direction based on that and road patterns. The global goal also determines whether the road branches or not. When a road branches, one or two additional modules are placed into the list with the same starting point but different directions. The local constraint checks the result from global goals to see if it is valid and whether it intersects with any existing roads. If the module passes the local constraint check, a road segment is created between the starting point from the module to the end point represented by the starting point plus the unit length direction vector multiplied by the length. Once that road segment is placed into the quadtree, the module is removed from the list, and a new one is created with the previous end point being the new starting point.

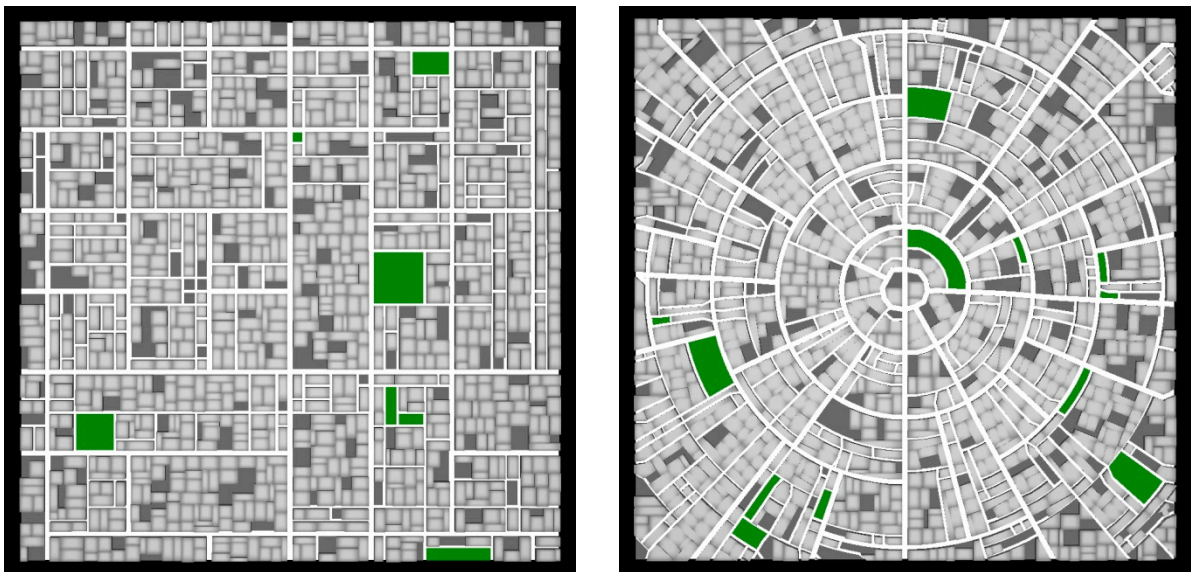
The implementation of the road generation system follows the L-system described in “Procedural Modeling of Cities” (Parish and Müller 2001) fairly closely with some modifications. Parish and Müller described using branching, road, and query modules in their system. Since road and query module will always come in pairs, there is really no reason not to combine them into one single module. Also, since branching modules turn into pairs of road and query modules by copying their data over, the system can be simplified to just one type of module with an extra field specifying whether it’s in the query, branching, or road generating stage.

Another change made to the grammar is the combination of global goals, local constraint, and actual placement of the road into the same iteration. The original grammar described in “Procedural Modeling of Cities” (Parish and Müller 2001) called for the three processes to be done in separate iterations. However, it is possible to have two different road segments that independently don’t intersect with anything but intersect with each other to

both pass the local constraint check in the same iteration. When both are placed into the map during the next iteration, the intersection is not detected. To solve this problem, the algorithm is simplified to perform global goal, local constraint, and actual placement of road in the same iteration. This way, road direction are determined, checked for validity and adjusted, and added right away. When the second road is placed, it can perform intersection check with the first one.

### 3.2.2 Global Goal

The global goal takes a starting point and the direction of the road segment and determines an ideal direction based on road patterns. A road pattern can either be raster, which forms rectangular grids, or radial, which creates circular roads. Each pattern is accompanied by a distribution map, a grayscale texture used to determine where the pattern will be applied. The weight of each pattern is determined by the color of the map at the starting point of the road segment. A white pixel on the map means the associated pattern will be applied at full intensity. Similarly, a black pixel means the pattern will be ignored. Each time a module goes through the global goal algorithm, the intensity of each pattern is determined and added together. Using the sum, each pattern's final contributing percentage is calculated and used. The ideal direction is the weighted average of the direction produced by each pattern weighted by the percentage contribution of the pattern.

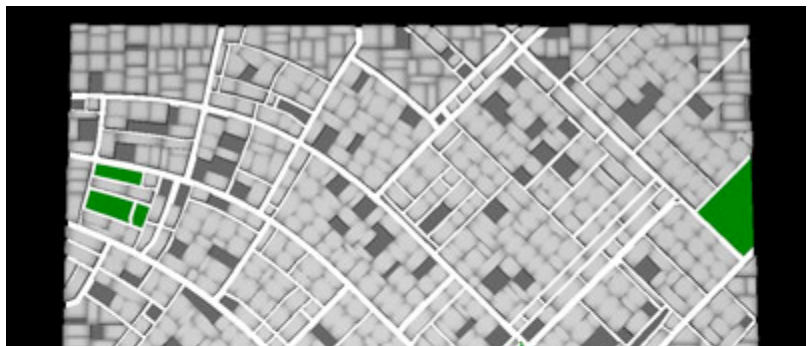
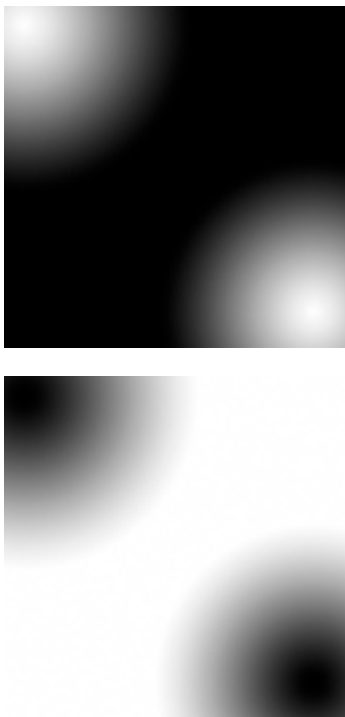


**Figure 3.2 Raster pattern and Radial Pattern**

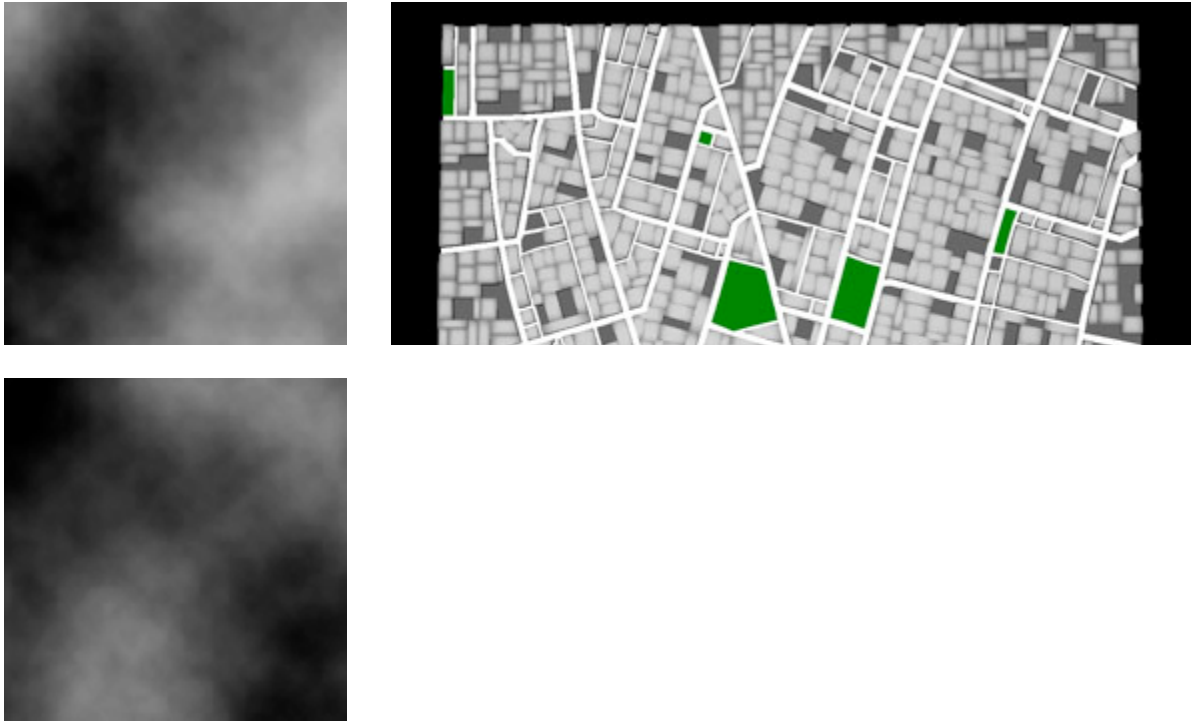
a)



b)



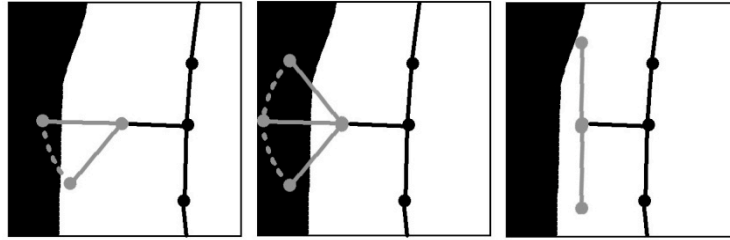
c)



**Figure 3.3 Each city uses the same raster patterns but with different distribution map. City a) uses maps that contains sharp drop between white and black, so the roads on the boundary make sharp turns as they switch pattern. City b) uses maps that contain a gradient between white and black, so the roads curve gradually. City c) uses noise for both pattern and produces a completely different look.**

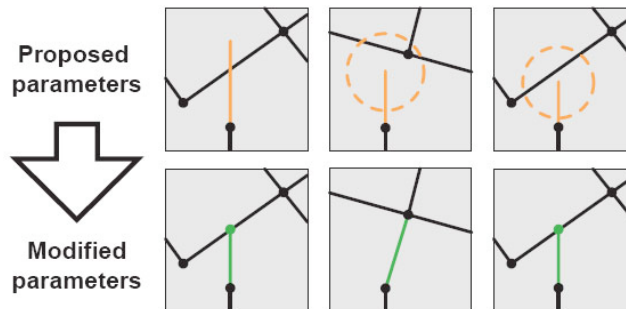
### 3.2.3 Local Constraint

Once an ideal direction is determined, the local constraint algorithm checks whether the end point is in an illegal boundary. The system uses a boundary map supplied by the user to determine if any point is in the illegal regions. If the end point does end in an illegal region, the system rotates the direction of the module within a small threshold to see if it can turn the road so the end point lands in the legal region. If turning the road within the threshold cannot turn the road out of the illegal area, the road module is removed, and two new modules are created with new directions at right angles on either side of the original direction creating a T-junction.



**Figure 3.4** When a road goes into illegal region, the system tries to curve it into the legal area as in the first picture. If it cannot be done, such as the case in the second picture, the road splits into a T-junction.

If the end point is inside legal region, either since the start or after being adjusted, intersection with existing road is checked. If the road segment represented by the module does intersect with an existing road, the intersecting point is determined. The existing intersecting road segment is split in two at the intersecting point. The road segment represented by the module is modified to also end at the intersecting point, and the module is removed after the road segment is inserted into the quadtree. If the road does not intersect with an existing road, the system checks if the end point is near any control point already in the quadtree. If one is found, the road segment represented by the module is modified to end at the control point, the road segment is placed in the quadtree, and the module is removed. Finally, if a nearby control point is not found, the road is extended a little. The intersection test is performed again and handled the same way. If nothing is found, the road segment is inserted into the quadtree as is, and a new module is created with the end point being the new starting point.



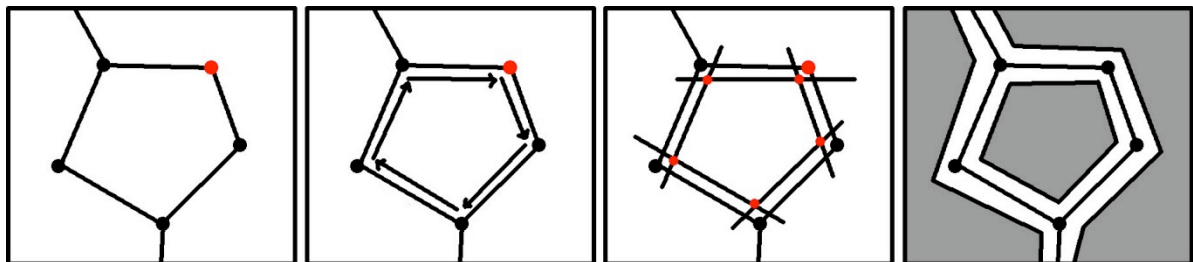
**Figure 3.5** Checks performed for creating intersection. (Parish and Müller 2001)

### 3.2.4 Block and Building Generation

Once the road generation stage is complete, the block generation process begins. The system collects all the leaves, which each contains a control point, from the quadtree and iterates through them to generate the blocks. At each leaf, the algorithm goes through all the road segments connected to the control point in clockwise order and creates the block that is on the right side of each road segment relative to the control point. The algorithm outlines the block by jumping from road segment to point to road segment until it comes full circle and the original control point is reached. It comes back at the left side of the next road segment in clockwise order, and the algorithm will repeat for the right side of the next road segment until all blocks for all road segments connecting to that control point is created. If a given block is already created by a previous control point, the process is simply skipped.

For each road segment that borders the block, it is translated towards the inside of the block by half of its width. Half widths are used so when the blocks on both sides of a road segment are created, the distance between them is the actual width. Intersecting points between neighboring borderlines are calculated, and these lines and points form the borders of the block. Once all the points of a block is determined, a triangulation algorithm is used to convert the block into triangles that are used for both creating the meshes for the block so it can be drawn and point in triangle test so a user can select a block with mouse click.

The triangulation algorithm loops through the points in clockwise order and checks if a point and its two neighboring points form a convex or concave angle. If the points form a concave angle, it's skipped since the triangle formed by these three points is not part of the



**Figure 3.6** The second picture shows how the border of a block is found by tracing the points and edges. The third pictures how the borders are translated towards the inside of the block, and intersection between neighboring edges are found.

actual polygon. If the angle is convex, then a triangle is created, and the point is removed from the list. This process continues until three points are left forming the final triangle.

To create buildings, the longest edge of the block is determined, and a random point along that line is determined. A line perpendicular to the longest edge at the random point is used to split the block in two halves. This process continues until each sub-region is smaller than a specified area. Finally, these regions are extruded from the plane to form a three dimensional box that represents a building.

## 3.3 Editing Tool

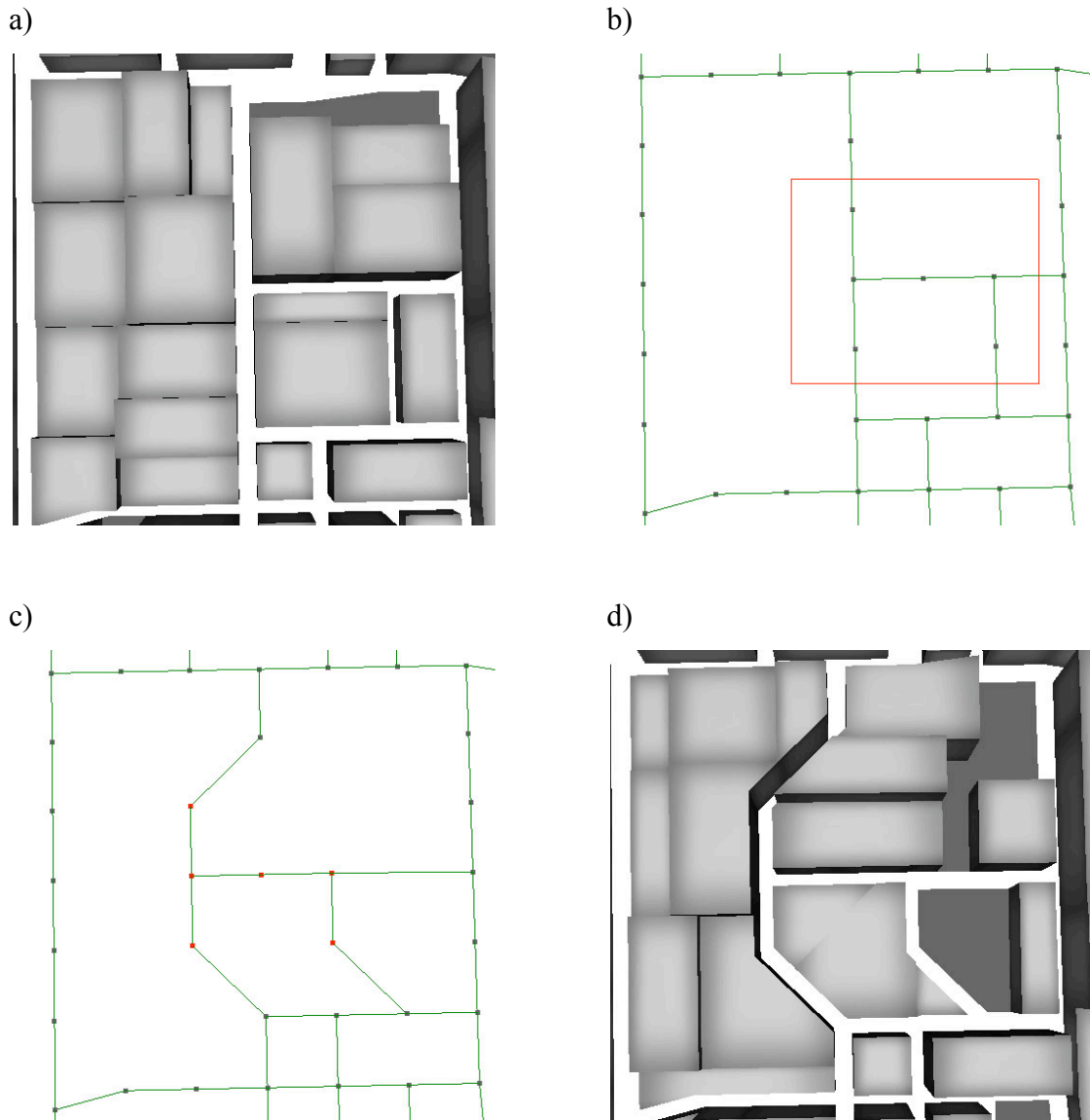
The system supplies user with a set of tools that allow the user to modify the procedurally generated city. The user can select control points and move them, and road segments and blocks connecting to the selected points will be dynamically adjusted. The user can add road by dragging a line and remove road by selecting a road segment and press the delete button, and control points and blocks will be added or removed accordingly. The user can select a block and change its property such as building size and height. Finally, the user can save and load the cities he created.

To make selections, the user's mouse click on the screen is converted to a point on the plane in world space through ray casting. The point in screen space is first converted to the corresponding point on the camera's near-plane in camera space. Then, the camera to world matrix is used to convert that point to world space, and the ray starting at the camera's position in world space to the point is tested for intersection with the XZ-plane. The intersection, if there is one, is the position of mouse click on the map, and it's used differently for each editing feature.

### 3.3.1 Moving Control Points

The control point editing feature allows user to drag a region on the screen which selects all the control points in the rectangular region. To do so, all four corners of the rectangular region are converted from screen to world space. Once the four corners are found, the region is divided into two triangles, all controls points within those triangles are found by performing point in triangle tests. Once all control points within the region are found, the user can then drag all the points. Each time the mouse is dragged, the validity of the move is checked. For a control point, a move is invalid if it brings the control point too close to another one or if it brings the point into an illegal region, and a move is only valid if it's valid for all control points selected. When a control point is moved, the corresponding leaf checks with its parent to see if it's still in the correct location in the quadtree. If it is, the position of the control point is simply changed. If it's not, the correct location in the quadtree is determined, and the leaf will be relocated. Since all road segments are stored in the lowest common parent, all road segments connected to the control points are relocated as

well. The same process is repeated for each control point in the selection. Once the selection is released or a new selection is made. The block and building generation algorithm is performed on all control points in the selection to create new blocks that match the new road layout.



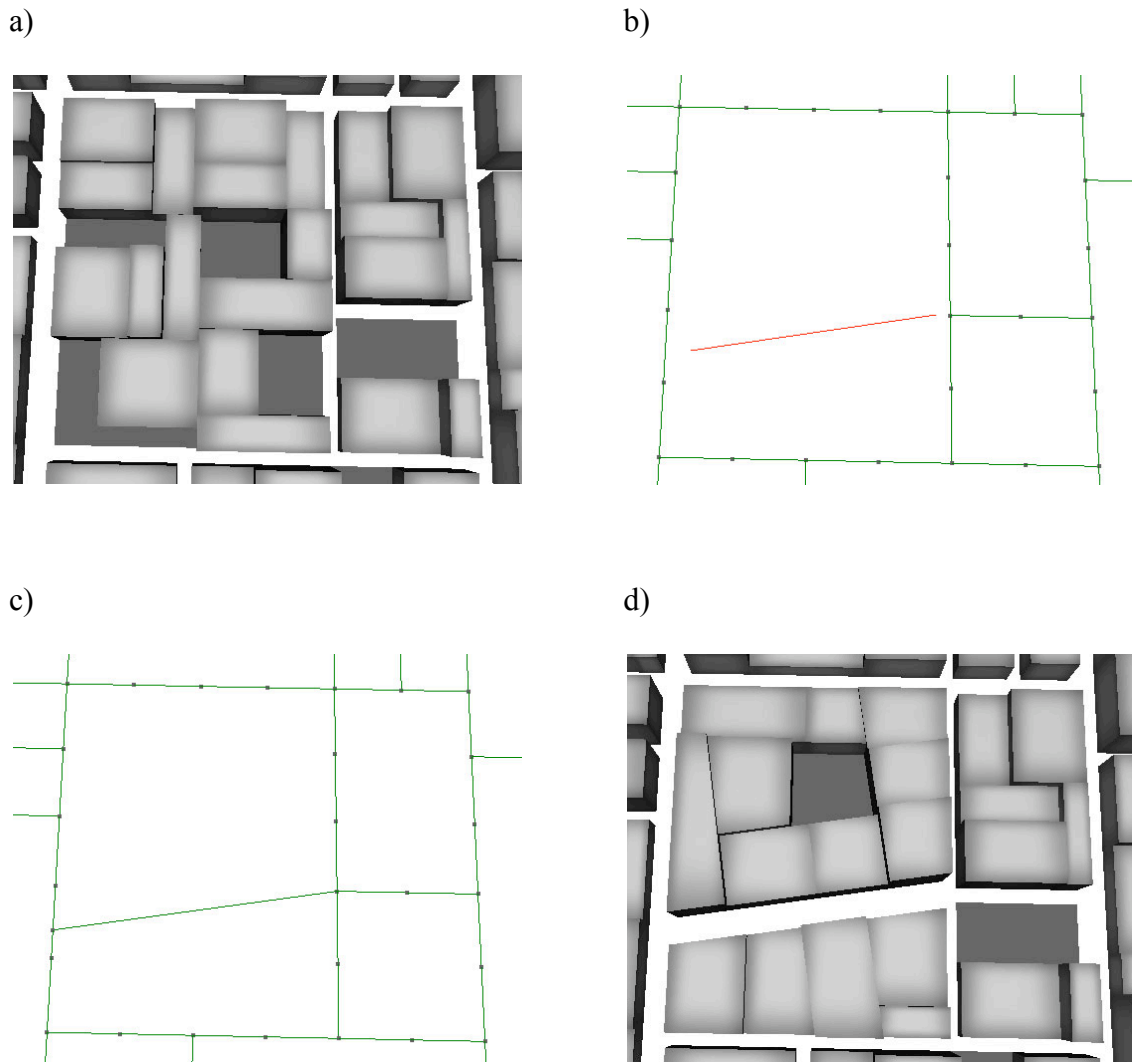
**Figure 3.7 a) before the modification b) dragging the mouse to select control points c) control points moved d) blocks are recreated**

### **3.3.2 Add Road**

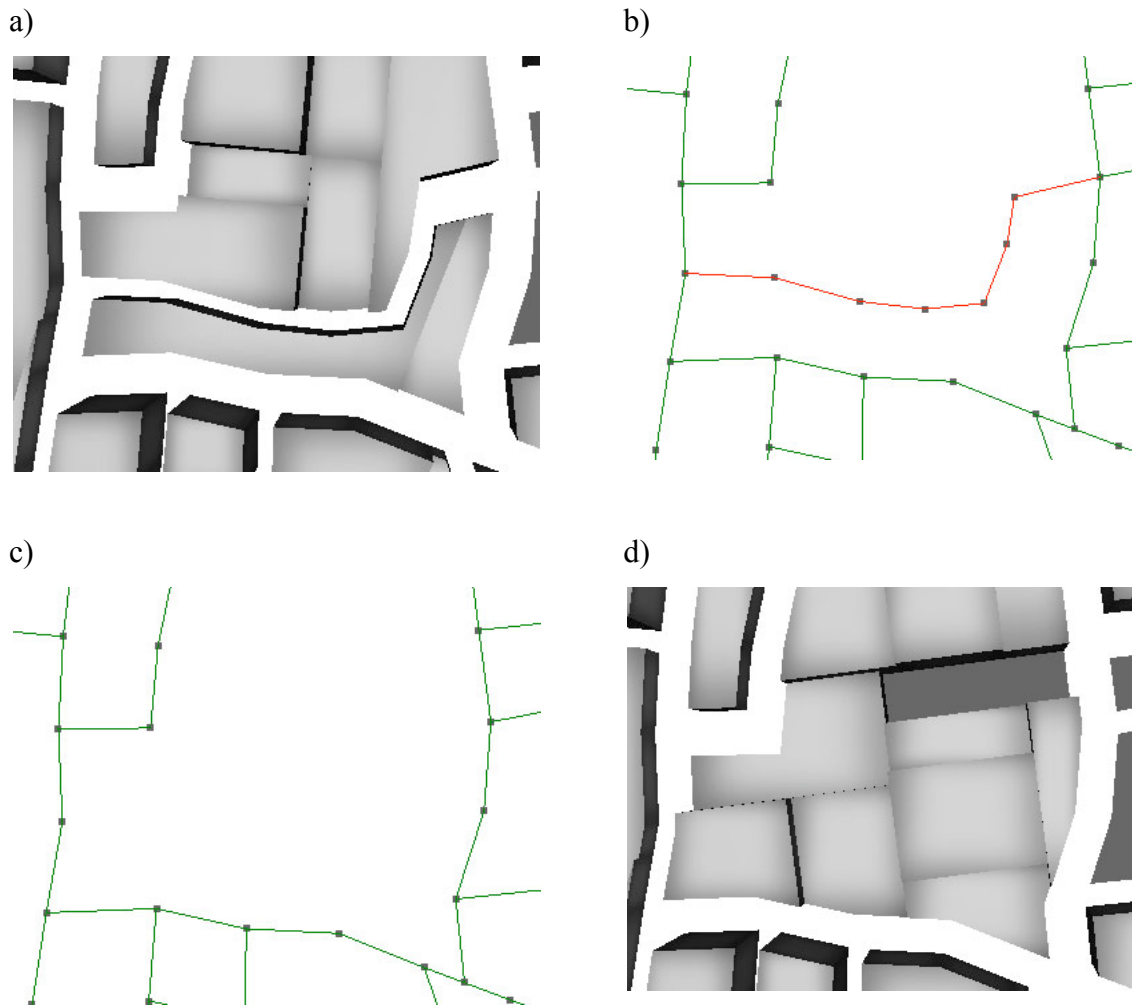
The add road tool allows user to add a road to the procedurally generated city by clicking and dragging a line. The position of the mouse when the left button is pressed down is the starting point of the line, and the end point is the position of the mouse when the left button is let go. Once the line segment in world space is determined, the tool performs several checks to see if the line is valid. If one or both of the point is outside the map, then the road cannot be added. If the two points are the same or really close, the system interprets that as a single mouse click and no road is added. If the line formed by the two points intersects with an existing road segment, nothing is done. Once the points pass the checks, the line segment is extended in both directions until intersections are reached on both sides. The two road segments on either side are split at the corresponding intersecting points, and the two intersecting points are connected by a new road segment that represents the line the user has dragged. The block that occupies that region will be deleted, and two new ones, one on either side of the new road segment, are created in its place.

### **3.3.3 Remove Road**

The remove road feature allows user to click to select a road and push the delete button to remove it permanently. A point on the plane is determined from the mouse click in screen space. The system traces the control points on either side of the selected road segment until intersections are reached on both sides. The system checks if the selected road belongs to the border of the map. The border is not allowed to be modified, so trying to remove a road belonging to the border is not allowed. If the selected road is not a boarder, when the delete button is pressed, all road segments that form the road are removed from the quadtree along with all the intermediate control points. The quadtree is rebalanced. The two blocks on either side of the delete road are removed, and a single block created in their place.



**Figure 3.8 a) before adding the road b) dragging a line to add new road c) road extend to intersect on both sides d) old block removed and two new blocks created**



**Figure 3.9 a) before removing the road b) clicking on a segment selects the entire road c) road removed d) old blocks combine to form new block**

### 3.3.4 Edit Block

The edit block tool selects a specific block by performing point in triangle test using the triangles calculated during the block generation's triangulation process. Once the selection is made, a dialogue box pops up prompting the user for building size and height. Each time the user presses the apply button, the buildings on the block are deleted, and new ones are created using the new parameter. The building size determines how far the subdivision of the block goes during the building generating process described previously. The minimum and maximum determines how far the buildings will be extruded from the

plane. Optionally, the user can also specify a block to be a park. In which case, the block will be colored green, and no buildings will be created on it.

### 3.3.5 Save and Load

Rather than saving all the mesh data generated for the city, the save feature saves only the initial parameters used to generate the current city and all subsequent changes. Saved initial parameter includes the boundary map, the seed for the random number generator, the number of patterns used, and properties of each pattern. For each road pattern, the name, map, and parameters are saved. For all the changes made to the city, the system keeps track of them using an internal stack. When saved, the stack of changes is also written to the file.

When loading, the system reads in all the initial data and uses them to generate a copy of the city. Since all parameters are the same and the same random number generator seed is used, the generated city will be exactly the same as the saved one before any changes. Once the generation is complete, the list of changes is loaded onto the stack and performed one at a time. After all changes are loaded, the final city is exactly the same as the one saved by the user in the previous session.

Table 3.1 Save file format	
Description	Size (Bytes)
Magic Number	4
Random number generator seed	4
Boundary map name length	4
Boundary map name	Boundary map name length
Pattern chunk ID	4
Pattern count	4
Pattern1 name length	4
Pattern1 name	Pattern name length
Pattern 1 type	4
Map name length	4
Map name	Map name length
Pattern1 parameter 1	4
Pattern1 parameter 2	4
Repeat for each pattern	
Changes chunk ID	4
Change ID	4
Data	See Table 3.2

Table 3.2 Size of each type of change		
Type	Data	Size (Bytes)
Moving Control Points	Chunk ID	4
	2D coordinates for the four corners of the selection	32
	2D move vector	8
	<b>Total</b>	<b>44</b>
Add Road	Chunk ID	4
	2D coordinate of starting point	8
	2D coordinate of end point	8
	<b>Total</b>	<b>20</b>
Remove Road	Chunk ID	4
	2D coordinate of selection	8
	<b>Total</b>	<b>12</b>
Edit Block	Chunk ID	4
	Park?	1
	Building height max	4
	Building height min	4
	Building size	4
	<b>Total</b>	<b>17</b>

# Chapter 4: Results and Analysis

As stated previously, the purpose of this project is to incorporate procedural city generation with traditional level design tools to create a system that can generate random city efficiently yet still be customizable. Attempting to achieve multiple goals at the same time, the final product excels at certain objectives more than others.

## 4.1 File Size and Efficiency

As stated in the previous section, the tool saves the initial parameters and subsequent changes rather than the generated mesh data. Doing so drastically decreased the file size of the save files. Assuming all file names have twenty characters, a city that uses two patterns can be saved in just 168 bytes. Consider saving one of the city by mesh data. Table 4.1 shows the amount of data and the time the generation process took for 10 different generated cities. Assuming all buildings have the smallest possible size with a triangular base, a building will have five faces which require 20 vertices. Each vertex needs a position, normal, and a set of UV coordinates, so a size of 32 bytes. Even assuming all buildings are triangular, each one will have a size of 640 bytes, which is much greater than 168 bytes already. According to Table 4.1, the generated cities have around 1000 buildings each, so to save the mesh data for the entire city will require 640 kilobytes. It would require over 14000 moving points operation for the tool's file size to add up to that amount.

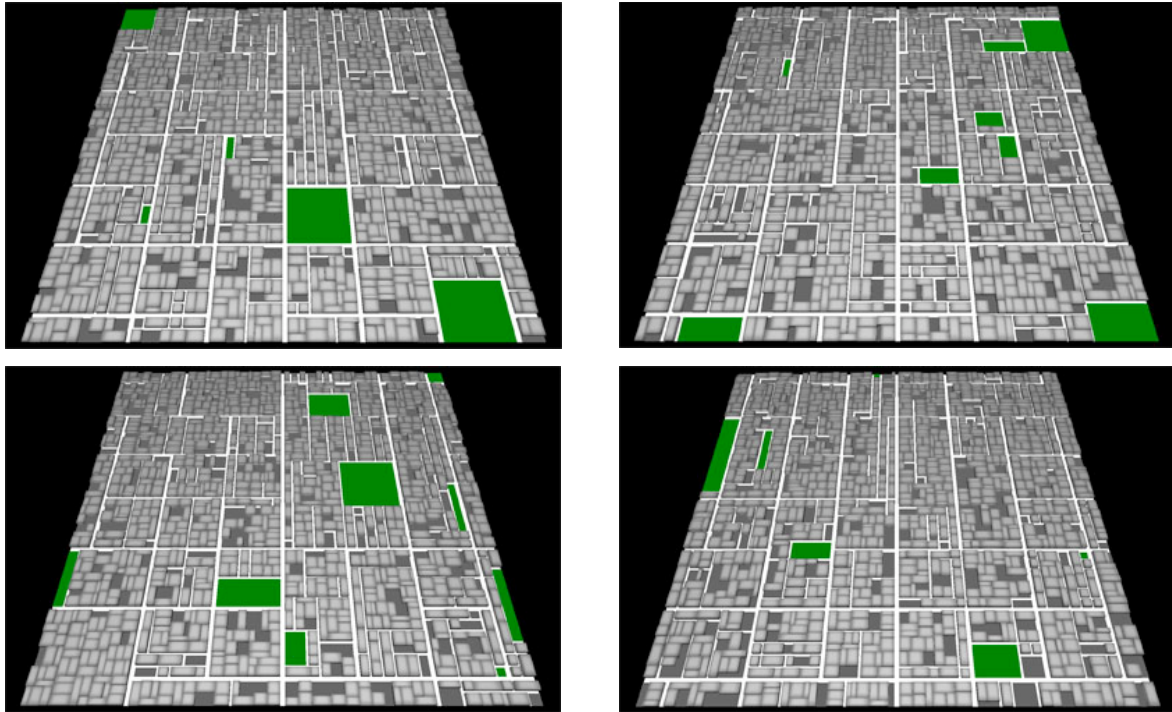
In addition to small file size, the tool is very efficient and can generate a city extremely quickly. It is clear that the bottleneck of the system resides in the block generation system where the triangulation and mesh generation takes place. But despite the fact that the block generation takes an order of magnitude longer than the road generation, the entire process takes no longer than a couple seconds, and in most cases, a city can be generate in under a second.

	Point Count	Road Count	Block Count	Building Count	Road Generation Time (ms)	Block Generation Time (ms)	Total Time (ms)
1	871	1062	192	1008	30	454	484
2	1393	1719	319	1061	48	671	719
3	914	1120	206	1102	16	439	485
4	876	1070	191	1075	32	454	486
5	928	1162	235	993	46	766	812
6	869	1075	200	1000	48	1015	1063
7	1324	1630	299	995	78	1391	1469
8	876	1065	190	1058	48	670	718
9	1016	1263	244	1057	94	984	1078
10	1357	1672	316	1065	94	1219	1313

The original intention was for the tool to be used during the production pipeline, and once the level is final, export the level as mesh data to be used in an engine. Considering the speed of the generation process and the size of the save files, it is feasible to replace the export to mesh data step and directly have the engine generate the level during load time. This will keep level files small without noticeable change to the load time, which can be extremely useful for console games where everything must fit on a disc with very limited space.

## 4.3 Randomness

Since any combination of patterns and distribution maps can be used, the system can generate endless number of unique cities. Since the system relies heavily on the random number generator, even with the same set of inputs, cities that are vastly different can be generated. Figure 4.1 shows four different cities generated with the same raster pattern. This allows user to generate any number of cities that are all different from each other yet share similar characteristics.



**Figure 4.1 Four different cities generated using the exact same input**

## 4.4 Customizability and Realism

The city is customizable in both the procedural generation part and the tool editing part. During the procedural portion of the process, the user gets to choose the road patterns to use and the maps that determine where each pattern has the most impact. However, the ease of customization of the road patterns conflict with realism during this process. Referring back to Figure 3.3, both a) and b) have very simple maps, and user can easily see the correlation between the maps and the resulting city and how to modify it to achieve different results. The produced city, on the other hand, doesn't look real. The city from c) uses 2 noise textures as the map and looks much more natural. However, it's extremely difficult to see how the maps and the patterns together translate to the resulting city. Therefore, even though it looks more natural, it's difficult to predict how changing the map would influence the produced city which makes it less customizable.

The editing tools have the opposite characteristic. The editing tools allow the user to directly manipulate the city. If realism is the goal, the user can think for himself what he thinks would look more real, and edit the city accordingly. Although lacking the ability to

modify individual building placement and property is a major weakness of the system, using the tools that are available can still achieve satisfactory results.

# Chapter 5: Conclusion

There is no argument that a designer can create a level that is more interesting than a level created by an algorithm. There is also no argument that algorithms can perform certain tasks much more efficiently than people can. As game scope expands, the asset creation becomes ever more time consuming. Procedural content generation allows artists and level designers spend more time improving and polishing important parts of the game and lets algorithm generate areas of lesser importance.

The objective for this project is to incorporate procedural city generation with traditional level design tools to create a system that can generate random cities efficiently while allowing them to remain customizable. The system can generate a realistic city from scratch using simple parameters, but it also features tools that can modify it every step of the way. If the user is not satisfied with the street layout or building configuration generated by the system, he will be able to add or delete them as he see fit.

As stated in the previous section, the project has succeeded in most areas. The tool is not only fast but also produces file size that are drastically smaller by saving only the data required to reproduce a replica. The procedural portion of the project struggles to achieve both customizability and realism, but the provided editing tool is able to offer both to a certain degree. Although the project in its current state cannot produce assets at the level of polish required for real games, it is still an efficient tool that can be used for quick prototyping and inspiration.

## 5.1 Future Work

As mentioned in section 4.4, the tool currently has no way for the user to manipulate individual buildings directly, so implementing a way for the user to interact with buildings would be a priority. Users should be able to click on a building the same way they do for control points, roads, and blocks, and be able to raise and lower it. Furthermore, users should have the tool that allow them to determine not just the height of a building, but where it is placed inside the block.

A feature what was considered during research but quickly dropped was procedural terrain. Perlin noise can be used to generate a 2D map that can be used as height values. Once the terrain is generated, brush tool can be used to raise and lower the terrain. To complement the terrain, in “Procedural Modeling of Cities” Parish and Müller describes two more road patterns including one based on elevation. Roads using the elevation pattern will try to extend towards area with the same height by sampling the height map. The population pattern that is also described in “Procedural Modeling of Cities” works in a similar fashion, but rather than sampling the height map for the closest value, it samples a separate population for the highest population density.

One element the project is currently lacking is visual fidelity. Currently all buildings have the same gradient texture applied to them. It’s better than using plain color, but not by much. The project would definitely benefit from having actual building textures such as bricks or wood applied to the buildings. Furthermore, building details can be procedurally generated. Rather than having boxes for buildings, details such as windows, doors, and balconies can all be procedurally generated, and tools can be created for editing those as well. This project just barely scratched the surface of what procedural content generation is capable of.

Finally, for this tool to be useful in actual game development, there needs to be a way for the data to be transferred to the game engine, and there are many different approaches, each with its own advantages and disadvantages. One way to incorporate this tool into the pipeline is to add an export feature that exports the mesh data for the generated city into the format an existing engine can use. The advantage of this approach is that no modification to the engine is needed. Once the mesh data is exported, the engine can handle it the same way it handles data exported from other tools. The disadvantage of this approach is that it loses one of the advantages of procedural content generation. As described in the results and analysis section, procedural content generation can produce assets with just a small amount of input parameters. With this approach, the user still benefits from the efficiency and customizability of the tool, but the small file size is lost. Another thing to consider when exporting the city as mesh data is how the city is divided. The entire map can be exported as one giant mesh, each block can be exported as a mesh, or each individual building can be exported as its own mesh. If the entire city can be seen at once in the game,

then exporting the city as one giant mesh makes rendering more efficient since it only requires one draw call. For *Grand Theft Auto IV* (Rockstar North, 2008) type of game where the player sees the city from the pedestrian point of view, exporting blocks or buildings individually can produce better result since it allows more aggressive LOD and different part of the city can be drawn with different shaders to produce different effect.

The second method to incorporate this tool into an existing engine would be adding procedural generation capability to the engine. The advantage being the tool can still export only the initial parameters and take advantage of the small file size. The disadvantage is the engine needs to be modified so it can generate procedural content during load time or if resources allow, during runtime as needed. Similar to method one, this approach can be modified treat each individual building as its own entity. For example, a building can be exported as the points on the 2d plane and the building height, and roads can be exported as the starting point, end point, and width. Once loaded, the engine will then take those data and procedurally generate the actual mesh data.

# References

- Blizzard Entertainment. 1997. “Diablo”; PC Game, Blizzard Entertainment.
- Blizzard Entertainment. 2000. “Diablo II”; PC Game, Blizzard Entertainment.
- Google. Google Maps. <http://maps.google.com/>
- Greuter, Stefan, Jeremy Parker, Nigel Stewart, and Geoff Leach. 2003. Real-time Procedural Generation of ‘Pseudo Infinite’ Cities. Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia. The ACM Digital Library. <http://portal.acm.org/>, <http://portal.acm.org/citation.cfm?doid=604471.604490>
- Interactive Data Visualization; 2002. “SpeedTree”; PC Software.
- Larive, Mathieu, and Veronique Gaildrat. 2006. Wall Grammar For Building Generation. Proceedings of the 4th international conference on Computer graphics and interactive techniques in Australasia and Southeast Asia. The ACM Digital Library. <http://portal.acm.org/>, <http://portal.acm.org/citation.cfm?doid=1174429.1174501>
- Maxis Software. 1989. “SimCity”; PC Game, Electronic Arts.
- Maxis Software. 2008. “Spore”; PC Game, Electronic Arts.
- Parish, Yoav I. H., and Pascal Müller. 2001. Procedural Modeling of Cities. Proceedings of the 28th annual conference on Computer graphics and interactive techniques. The ACM Digital Library. <http://portal.acm.org/>, <http://portal.acm.org/citation.cfm?doid=383259.383292>
- Rockstar North. 2008. “Grand Theft Auto IV”; PlayStation3 and Xbox360 Game. Rockstar Games.
- Sun, Jing, Xiaobo Yu, George Baciuc, and Mark Green. 2002. Template-Based Generation of Road Networks for Virtual City Modeling. Proceedings of the ACM symposium on Virtual reality software and technology. The ACM Digital Library. <http://portal.acm.org/>, <http://portal.acm.org/citation.cfm?doid=585740.585747>
- .theprodukkt. 2004. “.kkrieger chapter 1 beta”; PC Game, independent.
- .theprodukkt. 2007. “.debris”; PC Game, independent.
- Thomas, Gwenola, and Stéphane Donikian. 2000. Modelling virtual cities dedicated to behavioural animation. In M. Gross and F. Hopgood, editors, Eurographics'00, volume 19, 71-79, Interlaken – Switzerland.

Ubisoft Montreal. 2007. "Assassin's Creed"; PlayStation3, Xbox360, and PC Game. UbiSoft.

Wikipedia. Development of Spore – Wikipedia, the free encyclopedia.  
[http://en.wikipedia.org/w/index.php?title=Development\\_of\\_Spore&oldid=216789767](http://en.wikipedia.org/w/index.php?title=Development_of_Spore&oldid=216789767)

Wikipedia, L-system – Wikipedia, the free encyclopedia.  
<http://en.wikipedia.org/w/index.php?title=L-system&oldid=213423810>

Wonka, Peter, Michael Wimmer, François Sillion, and William Ribarsky. 2003. Instant Architecture. ACM SIGGRAPH 2003 Papers. The ACM Digital Library.  
<http://portal.acm.org/>, <http://portal.acm.org/citation.cfm?doid=1201775.8823>