4-25-2005

Algorithmic generation of cities using cellular automata in a dynamically generated world

Eric Garza
Trinity University

Algorithmic Generation of Cities using Cellular Automata in a Dynamically Generated World

Eric Garza

A departmental honors thesis submitted to the Department of Computer Science at Trinity University in partial fulfillment of the requirements for graduation with departmental honors.

April 21, 2005

_____ _____

Thesis Advisor Department Chair

_____

Associate Vice President for
Academic Affairs

Algorithmic Generation of Cities using Cellular Automata in a Dynamically Generated World

Eric Garza

Abstract

The procedures and algorithms used to create a realistic city in a virtual world are outlined in this paper. The world is algorithmically created from a single random number. The terrain, water, and population of various races [1]are created in real time as a viewer walks around the world. The world is created in various levels of detail; objects that are close to the viewer are fully generated in high detail, and objects further away from the viewer are not created. The locations of the cities are first based on the pre-existing world then on the set of tolerances of the races. To create an accurate representation of cities, they are built up from starting blocks. Starting blocks are small portions of the population placed within the bounds of the city and are used seed the Cellular Automata. The Cellular Automata then computes the new population and zone for each city block based on the surrounding blocks using three different levels of rules. With this approach cities build up to reflect the development of real cities, by small collections of people settling around attractive areas of land and grouping together with other settlements.

[1] The term race is used with the thought that other mythical races such as elf or dwarf would be incorporated into the world not human races.

Algorithmic Generation of Cities using Cellular Automata in a Dynamically Generated World

Eric Garza

Acknowledgements

I would like to thank Dr. Mark Lewis for his guidance and assistance in explaining the complicated techniques used in this paper. I would also like to thank my family for supporting me through the development and writing of this paper. I would like to thank Peter Newell for compelling me to carry out my thesis and providing helpful insight when needed. I would like to thank Shelly Nickels for giving me the strength and courage to continue writing this paper when I didn't know I could.

Table of Contents

List of Tables

List of Figures

1 Introduction

Building a city is not an easy task. There are many things to consider when one sets out to build a city. There is an inherit transportation infrastructure that builds up and evolves as the city is created. This infrastructure shapes the way the city moves and the way people within that city interact. There are also the different types of structures and relationships between those structures. Cities have districts for factories, shopping, and housing that are placed with great planning. City planning is a field that tries to map out this complex map of streets and buildings in order to provide a guide to follow as the city grows.

City generation has been approached in many different ways. One approach is to focus on the concept of a city shaped by how people traverse it. This approach was discussed by Ingram and Benford in Building Virtual Cities. In Building Virtual Cities two virtual city building algorithms are compared. One describes the work done by Kevin Lynch. Lynch created a system called LEADS which is a general algorithm to enhance legibility. This algorithm was used to improve the transportation throughout the city. The other is a system called the Virtual City Builder built by Hillier and Hansen which creates multi-user virtual cities. With these two systems a map of a city and traversals throughout the cities can be created. These sorts of approaches need to also model human interaction to be truly reflective of transportation.

In A Paradigm for Controlling Virtual Humans in Urban Environment Simulations a series of controllers are created to model human behavior and simulate interaction. Instead of creating a city from scratch a city is stored in a database that the human simulations can interact with.

The paper by Greter and Parker focuses more on the buildings within a city: Real-time Procedural Generation of `Pseudo Infinite' Cities. The buildings are generated based on their location within the city and are created by a random floor plan generator. Still the city has no meaning, only a vast array of buildings that are all different.

Parish and Muller applied L-systems to create an intricate road system to generate a city plot in which to build buildings on, but stopped there. The buildings still had no meaning and represented nothing more than a rectangle made of bricks. Most approaches focus on a single set of data that must be provided and then spend hours creating models of cities.

The techniques outlined in this paper describe a city generated on the fly from a single random number. The technique of generating cities in a dynamic world is a rare niche in the research field and no research was found that coincided with the goals set forth in this paper. Much of this research was based on the ideas of other's techniques but this research breaks new ground.

The applications of this paper are numerous and include movies, video games, and graphics. Animated movies such as Toy Story spend much of their time creating houses and cities to place their animated characters. With the ability to create cities on the fly production time decreases significantly and focus can return to the story and character development. Video games require months of detail planning and structure of cities and worlds. To create new worlds involves the same process and consumes much of the development time. With a world that can be generated on the fly, there is no development needed for world creation. In addition, creating new worlds is as simple as changing a single number the world seed. Graphic renditions of Spider Man swinging

from the tops of New York's sky scrapers can be reproduced simply with the aid of a self generating world.

This paper builds off of many other researchers' work from other Trinity University students. Ted Wilmes' work on the terrain modeling and display is outlined in his paper: A real-time archaeological data visualization tool. Plants were modeled using L-systems which continued research was done by Scott Schaefer in A technique for wind visualization in plants generated with modified L-systems. The ideas from Markus Haas in Dynamic virtual worlds: a notion of realism in future virtual reality applications were applied when designing the virtual world. The virtual world designed was taken directly from Mark Lewis in his paper Large scale virtual worlds: algorithmic construction and dynamic methods.

2 Design and Constraints

2.1. Description of the World

The world in which the cities are built in is no ordinary world. To manage a world size simulation several constraints need to be applied. The entire world can not be stored within memory, so levels of detail must be introduced. A large memory constraint would force the system to slow down, so a small data foot print is required. To solve this, the world is generated on the fly as users walk through it. To generate the world pieces, a fractal algorithm is used to create the terrain. Even though this world is generated programmatically the same world is generated each time. The ability to reproduce the same world is vital to properly test and perfect the way each object is made. In order to accomplish this, a single random number is used to seed the world. All calculations, additional random numbers, and variability of objects are based off of this number. Objects placed in the world seed their random numbers with their position so that when a viewer comes back to that position they will be presented with that same object. The world needs to be a framework to add all sorts of objects. Therefore objects are placed in the world dynamically as the world generates. To achieve this, each object modifies the terrain piece that it attaches to and, at a specified level of detail, renders itself. To remove objects from the world and likewise add them, the object's modifier is removed or added to the starting terrain piece. Objects in the world need to be accessible quickly, so each object adds itself to an octree. Within this octree objects can check for collision with other objects. Continuing this description a solution is provided for each constraint listed above.

When a viewer is far away from a tree they can not see the leaves on a tree, so why generate the leaves? This is the driving force behind the concept of levels of detail. The concept involves only showing a certain amount of detail at a given level that a terrain piece has been divided into. In addition to only showing a specified amount of information the objects that are created to show that information do not need to be created either. In the instance of trees, the tree may create a leaf object for each of the leaves which may in turn handle the way the wind blows them and render its self to properly account for all climate variables. These leaf objects would not be created until the terrain sends a notification that the user was coming fairly close. The octree passes the information that the viewer is coming closer along to all objects within that area. A common technique used in the object's method that decides whether or not to render was determining the level at which the triangles have broken down. At a small level, the triangles are still rather large; but as the level increases objects could assume that at a fixed size of the world the viewer was close and was viewing objects at high detail. Objects such as plants and animals could use this method to easily decide when to render an object. Each object added to the world is responsible for managing the level of detail to render at based on the distance from any given viewer. It is alerted of the viewer's presence when its notification functions are called. A viewer is a user running the program that is traversing throughout the world or anything that requires a certain level of expansion. The viewer is an object in the world that merely sends notifications of its arrival and departure optionally a camera can shadow its position. As mentioned before, each object is stored in an octree. As the viewer moves from node to node within the tree, a notification is sent to the surrounding nodes that the viewer is coming closer to or

further away from them. This is done by calling the `sendArriveNotification` and `sendDepartNotification` methods in the `WorldObject` interface. Each object that is to be placed in the world is to implement these functions and create its components at the proper levels. It is the responsibility for each object to create itself at the appropriate time as well as destroy itself.

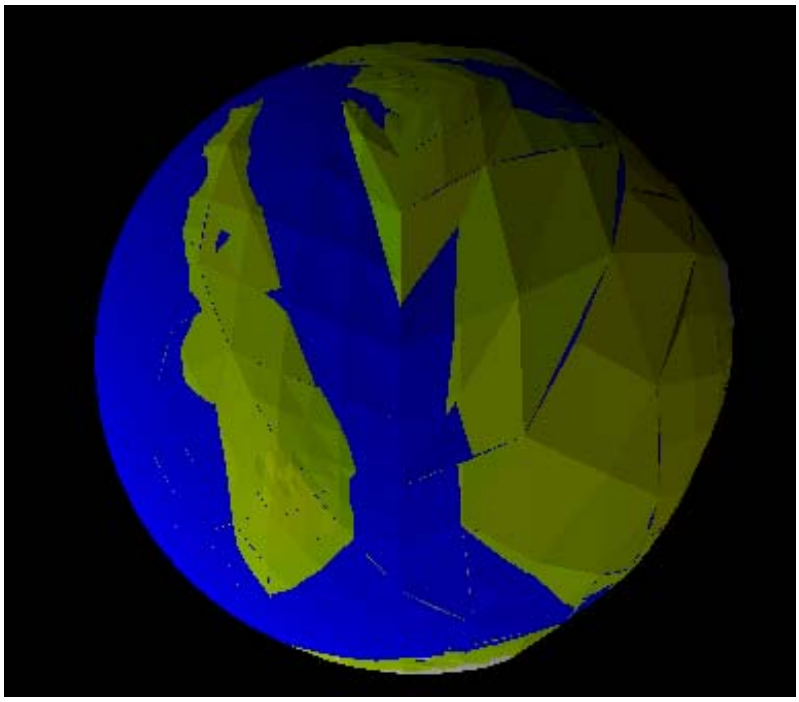

Figure 2-1 Globe showing large and small triangles

The figure shown illustrates how the triangles that make up the world split by the viewer causing the arrive notifications to be sent. This makes terrain pieces close to the viewer smaller and the terrain pieces further away are larger. With these procedures in place, memory can be capped to a fixed amount instead of growing as the viewer walks throughout the world. In addition, this approach allows for the world to scale easily. Objects can be added to the world without any change to the infrastructure of how the world renders. Each object renders itself, so there is no need for a large rendering object that must know how to render each object in the world.

A possible application of this virtual world is one in which a pseudo random world is needed for video games that run on most personal computers. To meet this requirement, little to no disk access is required to run this simulation. The simulation can run solely in memory which increases the overall speed of the program and opens it up to

environments such as personal computers or video game consoles. The architecture that makes this possible is each object is created and destroyed on the fly instead of storing them to disk. A possible drawback of this technique is that object creation must be efficient and quick. Objects must begin creating themselves early enough for a viewer to view the object in its final form when close enough. For example, cities must begin creating the city structure and layout before the viewer can see the city in order to place buildings and other objects. Those objects are not created till the viewer is close enough to see the objects; yet the city's structure is present. This also presents a problem with city generation. Cities are created from small towns that connect and group together with other small cities. They are usually not planned or structured yet a viewer must see a completed city and can not recount the evolution of a city in fast forward every time it wishes to visit. So there is a process of creating the city that must be accounted for as well. This means that before a viewer may be able to see the fine details of the city it still must start creating itself to allow for the time needed to create the city. Once again the city must regenerate itself each time a user visits a city and therefore the generation must be replicable.

2.1.1. Dynamic world creation

Seven days is too long to wait to create a world, so the world is created algorithmically on the fly. To manage this, four triangular planes are placed edge to edge to create a pyramid. A pyramid was chosen as the base structure to create the world because it is the easiest geometrical object to render. This base pyramid is the framework to build the remaining

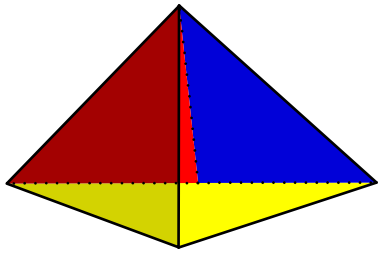

Figure 2-2 Triangles showing the initial setup of the world

pieces of the world. A world, however, is round and a triangle is not; to make the world appear round each triangle splits outwards approximately maintaining a radius. The triangle the viewer is placed on is where the transformation from pyramid to world begins. The viewer triggers the triangle to split from a single equilateral triangle into four. In doing so, each modifier can choose whether or not to place itself on the child terrain. Child terrain pieces are merely the four smaller triangle terrain pieces the parent triangle splits into. The splitting of triangles into smaller triangles is how the world creates its self. When a triangle splits into a small size, an object's modifier might decide to place an object in the world. A common approach taken with most modifiers is to always place a copy of its self on the child terrain piece. This approach works fine for plants and animals that might inhabit every part of the world, but this is inadequate for cities. If cities took this approach, a city would be on every part of the world. Another technique used in the world is to place a different modifier than the parent on the child terrain piece. This at first seems unnatural; if a plant is supposed to be on a terrain piece then it should remain a plant. However when considering a modifier that decides when to put a city, once a city is placed on a terrain piece the remainder terrain pieces need not consider placing a city on their terrain piece. In this case a dummy modifier that merely tells other modifiers that a city will be placed here is added to all child modifiers. The triangles split into smaller and smaller pieces so the viewer only sees tiny triangles that make up an object and not the larger triangles. In

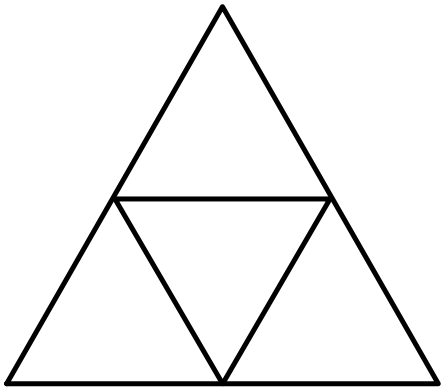

Figure 2-3 Triangle showing the way large triangles split into smaller triangles.

the same manner, objects break down to their smallest components when viewed closely by the viewer. The larger triangles are on the opposite side of the world; they do not need to be rendered at all because they are not seen.

How does an object get placed on the world if there is nothing to place it there? In order for any object to be placed in the world it must have a terrain modifier associated with it. The world is generated by the viewer splitting the terrain into smaller pieces, so in the beginning there are only terrain pieces and no other objects. Modifiers must be added to the terrain pieces for objects to appear. In addition to modifiers there is a notion that each terrain piece may have information stored within it. Each modifier must have information associated with it known as the modifier's information. This information does not seem appropriate when plants and animals are considered but make perfect sense with the climate and water modifiers. The climate modifier's information may store information about the average temperature and rainfall as well as many others for a particular terrain piece. With this information later modifiers can make decisions about the objects they are placing based on the previous modifiers. A natural ordering is then created in the adding of modifiers. The ordering follows the general pattern of large objects first and smaller objects last. This rule is broken by the climate modifier which is not a physical object at all but is the basis for many objects so it is first. The water modifier is next because it places large bodies of water such as oceans, lakes, and rivers. Cities are added

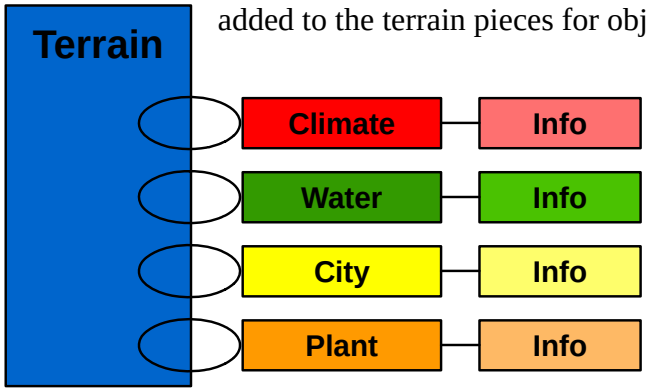

Figure 2-4 Terrain piece with a linked list of various Terrain Modifiers

to the terrain next being the next largest object in the world currently. The thinking here is not to place large objects on top of smaller objects or to have smaller objects placement determined by the larger objects.

To keep the world looking the same, the position of objects in the world is the seed for each random number it uses. The figure here shows nine different cities generated the exact same way except their location is different. As you can see when each city is generated all the random numbers used for the creation are seeded differently and therefore the world creates nine distinct cities. The representation used here is an aerial view that looks at the components of a city block. Each city shown has a population of one hundred thousand people in an area of roughly thirty-two units. This point is also illustrated when observing the way a flower might generate. A flower that randomly places pedals along its stem and generates a random color for its pedals generates these random values based on its position. So when the flower is placed in the same position, the flower will recreate itself exactly as before. This is an important point to note. Every object placed in the world

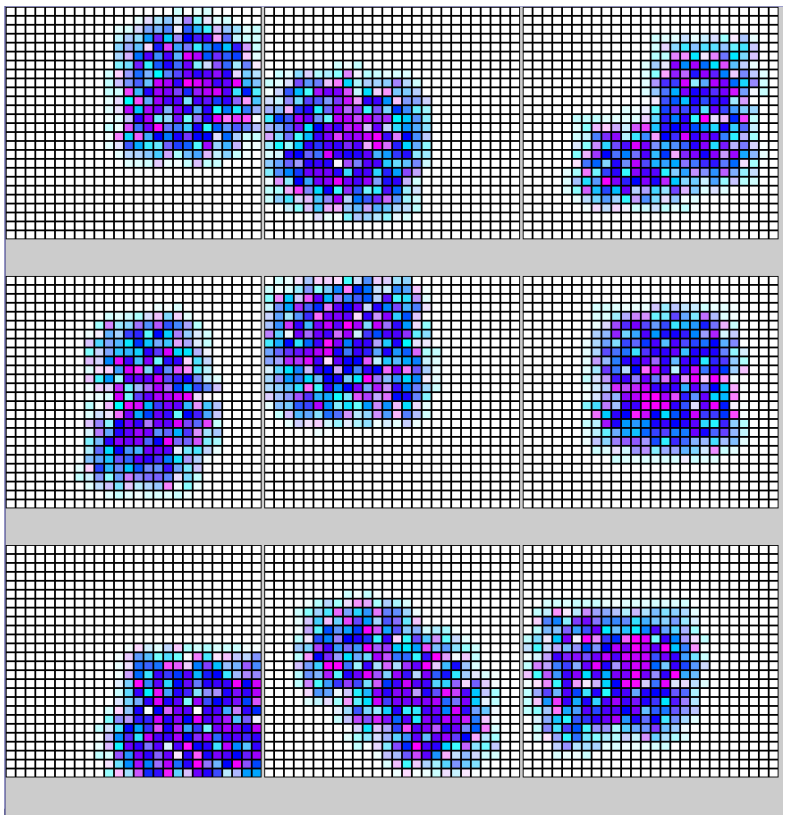

Figure 2-5 Nine different cities generated from the same algorithm

must have controlled random values and produce the same result each time the same numbers are passed to them. For objects to be placed in the same place each time, the object's modifier uses a global seed for all of its random numbers. Based on the condition of the climate and water, the flower modifier may choose to place a flower in a random location. Since the climate and water modifiers also base their values on the seed, the flower modifier will have the same information to base its decision on. The flower modifier seeds its random number generator using the global seed. With these two systems the world is completely replicable and can be completely changed by changing the global seed.

2.1.2. Modifying the Terrain

The world uses four main modifiers to add features to the world. There are two base modifiers that are vital to all other modifiers. The climate and water modifiers are the first modifiers to be applied to the terrain. The climate changes the color of the terrain based on the temperature that it determines the terrain piece to be and in the future would be used to set ground texture mapping. It also sets values for precipitation and temperature variability. The next modifier that is applied to the terrain is the water modifier. This modifier places oceans, lakes and rivers throughout the world. It modifies the terrain by dropping the elevation so that water can naturally collect in the basin. The fourth modifier is the city modifier. The order in which these modifiers are placed is important as to not place a city in the middle of a soon to be ocean. The city places itself based on the world's features. The city's modifier looks at the climate and water modifiers and compares them with the tolerances of the population it is placing. If the

population is the right size for this size of terrain and its tolerances match those of the climate then a city is placed.

2.1.3. Dividing population on split

The world is designed to handle multiple races and species each of which can have different cities. The term race is used with the thought that other mythical races such as elf or dwarf would be incorporated into the world not human races. At the onset of world generation, world populations are determined for each race and placed on the root terrain pieces. Each race has a preferred climate and terrain so that populations of races would naturally group together. Additionally the rules that govern a city's creation are also customized to each race. In this approach, a single methodology can be used for placing cities but the tolerances of races can be used to change the location and creation of cities.

The population must reside at the top level terrain and as the viewer tells each triangle to split into smaller pieces the population must split as well. This is no trivial task. Each race has a preference for a terrains temperature, rainfall, temperature variability, and amount of water on the terrain piece. In addition to these variables each race has a preference to how close each city is and how tightly packed the people living there are. Each of these variables can be combined to form a fraction of the population that is to be placed on a child terrain piece. A race can configure these variables as well as the formula that combines each of these variables.

2.2. Design

Various design patterns outlined by the Gang of Four (Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides) were used throughout this project. To make

sure the octree was only created one time throughout the lifetime of the program the singleton pattern was used. This pattern guarantees that an object has only one instance and provides a global reference to it. The manner in which the singleton accomplishes this is by defining a private constructor so no other class can instantiate it directly. The only way to instantiate it is to call a public method which returns a reference to itself that it manages internally. Another example of this pattern is the use of a dummy modifier. When a city is to be placed on a terrain piece, the city modifier which normally places cities is replaced by a dummy modifier that does nothing. This modifier is the same for all cities so it does not need to be instantiated more than once.

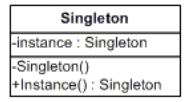

Figure 2-6 Singleton pattern class diagram

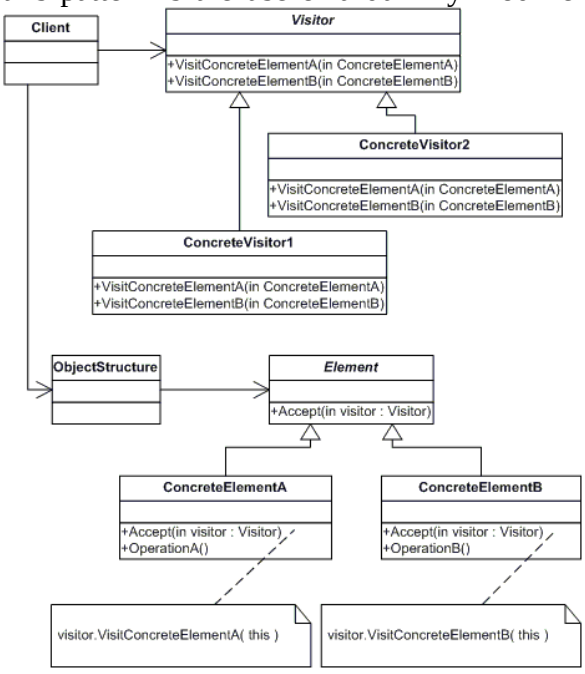

Figure 2-7 Visitor pattern class diagram

Objects in the world must be stored in a particular data structure that can be traversed quickly. An octree was implemented to accomplish this and utilized the Visitor pattern in doing so. The viewer has a system outlined for visiting each item within the tree. The viewer can then define a new operation to perform on each item within the tree without changing each class in the tree. This allows the world to be rendered in various ways independent of the objects and their implementations. According to the class diagram the client would be the viewer who uses a visitor to visit each object. The concrete visitor would be a

function to render each element in the world. The object structure would be the octree that housed all the objects. The element is the world object that each object must implement to be in the world. The concrete element is any object in the world. With this implementation new concrete elements can be added and as long as they abide to the world object interface, they will properly render without changing any other code. Likewise, a new concrete visitor could be implemented to perform additional operations on each element without changing the elements.

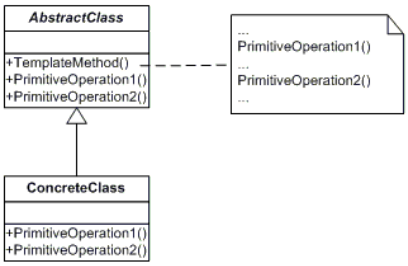

Figure 2-8 Template pattern class diagram

When designing the algorithm to generate the city structure the Template design pattern was utilized. This involves defining a general skeleton of an algorithm in a base class but leaving components of the algorithm to be defined in subclasses. There is an abstract class City which generates the city structure based on a generic algorithm. The algorithm carries out a cellular automata distribution of city blocks within the given space. The rules for that cellular automaton are defined in Human City which is a sub class of City. In addition, the seed that the cellular automaton works with is also defined in the base class, providing full customization to the generic algorithm.

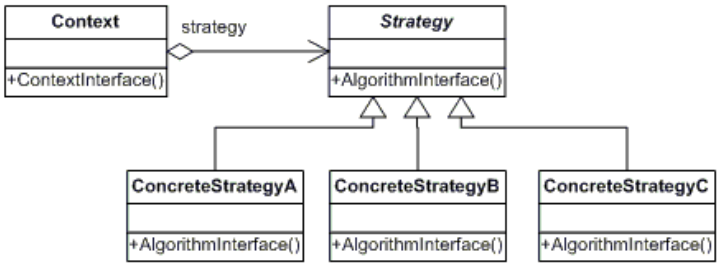

Figure 2-9 Strategy pattern class diagram

The generation algorithm also needed to be flexible and account for modification including using a completely new algorithm so the Strategy pattern was utilized. The

Strategy pattern provides an interface for an algorithm to implement. The client of the algorithm simply specifies which algorithm to use when performing the operation. The concrete strategy is the various algorithms that can be used to perform the operation. When applied to city generation, the algorithm used to generate the city can be substituted for various algorithms. There is a City Generator interface that the Cellular Automata Generator class implements and is instantiated by the Human City class. This keeps the ability to generate cities in various ways easy to change and maintain.

2.2.1. Modular Approach

A major design goal was to keep the code flexible and modular. This was accomplished in a number of areas. Looking at the design of the terrain and objects illustrates the various design patterns used as well as the interaction between the terrain piece and the modifiers.

The class diagram shown in Figure 2-10 illustrates the subclasses that implement the Terrain Modifier and Terrain Mod Info interfaces. Each derived class that implements the Terrain Modifier is collected in a linked list in the base class that implements the Terrain interface. Top Terrain, not shown here, is the base Terrain class that the world is initially made up of. Since any random number that is to be generated must be seeded by the location of the object, many classes use the Terrain class mainly for the location. The visitor pattern can be seen nested within the Octree class. This visitor class can be utilized to show the world in different views. By passing the Octree's method `visitTree` a different visitor the world can be drawn as a spherical globe or a flat map for example. The Octree is also accessed by the City Modifier Info in order to add the City object to the world. Another pattern implemented is the Decorator pattern.

The Terrain is figuratively decorated by the various modifiers that are added on the fly. Each modifier defines the objects that actually appear on the Terrain piece. Here only the three modifiers that the cities use are displayed in the form of their Modifier's Info: climate, water and city. Both the City Modifier and the City Place Holder classes are shown. The City Place Holder classes takes the place of the City Modifier after a city has been placed and so it to must inherit from the Terrain Modifier class. The Human Race Info houses the functions that determine when to place a city and how a race's population is to be broken down so it too is depicted as using the terrain class to get the size of the terrain piece.

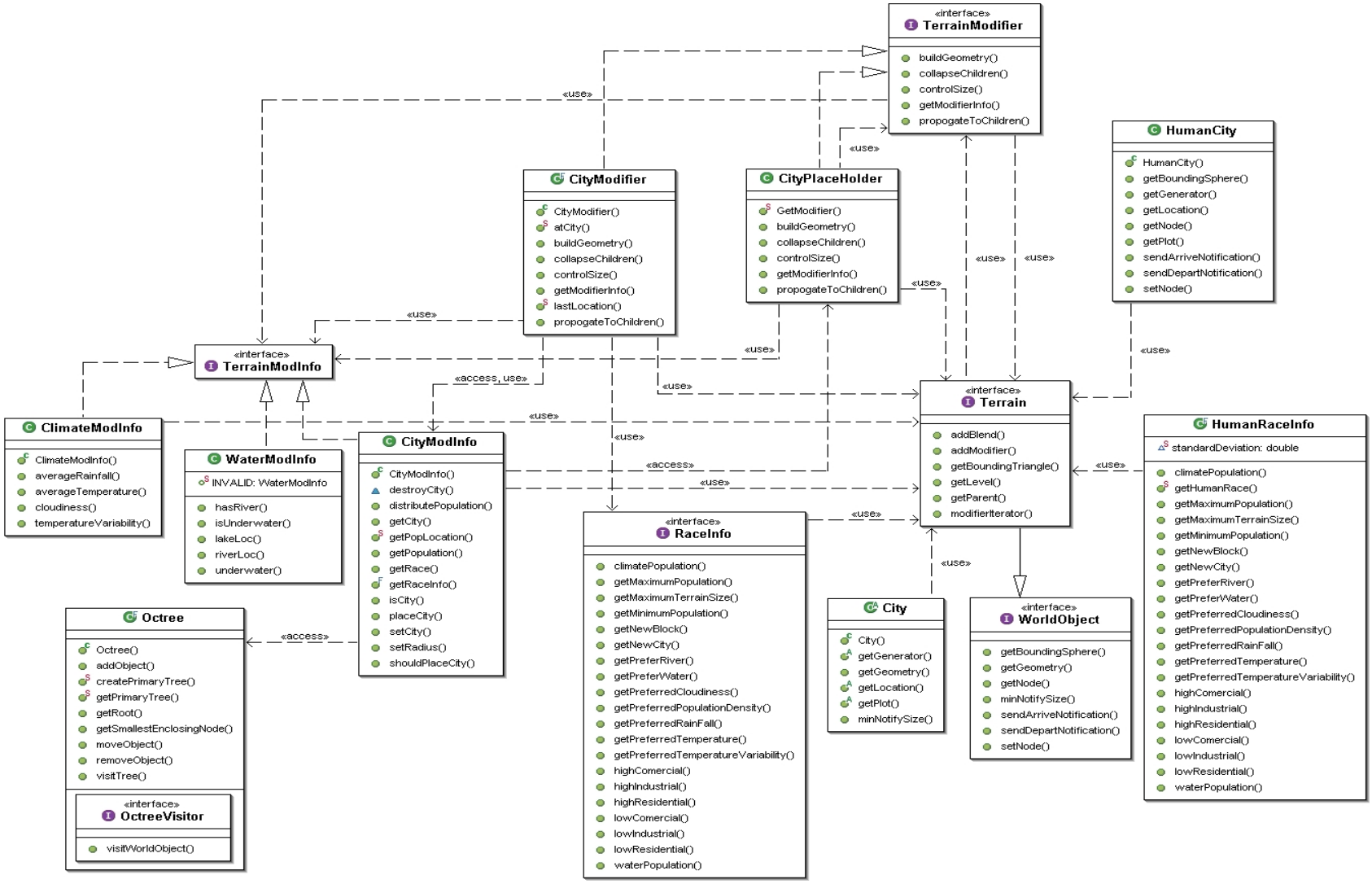

Figure 2-10 Terrain modifier class diagram

The class diagram for the city package is shown in figure 2-11. The relationship between Human Race Info is seen as the Human implementation of the Race Info Interface both uses and is a Race Info. With the Race Info class utility methods reside that can be utilized by all races as well as overridden. In addition the Human implementation of the City base class is shown. The Human City class mainly stores a local reference to the terrain piece passed from the City Modifier and chooses a City Generator to use. The City Generator class does all the work behind city creation. Two City Generators are shown a Cellular Automata Generator and a Road City Generator. The Road City Generator is an empty class designed to show that multiple generators can be created and is to be expanded upon in future work. In addition to specifying which generator to use, a race can also specify which set implementation of that generator to use as well. Cellular Automata can have many variations on setting seeds and traversing the plots. All of which can be customized within the base class or can be left up to the base Cellular Automata Generator to decide. The rules that govern the Cellular automata also reside in the customized version of the Cellular Automata Generator. So the rule sets can also be swapped completely by changing the City Generator. The plot of blocks is kept within the City Generator and consists of a matrix of Blocks and Block Information. Each city block is an instance of the Block class and every Block has a Block Info. Within a Block's Block Info resides the information about the proportions of the buildings within each block as well as any other information specific to that block.

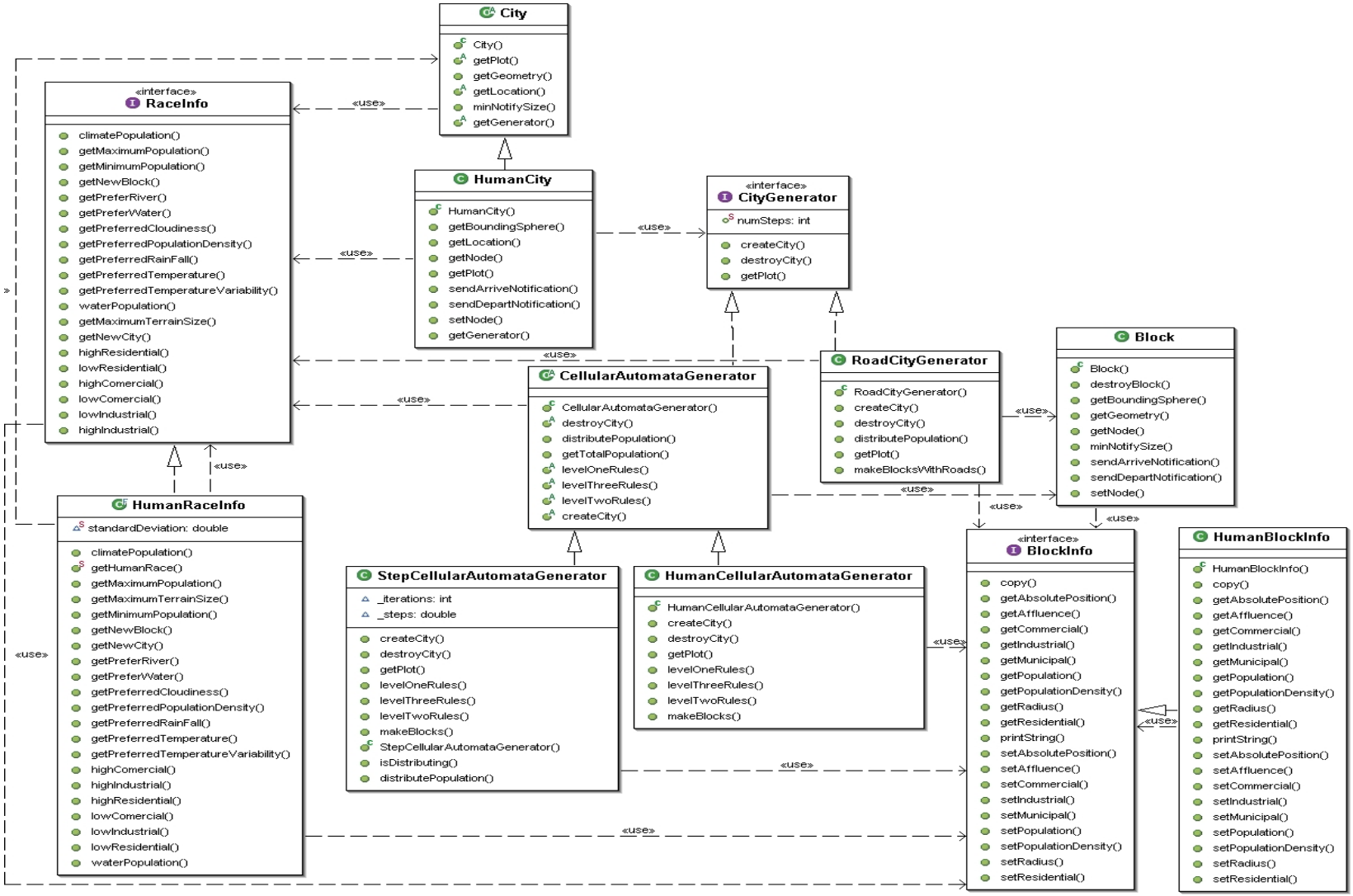

Figure 2-11 City class diagram

To understand how these classes are transformed in to objects and object model is shown in figure 2-12. This model shows a terrain object having a linked list of modifiers one of which is the City Modifier. The City Modifier will create city and keep a reference to it in the City Modifier Info. The Human City will be passed the Human Race Info and will also create a City Generator and have a matrix of Blocks. Each Block will have a Block Info.

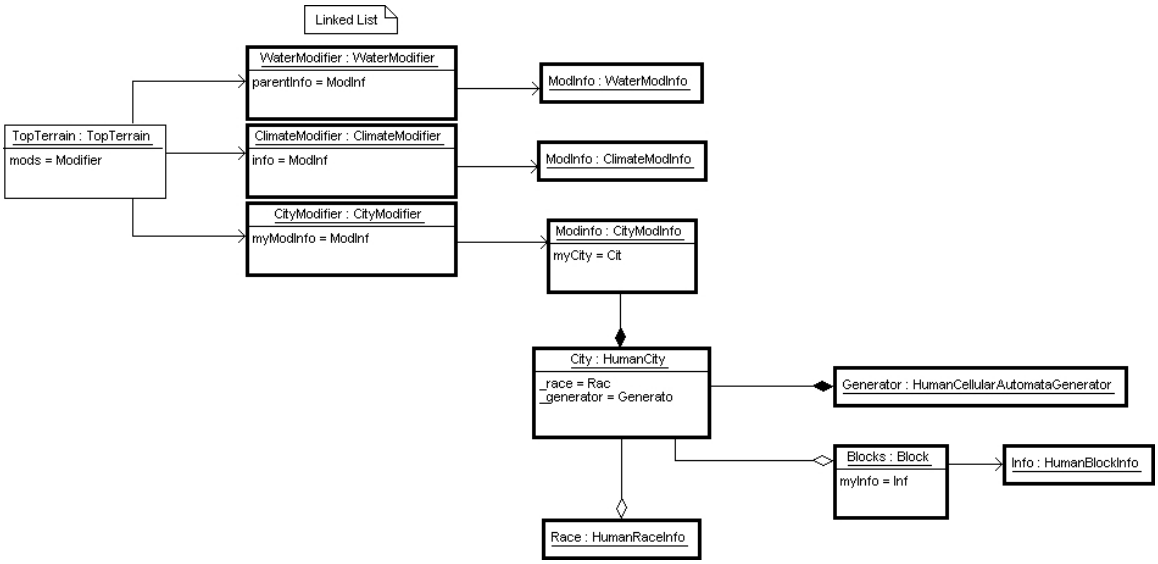

Figure 2-12 City object model

3 City Placement

3.1. Setting the Stage

Which came first the people or the city? To get an accurate representation of how cities are created cities could not be placed in attractive places, people must be placed. The people must come first to build the city. People build a city by grouping together in a small area that looks like a good place to live. Since the entire population of the world must be placed on the world the entire world population of every race must be determined before the world can be populated. For each race that is to populate the world an initial population must be determined. A mathematical formula based on the population density each race likes to live at, a coverage percent, and the size of the world is used to compute this number. Each race will configure a number that describes the amount of people per square unit. This number will be used to determine the area a person is comfortable inhabiting. This number is required as to properly space inhabitants throughout the world. Without this number people could be packed into small cities at the most attractive regions. The coverage percent is used to determine the amount of area people of this race should inhabit. This is an arbitrary number that is set when the world is created. The size of the world is also an arbitrary number that is roughly the size of Earth. With these number one can see how the area can be divided by people per square unit and the number of people is determined. For each race's population the total is divided by four since the City Modifier is placed on each of the four base terrain pieces.

3.2. City Modifier

The population for each race is stored in an array that is sent to the City Modifier. In addition to the population, an array of races that correspond to each population is provided. The City Modifier is added to four triangular terrain pieces placed end to end. Because this world population is set in the constructor of terrain piece the world population must be divided by four since the terrain piece is added four times.

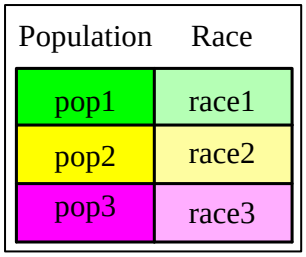

Figure 3-1 Race & population arrays

Distributing the population around the entire globe could possibly spread the population thin and make cities units apart. To alleviate this problem of finding a city, static variables are kept in the City Modifier to keep track of the nearest city. Users can then map a key that moves them to this location. This allows for finding a city quickly and easily.

If there are no cities around, mainly the viewer is not close enough to a city for it to create its self, and then the static location is mapped to the child with the highest population density in hopes of finding a city. With this technique, viewers can jump to large populations which could cause the creation of a city.

The information about the City Modifier is stored in City Modifier Information. This class is the repository for all information that later modifiers might call upon. In the case of a city, there needs to be only one piece of information that needs to be communicated to these later modifiers which is if it is actually a city or not. This prevents plants and animals from taking over city terrain. It is then the City's

responsibility to add plants, animals, and any other objects to the city. This is how things are done in reality as well. If a city wants trees in the middle of it, it must clear land and bring in trees and plants to make a central park. Animals must wander into a city or are brought in by people.

When the population is too small for a city to be placed on a large piece of terrain, the population must be split among the terrain piece.

The City Modifier handles the population distribution to each of the four child terrain pieces based on various factors. Each child terrain piece has a set of modifiers that have already been added to it, these modifiers are the basis with which the population is placed. At a high level of abstraction the City Modifier makes decisions on whether or not to place a city on the current terrain piece. If a city is not being placed, then the population for each race is distributed to the child terrain pieces.

When a terrain piece is moving out of focus, mainly its Depart Notification function is called, the Collapse Children function is called on each of its modifiers. Each modifier is responsible for removing them from the tree. In this case the Destroy City function is called. Each City class must implement a Destroy City function which destroys the blocks that were created when creating the city. Each block in turn sends a destroy building to all of the buildings within the block and it continues.

The City Modifier handles the high level decision making and calls upon the Modifier Information to do all the calculations. This design was chosen to limit the amount of code within the City Modifier and allow the Modifier Information to handle the bulk of the work. This was the City Modifier is more readable and cleaner.

3.3. Modifier Information

As mentioned before, the Modifier Information stores all the values for a particular Modifier. In addition, the City's Modifier Information stores the functions that determine whether or not to place a city and how to distribute the population.

3.3.1. City Size

When deciding on whether or not to place a city the City Size function is called to answer this question. This function determines if a city is to be placed on the terrain piece; it relies on several piece of information that is based on the race of the people. City size naturally various between cultures, for example cultures with small cities or towns might span at most five square units where metropolis cities might span more than one hundred square units.

To get adequate variations of cities sizes from race to race each race defines a maximum terrain size to build on. This prevents cities from populating very large terrain pieces and keeps them down to a reasonable level. This function guarantees that the terrain piece is smaller than the maximum size defined for the given race otherwise a city is not placed. Without this limitation and this check in deciding to build a city a large population could take over a large terrain piece such as on of the four main terrain pieces. This would mean a city spans ¼ the size of the globe. Races can define a maximum size that they would like their cities to become. Bringing this back to the real world, cultures with large cities might define a maximum size of 100 where cultures with smaller cities might set the maximum at 10.

In attractive areas such as along water and in moderate climates people might group together and once under the terrain size barrier try to make a large city of people. To also combat large cities concentrated with people, a maximum population size is determined for each race. This will prevent cities from having a very concentrated amount of people where in reality the city would be many cities around a central area. This promotes spreading people out along the globe and inhabiting less attractive areas to bring down the population density.

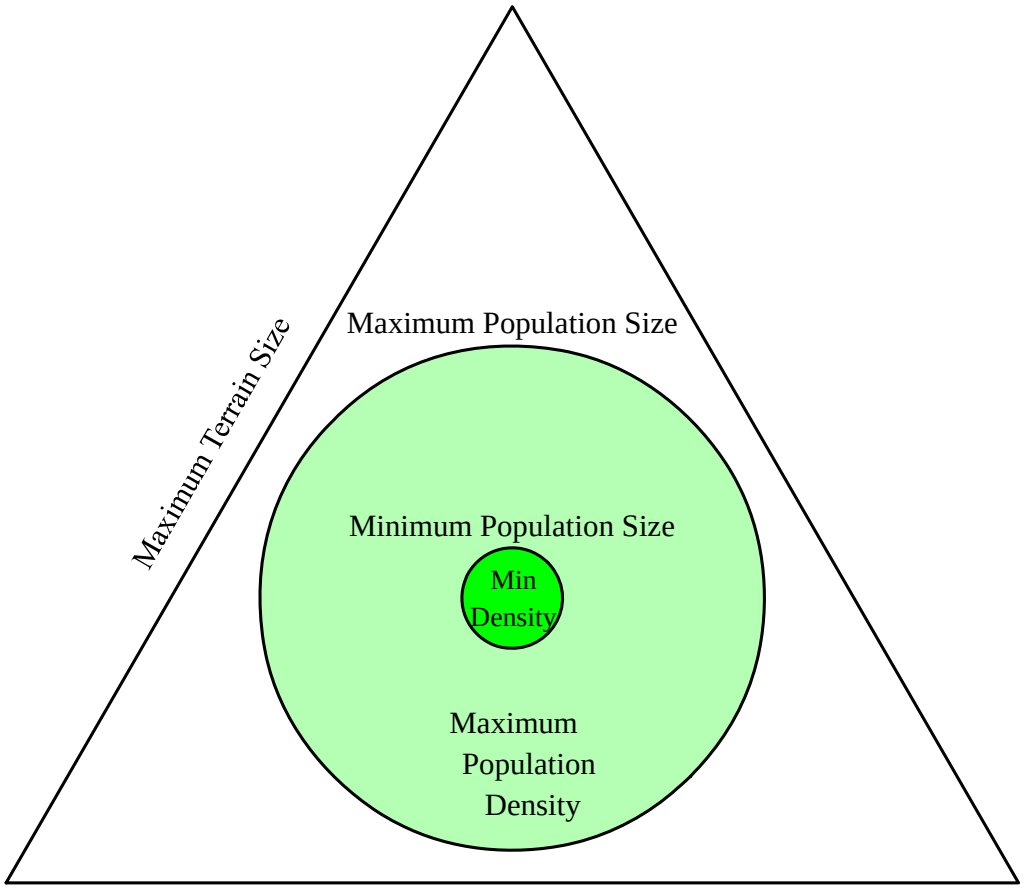

Figure 3-2 Restrictions on city placement

To further control the granularity of people within cities, a preferred population density is defined for each race. The population density is the amount of people per

square unit that the people of that race would prefer to live at. This allows the attractiveness of a terrain piece to be controlled. A lucrative terrain piece does not seem to appealing after ten million people have settled in.

Based on the preferred population density a minimum and maximum population per square unit can be computed using the size of the terrain piece. These two values in turn can also help to constrain the size of the cities. The minimum will promote grouping together of people so there are some cities that are large. The maximum population density will cut back on over crowding of people within cities. With these checks in place the proper city size can be matched to a terrain piece.

There are some cases when the population does not lend itself to creating a large city so there must be thresh hold in which some city is placed regardless of population density. Each race must also define a minimum population for a city. Once the population is below that number, a city is placed. This final check prevents people from spreading out perfectly and never grouping together enough to create a city.

3.3.2. Population Distribution

Distributing the population of several races depending on each of their tolerances as they relate to general numbers about the climate and water is no trivial task. At a high level, the function determines the population percentage of each race to place on each terrain piece based on their attractiveness; and if the population is too small, it places the entire population on the most attractive terrain. To break this description down further the first step is to determine the attractiveness of each terrain piece.

As noted earlier each terrain piece has modifiers added to it as it is created. The water and climate modifier determine these attractive qualities. Before the City Modifier is added the climate and water modifiers are added and their information is determined. Each terrain piece's climate and water information is examined and compared against the race's tolerances. Without these initial pieces of information the population would have no basis to split except by choosing a random number. With the climate and water modifiers in place the world is created more realistically and correctly.

Looking first at the climate several factors are considered: cloudiness, average rainfall, average temperature, and temperature variability. Each race must define a preferred value for each of these fields. These are used to determine how much the terrain corresponds to what they prefer. There could have been more factors to consider for each piece of terrain as it relates to climate, but these four fields are broad enough to cover most concerns.

In addition to the climate modifier, the water modifier information contains information on the percentage of terrain covered by water and if the entire terrain piece is under water. As before each race must specify a preference for water to determine how close to water they would like to live. This allows cities to be placed near or away from water and prevents cities from being placed underwater.

Each race also has the flexibility to specify the way this information is processed. Each race must define functions that determine the percentage of the population to place on a particular terrain piece for climate and water. These functions include Climate Population which determines the percentage of a given population that prefers this

climate. As well as Water Population that determines the percentage of a given population that prefers this amount of water.

3.4. Race Tolerances

The Race Information stores all the information about where they want their cities, how big they are going to be and how the cities are made. This section will focus on the race's preference for city location.

3.4.1. Climate Population

As mentioned before, each race must define a Climate Population function. This function looks at the values for cloudiness, rainfall, temperature and temperature variability of the terrain. It then pulls the race's preferred values for each of the climate's features and compares them one by one using Chebyshev's theorem discussed below. Each of these values is averaged. This final value is what Climate Population returns as the percentage of the population that should be placed on this child terrain piece.

3.4.2. Water Population

The Water Population function performs a similar calculation but varies slightly. The initial percentage of the population to be placed is set to zero. If the terrain piece is above water then the water percentage is evaluated. The water percentage is compared to the race's water percentage preference using Chebyshev's theorem. If the terrain is below water then zero is returned. This final value is the percentage of the population that should be placed on this child terrain piece.

3.4.3. Chebyshev's Theorem

Chebyshev's theorem is used to interpret the standard deviation of the data set. Since there is no data set for the values given by the climate and water modifiers. A standard deviation must be calculated based on the empirical rule. The value given from the modifiers is said to be the mean. The values from the modifiers are between zero and one and based on a standard bell curve the standard deviation is fixed at one third. This allows for a nice bell curve.

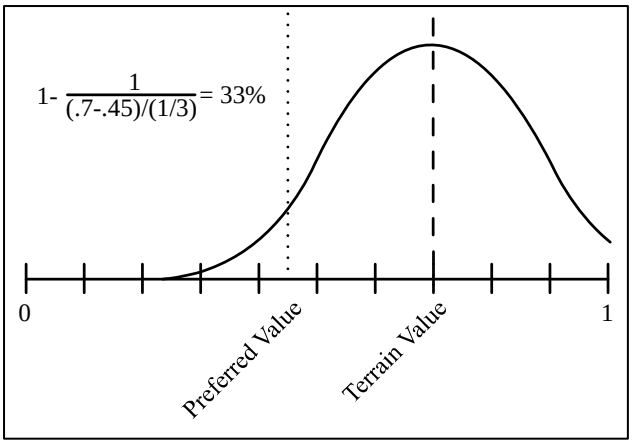

Figure 3-3 Chebyshev's theorem

```
return 1-(1/((terrainValue-prefferedValue)/standardDeviation));
```

3.5. Dummy Modifier

After the City Modifier Information tells the City Modifier to place a city on a terrain piece, the City Modifier places a Dummy Modifier on the terrain pieces below the city. The removes the heavy calculations that are done to decide if a city needs to be placed or not and replaces it with a modifier that does no calculations. The Dummy Modifier Information merely tells the later modifiers that a city is being placed on this terrain piece and propagates itself to the smallest terrain piece.

3.6. Destroy City

Each object is responsible for creating itself when the Send Arrive Notification function is called and destroying itself when the Send Depart Notification function is

called. The City Modifier calls the city's Destroy City function if it created a city. The city is then responsible for destroying the plot of blocks that was generated by the City Generator and passing the Destroy message down to the Blocks and the Buildings.

3.7. Population Distribution Figures

This section shows the population being distributed and shown in map form. The process of displaying this map is sending a new Visitor class to the Octree's Visit Tree method which traverses the tree according to the Visitor provided. Each terrain piece is examined for the amount of population the City Modifier has on it. Each race's population is shown in different colors.

3.7.1. Single Population Distribution

The map shown in Figure 3-4 is every triangle within the world displayed in two dimensions. The grey triangles represent terrain pieces with no population on them. Since there are no grey triangles, there is population on every triangle. The red triangles represent terrain pieces with the population on them. Black triangles represent a dense amount of people. The larger triangles are black because the population hasn't split.

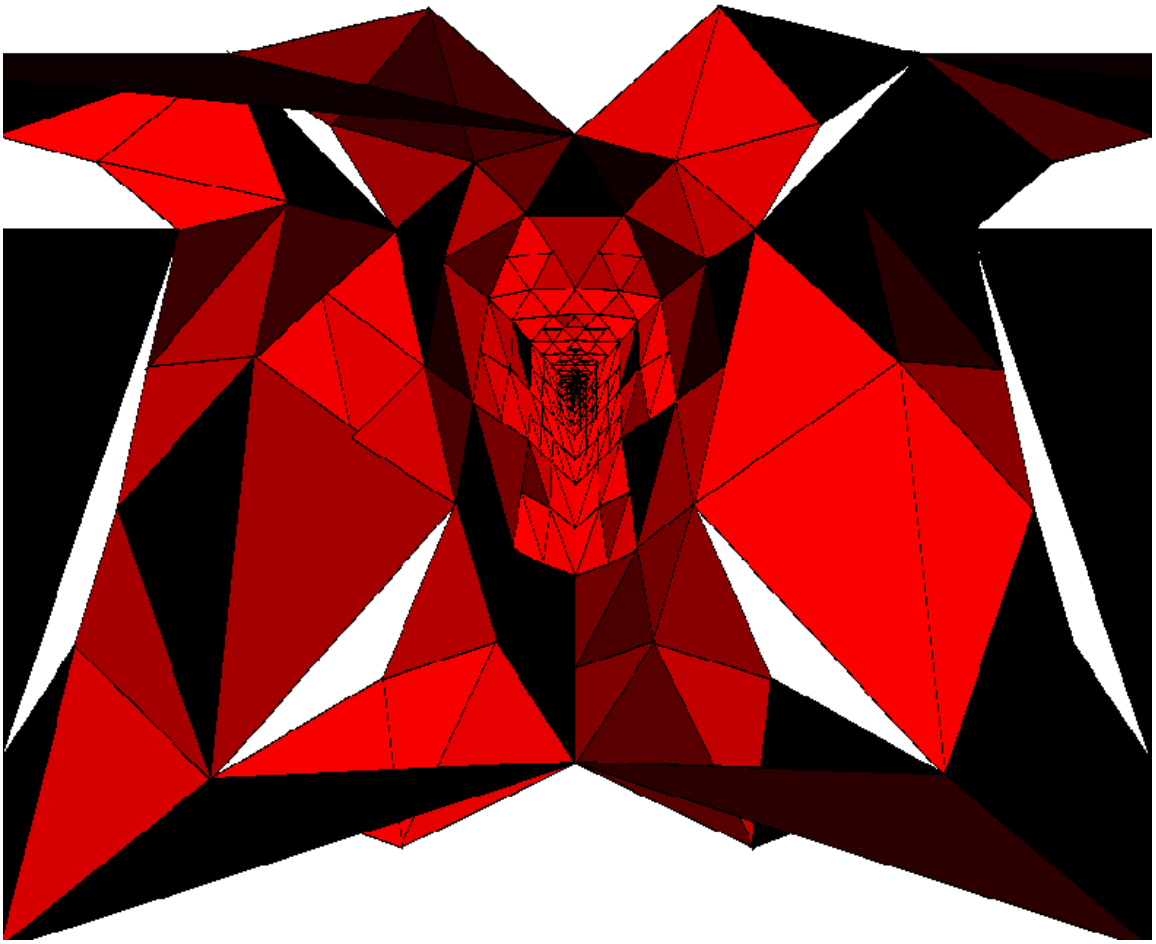

Figure 3-4 Single Race, World Population - 400 million, 2 cities

3.7.2. Multiple Population Distribution

As noted before multiple races can be added to the world. Each race defines what an attractive piece of terrain is; therefore each race is placed on different terrain pieces. The map shown below is two different races added to the world. The red color is the exact race shown before in the same world. The blue color is the new race in triangles without the red race. The purple color is both races in the same triangle. As before the grey color is triangles with no population. Here you can see how the two races pick different triangles to populate based on their tolerances.

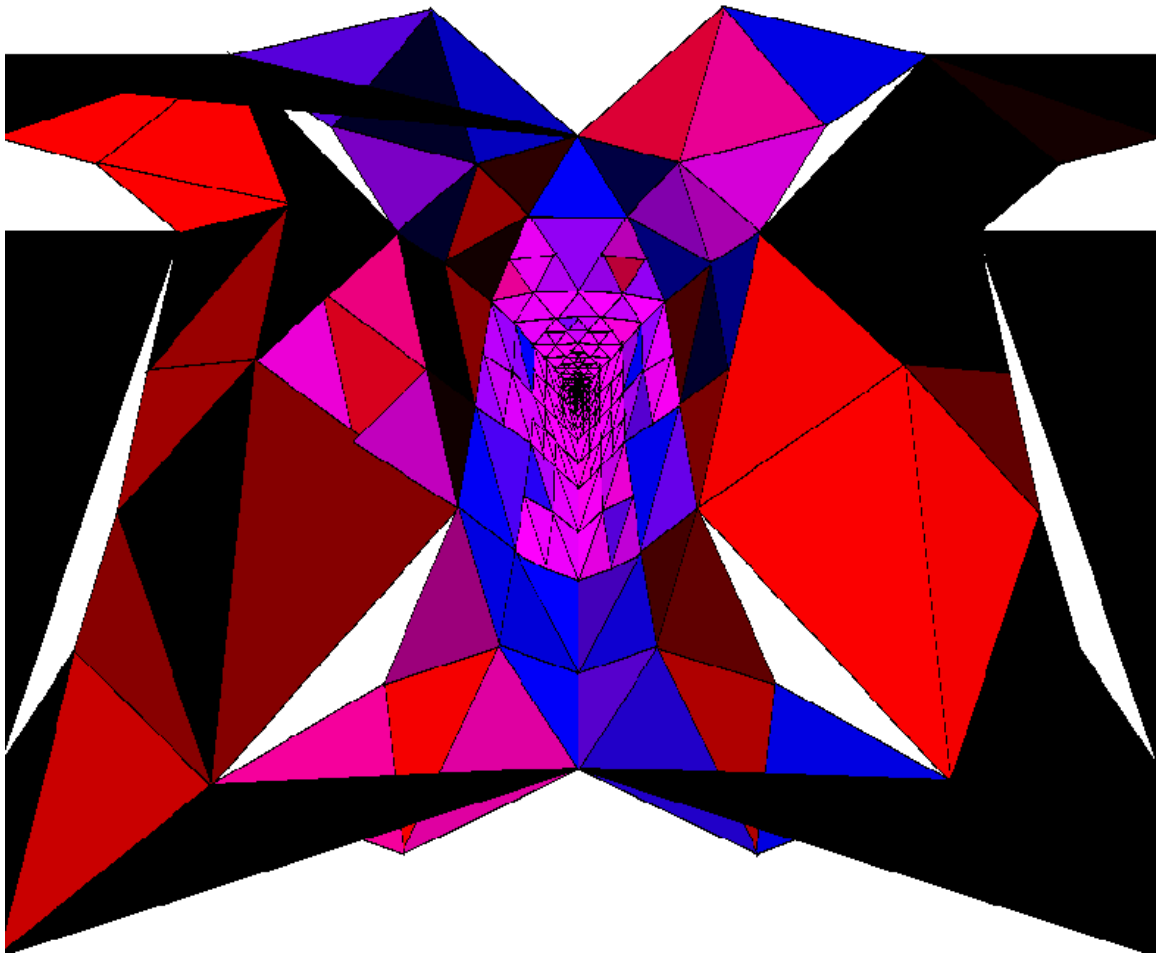

Figure 3-5 Two Races, World Population - 400 million, 5 cities

4 City Generation

When a city is created it calls upon a City Generator to create the city. Here in lies the bulk of the work and the fun. Involved in city generation is the creation of an interface between the City and the buildings within the city. The algorithms that create the blocks utilize a Cellular Automata approach.

4.1. Blocks

The basic composition of what the City Generator manipulates is a city Block. The Block consists of a basic World Object class that creates a series of buildings based on the Block Information. The Block Information is the main focus of what the city Generator deals with. The Block Information contains values for residential, commercial, industrial and population. These are the main variables that every City Generator manipulates when creating the city. In addition to these base variables the Block Information class is expanded to also encompass municipal, affluence and population density. Based on these additional figures buildings can be fine tuned to the area they are placed. The actual building generation is outside the scope of this paper. The Block Information also guarantees that every value that is configured is within its allocated range. Residential, commercial and industrial values must be between a range of zero and one.

4.2. City algorithm to block level

City generation was designed to be as flexible as possible, the design allowed for different city generation algorithms to be interchanged inside a single city class. The focus of city generation centered on the cellular automata algorithm.

All City Generation algorithms all have the same task to distribute a large amount of population over a given city radius. The manner in which each of them does that is entirely up to the City Generator. Not all of the Population even needs to be put down on the City. The City does not enforce any other restrictions on the City Generator but to create a plot of blocks that the City can refer to. The algorithm then is simple: place as much of the population in the city as you can and stay within the radius of the city. It is up to the each implementation of the City Generator to decide how to carry out this rudimentary algorithm.

4.3. Cellular Automata Method

Cellular automata were pioneered by John von Neumann. His techniques were applied by Conway to create a game called the game of life to model self-reproducing organisms. The Game of Life starts off with a seed of cells on a grid that are either alive or dead. The rules are as follows a dead cell with exactly 3 live neighbors becomes alive or is "born". A live cell with 2 or 3 live neighbors stays alive; otherwise it dies from "loneliness" or "overcrowding". The matrix of cells is parsed cell by cell each of which looks around and decides whether it should be born or dies. The interesting aspect of this is that patterns emerge from this simple game. This central idea of turning on and off

based on the surrounding blocks and deciding what it should become is the core of choosing a cellular automaton to model a city.

This approach was taken for city generation because it follows the model of real city generation. When a group of settlers come upon an attractive piece of land the build up the necessities for a town and begin living there. As more people come upon this town the town expands. The expanding process is done by a person looking at an empty piece of land and based on the buildings around it and the needs of the people create a new building. After a small town expands enough it may join other small towns and form a city or a metropolis.

4.3.1. Cellular Automaton Plots

For the remainder of the paper several city diagrams will be shown. These city diagrams are an aerial view of the largest area a city could possibly occupy. Each block is a one by one square unit block that could be an empty undeveloped area or the central part of downtown. Each block is shown by representing the amount of population and various city building types. The block color represents the building types. Green is residential, blue is commercial, and red is industrial. These red, green, and blue colors are combined to create the final block color. The brightness of the color represents the amount of population is on each block. A block in full brightness has high populations, while blocks that appear washed out are less populated. Below is a small city that is shown in a large scale as to highlight the different blocks. In later images, the blocks will be much smaller as to show the city shape and over arching patterns in building placement.

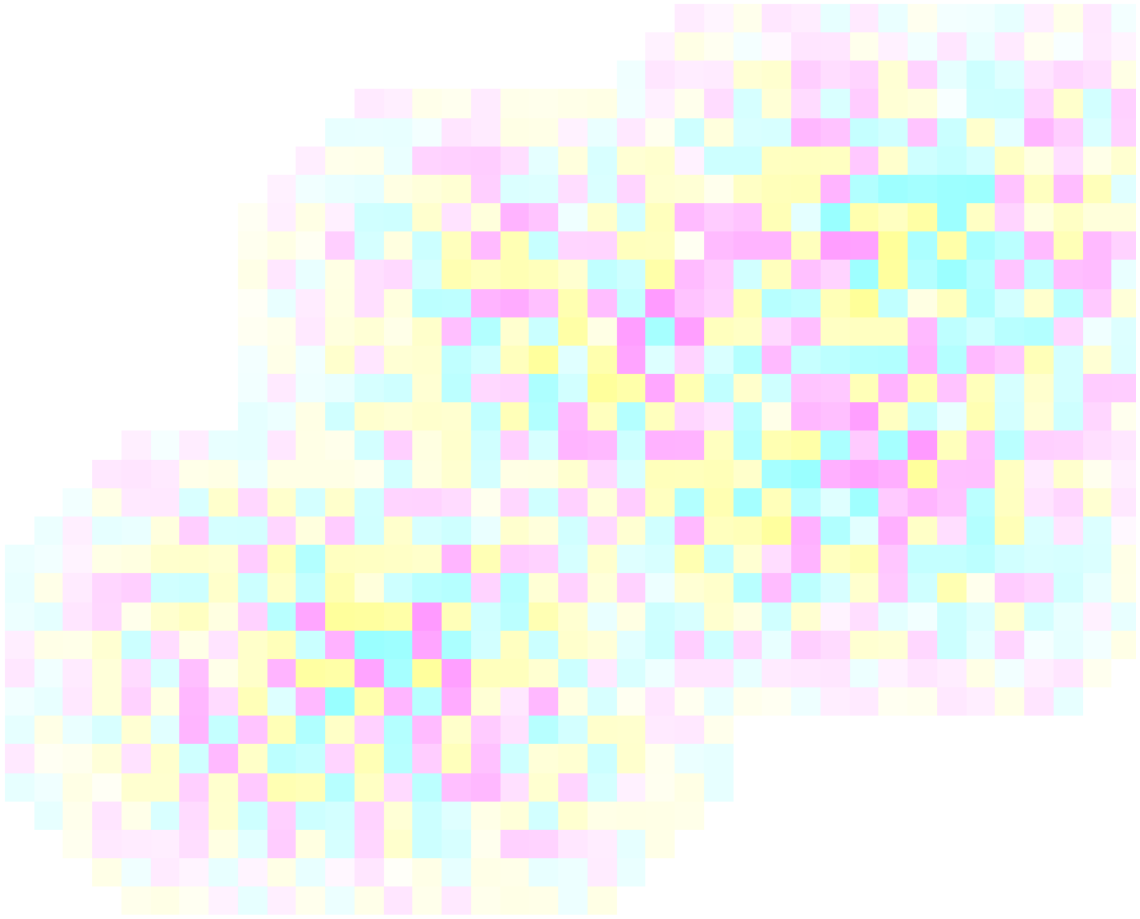

Figure 4-1 Small city shown with large blocks

4.4. Human Cellular Automaton

There are many ways the Cellular Automata can be configured and still be classified as a cellular automaton. For the remainder of this paper, the Human Cellular Automaton will be the algorithm referred to unless otherwise specified. All values that are configured can be changed in other implementations.

Instead of blocks turning on and off, the composition of blocks was centered on the proportions of types of buildings within a city block. The three types are residential, commercial, and industrial. The cellular automaton is seeded by randomly selecting cells within the center of the city. The seeds are composed of a small amount of residential, commercial, and industrial buildings which are considered to be the building blocks of a city. From this, the algorithm is turned loose on the larger matrix which now has only a few city blocks. The cellular automaton goes in and visits each block and looks around the block to decide if this block should be created. If it is created the proportion of residential, commercial, and industrial buildings is decided. The game of life was expanded further to not only examine the blocks surrounding the central block but look two and three levels beyond that to give it a more intelligent view on the way the city should be constructed.

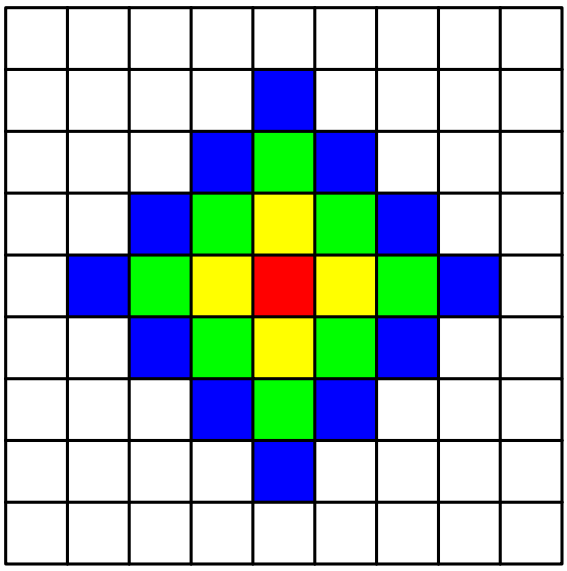

Figure 4-2 Levels of blocks (1=yellow, 2=green, 3=blue)

Based on this framework a simple or complex set or rules can be made for the cellular automaton that outline when a city block is to be created and how to adjust the ratio of buildings within a city to generate an accurate representation of cities. Cities were created in this manner to replicate the building of up cities that occurs in reality. This method also lends itself to reality by the common sense approach of a city block developing based on surrounding city blocks.

4.4.1. City Seed

Each base implementation of the Cellular Automata can implement each part of the algorithm uniquely. Each implementation must decide the manner in which the initial seeds of the city are placed. The seed is essential to the operation of the cellular automata algorithm. With out it, the algorithm would do nothing.

There are many effects of creating different seeds to start a cellular automaton. The number of seeds has a great impact on how the end city is going to appear. Each seed creates a ripple effect within the algorithm that collides and joins other seeds. How these seeds combine is determined by the rules of the cellular automata.

The decision to put a city down is difficult because the size of the terrain piece required for a given population is not clear. The city must not expand beyond the edge of the city because given the nature of the world each triangle must contain the whole of the object. If the city were to butt up against the edge of the triangle there would be a dense city region against an empty terrain. This is obviously an undesired effect of city generation. To combat this seeds are placed in the center of the triangles in hopes that the city does not expand beyond the outskirts of the allocated space. It is practically impossible to judge the behavior or size of a city given that the seeds are placed randomly in the center.

The random placing of the city seeds is done by seeding the random number generator with the position of the center of the triangle. The number of seeds is a random number based on the position of the city. Too many seeds lead to a spaced out city with

not much of a center, where too few seeds lead to a highly centric city with all the growth stemming from a few seeds.

The shape of the city is based largely on the seeds in the city. A city with a few seeds spaced relatively together will create a bubble effect. A city with many seeds creates more intricate shapes.

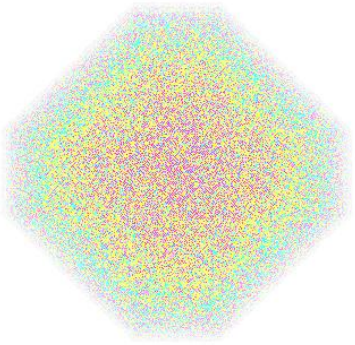

Figure 4-3 Human CA with 1 seed

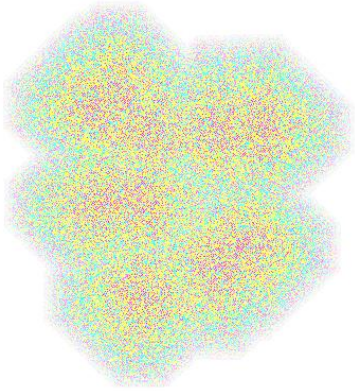

Figure 4-4 Human CA with 10 seeds

4.4.2. City Growth

With the seeds in place the Cellular Automaton can be set in motion. The processing of the Cellular Automata is all the same so the functions for processing are in the base Cellular Automata Generator class. The function Distribute Population is called which sends the generator into a loop that is contingent on the current population being less than the population to be placed. Each cell is checked one by one. A cell is checked for near by neighbors with any population. If there are neighbors with a population then that cell can be created. This is done using a number of methods.

The function Has Population is called sending the coordinates of the cell being examined. To guarantee that the cell is within the bounds of the city plot it is checked by a function called Is Valid Point which also takes the coordinates of the cell. It checks the coordinates against the city's bounds and return true if it is valid, false otherwise. After the point is validated the average population of the cells around it is found. This is done by calling a function called Get Average. Get Average requires the coordinates of the cell being examined the level of depth to check and the value to get an average of. The level is currently only between one and three. It is the number of blocks away from the cell to look at. The coordinates of each block in the varying levels are kept in a static matrix that contains the necessary offsets to the center cell to obtain the surrounding cells. The value to look at is passed in as a character and a switch resolves which field to examine. With this approach, the average of any field can be obtained easily. The offset coordinates are also validated before their values are retrieved to verify the cells are within the boundary. This is especially useful when examining cells at the edge of the city plot where the surrounding cells predominately lie outside the city plot.

When a cell is determined to be valid and has population around it, the coordinates of that cell are sent to the Compute Block Info function. This function goes through each level of proximity that a cell looks at and performs the following calculation. The averages of the residential, commercial, industrial and population are computed and stored. Those stored values along with the coordinates of the cell being computed are passed through the rule set for that level. Based on the rules which are defined in the base class, the cell is modified. The cellular automaton moves on to the

next cell after all three levels have been examined and each level of rules has modified the cell.

This process continues on each cell. When all cells have been examined and modified accordingly, the cellular automaton has gone through iteration.

The city's growth is polynomial; the degree at which it rises depends on the implementation used. The Human Cellular Automaton looks at all the cells immediately surrounding the candidate cell. Other Cellular Automata can look at all the cells at each of the three levels to make the city's population grow more exponentially. Table 4-1 shows the growth of a city of 500,000 people using the Human CA; each line represents the level that was examined when deciding when to create a block. The implementation that looked three levels away from the cell was able to grow more quickly where implementations that only looked one level away grew more gradually.

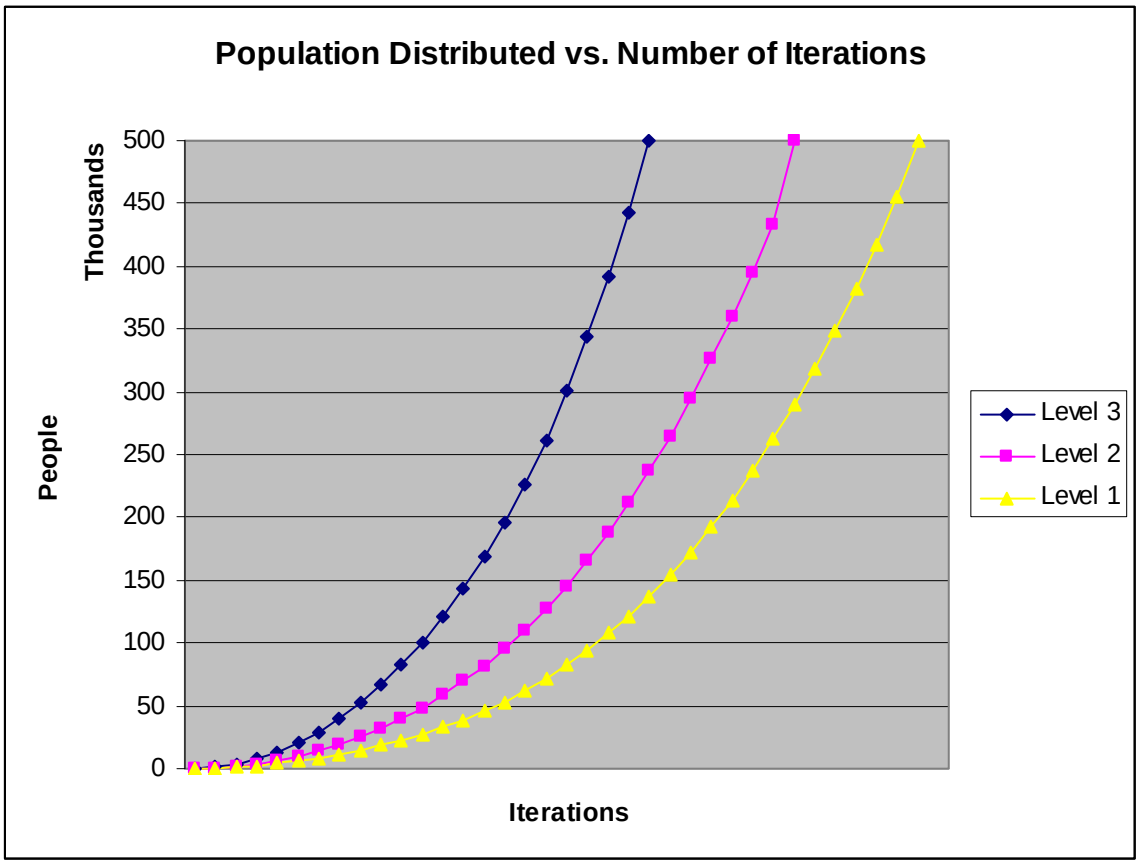

Table 4-1 The Cellular Automata process on city with population of 500,000 people.

4.4.3. City Evolution

The city slowly builds up by adding small amounts of the population to various cells. The city grows up polynomial because each cell causes the cells around it to take on apart of the population. As more cells become active they take on more of the population. There are instances when the population becomes stable. The rules of the Cellular Automata can remove portions of the population to simulate people moving away from that area. It is not unlikely as population distribution continues that the rules of the Cellular Automata put people in the city as fast as it removes people from the city. This is referred to as spinning. Spinning can be measured in a number of different ways.

Common characteristics of spinning include its appearance in the middle to late stages of population distribution. Spinning may occur early on but it is not dangerous to the program unless it occurs for an extended period of time. In this implementation of Cellular Automaton spinning is declared after ten consecutive iterations where spinning is occurring. Spinning can be detected in a number of ways as well, yet detected here by noting when the population has not increased by more than five percent.

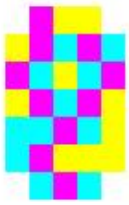

Figure 4-5 Single seed, one iteration later

As cities expand the phenomenon observed in the Game of Life is not here. That is the blocks themselves do not show the patterns. When a surrounding block is detected from any direction, a new block is created. When there are too many blocks, the block does not die but is changed in a different manner as when there are fewer blocks. So the patterns that are usually observed in the blocks are shown in the values of residential, commercial and industrial. The blocks merely expand uniformly around the seeds while the types of buildings show the interesting patterns.

The seeds expand in different ways depending upon the level the Has Population function is set to look at. If the level is set to one an oval like shape is produced. If the level is set to two then a rounded rectangle shape is produced; and if the level is three then a diamond shape is produced.

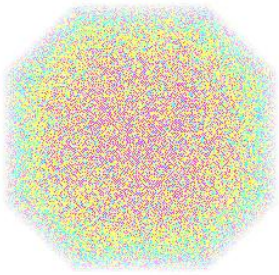

Figure 4-6 Level 1

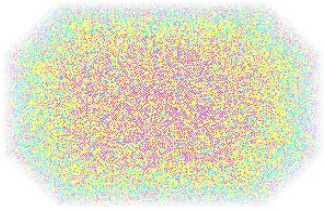

Figure 4-7 Level 2

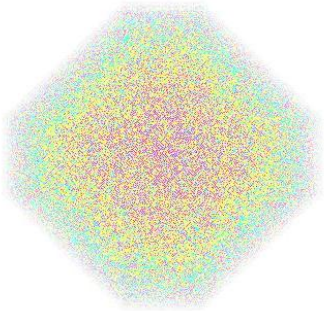

Figure 4-8 Level 3

The cities build up in a gradual manner starting from the seeds and create a kaleidoscope effect upon impact with each other. The expansion and collision is the key to creating a good rule set and having the rules properly interpreted within the cellular automata. The following figures are a city of size 700,000 people using the Human CA. There are 211,600 cells available within the terrain piece to place the city. The radius of the city can be up to 230 units. Each figure is five iterations later in the growth process. The entire process takes 25 steps. The first iteration is shown and each seed is highlighted with a red circle to mark its location.

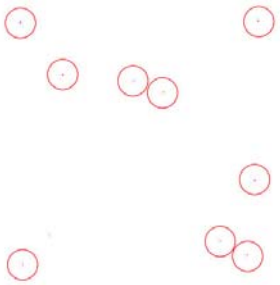

Figure 4-9 Iteration 1

Figure 4-10 Iteration 5

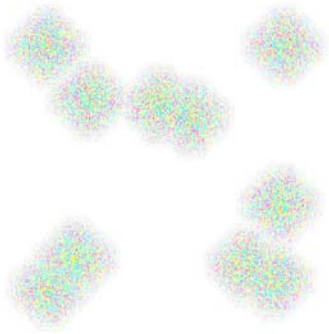

Figure 4-11 Iteration 10

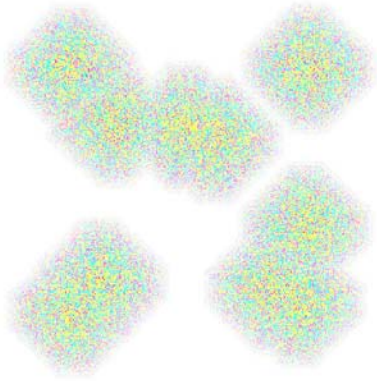

Figure 4-12 Iteration 15

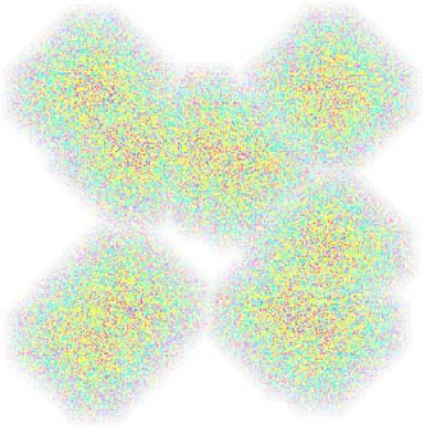

Figure 4-13 Iteration 20

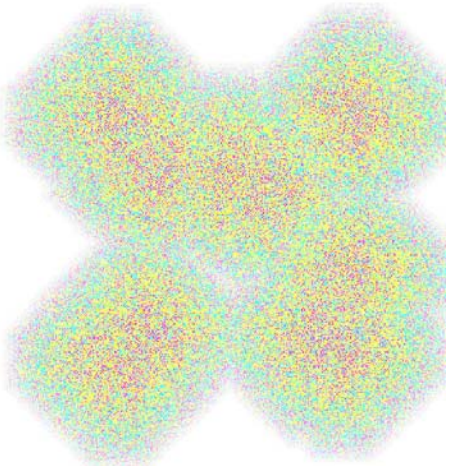

Figure 4-14 Iteration 25

4.4.4. Different Rule sets

As mentioned before the rules for the Cellular Automaton can be changed from race to race. Illustrated below in figures 4-14 and 4-15 are two different rule sets operating on the same initial city. The city is made up of one million people with a radius of 230 square units. Each block is one square unit and there are 52,900 blocks.

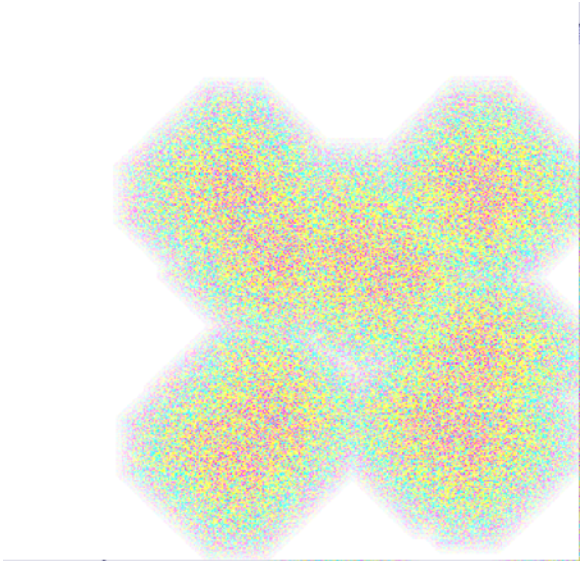

Figure 4-15 Cellular Automaton 1

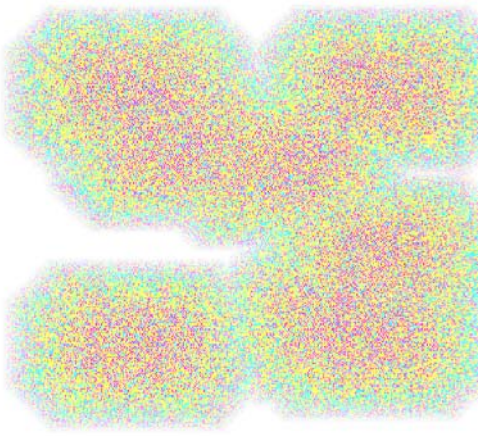

Figure 4-16 Cellular Automaton 2

5 Conclusion

In this paper the process of designing a robust system to place and populate cities was examined. The uses of several design patterns (Gamma) were incorporated to create a customizable interface for all aspects of placement and creation. In addition the technique of placing a city by dividing populations of races throughout the world was examined. The people of each race picked which triangle that they preferred based on their tolerances. When the size of the people matched the size of the terrain piece a city was placed. A Cellular Automaton generator was chosen by each race which involved choosing from a number of Cellular Automaton rules. The rules and the tolerances of each races determine the shape and growth of the city.

5.1. Future Research

City generation using cellular automaton is a fascinating approach and could be expanded to incorporate the transportation aspect of city growth. The city generator class can be implemented to generated cities that place roads within the city. It could also use the Cellular Automata Generator and place small roads between the seeds of the Cellular Automaton and grow up roads to highways. Roads could be created as cells are created and grow up into larger roads.

In addition other models of city generation using L-systems could be expanded upon to involve more meaningful building placement. L-systems could be used by placing a city seed with the essential components of a city complete with roads. As the L-system iterates over the seed, the city would grow up around the city seeds to create a city.

Ideally human simulations could be run within generated cities to see the patterns in the generated cities to validate their realism and further refine the rules that govern creation. In addition traffic simulations could be run within these dynamic cities to test out traffic control systems and mass transportation algorithms.

Bibliography

1. Gamma, E., Helm R., Johnson R., Vlissides J. *Design Patterns*, Addison-Wesley Professional, January 15, 1995
2. Gardner, Martin, *MATHEMATICAL GAMES The fantastic combinations of John Conway's new solitaire game "life"*, Scientific American, 223, October 1970, 120-123
3. Greuter, S., Parker, J., Stewart, N. and Leach, G. 2003, *'Real-time procedural generation of 'pseudo infinite' cities'*, Proceedings of the First International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia, D. Arnold and G. Wyvill (ed.), ACM Press, New York
4. Haas, Markus A. *Dynamic virtual worlds : a notion of realism in future virtual reality applications.* Trinity University San Antonio, TX 1996.
5. Hillier, B. & Hanson, J., *The Sociallogic of space*, Cambridge: CUP, 1984
6. Ingram, R., Bowers, J. and Benford, S., *Building Virtual Cities: Applying Urban Planning Principles to the Design of Virtual Environments*, in Proc. Symposium on Virtual Reality Software and Technology (VRST'96), Hong Kong, July 1996
7. Kevin Lynch, *The Image of the City*, M.I.T. Press, 1960
8. Lewis, Mark. *Large scale virtual worlds : algorithmic construction and dynamic methods.* Trinity University San Antonio, TX 1996.
9. N. Farenc, S. Raupp Musse, E. Schweiss, M. Kallmann,O. Aune, R. Boulic, D. Thalmann,, *A Paradigm for Controlling Virtual Humans in Urban Environment Simulations* , Applied Artificial Intelligence Journal, 1999
10. Palash Sarkar, *A Brief History of Cellular Automata.* ACM Computing Surveys, Vol. 32, No. 1, March 2000
11. Schaefer, Scott. *A technique for wind visualization in plants generated with modified L-systems*. Trinity University San Antonio, TX 2000.
12. Wilmes, Ted. *A real-time archaeological data visualization tool.* Trinity University San Antonio, TX 2004
13. Yoav I. H. Parish, Pascal Müller., *Procedural modeling of cities*, International Conference on Computer Graphics and Interactive Techniques archive Proceedings of the 28th annual conference on Computer graphics and interactive techniques table of contents. SIGGRAPH: ACM Special Interest Group on Computer Graphics and Interactive Techniques ACM Press New York, NY, USA. 2001 301-308