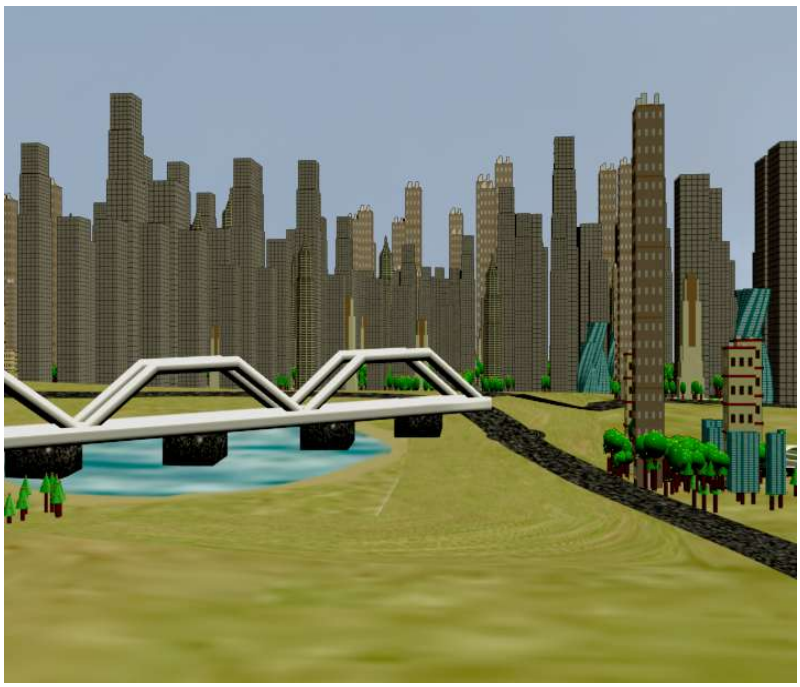
Procedural City Generator



MSc Master's Project

Praveen Kumar Ilangovan

i7834000

My Sincere Thanks to

Jon Macey

Peter Comninos

Phil Spicer

Peter Claes

Nicholas Hampshire

Michael Cahsmore

Udhay Shankar

Sundararajan Srinivasakannan

and all my fellow course mates

Graham and other sidefx forum members

Table of Contents:

List of pictures:

Abstract:

The objective of this project is to generate a set of digital assets (HDA) for Houdini which can be used to create an accessible and interactive tool to automatically generate a realistic and detailed city layout suitable for use in real time rendering. To accomplish the task, HOM – Houdini Object Model, an API (*Application Programming Interface*) which lets user get information from and control Houdini using the Python scripting language and the existing powerful Houdini nodes has been used.

1.0. Introduction:

Contemporary computer games are often situated in large urban environments. Animated movies and some feature films are also required to create a digital city for their visual and special effects shots. It necessitates a time consuming, complex and expensive process of content creation which involves modelling the terrain, road network, street patterns, vegetation and other associated features. Meeting the customers need in quality, realism and scale makes the process more complicated. As a result of this, the time and money that could have been spent in improving the game play or adding innovative features are lost on content creation. A potential solution to this problem is creating everything by procedural methods. Previous researches has showed that procedural techniques like Fractals, L-systems, Noises (Perlin noise, Voronoi noise, etc) can be used to recreate natural phenomena like plants, textures, etc.

The key aspect of procedural techniques is that it characterises the entity, be it geometry, texture or anything for that matter, in terms sequential instructions rather than a static block of data. These instructions can then be called on whenever an instance of the asset is initialized and the various characters can be parameterized to allow the generation of instances to be unique from other instances. A typical example is creating 3D primitives say, cuboids with random heights.

Now it is quite obvious why Houdini was chosen as a platform for this project. The node based environment of Houdini makes it completely procedural.

2.0 Previous Works in this field:

Procedural techniques like Fractals, L-Systems etc were once largely applied to the generation of natural objects like vegetation and textures. Only recently the researchers have turned their attention to their application in the context of man made phenomena especially in the application of recreating a city. In the recent years, many research papers and stand alone applications have been developed to generate a 3D city.

Yoah I H Parish and Pascal Muller of Switzerland came out with a stand alone application called “CityEngine” which they presented at ACM SIGGRAPH 2001, paper titled Procedural Modelling of cities. Their application is capable of creating an

urban environment from scratch, based on a hierarchical set of comprehensible rules that can be extended on the basis of user needs. [1]

Their system makes use of various image maps like elevation map, land/water/ vegetation map, population density map, zone map, street map, height map, etc to create a city. The system also invokes two different types of L-Systems, one to create streets and the other one to create buildings. User can generate different types of streets and buildings by manipulating the rules of the L-System which governs the production of streets and buildings.

Sun and Baciu proposed an alternative method to create virtual city in their paper titled "Template based generation of road network for city modelling" in 2002. Their proposed system uses simple templates and a population adaptive template to create a virtual city. Like the previous application, the system requires a lot of inputs in the form of 2D image maps. A color image map which contains geographical information (land/water/vegetation) of a place, an elevation map which contains the height (altitude) information of a place and the population density map of a place are the must needed inputs for the system. This system concentrates only on the generation of road network of a city. Although the output of this system reflects the patterns found in cities, it lacks the complexity and the scale of a real city road network. [2]

Watson et al applied an agent based technique to generate the city in their application titled "CityBuilder". This system is built on the NetLogo™ platform which is a multi agent programmable modelling environment based on the Logo programming language. It is designed to provide users with a platform to explore the emergent behaviour. The CityBuilder system not only models the road network and the street pattern but also simulates the growth and development of them over a time period. [3]

Kelly and McCabe presented a paper titled "CityGen: An interactive system for procedural city generation" in the *Fifth International Conference on Game Design and Technology* in 2006. CityGen is an interactive application that provides a complete integrated workspace for city generation and divides the whole city generation process into three stages. They are Primary Road Generation, Secondary Road Generation and Building Generation. To create primary roads, they use a sampling algorithm for computing road trajectories that follow underlying terrain in a

natural and convincing way and the secondary roads are generated by a subdivision algorithm. The buildings are generated using “string grammars”. [4]

3.0 Technical Background:

As mentioned above, techniques like Fractals and L-Systems form the basis of almost all the procedural city generators, although there are some exceptions.

3.1 L-Systems in City Generation:

Fractal, the word was coined by a mathematician Benoît Mandelbrot in 1975 and was derived from the Latin *fractus* meaning "broken" or "fractured." A **fractal** is generally "a rough or fragmented geometric shape that can be split into parts, each of which is (at least approximately) a reduced-size copy of the whole," a property called self-similarity. [5]

Fractals are considered to be infinitely complex since they appear similar at all level of magnification. They are used to generate mountain, clouds, blood vessels, snowflakes, etc. Being a procedural technique, a fractal shape is generated by recursive algorithm. The number of recursion defines the detail of the fractal shape. These are however limited to self similar structures and are often superseded by a more flexible method called L-Systems.

An **L-system** or **Lindenmayer system** is a **parallel string rewriting system**, namely a variant of a formal grammar, most famously used to model the growth processes of plant development, but also able to model the morphology of a variety of organisms. L-systems can also be used to generate self-similar fractals such as iterated function systems. L-systems were introduced and developed in 1968 by the Hungarian theoretical biologist and botanist from the University of Utrecht, Aristid Lindenmayer (1925–1989). [6]

In general, rewriting is a technique for defining complex object by successively replacing parts of a simple object using a set of rewriting rules or productions. An L-system is based on a set of production rules. Each string consists of a number of different modules which are interpreted as commands. The parameters for these commands are stored within the modules. The components of an L-System are

Variables – V – set of strings or symbols that can be replaced in each production based on the rule

Constants – S – set of strings or symbols that remain constant throughout the production.

Axiom - ώ - set of variables and constants that represent the initial state of an L-System.

Rules – P – set of rules that explains and governs the way the variables can be replaced with the combination of constants and other variables. [7]

Examples of a simple L-System:

These examples where taken from the book, "Algorithmic beauty of plants" written by

A system G is defined by four components as explained in the previous passage and therefore G can be written as a set of four components.

G = {V, S, ώ, P}

Where, V = {a, b}

ώ = a

P1 = a → ab

P2 = b → ba

Initial Generation (n = 0): a

Next Generation (n = 1): ab

(n = 2): abba

(n = 3): abbabaab

This example shows how the length of string grows in each generation based on the production rules. The above example explained L-System in its theoretical form. To see it visually, here comes an example.

G = {V, S, ώ, P}

Where, V = {X, F}

S = {+, -, [,]}

ώ = X

P : P1 = X → F-[[X]+X]+F[+FX]-X

P2 = F → FF

Other details are angle is 22.5 degrees and number of generation is 5.

F means draw Forward

- means turn left by the given angle (here 22.5 degrees)

+ means turn right by the given angle (here 22.5 degrees)

[means store the current position and angle

] restore the position and angle with the corresponding values

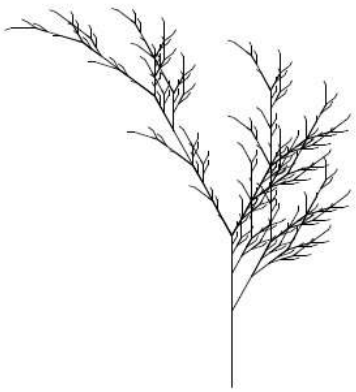


Fig 1: L-System plant

For quite a long time, this string rewriting technique was employed only for the generation of natural phenomena like plants, micro-organisms, fractals, etc. It was Parish and Muller of Switzerland who came out with the most significant and innovative method of using these formal grammars for generating the man made phenomena like roads, buildings, etc. Their system invokes two different types of L-System namely self sensitive L-System and stochastic parametric L-System.

Self sensitive L-System is an extended form of L-system which "CityEngine", the stand alone application developed by Parish and Muller uses to generate the road and street network. This system takes the existing shape into account before every generation and so they can be grouped under context sensitive L-System. Inputs for

this L-System are a set of 2D image maps. Geographical information on the elevation of the land, water boundaries are obtained from the elevation and land/water/vegetation map and the socio statistical data like population density of the region, street pattern type of region, etc are obtained from socio statistical maps like population density map, street network map, etc.

Once the required data are fed to the system, road generation application starts developing the network. Road generation is accomplished through the use of two rule sets. They are Global goals and local Constraints. Initial tentative road segments are plotted by the rule set defined in the Global goals which are then refined by the local constraints that reflects the practical constraints of the real world. User can manage the development of road segments by manipulating the rule sets and also specifying extra parameters like smoothening angle of road edges, road width, etc.

The land area is subdivided after the generation of road network to form the allotments where the buildings will be generated by a type of L-System called stochastic parametric L-System. For every allotment one building is generated. They are generated by manipulating the arbitrary ground plan. The modules of the L-system consist of transformation modules (*scale* and *move*), an extrusion module, branching and termination modules, and geometric templates for roofs, antennae, etc. The final shape of the building is determined by its ground plan which is transformed by interpreting the output of the L-system. The output of the L-system is fed to another parser, which translates the resulting string into geometry readable by the visualization systems.

Using extended L-System complex and detailed city can be developed. But the disadvantage in this system is the iterative nature of L-System. As the iteration increases, the number of variables that has to be replaced and the complexity of the system increase exponentially. This makes the computation really very expensive. Every time a new constraint is added, many rules have to be rewritten. This makes extensibility a difficult task. User is also expected to have high level of expertise in framing L-System grammars to build the city which in turn reduces the accessibility of this system.

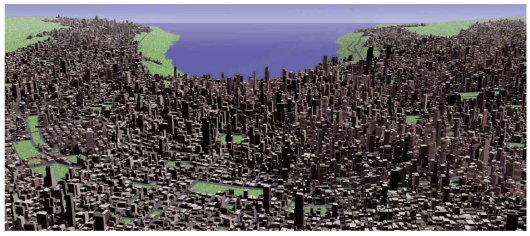


Fig 2: City created using CityEngine

3.2 Alternative approach to City Generation:

This project doesn't use L-Systems to generate the city, instead, uses a least computationally expensive process called sampling technique for road network generation and a voronoi pattern generator for the street network generation. The sampling technique employed in the generation of road network is adopted from a research paper titled "Citygen: An interactive system for procedural city generation" , authored by Kelly and McCabe presented in the *Fifth International Conference on Game Design and Technology, 2007.* Voronoi pattern is used to generate the street network for the city and a simple subdivision algorithm is implemented to create the plots where the buildings will be placed.

3.2.1 Sampling Technique:

The sampling technique used in this project for the generation of road network is not an exact implementation from the source research paper. It has been modified to suit the needs of this project. For instance, the sampling technique given in the research paper is bidirectional that is, the sample points are plotted simultaneously from the source and the destination point and terminates in the middle. But the algorithm implemented in this project is unidirectional. A unidirectional approach is used to reduce the computation and keep the workflow simple.

A road is generated starting from a source point and sampling a set of points at regular intervals to define a path to the destination. The number of samples (n) to be plotted between the source and the destination and the maximum deviation angle (θ) of a sample point that can be plotted from the source point are specified by the user.

Say, Number of Samples = n

Distance between the source and the destination point = D

Distance between the two sample points (d) = D / n

Each sample point travels a distance *d* and a random angle is chosen based on the user input and a quaternion is formed to calculate the position of the deviated point. A quaternion is formed based on the axis which is perpendicular to the source point and the angle chosen randomly in the range specified by the user and multiplied with the current sample position which is now distance "d" away from the source/another sample point and collinear with the source and the destination point.
Once the number of samples as mentioned by the user is plotted, the segment points are tested for any intersection with the illegal areas like ponds and other water bodies and then fed to a NURBS curve generating function which generates a curve between the source and the destination point taking these sample points as control points.

The checking algorithm to find whether a sample point is within an illegal area and moving it away from that area if it is within that area are explained in detail in the later part of this thesis.

3.2.2 Voronoi Pattern:

Voronoi Diagrams were demonstrated as a method of procedural generation by S.Worley in his paper titled "A cellular texture basis function", in which he detailed an algorithm that partitions the space into a random array of cells creating cellular looking cells. This technique is widely used in the procedural generation of textures like tree bark, skin, cobblestone, sun baked mud, etc.

Voronoi diagram can be defined (as in mathworld.wolfram.com) as the process of partitioning of a plane with *n* points into convex polygons such that each polygon contains exactly one generating point and every point in a given polygon is closer to its generating point than to any other. A Voronoi diagram is sometimes also known as a Dirichlet tessellation. The cells are called Dirichlet regions, Thiessen polytopes, or Voronoi polygons.

In procedural city generating context, the voronoi diagram is used to create cellular patterns which appear similar to the street patterns of a city. The idea of using voronoi pattern in a city generator was first proposed in a research paper titled "Template based generation of road network for city modelling" authored by Sun, Baciu et al in 2002. According to their method, the road network for a city can be developed from the voronoi pattern. User feeds a population density map of a city and their system is capable of extracting the density points from the map which is then fed

into a voronoi pattern generator. The generator uses these points as the attractor point and attracts all the points closer to it. The edges or cell boundaries of the resulting voronoi diagram are used to create the interconnected road network.

A similar technique is used in this project to generate the secondary roads and the street patterns. Based on the distribution of points fed to the voronoi generator, a variety of street patterns like the radial pattern one can see in the cities like Paris and Rome, organic pattern one can see in cities like Delhi, Chennai and raster pattern as in Newyork and manhattan can be generated.

3.2.3 Subdivision Technique:

Once the road network and the street patterns are generated for the city, the next step is the plot creation to place the buildings. Each cell formed as a result of voronoi pattern generator is subdivided to create plots. The plots are the bases on which the buildings will be placed latter. The level of subdivision of each cell can be specified by the user. By specifying the level of subdivision, the size of the plot and the density of the plots in a street vary which differentiates the various zones of a city. If the plots are small and closely packed, the street appears like a residential area. If the plots are big and closely packed, the street appears like a commercial area and an industrial area can be generated by having bigger plots than the commercial area and maintaining sparse density distribution of plots.

4.0 Procedural City Generator:

The procedural city generator developed in this project is an accessible and interactive Houdini tool comprising a set of digital assets to automatically generate a realistic and detailed city layout. Creating a city using this tool in Houdini is a four step process. The city generation starts of with generating the terrain of the city followed by the primary or highway road network generation. The third step is the street and plot generation and the final step being the distribution of the buildings onto the plots generated.

The rest of the thesis is going to be a detailed explanation of how the tool accomplishes each step to successfully generate a realistic and detailed city layout which will be suitable for use in real time rendering. The tool makes use of the

powerful node based Houdini environment along with powerful API (HOM – Houdini Object Model) to generate the procedural city.

This procedural city generating tool is a set of 8 digital assets (HDA) and 6 Python SOP nodes.

4.1 Assets and nodes:

As mentioned earlier, the city generation is a four step process. The custom developed assets and python nodes used in each step of the city generating process is listed in this passage.

Step 1: Terrain Generation

Asset → IPK_Terrain, IPK_Contour_Heightmap

Python nodes → IPK_Contour, Dist

Step 2: Road Network Generation

Asset → IPK_Road, IPK_RoadSegment

Python nodes → IPK_Roadsampler

Step 3: Street and Plot Generation:

Asset → IPK_Streetgen, IPK_Orgpattern, IPK_Radpattern

Step 4: Building Distribution:

Asset → IPK_Buildnetwork

Python node → IPK_Bgeodistributor, IPK_Scaler, newnode.

4.2 Terrain Generation:

Terrain generation is the first step in the process of generating a 3D city. The input for the terrain generator can either be a contour map or a greyscale height map which is widely known as elevation map.

4.2.1 Grey Scale height map as input:

If the input is a greyscale height map, create an instance of IPK_Terrain in the Sop level inside a geo node. The asset has the option to set the size of the city that is going to be generated. On setting the proper city size, a 2D greyscale height map is loaded in the parameters tab where it is asked for. On clicking the "Apply Height map" option, the grid will be converted into a terrain.

The actual operation that is being performed inside the asset which converts the grid into a terrain based on a 2D image map is transferring the grey value in the image map as a translating value in the positive Y axis to each point on the grid. Houdini has a built-in function called "pic( )" which does this. The white pixels return a value of 1 while the black pixel returns 0. The intermediate grey values return a float value between 0 and 1. The values can be multiplied with a height factor to increase the height of the terrain.

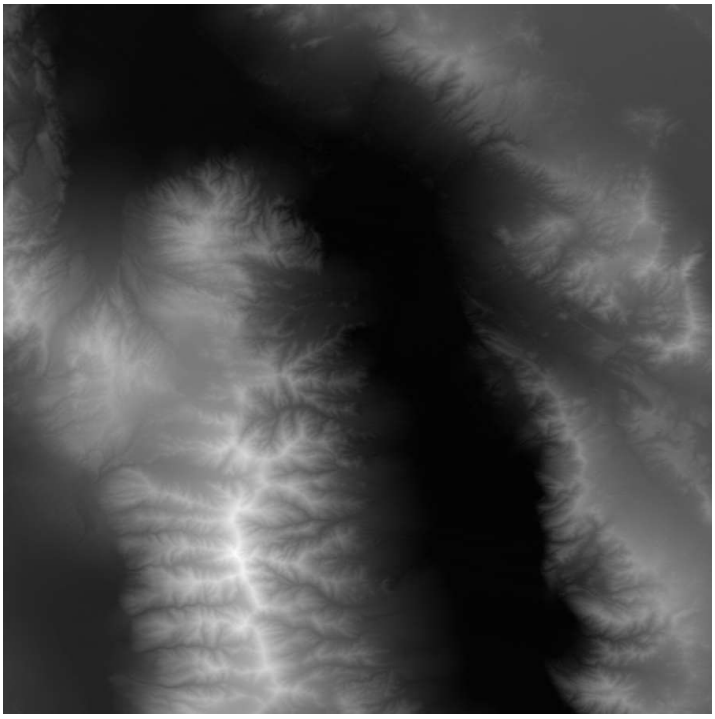


Fig 3: Grey Scale map

4.2.2 Contour map as input:

If the input is a contour map, then it has to be converted into a grey scale height map before feeding that to IPK_Terrain to generate the terrain. Converting contour map into a grey scale height map is a very easy process in this procedural city generator.

Contours are one of several common methods used to denote elevation or altitude and depth on maps. From these contours, a sense of the general terrain can be determined. In cartography, a contour line (often just called a "contour") joins points of equal elevation (height) above a given level, such as mean sea level. A contour map is a map illustrated with contour lines, for example a topographic map, which thus shows valleys and hills, and the steepness of slopes. The contour interval of a contour map is the difference in elevation between successive contour lines.

Feed the contour map as an input to IPK_Contour_Heightmap asset. The technique behind converting a contour map to a height map is nothing but converting each contour line in the map as a primitive and filling it with a grey value based on the altitude at that point. Trace sop is used to read the image information in the COP-Level and convert that into a set of primitives in sop level.

Fig 4: Contour map

4.2.3 Reason to create “Dist” python sop:

As mentioned above, the map is inverted in the COP-Level and it is traced using a trace sop in Houdini and displayed in the viewport. The way the trace sop converts the image into primitives was not as expected. Each contour line in the map was converted into two primitives while tracing. As a result, the traced geometry had a lot of primitives than the number of contour lines.

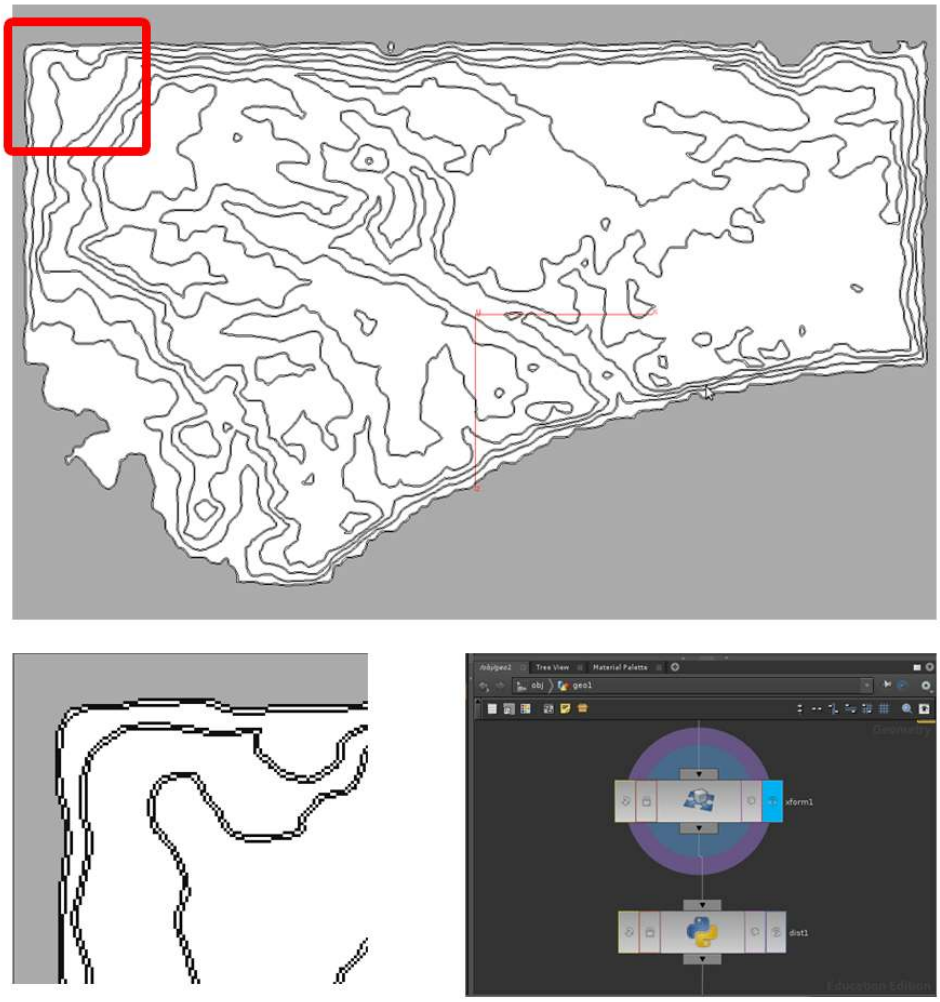


Fig 5: Trace contour geometry without Dist node

To overcome this, the traced geometry is fed into a python sop called Dist. The python sop clears up the traced geometry and returns it with the same number of primitives as the number of contour lines. Python sop achieves this by calculating the distance between a constant point outside the geometry and each primitive in the geometry. HOM has a built-in function in hou module which does this. The function

is hou.Prim. nearestToPosition(self, pos3). This function returns the distance between the primitive and a point. For each primitive in the geometry, the distance is calculated and since the two primitives formed for each contour line are very close to each other, the distance calculated will be almost same and one primitive out of the two which shares almost the same distance is deleted.

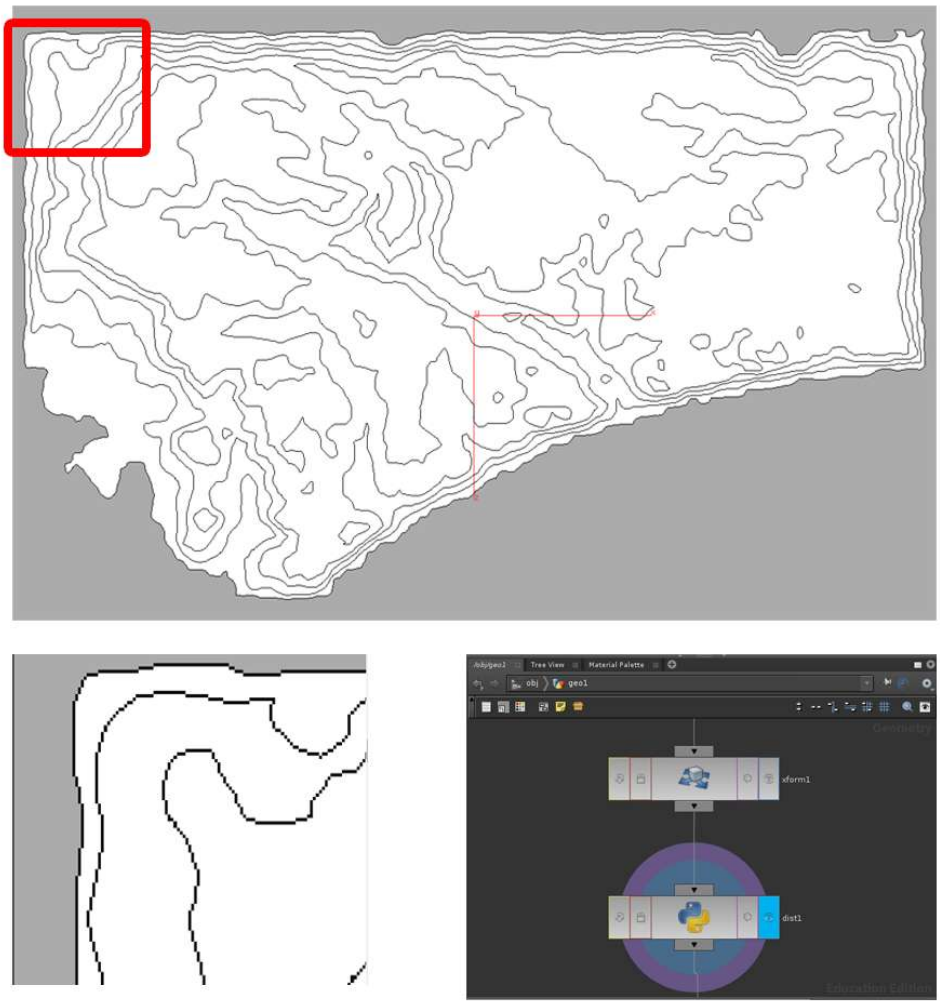


Fig 6: Trace contour geometry with Dist node

4.2.4 Creating Grey scale map:

Once the traced geometry is cleaned up, the user has to enter the details of the contour map to convert it into a grey scale map. The details to be fed are contour interval, altitude range (lowest altitude and the highest altitude) and the number of index contours in the map. Index contours are the contour lines that are found in the map along with a numerical value (it's altitude) labelled on top of it. Following the

number of index contours, the primitive number of index contours, the altitude of it and the group of contours whose altitude is dependent on this index contour are entered.

The process of converting the contour lines into a grey scale starts once the required inputs are entered. The primitive with the lowest altitude is filled with black color and the primitive with the highest altitude is filled with white color. The primitives with the intermediate altitude are filled with an interpolated grey value.
The user can render the scene now and save the grey scale map as an image which can then be fed to IPK_Terrain asset to generate the terrain.

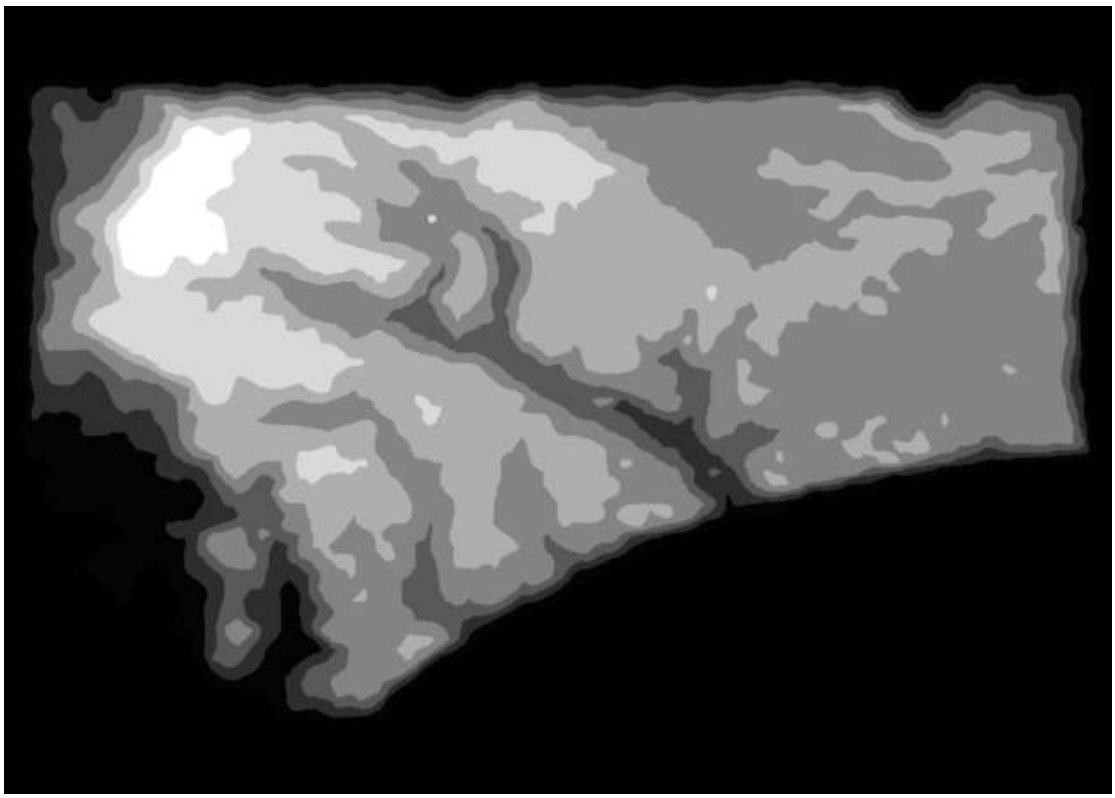


Fig 7: Grey scale Map created from a contour map using the asset

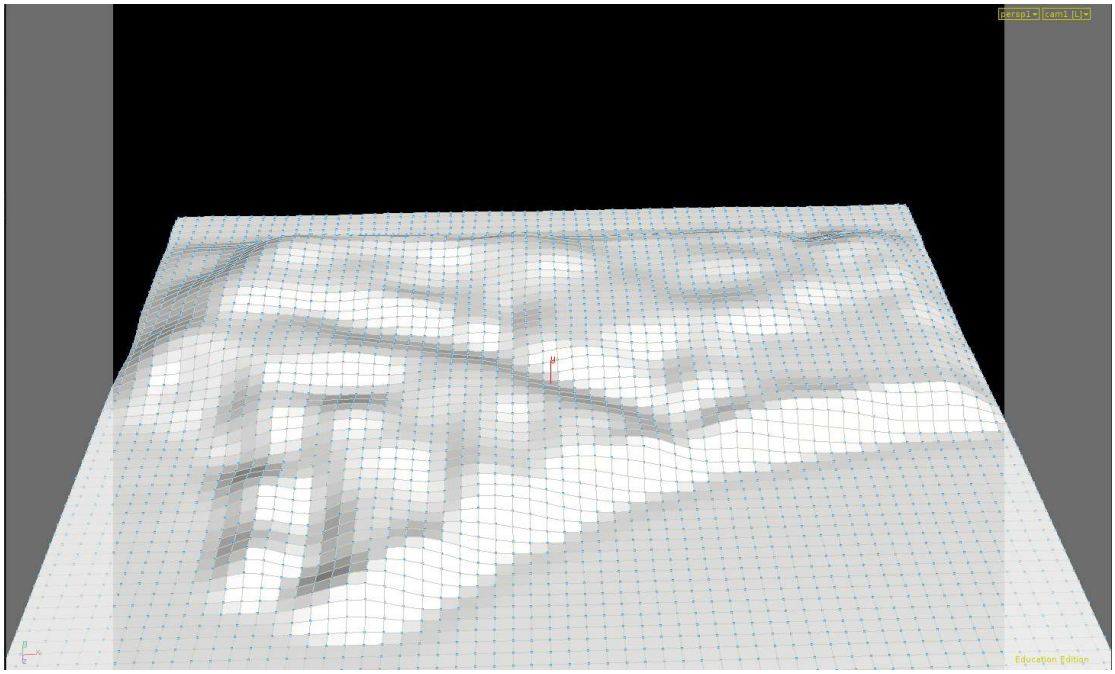


Fig 8: Terrain generated from the IPK_Terrain asset

4.2.5 Feeding the details of the water bodies on terrain:

IPK_Terrain also takes in a water map as input to store the details of the water bodies in a city which will be used in the road network generation. This step is optional. If the user wants water bodies in the city he is developing , he can feed a black and white water map where the white areas denote the water bodies and the information is stored in the asset.

Fig 9: Pond Map

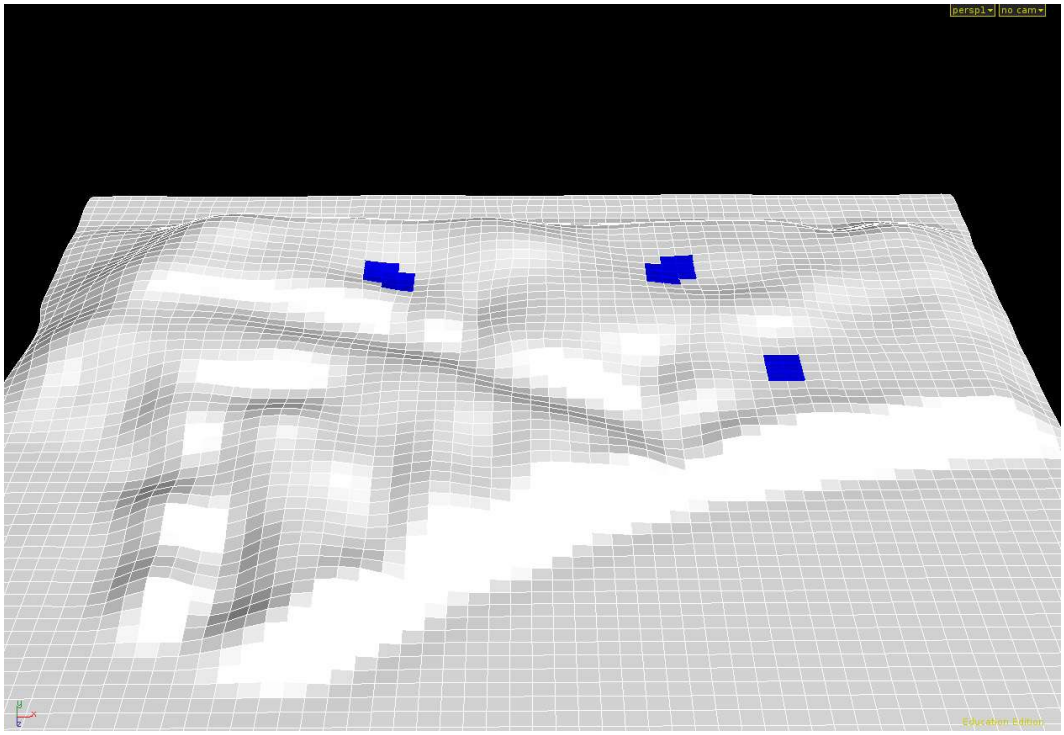


Fig 10: Terrain with areas allocated for ponds. (Primitives in blue)

4.3 Road Network Generation:

Terrain generation is followed by the process of generating the highways or the primary road network of a city. The input for generating the highways is the "Junction points". Junction points are nothing but a group of points in a city which acts as a source and destination of the highways. These junction points are fed to the road network generator in a form of map. The map is not a complex road map of a city. Just black dots on a white background. Black dots represent the junction points of the city.

4.3.1 Generation Process:

Create an instance of IPK_Road asset inside the same geo node where the terrain asset has been instanced. Connect the input of the IPK_Road to the output of the terrain asset. Feed the junction map to the road asset. The asset has a python node called "IPK_Roadsampler" in it which generates the highway network of the city.

As did in the terrain generation, the black dots in the image are traced as primitives using a trace sop. For each primitive a point is scattered and now the junction points are available for highway generation.

4.3.2 IPK_Roadsampler:

The steps performed by this python sop for generating the highways or the primary road network are as follows:

- Selection of the points between which the roads are going to be generated.
- Computing samples using sampling techniques.
- Checking the samples for their presence in the illegal areas like water bodies and changing its position if they are in such areas.
- Feeding the computed and checked points to a NURBS curve generating function which generates smooth NURBS curves which are nothing but the highways.

All the calculations done in generating the road network are 2D calculations without taking height of the terrain into consideration. Once the NURBS curves are generated, Houdini has a powerful node called “lattice” which can be used to ray the curves onto the terrain. This reduces a lot of computations.

4.3.3 Selection of Road points:

The user specifies the number of branches each junction point can have. Based on that, each junction point is assigned a random number as the number of branches within the range specified by the user. The junction points are stored in a list. First junction point is taken and it is looped through the rest of the points down the list to find the possible destination points between which the road segments will be generated. The loop is executed until the number of branches assigned for that point is reached.

The selection process involves checking the source and destination point (source point is the junction point currently being looped through every other point down the list and the destination point is the current point with which the source point is being checked with) for intersection with any other approved road segments. If intersection is true, the proposed road segment will not be approved and the source point is checked with the next point. If the proposed road segment doesn’t intersect with the approved road segment, the angle between the approved road segments is checked. The angle should be above a certain threshold value for the proposed road to become an approved road. Approved road points are stored in a list.

This process is looped in for all the junction points in the list and a set of approved source and destination points are stored in a list.

4.3.4 Computing the sample points:

Each element in the approved road points has two junction points in it. One is the source point and the other is the destination point. These points are fed to the sampling technique as explained in the earlier part of this thesis and the samples are calculated. User can specify the number of samples that has to be calculated between the source and the destination point and also the angle of deviation of a sample point form the source point.

4.3.5 Checking with the water bodies:

This is an additional preference the user can turn on if he had water bodies in the city. On turning this option on, the computed sample points undergo a series of checks before they are fed into a NURBS curve generator. The details about the water bodies are obtained form the terrain node.

The steps involved in checking are, each computed sample in each road segment is checked if it lies inside the illegal area. Illegal areas are nothing but the primitive numbers on the terrain which has water bodies in it.

If a computed sample is found to be within the bounds of the illegal area, the exact primitive in which it lies is found out. The point is then pushed out of the primitive in the direction perpendicular to the intersecting edge of the primitive with the line drawn between the computed sample and its source point. Pushing it either on the left or right of the primitive is depend upon the distance between the point and the left and the right bounding edges of the primitive. It pushes to the side whose distance is lesser. The moved point now lies outside the primitive and it is checked again until the point is found to be entirely out of the illegal area.

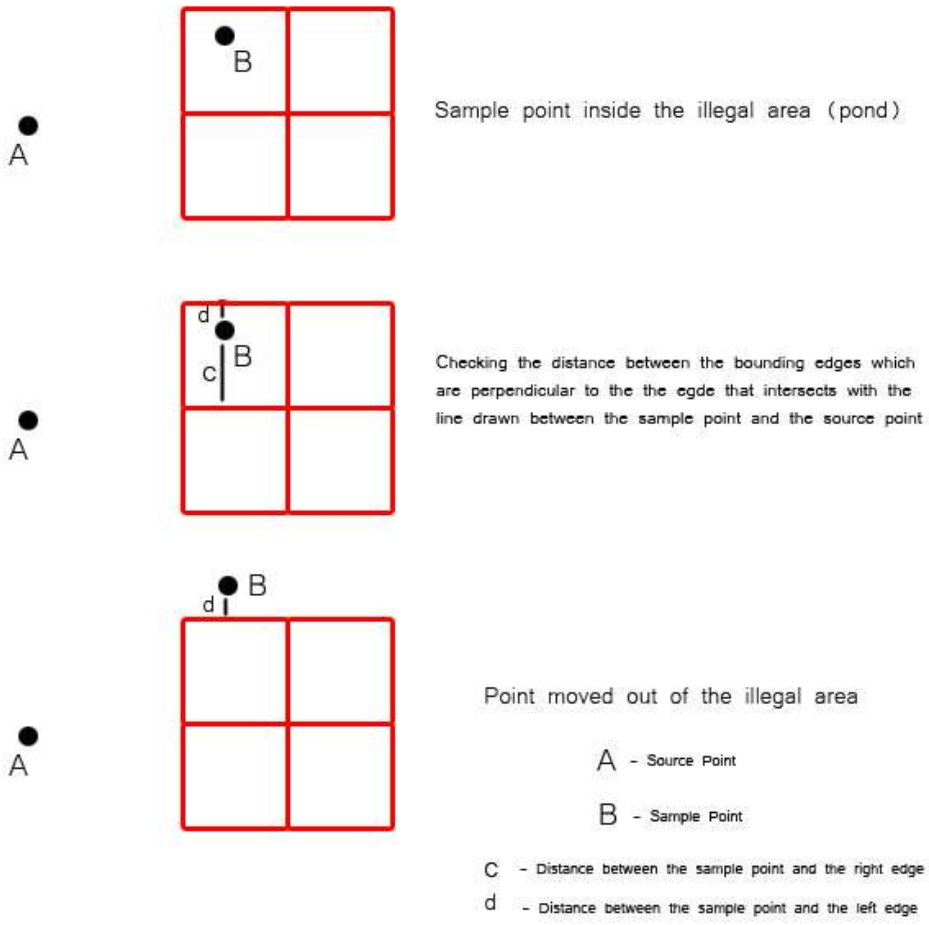


Fig 11: Diagrammatic representation of pushing the sampled point out of the illegal area

Once the point is pushed away from the illegal area, the new position of the point is updated in the list of approved road segments list. "Approved road segments" is a list which has the entire computed sample point's position along with its source and the destination point. The next check after this is to find out whether the road segment between the updated point and its previous point in the list is intersecting or crossing the illegal area. If it is found to be intersecting, the intersecting points are found out and they are moved out of the illegal area for a definite distance along the direction vector it makes with the centre point of the illegal area.

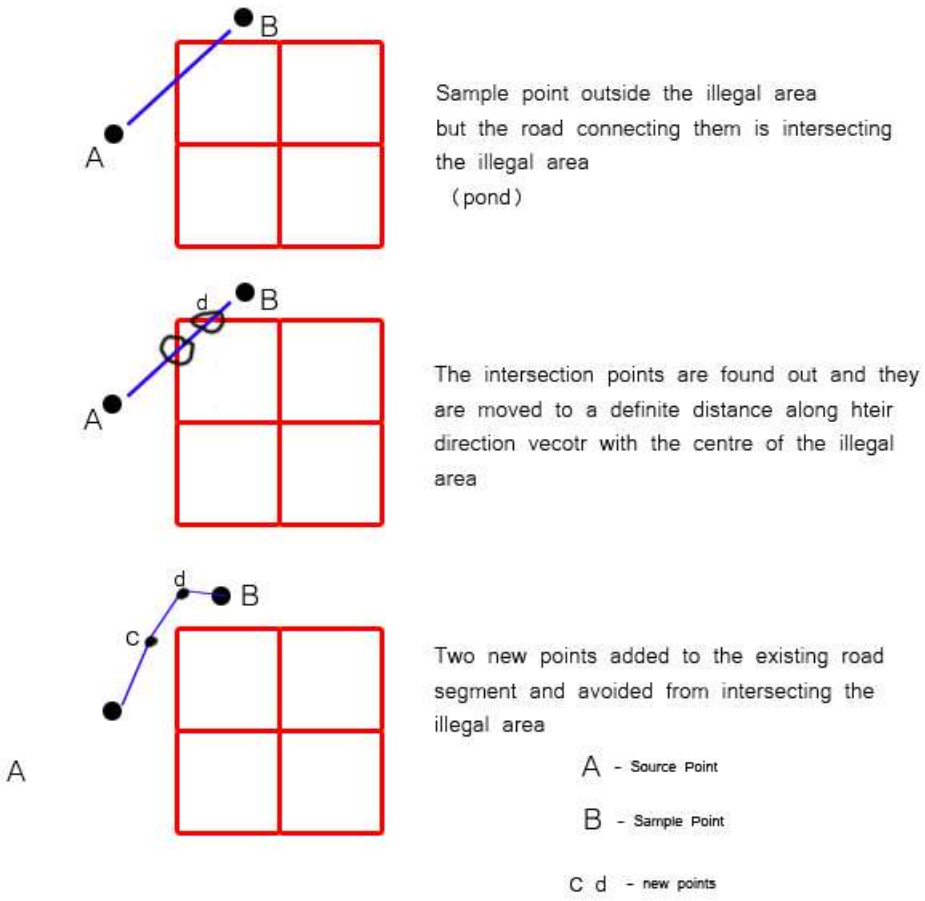


Fig 12: Diagrammatic representation of avoiding a road from intersecting the illegal area

The intersection between the lines is found out using the following algorithm adopted from the book "Real Time Collision Detection" written by Christer Ericsson. [8] The algorithm is that if the line segments AB and CD are intersecting, then the signed areas of the triangles ABC and ABD will be opposite. The signed area is positive if the triangle winds counter clockwise, negative if it winds in clockwise and zero if the triangle is degenerate (collinear or coincident points). The formula to calculate the signed area is as follows.

$$\text{Signed area} = (A.x - C.x) * (B.y - C.y) - (A.y - C.y) * (B.x - C.x)$$ [8]

This algorithm, in the case of segments intersecting each other, can be used to calculate the intersecting points.

If the moved out points still intersects with the illegal areas, to avoid the expensive computation that involves in recursive checking of these points, a bridge is proposed in that site. Now the approved road segments will be appended with the bridge points in it.

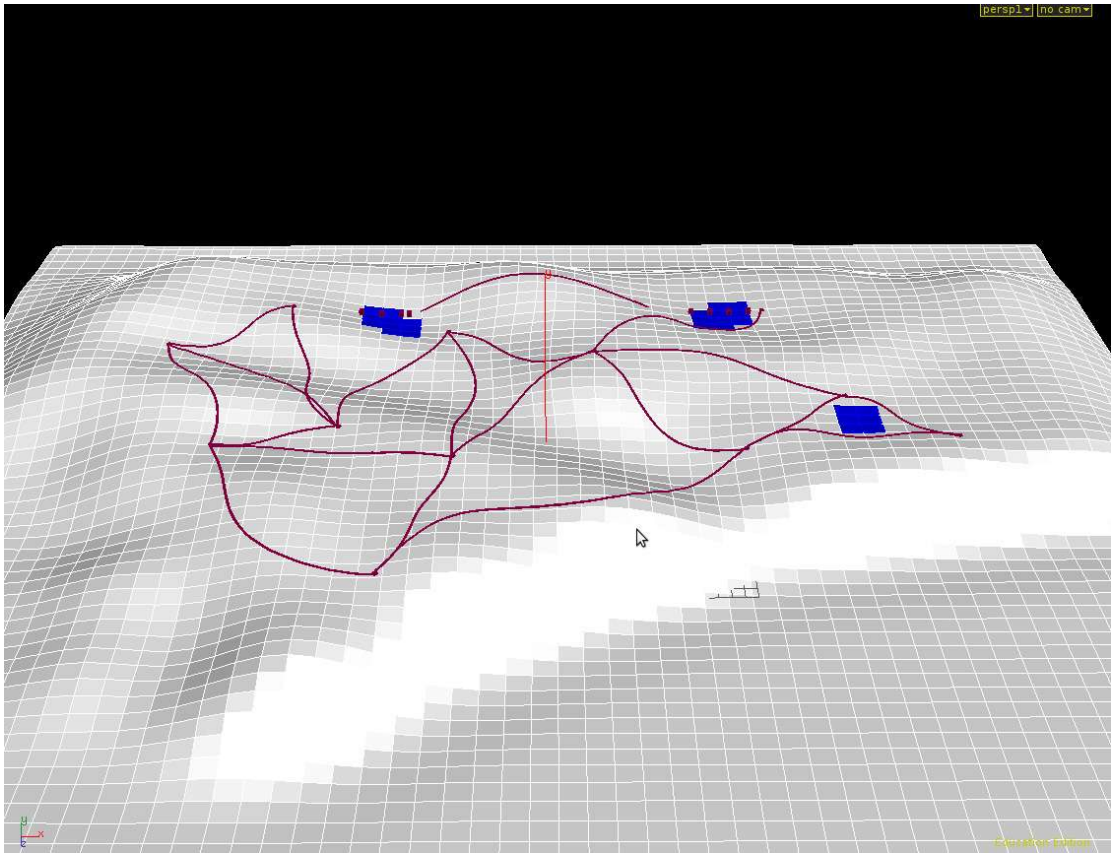


Fig 13: Roads on the terrain with the proposed bridges.

4.3.6 NURBS Curve generation:

After the checking for intersection with the illegal areas is done, approved road segments list is differentiated into two lists. One list stores the road points and the other list stores the bridge points. They are differentiated based on the bool value of each point. If the point is a bridge point, the bool value will be one and if it is zero then it is a road point.

The road segments are then fed into a NURBS Curve generation function which generates the road curves.

This is the operation that is performed in the IPK_Roadsampler python node. The junction points are fed to the python node and it returns the nurbs curves which are then sweeped with a line to form a visually appealing road.

The road network generated by the IPK_Roadsampler can be changed and customized. It can be changed by changing the seed value and other parameters like number of samples, deviation angle, etc. New roads can be added by the point numbers within which the roads are to be generated. Similarly existing road segments can be deleted by specifying the source and the destination point number.

The road network is latticed with the terrain using a lattice node which rays the node exactly on top of the terrain at the end.

4.3.7 IPK_RoadSeg:

IPK_Roadseg is an asset which can be used to create a road segment between any two points and they need not be in the list of junction points. User selects the source position and the destination using a mouse and the road segment will be laid and latticed on top of the terrain. It also uses the exact algorithm that is used by IPK_Road asset.

4.4 Street and Plot Generation:

The third step in the process of generating a procedural 3D city after the terrain and the road network generation is the street and the plot generation. In this step the street network and the plots on which the buildings will be distributed in the later stage are generated.

As mentioned in the earlier part of this thesis, the streets are generated by voronoi pattern and the plots are generated by a subdivision algorithm. Both street and and plots are generated by a single asset namely IPK_Streetgen. It makes use of two other assets, IPK_Orgpattern (which produces organic street pattern one can find in the cities like Newyork, New Delhi,etc) and IPK_Radpattern (produces radial streets as in Paris and Rome). Both these assets have the same workflow except in the way the points are distributed which is fed into voronoi pattern generator. In radial pattern generator, the points are distributed in a circular fashion. The voronoi pattern generator used in the asset to generate voronoi pattern is downloaded from a website [9].

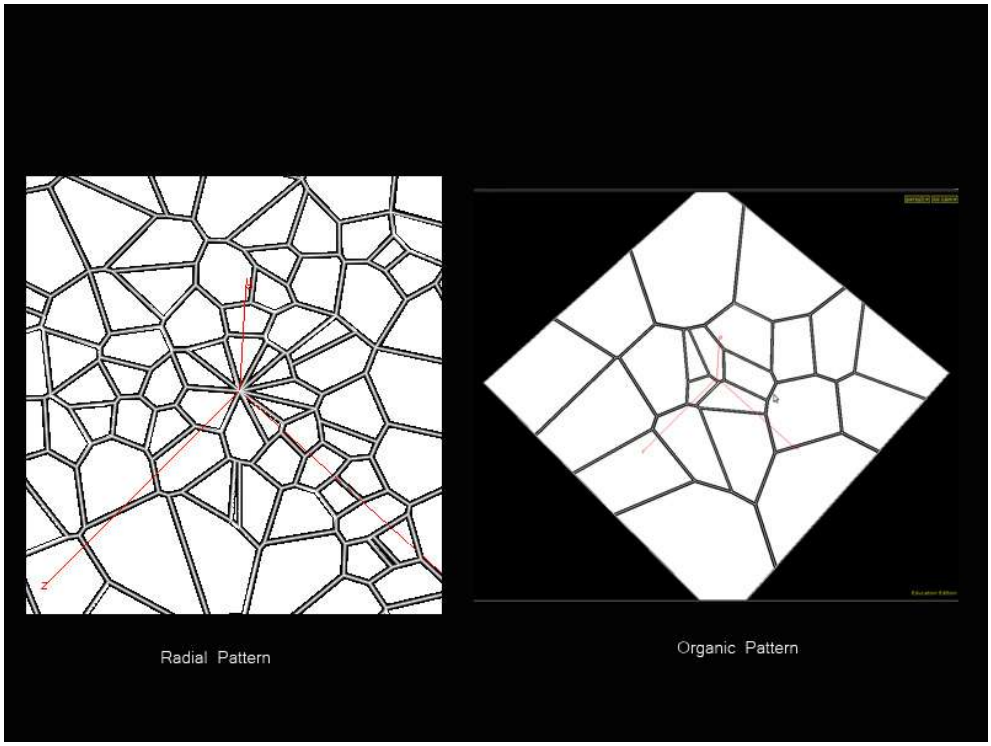


Fig 14: Street Patterns

User groups the region on the terrain based on the type of streets. Each group is then connected to the input of a IPK_Streetgen asset. Each group should have separate IPK_Streetgen asset. Inside the asset, the group is converted into a 2D plane on which the street pattern and subdivision occurs. The 2D plane is latticed with the terrain at the end.

Streets are generated and each cell of the street formed by the voronoi pattern is fed into a subdivision technique where the cell is cookied (Boolean operation) with a box of type mesh. The density of the plots and the size of the plots are directly related to the density and the size of the mesh. User can specify the density and the size of the box thereby affecting the density and the size of the plots.

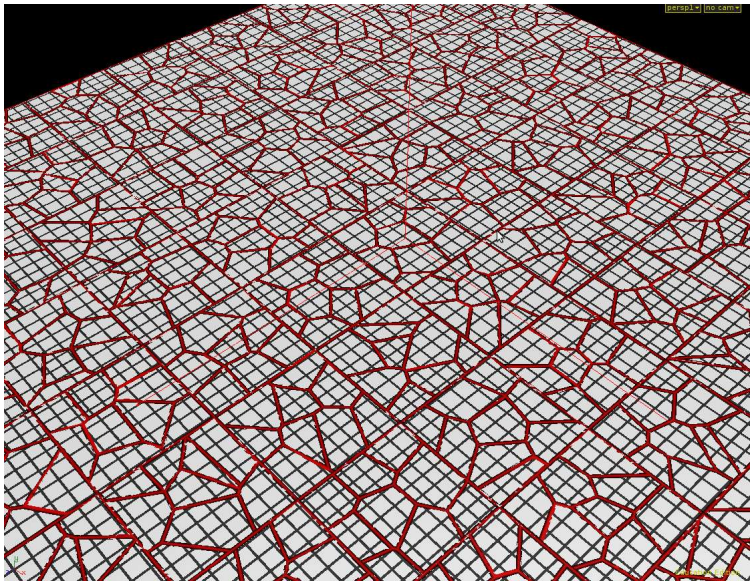


Fig 15: Subdivided Street Pattern

4.5 Building Distribution:

The final step in the generation of the city is the distribution of buildings and other geometries like vegetation, street lamps, etc onto the plots generated in the previous step.

To avoid Houdini from loading and saving the high detailed extremely large geometry files at every frame which may be costly in both memory and time, mantra can be instructed to load the geometry from disk instead of having it in the ifd files. This can be done using Mantra: Delayed load shader. Mantra still has to read the geometry, but instead of having to process the embedded geometry it can load the geometry directly from the disk.

This procedural city generator uses Mantra: Delayed load to distribute the geometries in the plots at the render time. In each plot, a point is created and for each point, geometry is instanced. The point stores the information about the geometry it should load and the scale of the geometry to fit within the plot.

This process is done with the help of an asset and three python nodes. They are IPK_BuildNetwork, IPK_Instancer, IPK_BgeoDistributor and IPK_scaler.

4.5.1 IPK_Instancer:

IPK_Instancer is a python node which sets up the entire geometry distribution network. This python node is an object level operator. Once the streets are created, this instancer is created for each street network. The user uploads the geometries that

should be instanced in that particular street. On clicking the “Create Node” button, the geo nodes for each uploaded geometry is created along with their respective delayed loads in the shop level. All the references are done by the python node itself. Similarly clicking the “Create Asset” after giving the path, will create the IPK_Buildnetwork node in the given path. The path is the path where the street network is. User can feed the output of the street network to the asset which will do the geometry distribution which is explained later in this part.

Once distribution is done, user can see the street filled with bounding boxes instead of geometries and this is to reduce the memory usage and increase the speed of the process. Then press the “Create Instance” button to create the instances which automatically creates an object merge inside the instance node that it creates and references the point group of the build network. Now on clicking the render button, user can see the streets with actual building.

4.5.2 IPK_BuildNetwork:

This is a digital asset which takes in each plot of the incoming geometry, scatters a point at the centroid of it and assigns attributes to that point which is later used by the delayed load to instance a geometry on to it at the render time. This asset has two python nodes within which actually do all the functions. They are IPK_BgeoDistributor and IPK_Scaler.

4.5.3 IPK_BgeoDistributor:

This is a python node which selects a geometry from the list of geometries user has given based on the probability set by the user. On selecting the geometry, it creates a detail attribute which stores the path of the selected geometry. It is then fed to a file node and a bounding box for the geometry is created. The bounding box is copied onto the point and the index number of the geometry, whose bounding box is on the point, is assigned as a point attribute to the point which will be made us by the delayed load at the time of instancing. The list of geometry loaded by the user is referenced to both the buildnetwork asset and the instancer. This avoids loading the entire path of the geometry as an attribute to the point.

The size of the incoming plot is noted and the bounding box is fit to the size of the plot. The scale value is assigned as a point attribute to the point which will be used by the instancer node at the time of instancing.

This process of reducing the size based on the plot's size will work as expected only in the case of square. To make the bounding box always fit within the plot, a "rescaling algorithm" is followed and it is done by IPK_scaler node. Rescaling algorithm is called only when the plot is not a square.

4.5.4 IPK_Scaler:

The base face of the bounding box is taken and checked for the intersection with any of the edges of the plot. If an intersection is found, then the distance between the centroid of the plot and the intersection point and the distance between the centroid and the vertex of the base bounding box which is outside the plot are calculated. Then the smallest among them is divided by the largest one and the result is the scaling factor. This scaling factor overwrites the scale value assigned to the point at the initial stage.

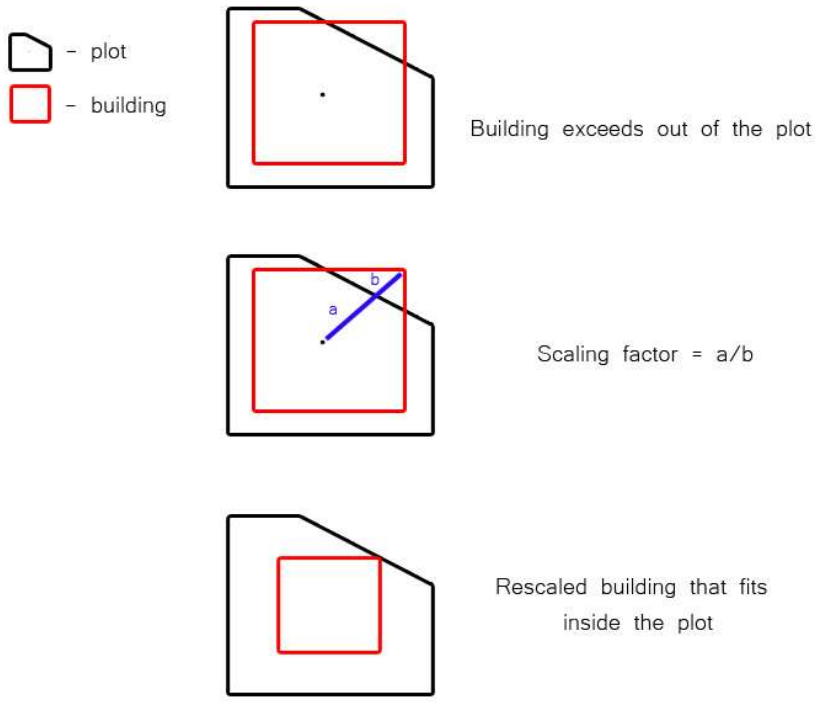


Fig 16: Rescaling the building to fit it within the plot

This algorithm works well as for as the incoming plot is a convex polygon. To make it work even on the concave polygon, linear programming has to be done. So, care has been taken in the plot generation step to avoid concave polygons. The disadvantage with this algorithm is that, sometimes it scales down the geometry really small. In that case, the geometry is replaced by trees and the index number and the

scale value assigned to the point is changed to the new value. The size of the geometry is checked by measuring the area of the base of the bounding box of the geometry after scaled down.

At the end of this process, each plot will have a point with the attributes required by the instance node.

This is a short description about how the procedural city generator works. For more information about the each asset and its parameters, kindly refer the help page of the assets.

Fig 17: City generated using the asset

Fig 18: City generated using the asset

5.0 Conclusion:

Procedural City Generator created in this project is capable of generating a digital city from scratch. It can generate terrain, road network, propose sites for bridges and street network. It can also distribute the buildings and other geometries which fills the city.

6.0 Problem faced:

A model city was set up using the asset and it failed to render everytime it was rendered in a sequence. The model city had around 8000 geometries in it. When it was rendered as a single frame, it rendered fine. Command line rendering, rendering from ifds, using batch_hrender scripts, etc failed to render the scene. Sometimes it would render a frame and then crash but most often it crashed at the end of the first frame.

The problem was approached with trial and error technique. First the shadows were turned down, then the motion blur and finally reduced the number of geometries to half. But nothing really helped.

I was able to render only after splitting my scene into different components like rendering my terrain and skyscrapers in one pass, rural buildings in a pass,

apartment type of buildings in another pass and finally the dome image and creating ifd files and using batch_hrender script.

7.0 Future Improvements:

- Shape grammars could be incorporated which can generate buildings of its own based on rules given by user which reduces the time spent in modelling the buildings. But user should have a certain level of expertise to frame rules for shape grammars.
- In the viewport, low level buildings can be displayed instead of bounding boxes.
- The road segments can be made editable after their creation. To make it happen, a way should be found out to make the edit node inside the digital asset to be the current node from outside the digital asset.

References:

[1] -- Yoav I H Parish; Pascal Muller; 2001. Procedural Modelling of Cities. In Proceedings of ACM SIGGRAPH 2001 / ACM Transactions on Graphics, ACM Press, 301—308.

[2] -- Sun, J., Yu, X., Baciu, G., and Green, M. 2002. Template-based generation of road networks for virtual city modeling. In Proceedings of the ACM Symposium on Virtual Reality Software and Technology (Hong Kong, China, November 11 - 13, 2002). VRST '02. ACM, New York, NY, 33-40.

[3] -- Kelly, G. and McCabe, H. 2006. A Survey of Procedural Techniques for City Generation. In *ITB Journal*, Issue 14. Available from http://www.gamesitb.com/SurveyProcedural.pdf

[4] -- G. Kelly and H. McCabe. 2007. Citygen: An interactive system for procedural city generation. In *Fifth International Conference on Game Design and Technology*, pages 8—16.

[5] – Anonymous. 2001. Fractals. [online] (Updated 17 August 2009)
Available at: http://en.wikipedia.org/wiki/Fractal [Accessed 18 August 2009].

[6] – Anonymous. 2002. L-Systems. [online] (Updated 13 August 2009)
Available at: http://en.wikipedia.org/wiki/L-system [Accessed 18 August 2009].

[7] – Prusinkiewicz, P. Lindenmayer, A., 1996, The algorithmic beauty of plants, Springer Virtual Laboratory Series.

[8] – Ericson, C. 2004, Real time collision detection, Morgan Kaufmann Series on Interactive 3D Technology, 2005.

[9] - John Lynch, 2009, Voronoi - dynamic - location based fracture (WIP), [Online] (Updated 30 June 2009) Available at: http://forums.odforce.net/index.php?/topic/9119-voronoi-dynamic-location-based-fracture-wip/page__hl__voronoi [Accessed 1 July 2009]

Other References:

[1] – David Gary's Houdini procedural city tutorial – cmiVFX

[2] - Side Effects Software. Houdini. 2005. "Houdini Documentation" http://www.sidefx.com

[3] - Alexander C, Ishikawa S, Silverstein M; A Pattern Language: Towns, Buildings, Construction. Oxford University Press 1977.