Procedural settlements generation

Content

1 Background ... 3
2 Introduction ... 4
3 Generation of settlement characteristics ... 6
3.1 Procedure input (independent parameters) ... 7
3.1.1 Geography ... 7
3.1.2 Culture ... 7
3.1.3 History ... 7
3.2 Settlement characteristics (dependent parameters) ... 8
3.2.1 Location ... 8
3.2.2 Architecture ... 9
3.2.3 Psychological characteristics of the people ... 9
3.2.4 Stage of development ... 10
3.2.5 Population ... 10
3.2.6 Government form ... 11
3.2.7 Military ... 11
3.3 Dependency formulas ... 12
3.4 Conclusion ... 12
4 Geometry generation ... 13
4.1 Road generation ... 13
4.1.1 Introduction ... 13
4.1.2 Algorithm ... 13
4.1.3 Optimizations ... 16
4.2 Building generation ... 17
4.2.1 Introduction ... 17
4.2.2 Algorithm ... 18
4.2.3 Relation with road network and settlements characteristics ... 21
4.2.4 Optimizations and other ideas ... 22
5 Conclusion ... 22
6 References ... 23

1 BACKGROUND

There has been a growing demand for procedural content generation (shortly PCG) algorithms in the game development industry for quite some time. With constant increase in hardware power and consumer appetite for more content, detail, realism and scale the, process of manual creation becomes more time consuming and more expensive. One of the possible solutions for that problem lies in PCG algorithms. Today there is a large number of such algorithms with various applications. We are only interested in those useful in games, so before we begin we have to define what it is that we want as a result and in what way do we want to achieve it. Do we need content to be physically real? Do results need to be "correct"? The answer is no. Players, probably, won't have any interest in those details. We need to focus on how interesting that content is for the player and how he will react when he sees it. So what we will be most interested in is efficiency, variety of output, flexibility of algorithm and, of course, aesthetic quality of generated content.

An algorithm for procedural generation of settlements will be presented here. The goal is to design an algorithm, which will meet all of the conditions stated above. Settlements have to be different, but at the same time similar. Combining different culture to create new architecture styles would be a great addition. Settlements should feel as real as possible (building placement, roads etc.).

2 INTRODUCTION

The algorithm is divided into several phases. Each phase generates a different type of content. On the highest level there are two phases:

1) Generating characteristics of the settlement: population, government form, stage of development etc.
2) Generating the geometry of the settlement

In the first phase, the key concept on which the algorithm is based on is called “settlement characteristic”. The characteristic represents an abstraction of something that defines a settlement. It can represent population, military, industry etc. Based on the independent parameters (input in the algorithm), various other mutually dependent characteristics are generated. Characteristics and their total number that will be presented here, don’t have to be in that particular form, this is just one example. They could be extended or narrowed depending on the needs of the specific game. The first phase is divided into several smaller steps.

Input in the procedure (independent parameters) are:

1) Geography: terrain, resources, arable land, vegetation and water/land
2) Culture: dogmatism/skepticism, spirituality/materiality, architecture
3) History: wars, natural catastrophes and previous settlements that used to exist in that location

Settlement characteristics (dependent parameters) are:

1) Location
2) Architecture
3) Psychological characteristic of the people
4) Stage of development
5) Government
6) Population
7) Military

In the second phase, algorithms for geometry generation are being called. The second phase is divided into two steps:

1) Road network generation
2) Generation of buildings

After the second, phase map creation is a trivial task.

A dependency graph of the listed parameters and steps, in which they are being calculated and generated, is shown on the figure 2.1

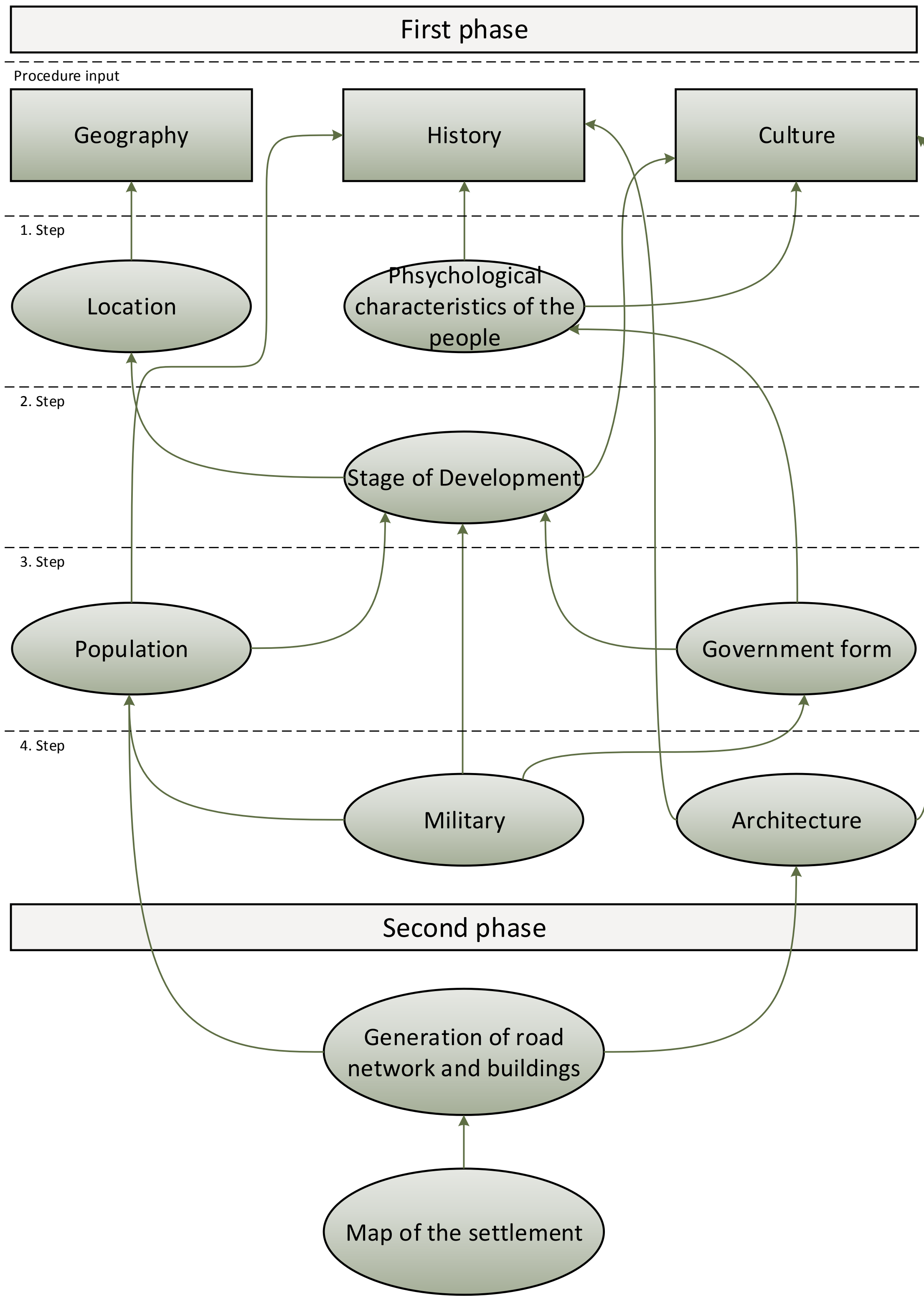

First phase
Procedure input
Geography
History
Culture
1. Step
Location
Phsychological characteristics of the people
2. Step
Stage of Development
3. Step
Population
Government form
4. Step
Military
Architecture
Second phase
Generation of road network and buildings
Map of the settlement

Figure 2.1

3 GENERATION OF SETTLEMENT CHARACTERISTICS

The goal is to design a flexible algorithm that could be used in different games where the settlement could be interpreted differently.

First of all we have to define what a settlement is, what settlement characteristics are and how we are going to use them. It needs to be as intuitive as possible, but at the same time simple for calculation and manipulation.

A settlement will be represented as an abstraction that contains a set of characteristics. The idea is that we can define any number of different types of characteristics (independently from settlements), dependency between them and the way they depend on each other (dependency formulas). Once defined, a characteristic can be saved and used anytime we need it. For example, the military would probably be of no importance in an RPG game, but crucial in an RTS. Contrary the quests would be crucial in an RPG, but unimportant in an RTS. One possible solution is shown in figure 2.1 and it will be used as an example further on.

For the sake of clarity, a characteristic will be imagined as a structure that contains fields (in most cases a numeric type) and a list of dependencies. The method for calculation of every field will be explained for every characteristic. Independent parameters (procedure input) contain only fields.

Implementation of characteristics and settlements could be done in many different ways depending on the programming paradigm we use. One of the possible implementations would be an object-oriented one, where a characteristic would be an abstract class that would contain a list of other characteristics it depends on and an abstract methods for calculating those dependencies. Specific characteristics, their fields and methods for calculating field values (dependency formulas) could be defined using inheritance. A settlement could be a class that would contain a list of all characteristics. A similar idea could be applied on component-based engines where characteristics as components would be “attached” to the settlements.

3.1 PROCEDURE INPUT (INDEPENDENT PARAMETERS)

3.1.1 Geography

Fields:

- water/land
- terrain
- resources
- arable land
- vegetation

It would be best to pass parameters of this type as image maps. For example a greyscale image could represent water and land, or an RGB image could represent desert (red), forests (green) and open healthy terrain (blue). Optionally an image with passable and impassable areas could be generated, as it is necessary information for the road generation algorithm. Resources will be implemented as a simple list of coordinates, type of the resource, and amount for each one of them.

3.1.2 Culture

Fields:

- Dogmatism/Skepticism
- Spirituality/Materiality
- Architecture

First two fields should be implemented as a floating point number (in [0, 1] interval) that would represent a ratio between corresponding values. For example, *Dogmatism/Skepticism* value could be used to calculate how much a population is educated, and how technologically and artistically advanced. For instance, in the Middle age everything was directed towards the church while arts and science were in stagnation under the influence of the church dogma. *Spirituality/Mentality* could be used to decide whether that settlement is going to help other ones in need, by giving them some of the resources. Spirituality could also be a way to unlock some "powers" that inhabitants could possess.

The concept of the architecture field will be explained later in the geometry generation algorithms. For now let's say it is a set of parameters and shapes for that algorithm.

3.1.3 History

Fields:

- Wars
- Catastrophes
- Previous settlement

Wars and catastrophes could be simply represented as a number that would represent their "amount" in given units.

Previous settlement could be either a reference to some other settlement that won't be placed in the game, or a subset of relevant characteristics. It could be used for calculation of some characteristics that the settlement could receive from the previous one (e.g. architecture, technology etc.).

3.2 SETTLEMENT CHARACTERISTICS (DEPENDENT PARAMETERS)

3.2.1 Location

Dependencies:

- Geography (all fields)

Fields:

- Coordinates
- Water availability
- Resources
- Arable land
- Vegetation
- Traffic

All fields could be represented as numbers that would represent their "amount" in given units. Naturally, *coordinates* would be terrain coordinates. The field *Resources* could contain a number representing the amount of each resource.

Calculation:

Settlement coordinates should be chosen as close to water, resources and healthy land as possible. If we represent the desired points as weighted points then an iterative realization of the Weber's problem (finding the optimal location between input points) could be used. Algorithm starts from some initial point in space and iteratively moves in the direction where the weighted sum of distances between the approximated point and the input points is smaller. The process is repeated until there's no smaller sum in any direction.

The algorithm could be optimized with following modifications:

- A smart selection of the initial point (e.g. mean value of product of all points and their weights)
- Length of the step along the axis could be increased as we don't really want precise location
- Maximal number of iterations could be implemented. That way the algorithm time of execution could be limited.

With the last two modifications we can calculate the time that the algorithm will work in the worst possible case. Now we can, by careful calculation of the step length and the maximum number of iterations, be certain that the algorithm won't work longer than we want it to and still get good results.

The input points would actually represent water, resources etc. Weights should be chosen like they are in the real world (e.g. water should have the largest weight). It would be best if they are parameterized so we can control the algorithm better (maybe some civilizations don't need water to survive).

After we calculate the coordinates, the remaining fields (except for traffic) could simply be calculated as ratio between amount and distance. Fields like vegetation and arable lands could be sampled, using image maps, from some region around the settlement (e.g. some given radius).

Calculation of traffic creates a small problem as it needs to be postponed for the moment after all settlements and roads are generated. It cannot be calculated without those informations. This creates one exception in the idea of dividing algorithm into steps.

3.2.2 Architecture

Dependencies:

- Culture (architecture)
- History (previous settlement (architecture))

Fields:

- Architecture elements (shapes)
- Street types

It will be explained in more details after we go over the algorithm for geometry generation (Geometry generation).

3.2.3 Psychological characteristics of the people

Dependencies:

- History (wars)
- Culture (Spirituality/Materiality)

Fields:

- Aggressive/Peaceful
- Helping/Selfish
- Friendly/Unfriendly

They would be best represented on scale [0, 1].

Calculation:

The *Aggressive/Peaceful* field could be calculated on the base of the number of wars that happed in the settlement. People who waged war frequently have a higher chance of being aggressive. This field could be used, for example, when deciding whether that settlement is going to attack its neighbors.

The *Helping/Selfish* could be calculated based on the *Spirituality/Materiality* field. Material people will probably want everything for themselves and won't share anything. It could be used to decide whether that settlement is going to donate some resources to others in need.

As for *Friendly/Unfriendly*, it is hard to explain real dependency, so it would be best to calculate it as a random number in [0, 1] interval. It could be used to decide whether that settlement would be willing to to trade, or form alliances.

3.2.4 Stage of development

Dependencies:

- Location (resources, vegetation, arable lands, traffic)
- Culture (Dogmatism/Skepticism)

Fields:

- Industry
- Trading
- Agriculture
- Livestock breeding
- Science/Technology/Education

Calculation:

Those fields should be implemented as numbers that would represent their “strength” in given units.

Every field would be calculated with a specific dependency formula using some of the listed dependencies. Industry would depend on resources in the region, trading on traffic, agriculture on arable lands, livestock breeding on vegetation, and Science/Technology/Education on Dogmatism/Skepticism.

Additionally climate could be included in the *location* field and used for calculation of fields.

3.2.5 Population

Dependencies:

- Stage of development (average value of all fields)
- History (wars, catastrophes)

Fields:

- Population count

Calculation:

Population count is proportional to the stage of development, and inverse to wars and catastrophes. Optionally we can add male/female ratio where female would prevail in case of a high number of wars.

3.2.6 Government form

Dependencies:

- Stage of development
- Psychological characteristic of the people

Fields:

- Military influence
- Type

Calculation:

The government type will be divided in four types:

1. Dictatorship,
2. Democracy (Republic),
3. Aristocracy (rule of the upper class),
4. Empire.

Each one of the listed types influences the population and the military in a different way, and each one of them is established under different conditions.

Dictatorship establishes if the stage of development is low, democracy if people are educated and aristocracy if the settlement is rich (high stage of development). For an empire to be established people have to be aggressive, well-educated and the settlement has to be highly developed. The chance of establishing each of the types needs to be calculated so the one with the highest probability could be chosen. Optionally some random deviation could be added so the algorithm would be less deterministic.

In accordance to the government type there will be more or less military power. For example, the empire would force military development so it could conquer. Also, the happiness of the people could be added. It would depend on the government type and it would manifest itself with frequent or rare occurrence of a rebellion. Rebellions could be represented with a decrease of population with some chance of changing the government. Some characteristics should be recalculated in that case.

3.2.7 Military

Dependencies:

- Population (count)
- Stage of development (Science/Technology/Education)
- Government (military influence)

Fields:

- Military count
- Weapon power

Calculation:

Military count depends directly on the population and government type. With more population comes more soldiers, and with more aggressive government more people will join the military. Weapon power, represented as some number in given units, would depend on technological development of the settlement. Advanced technology produces more powerful weapons.

3.3 Dependency formulas

Specific formulas could be any mathematic functions as long as they respect the proportionality of the phenomenon they describe. In the simplest form, they could be represented as a ratio between corresponding values multiplied by some coefficient that could be parameterized for better algorithm control. Once the algorithm is implemented a series of experiments could be executed to get the most balanced output.

More advanced approach would be to actually calculate desired values for each function, and then use some interpolation method to get analytical form of those functions.

3.4 Conclusion

With this we finish the generation of the settlement characteristics. The power of this approach lies in the fact that, once implemented, characteristics don't need to be implemented again and could be reused when we need them. With this approach we can easily approximate real world dependencies accurately enough for the video games and change them whenever we want.

Many other dependencies could be added, e.g. psychological characteristics of the people depends on stage of development, architecture also and so on, but the compromise between reality and simplicity needs to be made.

4 Geometry generation

After we've defined the settlement and all of its characteristics we need to create its physical form. But before we start with geometry generation we need to know the population density across the map. So, as settlements are being generated algorithm needs to draw population density into the population image map. That image map will be used by the road generation algorithm. After the whole road network had been generated, inside high density population zones (that's where settlements are), blocks between roads are being divided into smaller lots that will represent locations where building are going to be generated. After that, those locations are being passed to the building generation algorithm that uses the architecture characteristic information to generate the buildings. First the road generation and then the building generation algorithm will be explained.

4.1 Road generation

4.1.1 Introduction

The road network will be represented as an undirected graph, where each node in the graph will correspond to each node in the road network. It will be implemented as an adjacency list, since operations with neighbors are frequent.

There are two types of roads: streets and highways. Highways will be used to connect different settlements, and streets to create the road networks inside the settlements. First, algorithm generates highways in one pass, and then streets for every settlement in the second.

The algorithm works by serving road segment generation requests, and after each one successfully served, it creates a several new requests. The **road segment** is represented as an edge between two nodes.

4.1.2 Algorithm

Requests for road segment generation is represented as **req (timeDelay, node, metaInfo)**. Parameter *timeDelay* represents a delay after which a request is going to be served. Parameter *roadNode* represents the node that is requested to be added into the graph, and *metainfo* the informations relevant for the functions that the algorithm is using.

There are two functions:

- function *globalGoals()* (shortly **GG**) for creation of request, based on the global goals ,
- function *localConstraints*() (shortly **LC**) for processing requests, based on the local constraints

GG function creates new requests, **LC** processes them and if they are successfully served adds them into road network graph. Pseudocode and explanation of the algorithm are given below.

Requests are places in the priority queue **Q**. It is initialized with the first road segment. **LC** function processes it, modifies it if necessary and returns the status. If the request is successfully served the node is added into the road network graph **S** and then **GG** is called to generate new requests that actually represent branching of that last road segment. Those requests are then added into the request queue **Q.**

Algorithm repeats until queue becomes empty which indicates that graph generation is complete.

```
initialize priority queue Q with a single entry: r(0, node0, metaInfo0)
initialize segment graph S to empty

until Q is empty
  delete smallest r(timeDelay_i, node_i, metaInfo_i) from Q (i.e., smallest ‘timeDelay’)
  accepted = localConstraints(&r)
  if (accepted)
  {
    add segment(node_i) to S
    foreach r(timeDelay_j, node_j, metaInfo_j) produced by globalGoals(node_i, metaInfo_i)
      add r(timeDelay_i + 1 + timeDelay_j, node_j, metaInfo_j) to Q
  }
```

GG function can create different types of branching. It is specified with *metaInfo* parameter. Functions should be designed so they can be easily extended with new types. Few, most common, types will be showed here. There will be only one type of highway and four types of the streets (figure 4.1):

1. rectangular,
2. radial,
3. basic,
4. least slope.

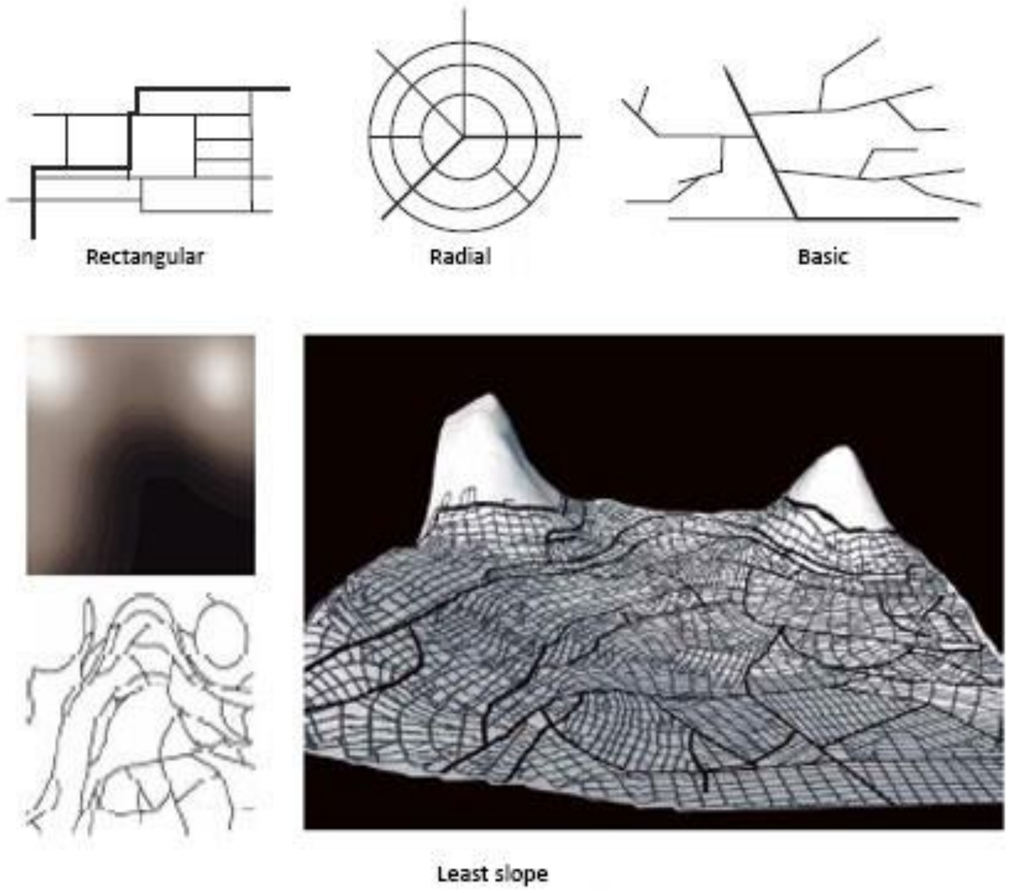


Figure 4.1

Each road type would use different parameters. For example, radial would use polar angle, basic maximum deviation angle etc. The desired road type could be set based on the architecture of the settlement, which leads us to the first field of the Architecture characteristic. Propagation of the road is realized using the concept of graphic turtle, that is, the movable cursor that always moves relative to the last position.

Highways are supposed to connect different settlements. Population image map will be used to navigate highways propagation. Direction in which the highway should propagate could be determined by raycasting in a few different directions and calculating the total sum along each ray proportional to the population divided by distance. Direction with the largest sum is selected (figure 4.2). If some of the sums are close enough to each other, then the road should branch in all of the corresponding directions. If the current node happens to be inside high population density area, it implies that the highway came into the settlement and should be branched in the basic way until it gets out of the settlement. Optionally, for the sake of lifelikeness, branches should be unconditionally created with some probability in some random direction. Those branches will either get in some of the settlements or connect to the rest of existing road network forming crossroads.

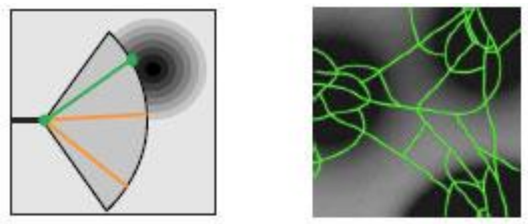


Figure 4.2

The **LC** function handles the processing of road segment requests and returns status whether it's successfully processed or not. There are a few types of actions that could be performed:

1. If the node gets into an invalid area then one of the following actions will be performed:
    1.1. Prune the segment so it fits into the valid area,
    1.2. Rotate the segment within maximal angle until it is completely inside the valid area
    1.3. If the node belong to the highway it could be flagged and later replaces by e.g. a bridge or tunnel
2. Self-awareness actions:
    2.1. If the segment intersects with an existing one, generate a crossing
    2.2. If the node is close to the existing crossing extend it so it reaches it
    2.3. If the node is close to an already existing segment then extend it so it forms an intersection
3. If processing is unsuccessful return *false* status.

As it could be seen, this algorithm is "self-aware" and it forms cycles/crossroads if the node is close to some other node or road segment.

After the whole graph has been generated it's time to call building generation algorithm and pass the graph on to the rendering system so roads could be actually drawn on the terrain. Between nodes we could interpolate splines to get a more realistic look.

4.1.3 Optimizations

The algorithm could become even faster by implementing multithreading version. Current thread could continue along one of the newly created branches, and a new thread could continue along other branches (thread pool would increase speed even more). Synchronization has to be done to prevent concurrent write into the queue **Q** and graph **S**.

Road network implemented this way allows us to use it in many different ways. We could use pathfinding algorithms with ease, or draw a map of the roads, generally any graph algorithm could be used if we need it.

4.2 BUILDING GENERATION

4.2.1 Introduction

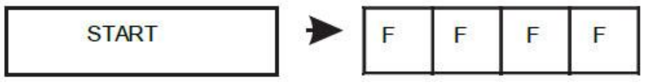
For the purpose of procedural building generation we will use the concept of the formal grammar, called *Split grammar*, based on *Shape grammar* which was introduced by Stiny[1]. The algorithm operates with shapes and as every formal grammar it contains a minimum of three sets (figure 4.3):

- V: set of variables
- W: set of axioms (initial state)
- P: set of production rules

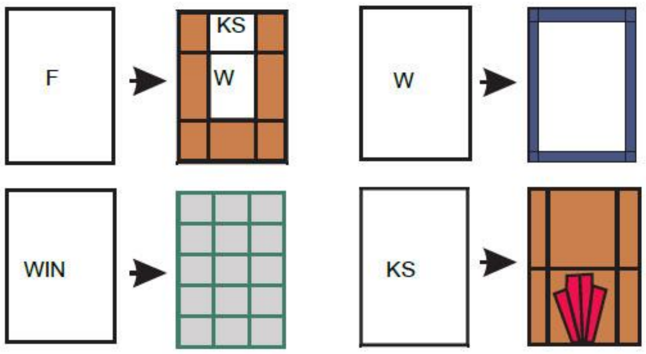
Set of variables **V** contains all basic shapes that we would use for generation of some building (cube, sphere, window, door, roof etc.). In theory, any shape could be synthesised form primitive shapes (cube, sphere etc.), but that approach would be quite unpractical. The idea is that the mass models are being generated only by primitive shapes. Later on we can add paint, façades, windows, chimneys, balconies and other architectural elements to those mass models. That way we managed to design a very powerful and flexible algorithm that could generate a great variety of buildings, by sacrificing only a small amount of procedural logic. Also, multiple architectures could be mixed into one, by taking some of the elements from each set of variables, which fits ideally into the needs of the settlement generation algorithm and the idea that the architecture of the settlement depends on history and settlement that was there before.

Set of axioms represents the initial state from which the production process begins.

Set of production rules represents all of the production rules that are possible in given system.



Set of axioms

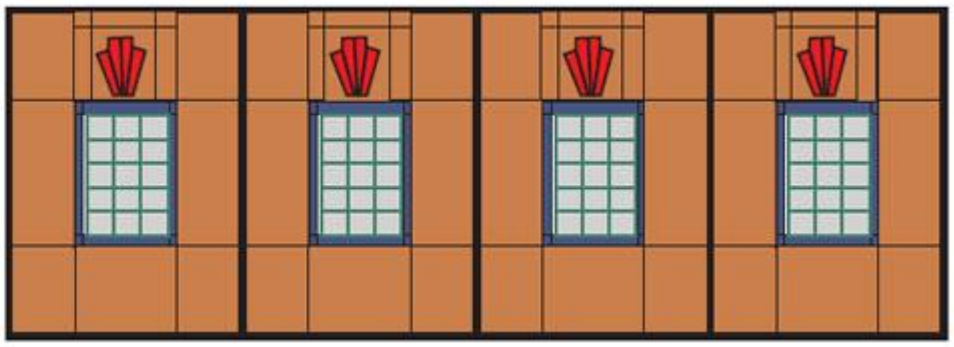


Set of production rules

Figure 4.3

[1] George Stiny is an American design and computation theorist. He co-created the concept of shape grammars.

After the reading of initial set **W**, the algorithm begins with the iterative process of transformations using production rules given in the production rule set **P**. For the example given in the figure 4.3, output is given in the figure 4.4.

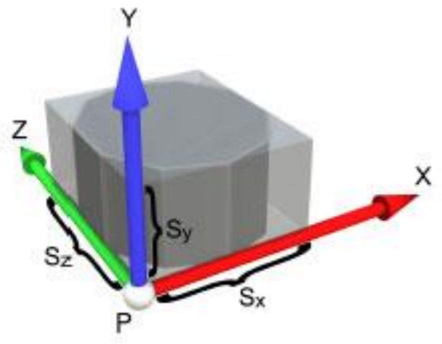


Final output

Figure 4.4

4.2.2 Algorithm

Shape grammar works with shapes that consist of: symbols (strings), geometry (geometry attributes) and numeric attributes. Symbols could be terminal and nonterminal, hence corresponding shapes are called terminal and nonterminal. The production process is finished when algorithm reads any of the terminal symbols. The most important geometry attributes are position *P*, orthogonal vectors *X, Y* and *Z* describing local coordinate system of the shape, and size (scale) vector *S*. Those attributes describe the bounding box called *scope* (figure 4.5).



Bounding box (*scope*)

Figure 4.5

Production process functions in three steps:

1. Select active symbol B
2. Find production rule for symbol B and calculate his successor BNEW
3. Flagg B as inactive and add BNEW shapes into the new configuration and return to step 1.

When there are no nonterminal symbols production process is finished.

Production rules are defines in the following form

id : predecessor : cond -> successor : prob

Id is the production rule identifier, *predecessor is* a symbol that needs to be replaced by a symbol successor, *cond* is a logical expression that needs to be true for the production rule to be applied, and *prob* is the probability for the production rule to be applied, represented by a number on [0, 1] interval.

Some additional functionalities that need to be implemented in an algorithm for easier and more efficient writing of the production rules are given below:

Transformations and instancing of a model/primitive:

Example: A -> [T(0, 0, 6) Rx(90) I("window.fbx")]

T, Rx, Ry, Rz and *S* are translation, rotations around axis of Decartes coordinate system, and scaling respectively, *I* is operation of instancing a model, and brackets are an operation saving the current or restoring the previous coordinate system.

Subdivision:

Example: fac -> Subdiv("Y", 3.5, 0.3, 3, 3, 3) { floor | ledge | floor | floor | floor }

Quotation marks are used to specify the split axis, and the remaining parameters describe the split sizes. Between the delimiter { and } a list of shapes is given, separated by |. It's useful to implement the usage of relative dimensions because we usually don't know the dimensions of the shape we need to split.

Repeat:

Example: floor -> Repeat("X", 2) { B }

The floor will be tiled into as many elements of type B along the x-axis of the scope as there is space.

Component split:

Examples:

fac -> Comp("faces") { A }

fac -> Comp("edges") { B }

fac -> Comp("vertices") { C }

This allows us to split shape into shapes of lesser dimensions. For example, we often need to extract faces of the cube to get facades. Optionally, because we often need to extract facades, command that does exactly side faces split should be implemented (*Comp("side faces")*).

Shapes could be implemented as an *octree.* That structure is really efficient in this algorithm since shapes are often being split and modified.

We could go further and implement functionality for intersection test so we can avoid some unwanted situations (e.g. generating window partially inside a pillar on the wall). Also functionality for testing the intersection of sightlines that could be used e.g. for generating doors if they are “looking” at the street. Another functionality that would be useful is snap, so we can generate windows properly.

This algorithm gives us good and efficient results. Using a relatively small set of rules (around 15) and elementary shapes databases, a great variety of building could be generated. (figure 4.6).

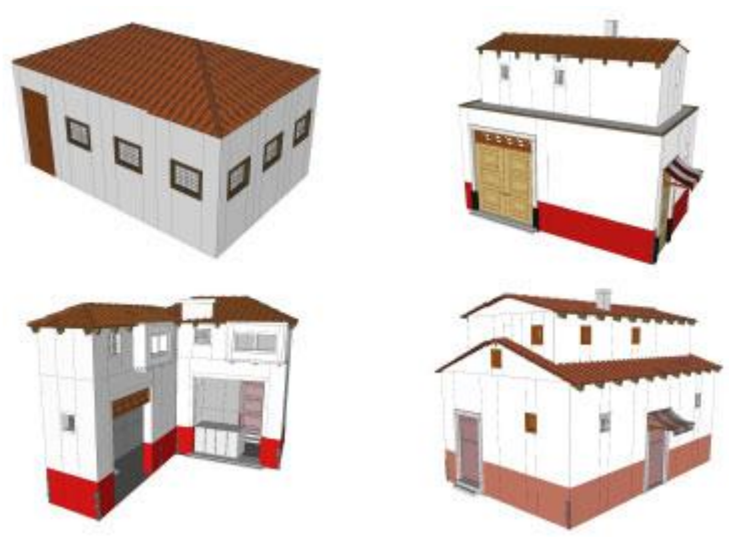


Figure 4.6

Shape grammar can also be used to generate and place other components in an urban environment (figure 4.7).

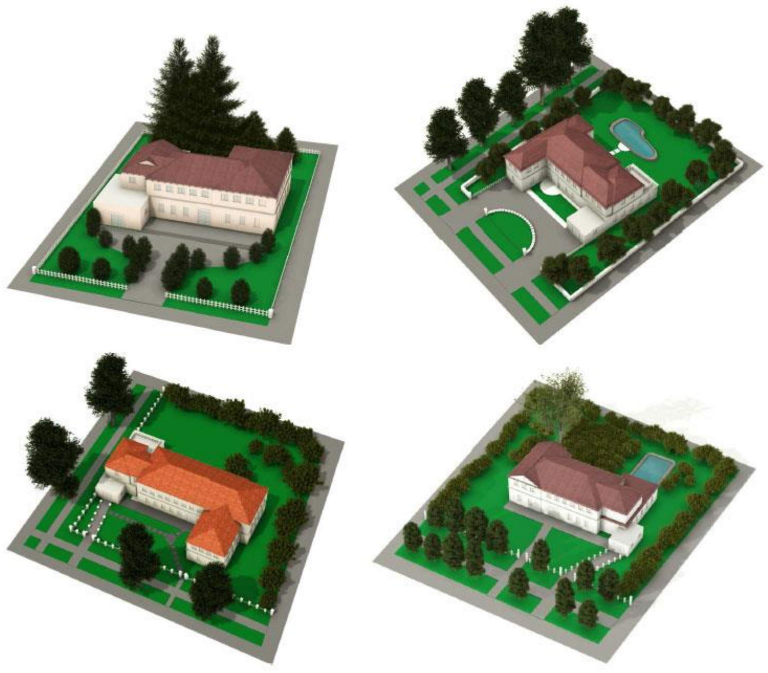


Figure 4.7

The grammar in this example uses the following strategy:

1. Split of the property edges with a component split and place shrubs near the fence
2. Split the property to model the front yard, backyard and the main building
3. Generate a sidewalk and place trees
4. Generate a building with some of the already defined set of production rules (presumption is that in this moment we already have at least one defined set)

Trees could be generated very efficiently using algorithms based on the L-system. They are really similar to the shape grammar algorithms.

4.2.3 Relation with road network and settlements characteristics

After we've designed an algorithm for procedural building generation all that's left is to extract building locations from the road network graph and extract shapes that represent architecture elements from settlements characteristic Architecture. Those elements will be added to the set of variables.

Now we have defined settlement characteristic Architecture that we've been postponing until now. It consists of the type of the streets, elements of architecture and it could contain some other various parameters (e.g. average number of floors).

To get exact lots where building are to be generated the road network graph needs to be divided into blocks. That will be accomplished using the MCB (*Minimum Cycle Basis*) algorithm defined by David Eberly. Having found that cycle, we remove it from the graph and add it to the list of blocks and repeat that until the graph becomes empty or acyclic. Then we recursively divide block into smaller lots until a minimum size is reached. Lots without road access are discarded (figure 4.8).

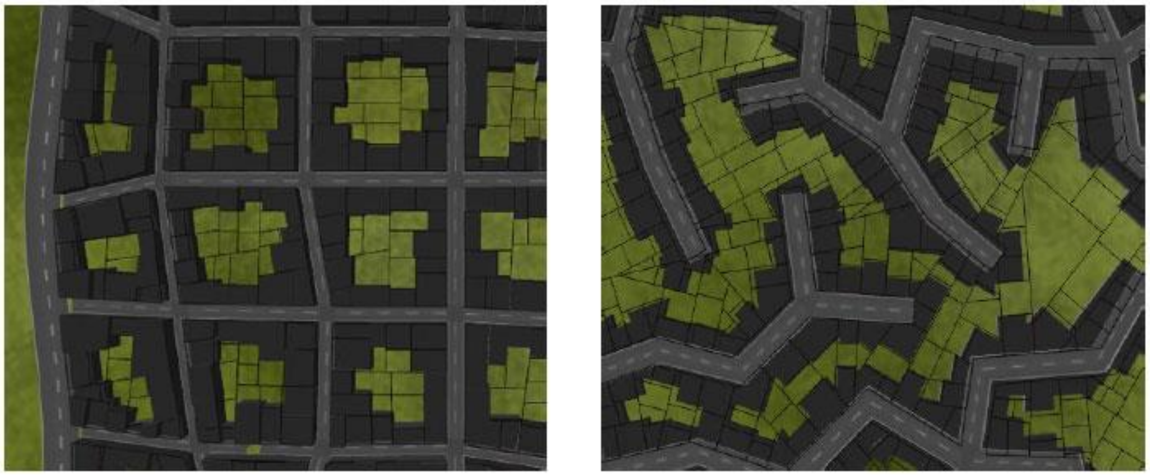


Figure 4.8

Empty regions surrounded by buildings could be used for parks, parking lots, magical totems, portals, practically anything that comes into our mind.

4.2.4 Optimizations and other ideas

Since all buildings follow the same template, regardless of the architecture (they all have doors, windows, one or more floors etc.), predefined production rules could be written and parameterized. That would add more flexibility to the algorithm. A lot of logic could be moved to the external functions that would be called by production rules. That way if we want to add some new logic we won't need to change the production rules, but functions which is a lot easier. A similar approach was introduced by Parish and Müller in their L-system algorithm for road generation.

Because there is 1 to 1 mapping of building onto lots and there are no changes being made on any of the shared resources (unlike road generation algorithm) there is massive potential for GPGPU parallelization. That would push the speed of the algorithm to a new level.

5 CONCLUSION

This algorithm gives us great flexibility for generating settlements with a great variety of possible outputs. It's designed so it could easily be extended with new features so it should evolve together with its usage in different games. There is a lot of space for improvement. One idea would be to extend it with functionality to generate plants procedurally on fields in settlements (e.g. using L-systems).

6 REFERENCES

P. Müller, P. Wonka., S. Haegler, A. Ulmer, L. V. Gool – *Procedural Modeling of Buildings*

Y.Parish, P. Müller – *Procedural Modeling of Cities*

G. Kelly, H. McCabe – *Citygen: An Interactive System for Procedural City Generation*

G. Kelly, H. McCabe – *A Survey of Procedural Techniques for City Generation*

Jingyuan Huang, Alex Pytel, Cherry Zhang, Stephen Mann, Elodie Fourquet, Marshall Hahn, Kate Kinnear, Michael Lam, and William Cowan, David R. – *An Evaluation of Shape/Split Grammars for Architecture*

Bjorn Gunnar Eilertsen – *Automatic road network generation with L-systems and genetic algorithms*

Nemanja Đurkić – *Diplomski rad* JAVA APLIKACIJA ZA VIZUELNU REPREZENTACIJU WEBEROVIH MULTIFOKALNIH KRIVIH U METRICI MINKOWSKOG