Delft University of Technology
Faculty of Electrical Engineering, Mathematics, and Computer Science
Computer Graphics and CAD/CAM Group

Simulating Urban Area Development for Semantic Game Worlds

Author:
G.R. de Ridder

Thesis committee:
Prof. dr. ir. F.W. Jansen
Dr. ir. R. Bidarra
T. Tutenel, M.Sc.
Dr. ir. K.V. Hindriks

July 31, 2012

Preface

This thesis is submitted in partial fulfilment of the requirements for the degree of master of science in computer science. It is the result of my master of science project which I conducted at the Computer Graphics & CAD/CAM group of the Delft University of Technology. This project started in November 2011 and ended in July 2012.

The goal of this project was to develop a procedural approach to generating virtual urban areas for semantic game worlds. The resulting models of this approach should be shaped by the rules of the semantic game worlds to be able to use them in semantics-based games.

I would like to thank the following people for their suggestions, discussions, reviewing, and moral support: Tim Tutenel, Marnix Kraus, Anne van Ee, Antony Löbker, Nick Kraayenbrink, Arthur Vromans, Christian Kehl, and Erik Jansen.

Abstract

Due to the growing need for content in games, more money is spent on creating game content. Many games are situated in urban areas, complex collections of buildings and roads. To create these by hand is costly, and therefore procedural techniques have been proposed in the past decades to automate the creation process to reduce the amount of work needed to create these complex scenes. The creation process and the resulting urban areas of these methods, however, lack meaning and cannot be applied directly in a semantic game world due to the lack of semantic information in the model.

In this thesis I present a simulator called *UrbSim* that simulates the development of an urban area over time for a specific semantic game world. The semantic information of the semantic game world is used to shape the generation process of the urban area and it also influences the shape of the created model. By doing this, UrbSim adds more meaning to both the creation process and the urban area itself than previously proposed solutions to procedurally generating virtual urban areas. It is able to produce urban areas that contain both a history and semantic background, each created element has semantic data linked to it, allowing it to be used in the semantic game world it was created for.

Contents

Chapter 1

Introduction

The active development in computer technology allows the game industry to create increasingly bigger, more detailed and realistic scenes for their games. Game developers are in a race to offer their consumers games that take as much advantage of the modern hardware as possible. This poses the game developers a challenge to create more content while budgets are limited. More artists are needed to create the content and this will increase the costs of a game tremendously if no other game features are left out which could result worse game play.

Many games, both entertainment games and serious games, are situated in urban areas. In the Assassin's Creed series [1] for example the player visits many big cities which can be explored. Also the games in the Grand Theft Auto series contain detailed cities [2]. From both game series a screenshot is shown in Figure 1.1. The cities in these game series were modelled by hand and took years and many people to create. Though still many parts of these cities are not interactive and purely act as scenery.

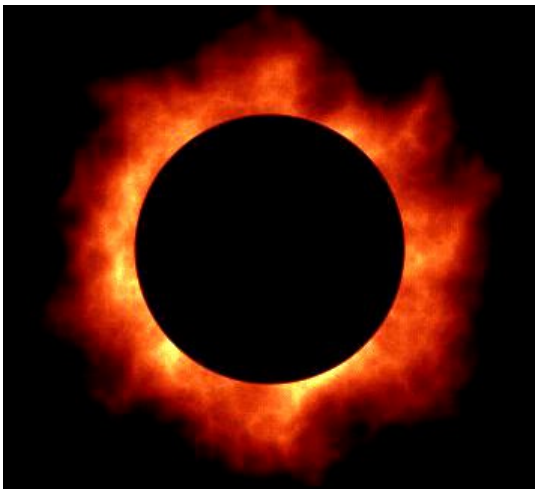
(a) A small view of Venice from Assassin's Creed II [1].

(b) A view from ground level in Liberty City (based on New York City) in Grand Theft Auto IV [2].

Figure 1.1: Two entertainment games which are situated in big urban areas are Assassin's Creed (a) and Grand Theft Auto IV (b).

For the past few decades many procedural techniques have been proposed for generating content for games. These methods promise to reduce the effort, and therefore money, needed to create detailed game content. The subjects range from creating textures and graphical effects [3], [4] and vegetation [5] to complete gaming worlds as done in SketchaWorld [6] and MojoWorld [7]. The latter two combine multiple procedural techniques to create all objects and terrain features needed in the virtual world. Examples of procedurally generated content are shown in Figure 1.2.

Many techniques for procedurally generating urban areas have been proposed, see Chapter 2. These are aimed at recreating patterns seen in real world cities, either by directly implementing templates for road networks or by using rules for growing road networks. Some also include determining land uses of lots created within the cells of the road network either via the user or



(a) One frame from a corona animation which is based on noise [4].

(b) A scene with thousands of plants created using L-systems [5].

(c) A SketchaWorld scene with mountains, vegetation and a city [6].

(d) An alien world generated and rendered with MojoWorld [8].

Figure 1.2: Procedurally generated content.

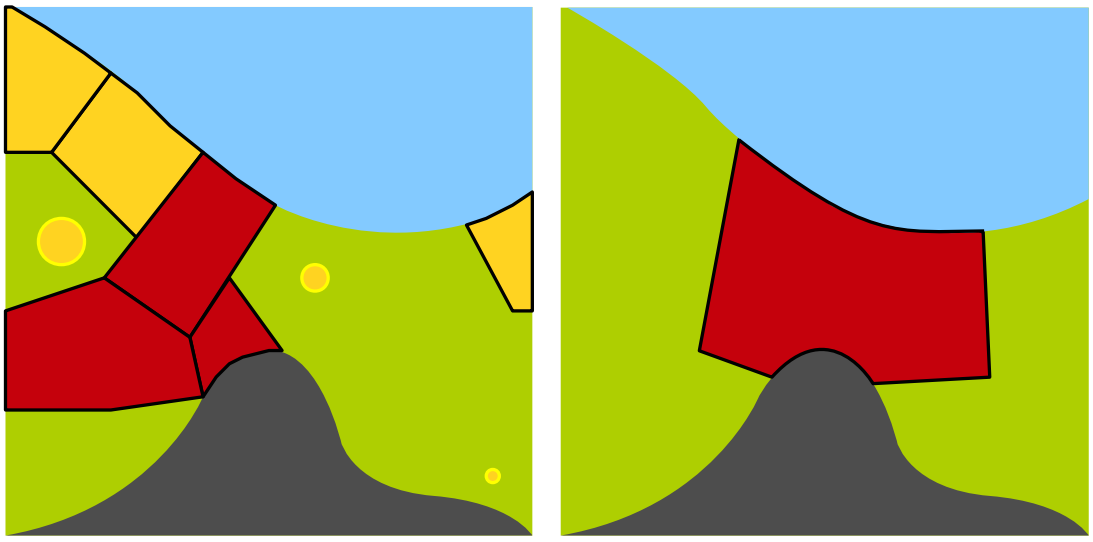
via simulation. Many methods proved to be able to generate realistic results. Unfortunately, the resulting urban areas lack meaning, both in history and semantic relations. Furthermore, the created models cannot be used in semantic game worlds due to the lack of semantic information. The created models are just scenery. Some of the methods do include meaning in the land use simulation and the generation of the road network, but many factors are or cannot be included. The generation process is static and does not change over the course of the simulation although these methods simulate an urban area over many decades.

The complete history of a city is important for its growth, especially the first settlement. A city can start as a small village around a bridge because it is a place for trade and transport. This will cause the city to develop a harbour for transport over water and gates to be able to levy toll on carts and keep unwanted people outside the city. Also the resources play an important role in the early development of villages. If the urban area is near mountains, stone is cheap and will therefore be used more often in buildings than villages far from rocky areas where buildings are more likely to be build from clay bricks or wood. This semantics influences the success and growth of a city and therefore also the later development.

1.1 Research Question & Approach

This brings us to the question: *how can one incorporate semantic information and history in the generation process of a virtual urban area suitable for use in a semantics-based game?* To answer this question I want to develop a generic urban area development simulation for semantic game worlds. *UrbSim*, as it is called, takes into account the meaning behind the steps made in the creation of an urban area, although these steps are not necessarily realistic since game worlds do not have to be realistic. They just need to obey the laws set for the game the urban area is generated for. This will ensure that the created content will fit into the semantics of the game and the created content could even be developed further during the game to make the game world more dynamic. The terrain on which it is build, the available resources, and events that occur (such as disasters) influence the growth of the settlement and also its type. Other factors like technological advancement and neighbouring settlements may also contribute to the final shape of the urban area.

UrbSim simulates the land use over time by placing, updating and removing lots from the urban area. It uses resources to define which lot types repel and attract each other. As can be seen in Figure 1.3(a) three locations were tested on the production values and their distance to the artificial patches. The location at the left near the majority of patches has the highest value and is likely to be chosen. Also the shapes of lots are affected by local terrain features, see Figure 1.3(b). The patch is shaped by the sea (blue) at the top and the rocky area (grey) at the bottom. This gives more meaning to the placement and shapes of lots. The resulting urban area is a collection of lots shaped by the terrain and the resources either produced by the terrain or the lots around it. Each lot has specific semantic data attached to it which could be used in the game to interact with the lot.



(a) Selecting a location for a new patch is affected by all patches in the neighbourhood. The location at the left (the yellow circle) has the highest value, while the one at the bottom right has the lowest value.

(b) The shape of a new patch is affected by the local borders. One edge follows the shoreline at the top while a rocky area creates an indent at the bottom of the patch.

Figure 1.3: The location selection and patch shape are affected by the local terrain features and neighbouring patches.

Novel to this method is the lot generation that is not based on subdivision but rather on growing and expanding from one location to find a suitable shape and size for a lot. Simulation of land use for procedural urban area generation is not uncommon, but the used approach is not grid-based which is not done often [9]. By focusing more on the semantics and game worlds, this solution to generating virtual urban areas differs even more from other proposed methods. It

will extend current techniques by adding more history and meaning to the procedurally generated urban areas.

1.2 Outline

The following chapter gives an overview of procedural methods that have been proposed in the past decade for generation of urban areas. It also discusses the techniques in the light of needing a semantic background. Chapter 3 covers the approach, and Chapter 4 explains the implementation of UrbSim. The fifth chapter shows what this method produces and how it performs. The implementation and results are discussed in Chapter 6. The last chapter gives a short summary of the report and concludes with possible future work.

Chapter 2

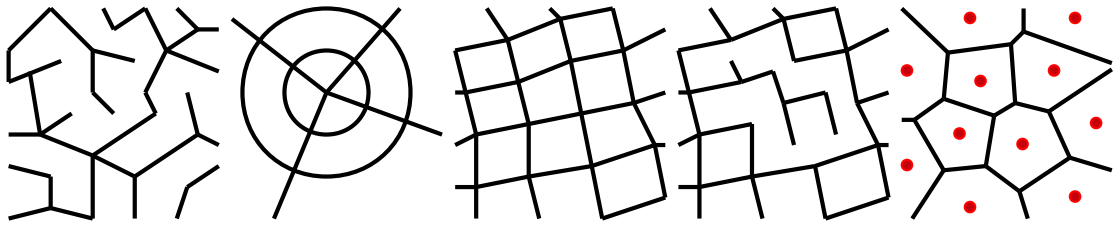
Related Work

In the past decades dozens of papers have been published on procedurally generating virtual cities. Some primarily focus on the creation of realistic buildings [10], some only create a road network [11], and others try to combine these to form urban areas such as [12] and [9]. All of these focus on generating roads prior to creating the lots in which the buildings are situated. Since my work does not aim to create realistic building geometry but rather on a functional simulation of an urban area, I will not cover the solutions for building geometry in detail.

The techniques proposed in the past decades can be divided into two major groups. The first group is inspired by real world city layouts and try to replicate the found road patterns, see Section 2.1. The second group, described in Section 2.2, focuses more on the land use within a city and the development of the city over time. The last section, Section 2.3, discusses the results of the two groups and explains what my focus will be in my urban area simulation.

2.1 Road Patterns

Most methods proposed for procedural modelling of cities involve the creation of road patterns. These patterns are based on road networks found in many real cities. Five of these patterns are listed below and shown in Figure 2.1.



(a) Organic (b) Radial (c) Raster (d) Industrial (e) Population oriented

Figure 2.1: The road network layouts used in the discussed papers.

Organic A tree-like approach for generating a road network (see Figure 2.1(a)). No global planning is needed, only local checks for intersections of roads. This corresponds to the road network of old city centres. This pattern is in many cases produced using an L-system, see Section 2.1.2.

Radial Another pattern observed in cities is a radial pattern, depicted in Figure 2.1(b), in which a few main roads start at the centre of the city and travel outwards. The city is divided into rings enclosed by circular roads. Such a pattern can be found in Paris for example.

Raster When the layout of a city is planned beforehand, the resulting road network is likely to be a rectangular grid (shown in Figure 2.1(c)). The Romans used to build army bases using a strict grid pattern and in some cities (like Florence, Italy) this raster can still be seen. A modern day example of the use of a grid road network is the road network in Manhattan (New York, United States of America).

Industrial The industrial road plan is used in [12]. It is a combination of a raster and organic network. It contains grids, but also dead ends as shown in Figure 2.1(d).

Population oriented The population based approach as used in [13] and [11] searches points of high population density and uses these points as the centre of a Vonoroi diagram. The edges of the diagram form the main road network (see Figure 2.1(e)).

Most proposed methods allow the user to use multiple patterns in the urban area to create a more realistic road network. This can be either for the main road structure or for the secondary roads filling the cells enclosed by the main roads. Also many methods adapt the road network to fit properly to the elevation of the terrain.

The first method described, see Subsection 2.1.1, uses three of the patterns mentioned above to generate a road network. Subsection 2.1.2 covers the Citygen [12], which allows the user to interactively design a city mainly with organic and raster patterns. The CityEngine [14], which is described in 2.1.3, uses a self-sensitive L-system to generate organic, radial and raster patterns based on input maps from the user. The road networks in the tensor field approach (Subsection 2.1.4) are also based on a map, but this map is a grid of tensors. The roads are formed by tracing paths on that map. The last approach described in this section, Subsection 2.1.5, is based on behavioural modelling using agents and adapts its geometry according to the behaviour of the agents.

2.1.1 Template-Based Generation

In the approach proposed in 'Template-based generation of road networks for virtual city modeling' [11] by Sun et al. the user is free in choosing the road patterns per area in the city. The user has to provide a map of legal (land) and illegal (water, parks) areas, a population distribution map and an elevation map. Based on these maps and a selected pattern the application is able to generate a complete network of roads.

The user is able to choose three different patterns, namely population based, raster, or radial. The user is also able to choose a mix of these three patterns. Each pattern is connected to a template that defines the rules and parameters.

A population-based pattern is covered by its own template that tries to extract a Vonoroi diagram from the population density map with smaller cells in more dense areas. From the edges of the diagram the roads are created. This results in more roads in denser areas.

The radial and raster pattern share a template. This template is based on iterative growing of roads. It creates a point at a start location. From that point more points are generated, either in a circle around the first point or in a rectangle, as a grid, depending on the pattern. Then pairs of points are linked to create a road network.

The last template used is the mixed template. This template applies the rules and parameters from the other templates for different regions in the city, creating a mix of different patterns.

All templates described above provide a network of highways for the main transportation of the city, without paying attention to illegal areas such as water and parks. When a road crosses an edge of such an illegal area, a break point is created. These break points can also be created due to the change in template between regions. After finding the illegal road segments between breakpoints, a new road around the illegal area can be created. Another option is to have it simply removed or a bridge built over it.

Once the main road network is completed, a secondary road network for each region enclosed by main roads can be created. Sun et al. have chosen to use a raster pattern for these regions.

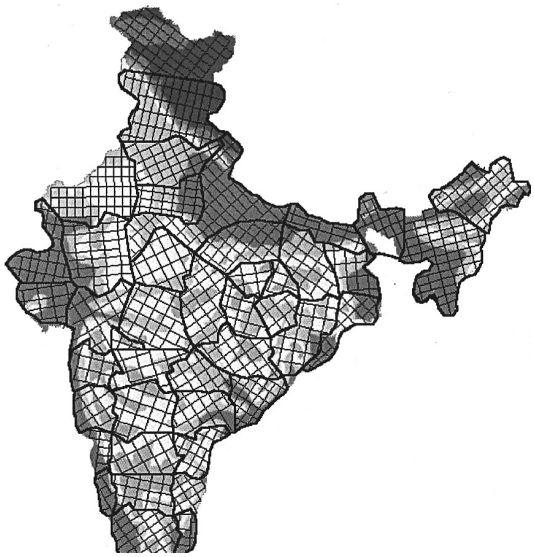


Figure 2.2: A road map with highways and streets in different patterns. [11]

The result of their approach is shown in Figure 2.2. The resulting urban area does not include lots with buildings.

2.1.2 Citygen

Citygen, an application partly inspired by [11], was presented by George Kelly and Hugh McCabe [12]. It does not only feature the creation of highways and streets, but also the procedural generation of building geometry. In their review of procedural techniques for city generation [15] Kelly and McCabe found methods that were lacking realism, needed too much input from the user, or were too slow to be interactive. The creators aimed for an interactive content generator and therefore Citygen had to be fast, while still being able to create realistic cities. The application should allow the user close control over the generation process and the manipulations by the user should be computed and rendered in real-time.

Since the essential character of many cities is often dictated by the pattern of the primary road network [12], the generation of a new city starts with the construction of the primary roads. The roads are presented by two graphs. The first graph is a high level representation of the road network with straight lines between the junctions, called nodes, of the primary roads. The nodes in this network can be directly controlled by the user to change the shape and topology of the network. A second graph is used for more detail of the road. The roads still meet at the same junctions, but can have other trajectories. This is shown in Figure 2.3.

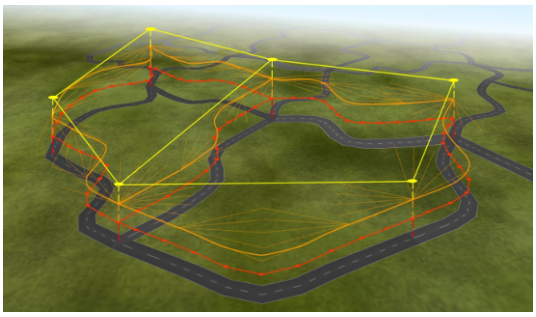


Figure 2.3: The primary roads are represented by two graphs: a high level graph with straight lines (yellow), and a more detailed low level graph (red). [12]

The low level graph is created by growing 'adaptive roads' between the nodes. The algorithm used for this works bidirectionally between two nodes. From each node, samples are proposed in the direction of the other node but with a different angle. For each, the best sample for the next node in the low level graph is chosen. The selection depends on the elevation differences and the distance to the destination (the other node). This process is repeated for the newly created nodes until the two sides meet in the centre. After the creation of the low level graph, the final path is determined using a Catmull spline [16].

To create the secondary roads, first the city cells are extracted from the primary road network using the minimum cycle basis algorithm described by David Eberly in [17]. These cells are regions enclosed by the main roads. In these regions the secondary roads are generated based on a pattern choice by the user. This can be either a raster, industrial or organic pattern.

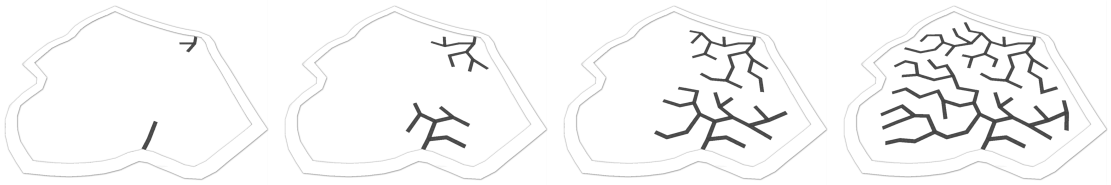


Figure 2.4: The growth of a secondary road network using an organic pattern after 10, 100, 300 and 1000 steps. [12]

The growth of the secondary road networks in Citygen was done using Lindenmayer Systems [18] or L-systems for short. L-systems were originally introduced as a mathematical theory for biological development by Astrid Lindenmayer [19]. In this theory the objects of study are represented by a string of modules that describes all properties of the complex system. By parallel rewriting of the string using predefined rules a new string is generated and the system is updated. In the case of a road network, the roads and intersections are modules in a string that is rewritten per module.

The secondary roads start growing at two points at the boundaries of the city cells (Figure 2.4). The two seeds grow in parallel into a street pattern while being sensitive for existing roads. The growth can be controlled by changing many parameters. For example the segment size controls the length of the proposed segments in the secondary road graph, and the deviance parameter controls the noise introduced to the growth process. Another important aspect of the growth is snapping. When a raster or industrial pattern is chosen, the proposed segments tend to snap to existing junctions and roads where possible.

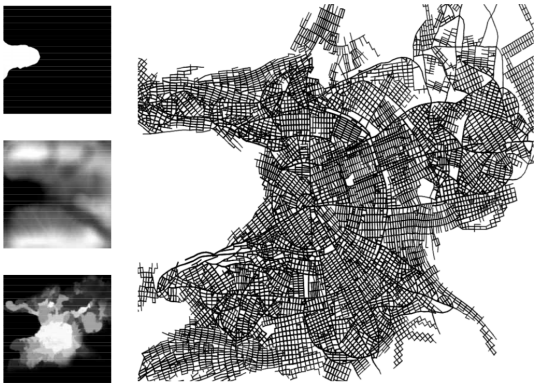
The last stage of the city generation is the creation of the building geometry. For this lots are extracted from the generated road network by iteratively subdividing the space into smaller spaces until the target lot size is reached. Both concave and convex boundaries are supported which enables Citygen to split up complex regions that may occur due to the different patterns used.

The created lots are used for the placement of buildings. The buildings are generated according to what type of area they are situated in. Down-town lots, like those in a raster pattern, will have buildings using as much space available in the lot as possible, while suburban lots, seen adjacent to roads in organic patterns, will have gardens and therefore less space used up by buildings.

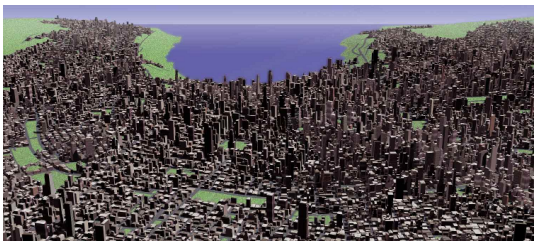
2.1.3 CityEngine

L-systems were also used in the procedural creation of large-scale road networks in CityEngine [14]. Parish and Müller introduced this application in 2001. The goal of their work was to create a system that needs little input, but is highly controllable. It creates a traffic network and buildings, based on the maps provided by the user. These maps define the elevation, placement of water and vegetation, population density, land use, street patterns and the maximum building height.

In Figure 2.5(a) an example of a procedurally generated road network is shown which is based on input maps of water, elevation, and population density.



(a) A generated road network with (on the left) the corresponding geographical and socio-statistical input maps.



(b) A complete city with approximately 26000 buildings.

Figure 2.5: The creation of a three-dimensional city generated with CityEngine starts with a description of the terrain on which a road network is grown. In the last step buildings are placed in the cells of the road network. [14]

To generate such a road network Parish and Müller use an extended self-sensitive L-system. By this is meant an L-system with additional functions that check for global goals and local constraints. The application creates highways to connect areas of the city with high population density. A secondary network of streets is generated to give the inhabitants access to the highways.

The application starts with an initial road segment. In each cycle the L-system proposes new road segments, which are branches of existing roads. When the new segment has been proposed, the global goals function will assign parameter values to it. These parameters, such as orientation and length, aim to create the dominant pattern. A final step tries to adjust the segment to fit in the local area (snapping to other roads, not running into water). If this last step cannot fulfil the local constraints, the proposed segment is discarded.

To generate highways, each highway road-end shoots a number of rays of predefined length at random angles within a predefined range. Samples along this ray are taken from the population density map, weighted and summed up to select the ray with the largest sum. This will be the next road segment. The streets on the other hand typically follow the dominant street pattern (in many cases a raster). The streets stop growing when they reach an unpopulated area.

The creation of highways is also depending on the global pattern chosen by the user. This can be organic, in which the highways follow the population density. The raster pattern forces the roads to make sharp angles and form rectangular blocks. The radial pattern lets the highways form circles around the city centre. Parish and Müller also introduce a specific rule for mountainous terrain, such as in San Francisco, where both streets and highways follow the elevation of the terrain and the roads at different height levels are connected with smaller streets. To enhance the results that were constructed with straight lines, a simple subdivision scheme based on [16] is used which results in smooth roads.

After the creation of the roads, the convex spaces enclosed by roads are subdivided into lots. This subdivision algorithm recursively divides the spaces over the longest edges in half until the areas of the created lots are under a given threshold. Then another L-system creates the geometry for the buildings and for the textures on the buildings a procedural method of layered grids is used. The textures are composed out of several smaller façade elements that are repeated on the building. This in combination with the different geometry created gives a wide variety of buildings. These last steps for creating the buildings is by far the most time consuming part of the process and could easily take several minutes to complete. Therefore this approach could not be used in

real-time, but it did produce very interesting cities such as the one in Figure 2.5(b) which is the virtual city created with the data shown in 2.5(a).

2.1.4 Tensor Fields

A quite recent development is the use of tensor fields to model the road network of a city. An intuitive and flexible modelling framework is presented by Esch et al. in [20] and enhanced by Chen et al. [21]. In this system the user is able to create a street network from scratch or modify an existing network by changing the tensor field on which the network is based by applying operations such as brush strokes and smoothing. It can also be edited directly via changing the graph elements.

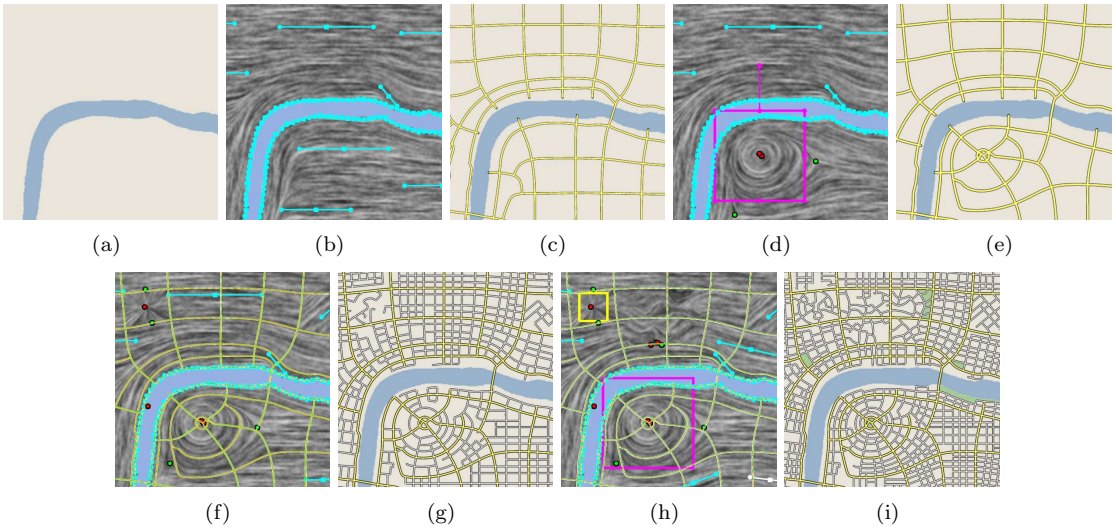


Figure 2.6: Street graph production example. [21]

Figure 2.6 gives an example of how the system can be used to produce a street graph. First a water map is loaded (a) and by adding some tensor field design elements a tensor field is created (b). Then the system generates a main road network following the tensor field (c). The user can also add more elements to refine the network (d, e). The changes can also be done on a local scale, within city cells enclosed by main roads for example (f). Based on the local tensor field the secondary roads are generated (g). The user can modify the field even further by using a rotation noise field to create irregular structures (see (h) and (i)). The following paragraphs will describe these steps in more detail.

The system starts with four maps that describe the environment the city is based in. These maps can be loaded as images. There are two binary valued maps for water and forests. The third map is a height map and also a population map is used. Based on these maps a tensor field can be created. This tensor field is a continuous function that associates every point with a tensor. In this context a tensor is a 2 by 2 symmetric and traceless matrix. The eigenvectors of these matrices are used for finding the paths the roads should follow. Such a path, called a hyperstreamline, describes a curve that is tangent to the eigenvector field everywhere along its path. The hyperstreamlines used follow either the major eigenvector field or the minor eigenvector field. When these hyperstreamlines are parallel for both fields, a grid road pattern is created. A radial pattern is produced when the major hyperstreamlines are circular around a single point, while the minor hyperstreamlines emanate from that point. The system can also create a tensor field from shorelines or other boundaries. In this field the streamlines follow the boundaries. A last form of basis field described is the height field. This field has streamlines following the elevation of the terrain.

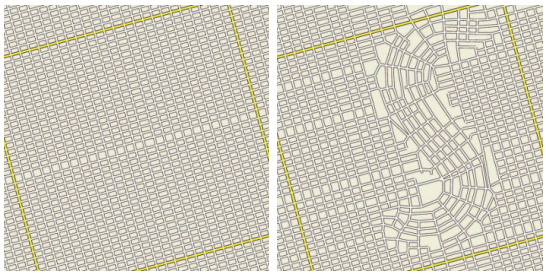
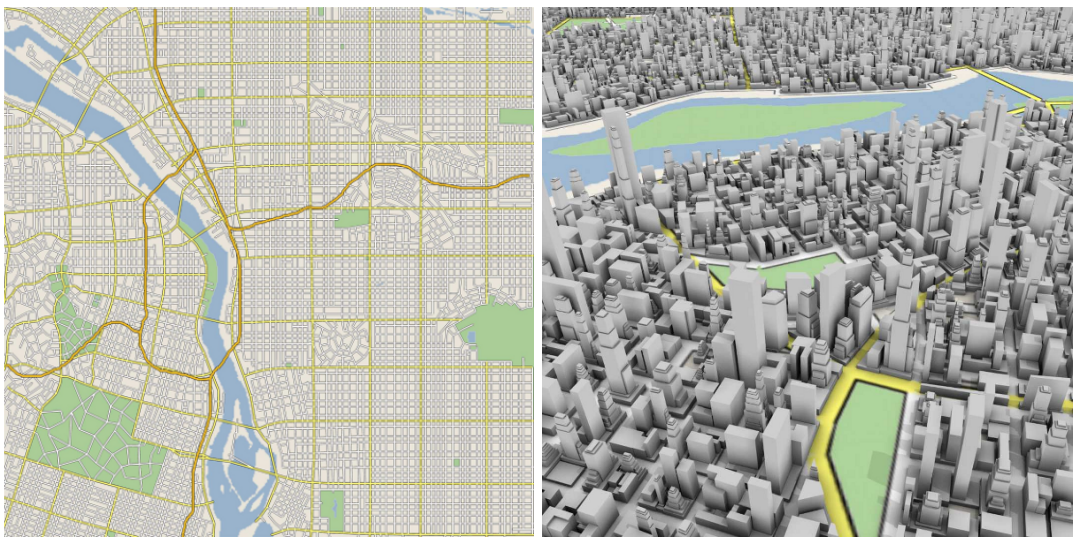


Figure 2.7: The use of a brush stroke to orient streets. [21]

The user can let the system extract the tensor field using the basis fields described above at a global scale, but those can also be applied on a local scale at specific locations to combine multiple patterns. Furthermore the user can apply Laplacian smoothing on the tensor field to reduce its complexity. Another useful tool is the brush interface. This enables the user to make specific changes by drawing curves on the map, such as the S-curve in Figure 2.7.



(a) (b)

Figure 2.8: (a) A generated street graph for down-town Portland, OR, USA, (b) a city complete with buildings based on a stretch of the Benue River in Nigeria. [21]

The road graph is created by following the hyperstreamlines. By repeatedly tracing major and minor hyperstreamline at a pre-set distance, a network of roads is created. This can be done for both major and minor road, but for minor roads, the distance between the streamlines is smaller. The network of roads is converted to a graph of vertexes (crossings and loose ends) and edges (road segments). Also road attributes such as road width and type of lanes is stored at these vertices and edges. Figure 2.8(a) shows the result of an iterative editing process. The highways (in orange) have been added manually.

The procedural generation of buildings was not the focus of the work of Chen et al. in [21], but a module for three-dimensional street and building geometry was included in the system. This module, also used in a newer version of the CityEngine, is further described by [22]. The cells enclosed by the road network are subdivided and for each created lot a building is generated. A resulting city is shown in Figure 2.8(b).

2.1.5 Interactive Agent-based Modelling

Vanegas et al. [23] describe an agent-based system, inspired by UrbanSim [24], that is capable of procedurally generating realistic urban areas spanning over 200 square kilometres within minutes on a standard computer, including the time taken for iterative and interactive design process. The main goal of the system is to close the loop between behavioural and geometrical modelling of cities. Furthermore the system should provide the designer an efficient means to create plausibly realistic urban areas that are internally consistent, without capturing the numerous behavioural nuances and modelling the evolutionary processes that shape cities.

The input from the user consists of an initial terrain map and initial values for a set of behavioural and geometrical variables. These can be edited via a GUI and a paint-brush like tool. When the user changes the terrain or the values of the variables, the system tries to find a new equilibrium of the variables and generates new geometry when this stable solution of the variables is found. Such an editing process can be seen in Figure 2.9. The creation of the geometry mainly depends on the simulation of the distribution of population and jobs. A higher population density or job density automatically results in a high density of roads and higher buildings. To allow incremental editing and editing at various scales the user is able to constrain parts of the generated model and let the system create a new city model.

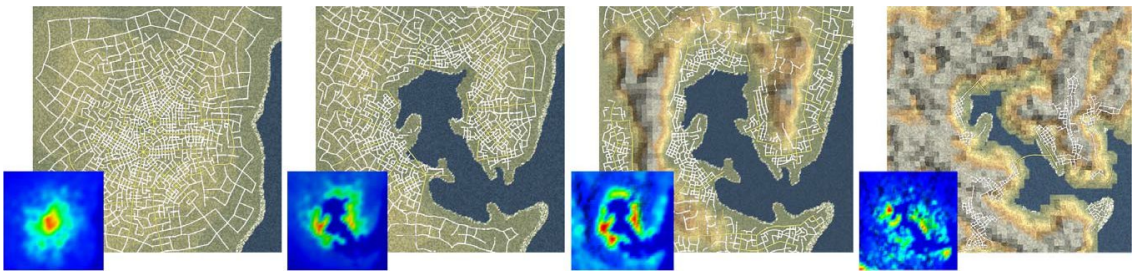


Figure 2.9: The road network is updated when the terrain is changed. The population is kept constant, but its distribution changes (as can be seen in the secondary images). The last image (d) shows an over-constrained situation in which there is not enough land to allocate the whole population and the desired combination of values for the variables is infeasible.[23]

All variable changes described above are conducted on values stored on a two-dimensional grid of cells. Each cell holds the values for the set of variables. These values can be changed locally or globally. When this happens, the system iteratively updates the other variables to find a stable solution. To reduce the load for the user and because not all variables are easy to change intuitively, some variables can be changed directly, such as population and job count, while others like land value and land use are computed by adaptive algorithms.

The behavioural variables used in the simulation are population count, job count, accessibility, and land value. The latter two cannot be edited manually. For the placement of the geometry the variables roads length, average tortuousness, building volume, terrain elevation, parcel size, and land use are used. The latter two variables cannot be edited directly, but are computed algorithmically. The return to an equilibrium after a change mostly depends on moving jobs and population by moving the agents to new locations.

Although most of the construction of the geometry is automatic, the user has to sketch the highways. The construction of the secondary and tertiary roads is done using seeds that can be freely placed. The first set of seeds is generated based on a user defined pattern (either radial or raster), the placement of the highways, and on the distribution of population and jobs. Each seed is an intersection of the secondary roads connecting them. The seeds for the tertiary roads are generated along the secondary roads. The streets grow further from these seeds.

In the case of a radial pattern three or more road segments depart from an intersection at equally spaced angles. The raster pattern lets depart up to four road segments almost perpendicular to each other. The raster pattern does not have to be complete and can therefore contain

dead-ends. The placement of the road segments in both patterns is guided by the elevation of the terrain and the distribution of population and jobs.

When the road network has been formed, each cell surrounded by roads is geometrically partitioned. For this, the strategy of Parish and Müller [14] is applied. However, the number of patches in a cell differs per area. It depends on the size of the cell, the total count of population, and the jobs contained within the cell. If the needed number of lots cannot be reached, the height of the buildings is increased.

2.2 Land Use Models

So far the papers discussed in this chapter were focused on creating a realistic road network and building geometry. The type of buildings created were mostly commercial and residential buildings mixed throughout the city not taking into account the functions of those buildings. The placement of a particular type of building in a city is not random and therefore more recent work focuses more on land use models.

The approach in Subsection 2.2.1 for example generates an urban layout of roads and land use per area based on high-level input from the user. A more advanced method is described in Subsection 2.2.2 which covers the simulation of land use over time including building geometry, roads and traffic. The last subsection (Subsection 2.2.3) shows a method based on agents. The agents build the city over time while simulating the land use and road construction.

2.2.1 Urban Land Use Layouts

In [13] a method is proposed that has the main focus on the land use within a city. The method procedurally creates layouts of large cities using only high-level user input in a matter of seconds. Due to the simple input, no expert knowledge is needed for the creation of the city.

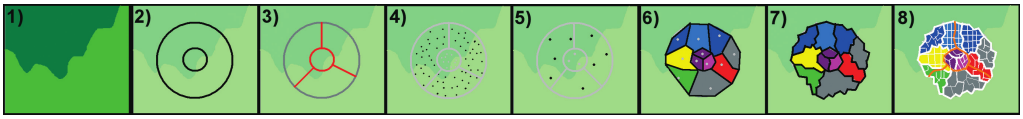


Figure 2.10: The creation of a city. [13]

Figure 2.10 illustrates how the city layout is generated. The application starts with a terrain map (1) and the user input of location, city size, and some more parameters to control the generation process (2). After these first steps the system starts generating districts. Eighteen district types are used in the model: 3 residential, 2 industrial, 2 commercial, transportation nodes, green spaces (parks), and 8 special core types (such as cathedrals or town squares). For the placement of the districts, a road network is created. First highways are generated in a radial pattern (3) after which candidate locations for the districts are generated (4). Then the best locations are chosen (5) and a Vonoroi diagram is produced using the chosen locations (6). This Vonoroi diagram dictates the secondary road network and some noise is applied to make it look more realistic (7). To finish the road network, known methods such as [14] can be used to create the street networks in the districts (8).

The positioning of the district is influenced by five parameters: type of neighbouring district, terrain type, area within the city, distance from rivers, and distance from highways. For each district that has not been allocated yet, the suitability is computed for all possible locations left using the mentioned parameters. These parameters are weighted according to their importance for the city type (western European city, such as the one in Figure 2.11, or Northern American city). The district is assigned to the location that has the highest suitability. By repeating this, all districts are added to the city and the land use layout of the city is generated.

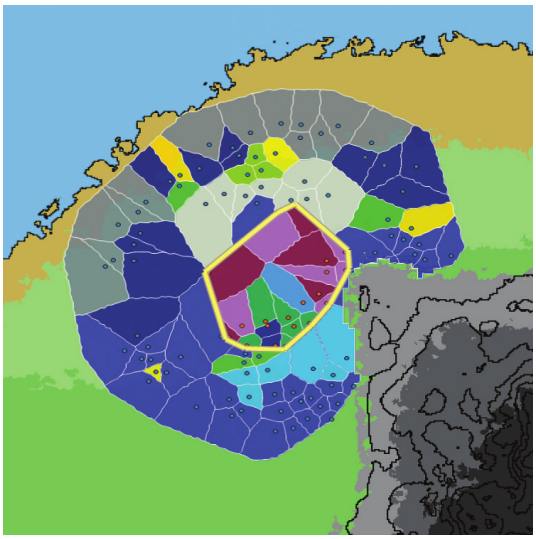


Figure 2.11: A western European city. [13]

2.2.2 4D Cities

All previously discussed papers focus on the final city layout and its generated geometry. Often first a road network was produced followed by the creation of land use, either just a label or complete geometry and textures. Weber et al. in [9] describe a method that is related to [14], but has many improvements. It simulates the land use, road network construction and building geometry over time. In each step of the evolution from village to city, the produced model is realistic and complete (both a road network and building geometry).

In contrast with other systems that simulate the land use and the creation of a road network, Weber et al. also simulate building geometry. A more common approach for land use simulation is using a set of agents, discussed in the next subsection. In that approach a regular grid of land patches is used. The application described in [9] includes realistic geometric configurations that do not rely on such a grid. Furthermore it aims at fast simulation and interactive speed.

The system starts with an environment defined by the user, this includes maps for elevation and water. The user can also specify the initial urban layout. This can be a single street, or an already developed city. Furthermore the land use type definitions have to be specified. The system has to know how to assign a value and land use to an area.

Based on this initial data set the application starts by expanding the network of main roads according to the city centres and the defined growth rate. The road expansion is inspired by [14], but instead of using an L-system, it uses grow seeds to the expansion faster. New street segments grow from the nodes in a temporally meaningful order. For each node, three values have to be chosen for expansion: direction (straight, left, right), deviation, and length. The values for deviation and length are chosen based on the active pattern (organic, grid, or radial). These patterns can be imposed by the user via painting weights or via the chosen land use type. The snapping rules as defined in [14] still apply and help the creation of patterns.

During the road expansion the main roads will form quarters, see Figure 2.12. When such a quarter is created, the dominant land use type for that quarter is estimated based on the land use type definitions. Based on this type a secondary road network will be constructed. In turn these minor streets will surround blocks and the blocks are then subdivided into lots. For both the blocks and the lots the land use is estimated. The land use of a lot and the placement with respect to the road are of influence on the calculation of the zoning regulation within the lot. This zoning regulation describes the volume wherein the building on that lot will be constructed.

An important feature of the created application, is the traffic demand model. This model is used to determine if a new road segment should be built and if existing roads should be wider to

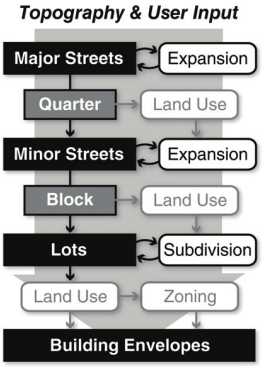


Figure 2.12: The structure of the system from major street expansion to building generation. [9]

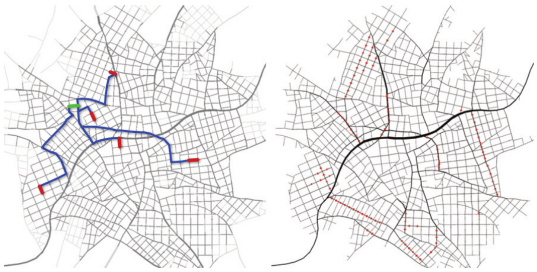


Figure 2.13: The number of trips (blue) starting at a street (green) depends on the number of residents on the number of residents (left) and the overloaded streets after the simulation (right). [9]

handle the traffic load. Since existing traffic demand models did not offer the detail, or needed too many external variables to run, a new traffic demand model is proposed.

This simulation calculates the traffic load for each road segment by stochastic sampling. Both proposed and existing road segments are included in this simulation. The algorithm distributes a discrete number of trips in the road network by selecting a start and end point for each trip. Then the shortest path is calculated and the traffic value of all road segments visited on the trip are updated according to the volume of the trip. The number of trips starting at a street, see Figure 2.13, depends on the residents present at lots at that street. This number depends on the land use type and value.

When the simulation has finished updating the traffic load values, the system looks for proposed roads that have enough traffic to be constructed. If the traffic value of such a road exceeds a user-defined threshold, the road segment is constructed. Also the width of streets that need more capacity are updated.

The goal of the land use simulation is, unlike many agent-based approaches, to simulate a system that strives towards better land use configurations, instead of finding the optimal use of land. The user of the application can define a set of possible land use types (such as residential, commercial, and industrial). These land use types are defined as a combination of land use evaluation functions. These define how well the land use configuration matches the user-specified percentages of land use. An example of a growing city with changing land use over time is shown in Figure 2.14.

Figure 2.14: The growth of a city after 10, 30, and 50 years (simulated in 58 seconds). [9]

In the update cycle random lots are selected and their land use type is changed if it increases the land use value enough. This simple algorithm is fast, and is configurable for changes over time of demands and planning strategies. The land use value, used as a decision basis in the algorithm, is affected by both global goals (such as the user-specified land use percentages) and local goals (such as clustering, neighbourhood land use and traffic load of nearby streets).

2.2.3 Agent-based Modelling of Urban Land Use

Most methods described above all use a global authority to control the growth of the city. The agent-based approach by Lechner et al. [25], which is later extended in [26], releases agents in a city to let the city grow locally instead. Each agent has a specific task; one might be a road extender, while another has the task to build schools. The agents themselves have a simple rule set, but due to differences between the rule sets of agents, the agents together can perform complex behaviour such as the construction of a city.

Agents have the advantage over L-systems of forming a more robust system [25] and are easier to change and extend for more cases (cultures, eras, unique buildings, etcetera). Another advantage is that the created land usage and road network has a meaning for the city. Though this approach does take over much work of the designer which might not be desirable for artist and architects.

The goal of the approach by Lechner et al. is to generate convincing and plausible cities by capturing developmental behaviour. Furthermore the cities generated should look compelling in every stage of the growth of the city. Instead of focusing on creating geometry around a generated road network, the project focuses on the land usage and building distribution in the city.

These agents interact directly with their environment, which consists of a set of rectangular patches. Each individual patch can also follow behaviours, but cannot move. By interacting with the patches, the agents can interact indirectly with each other. Although this does give realistic results, it is still slow and does not generate building geometry. A city of roughly 23 square kilometres, such as the one shown in Figure 2.15, takes several hours to generate.

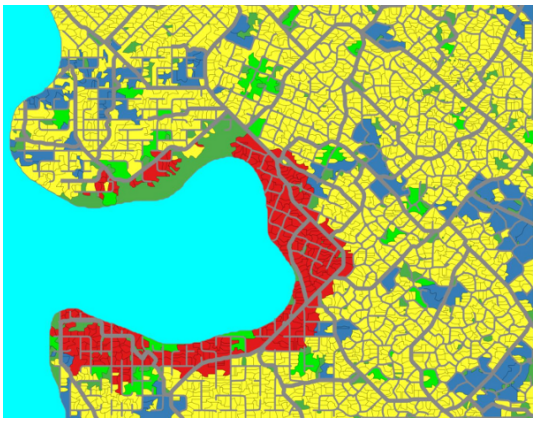
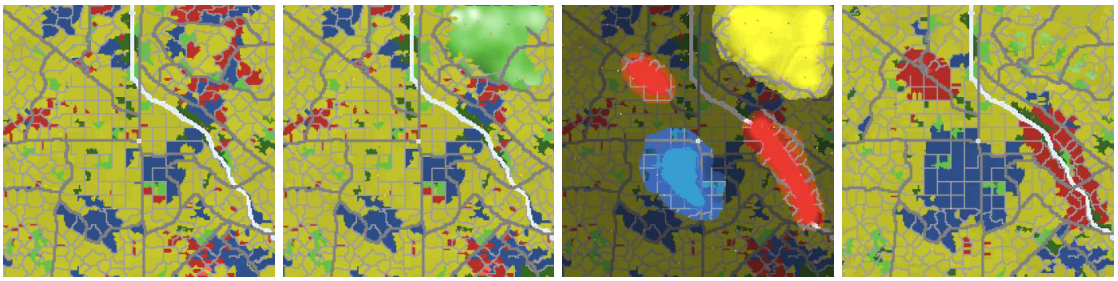


Figure 2.15: A vectorised output from a procedural city model with residential (yellow), commercial (red), industrial (blue) and park (light green) regions. [26]

The only input the user should give to the application is a terrain map. This map consists of three elements: land (green), water (blue) and a growth centre (grey). The shade of green determines the elevation of the land. Based on this terrain map, the simulation can start building a city starting at the growth centre. During the simulation the user can change parameters to tune the behaviour of the agents and by that the resulting city. The user can for example change the possible deviation of newly created roads to have a raster pattern used or a more organic road network.

The application also offers artists the possibility to pause and alter the course of the simulation freely. One important way is the painting interface that enables the use of removing unwanted development by erasing patches. Another feature is painting honey that will attract the development of a certain property. In Figure 2.15 honey was painted in the harbour area to attract commercial development. The user can also paint reserved area to prohibit development in a specific area. Another way of changing the development is by changing the parameter values and weights used by the agent to determine land values and road construction. These can be applied locally or on a city wide scale. The use of a few features is shown in Figure 2.16.



(a) (b) (c) (d)

Figure 2.16: Steering the simulation, starting with (a), by removing an undesired commercial and industrial cluster of the city (b) and painting honey (c). The result is shown at the right (d). [26]

The growth of the cities in the application highly depends on the values of the patches and the added value of changing the patch done by the agents. The agents will only replace a patch if it results in a higher land value. The value of a patch or collection of patches (parcel) depends on the weight inherent to the property developer type. For residential developers, elevation and being close to other residential areas gives higher land values, but industry tends to cluster near the main roads and water. Figure 2.17 shows this process.

Figure 2.17: The growth of a city from a few roads and growth centres around road junctions and near water. [26]

The application has four types of developers, namely residential, commercial, industrial and park developers. They create parcels of corresponding patch types. The residential patches cannot be converted directly to industrial and vice versa. Park patches cannot be converted at all. Those are created slightly differently. The creation of parks needs a minimum city size and population and depends on the total amount of already created park patches. Furthermore, these can only be built on unused land and are generated using a flood fill operation instead of the method described above.

These developer agents wander around the terrain and search for unused cells. The agent chooses a random target parcel size (from a range specific for the type of developer) and starts creating a parcel starting from the found patch and adding patches while moving away from the road. When half a block length has been travelled, the agent starts widening the created strip to the sides, while keeping neighbouring parcels intact. When the agent is not able to add the chosen number of patches to the parcel, it tries to merge the parcel with neighbouring parcels of the same type. If then still the preferred parcel size has not been reached, the creation of the parcel is aborted.

The roads in the city are generated by three types of developers. The tertiary road extender which makes sure parcels can be reached by road, the tertiary road connector that makes sure the roads are properly interconnected and the primary road developer. This primary road developer is unlike the other two less bound to the grid patterns and creates a more organic pattern. Its goal is to connect the centre of the city to its surroundings and ensure that primary roads are not

always nearby to parcels. The primary road developers try not to get close to other primary road developers and avoid recently visited patches to prevent overly dense road structures.

Figure 2.18: A visualisation of one of the cities using SimCity 3000. [26], [27]

For the visualisation of the city, the generated maps were imported by the authors into Electronic Arts' SimCity 3000 [27], see Figure 2.18. SimCity 3000 comes with a library of buildings that is used for this visualisation.

2.3 Discussion

The previous two sections covered methods proposed for generating realistic urban areas. Many techniques try to recreate typical road patterns in the generated cities. This can range from using just a rectangular grid or let the user define patterns and their locations and generate a graph between the crossings in the selected pattern [11], [12]. Other techniques tend to focus more on the functional background of a city and how the land use within it changes over time.

2.3.1 Road Patterns

The technique using road patterns described first (Sun et al. [11]) is fast and creates a realistic road network, but lacks building geometry. The artist has some freedom in shaping the road network by setting the types of patterns to use and their position. The roads are then generated in between the crossings defined by the chosen pattern. The second technique (Kelly et al. [12]) also grows roads between crossings, but also adds building geometry and local pattern differences. The technique is fast enough to generate a complete city in real-time. Just like the first technique, the created city has no history and does not evolve over time.

The system described in [12] uses L-systems to create a secondary road network, inspired by the CityEngine [14]. CityEngine cannot produce cities in real-time, and is therefore not as interactive as [12], but the created cities are realistic. A remarkable property is the meaning behind the construction of the road network. The roads try to connect parts of the city having a high population density. Buildings in generated cities depend on the land use chosen by the user. The created geometry is either a skyscraper, commercial, or residential.

To have a more interactive design tool, a faster technique than L-systems was needed and therefore tensor fields were used to guide the creation of a road network [20], [21]. The user can manipulate a tensor field using many tools like brush strokes and rotations. The road network is created by tracing lines in the tensor field which creates a raster at first that can be bent or broken to generate any pattern. This method offers fast and interactive road network design. It does not offer historic context of the city, nor a land use simulation.

Another fast and interactive approach was given by Vanegas et al. [23]. This approach moves jobs and population to find an equilibrium in the variables saved on a 2D grid. Primarily based on these variables the roads are generated. It offers the user a tool to iteratively create a large urban area for which the geometry and the behavioural modelling are linked. The whole process can be completed in just a few minutes. This approach is interesting due to its fast simulation of how a city will look like based on the terrain and how the agents move within the city, though it does not simulate over time and requires a lot of input from the user.

2.3.2 Land Use Models

The methods discussed above focus on the creation of known patterns in roads, a few techniques however focus on the land use within cities. Three different land use related techniques have been discussed. The first proposed in [13] aims specifically at generating a realistic layout of the different land uses. The primary road network is radial, and the secondary network is created by using the edges of a Vonoroi diagram as roads. The secondary roads also mark the edges of a land use area. A tertiary road network can be created within the land use areas using a system like [14]. This technique can generate the layout in seconds using just a minimal input from the user. The land use in the city has a meaning and is related to the land use in neighbouring areas. However, this system does not include buildings.

The technique proposed in [9] does include geometry and has many similarities with [14]. It simulates the land use, building geometry and road construction over time which results in visually realistic cities. It also simulates the traffic in the city to determine if roads should be built or replaced. The system can simulate a year of development in a second. The controls can be difficult since a good configuration of parameters can be hard to define for a non-expert.

The last described approach simulates land use using agents [25], [26]. Each agent in the system has a specific task and by simulating them over time a city is created based on simple rules. Every construction operation has a meaning and the growth is simulated over time. The system can easily be extended by adding new agents with new rules. Though the major disadvantage is the speed. It can take hours to create big cities.

Chapter 3

Approach

Most of the procedural techniques for creating virtual urban areas described in the previous chapter are capable of creating visually realistic content. Some methods take hours to create an urban model, while other approaches only need a few seconds. The resulting urban areas, however, lack meaning and in many cases they also lack history. Some content generators included meaning into the creation process through land use simulation. However, the used static generation processes were not very generic and were mostly controlled via unintuitive parameters.

The goal of this project is to create an approach that incorporates semantic information and history in the creation process of a virtual urban area. The generated model should be suitable for use in a semantics-based game. It should therefore obey the rules of the semantic game world it is created for. How this abstract idea is translated into a description of a simulator, is explained in Section 3.1. The second section elaborates on the relations mentioned in the first section and describes in detail how the semantic information is used to simulate urban area development.

3.1 Semantics of Urban Areas

To create a procedural method for generating virtual urban areas that are suitable for a semantic game world, the semantics of the game world should be incorporated in the generation process of the urban areas. The resulting models should reflect the semantic information specified by the game world. The diagram in 3.1 shows how this idea is split up and how the components relate to each other to form a description of a simulator. The yellow blocks represent a high-level description and the actual game world. The blue blocks are part of the government instance that guides the simulation and red blocks relate to the actual patches that are used in the simulation. The blocks and their relations will be described in this section.

3.1.1 Terrain

A major part of the semantic game world is the terrain map. The features of this terrain, such as mountain ridges, rivers, and forests, define how the terrain looks. These features are not only of visual importance, but they also contribute in the meaning of the terrain. They affect other objects on the terrain. For example, if a city is located near a river, the river provides the city with plenty of water and also offers a possibility for transport. If the city happened to be near a mountain ridge, the rocky areas could supply the city with stone if the citizens would build a quarry in the rocky areas.

The resources offered by natural terrain features attract people that need those resources. This will result in a settlement near those resources to harvest them. So, the locations of the features also influences where a settlement is located. A settlement in itself can become a place that attracts people, by offering housing and other facilities wanted by people. as a settlement grows, it might reach natural borders, such as a steep cliff or a river. When lots of the settlement

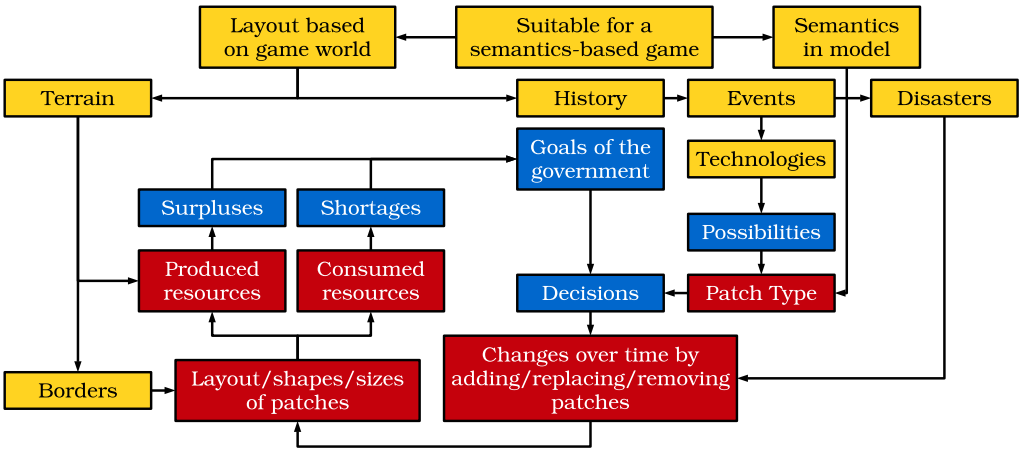


Figure 3.1: This diagram how the goal of the project is related to the features of the created simulator.

are placed near the borders, the borders of the lots, on which the buildings of the settlement are located, will follow the natural borders. In this way, the borders of terrain features can shape a settlement and its lots.

3.1.2 History

The shape and layout of the terrain features are not the only important aspects for the development of an urban area. The history of the game world also plays a big role. The events that happened over time shaped the world to what it is. This feature does not occur in other procedural urban area generators. Events change the environment of the urban area and allow for a content generation process that is less static. Events can include e.g. disasters that destroy parts of the urban area, and technologies that change the possibilities of a civilisation. The former shapes the urban area directly by damaging and vanishing buildings, while the latter allows for creating new types of buildings and changes how the urban area is regulated. For example, the invention of reinforced concrete made it possible to create taller and stronger buildings, like skyscrapers, that would not have been possible without it. Nowadays most modern cities have tall buildings that were constructed with reinforced concrete.

3.1.3 Government

During the development of the urban area, decisions have to be made about which of the possibilities are applied to keep the civilisation running. The instance responsible for making those decisions will be referred to as the government. The decisions made by the government cannot only be based on what is possible, but should also reflect the desires or goals of the government. In turn, these goals are based on what is needed to sustain the urban area. The people in the urban area, for example, need water and food to survive. They also need a place to find shelter and a place to rest. For that reason, they also need houses and to build houses, building materials are required. This example shows how the presence of people requires resources to be available. If more resources are needed than that there are produced, shortages will be created. The government will adjust its goals to create more facilities that produce the needed resources. It will try to find suitable locations for the facilities, near the locations where the resources are required. In Figure 3.1 and this chapter, the lots containing the facilities are referred to as ‘patches’. The semantic information of the patches is referred to as ‘patch types’.

The location, size, and shape of the patches is affected by the terrain features and other facilities placed in the urban area. The following example will try to clarify this. In general, farms that grow crops are unlikely to be built in deserts or rocky areas, since crops do not grow well in those areas. These farms will be built on fertile ground. To operate, the farm also needs resources such

as water. It is therefore likely to be located near a river, for example, and the people that work on the farm need housing, which implies that a house will be situated near the farm. The size of the farm and its type will rely on the amount of food that is needed, how many resources are available, and what the technologies allow to be built.

Over time, the government will add more patches. This can result in producing more resources than needed. These surpluses will, as the shortages did, influence the goals of the government. It will trigger the government to add patches that consume the resources, or it will replace patches that produce the resources. The government could also choose to remove the unwanted patches to lower the production of certain resources. This process of adding, replacing, and removing patching helps to find a balance between production and consumption of resources.

3.2 Land Use Development

The government manages, based on its goals, patches that produce and consume resources. This leads to surpluses and shortages which affect the goals of the government. This loop is the basis of UrbSim. It simulates the land use development over time while incorporating the history and semantics of the game world. Each simulation cycle, the goals and stock (surpluses and shortages) of the government are updated (Subsection 3.2.1) to meet the current situation. Then all patches are updated, which includes the collection and distribution of resources, see Subsection 3.2.2. These resources are stored in the stock of the government. Old patches that are heavily damaged are removed and new patches are created (Subsection 3.2.3-3.2.6). Also existing patches are evaluated for their contribution to the area. If another type of patch would give better results at that location, the patch is replaced. This can be used for both updating old patches and changing the local area. This is part of the patch maintenance, see Subsection 3.2.7. After all updates of the urban area are done, technology and disaster events are triggered. The technology event will enable and disable patch types for the coming cycles and the disasters might damage existing patches. These events are not described in more detail in this section.

3.2.1 Updating Stock and Goals

In each cycle the stock of the government is degraded. Both the surpluses and shortages are reduced since the patches that needed the resources might have been removed or replaced and the quality of the resources in stock might degrade. Food for example will not be of any value when it is kept for a long time and starts to rot. The stock reduction is done by multiplying the old stock value $s_{r,t-1}$ of resource r at time $t-1$ by $1-ds_r$ where ds_r is the degradation speed of the resource, see Equation 3.1.

$$s_{r,t} = s_{r,t-1}(1 - ds_r) \tag{3.1}$$

To help the user even more to get the simulation find an equilibrium between the production and consumption of resources, the goals of the government are based on the surpluses and shortages for each resource. The calculation of the new goal value $g_{r,t}$ at time t for resource r is given by Equation 3.2. The new goal value depends on the finite market restrictions ($fmr_{r,t}$, see Equation 3.3) and the previous goal value for that resource ($g_{r,t-1}$). The weight w_g determines how much of the old goal value is retained and how much the current stock influences the new goal value. This allows for more fluent changes of the goals rather than rapid changes of the government's strategy.

As will be explained in more detail in Subsection 3.2.3, the selection of patch types depends on the goals of the government and the production values of the available patches. The production values are compared with the goal values to determine which patch types are preferred. If the goal values however grow to multiples of the maximum production values offered, the differences between the patch types vanishes with respect to the goal. For example if the goal value is 1000 tonnes of food, there is not much difference between a patch type that consumes 1 tonne of food

and a patch type that produces 5 tonnes of food. On the other hand, if the goal value would be 6 tonnes, the latter patch type would have a much higher value than the first one.

Although this makes sense, it will lead to a lack of proper decisions to reach the goal value. Since the difference in production values is no longer significant, the government will assign the same value to the patch types. To overcome this, the goal values are capped at the maximum and minimum production values as shown in Equation 3.3. The capped goal value or finite market restriction $fmr_{r,t}$ is based on the stock value $s_{r,t}$ for resource r. As long as the shortage or surplus is between the lowest production value ($p_{r,min}$) offered by the available patches and the highest production value ($p_{r,max}$), the finite market restriction is the same as the negated stock value. Otherwise it is capped by the mentioned limits.

$$g_{r,t} = (1 - w_g) fmr_{r,t} + w_g g_{r,t-1} \tag{3.2}$$

$$fmr_{r,t} = \max(p_{r,min}, \min(p_{r,max}, -s_{r,t})) \tag{3.3}$$

3.2.2 Patch Update

When the existing patches are updated, the resources are collected from the patches. The production of a patch is equal to its default production multiplied by its area, the inverse damage level and the time that past since the last update. So less damaged and larger patches will produce more than smaller and damaged patches. During the update also the damage level of the patch is increased to mimic ageing. The speed of degradation of patches can be different per patch type and can if needed even be zero.

The update of patches not only includes the gathering of resources but also consuming resources by the patches. This is done via negatively valued resource productions. The calculation of the net local production $ls_{x,r}$ for patch x and resource r is given by Equation 3.4. A negative value indicates a shortage of resource r at the location of the patch and a positive value indicates a surplus. In this equation the production of all patches in the neighbourhood (N) is cumulated. Each production $p_{n,r}$ (see Equation 3.5) is weighted by the distance weight $w_{d,n}$ (see Equation 3.6).

$$ls_{x,r} = p_{x,r} + \sum_{n \in N} w_{d,n} p_{n,r} \tag{3.4}$$

The production ($p_{x,r}$) of the patch x for resource r depends on the health of the patch ($1 - dmg_x$), the area of the patch a_x, the envisioned size ae_x, and the production value $pv_{x,r}$. The damage level influences the production of the patch negatively as Equation 3.5 shows; heavily damaged patches produce less resources than equal patches with less damage. To translate the patch type its production value $pv_{x,r}$, the production is also scaled by its size divided by the envisioned size. The latter is the area ($width \cdot height$) set by the description of the patch type.

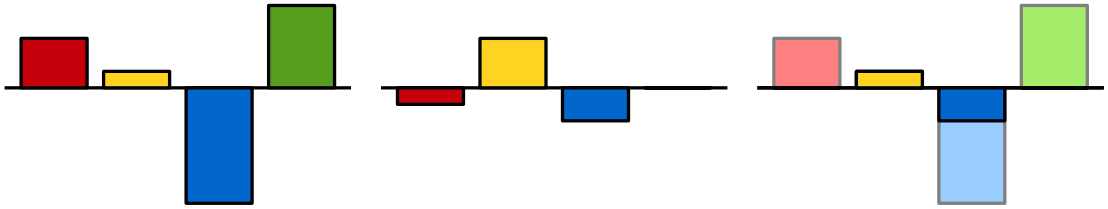
$$p_{x,r} = (1 - dmg_x) \frac{a_x}{ae_x} pv_{x,r} \tag{3.5}$$

The distance weight of patch x is equal to one and is therefore not included in the equation. The distance weight for the neighbouring patches, elaborated in Equation 3.6, depends on the radius of influence $r_{i,n}$ and the bounding radius $r_{b,n}$ of neighbour n. When the centre of the patch is within the bounding radius, the weight is one. If it is outside the radius of influence, the weight and therefore the contribution of the production value is zero. When it is between these radii, the weight is closer to one when the Euclidean distance is smaller. This results in neighbours close by to have more effect on the local production $ls_{x,r}$ than neighbours that are further away.

$$w_{d,n} = \begin{cases} 1 & d_n \leq r_{b,n} \\ \frac{d_n - r_{b,n}}{r_{i,n}} & r_{b,n} < d_n < r_{i,n} \\ 0 & r_{i,n} \leq d_n \end{cases} \tag{3.6}$$

To determine what part of the surplus and shortage is caused by the tested patch x, and thus what is added to the global stock, the production of the patches $p_{x,r}$ is tested against the local production $ls_{x,r}$ (Equation 3.7). If $p_{x,r}$ is negative and $ls_{x,r}$ is positive, or vice versa, the contribution of $p_{x,r}$ is equal to zero. If the values are both negative or positive, and the value of $p_{x,r}$ is closer to zero, this value is added to the stock, otherwise the value of $ls_{x,r}$ is added to the stock, see Figure 3.2.

$$s_{r,t} = s_{r,t-1} + \sum_{x \in X} \text{sign}(ls_{x,r}) \min(|ls_{x,r}|, \max(\text{sign}(ls_{x,r}) p_{x,r}, 0)) \tag{3.7}$$



(a) The local shortages and surpluses ($ls_{x,r}$).

(b) The production values of patch x ($p_{x,r}$).

(c) The contribution of $p_{x,r}$ to the global stock.

Figure 3.2: The bar graphs represent the production values. The contribution of the production values of patch x to the global stock is limited by the local shortages and surpluses. In this example only the yellow and blue resource are added to the stock.

When there are not enough resources available in the local neighbourhood to maintain a patch, the patch will be damaged proportional to the shortage, see Equation 3.8. By doing so, also the value of the patch decreases. It ensures that patches that are consuming too much or do not fit in the local area are more likely to be replaced or removed.

$$dmg_{x,t} = dmg_{x,t-1} + w_{dmg} \sum_{h \in H} \frac{|w_h| \max(ls_{x,h}, \min(p_{x,h}, 0))}{\sum_{r \in R} \min(p_{x,r}, 0)} \tag{3.8}$$

The amount of damage done is a positive value between zero and one. The damage is based on resources for which the patch causes a shortage (H) and thus both the production $p_{x,h}$ and the local production $ls_{x,h}$ are negative. The value of these two that is closest to zero is multiplied with the absolute weight w_h for the resource h to let more important resources have a higher impact on the damage. The product is divided by the total negative production of the patch to let shortages of heavily consumed resources do more damage than those that are used in smaller quantities. The sum of these fractions is weighted with the damage weight (w_{dmg}) and added to the old damage value of the patch ($dmg_{x,t-1}$). The damage weight is introduced to give some control of the maximum amount of damage done each cycle. This gives an option for not directly destroying a patch when there is a shortage of needed resources, allowing the system to restore the stock values and maintaining the patch.

If the patch does not consume anything, the negative production is zero in which case the patch is not affected by shortages. The damaged patches that exceed the damage threshold of one are removed from the area. The borders are updated to allow new patches to be build on the free space that became available.

3.2.3 Patch Type Selection

The patch update is followed by the creation of new patches. To determine which patch type should be built, all available patch types are tested for their production values and creation costs against the current goals of the government. As Equation 3.9 shows, the value of a patch type is a weighted sum of the value for the costs ($v_{t,c}$) and the value for the production ($v_{t,p}$). The weights

w_c and w_p represent a preference of the government for costs and production respectively. To be able to control the percentages of patches that are of a certain type, the value is also weighted by the appearance weight $w_{t,a}$. The calculated value is between zero and one, where a value between zero and a half means a negative effect on the available resources and a value above a half indicates a positive effect on the stock.

$$v_t = w_{t,a}(w_c v_{t,c} + w_p v_{t,p}) \tag{3.9}$$

The values for costs (Equation 3.10) and production (Equation 3.11) are calculated by comparing the negative production or consumption ($c_{t,r}$) and positive production ($p_{t,r}$) value, with the goal value (g_r) for each relevant resource r. Resources for which the goal value is zero are not taken into account. In the absolute calculation the difference between the goal and production or cost is compared with the absolute value of the goal. So the closer the values are to the goal value, the higher the added value is. Since the costs are positive numbers indicating a negative impact, the costs are added to the goal in Equation 3.10 rather than subtracted from it. The resulting value is bounded by zero and one, summed up with all values for all resources and normalised by the number of resources tested. Note that the sets of resources (R_p and R_c) used in the calculation for the production and costs value can differ. For example, a patch might have a cost value for wood, but does not have a production value for wood. The resource wood would not be part of the set of resources R_p, but it would be part of R_c. The normalised sum is subtracted from one to give a higher value to productions and costs closer to the goal.

$$v_{t,c} = 1 - \frac{1}{|R_c|} \sum_{r \in R_c} \min\left(1, \frac{|g_r + c_{t,r}|}{|g_r|}\right) \tag{3.10}$$

$$v_{t,p} = 1 - \frac{1}{|R_p|} \sum_{r \in R_p} \min\left(1, \frac{|g_r - p_{t,r}|}{|g_r|}\right) \tag{3.11}$$

To select a patch type, the values are used in a fitness proportionate selection procedure [32] which is also known as roulette wheel selection. This kind of procedure is often used in genetic algorithms for selecting individuals from a population to generate offspring [33]. The chance for selecting a patch type is proportional to its value compared to the other calculated values.

The fitness proportionate selection procedure first creates an array of increasing values. The first slot in the array is set to the value of the first patch type. The second slot in the array is the sum of the first slot and the value of the second patch type. The third slot is the sum of the second slot and the value of the third patch type. This continues until all patch type values have been used. A random number between zero and the value of the last slot is chosen as the input of a binary search for the index of the slot in which that value lies. The resulting index points to the patch type that is selected. This patch type will be used as the basis of location selection in the next subsection.

Note that this procedure could result in selecting a low valued (a value less than one) patch type. Although this has a negative effect in this local time span, it could have a positive effect on the overall process. This prevents the algorithm from getting stuck in a local optimum.

3.2.4 Location Values

Before a new patch can be created, a suitable location has to be selected to the patch. First, a fixed amount of random points is selected. These random points can only lie on ground types on which the patch can be built. For each of these points, a suitability score is calculated by adding up all relations scores that determine how the patch is attracted or repelled. The suitability score for a location (s_x, see Equation 3.12) is a weighted sum of the normalised production value nps_x, the number of artificial patches within the influence radius of the patch (N_x) normalised by the maximum value found for N_x: N_{max}, and a penalty (py_x) for not having any border within its snapping range. The snapping range is equal to half the diagonal of the envisioned patch: $\frac{1}{2}\sqrt{(width^2 + height^2)}$, where $width$ and $height$ are the desired dimensions of the patch.

The first part of the suitability score equation, weighted with the production weight w_p, is transformed to emphasise the difference between negative and positive values. The last two steps encourage the algorithm to select points that are near artificial patches to promote clustering. If $N_{max} = 0$, then the value for the fraction $|N_x|/N_{max}$ is set to zero. Otherwise its value between zero and one is weighted with the neighbours weight w_n. To ensure the value for s_x is positive and can be used in a fitness proportionate selection, all values below zero are set to zero and are therefore neglected in the selection process.

$$s_x = \max\left(0, w_p \operatorname{sign}(nps_x)\sqrt{|nps_x|} + w_n \frac{|N_x|}{N_{max}} - py_x\right) \tag{3.12}$$

The most important part in Equation 3.12 is the normalised production score nps_x which represents how the production of the surrounding patches contribute to the score of the location (x). The normalised production score is calculated by summing up all weighted and normalised production values in the neighbourhood, as shown in Equation 3.13.

$$nps_x = \sum_{r \in R} w_r \min\left(\max\left(\frac{sp_{x,r}}{sp_{max,r}}, -1\right), 1\right) \tag{3.13}$$

In this equation R is the set of all resources which are relevant to the chosen patch type. For each of these resources a weight, w_r, has been set. The weight determines if the patch type is attracted or repelled for this resource. These relations are stored as a rational number between -1 (repelling) and +1 (attracting). The weighted sum of productions of resource r in the neighbourhood is denoted as $sp_{x,r}$, see Equation 3.14. The biggest value found for $|sp_{x,r}|$ over all locations is $sp_{max,r}$ and works as a normalising factor. This ensures that nps_x, given that all w_r are between -1 and +1, is also between -1 and +1.

$$sp_{x,r} = \sum_{n \in N_x} w_{d,n} p_{r,n} \tag{3.14}$$

The value of the weighted production sum $sp_{r,x}$ is found by adding up all production values for resource r of the neighbouring patches N_x. The production values are multiplied by a weight $w_{d,n}$ which is based on the distance from the tested location to the neighbouring patches. The calculation of this value was given in Equation 3.6. Instead of using the centre of a patch, the tested location is used to calculate the distance weight.

The scores for location selection are used in a fitness proportionate selection just like the patch type values in Subsection 3.2.3. The selection procedure returns the index of the chosen score and location. The location that corresponds to the found index is used as the starting point for creating the patch, see Subsection 3.2.6.

3.2.5 Location Selection Example

To show how the mathematics from the previous subsection work, a detailed example is presented in this subsection. This example starts with three randomly chosen locations and three existing patches, as shown in Figure 3.3. On one of the three locations a house should be constructed. A household requires food and water, which results in positive weights: $w_{food} = 0.60$, and $w_{water} = 0.40$. In this case the weights add up to one, but this is not necessary for the final selection as long as the sum of the weights is between -1 and 1.

For each of the patches the production values of both food and water are presented in Table 3.1. Also the distance weight for each location to each patch is given (Table 3.2). Location C is not in range of the house patch and the well patch. Therefore it will only be affected by the farm. Location A on the other hand is in the range of both the farm and the well, which results in both a production of food and water for location A. Location B is not near the well and will not be supplied with water.

To make sure that the location with the best circumstances (food and water are produced near the location and artificial patches are near) receives the highest score, all productions are

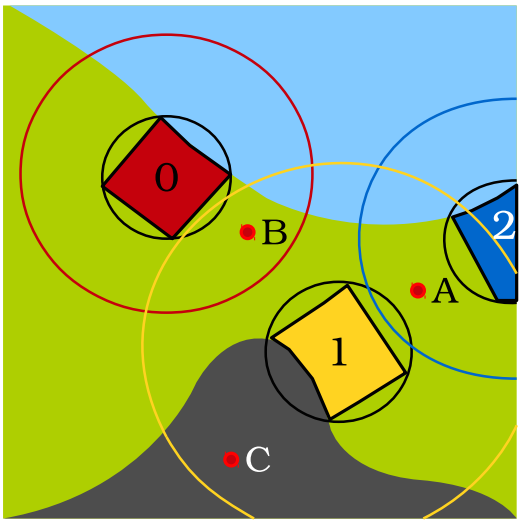


Figure 3.3: Three proposed starting locations (A-C) for which the score should be calculated, based on the surrounding patches (0-2). For the patches the bounding circles (black) and the influence range (red, yellow, and blue circles) have been drawn. The ocean, at the top of the image, does not offer any resources in this example.

n	p_{food}	p_{water}
House (0)	-10.0	-10.0
Farm (1)	100	-40.0
Well (2)	0.00	50.0

Table 3.1: The production values per patch type.

n	A$w_{d,n}$	B$w_{d,n}$	C$w_{d,n}$
House (0)	0.00	0.70	0.00
Farm (1)	0.80	0.50	0.20
Well (2)	0.60	0.00	0.00

Table 3.2: The distance weights for each proposed location.

added up, normalised and weighted (Table 3.3). The cumulative food production at location A for example is $-10.0 \cdot 0.00 + 100 \cdot 0.80 + 0.00 \cdot 0.70 = 80.0$. This is also the biggest value found for food production among the three locations ($max_{food} = 80.0$), and so the normalised score for food for location A is equal to one. When the food weight ($w_{food} = 0.60$) is applied, the final weighted score for food for location A is $p_{food} = 0.60$. Although there is some water production near this location, the consumption of water by the farm and the distance to the well cause the value for p_{water} to be negative. Note that the normalising factor for water is the absolute biggest value, which is 27.0, and does therefore not promote the negative value. Only if the weight for water was negative, a negative production score would have a positive effect on the final score.

	A		**B**		**C**	
	p_{food}	p_{water}	p_{food}	p_{water}	p_{food}	p_{water}
$sp_{r,x}$	**80.0**	-2.00	43.0	**-27.0**	20.0	-8.00
Normalised	1.00	-0.07	0.54	-1.00	0.25	-0.30
Weighted	0.60	-0.03	0.32	-0.40	0.15	-0.12
nps_x	0.57		-0.08		0.03	

Table 3.3: The calculation of the normalised production score for each location. The production values are first summed up, then normalised and weighted. The resulting values are again summed up to get the normalised production score. The bold values are used to normalise (max_r).

Based on the normalised production scores location A has a clear advantage over the other two locations. However, using this score directly for selecting a proper location might lead to selecting locations that are far away from the urban area centre. This could be resolved by using a

resource that attracts all patches to promote clustering. To keep the patch types simple, another option was introduced through counting the neighbouring artificial patches (see Equation 3.12). The locations A and B are both near two artificial patches while location C is only in the range of one. Furthermore, location C is not near a border and will therefore get a penalty ($py_C = 0.1$). When the weights are set to $w_p = 0.5$ and $w_n = 0.5$, the location scores are calculated as shown in Equations 3.15-3.17.

$$s_A = \max(0, 0.5 \cdot \text{sign}(0.57)\sqrt{|0.57|} + 0.5 \cdot \frac{2}{2} - 0) = 0.88 \tag{3.15}$$

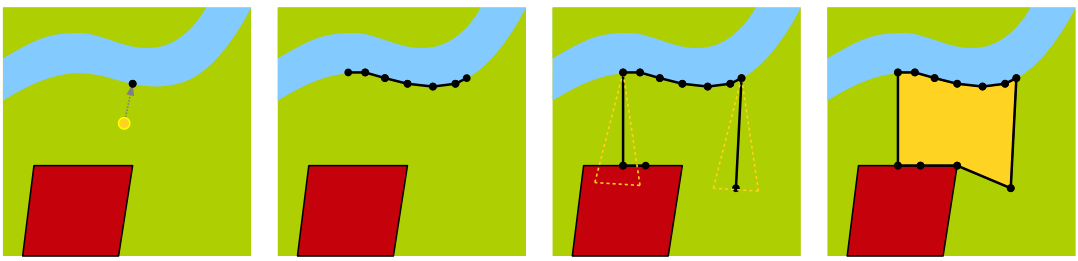
$$s_B = \max(0, 0.5 \cdot \text{sign}(-0.08)\sqrt{|-0.08|} + 0.5 \cdot \frac{2}{2} - 0) = 0.36 \tag{3.16}$$

$$s_C = \max(0, 0.5 \cdot \text{sign}(0.03)\sqrt{|0.03|} + 0.5 \cdot \frac{1}{2} - 0.1) = 0.24 \tag{3.17}$$

The found scores clearly indicate that location A is the best location. It has the highest food production and not as many losses on water as on the other locations. Although location C offers slightly better production values, location B has a higher score since it is near more artificial patches. To select the starting location for patch allocation, all scores are placed in the array as described in the Subsection 3.2.3. This results in the following array: $[0.88, 1.24, 1.48]$. A random number between 0.00 and 1.48 is chosen. For example 0.84 is chosen, and index 0 is found by binary search. Index 0 corresponds to location A and therefore this location is chosen as the starting point.

3.2.6 Patch Allocation

The starting point is first snapped to the nearest border border, Figure 3.4(a). By doing so, the algorithm for creating a new patch is more likely creating a patch near other patches and therefore a more coherent urban area. It also adds meaning to the orientation of the patch since it will be aligned with the border it snapped to. The closest point on the nearest border replaces the starting point. The snapping range is half the length of the diagonal of the envisioned patch (see Subsection 3.2.4). If no border is found within that range, the algorithm uses the given location, but with a random starting orientation.



(a) A snapped starting point.

(b) Expansion to the sides.

(c) Outward expansion.

(d) Connect the two end points.

Figure 3.4: The allocation of a patch starting by (a) snapping to a border, continuing by (b) expanding the two traces to the sides, followed by (c) growing perpendicular to the first trace and finally (d) connecting the end points of the two traces.

The algorithm for allocating a patch grows two traces of vertices, see Figure 3.4(b), one to the left and one to the right relative to the normal of the edge it snapped the starting point to, or a random normal in case of no snapping point was found. While adding new vertices to the traces, the algorithm tries to follow or snap to the borders of the terrain and other patches without crossing borders. By snapping and following the borders on the terrain, the algorithm tries to use the terrain data to shape the patch to give more meaning to the final shape.

When the expanding trace is snapped, it will try to follow the current border it is on until it has reached half the width of the patch or when the next edge of the border diverges more than $\frac{\pi}{16}$ radians compared to the angle of the last edge of the trace, in which case the trace releases the border. The latter does not apply to cases in which the border diverges to the right (when tracing clockwise) and to the left (when tracing counter-clockwise). This prevents the trace from crossing the border.

If the trace is not snapped to a border or stopped because of the diverging border, it will run freely in its current direction until it snaps to another border or until it reaches the predefined length. The direction of this run is the same as the last edge of the trace. This extends the current borders and creates more consistency in the landscape. Snapping to another border occurs when a border is within a the specified range of the trace and an angle of $\frac{\pi}{16}$ radians relative to the direction of the trace. When multiple snapping points are found, the closest point is used.

When both traces have reached the specified length, the traces should expand parallel to each other, perpendicular to the angle between the end points of both traces, as in Figure 3.4(c). Again the same procedure is applied for expansion of the two traces and so they can snap to and release borders until they have reached the predefined length, which is the height set for the patch type. In the example both traces release from the polygon on the top and try to grow parallel to each other. The trace on the left snaps to the border it meets within its range (the yellow dashed triangle).

The last step in the process tries to reconnect both traces by expanding the traces towards each other, Figure 3.4(d). When nothing is near the line between the two trace ends, the trace can be completed by adding an edge between the end points. However if only one of the traces is snapped, only this trace may expand for one step. After that another check is done to see whether there is a line without intersections between the end points. In the case of both traces being snapped to a border, the system checks if the traces will meet when expanding towards each other on the snapped borders. If the traces meet, the polygon is linked to the patch type and placed on the terrain as a patch.

3.2.7 Patch Maintenance

The maintenance of patches ensures that patches are replaced when they do not fit properly in their neighbourhood or are not producing the right resources for reaching the global equilibrium. Also, by replacing bad patches rather than removing them, the structure of the local area is maintained. Since a fully grown urban area will contain many patches, only a random subset is chosen to reduce the number of calculations needed. For each patch x in the subset a location ($v_{x,p}$) and production value ($v_{x,l}$) is calculated, see Equation 3.18.

$$v_x = 1 - w_p v_{x,p} + w_l v_{x,l} \tag{3.18}$$

The production value $v_{x,p}$ depends on the goals and was also used in Equation 3.11 in Subsection 3.2.3. It is weighted by the production weight w_p. In contrary to Equation 3.11 actual production values are used instead of predicted production values. This means that both the area and damage level of the patch contribute to the final production value.

The location value $v_{x,l}$ is based on the current consumption by the patch and whether the surrounding neighbours can supply the patch with the needed resources as shown in Equation 3.19. It is calculated in the same way as the impact of shortages was calculated for damage in Equation 3.8. The set H holds all resources h for which both the production of the patch $p_{x,h}$ and the production from neighbouring patches $ls_{x,h}$ is negative. These values are compared with the total consumption by the patch and weighted per resource with w_h. By subtracting the sum of the fractions from one, a value between zero and one is created in which zero stands for a bad fit with many shortages and a one for a proper location without shortages. If no resources are consumed, the value of $v_{x,l}$ is equal to one.

$$v_{x,l} = 1 - \sum_{h \in H} \frac{|w_h| \max(ls_{x,h}, \min(0, p_{x,h}))}{\sum_{r \in R} \min(0, p_{x,r})} \tag{3.19}$$

When all values for the subset of patches have been calculated, the values are used in a fitness proportionate selection (see Subsection 3.2.3). From the subset of patches, a small subset of patches is selected and used for possible replacement. For each of the selected patches, the algorithm searches a suitable replacing patch type.

To find a suitable patch type to replace a patch from the small subset, the value of each patch type t is calculated as in Subsection 3.2.3 was described, based on the centre l of the selected patch. Though in this case all patch types that do not suit within the bounds of the area of the selected patch are omitted, so only patch types aimed at a similar size are used. Subsection 3.2.6 already explained the thresholds used for area size. Excluding patch types by size could result in an empty list of possible replacements. If this happens, the replacement process stops.

The calculation of the score for a patch type consists of calculating the location value $v_{t,l}$ and the production value $v_{t,p}$. The latter was already calculated for the patch type selection in the patch creation phase (Subsection 3.2.3-3.2.6). This results in a score for each patch type (Equation 3.20):

$$score_t = w_p v_{t,p} + w_l v_{t,l} \tag{3.20}$$

This score is also calculated for the selected patch to be able to compare the scores. This score is based on the current production values and therefore a less damaged patch will have a higher score. Using fitness proportionate selection, one patch type is selected for replacing the patch. Though this will only happen if the score of the selected patch type is higher than the score of the patch itself. The chances of replacing the patch are therefore higher when the patch is damaged or in a wrong neighbourhood. The costs of the replacement are the same as for placing a new patch.

Chapter 4

Implementation

This chapter explains how the approach described in the previous chapter is transformed into software, including some solutions to problems that were encountered during the process. For many parts of the design, implementation aspects are mentioned here. Also the global structure, communication within the software, and used libraries are discussed.

4.1 Process

To implement the software, an agile software development approach was used. First a rough planning was made covering the whole project, starting in October 2011 and ending in June 2012. Each month a working prototype was planned to be delivered covering new features. Before each monthly cycle the project planning was updated to remain accurate with the current direction and state of the project. The features of the monthly cycle were spread over the weeks of the month and for each week a specific planning was made just before the week started. The final rough planning for the project can be found in Appendix A.

During the implementation process a few parts turned out to be more difficult to implement, such as the creation of patches and the maintenance of the borders. A lot of time went into implementing those features. This forced me to adapt the planning causing a few major elements, like road and traffic simulation that could greatly enhance the simulation, to be left out.

The simulation is run on a terrain created in *SketchaWorld* [6], an approach for the declarative modelling of virtual worlds. With it, designers can create three-dimensional game worlds that include forests, rivers, roads, and cities by using simple, high-level constructs. SketchaWorld also provides a Polygon class which is used by UrbSim to indicate the borders or patches. This class relies on the General Polygon Clipper library (GPC) [34] for doing boolean operations on polygons. The semantic information needed for the simulation is supplied through a database created with *Entika* [28], a framework to facilitate the deployment of semantics in games. UrbSim relies on many elements provided by both the SketchAWorld project and the Entika project. Since both projects are written in C#, it was convenient to write UrbSim in C# as well.

4.2 Structure

The software system has the Simulator class as its basis. This class controls the simulation by sending updates to the Display and the Government. The Display, explained in more detail in Section 4.5, handles user input and also provides graphical feedback to the user. It is created by the Program class at the start of the program. The Government, see Subsection 4.3.1, manages the patches. For this process it uses the AllocationManager for creating new patches, and the Solutions for replacing patches. Figure 4.1 shows these major classes and their associations.

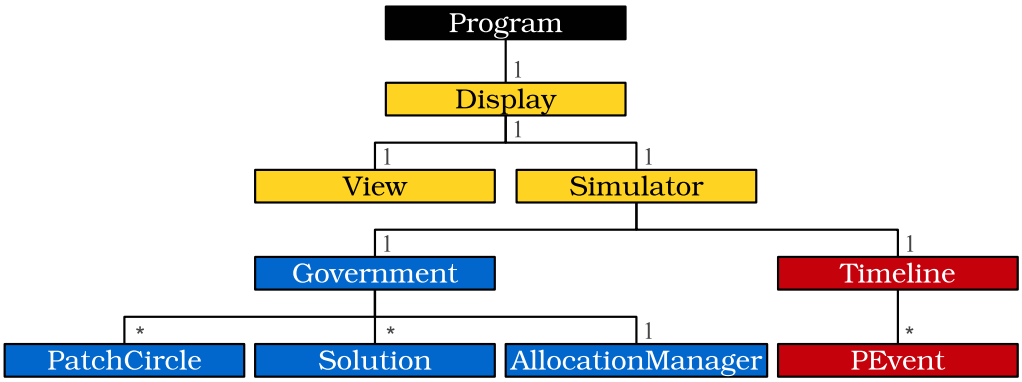


Figure 4.1: A simplified class diagram of the system. The complete class diagram is posted in Appendix G.

4.3 Semantics

All semantic information used in the simulation originates from a database created by the user with Entika [28]. This semantic information defines all relations between the objects in the world, their properties and also the types of interaction possible with the objects. Since it can be hard for designers to write all default information themselves, they can choose to extend a database that holds the bare necessities for the simulation and game world. In this and the next chapter, this database will be referred to as the 'default database'. More information on this database can be found in Chapter 5 and Appendix D. The designers are not restricted to using the default values. They can still change the generic types which allows them to control the semantics both on a global scale and in detail.

The semantic information is distributed over a few classes. Most information is stored in the Solution instances (see Subsection 4.3.2), including the repelling and attractive factors among the different patch types available. The government receives information about which Solutions can be used and when. This last part is stored in a subset of the events. The other events hold information about destruction, see Subsection 4.3.4.

Originally the system intended to use the Semantics Engine from the Entika project rather than just copying the semantic information into the different classes. This would leave the Semantics Engine to process most semantic information for distributing and collecting resources, executing the events and changing patch values and damage levels. However, this turned out to be unfeasible to create in the database and it would be slower than it is with the use of specialised classes. On top of that, not all distance checks would be run each cycle to save time which would result in lacking information.

4.3.1 Government

The most important class in the simulation is the Government class. As described in the previous chapter, the government is responsible for adding, replacing and removing patches. The government holds a collection of created patches (PatchCircles) and a list of called patch types (Solutions) that can be applied to new patches. The collection is updated every cycle by adding new patches, and replacing and removing low valued patches.

During each cycle, also new goal and stock values are calculate. These are both stored in a Dictionary, a class that holds a collection of values that can be accessed by giving a key value. Each key points to a single value. Retrieving a value using a key is very fast (O(1)), because the Dictionary is based on a hash table. Using the Dictionary class for storing the goal and resource values does make it more complex to work with compared to arrays, but makes it incredibly flexible for the generic input given by the semantic database, allowing the user to create any wanted new type of resource. The keys used in both collections are the same as those used in the patches to

make it possible to compare resources, goals and production values.

4.3.2 Solutions

The semantic information of patches is stored in Solution instances which in turn are kept by the government. These patch types, as they were called in the previous chapter, hold information like envisioned size and corresponding production and costs, the category of the patch, and the attraction weights. Just like the resources and goals of the government, the production values, construction costs, and weights of the solutions are stored in a Dictionary. The keys used in these Dictionaries correspond to those used for the resources and goals. These values are used for calculating the production values and construction costs of a PatchCircle when created with a Solution and for calculating the value of a Solution compared to the current goals and stock of the government when new patches have to be created.

Which Solutions are available to the government, depends on the technological advancement of the government. The technologies that enable or disable solutions are stored in a technology tree. A technology tree is a network of technologies that succeed each other. A technology might depend on other technologies and can only be triggered once its dependencies have been researched first. Each technology has a starting time linked to it. When the simulation reaches that point in time, the technology is triggered. However, if the dependencies of the technology have not been triggered, the technology is pushed onto a stack of technologies that have to be checked again in the next cycle. The technology will stay there and it will not be executed as long as the dependencies have not been researched.

If a technology is executed, all relations are read. A technology may refer to multiple solutions to be enabled and disabled at the same time. All patch types that should be enabled are added to a list of available solutions. All patch types that should be disabled are removed from that list. The latter is useful when a patch type should be replaced or upgraded to allow a farm to produce more food or change the building style of houses, for example. It can also be used to simulate the end of an advanced era and introduce old style patches again. However, this will not directly change the existing patches. The new patch types will be introduced gradually by adding and replacing patches.

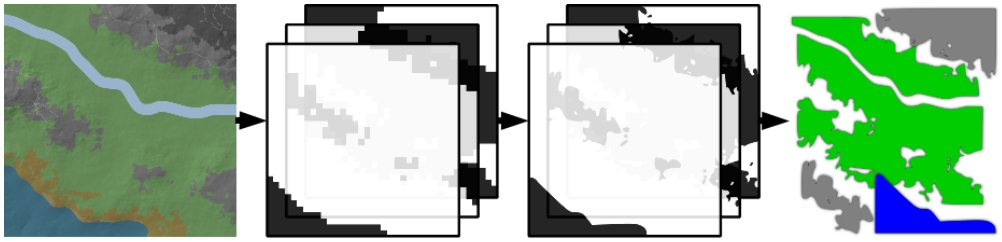
Solutions are also linked to natural patches which are created in the segmentation of the terrain, which is explained in the following subsection. To inform the system about which natural patch type is related to which terrain feature, a separate file holds all database references. This file assigns each ground type and terrain feature found on the terrain to a Solution. Each Solution can cover one or more terrain features, so not every terrain feature has to be mapped one on one with Solutions to reduce the number of natural Solutions needed and to keep an overview. For example, the user can create a new Solution for a mountain top, if specific relations for snowy mountain tops are needed in the simulation. The database references file can also be used for changing the 'Natural' category reference if one would like to use another category as the basis for natural Solutions.

4.3.3 Borders

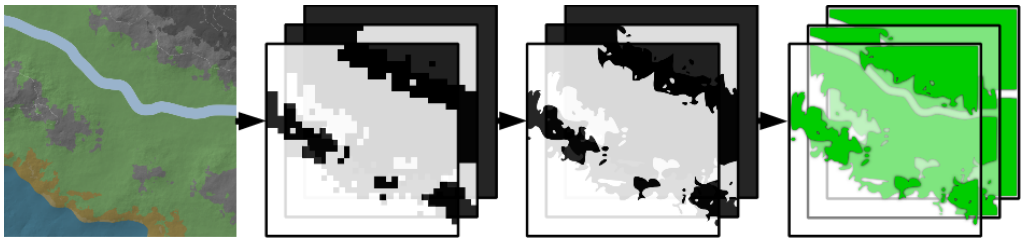
The creation of patches largely depends on the features of the terrain. To reduce the system load when creating new patches, these features are converted into polygons that mark the borders of the features. The terrain features are stored as natural patches in the allocation manager. This is done by the TerrainLoader class.

The simulation starts on a terrain created in SketchaWorld [6]. The terrain features are saved in five different layers. Each layer has to be processed separately. The last two layers, roads and buildings, are neglected because the simulation needs a terrain without artificial elements. The first layer is the terrain layer. This layer is a height map of the terrain combined with a ground type at each point on the grid. This is converted into maps, one for each category of patch types. For the 'Natural' category even for each patch type.

All grounds on which the patch can be built are merged together to form one big polygon called the allowed building area. The conversion to an allowed building area map is done in four steps (see Figure 4.2): first a black and white bitmap is created, then the bitmap is converted into scalable vector graphics (SVG) and finally it is converted into polygons that are used to create patches which can be used in the simulation. Normally less steps would be needed for such an operation, but no library for transforming the terrain into polygons was available. The following paragraphs will elaborate on each step and will cover both the conversion for natural patches (the creation of natural patches that will be stored in the collection of patches, depicted in Figure 4.2(a)) and the creation of allowed building maps per patch category (shown in Figure 4.2(b)).



(a) Each area with a different ground type is converted into a patch. This conversions results in a set of separate patches with semantic information attached to them.



(b) To get a set of allowed building area polygons, areas of different ground types are combined based on the information from the database per patch type category.

Figure 4.2: The terrain is segmented into natural patches (a) and allowed building maps (b).

Sampling First a bitmap of 128 by 128 pixels is created. For each pixel in the bitmap the corresponding point on the terrain is calculated. For this location the ground type is compared to the list of allowed ground types. If this border map is created for a natural patch, this list contains all ground types that correspond to the patch. Otherwise it is for a patch category and will therefore include all ground types on which patches from the category can be built. If the found ground type is in the list, the pixel is set to black, otherwise it is painted white. This results in a bitmap indicating all allowed areas with black.

SVG The conversion of the bitmap to a scalar image is done using a library called 'Potrace' which was created by Peter Selinger [35]. It traces borders in the bitmap for finding smooth traces.

Polygons The vector image is saved as a set of lines and curves. The polygons can be constructed by reading the image file step by step, saving the lines as lines in the polygons directly and approximate the cubic Bèzier curves at 25, 50 and 75 percent of the curve as connected points of the polygon.

Patches To complete the conversion, the created polygons are saved as patches (in the case of natural patches) in the collection of patches, or as an allowed building area map (for categories). For the natural patches this means that the semantic data from the database is linked to the polygon. The allowed building areas are stored in the allocation manager that will use them later on in the creation process of artificial patches.

The second layer contains all water features like rivers and lakes. These are already stored as polygons and can therefore be used directly. If a category of patch types cannot be built on the water features, the polygons are subtracted from the allowed building area map of that category. If the patches can be built on water, the polygons are unified with the area map. The polygons are also added to the collection of patches since they can be set to produce water and therefore attract patches that need water.

The third layer consists of vegetation. Forests are defined as polygons and are treated in the same way as the water features. The forests are either subtracted from or added to the border maps for the different categories. The layer also contains pastures and fences, which are neglected since these are artificial. Applying the vegetation layer concludes the conversion of the SketchaWorld terrain.

After the conversion, the allocation manager has a set of natural patches and a set of allowed building area maps. The maps that define the allowed building areas are not static, but are updated every time a new patch is created or removed. When a patch is added, its footprint polygon will be removed from all allowed maps with which it intersects, as shown in Figure 4.3. This prevents new patches from being built on top of other patches. When a patch is removed, the polygon is merged again with the maps, but only the intersection with the original area is added.

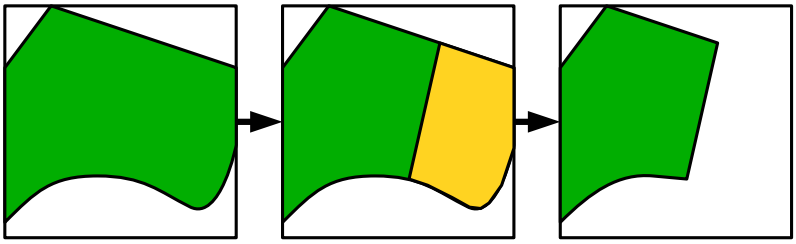


Figure 4.3: For each patch type category an allowed building area (green) is defined. In this area new patches (yellow) from this category can be created. The area covered by the new patches is subtracted from the allowed building area.

4.3.4 PEvents

Events can be used to give the user more control over the course of the simulation and to make the simulation less static. In UrbSim only two types of events are used. The most important event type is *technology*, which was explained in Subsection 4.3.2. The other implemented event is a disaster: *destruction*. This event will increase the damage level of all patches within the area of the event. The amount of damage done by the event is proportional to the amount of overlap the patch has with the destruction area, see Equation 4.1. The area of overlap ao_x is divided by the total area of the patch a_x to find the percentage of overlap. This fraction is added to the old damage level $dmg_{x,t-1}$, resulting in the new damage level $dmg_{x,t}$.

$$dmg_{x,t} = dmg_{x,t-1} + \frac{ao_x}{a_x} \tag{4.1}$$

The destruction area can have any shape and size. If multiple areas are affected, multiple events should be created. After the execution of the event, the patches that were completely in the area of destruction, will be removed in the next cycle during the update of the patches, see Subsection 3.2.2.

The timeline in which the events are stored is both present in the simulator as well as in the TimelinePanel. The former keeps a list of triggered but not executed events and calls the government to execute the event. It also asks the timeline to return all events in the current timespan of the simulation and also forwards them to the selected government. If the government is not able to execute it, it is temporarily stored in the list of events that have to be executed

later. The TimelinePanel just uses the timeline for displaying the current state of the simulation including the events and the current time.

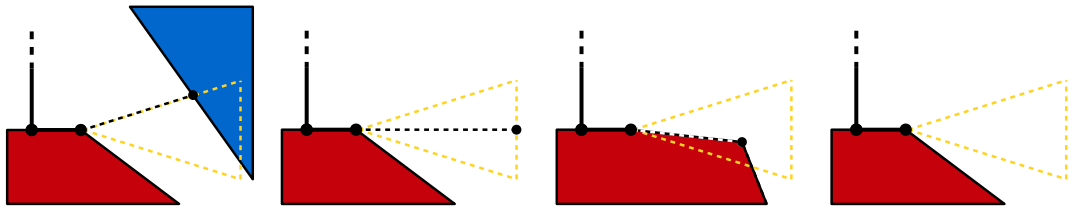
4.4 AllocationManager

The AllocationManager instance fulfils the task of finding proper locations and shapes for new patches. It receives a Solution from the government. Using the properties of the solution and randomly chosen points, the allocation manager tries to select the best location. After that it grows two traces parallel in time until these join again, see Subsection 4.4.1. Afterwards, the created polygon is checked to see if it meets the requirements set by the Solution. With over three months of designing, writing, and testing, this part of the program took by far the most time to complete. This was not only because of its size and the lack of some mathematical operations in the used libraries, but also because of its complexity (especially the patch creation) and the many bugs that popped up during those months. Most severe bugs were related to problems with the polygon clipper.

4.4.1 Patch Creation

Allocating the polygon for a patch is done in three stages: expansion to the sides, outward expansion, and connection. The first stage is executed using the angle of the snapped element, the second stage starts both traces perpendicular to the angle made by an imaginary line between the end points of the two traces and the last stage tries to link the end points. The stages are all quite similar, except that the last stage adjusts the direction every step to make sure the two traces will meet.

In each step the algorithm tries to expand both traces. It first calculates a candidate point for expansion of the trace, according to its position, direction, and distance it has to travel. This step also determines the state of the trace (see Figure 4.4). Based on the states of both traces, a decision is made for either adding the expansion candidates or not. The latter option allows the traces to wait for each other to avoid missing each other during expansion. The traces can be in four states:



(a) SNAP (b) END (c) NEXT (d) RELEASE

Figure 4.4: The traces switch between four states. This is done during the selection of a expansion candidate.

SNAP The trace is not following a border, but encounters an object (blue) in its view (yellow). The trace will snap to it. The angle of view of the traces is $\frac{\pi}{8}$ radians. The value can be changed via the database as a parameter of a patch type.

END The trace is not following a border, and does not see a border in its view. The end point of the trace can be set and the trace is finished.

NEXT The trace is following a border, and has the next point of that border in its view. It will add that point to the polygon.

RELEASE The trace is following a border, but the next point of that border is not in its view. The trace will release the border.

The transitions between these states can be seen in Figure 4.5. This is a simplified state diagram showing the two decision moments: finding a snapping point, and checking whether the next point diverges too much from the current direction of the trace. The actual implementation, however, is a bit more complex. During the allocation of the patch, two traces are expanded in parallel, giving combinations of states. The essential combinations of states are shown in Table 4.1. The states SNAP and NEXT have an overlap in properties and are therefore shown as one state: SNAP/NEXT. The actions shown in the table are results of the combinations of actions.

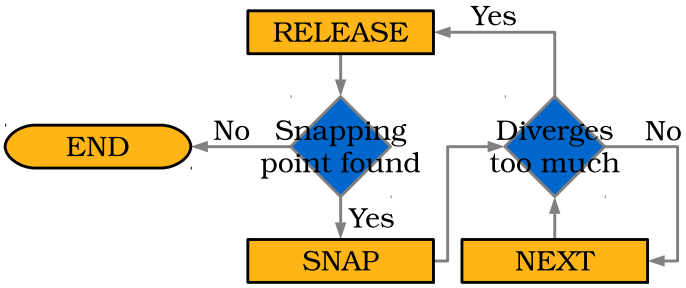


Figure 4.5: The transitions between states are based on finding a snapping point (when the trace is not attached to a border), or finding a next point in range (when the trace is attached to a border).

State A	State B	Action A	Action B	Path found
SNAP/NEXT	SNAP/NEXT	Expand	Expand	Maybe*
SNAP/NEXT	END	Expand	Wait	No
SNAP/NEXT	RELEASE	Expand	Nothing	No
END	END	Expand	Expand	Maybe*
END	RELEASE	Wait	Nothing	No
RELEASE	RELEASE	Nothing	Nothing	No

Table 4.1: The states of two traces (A and B) after calculating the possible next step influence the decision whether the trace is actually extended or not. * = more information is needed to decide whether the traces are complete.

The first row, for example, shows that both traces are in the state SNAP/NEXT. Both traces are following a border. In that case both can proceed in adding the expansion candidate to their trace of points. This is referred to as 'Expand' in the table. If both traces are on the same border and will, after the addition of the expansion candidates, either meet each other or be just one edge apart, then a path is found between the two trace ends and the polygon for the patch is finished. If this is not the case, the traces will have to continue following the border.

In the fifth row an example is shown of a trace in an END-state and one in a RELEASE state. The first trace waits for the the second trace to finish expanding. This results in a cycle in which none of the traces expand. The second trace will either snap to a new polygon or jump to an END-state.

The fourth row shows a case in which both traces have reached an end point. However, this does not necessarily mean that the polygon is finished. If the ends of the traces meet, then one of the expansion candidates is added to the trace and the polygon is finished, but otherwise the traces will continue in the next stage of the expansion process. In the latter case, the two release candidates are added afterwards.

Once finished, the connected traces form a polygon that is probably not precisely of the same size as the intended width and height suggest. To overcome these differences thresholds are introduced. If the area of the polygon is smaller than the (default) lower threshold of 70% or bigger than 143% of the envisioned $width \cdot height$, the polygon is rejected. The thresholds can be set per patch type so for some patches this rule is more important than for other patches. A polygon is also rejected when the polygon intersects with itself. If the polygon is rejected, a new

allocation should be found. If the polygon is not rejected, the patch type is linked to it to form a patch. The patch is then added to the list of patches of the government so the government can keep track of the patch. It is also added to the quadtree of the allocation manager in which it stores all borders. These borders are used during the process described above when searching or following borders. The border maps are also updated by subtracting the area from the allowed area to prevent other patches from being built on top of the newly created patch.

4.4.2 Polygon Artifacts

The creation of patches is not perfect and can result in polygons that are too small, too big, crooked or even overlapping other patches. This is partly due to a wrong update of the borders on adding and removing patches caused by the polygon clipper that creates artifacts in the borders. During clipping of polygons, some edges are not always removed, as shown in Figure 4.6. Another problem that arose due to artifacts is that the patch creation algorithm can get stuck in an endless loop. The later implementation of the algorithm did not show this error, but to be sure that the algorithm will not get stuck, a maximum number of segments can be specified. If the number of segments in the created polygon hits this threshold, the AllocationManager will stop and the created polygon will be rejected.

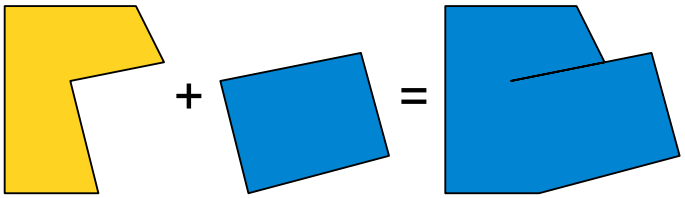


Figure 4.6: The general polygon clipper (GPC) does not always correctly add and subtract polygons. In the example, the blue polygon is added to the yellow polygon. Although the polygons fit perfectly together, the clipper did not remove some overlapping edges.

To be more certain proper patches are created and added, a few more checks are introduced. If the patch passes these tests, a PatchCircle is created using the given Solution. Otherwise the patch is rejected and the government has to start over. The first check subtracts the allowed area from the patch. This also helps to create holes if the polygon was grown around a smaller patch. If nothing of the patch is left, the patch did not pass the test. This test uses the polygon clipper and therefore it can result in wrong patches, though most will be filtered by the following tests. The second and third test check for area. The user can specify in the database how much deviation from the intended size is allowed. The last check runs through the polygon to check for intersections. If the polygon intersects with itself, the polygon is rejected.

4.5 Graphical User Interface

The focus in this project is on creating a simulation, not on creating a vast tool to manipulate a created urban area via a graphical user interface (GUI). Therefore the GUI, shown in Figure 4.7, is very simple and mainly contains elements for feedback to the user about the simulation. Controlling the execution of the simulation can be done via the GUI, but controlling the semantic information used in the simulation is done through the Entika Semantics Editor.

The GUI contains the following elements in order of appearance from top to bottom in Figure 4.7:

Menu and Control bar The two top bars offer controls for loading a SketchaWorld terrain, a semantic database from Entika, or a whole project that includes both and some side information like database references and random seed.

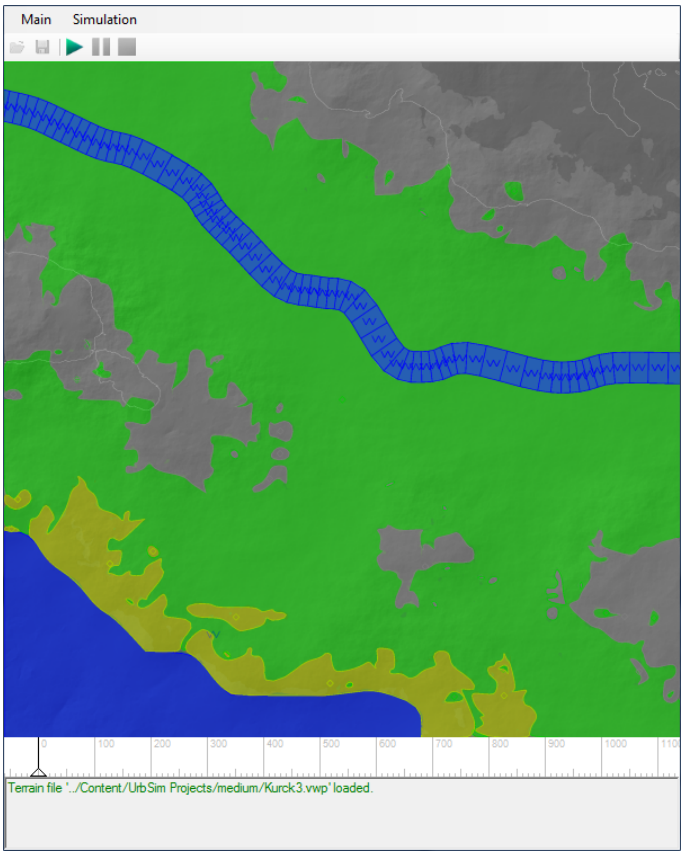


Figure 4.7: The GUI of UrbSim.

World view The world view shows the current situation of the semantic world including the natural features and the created objects. The latter two are drawn on different layers to be able to change the created objects layer every cycle while keeping the natural features layer unchanged.

TimelinePanel The panel of the timeline displays the current time of the simulation. Events such as disasters (red) and technological advancement (blue) are displayed on the timeline as lines to indicate when they will occur in time. The user can zoom in and out to get an overview of the events.

Console Textual feedback is given to the user via the console at the bottom of the GUI. World events are displayed here. Also states for loading projects, and starting and stopping the simulation are written to the console.

Chapter 5

Results

The results presented in this chapter show how urban areas are generated by UrbSim. It covers which information is supplied via the database and how this is used to allocate new patches. This includes on which values the selection of patch types is based, how a location is selected and how the resulting patches are shaped. The results are shown both as geometric output as well as the statistical background of the simulation. To see how the simulator performs in different settings, it is first tested with the default database on a terrain with a river (see Section 5.1). The selection of locations is explained in more detail in Section 5.2. The default scenario is used as a basis for comparison with other settings presented in Section 5.3, including the use of a less complex database and a scenario with another terrain. A discussion of the found results can be found in the next chapter.

5.1 Default Scenario

The performance of the simulator is tested by running it with the default database. The patches provided in the default database are presented in four main categories: agricultural, industrial, water supplies, and residential. The agricultural patches (*garden*, *small farm*, *medium farm*, *big farm*, *giant farm*) provide food, but require manpower and water. These can only be built on grassy areas. The industrial patches *clay quarry* and *stone quarry* both produce stone, but require manpower. The water supplies (*well* and *water pump*) produce water and require manpower. The last category contains the residential patches (*hut*, *woodcutters hut*, *simple house*, *house*, *good house*, *mini flat*). These patches deliver manpower, but require food and water. This database does not have a patch type that consumes the resource stone directly, but stone is part of the building costs of many buildings. A detailed overview of the used patch types and their properties can be found in Appendix D.

The patches are enabled through technologies over time. The technology tree in the default database is shown in Figure 5.1. At the start of the simulation only small and simple patches are available. Over time bigger patch types such as the *giant farm* and *mini flat* are added that provide more resources, but also demand more.

The terrain, shown in Figure 4.7, used in these tests contains mountain ridges (grey), grass lands (green), an ocean at the bottom left (blue), a shore area (yellow), and a river (blue) between the mountain ridges. The river is subdivided into several patches that provide water to their surroundings. The other areas do not provide resources. The segmentation of the terrain displayed in Figure 4.7 is done with a sampling resolution of 128 by 128, but the tests shown in this chapter are conducted with a segmentation based on a sampling resolution of 32 by 32. This speeds up the simulation and offers more stability since the polygon clipper has less work to do. A lower resolution however does result in a less accurate representation of the terrain and therefore also in a less accurate model.

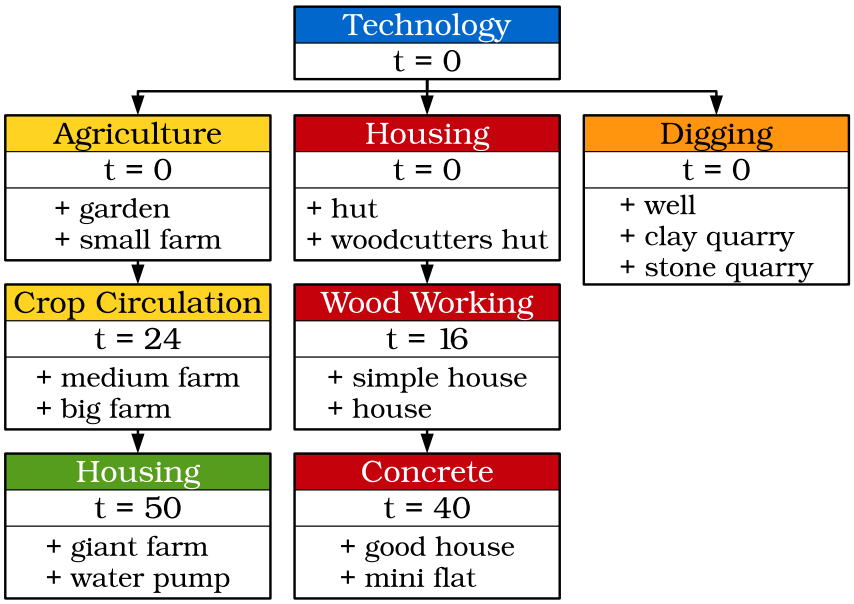
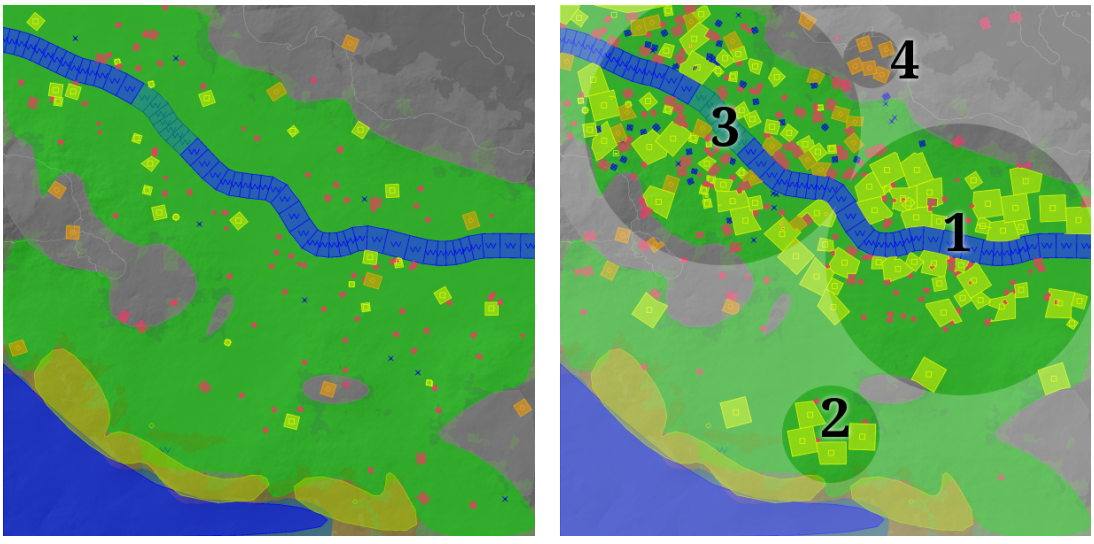


Figure 5.1: The default technology tree.

5.1.1 Visuals

The default database can be used to generate a great variation in urban areas. The output of the program depends on the random numbers from the pseudo-random number generator. The project defines a seed used for the generator. To see how the simulator performs with the default database, it is ran for 200 cycles. More cycles would increase the risk of running into clipper errors and crashing of the polygon clipper. Figure 5.2 shows the development of an urban area in an early stage (after 50 cycles) and at a more advanced stage (after 200 cycles).



(a) After 50 cycles.

(b) After 200 cycles.

Figure 5.2: An example of an urban areas that developed for 200 cycles with the semantics from the default database.

The placement after 50 cycles, shown in Figure 5.2(a), seems to be rather random with a slight

attraction by the river. The patches are scattered over the terrain, but along the river. This could be due to the fitness proportional selection which in many cases needs some time to let the results converge to the aimed clustering. Figure 5.2(b) shows that after 200 cycles more and bigger patches have created an obvious cluster along the river (region 1 and 3). Unfortunately the government did not place the patches such that no gaps were present in between the patches.

In the first stage mainly houses (red) were created, but as time goes by and the area starts to get more crowded and clustered, farms (yellow) tend to use up most of the space available. On the right of the second image, in region 1, only farms and houses are situated. This is possible due to the production of manpower by the houses, production of water by the river and production of food by the farms. No other resources are needed to sustain the patches in this area. This can also explain why some of the houses which were already present after 50 cycles are still present after 200 cycles. The giant farms in between these houses have replaced their smaller predecessors in the timespan of 150 cycles. A remarkable feature in this example is the houses near farms. Most of the farms have a house nearby. The group of farms in region 2, for example, is accompanied by a few houses, just like the farms in region 1 on the right of the image.

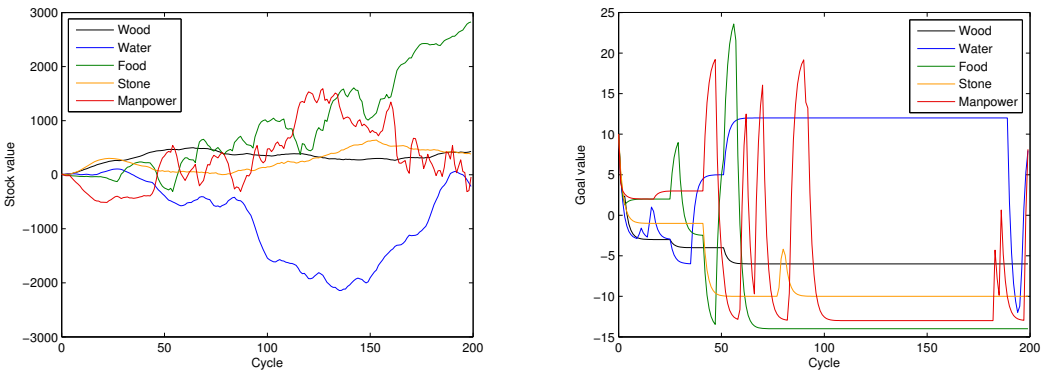
The mix of patches on the top left, in region 3, is completely different from the previously discussed areas. Although many houses and farms are situated on the top left, the houses are generally bigger and the farms smaller. Furthermore also water supplies and quarries are present in this area. Apparently the mini flats and quarries require more water and therefore water supplies were needed in that area. The quarries are scattered through the area, but some snap to the river or to each other as expected (region 3 and 4). These quarries provide the government with enough stone to build new buildings.

The result of a simulation can be saved at any time as a SketchaWorld [6] project to be edited further and to generate a three-dimensional model. Figure 5.3 shows the three-dimensional model of the urban area generated in the simulation. SketchaWorld does not allow arbitrarily shaped patches and therefore divides the patches into rectangular spaces. For each rectangular space, a CGA grammar [14] is used to generate the three-dimensional geometry. SketchaWorld tries to flatten out the terrain underneath the patches, but in doing so also creates steep slopes in between patches. This causes some buildings to be partly underneath the terrain. For some farms, for example, this causes the land plane to invisible.

Figure 5.3: A render of the model created with SketchaWorld showing the urban area after 200 cycles.

5.1.2 Statistics

To get a better view of what is actually happening during a simulation and why certain decisions were made by the algorithm, this section will describe the statistical background of the simulation including the development of stock and patch type values. The former is shown in Figure 5.4(a). All resources start at a value of 10, but these values quickly change when the government starts to construct new buildings. Within a few cycles a shortage of manpower is created, while a shortage of water and food prevents the government from building new houses. The shortage of water is caused by patches that are not near the river. As soon as the food stock becomes positive new houses are built and the shortage of manpower is converted into a surplus of manpower. This however has a negative effect on the water stock. The shortage of water gets worse up to cycle 60, 10 cycles after the introduction of the water pump. It takes some time for the government to place water pumps. With the placement of the new pumps, the shortage seems to stabilise, but after cycle 80 a huge shortage is created while there is plenty of all other resources at that moment. In the clustered regions near the river, a group of farms and houses is able to sustain and even grows, which causes an increase in manpower and food production. The manpower stock keeps increasing up to cycle number 125, but afterwards the government manages to bring it back to around zero. On the other side of the zero line, a similar process can be seen for the water stock. The shortage grows for a long time but eventually it returns to zero around cycle 190, just like the manpower stock. After this both drop below zero. The food resource gets out of control. The stock value first grows together with the manpower stock, but when the manpower stock starts to decrease, the food stock keeps on growing. A possibility exists that it will return to zero just like the values of manpower and water did, but in these 200 cycles clearly too much food is produced and too little houses were built to consume the produced food. The stock values of the two other resources, stone and wood, are stable throughout the whole simulation. These resources are only used as building materials and are not consumed directly, which can explain the smooth development of their values.



(a) The changes of stock over time showing shortages and surpluses.

(b) The goals over time.

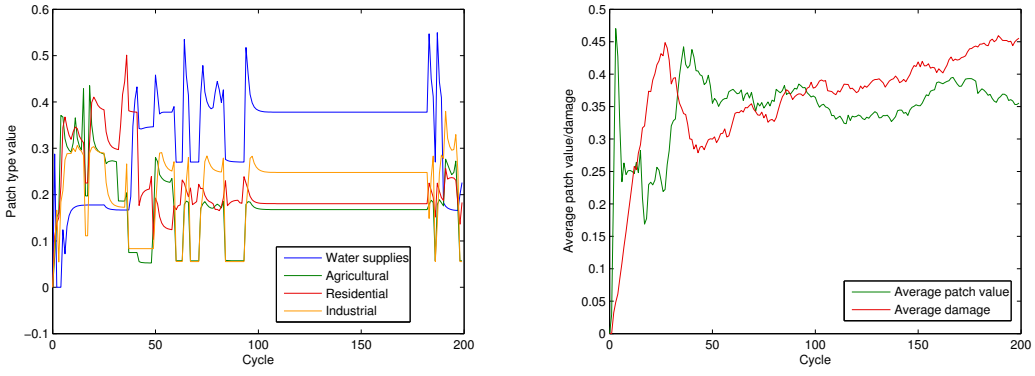
Figure 5.4: The calculation of the patch type values is based on the goals (b) which in turn are based on the stock (a).

Just like the resources the goals (shown in Figure 5.4(b)) start at a value of 10, but since the goals try to bring the stock to zero, the goal values soon drop. The shortage of manpower and food after the start result in a positive goal value for both resources, while the surplus of water, stone, and wood result in negative goal values. The goal values are not only affected by the stock values, but also by the available patch types, and more specifically the production values of these patch types. The technology 'Crop circulation' for example, which is triggered at cycle number 24, offers new farms that produce more food, but also require more wood to be built and water to operate. The corresponding goal values quickly converge to the new values. A similar event can be

seen after cycle 40 when the 'mini flat' is introduced. It offers more manpower in return for more stone and food. Again the goal values adapt to the new maximum values. After the latter event the manpower goal alternates the maximum production value and maximum consumption value for some time due to the changing stock value of manpower which switches between a surplus and a shortage a few times. This period is followed by a stable period of 70 cycles. The goals do not change as a result of the stock values that do not switch from shortage to surplus and vice versa. When the manpower stock and water stock do break this silence, the goals rapidly change. The manpower goal starts to increase and the water goal value drops. As soon as the water stock value drops again, the corresponding goal value increases.

The patch type values (used for selecting patch types that produce needed resources), Figure 5.5(a), are calculated based on the goal values and the production values of the available patch types. This results in similar patterns in both the plots of the goal values and the plots of the corresponding patch types. The positive manpower goal for example raises the values of patch types that produce manpower, namely the residential patch types. The positive value of the food goal results in a raised value of food producing patches such as the agricultural patches. Due to the long period in which there is a shortage of water, the water supplying patch types (well and water pump) have the highest value. The drops in this value are caused by shortages of manpower since the water supplies need people to operate them. As the goals do not change in the period between 100 and 170 cycles, neither the patch type values change. After this long period the order patch type values changes. The water supplies drop in value, while the values of quarries, farms, and houses rise.

Figure 5.5(b) shows that the average patch value starts off at just under a half due to a lack of resources and soon drops when more patches are placed with even worse placement. The lack of resources increases the damage level of the created patches which lowers the average patch value even more, but as soon as the government starts to replace and remove water supplies and quarries, the average patch value increases and the average damage value decreases. After this event that happened between cycle number 30 and 45 the damage level starts to increase again, but not as fast and only after 155 more cycles it has reached its old level again. This steadily growing damage level also results in a steadily growing number of removals. The average patch level is steady in two periods of about 40 cycles. It has a value around 0.37 between cycle 50 and 90, then it drops to a value of 0.33 at cycle number 110. This value increases after cycle 150 and then slowly drops again. In these periods the shortage of water does not change a lot either. This last drop could be a result of the removal of water supplies which results in a shortage of water. The increase in average value before the drop can be explained by the reduction of the shortage in that period.



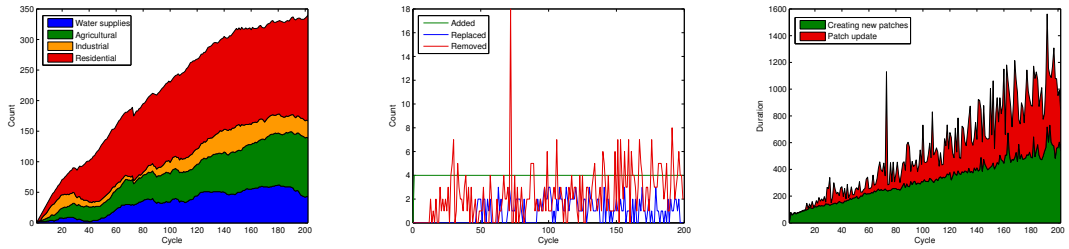
(a) The values of the patch types over time.

(b) The average patch values and damage levels over time.

Figure 5.5: The development of patch type values, patch values and damage levels over time.

Figure 5.6(a) shows how the number of patches per patch type category changes over time. At the start there is no clear preference for one of the patch types and all grow equally fast. After cycle number 20 this changes and around cycle 40 there is even a dip in the number of patches mainly caused by the removal of water supplies and quarries. Due to the high patch type value for residential patches, the number of houses increases rapidly. This was enabled by a surplus of both food, stone, and wood and at the same time a shortage of manpower. The only problematic factor is the production of water. This becomes worse as more houses are built which require water. After cycle 40 the number of water supplies grows in an effort to reduce the water shortage.

After cycle number 70 a small dip in the number of patches occurs. This is caused by a destruction event at cycle 70 in the top left part of the map. Mainly farms and some houses were damaged in the event. The effect of this event can be seen in the other figures as well. Figure 5.6(b) which shows the mutations over time, shows a peak of 18 removed patches at cycle 71. This operation takes some time to compute which can be seen in the third graph, Figure 5.6(c). The sudden removal of patches does not have a big effect on the simulation since the number of patches quickly recovers and continues the growth it was working on. This growth slows down after cycle 150. After cycle number 180 when both the stock values of manpower and water reach zero, the number of water supplies drops, resulting in a new shortage of water. The number of residential patches however fills this gap and grows faster than before.



(a) The number of patches of each patch type category.

(b) The number of mutations done each cycle.

(c) The duration of each cycle.

Figure 5.6: The patch type counts, the number of mutations and the duration of the updates per cycle.

The second graph (Figure 5.6(b)) shows the mutations over time. Each cycle a steady number of patches is created. This number is set in the database, but only sets the upper limit. If the allocation manager is unable to find a good location for a new patch, when there is too little space for example, the number of patches added per cycle might drop. The government starts replacing patches after cycle 50. Before this moment in time, the government was unable to find good replacement candidates, which could be a result of bad location production and with the lack of patches that have about the same size. The number of patches removed per cycle slightly increases over time due to the increasing damage levels. Exceptions to this rule can be found at the beginning when badly placed and unneeded water supplies and quarries were removed and the destruction event at cycle number 70.

All these mutations come at a cost. The third graph, Figure 5.6(c), shows the time consumption of adding and removing patches. Compared to these two, the other processes (replacement and updating of stock and goals) are negligible and would not be visible in the chart since these actions combined take less than five milliseconds to complete. When the graph is compared to the plot of the patch removal count in red in Figure 5.6(b), one can clearly see that removing patches takes up a lot of time. This is due to the clipping operations involved in updating the border maps. Also adding patches requires these clipping operations and therefore also this takes a lot of time.

The time consumption of creating the new patches increases over time since the used polygons grow more complex over time, making it harder to work with. The number of patches added is constant over time, but the time increases from just under 100 milliseconds to over 600 milliseconds in 200 cycles. The same process is seen in the removal of patches. Figure 5.6(b) shows a spike at

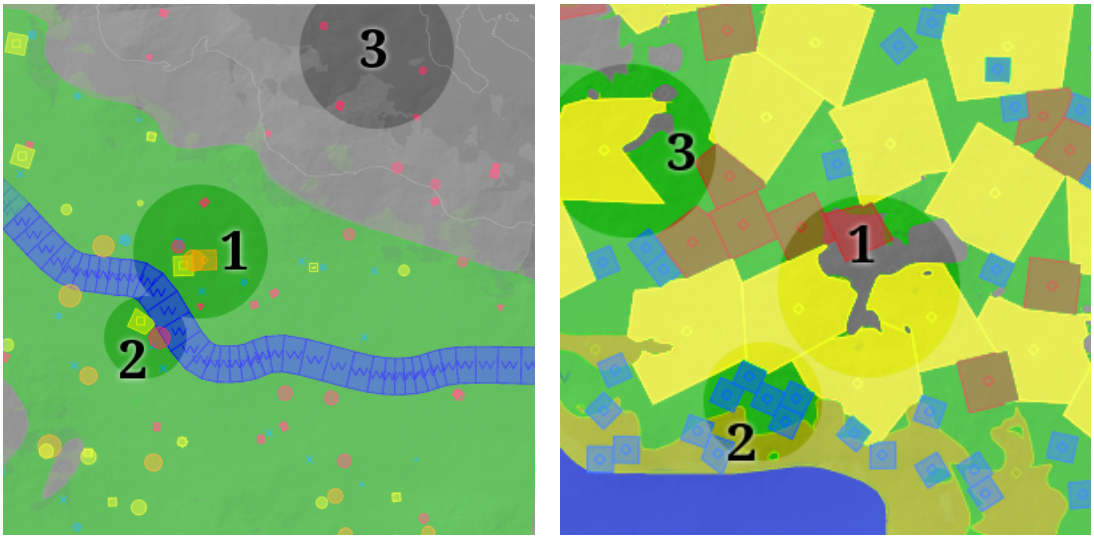
cycle 70 of 18 removals and a much smaller spike of 8 removals at cycle 190. Both appear to be of the same size in Figure 5.6(c).

The patch update step does not only include the removal of patches, but also the calculation of patch values and updating damage levels. This process however does not take a significant amount of time compared to the clipping operations. The patch creation phase consists of finding a proper location, growing traces, and finally updating the allowed building area borders. If the latter would not be included, the creation process would only take up twenty milliseconds.

In this simulation example, clearly more food was produced than consumed. In Appendix F another example is shown. The example shows that a slightly adjusted default database can lead to an urban area in which the government succeeds in bringing the resource production and consumption to an equilibrium.

5.2 Patch Allocation

As can be seen in Figure 5.2 the selection of locations is not always optimal. To investigate why the location selection did not always return good results, the selection is visualised by showing all tested locations as a coloured circle, where the size corresponds to the value of the location and the colour corresponds to the type of patch the location was tested for. An example output of this is shown in Figure 5.7(a). In this case the algorithm selected a patch type from all four categories: residential (red), industrial (orange), agricultural (yellow), and water supplies (blue).



(a) The selection of new locations (circles) per patch type category (colour). The black text at the top left of the image shows the stock values.

(b) When a new patch is created, it tries to follow the borders.

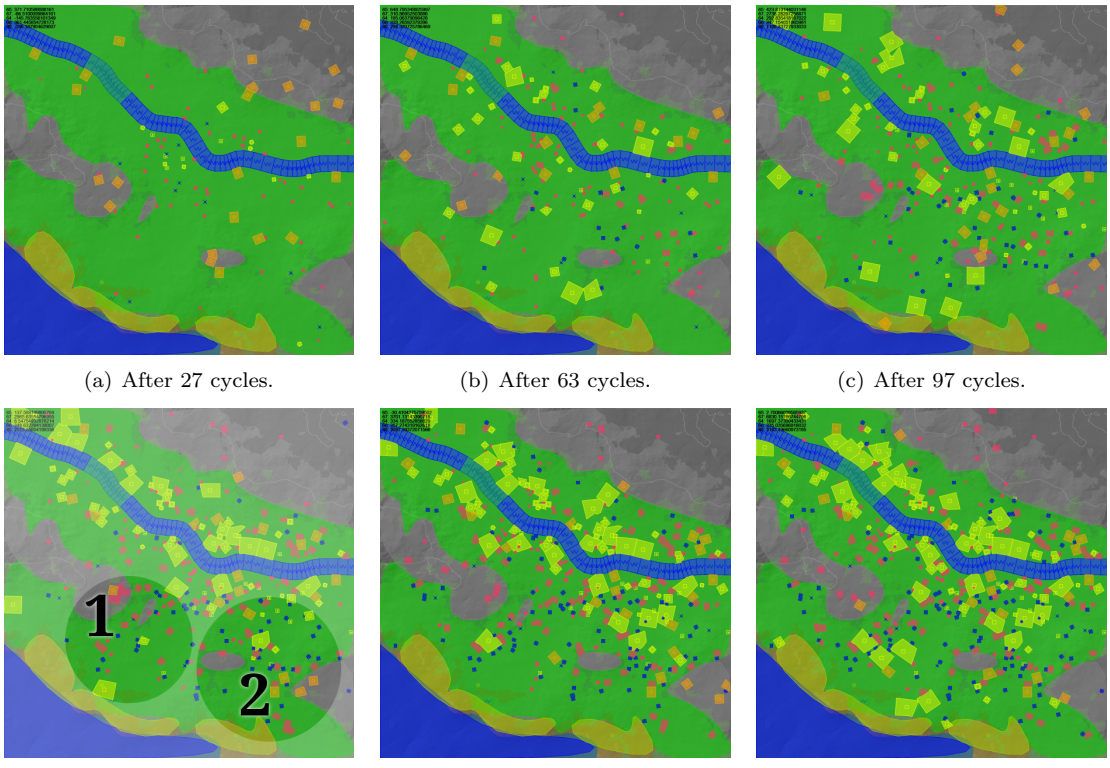
Figure 5.7: The selection of locations (a), and the creation of new patches that follows borders (b).

The value of a location is influenced by the resources available at that location and also whether a patch created at that location would snap to the local borders. For example, in the centre of the image, in region 1, two patches are situated; a farm and a quarry. There is an orange circle and a red circle nearby. The orange circle (industrial) is much bigger than the red circle (residential). The site is more suitable for an industrial patch than for a residential patch, partly because the location for the industrial patch type is close enough to a border to be snapped to it. There are also some houses further away that could offer some people to work at the quarry. Although the farm requires some manpower, it is not substantial enough to let the value of the residential location rise above that of the industrial location. In this case only the quarry was constructed (the orange patch underneath the circle), but both could have been picked. A location just under

the river (region 2) is also close to a farm, but with less houses near it. It therefore has a higher value than the tested location above the river. Other tested locations further from any patches like those in the mountains (region 3) tend to get much lower values, pushing the selection mechanism to pick a location closer to other patches. However, due to the nature of the selection procedure, there is still a small chance that such a location is picked, which can result in scattered patches.

After a location has been selected, the allocation manager starts to expand traces from the starting location to form the enclosing of the patch. In doing so the algorithm tries to snap to borders and follow them. This turned out to work effectively as can be seen in Figure 5.7(b). The borders of the agricultural patches (yellow) in the centre (1) follow the natural borders such as the mountains and dunes. Snapping and following borders also helps to align the patches as can be seen at the bottom left in region 2. The group of water supplies are snapped to each other and all have the same orientation.

Although snapping to borders and following them works as designed, it does leave some small gaps and can sometimes create patches that are not shaped as one would expect. The farm in region 3, for example, follows the rocks at the top and right of it, but has a spike at the bottom right of it. The trace at the bottom first expanded to the right and then had to wait for the other trace to come in view to jump to it. This however does not always happen as can be seen just a bit lower with another farm. This is probably due to the different starting point and orientation which can have an effect on the final shape of the patch.



(a) After 27 cycles. (b) After 63 cycles. (c) After 97 cycles.

(d) After 130 cycles. (e) After 162 cycles. (f) After 180 cycles.

Figure 5.8: This sequence of screenshots shows that the artificial patches tend to cluster around the river and how the urban area evolves over time.

To clarify how location selection and clustering work over time, a sequence of images showing the growth of an urban area is given in Figure 5.8. It shows a lot of scattered patches, but most patches tend to gather around the river since the river provides the important resource water. The first frame shows the patches created in the first 27 cycles. Many small patches were placed in the centre, close to the river. These are mostly small farms and simple houses, since bigger residential

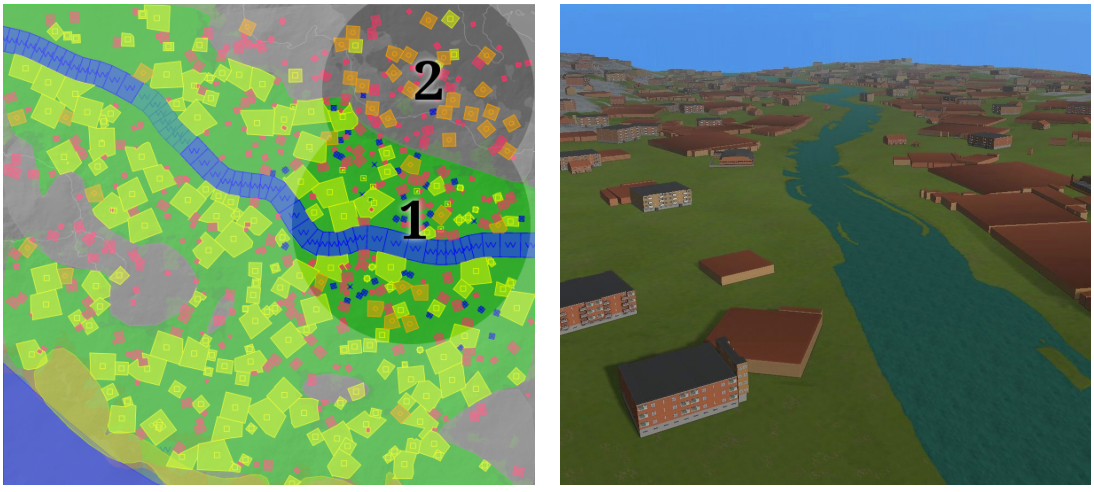
and agricultural patches were not available yet. The quarries are mostly placed in rocky areas and near the river. However, due to a lack of manpower and a surplus of the resource stone, most of these quarries are removed in the following 36 cycles. During these cycles all technologies have been executed and bigger patches have become available. This immediately results in bigger patches being placed on the terrain. A shortage of food results in building more farms near the river. Also some water supplies were needed and placed mostly south of the river. After 97 cycles more quarries are created since more manpower is available after the creation of more residential patches. In this period a lot more big farms are created along the river to supply the town with food. It also seems that a trail of water supplies and houses has formed from the centre to the sea at the bottom left (region 1). Another trail on the right (region 2) also contains some farms and quarries. In the period between the last two frames, the trails grow thicker because the government places more patches near them. The patches become better clustered but gaps still remain.

5.3 Alternative Settings

This section covers three alternative settings that show how changing the database or terrain can have a significant effect on the creation process of the urban area. The results are explained and compared to those from the default scenario.

5.3.1 Ocean Water Use Problem

In the default scenario only the river provides water, and although it is the only source of fresh water on the terrain, also the ocean on the bottom left can be a source of water albeit seawater. If the ocean is set to offer water, the resulting urban area is shaped completely differently from the default scenario urban area. Instead of clustering along the river, the patches are scattered over the whole map, as can be seen in Figure 5.9(a) and as three-dimensional geometry in Figure 5.9(b). The majority of patches consists of houses and farms. The wild scattering is caused by a surplus of water on the terrain. The range of influence of the ocean is large enough to easily cover the complete map and the large size of the ocean promises a large production of water, which makes the effect of water provided by the river negligible.



(a) Top view of the map.

(b) A 3D view of the map. The farms are shown as brown blocks.

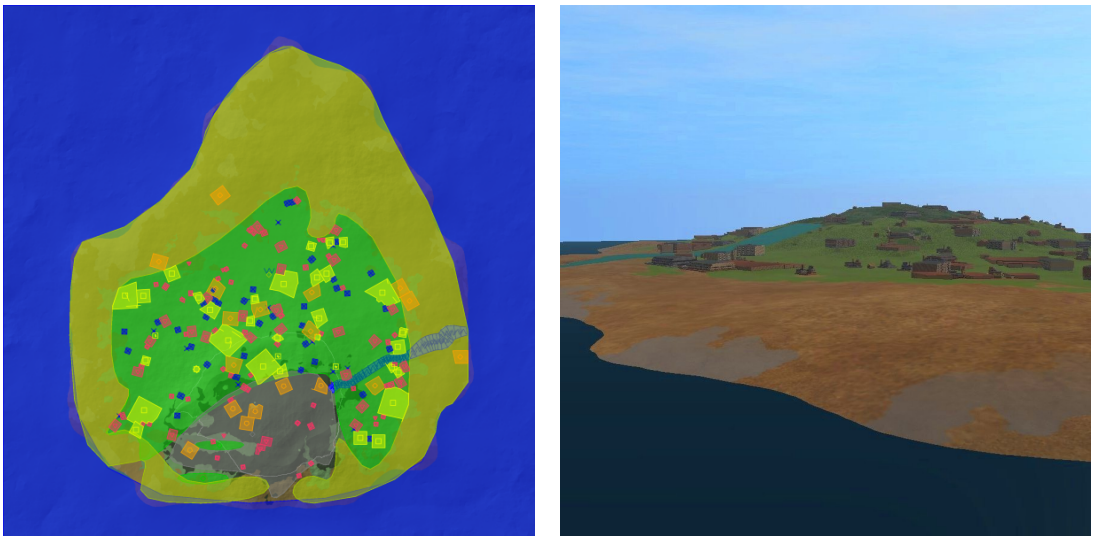
Figure 5.9: The default terrain map with patches scattered over it.

Although there is a great surplus of water in the area, a group of scattered water supplies is located on the right of the image (region 1). Interestingly water supplies are not located anywhere

else on the map. This can be caused by the larger number of residential patches in that area. Also the quarries seem to form such a group, just above the group of water supplies (region 2). Both groups are accompanied with many houses. Another interesting feature in this example is the farms in rocky areas (also region 2). These are constructed there by replacing low valued patches in those areas. This is a gap in the replacement process that does not check the underground of the patch when selecting candidates for replacement and thus allowing patches to be built on areas on which they could normally not be located according to the semantics.

5.3.2 Island Scenario

The default database is also tested on an island scenario (see Figure 5.10). This island has large dune areas surrounding a grassy mountain peak. On the right of the mountain is a creek. This creek however is too small to supply a large amount of water and therefore has no significant influence in clustering the patches, as can be seen in Figure 5.10(a). In this example less clustering can be seen compared to the result of the default scenario. The patches are scattered around the island within the zone of grass. Although water supplies and quarries can be built in the dunes, there are just a few situated there since the location values near the grassy areas, where the other patches are situated, are higher.



(a) Top view of the island.

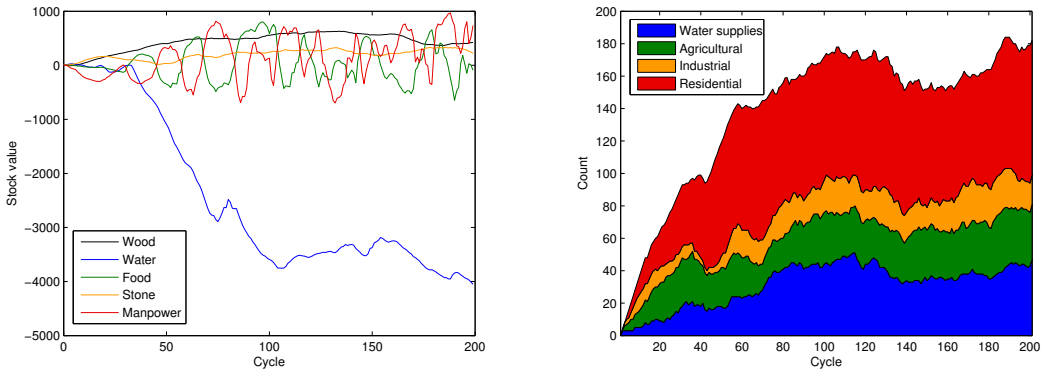
(b) 3D view of the island.

Figure 5.10: The island map with patches scattered over it.

The statistics in Figure 5.11 show a huge shortage of water which starts at cycle 30 and never returns to a value near zero. It does however stabilise for over 50 cycles after cycle 100, but continues dropping afterwards. The other resources however seem to be stable although the corresponding number of patches fluctuates. the government does not build enough wells to produce enough water to supply the big number of houses and farms although the patch type values (not shown in the statistics) of the water supplies are twice as high as those for the other patch types.

5.3.3 Small Database Scenario

To see how the simulator performs with a much smaller database, a new database was created with only three patch types (houses, stores, and factories) and two resources (urbanity and pollution). Houses produce urbanity and are repelled by pollution, factories produce pollution and are repelled



(a) The stock values.

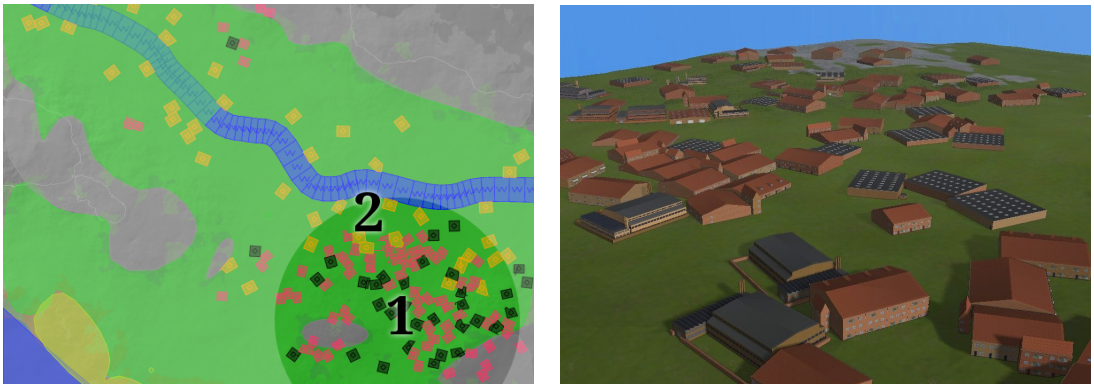
(b) The patch counts.

Figure 5.11: The statistics of the island scenario over 200 cycles.

by urbanity. Stores are attracted by both, but prefer urbanity. All three patch types are available from the start. The details of this database are presented in Appendix E.

This example shows that if the properties of the patch types are chosen wisely, the patches can cluster without the strong attracting force of a river. Also the resources in this example are under control and do not grow to extreme values as seen in the previously shown results. Note that the patch sizes are almost equal which makes it easier for the government to replace patches by any other type which can help the government to control the resource productions.

Figure 5.12(a) shows a result of the small database scenario after 199 cycles of development. The houses are attracted to each other by the resource urbanity which results in a cluster (in region 1). The factories on the other hand, are more attracted by the river and are scattered along it. They do not cluster together as the houses did. This could be a result of the river producing a small amount of the resource urbanity which repels factories. The stores (shown in black) are mostly attracted by the houses and are therefore scattered through the cluster of houses. The example also shows some clipping errors at the top of the cluster (2) between a few houses and three factories. These errors are not visible in the three-dimensional view in Figure 5.12(b) on the bottom left. SketchaWorld omitted the malformed patch in the model. The three-dimensional view of the cluster also shows that the patch sizes chosen were not selected to fit the used CGA grammars, but rather to make the patches pop out in the two-dimensional view.



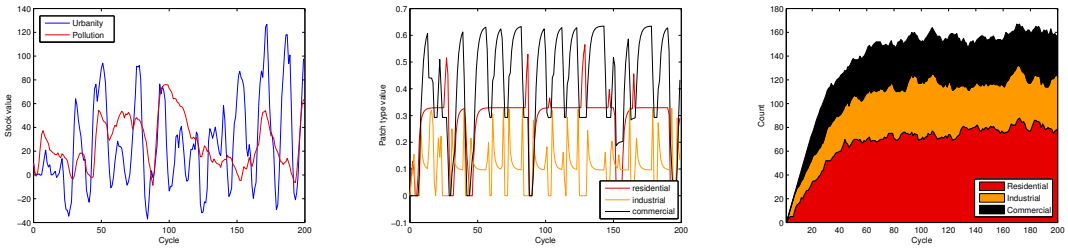
(a) A top view of the map after 199 cycles.

(b) The cluster of houses and stores in SketchaWorld.

Figure 5.12: The houses and stores are clustered together. The factories are located along the river.

In contrast with the default scenario, the resources are under control. They still fluctuate,

as Figure 5.13(a) shows, but they do not grow to large values. Interestingly there is rarely a shortage of the resource pollution. When thinking about the meaning of the term 'pollution' a shortage might sound strange, but in this scenario 'pollution' is just the name for a resources that tries to glue the factories together and to the river. The patch type values in this example also follow the stock values resulting in high values for stores (which are attracted by both resources) and low values for houses (which are repelled by pollution). The number of houses however is much greater than the number of stores and factories. Apparently the clustering of the houses allows the houses to exist longer and are therefore present in higher numbers. Another remarkable feature is the stabilisation of the number of patches after about 50 cycles. This is probably due to the predictable behaviour of the stock and patch type values. Moments at which this behaviour changes, for example, after cycle 150 when the pollution stock hits zero, results in spikes in the number of patches. Afterwards the distribution of patch types restores to its previous state.



(a) The stock values.

(b) The patch type values.

(c) The number of patches per patch type.

Figure 5.13: The statistics of the small database scenario over 199 cycles.

Chapter 6

Discussion

In this project, I developed an urban area development simulator for semantic game worlds. The system is capable of simulating urban areas over time based on the semantic information assigned to the different kinds of lots available. It proved to be too hard to create a compelling simulation that can achieve comparable results to those created by Weber et al. [9] and Vanegas et al. [23]. Reaching that level would take more than nine months and more programmers. Therefore the focus of this project was more on semantic relationships, resources, and events that shape the possibilities during the simulation.

During the project, some major concessions had to be done to finish in time. Features like instant three-dimensional feedback, roads, and traffic simulation were not added in the simulation. More on this can be read in Section 6.1. These changes to the original plan (see Appendix C) were needed due to unexpected problems during the implementation of UrbSim. The polygon clipper that was used for updating the borders for example, was not working as expected. Also some tested solutions to this problem did not solve all bugs. This is discussed in the second section, Section 6.2, along with some possible big changes to the design and implementation that would get around the problem. The last section covers the discussion of the results from the previous chapter. It evaluates how the simulator performs in terms of clustering, resource control, and speed. It gives suggestions for improvement and also compares this methods to other methods to generate urban areas.

6.1 Semantics

The generic nature of UrbSim allows the user to configure every small detail of the patch types and the government. This however can be overwhelming to the user since so much parameters are involved. To help the user overcome this problem, the custom patch types can inherit their configuration from the default patch types. The user can then override a small custom set of parameters. Though still it is not very user friendly and can still be confusing to the user. It was not the aim of this system to create a grand user interface to let the user design the behaviour of the simulation. Such an interface would be a good improvement to help users to configure the simulation. A feature that could improve the control for example would be a slider that changes the preferences of the government during the simulation so the user can see the result directly. Another example is a feature to add new events during the simulation. This would make it a lot easier to add an event compared to editing the database via Entika.

Given that the user knows how to properly change the database through Entika, this does not mean the supplied patch type information will lead to a stable simulation. It might result in a bad selection of patch types resulting in surpluses and shortages running out of control and location selection might fail to cluster the patches properly. If the user does specify enough different patch types in the database, the algorithm might not be able to converge to an equilibrium. For each consumed resource, either via building costs or via negative production, there should be a patch

type available that produces it. On top of that, the stability of the simulation can be improved by adding different kinds of patches that produce the resource, so there is more choice. One could vary between sizes and production scales or between the costs of the solutions.

If the user however does not follow these guidelines, it could result in an unstable system that first creates new patches, then discovers that these patches cannot be supplied with the right resources and removes them from the area after a few cycles. This process could repeat itself until the government does have the resources to create any patch. The urban area will therefore pulsate, although the maximum and minimum size and the duration of the pulses can differ.

6.1.1 Events

Currently only two events, namely technology and destruction, are implemented. The technology event has a clear contribution to the system by changing the options throughout the simulation. The destruction event on the other hand does not have a huge impact on the simulation. The event is lacking semantic information. A hurricane for example would damage a wooden structure differently than it would damage a concrete building. An earthquake on the other hand could result in the opposite effect. Therefore the destruction event should be split up into specific types of events and extended with more semantic information.

It would also be nice to extend the set of events with other influences that could shape the simulation and therefore the urban area. These new events could include a government change that changes all preferences or goals of the government, or a change in production rate caused by drought or abundant rainfall, and land deformations and floods that could change the terrain and the objects on it.

6.1.2 Roads and Transportation

An important feature lacking from UrbSim is roads. This feature could greatly enhance the resulting urban areas. Patches are heavily dependent on transportation possibilities and would therefore be likely to be near rivers or roads. In this case a resource called 'transportation' could be used to cluster patches along the roads. The implementation would try to find parts of the urban area that are lacking transportation possibilities and try to connect these to the known road network.

A possible implementation for roads could be based on a resource labelled 'transport'. It could start by locating an area in need of transport. From this point, the closest point in the existing roads network should be found. If no roads are present on the terrain, another location in need of transport might be selected. The points can then be connected by growing adaptive roads (as done by Kelly and McCabe [12]). By iteratively adding road segments between the points, a road is generated in between the points. During this process, the segments will favour areas that are in need of transport, but they would snap to existing road segments to extend the current road network.

The lack of roads implies that there is no simulation of the transportation of resources. This traffic simulation could be used for determining the load on the road network. This load factor could be used to upgrade or downgrade roads. Also the placement of patches could be enhanced. Commercial patches for example would prefer busy roads for example while houses are attracted by quiet areas. The transportation of resources itself could be used for adding some distance between resource producers and resource consumers, allowing for separation between industrial and residential areas. Now this has to be covered by changing the influence radii and creating extra resources (like a population resource) to cluster same type patches more.

6.1.3 SketchaWorld Terrain

The SketchaWorld terrain that is used in the simulation is not fully imported. The layers that contain roads and buildings are excluded. It would be a great enhancement to also incorporate these layers in the simulation so one can already start with a filled urban area. The roads that

were designed by hand in SketchaWorld could for example help shape the simulated urban area since the newly created patches that are in need of transportation are likely to cluster along the roads.

Another addition to UrbSim would be the possibility to import another terrain source. For example a designer might want to import a terrain it made using another editor than SketchaWorld such as Maya [36] or 3D Studio Max [37]. Also bitmap images could be used as input, as done in [14] and [26].

6.1.4 Environment Simulation

The created simulation only simulates the artificial patches created by the government instance. Though a simulation of the vegetation and other terrain features like the water might also enhance the simulation. When woodcutters for example chop down trees, the trees should be removed from the terrain and the woodcutters would have to move to chop down other trees and when all trees are cut down, it would take a long time for new trees to grow to a proper size to be chopped down. The same could be done for fish in the sea and rivers; when there are too many fishermen, no fish will be left and other food source has to be found.

6.2 Patches

The simulation is based on the creation, maintenance, and deletion of patches. This process is not implemented flawlessly. Especially the use of borders gave problems. Some solutions and improvements are proposed in this section.

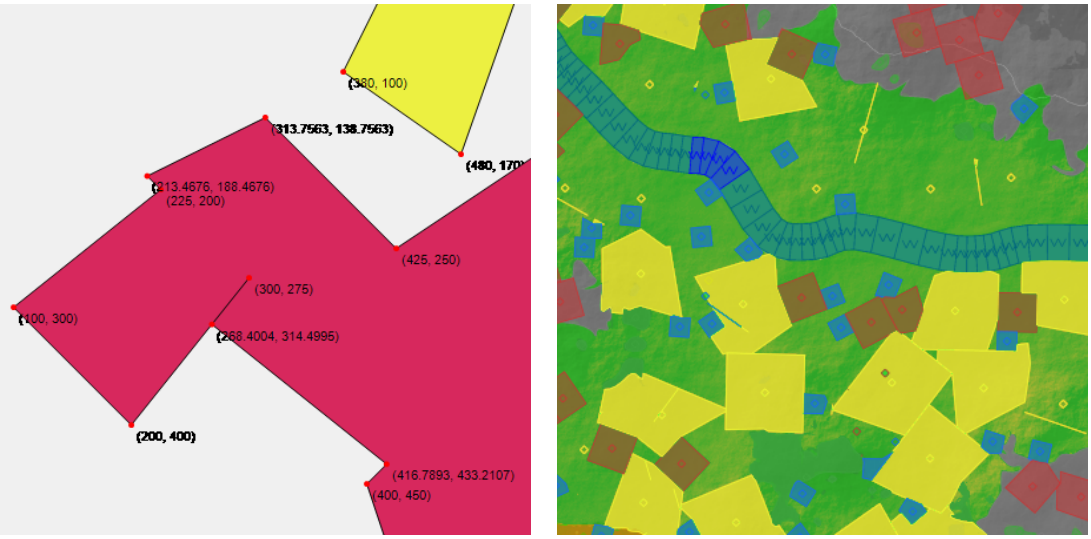
6.2.1 Polygon Clipper

During the simulation the borders are constantly updated to keep track of where new patches can be built. This is done by subtracting the polygons that are constructed for new patches from the allowed building area using a polygon clipper. Also when removing patches, the polygons are added to the area again. The used polygon clipper from the General Polygon Clipper library (GPC) [34] however does not cope well with polygons that overlap just on their borders. So when only edges overlap, this can result in strange artifacts like thin protrusions and malformed patches, as shown in Figure 6.1.

Old borders are thus kept in the allowed area polygon which is used for constructing new polygons. These can be just a few free lines, but sometimes even L- and U-shaped polylines. On one hand this would encourage the system to retain some structure over time, but on the other it disrupts the process of creating new patches. The patch creation algorithm finds the borders that are left on the terrain and will follow them, resulting in malformed patches. Moreover the artifacts in the allowed area cause the polygon clipper to slow down, since the malformed allowed area and patches are harder to process, and sometimes even crash due to errors in the patches created by the polygon clipper itself. This becomes worse over time since errors in the polygons are rarely resolved.

A way to get around this problem would be filtering the produced polygons, drop the concept of borders used in this way, or using another polygon clipper that does the job well. Two of these were explored during the project. Filtering malformed polygons was tried as explained in Chapter 4 by testing polygon size, self-intersections and clipping the polygons using the allowed area. This can however also cause errors to be created in the polygons. Checks could be added to delete contours that do not span an area. This check could also be used for creating a list of allowed areas of a certain size that could be directly filled with a patch without the need of tracing borders.

The other option for resolving this issue that was explored was using another polygon clipper that is specialised in solving issues of edges that are overlapping. One month was spent on creating a new polygon class and an integrated polygon clipper to replace the GPC used by SketchaWorld. This class would speed up the clipping process, since the polygon would not have to be converted



(a) A union of three patches (a quadrilateral on the left, an L-shape on the right and a shape in between these two) resulted in one large polygon with an protrusion.

(b) Due to clipping errors, thin patches and patches with protrusions were created. In the latest version of UrbSim these patches are filtered out when detected.

Figure 6.1: The polygon clipper does not cope well with edge-on-edge intersections resulting in malformed patches.

forth and back for using the GPC and all edges were saved in a quadtree to reduce the intersection checks. The latter could also be used for the patch creation step. Unfortunately the new polygon clipper did not meet my requirements and remained to cause bugs. Therefore I decided to return to the GPC although it did not perform well either. The newly created polygon class was kept to have an advantage in the patch creation process.

Discarding the idea of using the updating borders concept was considered, but was not executed due to time constraints. The following subsection will explain this approach in detail.

6.2.2 Borders

Currently the borders are saved for both the individual patches and the allowed building area boundaries for each patch category. The latter is a point of concern, as explained in Subsection 6.2.1, due to the clipping operations performed on it which take a lot of time. The allowed building areas were originally meant for enabling random point selection within the boundaries and for making the patch creation easier. The selection of points for starting the creation of patches however becomes slower as the allowed building area polygon complexity increases. This is due to the selection of random points based on a triangulation of the area.

In Chapter 4 was stated that an alternative approach for selecting points could be generating random points and check whether these lie within the allowed area. This does not guarantee that points will be found and finding proper points will take longer when the allowed area decreases in size. Though this method can offer a good solution. Instead of checking whether the generated points fall within a polygon that reflects the allowed building area, the algorithm should check whether it falls in an already generated patch or somewhere on the terrain. This could be very fast since it only has to check a point within a quadtree of patches. If it hits a patch, this patch could be added to the list of patches that could be replaced during this cycle. If it does not hit a patch, the allocation manager should try to create a polygon from that location, based on the type of ground. This could be improved further by allowing the user to set constraints, such as that a location can only be used if a certain resource is available. This would allow the user to force the system only to select locations near roads (which offer the resource 'transport') for example.

This could replace the current maintenance selection algorithm and the random point selection algorithm.

This approach would make the allowed building areas superfluous if these were not needed by the patch creation algorithm. The borders were introduced to make it easier to detect whether two patches were snapped to each other. This is needed when the patch creation algorithm snaps to one of the adjacent polygons and follows it. If the algorithm would not know there is a polygon snapped to the polygon it is following, it could get stuck in between the two polygons. To overcome this, all created patches were merged and the inverse was used as boundaries of the building area. The problem could however also be solved by checking intersections when the algorithm follows the polygon it is snapped to. This would save a lot of computations used for clipping though it would make the algorithm more complex.

A positive effect of keeping track of the allowed building area though could be tracking the allowed areas by size. This way the patch creation algorithm would not have to expand traces to find a polygon, but when an allowed area is already of the proper size, it can directly use it. This would also help in filling the gaps created during the simulation making the created urban area more appealing.

After these considerations the random selection and patch creation without the allowed area borders is very compelling. If the simulation were to be developed further, this approach should be used in favour of the current system. This would make the simulation both faster and more reliable, given the patch creation algorithm does not fail in the new situation.

6.2.3 Patch Influences

Another part of the system that can be improved is the way patches influence their surroundings. Currently the influence is based on the bounding circle of the patch and a scaled version of that bounding circle to indicate the outer edge of influence. This can result in too large influence areas for irregular shaped patches. One solution is to split up irregular shapes into multiple smaller but regular shapes. The current implementation misses the option to split up large natural patches. Only for the rivers this option has been implemented. Splitting up these patches is needed to calculate the distance weights properly. These weights are based on the bounding circle and the corresponding influence range. The bounding circle should fit tightly around to make the weights accurate. To do this the large natural polygons could be slit up recursively until the area of the created parts is under a given threshold.

Another way to resolve this problem is using a grid of influence values that are updated throughout the simulation. Each cell would then carry the production values for all resources produced for that location. When a new patch was added, the grid cells would be updated with the resource production values from that lot based on the distance to that lot. In an early implementation of the simulation this method was applied, but this turned out to be rather slow and was therefore discarded.

6.3 Results

This section evaluates the performance of the simulator based on the results presented in Chapter 5. The location selection and corresponding clustering are discussed, along with the shapes of the created patches. The resource control is analysed and the duration of the cycles is discussed. This section ends with a comparison of the proposed method and the other methods which were introduced in Chapter 2.

6.3.1 Test Parameters

The number of cycles of the simulations presented in the previous chapter was bounded to 200 cycles. This limit was chosen to be able to run the simulation without the danger of it crashing due to a memory shortage. For adding and removing patches the polygon clipper is put to work

to clip the allowed area borders. The polygon clipper, as explained before, does not always do this correctly. This can lead to corrupted polygons returned by the polygon clipper. When the clipper has to process the corrupted polygons, it can slip into an infinite loop if the computer had an infinite amount of memory. Since the latter is not the case, the polygon clipper stalls the program for a minute and then crashes the simulator. The moment at which this event occurs largely depends on the input from the database (previous runs could get up to 500 cycles without any problem) and the seed set for the random number generator. To resolve this problem, the clipper could be adjusted or replaced or even made obsolete by changing another strategy than updating the borders every time a patch was created. It can also be postponed by changing the database to allow for less removals. Also replacing patches instead of removing them could help to postpone the crash.

6.3.2 Patch Allocation

The results from the previous chapter show that the simulator has its difficulties with making a compact clustering of patches. Many patches tend to scatter over the terrain, leaving gaps in the urban area. This inconvenient placement of patches is caused by how the database dictates how patches are repelled and attracted. The database should be tuned further to make the clustering better, although it is hard to combine this without letting the resources run out of control. Instead of editing the weights for the resources per patch type, the weight for neighbouring patches could be increased or an extra resource could be introduced that pulls all patches together. A solution on the side of implementation can be found using only the best locations for fitness proportionate selection. For example only the top five locations are compared. This last option would also make it easier for the designer of the database.

Another source of gaps in the urban area is the removal of patches. Patches are removed when damaged, but replacing them rather than removing them could help to make both the simulation faster since less clipping operations are needed and the clustering is improved since no patches are removed. Though removal should probably be still available for cases in which the patch has no neighbouring patches.

The creation of patches itself works as intended in most cases. The traces that grow to form a patch border follow the borders on the terrain as was shown in Figure 5.7(b). Furthermore it helps to align the newly created patch with neighbouring patches and lets them snap together. However during the creation the traces sometimes snap too early, leaving small gaps in between the building area borders and the created patch. The implementation should be improved to also fill up these gaps. This could be done by analysing the resulting patch and the neighbouring polygons to see if small gaps were created and if so these should be merged with the patch. In some cases when objects are encountered in the area where the patch grows, one trace is snapped to it while another might be extended in its original direction. Once these are joined, the second trace might have created a spike (also visible in Figure 5.7(b)). This results in unexpected shapes, which could be a problem for the CGA Grammars that are executed on the patches. This can be either solved by writing smart CGA Grammars or improving the allocation algorithm so that one trace does not grow much further in a certain direction compared to the other trace.

6.3.3 Resource Control

The next major part of the simulation next to patch allocation is the control of resources, which is done through patch allocation and removal. The selection of patch types is based on their values. The average values presented in the previous chapter in Figure F.2(a) however are quite low and are never near the maximum value of one. This is because the patch types of one category are not all at their maximum value since the patch type values are normalised by the goals which are never exceeding the biggest production value and in the case of the default database just one patch offers this value while the others produce less. This is probably also why more big farms are created once these become available since these get a higher patch type value than the other

farm which produce less food and use less water and manpower. Only at lower goal values this will change.

The patch type values follow the goals and stock values as intended, when the stock drops below zero, the goal becomes positive and patch types offering the requested resource increase in value. As explained in Chapter 3 the goals try to bring the stock values to zero by converging to the negated stock value, limited by the biggest production values. Figure F.3(b) shows that this is working correctly. Also when bigger production values become available through the execution of technologies or when the stock changes from a shortage to a surplus or vice versa, the goal values adapt to the new value. A clear example is the water supply category that has a high value for a long time until there is a surplus of water. Then the goal value for water and patch type value for water supplies suddenly drop. This process helps to finally restore the production an consumption of the different resources and the resources start to alternate between shortage and surplus during the process.

The current implementation of patch type values is not perfect. The values stay constant as long as the goals do not change in value, but the stock values might change. One might want to have weights linked to the resources that are most wanted to increase the value of their producers more. this could be done by sorting the absolute stock values and assigning the biggest weight to the resource with the biggest stock value and the smallest weight to the resource with the smallest stock value. This way the patch type values could be guided to solve the biggest surplus or shortage first, rather than trying to solve all surpluses and shortages at the same time.

The shortages and surpluses from Figure F.3(a) can become quite extreme sometimes. This is a result of both bad location selection and bad patch type selection. Though the simulation does manage to solve the big shortages and surpluses after many cycles, one would still prefer a less extreme shortage and surplus development. Once the algorithms for location and patch type selection have been improved, these problems might still exist. The best way to resolve them would be by altering the values in the database, and maybe by adding new patches, to make it easier for the simulator to find an equilibrium between consumption and production. Another solution that might help is letting the number of added patches depend on the needs of the government. Rather than adding a fixed number of patches each cycle, the government might add more patches when there are bigger shortages and surpluses then when the stock values are smaller.

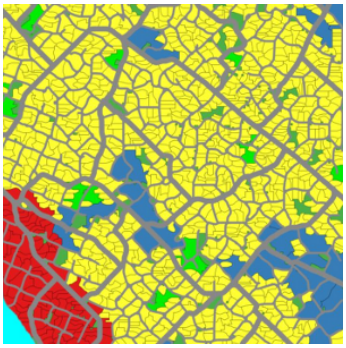
6.3.4 Time Consumption

The duration of the cycles increases over time. This is due to the time the polygon clipper takes to process the allowed building area borders that get more complex over time. Also errors made by the polygon clipper make the borders more complex and harder to work with resulting in more time used for clipping polygons. After 200 cycles the total time used for one cycle can take up to 1500 milliseconds and it is still gradually growing. This is too much if one would like to rerun a simulation of 200 cycles when he adjusted one variable and wants to know the effect. One would rather aim at less than 10 seconds for the complete simulation of 200 patches. A minimum of time use would also be nice when it would be integrated into a game in which the game world evolves. In that case just a few milliseconds could be dedicated for simulation, since a game developer wants the game play to be as smooth as possible and simulation should not slow down the game.

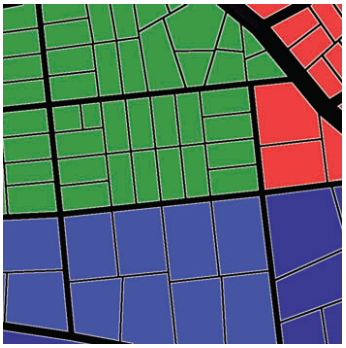
If the polygon clipper would not be used, the time usage would be less than 30 milliseconds per cycle and the slowing down would be less severe as in the current version. Another possibility to reduce the load would be to stop removing patches and just replacing them, though still a big increase in time usage would be seen due to the adding of patches. To improve the performance even further parallel computing for example with the GPU could be used. Most of the updates done in the program can be computed in parallel. Also the polygon clipper of the GPU might be used for clipping the borders which would greatly increase the speed of the simulation.

6.3.5 Comparison of Urban Area Simulations

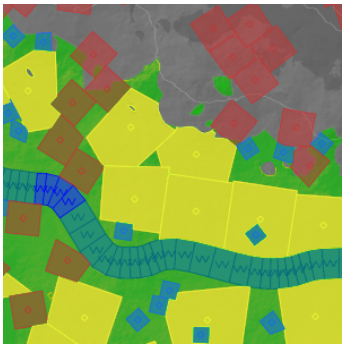
When this simulation is compared to the existing urban area simulations discussed in Chapter 2 one will immediately notice that those all include the creation of roads. In contrast with UrbSim the other approaches are based on creating a road network first and then placing the patches. This last steps is done by subdivision of enclosed areas [23], [9] or by flood filling as in the grid-based approach from Lechner et al. [26]. The latter has some similarity with the growing of traces to form a border of the patch, though it rarely produces regular shapes. The subdivision approach does not produce irregular shapes at all. Figure 6.2 shows the differences between the resulting patches from the different approaches. Although the roads are not necessary for generating all patches, it is still needed to make the simulator practical as a procedural content generator. It would also greatly enhance the generated urban areas since it allows for better clustering.



(a) The bucket fill technique used by Lechner et al. [26] creates irregularly shaped patches.



(b) Subdivision, as used in [9], results in convex patches.



(c) The patches in UrbSim are shaped by the local terrain features.

Figure 6.2: The differences between the patch shapes from the different approaches.

Just like the approach described by Weber et al. [9], UrbSim does not rely on a grid like the approach by Lechner et al. [26]. This allows for free placement of patches and also free geometry that is not bounded by fixed sized and oriented square cells. A possible improvement is shown by Vanegas et al. [23]. The approach presented in [23] uses a grid for storing values, but does use geometric modelling. This was one of the discussed solutions for tackling the patch influence issues described in Subsection 6.2.3.

Another difference with many solutions proposed to procedurally generate urban areas is the selection of the city centre. Instead of letting the user select a location, the system itself tries to find the best location. This however can be undesired by a designer that wants to have more control. A possibility to set a starting point for the city might therefore be a useful contribution, but this would also weaken the semantic background of the urban area.

An advantage of the simulator is the use of a seed for the random number generator to generate an urban area. This way a simulation can be ran again and it will return the same output. This can also be used to compare how the urban area would evolve differently when another patch type was introduced at a certain moment in time.

The current system is not ready yet for practical use. It lacks an easy interface for interacting with the resources and patch types. It is therefore hard to control the simulation. Also the geometry is not generated on the fly but afterwards through SketchaWorld. Since not many CGA Grammars are available in SketchaWorld, the variety of buildings in the resulting three-dimensional model will be small. Though there is an option for the user to add them manually to the configuration of SketchaWorld and in the database for the simulation.

In contrast with many other urban area simulators, UrbSim includes meaning to the placement and shapes of the lots by taking into account the terrain features and the resources available. These in combination with economic fluctuations, disasters, and technologies help to shape the created urban area. The creation process is not as static as in the other simulations, but rather tries to

incorporate changes that might during the development of the urban area. This ensures more meaning is added to the shape and the configuration of the urban area.

Chapter 7

Conclusion

Due to the growing need for content in games, more money is spent on creating game content. Many games are situated in urban areas, complex collections of buildings and roads. To create these by hand is costly, and therefore procedural techniques have been proposed in the past decades to automate the creation process to reduce the amount of work needed to create these complex scenes. The creation process and the resulting urban areas of these methods, however, lack meaning and cannot be applied directly in a semantic game world due to the lack of semantic information in the model.

This raises the following question: *how can one incorporate semantic information and history in the generation process of a virtual urban area suitable for use in a semantics-based game?* To answer this question, I developed a simulator called *UrbSim* that simulates the development of an urban area over time for a specific semantic game world. This game world includes a terrain of which the features influence the growth of the urban area by providing resources to their surroundings. The game world also contains a semantic database that holds all information about the terrain features, but also about the type of buildings that may be constructed. For each of these buildings also information is provided through the database about their needs and productions in terms of resources. The semantic database can also include technologies and disasters to change the course of the simulation over time.

During a simulation one instance, named the government, leads the creation process. The decisions made by the government are based on the properties of the game world it operates in and the shortages and surpluses of resources experienced by its patches. These patches are the lots on which the buildings are built and carry the semantic information supplied for the buildings. Based on the shortages and surpluses the government calculates it goals. These goals aim to bring the production and consumption of resources to an equilibrium. In turn the goals are used to calculate which types of buildings (patch types) are needed by giving a score to each of the available patch types. The availability of patch types is determined by technologies. Each technology is bound to a specific point in time and can both enable and disable patch types.

When all available patch types have been given a score, fitness proportionate selection is applied to select a number of patch types to be built. Patches that fit the goals of the government better, gain a higher score and are therefore more likely to be picked. To find a proper location for each selected patch type, a number of random points is spawned in the allowed building area of the patch type and evaluated. Based on the preferences of the patch types, its neighbouring patches, and fitness proportionate selection a location is selected. To start the creation of a patch, this location is snapped to the closest border. From there a polygon is created that follows the local borders and is approximately the intended size (specified per patch type). Once the edges of the patch are set and the semantic information is linked to it, the patch will start to interact with its neighbouring patches by producing and consuming resources and influencing the placement of new patches.

Over the course of time the patches loose value due to shortages of resources they consume or due to damage gained by destruction events. Patches with a lower value are more likely to be

removed or replaced by the government. In the case of replacement, a new patch type is selected and applied that suites the size of the patch and the local production of resources. As before, the selection is fitness proportionate and the values of the patch types depends on the goals of the government. In the case of removal, the patch is just removed and the land on which is was situated is released for later use. By repeating addition, replacement, and removal of patches over time, the urban area evolves and is shaped by the terrain, its resources, the technologies available to the government and disasters that occur.

The resulting urban areas show clustering of patches around resource producing patches. For example, rivers provide water, patches that are in need of water cluster along the river. These patches may produce food or manpower, which attracts other patches, resulting in a cluster near the river. The selection of patch types also follows the needs of the government. If there is a shortage of a resource, the government tries to counter that by creating more patches that produces the needed resource. This also works the other way around. If there is a surplus of a certain resource, the government will try to build more patches that consume the resource. This allows the government to control the resource production and consumption. The selection of patch types is also affected by technologies. At the start of the simulation, the government can only select small and simple patches. When the technology allows for more advanced patches, the government takes advantage of this by selecting these new patch types to control the resources.

Despite the results correspond to the objective of the project and its design, the simulator is not yet ready for use by game designers. Currently, it is hard to design a semantic database that results in a stable simulation: a simulation in which resources are not running out of control and in which the patches are located in compact clusters. Furthermore, many simulations may end up creating urban areas with many gaps in them or even scattering patches because they can repel each other by offering certain resources.

One of the major features that has not been added to the simulator is roads. Roads will help to enhance the urban area, and to provide options for clustering (like the river does) and for transport of resources. The latter would allow for patches growing far away from the needed resources, but still being supplied to them. Another downside of the simulator is its speed. This is mainly caused by the duration of updating the borders. When a patch is added or removed, the borders of the allowed building area are updated. This is done with a polygon clipper. This process is slow and error-prone which can result in artifacts in the patches. To solve this problem, and to greatly speed-up the simulation, one could remove the allowed building area borders from the implementation. The new implementation could rely on selecting random locations in the whole area and only selecting points that are on a ground type on which the patch type may be build. When the selection hits another patch, the government might consider replacing the hit patch. This strategy will probably be more reliable and be a lot faster than the current implementation.

Although many improvements can be thought of, UrbSim does add more meaning to both the creation process and the urban area itself than previously proposed solutions to procedurally generating virtual urban areas. It is able to produce urban areas that contain both a history and semantic background, each created element has semantic data linked to it, allowing it to be used in the semantic game world it was created for.

Bibliography

[1] Ubisoft Montreal, "Assassin's Creed II," 2009.

[2] Rockstar North, "Grand Theft Auto IV," 2008.

[3] D. Ebert, S. Worley, F. Musgrave, D. Peachey, and K. Perlin, *Texturing & Modeling, a Procedural Approach*, 3rd ed. Elsevier, 2003.

[4] K. Perlin, "Making noise," http://www.noisemachine.com/talk1/, 1999.

[5] O. Deussen, P. Hanrahan, B. Lintermann, R. Měch, M. Pharr, and P. Prusinkiewicz, "Realistic Modeling and Rendering of Plant Ecosystems," in *SIGGRAPH '98: Proceedings of the* 25^{th} *Annual Conference on Computer Graphics and Interactive Techniques.* ACM, 1998, pp. 275–286.

[6] R. Smelik, T. Tutenel, K. de Kraker, and R. Bidarra, "A declarative approach to procedural modeling of virtual worlds," *Computers & Graphics*, vol. 35, no. 2, pp. 352–363, 2011.

[7] K. Musgrave, "Mojoworld," http://www.pandromeda.com/products/, 2006.

[8] A. Postma, "On higher ground," http://www.pandromeda.com/gallery/, 2004.

[9] B. Weber, P. Müller, P. Wonka, and M. Gross, "Interactive geometric simulation of 4d cities," *Computer Graphics Forum: Proceedings of Eurographics 2009*, vol. 28, pp. 481–492, April 2009.

[10] S. Greuter, J. Parker, N. Stewart, and G. Leach, "Real-time Procedural Generation of 'Pseudo Infinite' Cities," in *GRAPHITE '03: Proceedings of the* 1^{st} *International Conference on Computer Graphics and Interactive Techniques in Australasia and Southeast Asia.* ACM, 2003, pp. 87–94.

[11] J. Sun, X. Yu, G. Baciu, and M. Green, "Template-based generation of road networks for virtual city modeling," in *VRST '02: Proceedings of the ACM Symposium on Virtual Reality Software and Technology.* ACM, 2002, pp. 33–40.

[12] G. Kelly and H. McCabe, "Citygen: An interactive system for procedural city generation," in *Proceedings of GDTW 2007: The Fifth Annual International Conference in Computer Game Design and Technology*, 2007, pp. 8–16.

[13] S. Groenewegen, R. Smelik, K. de Kraker, and R. Bidarra, "Procedural city layout generation based on urban land use models," in *Eurographics 2009: Short Papers.* Eurographics Association, 2009, pp. 45–48.

[14] Y. Parish and P. Müller, "Procedural modeling of cities," in *SIGGRAPH '01 Conference Proceedings*, 2001, pp. 301–308.

[15] G. Kelly and H. McCabe, "A survey of procedural techniques for city generation," *Institute of Technology Blanchardstown Journal*, vol. 14, pp. 87–130, 2006.

[16] E. Catmull and J. Clark, "Recursively generated b-spline surfaces on arbitrary topological meshes," *Computer-Aided Design*, vol. 10, no. 6, pp. 350–355, 1978.

[17] D. Eberly, "The minimal cycle basis for a planar graph," 2005.

[18] P. Prusinkiewicz and A. Lindenmayer, *The Algorithmic Beauty of Plants*. Springer-Verlag, 1990.

[19] A. Lindenmayer, "Mathematical models for cellular interactions in development, parts i and ii," 1968.

[20] G. Esch, P. Wonka, P. Müller, and E. Zhang, "Interactive procedural street modeling," in *SIGGRAPH '07: ACM SIGGRAPH 2007 sketches*. ACM, 2007.

[21] G. Chen, G. Esch, P. Wonka, P. Müller, and E. Zhang, "Interactive procedural street modeling," in *SIGGRAPH '08: Proceedings of the* 35^{th} *Annual Conference on Computer Graphics and Interactive Techniques*, vol. 27, no. 3. ACM, 2008, pp. 1–10.

[22] PROCEDURAL, "Cityengine," http://www.procedural.com, 2008.

[23] C. Vanegas, D. Aliaga, B. . Beneš, and P. Waddell, "Interactive design of urban spaces using geometrical and behavioral modeling," *ACM Transactions on Graphics: Proceedings of ACM SIGGRAPH Asia 2009*, vol. 28, no. 5, pp. 1–10, 2009.

[24] P. Waddell, "Urbansim: Modeling urban development for land use, transportation and environmental planning," *Journal of the American Planning Association*, vol. 68, no. 3, pp. 297–314, 2002.

[25] T. Lechner, B. Watson, U. Wilensky, and M. Felson, "Procedural city modeling," in 1^{st} *Midwestern Graphics Conference*, 2003.

[26] T. Lechner, U. Wilensky, M. Felsen, P. Ren, B. Watson, S. Tisue, A. Moddrell, and C. Brozefsky, "Procedural modeling of urban land use," in *SIGGRAPH '06: ACM SIGGRAPH 2006 Research posters*, 2006.

[27] Electronic Arts, "SimCity 4 Deluxe," 2003.

[28] J. Kessing, T. Tutenel, and R. Bidarra, "Designing semantic game worlds," in *Proceedings of the The third workshop on Procedural Content Generation in Games*. PCGames, 2012.

[29] R. Ensemble Studios, Big Huge Games, "Age of Empires," 1997.

[30] Westwood Studios, "Command & conquer," 1995.

[31] Wikia, "Age of empires tech-tree," http://ageofempires.wikia.com/wiki/Age_of_Empires_Tech-Tree, 2012.

[32] J. Holland, *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI: University of Michigan University Press, 1975.

[33] T. Bäck, *Evolutionary Algorithms in Theory and Practice*. Oxford University Press, 1996.

[34] A. Murta and T. Howard, "General polygon clipper library," http://www.cs.man.ac.uk/ toby/gpc/, 2011.

[35] P. Selinger, "Potrace: a polygon-based tracing algorithm," http://potrace.sourceforge.net/, 2003.

[36] Autodesk, "Autodesk maya," http://usa.autodesk.com/maya/, 2012.

[37] ——, "Autodesk 3ds max products," http://usa.autodesk.com/3ds-max/, 2012.

[38] M. Alberti and P. Waddell, "An integrated urban development and ecological simulation model," *Integrated Assessment*, vol. 1, pp. 215–227, 2000.

Appendix A

Planning

A.1 Prototype I

- Continuing literature survey (urban simulation)
- Software orientation (C#, Terrain, Game Semantics Engine, Shape grammars/models)
- Showing image of a terrain
- Drawing on terrain image
- Making planning
- Writing proposal
- *Deadline*: 2011-11-25

A.2 Prototype II

- Loading an XML-file containing events that have to be triggered
- Display time line
- Trigger the events stated in that document
- Simple semantics (connection with Semantics Engine)
- Updating drawing on changes world
- Linking drawing and semantic data
- Start writing document (Related Work)
- Presentation of seminar
- *Deadline*: 2011-12-16

A.3 Prototype III

- Document: Related Work
- Government instance that regulates use and creation of patches
- Introducing land use development
  - Lot creation
  - Update borders on lot creation
  - Resource production facilities
- Resources
  - Resource gathering
  - Resource consumption
- Two-dimensional representation
- *Deadline*: 2012-02-10

A.4 Prototype IV

- Extracting natural borders from a terrain map
- Loading properties and possibilities from database
- Enhancing allocation algorithm
  - Linking vertices/intersections
  - Triangle patches
  - Single point connected patches
  - Better compensations (measuring width/height)
  - Extrude using average angle over bottom trace
  - Smaller alpha, but adaptive per extension step (relative to current direction)
  - Not growing outside proper ground (fertile terrain for example)
- Extend land use development
  - Residential/Commercial
  - Resource processing plants (industry)
- Land use depending on terrain
  - Proper ground (ground should be fertile for a farm)
  - Elevation
  - Water
  - Distances to other lots
- Mid-term presentation (2012-03-22)
- Assembling committee
- *Deadline*: 2012-04-13

A.5 Prototype V

- Tech tree based improvement of possibilities
- *Deadline*: 2012-04-20
- Event execution
- *Deadline*: 2012-04-27
- Land use maintenance
  - Buildings become older and are less worth when badly maintained
  - Replacement of low value lots
  - Removing patches that are damaged
- Creating and loading pre-set profiles
- Running through TODO-list
- *Deadline*: 2012-05-04

A.6 Release Candidate I

- Tuning semantics
- Document: Introduction
- Document: Design and Implementation
- *Deadline*: 2012-05-25

A.7 Release Candidate II

- Fixes and tuning
- Designing examples for report and presentation
- Document: Results

- Document: Discussion
- *Deadline*: 2012-07-02

A.8 Final Release

- Fixes and tuning
- Correct errors in document and add more images
- Finish document
- *Deadline*: 2012-07-23

Appendix B

MoSCoW

B.1 Must have

- Land use simulation
  - Lot creation
  - Simple buildings (filling the lot)
  - Residential/Industrial/Commercial
  - Resource production facilities (industrial)
- Resources
  - Resource gathering
  - Resource consumption
- Two-dimensional representation

B.2 Should have

- Building construction depending on resources
- Land use depending on terrain
  - Elevation
  - Water
  - Distances to other lots
- Buildings become older and are less worth when badly maintained
- Replacement of low value lots
- Technologies
- Resource, disaster and technology event execution

B.3 Could have

- Zooming the time line (with scroll wheel)
- Road network simulation
  - Extending existing roads
  - Creating new connections
- Three-dimensional representation
  - 3D road display
  - 3D buildings (extruded lots)
- Saving & loading projects

- Going backward in time to test changed parameters
- Multiple governments

B.4 Would have

- Traffic Simulation
- Resource transport
- Event editor
- More buildings (custom made CGA-buildings for example)
- Special land/road elements (bridges, dykes, tunnels, etc.)
- Pop up at hovering an event at the time line
- Terrain simulation

Appendix C

Proposal

C.1 Introduction

Over the years many methods have been proposed for the procedural generation of urban areas, like [14], [10], [12], and [9] to cite a few. The road structures in those generated cities were mostly dictated by strict rules based on real world city patterns. In most methods the road network shapes the city. The placement of buildings depends on the creation of cells in the road pattern. After the creation of cells enclosed by roads, the cells are divided into lots so that buildings can be placed along the roads.

Other methods that are more aimed at semantics, mainly use a different approach. Urban area simulations done using agents [25][26] and applying ecological and urban development models [38] focus on the development of patches. The simulations evolve a city over time by changing the properties of a grid of patches. Each patch can be either road, water, residential, industrial, commercial, or natural (like a patch of forest or grass). The simulation selects patches at random and evaluates if a patch type change would increase the value of the patch taking into account the surrounding patches and economic models of the city. If changing does increase the value, the patch is replaced; if it does not, it is kept the same.

Later work extends this principle for geometric modelling and urban land use simulation which is not tied to a grid of patches. This allows free forms of roads and building lots and therefore more realistic cities. An important example is the land use simulation and traffic simulation that form the basis of the procedural city generator proposed in [9]. Another simulation, which is based on agents and focusses on behavioural modelling and how this is tied to geometric modelling, is shown in [23]. Both result in very attractive city models that can change iteratively based on modelling relations seen in real world urban areas.

Although some of the presented methods do include meaning to the land use simulation and the generation of the road network, many factors are not included. The complete history of a city is important for its growth, especially the first settlement. A city can start as a small village around a bridge because this is a place for trade and transport. This will cause the city to develop a harbour for transport over water and gates to be able to levy toll on carts and keep unwanted people outside the city. Also the resources play an important role in the early development of villages. If the urban area is near mountains, stone is cheap and will therefore be used more often in buildings than villages far from rocky areas where buildings are more likely to be built from wood or clay bricks. These factors influence the success and growth of a city and therefore also the modern development.

In my research I want to develop a urban area simulation that takes into account the meaning behind the steps made in the creation of the city. The initial placement, resources, and events (such as disasters) influence the growth and type of the city. Other factors like economic fluctuations, technological advancement, and neighbouring cities may also contribute to the final shape of the city. These events might be triggered automatically or by the user.

This research poses interesting and exciting challenges. It will extend current techniques by adding more history and meaning to the procedurally generated cities and will therefore create functionally realistic cities. This however will not be limited to real world situations, but could also be used for creating cities in fantasy worlds in games.

C.2 Planning

This section covers the global planning for this project. It has been subdivided into eight stages each with a resulting working program and a part of the documentation. Due to lack of knowledge, this planning will change over time under influence of new insights. Each stage will start with a refinement of the goals in that stage.

C.2.1 Prototype I

- Continuing literature survey (urban simulation)
- Software orientation (C#, Terrain, Game Semantics Engine, Shape grammars/models)
- Showing image of a terrain
- Drawing on terrain image
- Making planning
- Writing proposal
- *Deadline*: 2011-11-25

C.2.2 Prototype II

- Loading an XML-file containing events that have to be triggered
- Applying the events stated in that document
- Simple semantics (connection with Semantics Engine)
- Updating drawing on changes world
- Linking drawing and semantic data
- Start writing document (Related Work)
- *Deadline*: 2011-12-16

C.2.3 Prototype III

- Extending semantics
- Document: Related Work
- *Deadline*: 2011-12-23
- Introducing road development
- Introducing land use development
- *Deadline*: 2012-02-10

C.2.4 Prototype IV

- Extending semantics
- Enhancing road and land use development
- Introducing 3D view (buildings and terrain)
- *Deadline*: 2012-03-09

C.2.5 Prototype V

- Tuning semantics
- Integrating road visualisation
- Document: Design
- *Deadline*: 2012-03-23

C.2.6 Prototype VI

- Creating and loading pre-set profiles
- Tuning semantics
- Tuning visualisation
- Introducing own geometry
- *Deadline*: 2012-04-27

C.2.7 Release Candidate

- Loading and saving results
- Speed improvements
- Tuning visualisation
- Document: Introduction, Results
- *Deadline*: 2012-05-16

C.2.8 Final Release

- Fixes and tuning
- Writing final report
- Assembling committee and picking a date
- *Deadline*: 2012-06-29

Appendix D

Default Database

D.1 Abstract Entities

D.1.1 Government

The properties of the government are described in Table D.1 and D.2.

Parameters	Value	Explanation
Damage weight	0.1	The maximum amount of damage due to shortages.
Goal weight	0.6	The proportion of the old goals that is retained on update.
Costs weight	0.4	The importance of costs in solution selection.
Production weight	0.6	The importance of production in solution selection.
Grow speed	4	The number of patches the government tries to construct.
Allocation tries	20	The number of times it tries to construct a patch.
Replace count	3	The maximum number of patches replaced per time unit.
Replace subset	25	The number of patches randomly selected for replacement.

Table D.1: The properties of the default government, their values and a short explanation.

Resource	Stock	Goal	Degradation
Food	10.0	10.0	0.10
Manpower	10.0	10.0	0.10
Stone	10.0	10.0	0.02
Water	10.0	10.0	0.02
Wood	10.0	10.0	0.02

Table D.2: The resources of the government at the start of the simulation.

D.2 Patch Types

Table D.3 describes the properties of the default patch. The resource production values, costs, and weights are shown in Table D.4, which are all zero.

D.2.1 Natural

The natural patches are described in Table D.5.

Parameters	Value	Explanation
Appearance weight	1.00	A weight for selecting this patch type.
Area threshold	0.00	The lower limit for the area of the patch.
Building type	Automatic	The name of the building type used in SketchaWorld.
Colour	0,0,0	The colour of the patch (RGB values between 0 and 1).
Degradation speed	0.005	The speed for losing its value and gaining damage.
Influence range	6	The multiplier for calculating the range of influence.
Location tries	20	The number of random locations tested to build this patch on.
MaxSegments	40	A polygon segments threshold for cancelling the allocation process.
Precision	0.00005	The precision of the allocation manager.
Shape	Diamond	The icon displayed at the centre of the patch.
Width	20	The envisioned width of the patch.
Height	20	The envisioned height of the patch.
View angle	0.314	The size of the expansion view in radians.

Table D.3: The properties of the default patch (root) and a short explanation of the parameters.

Resource	Weight	Costs	Production
Food	0.00	0.00	0.00
Manpower	0.00	0.00	0.00
Stone	0.00	0.00	0.00
Water	0.00	0.00	0.00
Wood	0.00	0.00	0.00

Table D.4: The weights, costs, and production values for the default patch. It does not produce or cost anything and does not have preferences.

D.2.2 Residential

The residential patches (Table D.6 and D.7) can be built on grass and on mountains.

D.2.3 Industrial

The industrial patch type contains two quarries: clay quarry and stone quarry. Both produce the resource stone. the clay quarry can only be built on grass, preferably near water. Stone quarries can only be positioned on rocky areas like mountain ridges. The properties of the quarries are shown in Table D.8 and D.9.

D.2.4 Water Supplies

Water supplies (Table D.10 and D.11) can be built on grassy and desert areas.

D.2.5 Agricultural

Farms can only be positioned on grass and forest areas. The other properties are described in Table D.12 and D.13.

Parameter	Desert	Forest	Grass	Mountain	Water
Colour	0.65, 0.80, 0.00	0.00, 0.50, 0.00	0.00, 0.80, 0.00	0.50, 0.50, 0.50	0.00, 0.00, 1.00
Wood production	-	0.50	-	-	-
Water production	-	-	-	-	1.00
Shape	-	-	-	-	Wave

Table D.5: The specific parameter values per natural patch type. A dash indicates no change compared to the values of the default patch.

Parameters	Value	Explanation
Building type	RowHouse	The name of the building type used in SketchaWorld.
Colour	1.00, 0.20, 0.40	The colour of the patch (RGB values between 0 and 1).
Influence range	20	The multiplier for calculating the range of influence.
Shape	Triangle	The shape displayed at the centre of the patch.

Table D.6: The changed properties for the residential patches.

Parameter	Hut	Woodcutters hut	Simple house	House	Good house	Mini flat
Building type	Shed	Woodcutters hut	Village House	-	-	MiniFlat
Influence range	30	25	-	-	-	24
Width	8	16	20	20	24	40
Height	8	16	16	16	20	32
Food weight	0.50	0.20	0.40	0.40	0.30	0.25
Manpower weight	0.20	-	0.35	0.40	0.50	0.50
Stone weight	-	-	-	-	-	-
Water weight	0.30	0.10	0.25	0.20	0.20	0.25
Wood weight	-	0.70	-	-	-	-
Food costs	-	-	-	-	-	-
Manpower costs	1.00	2.00	3.00	4.00	5.00	8.00
Stone costs	-	-	-	1.00	2.00	10.00
Water costs	-	-	-	-	-	-
Wood costs	1.00	2.00	2.00	2.00	1.00	3.00
Food production	-1.00	-2.00	-2.00	-2.50	-3.00	-14.0
Manpower production	1.00	2.00	2.00	3.00	4.00	20.0
Stone production	-	-	-	-	-	-
Water production	-1.00	-2.00	-2.00	-2.50	-3.00	-14.0
Wood Production	-	2.00	-	-	-	-

Table D.7: The specific parameter values per patch type. A dash indicates no change compared to the values of the default patch.

Parameters	Value	Explanation
Building Type	Factory	The name of the building type used in SketchaWorld.
Colour	1.00, 0.65, 0.00	The colour of the patch (RGB values between 0 and 1).
Shape	Pentagon	The shape displayed at the centre of the patch.

Table D.8: The changed properties for the industrial patches.

Parameter	Clay Quarry	Stone Quarry
Influence range	14	12
Width	50	50
Height	60	50
Food weight	-	-
Manpower weight	0.50	0.60
Stone weight	0.30	0.30
Water weight	0.20	0.10
Wood weight	-	-
Food costs	-	-
Manpower costs	3.00	3.00
Stone costs	-	-
Water costs	3.00	2.00
Wood costs	2.00	3.00
Food production	-	-
Manpower production	-11.0	-10.0
Stone production	6.00	8.00
Water production	-1.00	-1.00
Wood Production	-	-

Table D.9: The specific parameter values per patch type. A dash indicates no change compared to the values of the default patch.

Parameters	Value	Explanation
Building Type	Factory	The name of the building type used in SketchaWorld.
Colour	1.00, 0.50, 1.00	The colour of the patch (RGB values between 0 and 1).
Shape	X	The shape displayed at the centre of the patch.

Table D.10: The changed properties for the water supplies.

Parameter	Well	Water Pump
Influence range	50	25
Width	6	20
Height	6	20
Food weight	-	-
Manpower weight	0.75	0.85
Stone weight	-	-
Water weight	-0.25	-0.15
Wood weight	-	-
Food costs	-	-
Manpower costs	1.50	4.00
Stone costs	1.00	4.00
Water costs	-	-
Wood costs	1.00	-
Food production	-	-
Manpower production	-1.00	-5.00
Stone production	-	-
Water production	4.00	10.00
Wood Production	-	-

Table D.11: The specific parameter values per patch type. A dash indicates no change compared to the values of the default patch.

Parameters	Value	Explanation
Building type	Farm	The name of the building type used in SketchaWorld.
Colour	0.90, 1.00, 0.00	The colour of the patch (RGB values between 0 and 1).
Influence range	12	The multiplier for calculating the range of influence.
Shape	Square	The shape displayed at the centre of the patch.

Table D.12: The changed properties for the agricultural patch types.

Parameter	Garden	Small Farm	Medium Farm	Big Farm	Giant Farm
Influence range	6	10	-	-	-
Width	8	20	32	50	100
Height	4	20	32	50	100
Food weight	-	0.20	0.25	0.20	0.15
Manpower weight	0.80	0.50	0.10	0.10	0.05
Stone weight	-	-	-	-	-
Water weight	0.20	0.30	0.65	0.70	0.80
Wood weight	-	-	-	-	-
Food costs	-	-	-	-	-
Manpower costs	0.50	1.00	3.00	6.00	10.00
Stone costs	-	-	-	1.00	3.00
Water costs	1.00	1.00	1.00	2.00	3.00
Wood costs	-	1.00	2.00	4.00	6.00
Food production	1.00	2.00	4.00	10.0	25.00
Manpower production	-1.00	-2.00	-4.00	-8.00	-13.00
Stone production	-	-	-	-	-
Water production	-	-1.00	-3.00	-6.00	-10.00
Wood Production	-	-	-	-	-0.50

Table D.13: The specific parameter values per patch type. A dash indicates no change compared to the values of the default patch.

Appendix E

Small Database

The small database contains just three artificial patch types and two resources. The government instance has the same properties as the one used in the default database (see Appendix D). Also the default patch type and the natural patch types have not been changed, except for the river patch type. The database is designed to cluster factories together and houses together. Commercial patches (stores) are attracted by both, but mostly by houses.

E.1 Abstract Entities

Resource	Stock	Goal	Degradation
Urbanity	10.0	10.0	0.10
Pollution	10.0	10.0	0.10

Table E.1: The resources of the government at the start of the simulation.

E.2 Patch Types

Houses can be built on grass and mountainous areas. Factories can be built on forests, grass and desert patches. Stores can be built on both grass and desert areas. All patch types are available from the start of the simulation. They are described in Table E.2.

Parameter	House	Factory	Store
Building type	RowHouse	Factory	Automatic
Colour	1.00, 0.20, 0.40	1.00, 0.65, 0.00	0.00, 0.00, 0.00
Shape	Triangle	Diamond	Diamond
Influence range	4	4	4
Width	40	50	40
Height	32	40	40
Urbanity weight	0.80	-0.20	0.80
Pollution weight	-0.20	0.80	0.20
Urbanity costs	-	1.00	1.00
Pollution costs	1.00	-	1.00
Urbanity production	2.00	-0.50	-3.00
Pollution production	-0.50	2.00	-1.00

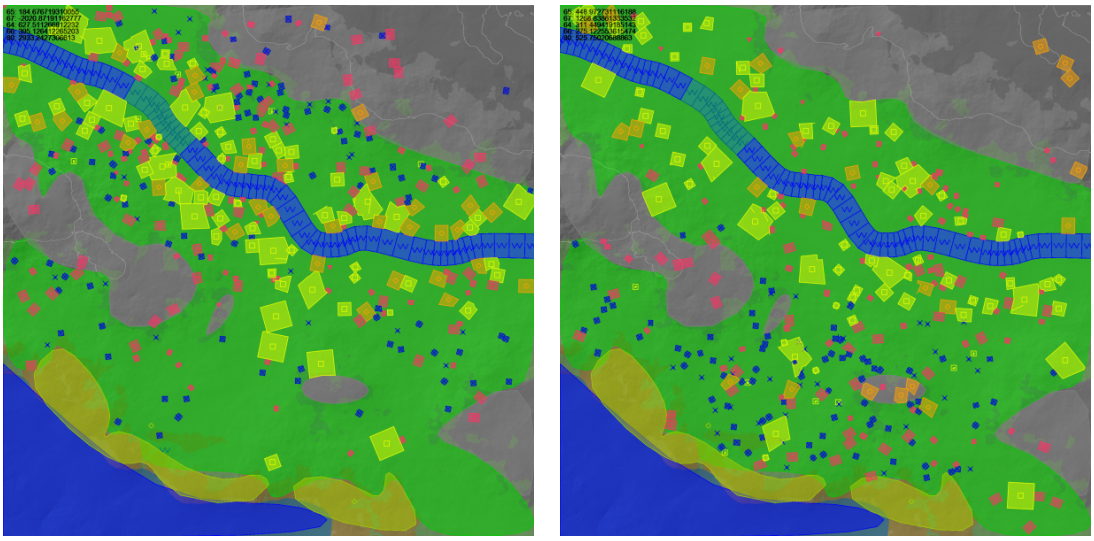
Table E.2: The specific parameter values per patch type. A dash indicates no change compared to the values of the default patch.

Appendix F

Default Database Examples

F.1 Visuals

The default database can be used to generate a great variation in urban areas. The output of the program depends on the random numbers from the pseudo-random number generator. The project defines a seed used for the generator. To see how the simulator performs with the default database, it is ran for 200 cycles with different seeds. Figure F.1 shows that great differences can exist between results.



(a) (b)

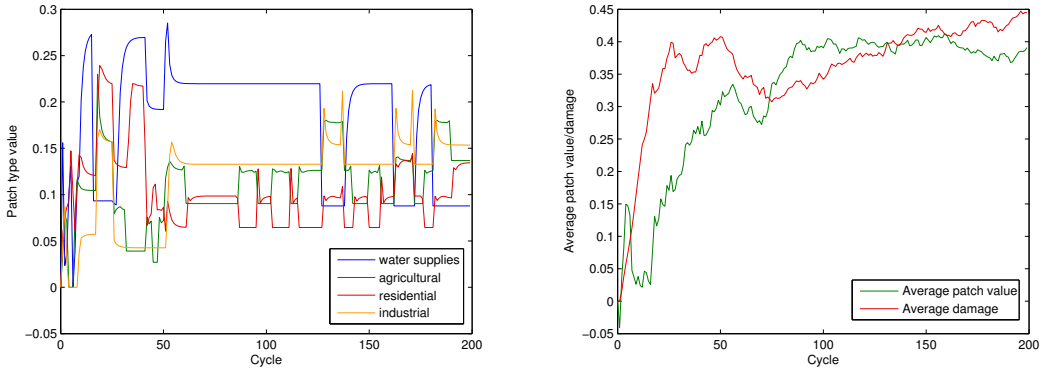
Figure F.1: Two examples of urban areas that developed for 200 cycles.

On the left, a clustering near the river is shown with a centre at the top left of the terrain. The yellow patches are farms, the red patches represent houses, the blue squares are water supplies and the industrial patches are orange. The farms and quarries seem to be closer to the river than the water supplies and houses. This is a result of the water supplies being hydrophobic which also causes them to scatter more, rather than clustering together. The smaller houses are also scattered, especially the bigger patches. This is caused by a low value of attraction by water for bigger residential patches. The black text at the top left shows the current stock values and indicate a tremendous shortage of water and a huge surplus of manpower. In this scenario the clustering is going well but the surpluses and shortages are only getting bigger, indicating that the database is not very balanced yet.

The figure on the right, Figure F.1(b), shows a different pattern, resulting from a slightly different database and a different seed as the previous example. The parameters are explained in Appendix D. The water supplies are only scattered under the river causing a surplus of water on below the river, while water consuming patches far from the river have a shortage of water. This placement of water supplies can be explained by the repelling force that acts on the water supplying patches. Therefore water supplies are preferably not build near each other and not near the river. Since the water supplies cannot be built in the mountain areas, only the area under the river is a possible location for water supplies. Most water consuming patches grow in between the water supplies and along the river, but less clustered. Small clusters do exist near the river. Again a few bigger residential lots are placed on the mountain ridge on the left. The stock values are still big but seem to fluctuate rather than grow endlessly.

F.2 Statistics

During the simulation the government can only control the resources by building new patches, replacing patches and removing patches. These procedures are guided by patch type values (Figure F.2(a)) and patch values (Figure F.2(b)). At the start of the simulation the patch type values rapidly change since there is not much difference between the stock values (see Figure F.3(a)). Eventually a shortage in manpower pushes the residential value to rise. Also the water supplies are promoted due to a small shortage of water, and a surplus of both wood and stone. This short period of 15 cycles is followed by a period with a surplus of water causing the value of water supplies to drop. The value restores when there is a shortage again. After 50 cycles there is no shortage of manpower any more because of the promotion of housing. During the remaining part of the simulation the value of houses is therefore only bound by the surplus and shortage of food. Since the food stock value fluctuates, the value of houses also fluctuates. The short transitions are caused by a rapid change in both stock and goal value on which the patch type value is based. The same counts for the producer of food and consumer of manpower: agriculture. The value of quarries is for long periods constant and only rises when there is enough manpower (after 50 cycles) and when enough water is available (130-140 cycles, 160-170 cycles, and after 180 cycles). Compared to the other patch types, water supplies are by far the most wanted patch types due to long periods of water shortage.



(a) The values of patch types over time.

(b) The average patch values and damage levels over time.

Figure F.2: The development of patch type values, patch values and damage levels over time.

The graph in Figure F.2(b) shows the development of the average patch values and damage levels over time. The first generation of patches starts without any damage but the damage level rapidly increases do to lack of clustering and therefore a lack of resources in their neighbourhood. As the number of patches and the clustering increases, the average patch value increases, although

the damage level increases during that same period. The first big dip in the damage value is caused by the removal of badly placed water supplies and quarries. Just after 50 cycles, when the manpower stock is positive, both the damage levels and the patch values decrease. The decrease in average patch value can be explained by changing goal values, since the goal values are also used to calculate the value of patches. The average patch value however recovers quickly from it, probably because of the placement of many new patches, and grows to a higher and stable value around 0.4. This while the average damage level slowly increases over time.

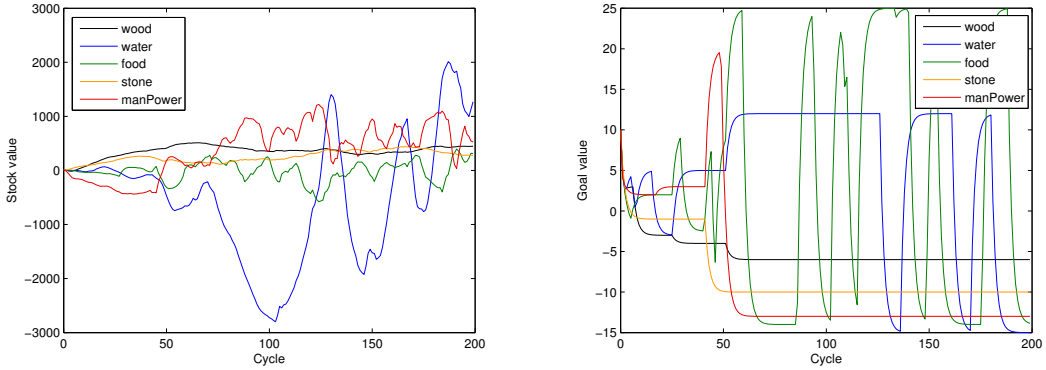
The changes seen in the previously described graphs are based on the goals of the government which are in turn influenced by the stock values. Figure F.3 shows how the changes in stock values cause the goals to alternate. The goals are limited which causes them to change when a new patch types becomes available. For example, the wood stock first drops to -2.5, but after 24 cycles it drops further to -4. After 50 cycles it drops to -6. This is all because new buildings became available which cost 2.5, 4, and 6 units of wood. The smoothing factor applied to the goals causes the goals to converge to the new value in a few cycles rather than switching to it directly. This smoothing directly affects the patch type values as can be seen in Figure F.2(a), which in turn smooths the decision making over time.

At the start of the simulation, all resources start at a value of ten as well as the goals (see Figure F.3(b)). Since the goals aim at an equilibrium of resource production and consumption, the goals rapidly drop below zero to pull the stock values down to zero. A few cycles later, after cycle number 5, some goal values however increase due to a shortage of food, water and manpower. Especially the latter resource causes problems. Due to a lack of houses being build, a shortage in manpower is created. Also due to the low food and water stock values, two resources consumed by households, the residential patch type value stays low for some time. The removal of quarries and creation of new houses (around cycle 40) helps to turn the shortage of manpower in a surplus, although this causes a shortage of food and water since all new people have to be fed. As a result, the food goal rises and the manpower goal drops to a negative value and stays there since from then on the manpower stock does plunge to a negative value any more.

The water stock has an even bigger effect since it is related to most patch types. It also shows a shortage up to the point when a new water resource (water pump) is introduced (cycle 50). The patch type value peaks and the shortage decreases while new water pumps are installed. However due to the increasing number of houses and farms, the water shortage grows after cycle 70. The goal stays high until the water stock turns into surplus, near cycle 130, which heavily affects the patch type values. At that point the goal drops to -15 and recovers as soon as there is not enough water produced around the near patches. This is repeated a few times (at cycle 165 and at cycle 180). The last time the goal does not recover to a positive value. The food goal and stock also alternate during the simulation, but not as severe as the value for water does. The stone and wood stock seem to stabilise around a surplus of 500 which helps the simulation to keep on building new patches for which these resources are needed.

Figure F.4(a) shows how the number of patches per patch type category evolves over time. At the start of the simulation a shortage of manpower causes the number of patches that require manpower to stay low, allowing residential patches to be placed. The number of water supplies slightly drops during a period of surplus on water. Though as soon as the number of residential patches increases even more, the water and food stock drop and the manpower stock increases, triggering the creation of more water supplies and farms. This continues for a long period. A small dip can be seen at cycle number 70 which is caused by a destruction event. Since not many patches were located at the north-west of the terrain where the destruction occurred, not many patches were damaged by the destruction event at cycle number 70, though still a few houses were removed. This moment can also be seen in Figure F.4(b) and F.4(c) as a spike around cycle number 70. Afterwards, the water shortage increases rapidly. Once the water shortage is resolved (around cycle 130), the pace at which the number of water supplies grow slows down. It even drops a few time during periods of surplus, which is also visible in Figure F.2(b). After 150 cycles there is a big dip in the number of patches. Many patches are removed during that period. Mainly houses are removed due to a lack of resources near the patches.

The removals of patches are more apparent in the second graph, Figure F.4(b). The highest

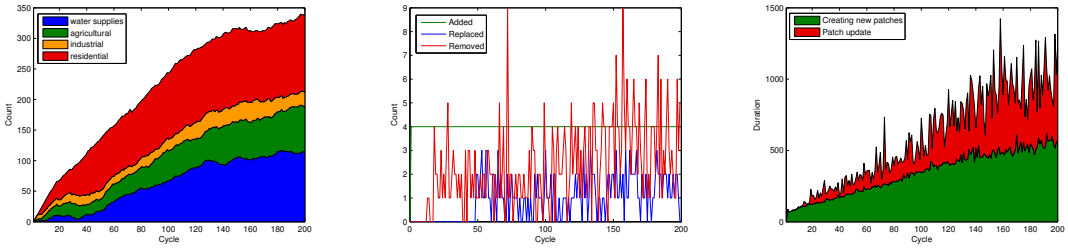


(a) The changes of stock over time showing shortages and surpluses.

(b) The goals over time.

Figure F.3: The calculation of the patch type values is based on the goals (b) which in turn are based on the stock (a).

spikes can be found at cycle 70 and cycle 160. As noted before, the former is caused by a destruction event and the latter by plain removal of bad patches, albeit in a higher rate than normal. The number of added patches is constant and equal to the number of patches that the government should produce per time unit according to the database. It can occasionally occur that no suitable allocation can be found in which case the number of added patches for that cycle drops. The number of replacements differs more over time. Just after cycle 50 it is high during a shortage of both water and food, creating new wells to help restore the stock values. It also replaces the water supplies around cycle 140 to create small farms.



(a) The number of patches of each patch type category.

(b) The number of mutations done each cycle.

(c) The duration of each cycle.

Figure F.4: The patch type counts, the number of mutations and the duration of the updates per cycle.

Appendix G

Class Diagram

UrbSim is constructed out of four major parts: one for loading the project settings (see Figure G.1), a simulation part that contains the government and its patches (shown in Figure G.2), the allocation manager and all of its required elements (Figure G.3) for allocating new patches, and finally the graphical user interface to display the results (see Figure G.4). The presented class diagram does not contain all classes present in the code. It lacks the class for data recording (used for exporting results) and a few redundant classes used for testing purposes. These classes are not necessary for the simulation to work.

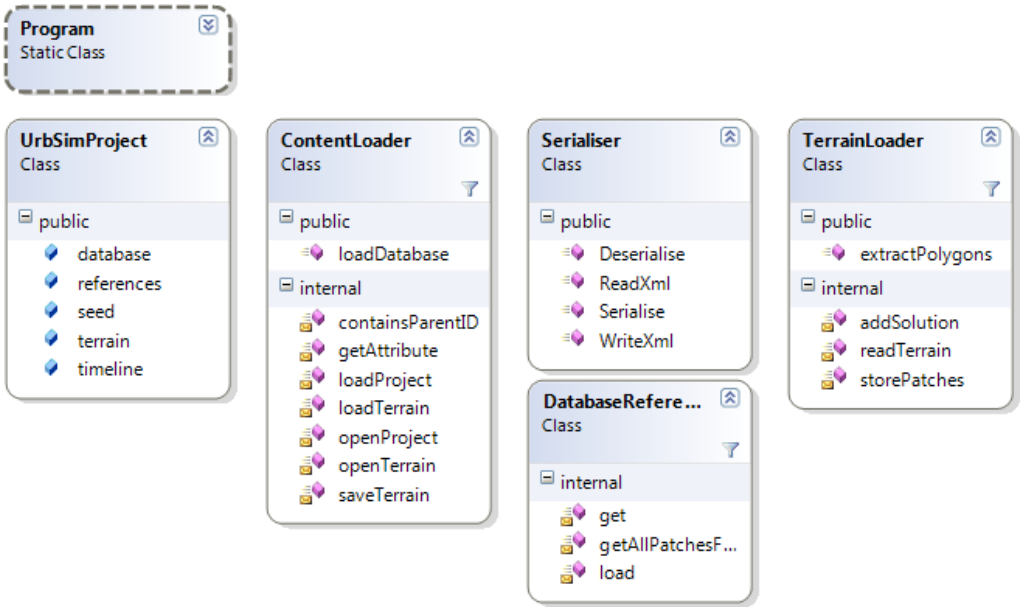


Figure G.1: Loading the database, the terrain and other project settings is done by the ContentLoader.

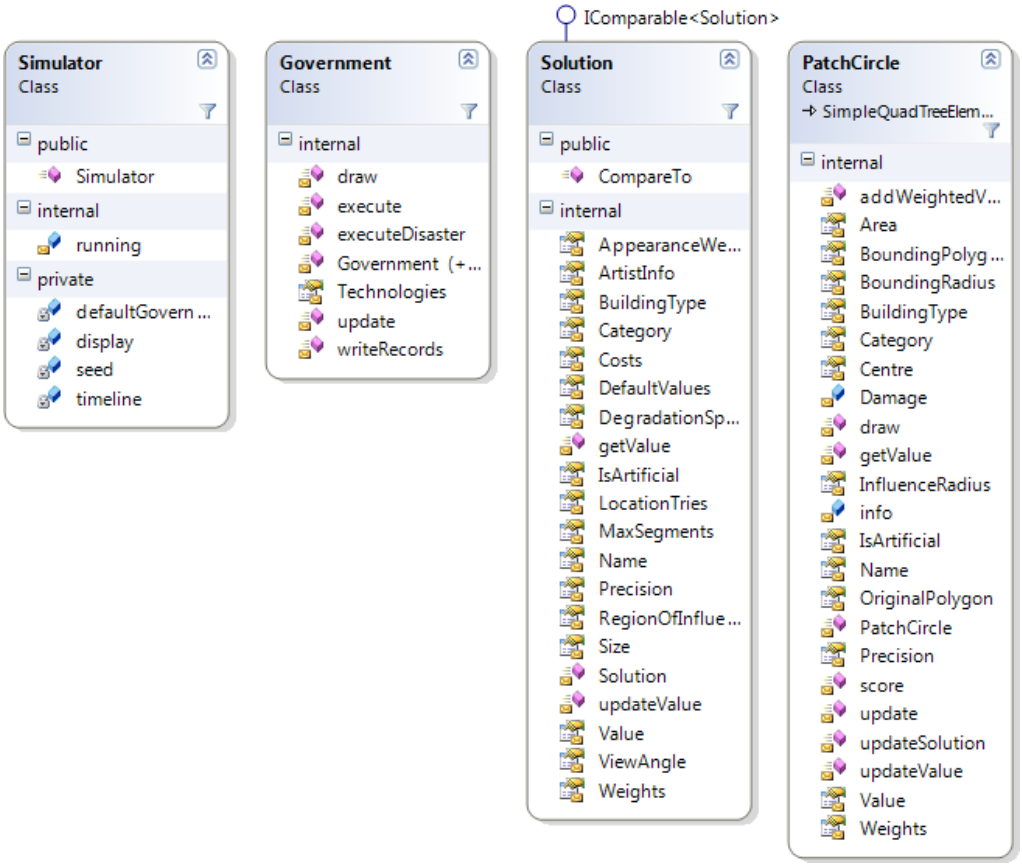


Figure G.2: The Simulator calls the Government and Display each cycle for an update. The Government relies on its stock (private), its patch type (Solutions) and patches (PatchCircles).

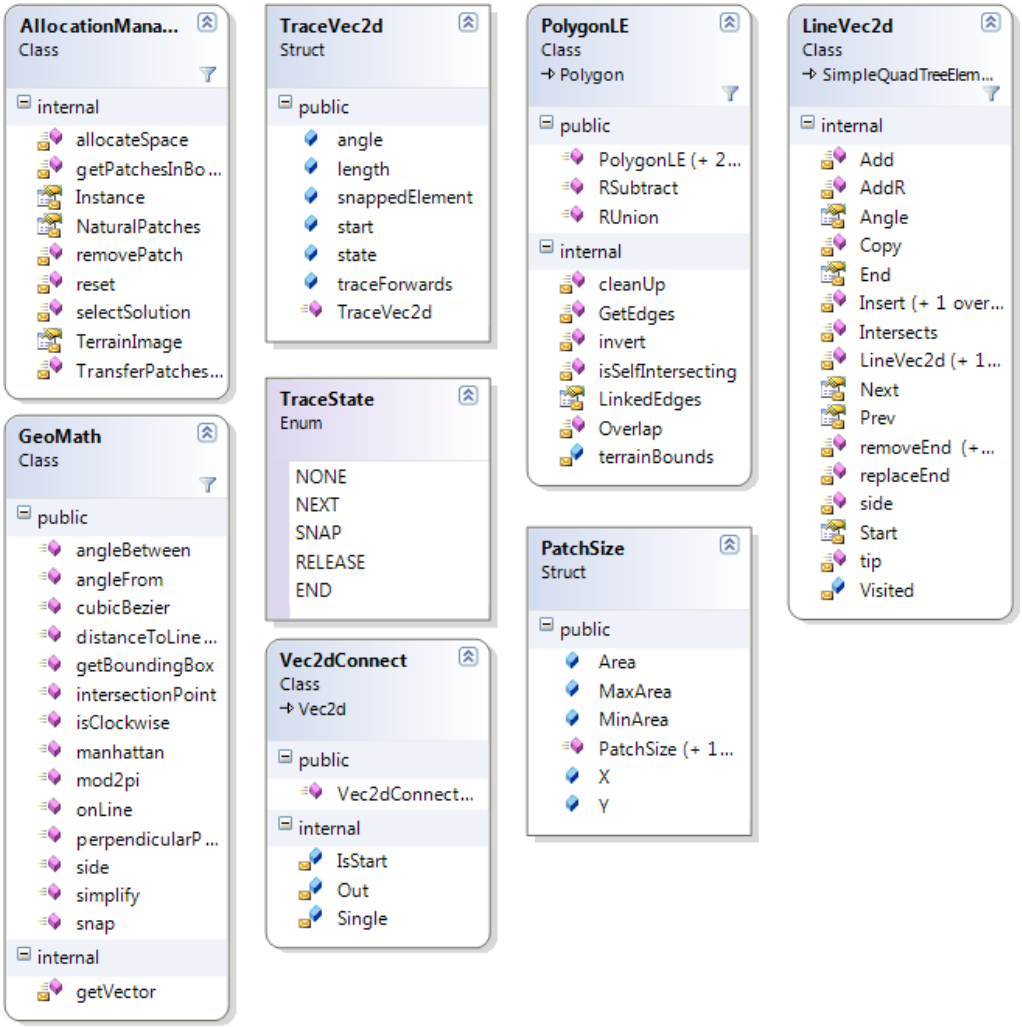


Figure G.3: The allocation of a new patch is done by the AllocationManager, which grows traces (TraceVec2d) to form new polygons (PolygonLE) with linked edges (LineVec2d). Basic mathematical functions are carried out by the GeoMath class.

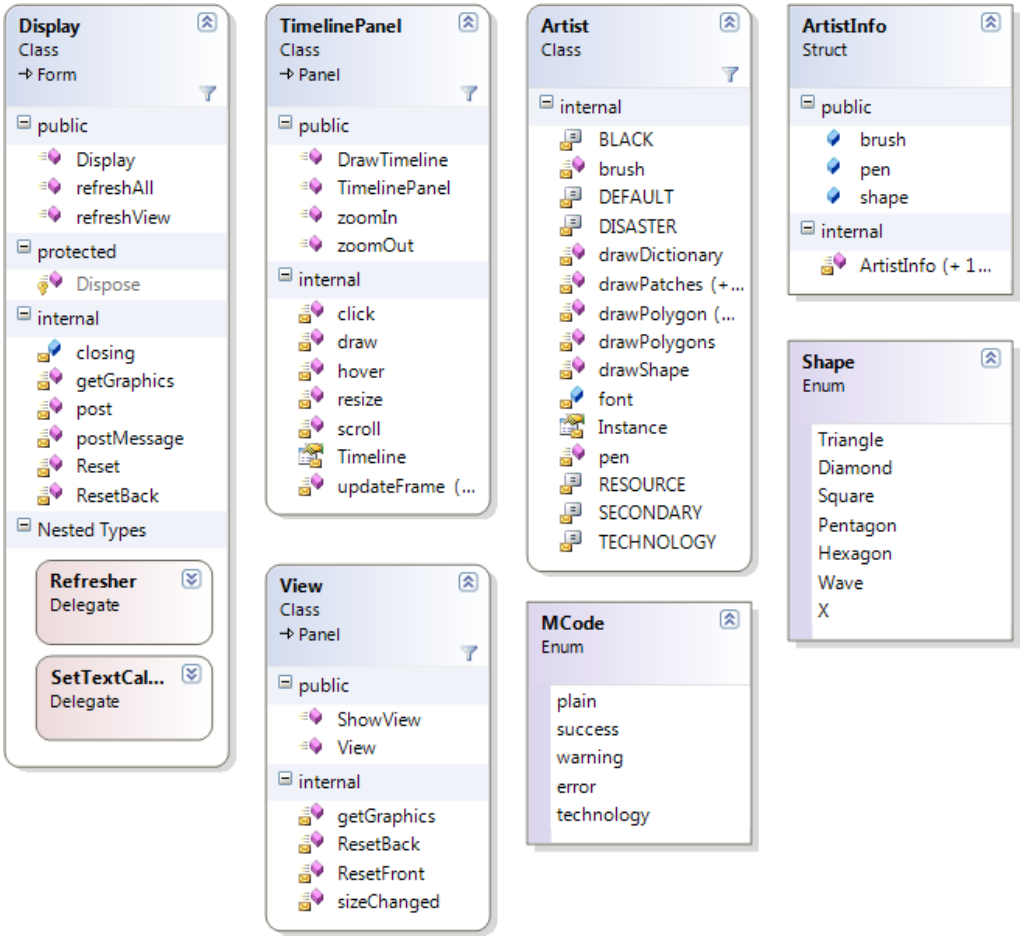


Figure G.4: To give the user feedback on the state of the simulation, the Display shows the Timeline through the TimelinePanel and draws the patches on the View. The Artist is responsible for linking semantic information and how features are drawn on the View.