Real-time generation and rendering of realistic landscapes

Hans Häggström

Helsinki August 15, 2006

UNIVERSITY OF HELSINKI
Department of Computer Science

Real-time generation and rendering of realistic landscapes

Hans Häggström

Master Thesis
Department of Computer Science
University of Helsinki
August 15, 2006, 87 pages

With the increasing size of virtual landscapes in games and other applications there is a growing need for generation algorithms that can help designers to produce large realistic landscapes. Parametrized procedural and fractal systems provide this, and also enable on-the fly data generation that minimizes required storage space. Ecotopes provide a way to introduce natural variation in automatically generated landscapes by varying the generation parameters based on the location. Interactive performance can be achieved by using geometry with a level of detail that decreases with the distance to the observer. In this thesis I evaluate different algorithms for generating and rendering terrain, vegetation, buildings, cities, and the sky. New algorithms are sketched out for terrain generation through successive uneven mass deposit, elevation map modifying textures, river system generation, pattern based city generation, and weather modeling. A subdivision based house generation algorithm is also presented and partially implemented. Finally opportunities for further research in conveying emotions with landscapes are identified.

ACM Computing Classification System (CCS): I.3.7 [Virtual reality], J.2 [Earth and atmospheric sciences], J.3 [Biology and genetics], J.5 [Architecture]

Keywords: landscape, procedural, realistic, real-time

E-mail: Hans.Haggstrom@iki.fi

Homepage: http://www.iki.fi/hans.haggstrom/

Contents

Figure 1: Adrianus van der Koogh, View of Beek from the Ravenberg, 1820.

1 Introduction

Through the ages people have been fascinated by landscapes. Nineteenth century painters perfected landscape painting to an art form that captures the minute details in a scene, but also confers feeling and atmosphere *(see Figure 1)*. The modern electronic landscape artists try to produce similarly realistic and atmospheric landscapes in a three dimensional form on computers. Although detailed three dimensional landscapes can be created manually, it is a costly and slow process. For this reason a number of algorithms have been developed for automatically generating different aspects of landscapes.

A landscape can be divided into several different kinds of features. The basis is a terrain with a varying topology and surface structure. On the terrain we may have plants arranged into forests and fields, individual buildings as well as cities, roads and infrastructure, rivers, lakes and seas, and animals and people.

As a background to it all, there is a sky that may contain clouds, the sun, moon, and stars.

To create a realistic landscape it is not enough that the elements making up the landscape look realistic by themselves, they should also be placed naturally in relation to each other, mimicking the way real landscapes are structured. A consistent landscape with variation both at local and global scales is more interesting to view and explore than a completely random or homogeneous landscape. Ecotopes provide a way to model this.

Real time generation of a landscape can be achieved by storing it compactly as generation rules, parameters, and random seeds, and using various fractal and procedural algorithms to expand these inputs into the visible part of the landscape. Real time rendering requires good management of level of detail; the nearby landscape features around the observer are accurately rendered, but more distant features are approximated to a sufficient detail level.

In this thesis I first list some applications of landscape generation, and present general principles of landscape rendering. Then I address the different landscape features – terrain, plants, buildings, cities, and the sky – examining previous work, and introducing some new ideas. Finally I conclude by listing some of the work that remains to be done on the suggested new algorithms, and list some possible future research directions.

1.1 Applications

There are numerous fields that need to model landscapes in some way. The application areas include flight simulators, computer games, visualization in architecture and land use planning tools, geographical visualization, landscape design, background generation for movies, and rendering of natural scenes in art. Other interesting applications include providing realistic environments for use in artificial life, robotic simulation, and machine vision research [TR97].

The applications can be roughly divided by requirements on interactivity and source data. Computer games are an example of an application that require dy-

namical and interactive landscapes, but can use artificially generated data, while for example geographical information systems manage with more static landscapes, but need accurate real-world data as a basis for the visualization.

1.2 Procedural vs. declarative

For most applications, there needs to be some way that landscape designers can specify the appearance of the landscape. There are two extreme ways to approach this problem – declarative and procedural – and a broad middle ground between them.

Using the declarative approach, we could define every height point in the terrain, and the position and properties of every object in the landscape. This has the advantage that landscape designers have absolute control over the landscape. The disadvantages are firstly the amount of storage space needed, and secondly the amount of work that landscape designers have to do to define the landscape. This is the approach usually used for real world geographical systems, where conformance to the real world geography is important, and the terrain height and object placement information is often possible to obtain from measurement data. In this case we have huge amounts of data, but no landscape generation algorithms are needed.

The procedural approach is to generate the landscape using algorithms that produce varying, natural looking data. The landscape designer is basically encoding her landscape generation knowledge into code. This way the amount of landscape data we can generate from a single random seed is unlimited once we have the algorithms written. This approach can be used if the landscape does not have to match any real life location. We have almost no input data (just a random seed value), but extensive, specialized algorithms for generating a vast landscape randomly from the seed value.

The procedural approach has a small memory footprint and disk storage size, and also saves the landscape designer a lot of tedious detail work. The declarative approach on the other hand, allows for more exact definition of the landscape. The strengths of both approaches can be combined, by using the procedural approach

by default, but allowing exact definitions in places where they are needed, and allowing parametrization of the procedural functions, so that the landscape designer can more accurately specify what kind of landscape is generated in different areas. This gives more control and creative freedom to the landscape designers, while still providing algorithms for taking care of the tedious and repetitive aspects of landscape generation. I focus mainly on this approach in this thesis. The challenge with this approach is developing fast landscape generation algorithms that both produce good results and are easy to parametrize by landscape designers.

A drawback with a procedural approach, where the landscape is generated on demand, is that some landscape generation processes are by nature chaotic, iterative and non-local, that is, small changes in the landscape far away can have large impacts on the landscape nearby that are time consuming to calculate. This makes it very hard to simulate them accurately on the fly for a small target area around the observer. The best example of this is hydraulic erosion – streams and rivers have a large impact on many landscapes, but their exact route and size depends on where the water is coming from and how the river has meandered over time. A potential solution is to approximate the effects of these process in some ad-hoc way.

1.3 Level of detail

When observing a landscape from a point on the ground surface, any objects and ground close to the observer are seen as much larger on the screen than objects and ground further away from the observer. The amount of detail (texture size and amount of geometry) required to represent the close-by ground and objects is thus higher than for far away areas. This allows us to do an important optimization for the landscape rendering: objects and ground far away from the viewer can be simplified, thus requiring less geometry and less time to render than they would if they were rendered at full resolution. On the other hand, we need to change the detail level of objects as the observer or the objects move, and the distance to the observer changes. This presents us with another challenge though; switching between detail levels for objects and the ground easily causes disturbing visible artifacts as the object or ground suddenly changes shape slightly.

With properly implemented level of detail handling and landscape generation algorithms that provide data as needed for a given area and detail level, it is possible to move all the way from an orbit view of a planet to a closeup of small pebbles on the ground with interactive frame rates [MME98].

1.4 Evaluation criteria for algorithms

Each algorithm used to produce some kind of landscape feature can be evaluated on several different metrics; performance, memory usage, disk usage, bandwidth usage, ease of defining the content, ability to cope with changes to the content, and the (visual) quality of the generated landscape feature.

Algorithms can be run either in a pre-calculation phase where just the results of the algorithm need to be loaded each time a landscape is visualized, in a set up phase just before the landscape is visualized, or on a demand basis when the camera moves in the landscape. These three cases have different performance requirements. Pre-calculation steps have no critical time limits, as they are done only once before the application is released. Algorithm steps done at application startup can together use up to at most half a minutes or so of time, depending on the type of application, before the user starts to perceive the application as too slow. Startup initializations might in some cases be run in a background thread while the user does something else, if the landscape visualization is not needed immediately when the application opens. On demand algorithms are usually level of detail centered algorithms, where some more detailed features of the landscape are calculated as the camera gets closer. They should be fast enough that they do not slow down the frame rate, and it is best if they are performed in smaller steps over several frames, rather in big chunks at a single time, as that can result in occasional slow rendering frames when the camera moves.

The memory usage of an algorithm needs to be reasonable. Algorithms and data access from disk can often be sped up by caching the results in memory, but especially the amount of needed texture memory on the graphics card also has to be taken into account, as texture memory is still a limiting factor for many algorithms, and is also relatively slow to transfer data to and from.

Bandwidth usage needs to be considered in the case of a client-server application, where the server holds a landscape description and the client visualizes it. Here procedural algorithms are useful, as they are data expanding – they only need a relatively small input to generate complex landscape features.

It should be easy for landscape designers to define the landscape. An algorithm that uses high level, easy to understand concepts is faster to work with than an algorithm that requires input in some complex low level format, where the effects of the input on the generated result is hard to predict for the designer. In some cases a good editor application can hide some of this complexity, but implementing the editor is easier if the algorithm already operates with concepts the users understand.

Some applications require ability to change the landscape on the fly. For example, a landscape architecture visualization tool used for land use planning needs to support easy modification of the landscape to visualize different scenarios, or a game might allow the landscape to change as a result of player actions. In these cases the changes to the landscape usually need to be stored as exceptions to the original generated landscape, which can be somewhat storage intensive if the changes are extensive and complex. Just changing the input parameters of the algorithms would require less storage space, but it is usually hard or impossible to come up with a set of new inputs that generates the original landscape plus the modifications by the users.

Finally, algorithms should produce realistic and visually pleasant results that help in user immersion. However, as added realism usually introduces an added cost in terms of rendering time or storage requirements, the visual realism of the landscape needs to be balanced with performance aspects. Also, we can use a lower degree of realism where the human eye and brain do not notice it. For example, in the case of complex foliage in trees, humans will usually not notice if the perspective is not entirely correct [SK04]. So exact realism is not required, just the appearance of realism. Maintaining interactive frame rates is usually more important than achieving a high degree of realism. In some applications a more stylish, simpler visual appearance can be used, which might be faster to render.

1.5 Ecotopes

Different areas in an extensive natural landscape may have very different appearances. For example, one could differentiate between rocky ground, forests, steppes, marshlands, lakes, and so on. Ecotopes are a way to implement this kind of variation. An ecotope contains landscape parameters that affect the landscape in the areas the ecotope is applied to, and distribution parameters describing the places where the ecotope should be applied. Several ecotopes may be applied to the same location, weighted by how well their distribution properties match the landscape at that point. This kind of ecotope system has been used for example in the Terragen landscape renderer [Sof06] and described in [Ham01].

Landscape parameters of an ecotope could include height functions or filters that can modify the current terrain height by some amount, a number of different plant species and their densities, as well as information on rain amounts which affect the number of rivers and lakes generated. It could also contain information on population density and parameters controlling other aspects of city and building generation. The ecotope also specifies the ground texture to be used on the terrain.

Distribution properties of ecotopes could include terrain elevation, relative elevation, slope angle, proximity to sea, proximity to a river or lake, and latitude. A randomly generated noise function could also affect the distribution. The importance of each distribution property could be weighted. Landscape designers could also have more direct control of ecotope placement in addition to the distribution properties. For example, bitmaps with ecotope distributions could be imported, or areas defined and some ecotope applied to that area (with a soft transition at the border of the area).

An ecotope could also contain subtypes that are applied on areas within the parent ecotope that match their distribution properties (after any modifications specified by the parent ecotope have been applied to the landscape). An ecotope always operates at some level of detail. This affects the sampling sizes used when calculating the distribution properties of the landscape, and the smallest area that the ecotope can be applied to. Subtypes must have a higher or equal level of detail as their parent ecotope. This way we can apply sub-ecotopes to the visible

landscape only when it is close enough to the viewer that the resolution of the sub-ecotope is visible.

Ecotopes provide a flexible framework that can be used to implement both macro scale features such as climate zones, as well as medium scale features like forests along rivers in dry landscapes, and micro scale features such as tall grass growing next to stones and trees. They give landscape designers both controls over the characteristics of different types of landscape, as well as control over where to apply what type of landscape.

2 Terrain

The basis for a landscape is the shape of the ground. It is what the rest of the landscape content, such as trees and buildings, rest on. It was also what the first computer generated landscapes focused on, such as the famous mountain ranges and planets created by Mandelbrot [Man82]. In this chapter we will explore different types of algorithms for generating natural looking terrains.

To enable efficient modeling and rendering of the ground, some simplifications are usually made. Firstly, it is assumed that the ground surface has no overhangs, that is, any ray from the center of the planet intersects the planet surface exactly once. With this simplification, the ground shape can be defined by a ground height function that maps a position on the globe to the ground height at that position. A further simplification that can be made when modeling and rendering only a small part of the planet surface is to treat the surface as basically planar, instead of spherical. In this case the ground height function is a function from x and z coordinates to the y coordinate of the ground surface, assuming y is used as the vertical axis.

For rendering a ground surface, it is often practical to only store samples of the ground height function at some intervals along the x and z axes. A set of such samples for some area is called an elevation map or height map. If the height is encoded as colors, it can be stored as an image (this is a common way to store terrain data in applications).

When comparing the algorithms, I have focused on the following aspects: the speed of the algorithm, the naturalness of the generated terrain, and the ability to generate portions of the terrain efficiently. The last aspect is important when generating a landscape for interactive viewing, usually only the area around the viewer needs to be generated. If the algorithm needs to generate the whole terrain even when just a small part of it is needed it will probably be inefficient for the purpose of interactive landscape visualization.

2.1 Random terrain generators

The simplest way to generate random terrain seems to be to assign each position on the ground a random height. The result of that bears little resemblance to natural terrain though. In nature, the ground height at a given location is on average more similar to nearby locations than some arbitrary locations – the natural ground is more or less continuous, while still varying in height in complex ways depending on the position. Creating this kind of random, yet continuous ground is the challenge of random terrain generators.

Terrain generation algorithms often approximate a fractal noise, which also has more general applications within landscape generation; such as texture synthesis and creating random distributions for ecotopes.

2.1.1 Stochastic subdivision algorithms

Synthetic landscape generation has its roots in the fractal mountain ranges first produced by Mandelbrot [Man82]. The technique used was iterative subdivision with pseudo-random midpoint displacement [Car80] [Mil86]. This is a simple and efficient method that has been widely used for random terrain generation. In its basic form the algorithm is as follows:

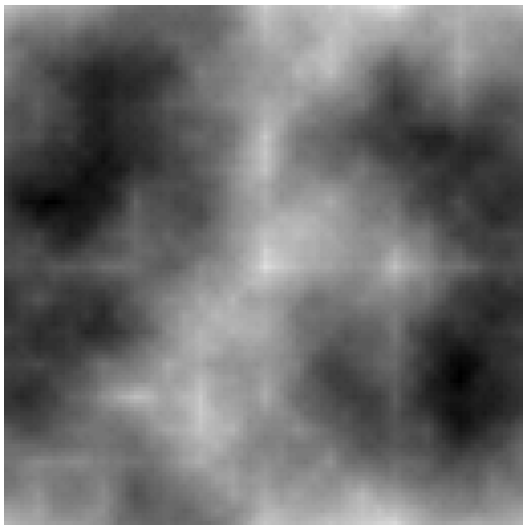

Figure 2: A height map created with stochastic subdivision [Bur06]

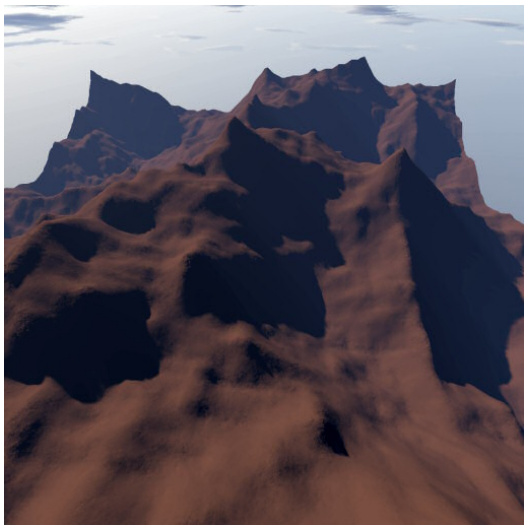

Figure 3: A 3D rendering of the height map in Figure 2 (created with Terragen [Sof06]).

1. The terrain starts with a single large square, with a height value of zero at each corner.
2. A pseudo-random height offset that is proportional to the size of the square, is added to each corner of the square
3. The square is divided into four smaller ones, with the height of each new corner interpolated between the heights of neighboring corners of the original square.
4. The algorithm is repeated from step 2 for each smaller square, until the squares are at the desired level of detail.

The result is a fractal surface that bears some visual resemblance to a natural mountainous landscape, and that can be regenerated again by using the same seed value for the pseudo-random number generator *(see Figures 2 and 3)*.

A drawback of this simple approach are that it often produces unnatural looking regularities (in the form of sharp ridges or peaks) along the edges of the squares [Lew87]. Another weakness is that the random variation varies linearly with the scale of the features. In natural terrain the amplitude of height variation does not depend linearly on the scale of the features.

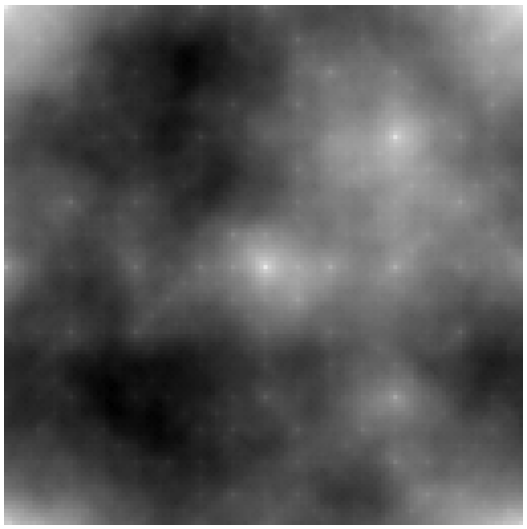

Figure 4: Diamond-Square Subdivision [Bur06]

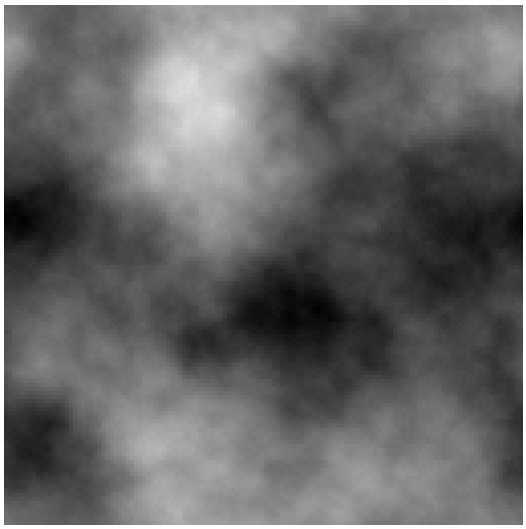

Figure 5: Offset Square Subdivision [Bur06]

To overcome these problems there have been numerous improvements and modifications developed for the basic iterative random subdivision algorithm.

In the diamond square subdivision technique the algorithm divides a square into four smaller squares rotated 45 degrees in relation to the original square, so that they partly overlap the neighbors of the original square [FFC82]. This approach eliminates some of the more visible artifacts, but has some quite noticeable point-like artifacts of its own too *(see Figure 4)*.

The offset square subdivision method avoids most of the artifacts, but with somewhat increased performance cost *(see Figure 5)*. When a square is subdivided the smaller squares are offset from the larger square corners, and the initial values for the smaller square corners are calculated with a weighted average, resulting in a more smooth terrain [PS88].

The random variation can be adjusted by other factors, such as the height. For example, Mandelbrot raised the height values in some of his landscapes to a power greater than one, thus producing lower height variations for low-altitude terrain and larger variations for high altitude terrain, roughly simulating the effect of erosion, glaciation, and similar processes that tend to gather material irregularly from high altitudes and spread it out to smoothly cover low-altitude terrain.

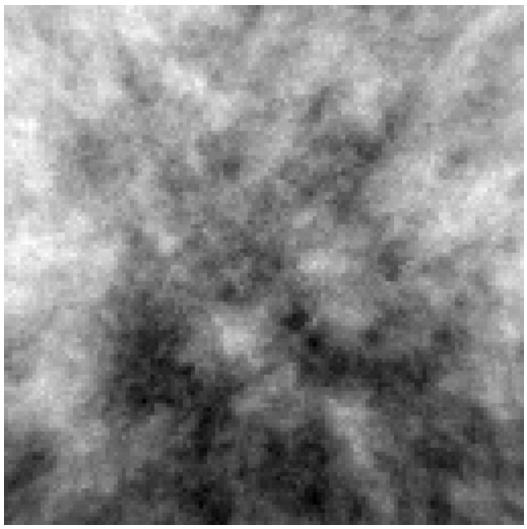

Figure 6: Iterative Faulting [Bur06]

It is also possible to vary the amplitude of the random offset at different scales. This enables an approach where we sample the height variation of some natural terrain at different scales, and then use that information to produce statistically similar synthetic terrain.

2.1.2 Faulting algorithms

Faulting algorithms generate fractal data by repeatedly dividing the terrain with a faulting edge, raising the terrain on one side of the edge and lowering it at the other to achieve a height difference along the faulting edge. Over time the height difference is reduced, and when it arrives at zero the terrain is ready *(see Figure 6)*.

There are several variants of this algorithm. One of the earliest variants works on a sphere, using a randomly directed plane passing through the center of the sphere as the faulting edge [PS88]. It produces a non-distorted fractal terrain on a sphere, but has the disadvantage of mirroring a point at one side of the sphere on the other, with inverted height. This can be avoided by removing the requirement of the faulting planes having to cross the center of the sphere, allowing any kinds of intersections between the plane and the sphere.

Although the produced terrain does not suffer of similar artifacts as subdivision

terrains do, it has the drawback of being very slow. A way to speed it up could be to use a softer faulting edge, spreading out the height change across the faulting edge over a broader area. Despite this improvement it still falls short of subdivision algorithms in speed, as it has to touch each point on the map on average $NumberOfIterations/2$ times (each fault modifies half of the landscape).

Also, calculating just a small area of the terrain still requires taking into account half of all the faults, because each fault affects half of the terrain on average, so the faulting algorithms are not suitable for applications where a small visible area of a larger terrain needs to be generated.

2.1.3 Perlin noise

Noise is useful as a basic component when creating terrains. An example of a basic noise is smooth white noise limited to a specific frequency. It can be produced by assigning a pseudo random value of 0 or 1 at regular intervals in the plane, and then interpolating between these values using cubic or cosine interpolation. However, the resulting noise still has the artifact of having a derivative of zero at regular intervals in the space, possibly creating a repeating pattern.

Ken Perlin overcame this artifact by shifting the minimum and maximum values of the noise away from the regular grid. Perlin noise approximates smooth white noise of a given frequency in one or more dimensions *(see Figure 7)* [Per02] [Per06]. The noise is not completely limited to a specific frequency in the case of two or more dimensions, since if the frequency is f along the vertical and horizontal axes it will be $\sqrt{2}f$ along the diagonals.

White noise like this is not too useful as terrain in itself, but a very widely used application of Perlin noise is to combine several layers (called octaves) of noise at different frequencies and amplitudes to form a natural looking fractal noise *(see Figure 8)*. This combined noise is sometimes called Perlin turbulence, but often just Perlin noise.

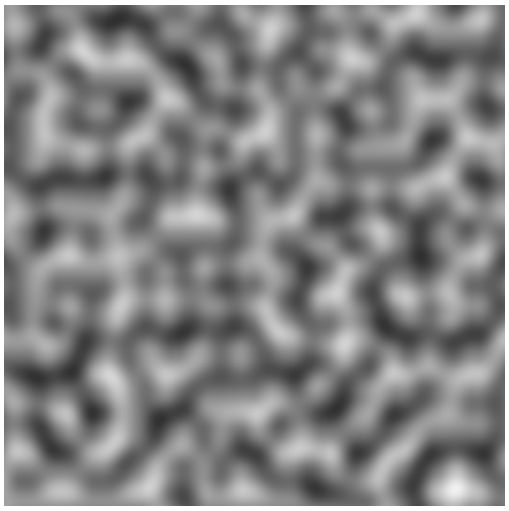

Figure 7: One layer of Perlin noise produced using the reference implementation at [Per06].

Figure 8: Perlin turbulence produced by combining the layers of Perlin noise seen in Figure 9.

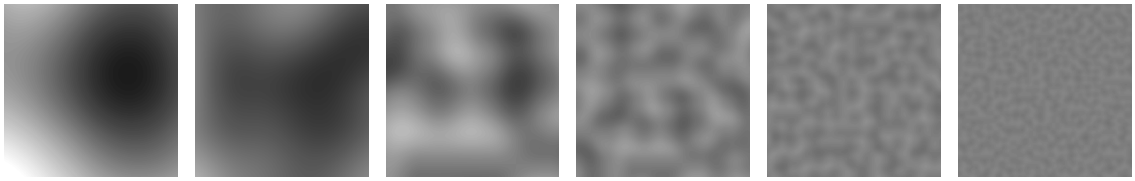

Figure 9: Perlin turbulence is created by combining many octaves of simple Perlin noise with different frequencies and amplitude.

The characteristics of a terrain height field produced with Perlin turbulence can be adjusted by changing the number of octaves and the amplitude and frequency of the octaves. A natural looking terrain can be achieved with four to eight octaves, where the amplitude of each octave following the first is halved and the frequency doubled, adding finer and finer details to a basic landscape *(see Figure 9)*.

Perlin turbulence does not have any regular visible artifacts like the ones affecting the subdivision algorithms, and is faster than the faulting and mass deposit algorithms, so it is a popular choice for random terrain generation in applications. It also has the advantage that it can be calculated for any position of the terrain without the need for calculating surrounding values, and it can be calculated up to any desired detail level (by adding more octaves if smaller details are needed), so it fits perfectly for an on-the-fly generated, multiple level of detail landscape.

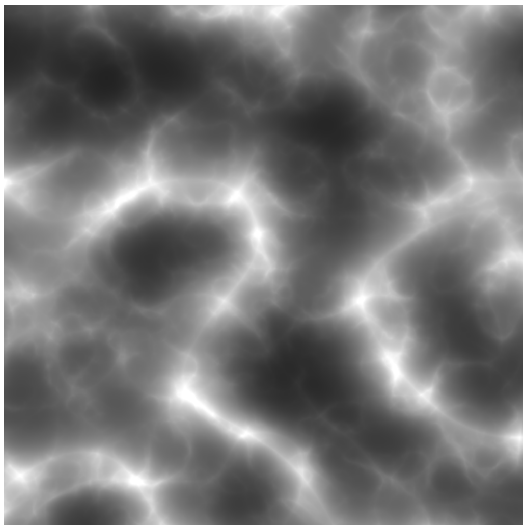

Figure 10: Ridged Multifractals can be used to approximate natural mountain ridges. Based on sources from [EMP+03]

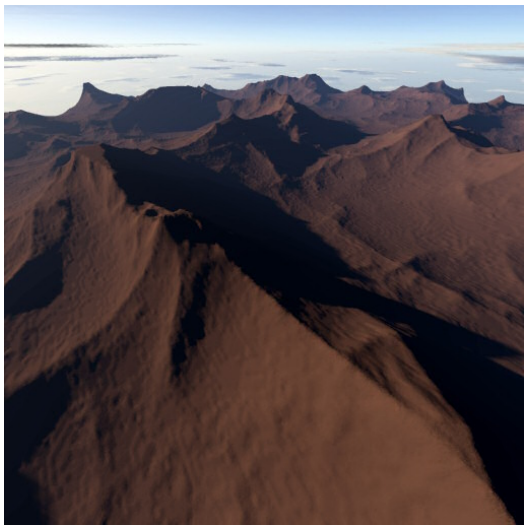

Figure 11: A 3D rendering of the ridged multi-fractal height map in Figure 10 (created with Terragen [Sof06]).

There are some interesting variations of Perlin turbulence, such as the ridged multi-fractal noise which uses absolute value functions to produce features with a ridged appearance. This fractal can be used to for example to approximate eroded mountain ranges *(see Figures 10 and 11)*[EMP+03].

2.1.4 Successive mass deposit

The successive random mass deposit techniques are based on the idea of repeatedly adding some mass at a random location of the terrain. The mass usually has a Gaussian distribution profile around the addition point. The addition is repeated with successively smaller masses. This resembles the faulting algorithms in that the running time is relatively long, and the result lacks the artifacts that most subdivision based algorithms have.

The advantage over fault based algorithms is that it is possible to calculate a small area of the terrain, and only take the mass deposits near that area into account, provided that every mass deposit has a finite area of effect. To avoid having to generate all the mass deposit positions and checking them against the target area, a quadratic subdivision data structure can be used that provides a random seed

for each square at different detail levels. This yields an algorithm that seems like it could be relatively competitive. It resembles the Perlin noise algorithm in that smoothly interpolated particles with smaller amplitude at smaller detail levels are added together, but unlike Perlin noise the particles are not in a rigid grid. On the other hand, in Perlin noise each pixel is calculated only once for each detail level, while in the random mass deposit algorithm each pixel is modified once for each mass deposit near it. To get a smooth result, it is best to cover the whole terrain at each detail level with deposits, so that no empty space remains at each detail level. This produces quite a bit of overlapping deposits in order to get a high probability of total coverage at each detail level. Thus Perlin noise can still be expected to be faster than successive mass deposit algorithms.

2.1.5 Successive uneven mass deposit

If we change the mass depositions from smooth Gaussian distributed mass additions into precomputed uneven, fractal like height maps with edges that fade out, we can manage with far fewer mass deposit points to get a random looking terrain. By transferring a part of the calculation to the pre-computation step we gain more performance at the generation step, while sacrificing some amount of randomness – after all, with a precomputed height map applied repeatedly on the terrain, we will get some repeating artifacts. These can be reduced by using a few different pre-computed height maps, and rotating them randomly, as well as combining height maps scaled to different resolutions on top of each other. In practice it will be hard to notice any repeating artifacts, unless the precomputed height maps contain very distinct details.

This algorithm also allows world designers to specify the deposit height maps for different resolutions, enabling detailed control of the characteristics of the landscape at varying levels of detail. It might be possible to use this algorithm to produce eroded looking terrain by preparing input height maps with eroded hills.

2.2 Geological effects on the terrain

The preceding terrain generation algorithms produce a fractal height field that may statistically resemble the real landscape topographies, but visually they lack many of the distinct geological features found in real world landscapes. These features are the result of various geological processes. Processes that modify the geography of an earth-like planet can be divided into two broad categories; topology building processes that move the planet crust or allow magma to rise to the surface, and erosive processes that on one hand wear down the planet crust to progressively more fine grained particles, and on the other hand transport these particles and deposit them in new places. In addition there are some other geography shaping processes, such as meteorite impacts, coral reef growth, and human activity.

This chapter will briefly present these processes, and examine various algorithms used to simulate their effects on artificial terrains. A randomly generated height map can be used as a base, and the geological processes applied to it to make it appear more natural.

2.2.1 Plate tectonics

Earth's surface consists of large, mostly solid continental plates, floating on top of a liquid inner core. The continental plates move as a result of magma flows in the inner core and the effects of gravity on the mass of the plates. At the seams, the continental plates push on top of each other, creating mountain formations, or flow away from each other, creating deep faults in ocean floors. There are also smaller faults inside the plates, caused by strain. Near the edges of the plates and in other places where the plates are thin, magma from the liquid inner code can reach the surface, producing volcanoes [SS02].

Plate tectonics has mostly macroscopic effects on the landscape. At a local level it can be seen as the general characteristics of the landscape (mountainous near plate borders, and more flat where the plate is older and more worn down). There are some smaller scale artifacts though, such as volcanoes and local faults in the landscape (where the land areas on the opposite sides of the fault have moved

vertically or horizontally relative to each other).

The effects of plate tectonics could be approximated by randomly dividing the surface of a planet into plates, and assigning some velocity for each plate. Plate border areas would have their height increased if the plates are moving towards each other at that place, and reduced if they are moving away from each other. In addition, volcanoes would be scattered along plate border areas.

In practice plate tectonics is a too macroscopic effect for many applications that focus on more local landscapes. In these cases large features such as mountain ranges could be defined manually by world designers.

2.2.2 Erosion

Erosion has a widespread and visible impact on landscapes, and is thus important to simulate for artificial landscapes to make them appear natural.

Erosion is the process of breakdown and transportation of landmass as a result of various natural phenomena. Breakdown of rock into smaller particles can happen through the effects of water, wind, temperature changes, glaciers, plant growth, animal activity, or chemical reactions. Transportation happens mainly through gravitation and suspension in moving water or air.

Erosion can be modeled by running several simulation iterations over a terrain, where each iteration moves some of the soil depending on the strength of the local water flow and the hardness of the ground [MKM89]. Erosion models usually take into account at least ground breakdown and transport by suspension in water. Vegetation tends to reduce erosion, by binding the soil more firmly into place with roots than it would be if the ground was bare. This can be simulated by treating vegetation covered ground in a similar way to harder ground, making it less prone to lose matter.

An iterative model for erosion does not fit well with the approach used in this thesis work, where the goal is to generate only the part of the landscape surrounding the viewer. The traditional erosion simulation would require storing the height data for the whole eroded terrain, or calculating the erosion on the fly.

The problem with calculating the erosion on the fly for only the visible part of the landscape is that erosion is not a local process - rivers may carry soil and water long distances.

There are also some statistical erosion models targeted for agriculture, but they are more focused on calculating soil and nutrition outflow over short time spans than the effects erosion has on the landscape shape over geological time periods [MQS+98].

Thus we need some kind of approximation for erosion that is faster to compute for a local terrain area. A promising starting point appears to be treating river generation separately from the local effect erosion has on the terrain shape. Rivers will be covered in the next section.

It seems relatively easy to locally approximate effects of erosion such as forming gullies in steep slopes and depositing sediment below them. This could be done on the fly for the area around the viewer, in a level of detail fashion, so that for far away slopes only the largest gullies would be created, while on closer by slopes there could be smaller gullies also.

Tendency for silt to gather in low lying areas and plains could be simulated by using filters that smooth the landscape for areas that are low compared to surrounding areas.

The typical ridge like pattern that mountains and hills form could be approximated by generating the ridges, starting from hilltops. Variants of Perlin noise, such as ridged multi-fractals mentioned above can also be used to approximate ridges. Another alternative is to use the successive uneven mass deposit algorithm for terrain generation suggested above, with pre-created hills with eroded slopes being added together to form a terrain that contains local eroded features.

2.2.3 Water systems

Generating rivers and lakes is problematic, as they are much more non-local, being affected both by water flowing in from upstream, and the amount of water that can flow away downstream, in addition to the local rain and evaporation.

There is not much research on river system generation. P. Prusinkiewicz and M. Hammel suggest a fractal river generation algorithm, but it only generates one river without branches, and cuts right through the terrain instead of following or shaping the terrain naturally [PH93].

Locating watersheds and drainage areas in existing elevation data has been studied extensively however, because of applications in geographic information systems and image analysis [McA99] [RM00]. Knowing the drainage areas and watersheds can be used to calculate a river network. The algorithms are non-local though, and require the whole terrain data to be available to work correctly, so they are not applicable for a level of detail based real-time landscape generation approach.

In the following I suggest an algorithm that does level of detail based river system generation. This algorithm has not yet been verified with a real implementation, so it is effectiveness and usefulness is unproven.

A water system is formed by drainage basins, rivers, lakes, and oceans, with some flow of water between the different components. A drainage basin is the land around a body of water that slopes down towards the water. Water appears in the system through rain. The amount of rain is given by local climate and weather conditions. Water disappears from the system through evaporation into the atmosphere, and absorption into the ground. Water may also appear into the system from ground water reservoirs, in the form of springs, although modeling that is optional. A river may start from nothing, or from one or more other rivers, or a lake, and may end in nothing, one or more rivers, or a lake/ocean. Other branch rivers may flow into the river along its way. The river has some influx of water at its start point, and some evaporation, soil absorption, rainfall, and branch river inflow of water along the way, altering the amount of water in the river, and producing the total out flux form the river at its end. Lakes, seas, and oceans have evaporation, rainfall, incoming rivers and drainage areas, and outgoing rivers that can affect the amount of water in them. Summing up all the in and out flows gives a total change in the water level over time, if it is zero the water level remains constant, otherwise it changes, which may affect the in and outflows also.

When generating and simulating a water system, a level of detail algorithm can

be used. Water system components can be divided into different detail levels, with e.g. 1 as the most coarse level. When rendering a landscape for a viewer, first all water system components at level 1 are generated, then the level 1 drainage areas close to the viewer are expanded into level 2 water systems, with subsidiary rivers, smaller lakes, and their drainage areas. The level 2 drainage areas close to the viewer can further be decomposed into level 3 water system components, and so on.

The route of a river between its start and endpoint is defined by a set of points. For the parts of a river closer to an observer, more points can be generated by offsetting points from the centerline using a one dimensional Perlin noise, giving the river a natural meandering appearance. The amount of meandering would be determined by the slope of the terrain.

To maintain the invariant that water always flows downhill, we can let the river cut into the terrain where it would otherwise flow uphill.

We can also raise the edges and lower the middles of drainage basins, to create a natural looking landscape with ridges and valleys.

2.2.4 Coral reefs and craters

Atolls are formed when a coral reef grows up around a central island, usually a volcano. When the central island is eroded down, the coral reefs continue to grow, forming a ring-shaped low island around a central lagoon. Atolls and meteorite impact craters can be created in a similar way, basically adding or removing land around a center point, using a distance based profile that tells how much to add / remove to the ground height / water level based on the distance from the center point. In the case of atolls some radius and amplitude variation depending on the angle can be added, to make them more irregular and natural looking. Coral reefs can also be created along costs by raising the sea floor to almost the water surface in irregular bands along the coast.

Meteorite impacts have been used in generated landscapes for a long time, an example is [MME98]. When generating a heavily cratered moon surface, the crater sizes should follow a distribution where there are few large craters, but

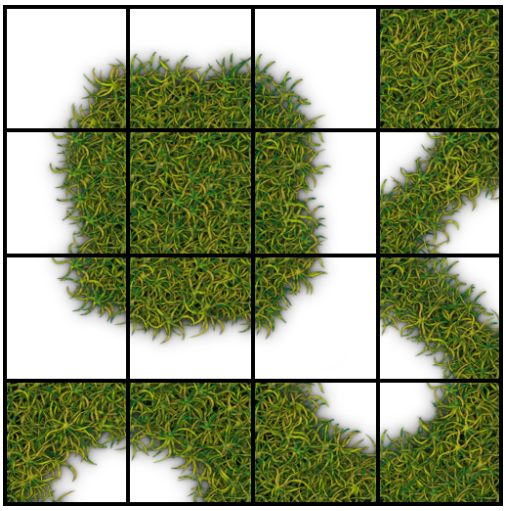

Figure 12: A set of 16 predefined border pieces that can be used to create a continuous edged texture on a landscape. The layout in this picture is modified to make texture painting easier than with the original layout presented in [LN03].

many smaller ones. On a planet with a thicker atmosphere such as earth, only the largest meteorites reach the ground and leave lasting impressions, smaller impact craters are usually quickly eroded over geological time spans.

2.3 Terrain texturing

After generating and rendering the geometry of the landscape we still need to texture it. Texturing means covering the triangles making up the landscape with images of the ground in that area.

A simple way to texture is to just repeat a texture over the landscape. This results in visible tiling artifacts, and a quite boring landscape.

To get rid of the repeating artifacts Wang tiling can be used [CSHD03]. It uses a number of different textures which match each other along some edges. We fill the plane with these textures, making sure adjacent texture edges match each other, to create a seamless, varying landscape texturing. This will still be a somewhat boring landscape, as only one set of textures is used.

To make the landscape more varying, we can texture different ecotopes with different textures. For example, mountain tops and hill slopes can look bare, while valley floors can be more lush. Using this method we blend between different textures based on the strength of different ecotopes. Blending can be implemented in two ways. The first way is by using the graphic card to blend the textures, by

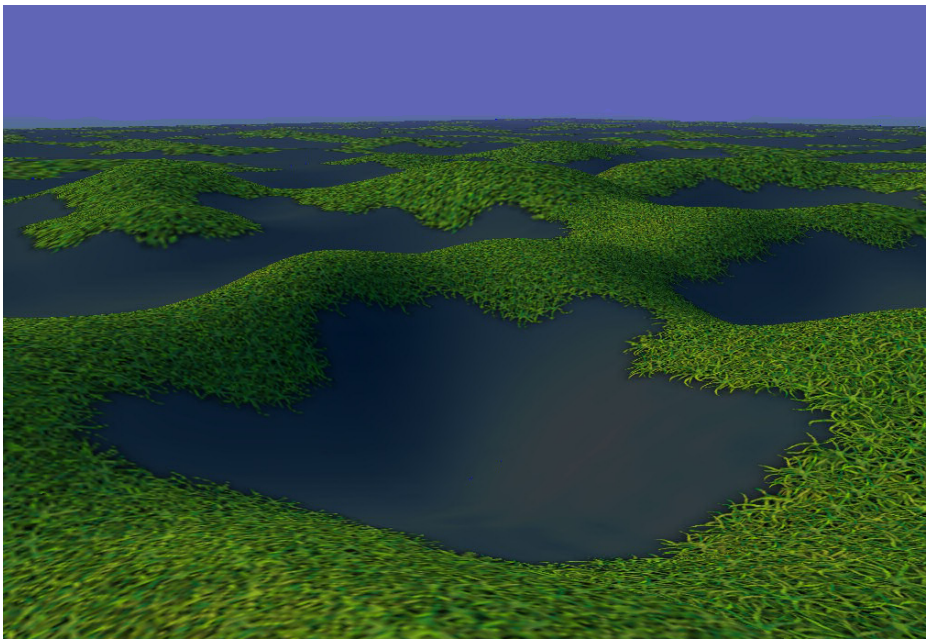

Figure 13: A landscape textured with a dark underlay color and a texture composed of the edge tiles in Figure 12 on top (taken from a landscape engine developed by the author during the thesis). The terrain height function was used to select the area to cover with the edged texture.

rendering the terrain multiple times, each time with a different texture, and varying the rendering opacity for each texture based on the strength of the ecotope associated with it. However, this increases the rendering time for the landscape noticeably. Modern graphics cards have multiple texturing units that allow us to render a limited number of textures (for example 8) at the same time, requiring only one pass. That puts a limit on the number of ecotopes we can use at the same time, and also prevents us from using the extra texturing units for other effects.

The other way to implement texture blending is to blend the ecotope textures in advance onto large terrain textures that cover the whole visible area, and then simply texturing the terrain with this single texture, stretching it over the whole visible terrain. This has the advantage that texture blending can be done in advance, at the point when new terrain appears in the view, and does not have to be done by the graphics card each frame. When preparing the terrain texture in software, we can also apply other effects to it easily, such as adding shadows and fallen leaves under trees, or drawing in paths and roads. The disadvantage is that the required texture size can be very large, if we want a detailed ground texture close to the viewer. However, this can be solved by using different terrain

textures at different distances. More distant terrain can have lower texture resolution. The ideal case is that the texture resolution varies with distance so that each texture pixel maps directly to a screen pixel. Thus a perfect implementation would only use as much texture memory as is needed for one picture the size of the screen. In practice, more is needed, but it is still a low, constant amount, making this texturing approach attractive.

Instead of simply blending between different terrain textures, more natural edges can be achieved by using custom textures for the edges between textures. This is usually implemented so that a given terrain texture type has some normal repeating tiles defined, as well as 16 different edge tiles, for all possible edge conditions that can arise depending on which texture should be present at each of the four corners of a tile *(see Figure 12)* [LN03]. The edge tiles can be drawn so that the underlying area is left transparent, allowing the topmost texture to be simply drawn on top of underlying textures. Figure 13 shows a landscape textured with edge tiles.

Different textures can be used for different levels of detail of an ecotope. If we use the same texture for far away ground that we use for close by ground, the details in it will be too small and too densely packed, resulting in a homogeneous mess. We can instead use own textures for far away areas, showing more macro scale features of the ecotope [Ham01]. At the transition from nearby to far away ground we blend between the different level of detail textures.

By defining elevation maps for textures, we can use pixel shaders to implement bump or normal mapping, which increases the sense of three dimensionality of the texture. But the texture does not actually modify the ground surface geometry, it just changes the appearance of it. However, we could also use the elevation data associated with a texture to change the height data for the actual geometry in the landscape. The elevation data could be specified with a greyscale image. Middle gray colors in the image would result in no changes in height, dark shades in lower altitude, and light shades in high altitudes. This would provide a simple way for things such as roads, stones, uneven ground, and other landscape features to have terrain topology that is better synchronized with the terrain texture.

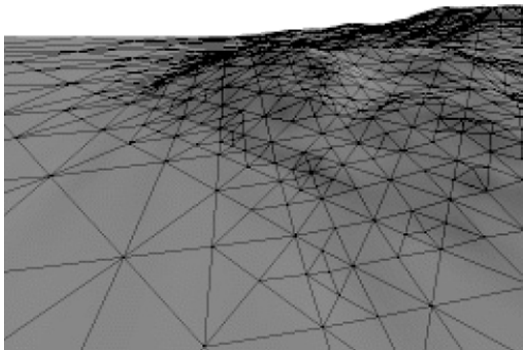

Figure 14: The ROAM algorithm in action. Note the adaptive subdivision of the landscape. From [DWS+97].

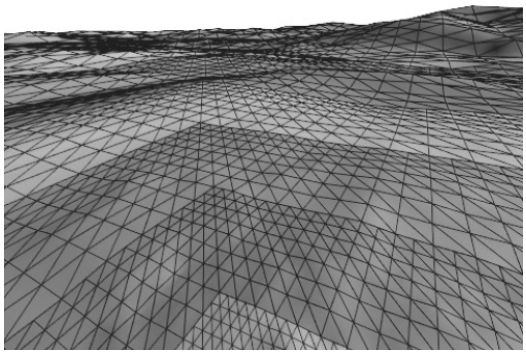

Figure 15: The geometry clipmap algorithm applied to a similar terrain. Note the concentric rectangular grids with decreasing level of detail. (Since the clipmap algorithm implementation is more recent, the total number of triangles is higher). From [LH04]

2.4 Terrain rendering

There are several different approaches for interactively rendering terrain, from pre-computed triangular irregular networks (TINs) to adaptive triangulations and to hardware optimized regular tilings, and various hybrids of these, for example using blocks of pre-computed triangulations loaded on demand. In the following I focus on two algorithms that both allow dynamic on-the-fly generation of the terrain.

The ROAM algorithm is based on a triangle subdivision strategy [DWS+97]. The terrain is rendered using triangles that are subdivided until they are detailed enough to relatively accurately represent the visible landscape. The difference between the landscape height map values and the actual height values visible on the screen is measured as a pixel error value (how many pixels the height differs from the correct one). As the viewer moves, if the error rate climbs above a certain threshold, the concerned triangles are further subdivided to give a more accurate representation of the height map. Two adjacent triangles will be joined back together into one triangle if the error rate would stay below the threshold even if they are joined *(see Figure 14)*.

However, modern hardware is optimized for bulk rendering of static geometry and does not cope well with the frequent changes to the geometry that ROAM requires. The new generation of landscape rendering algorithms focus on ways to achieve efficient terrain rendering using the features of modern hardware. One of these algorithms is the geometry clipmap, developed by F. Losasso and H. Hoppe [LH04]. It renders the terrain on a series of nested regular grids with different levels of detail. The innermost grid has the highest detail and each surrounding grid has half the resolution but twice the size (keeping the amount of information constant on each grid). The center of each grid is cut out and occupied by the next more detailed grid, except for the center-most (most detailed) grid, which is solid *(see Figure 15)*. The geometry near the outer edge of each grid is morphed towards the lower resolution geometry of the surrounding grid to ensure that seamless transitions between the grids of different resolution. Texturing of the ground can be handled using the same division into different level of detail areas. This method has the advantage of supporting iteratively refining height and texture data calculation. If elevation data is calculated from several layers of Perlin noise, each more detailed layer can be easily added to the already calculated coarser layers for a new region that becomes visible, making simple on-the-fly terrain generation very fast. More complex terrain generation algorithms may not be able to take advantage of already calculated coarser height data, but at least they only need to calculate data for new areas as they become visible, as there is no pre-computation step. A more recent version of the algorithm works almost completely on the video card, utilizing recent vertex program features for rendering, and storing both the elevation data and the terrain color in texture buffers [AH05].

3 Plants

After the terrain has been generated, there is still something missing before we have a natural looking landscape. Most visually important among natural entities are the plants, from grasses to trees.

There are also some other natural entities, such as stones, which can use similar distribution algorithms as plants.

3.1 Plant ecosystems

The plant distribution in a real landscape depends on many things, such as the soil chemistry, local climate, competition with other plants, and the botanical history of the area. Plant growth locations in an area is an iteratively evolving system, where the previous state, along with changing external environment variables determines where new plants take seed, and how existing plants grow and die. Models of such evolving plant landscapes have been used to create plant distributions for computer graphics, for example in [DHL+98]. Over longer time intervals different types of species replace each other and the soil itself changes, a phenomenon called ecological succession. For example, after a forest fire the charred ground is first populated with grasses and plants specialized for open areas, followed later by bushes and fast growing trees, and later by trees that grow slower and can grow shadowed by other trees.

Simulating iterative processes like plant population growth is computationally expensive if the landscape needs to be generated on demand for visualization, so some way to approximate the result of the simulated process is desirable. Ecological succession is so slow that it is not necessary to simulate for most applications. Statistical analysis of plant growth locations indicate that individuals of some plant species tend to grow together in groups, while in the case of some other species the plants grow at some minimum distance from other plants, and yet other species are distributed in a more random way [SLDD93]. This results in visually distinct patterns that can be approximated with non-iterative algorithms.

In addition, different plants species prefer different growth conditions, some re-

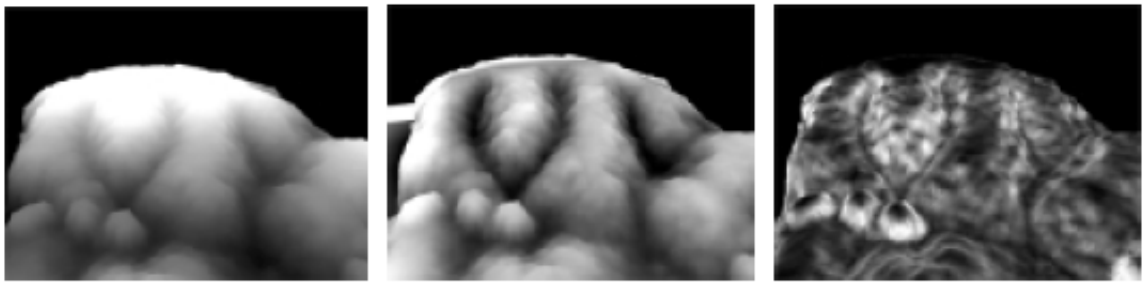

Figure 16: Screenshots showing (from left to right) elevation, relative elevation, and slope functions calculated for a mountainous area. From [Ham01].

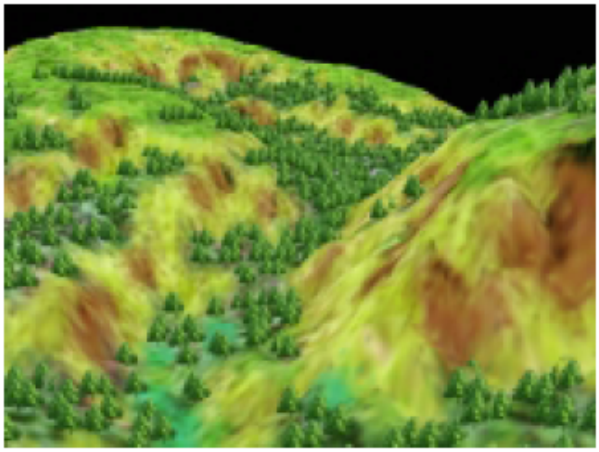

Figure 17: Distribution of trees over a part of the mountain seen in Figure 16. The tree distribution function is placing trees at mostly low altitudes, preferring low relative elevation (valleys protected from the wind), and avoiding steep slopes. From [Ham01].

quire the shade and moisture of a forest valley, and others manage in more dry and sunny places like hill slopes. The ecotope of an area can be used to define the general conditions for an area, specifying the types of species present and their approximate density. The local growth conditions, such as soil type, soil moisture, amount of sunlight, and micro-climate can be deduced from a combination of the ecotope properties, terrain height map, and pseudo-randomly generated two-dimensional noise functions. The slope and relative height of the terrain also affects the soil characteristics and micro-climate of an area *(see Figure 16)*. The amount of sunlight or shade in a given area can be determined by first generating larger plants such as trees, and after that generating smaller plants like herbs, taking into account the average shade cast by the trees at the location. The random noise functions can be used to add some variation and appearance of naturalness. This approach to plant distribution has been used for example in [Ham01] *(see Figure 17)*.

3.2 Plant distribution

The factors affecting plant growth described in the previous section are combined into two dimensional density functions that specify the densities of given plant species. The placement of a plant is also affected by the presence of other nearby plants. On one hand plants of the same species often tend to grow close to each other because seeds are deposited close to the mother plant, and because plants of the same species like similar growth environments. On the other hand plants are competing for resources such as sunlight, water, and nutritions, and this tends to lead to bigger plants dominating smaller ones, as they manage to grab most of the resources and the smaller plants stop growing or die. So there can be both attractive and discouraging zones around a single plant, at different distances, that affect the probability of other plants to appear there. The zones can be different between different species, for example the same species may tend to cluster together, while different species may only compete with each other.

The plant distribution can be calculated in an iterative manner by simulating the growth of plants in an area, using a simple ecological model where plants grow over time, bigger plants will dominate adjacent smaller ones, and too dominated plants as well as too old plants die out. This approach was used in [DHL+98] and presented as an alternative in [LP02]. However, due to its iterative and simulation focused nature it has too low performance to be useful for real-time on-demand landscape rendering.

By modifying the plant density function as each plant is added, both clustering and minimum distances between plants can be simulated. As each plant is added, it modifies the density function around it with a modification kernel to either encourage more plants at a certain radius around it, and/or to discourage plants at some other distance *(see Figure 18)*. The algorithm is described in [LP02]. The probability deformation kernel approach has the problem that it is non-local, an initial plant may have a deformation kernel that affects the placement of a plant next to it, the plant next to it may affect the placement of yet another adjacent plant, and so on. If we only want to generate a certain area, we may lose the effect of the original plant in the chain, and thus all the other plants in the chain that were affected by its deformation kernel will be in other locations.

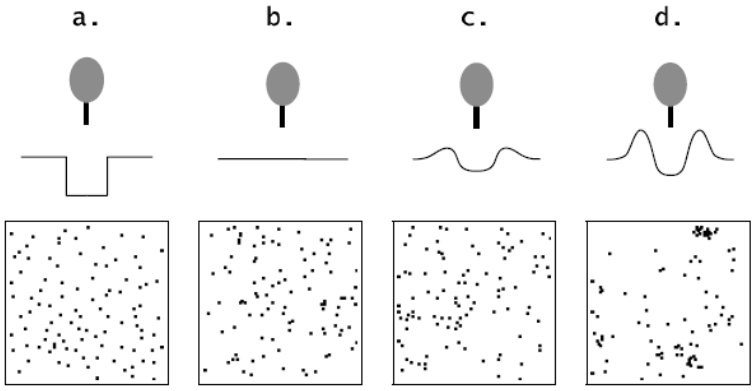

Figure 18: Different density modification kernels. The (a) kernel just makes sure plants do not get too close to each other. The (b) kernel does not modify the distribution, resulting in a random distribution. The (c) and (d) kernels encourage clustering of plants, while also avoiding too small distances between plants. From [LP02].

The clustering can also be approximated already when generating the density distribution, by multiplying the original density field with a high contrast fractal noise function to get a very clustered distribution, or with a smoother, less contrasting noise function to get a more even distribution. The density functions can then be converted to a set of points using a grid based approach. The area is divided into rectangular grids with some side length r. For each grid cell, the density function is sampled with a sampling size equal to r. The density is then divided by the area of the grid cell to get the number of points in that cell. If the density is not an integer number, a random number between 0 and 1 is generated, and compared to the remainder of the density. If the random number is smaller than the remainder, one is added to the number of points to generate. The points are then randomly placed in the cell using a pseudo random number generator, which is seeded with the cell coordinates, so that the points in a given cell are always generated at the same positions. This algorithm has the advantage of being relatively local, only the plant locations in grids overlapping the visible area need to be calculated. The disadvantage is that there is no minimum distance between plants, trees may grow into each other. In practice this may not be a very serious problem.

A dithering algorithm might be used to avoid trees growing too close together, as it creates a relatively even distribution of dots to represent the specified plant density. Instead of adding points at random positions in a grid cell, points could

be added according to a rasterization pattern that depends on the density. The downside is that there might be some regular artifacts, and the error diffusion algorithms that give the best results are non-local in nature [Ost01].

3.3 Plant generation

Plant models could be hand crafted, but there are several advantages with using one of the many plant generation algorithms. Using an algorithm cuts down the modeling time for a plant significantly, and allows producing many plants of a given type by varying the random seed used in the generator. Generated plants also allow easy parametrization, that is, environment, time of the year, as well as plant age can be used as inputs to the plant generation algorithm to produce uniquely tailored plants.

In the following sections we explore three different plant generation algorithms, and ways to model the effects of the local environment on plants.

3.3.1 L-systems

The seminal work in the field of realistic plant generation is the L-System method developed by Aristid Lindenmayer [PL90]. An L-System consists of a set of string rewriting rules that are repeatedly applied to an initial seed string. Each character in the string is then interpreted as representing some geometrical structure or operation. When the geometrical representation of the L-System is created, each structure or operation is applied in sequence to the geometry, at a position indicated by an insertion cursor. When a stem segment is inserted for example, the cursor moves to the end of the segment, pointing in the direction of the segment. Structures can be things like plant stem segments, leafs, or petals, while operations can be rotations and translations of the cursor, or branchings that execute a specified set of operations and then return the cursor to the position it was in before the branch. By using operations with randomness, it is possible to create more natural looking plants.

An example of a plant generated with an L-System is shown in Figure 19. The

Figure 19: A plant generated by an L-System. From [PL90], page 27.

n=5, δ=18°

ω : plant
p_1 : plant → internode + [plant + flower] − − //
[− − leaf] internode [+ + leaf] −
[plant flower] + + plant flower
p_2 : internode → F seg [// & & leaf] [// ∧ ∧ leaf] F seg
p_3 : seg → seg F seg
p_4 : leaf → [' { +f−ff−f+ | +f−ff−f }]
p_5 : flower → [& & & pedicel ‘ / wedge //// wedge ////
wedge //// wedge //// wedge]
p_6 : pedicel → FF
p_7 : wedge → [‘ ∧ F] [{ & & & & −f+f | −f+f }]

Figure 20: The L-System used to generate the plant in Figure 19. From [PL90], page 27.

L-System used to generate it is shown in Figure 20. Here is a brief explanation of the syntax and symbols used in the L-System: F stands for forward movement that produces a line, f stands for forward movement without drawing a line, [and] are stack operations that save and restore the insertion turtle position and orientation, { and } create a filled polygon of the path the turtle takes within them, and + - & ∧\ / | stand for turning left, right, down, up, rolling left, rolling right, and turning around, respectively. The initial string is ω. Each iteration all named symbols in the string are replaced with the characters they represent ($p_1 - p_7$). The number of iterations to run is specified by n, and δ specifies the rotation angle for the various turning operations.

L-Systems have been expanded by others to simulate light-seeking behavior of plants, gravitation, effects of pruning, and competition for space [MP96] [PJM94].

Traditional L-Systems are powerful, but not very intuitive to create, as they are built up of many transformation rules whose effect on the final plant can be difficult to anticipate.

3.3.2 Component based plant generation

A more intuitive component oriented approach to plant modeling has been developed by O. Deussen and B. Lintermann in [LD98]. In it, plants are composed of higher level components, where each component has various parameters of different types, possibly including other components. A branch component for example has parameters specifying the component to use for sub branches, the component to use at the end of the branch, and a number of parameters modifying its shape *(see Figure 21)*. An editor application allows the designer to interactively connect component instances together into a directed graph, edit their parameters, and get a real time preview of the produced plant *(see Figure 22)*.

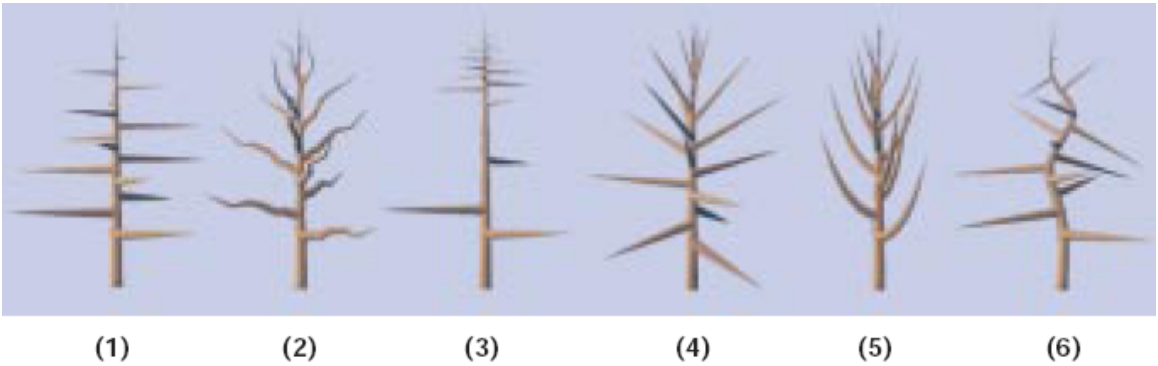

Figure 21: The effects of different branch parameters. (1) default outline; (2) gnarled branches; (3) branching density; (4) branching angle; (5) phototropism (branches grow towards the light); (6) stem curvature. From [LD99]

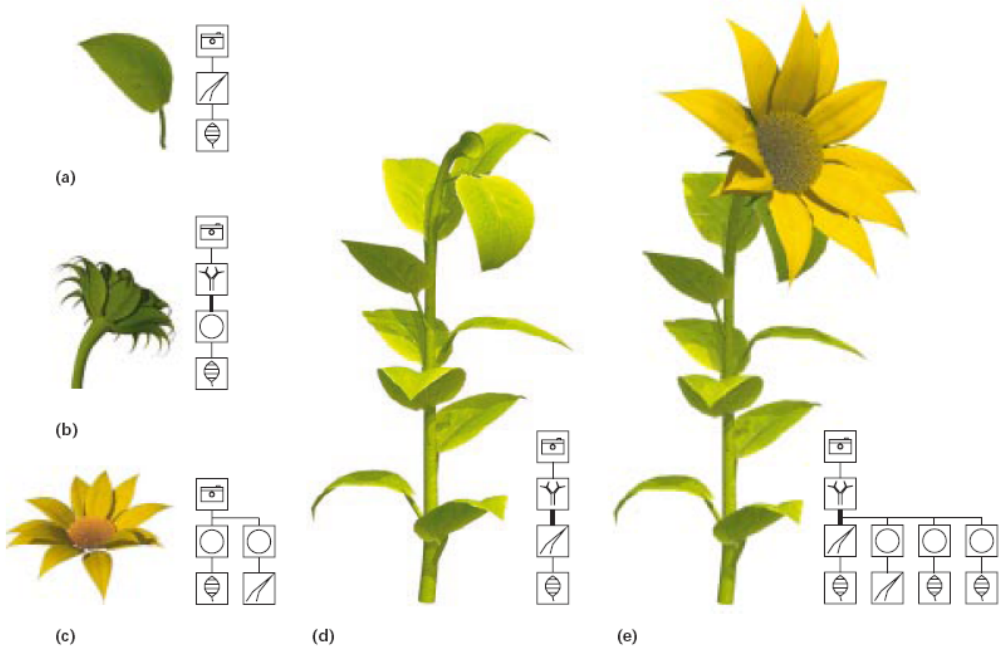

Figure 22: A component based plant, showing the directed component trees and the 3D models generated by them. The parameter settings for each component are not shown. From [LD99]

Figure 23: A Black Tupelo without and with foliage. Specified by 33 different parameters. From [WP95]

3.3.3 Parametrized trees

A further simplification of the plant modeling process is to represent a plant species as a collection of high level parameters that the world designers can edit. Jason Weber and Joseph Pern use this approach when modeling realistic trees [WP95]. They focus on the restricted plant domain of trees, and identify a set of around thirty parameters that can be used to describe different kinds of trees. Instead of trying to model the tree growth from a botanical point of view, they focus on defining the geometrical structure of the tree. They divide the branch system into three or four recursive levels of branches, with the stem being level 0, primary branches branching from the stem level 1, secondary branches level 2, and so on. Branches can have child branches growing out along their length, but they can also split into several cloned branches . The cloned branches remain at the same recursion level than the original branch. Each branching level has its own set of parameters, such as the angle between the parent branch and a child branch. In addition there are some global parameters, such as the height of the tree and parameters defining a pruning envelope for the tree. The pruning envelope is used to more precisely define the shape of the tree than the various branching parameters allow. It defines a parametrized space, and branch lengths

are adapted so that the whole tree fits inside the pruning shape. There is also a parameter that allows blending between the unpruned and the pruned shape of the tree. The parameters are at a high enough level that they can be set by studying pictures of trees and manually measuring branching angles, relative branch lengths, and so on. Most parameters also have an additional variation parameter associated with them, describing the amount of random variation to apply to the parameter when using it. The model allows the appearance of most normal trees to be captured very realistically, even palm trees and some cacti can be modeled *(see Figure 23)*.

3.3.4 Environmental effects on plants

In nature, single individuals of the same plant species can differ considerably depending on the local conditions near their growth location [SLDD93]. For example, if one moves plants that normally grow in lowland areas to mountainous areas, they usually grow lower, have fewer flowers, and so on. Trees that grow in forests tend to be long and thin, with most of the foliage near the top, while trees growing alone in open places can be lower but with a wider branch system and denser foliage.

To model variation of the plant parameters based on growth location, we can introduce a few variables that describe some of the key conditions at the growth location, and allow the world designers to write formulas for the plant parameters that include the environment variables, or define several variants of a single plant species for different types of locations, and blend between the parameter sets based on the local environment.

Environment variables could be for example average temperature, fertility and moisture of the soil, density of surrounding plants of the same size, and amount of sunlight or shade. The environment variables can be provided by the ecotope of the area where the plant is located.

3.4 Rendering plants

The above plant modeling systems all produce detailed plants where a large number of polygons are used. This is acceptable for still renderings, but for a real time application with a large number of visible plants it is not a feasible approach with current hardware. The basis for the different rendering optimizations is that distant plants can be simplified, as the amount of visible detail is not as high as if the plant was viewed at a close distance. The plant simplification methods can be roughly divided into impostor based algorithms and geometry simplification. It is also possible to speed up rendering of forests or fields by using one impostor to show several plants. There are even some methods that use line and point primitives for rendering plants [DCSD02], but because of the amount of transformation operations needed for this the performance is not as good as for texture based methods. An overview of various real-time plant rendering algorithms is presented in [MTF03b].

3.4.1 Impostors

The meaning of the word impostor is not unanimously agreed on in the literature. In this thesis I use it to mean a texture based simplification of a complex object that can be used as a faster stand-in instead of the original object geometry to render the object.

Billboards are a form of impostor commonly used for complex objects that are some distance from the viewer. The complex object is rendered to a texture with a transparent background, and the texture is shown on simple rectangular polygons that are always turned towards the camera *(see Figure 24 a)*. The billboard may turn freely towards the camera (useful for round shapes, like clumps of leaves), or only turn around the vertical axis (useful for objects placed on the ground, such as trees). They work well at some distance from a stationary viewer there is not much difference between using the real object or an image of it, as long as the lighting conditions do not change too much from when the objects was rendered to the texture. But when the viewer moves around the billboard, the way the object on the billboard stays the same but the billboard turns towards the viewer gives an unrealistic impression of the object turning towards

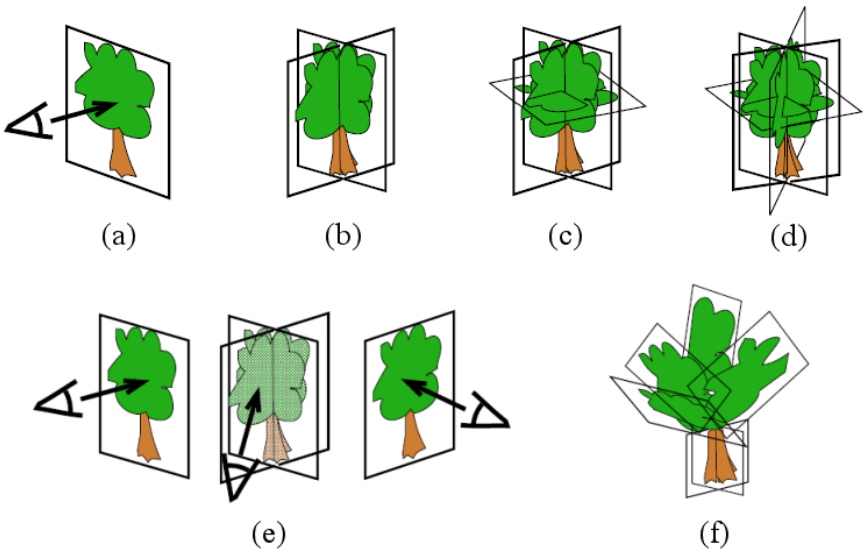

Figure 24: A taxonomy of impostors for plant rendering: (a) billboard; (b) crossed planes; (c) three crossed planes; (d) four crossed planes; (e) multi-directional billboard with cross fading; (f) simplified textured tree. Figures borrowed from [DN04].

the viewer. One solution to this problem is to render multiple views of the object, from different angles, and then blend between them as the billboard turns *(see Figure 24 e)*. However, at short distances the lack of any parallax effects or depth in a billboard makes it appear flat.

Nailboards are an improvement to billboards, suggested in [Sch97]. They improve billboards by including depth data in the billboard – for each pixel in the texture, there is a pixel in a greyscale texture that stores its distance from the nailboard center plane. Nailboards utilize the z-buffer, which is a floating point image buffer the size of the screen, used in the low level 3D rendering architecture. The z-buffer normally stores the distance from the viewer to each pixel on the screen. When a new pixel is being drawn, it is only drawn if it is closer than the previous pixel, as indicated by the distance value at that point in the z-buffer. After the pixel is drawn, the z-buffer is updated with the distance to the drawn pixel, at the location of the pixel. When rendering a nailboard, the depth value of each pixel plus the distance to the nailboard from the viewer is compared to existing depth values in the z buffer, and the pixel is only rendered if it is closer than any previously drawn pixel at that point. This technique allows nailboards to partly overlap each other and other objects in the scenery in a realistic way. This gives good results when rendering clumps of foliage for trees, as demonstrated

in [SK04]. Nailboards are implemented with programmable fragment shaders, so they are a bit slower to render than normal textures on current hardware.

Instead of always turning a billboard or nailboard towards the viewer, one can use several textured planar surfaces, locked in place in space. For example, a simple tree can be modeled as two rectangular planes with an image of the tree on a transparent background, placed so that they form an X when viewed from above *(see Figure 24 b,c,d)*. This arrangement avoids the turning towards the viewer -effect that normal billboards have, but from close up the plant still looks unrealistic.

A tree model may be manually crafted or automatically generated to use a low number of textured planes for foliage instead of trying to achieve realism by modeling each leaf with geometry. In this case the tree consists of 3D geometry for a trunk and the larger branches, and textured planes or billboards for the foliage *(see Figure 24 f)*.

A set of textured planes can also be algorithmically generated from a 3D model [DDSD03]. The algorithms try to approximate the appearance of the original model by placing planes in space to capture the original model's most prominent geometrical features, and then rendering parts of the original model to textures on the planes. This can produce quite good approximations for many objects, although trees are a problematic case because of the fractal, irregular nature of their foliage. A simple approach is to just divide the foliage by a number of textured planes placed at regular intervals along the ground coordinate axes, and render the foliage to the closest plane *(see Figure 25)* [Jak00]. The tree trunk can be rendered as geometry as it contains relatively few polygons. Algorithms that take the tree shape better into account can achieve better quality with fewer polygons, however. It is possible to use structural information about what parts of the model belongs to which branches, and create different planes for different branches *(see Figure 26)* [CCDH05] [BCF+05].

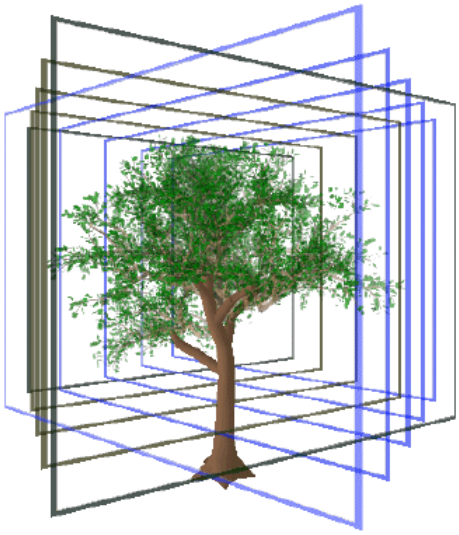

Figure 25: The foliage of a tree rendered to a set of crosswise textured polygons. From [Jak00]

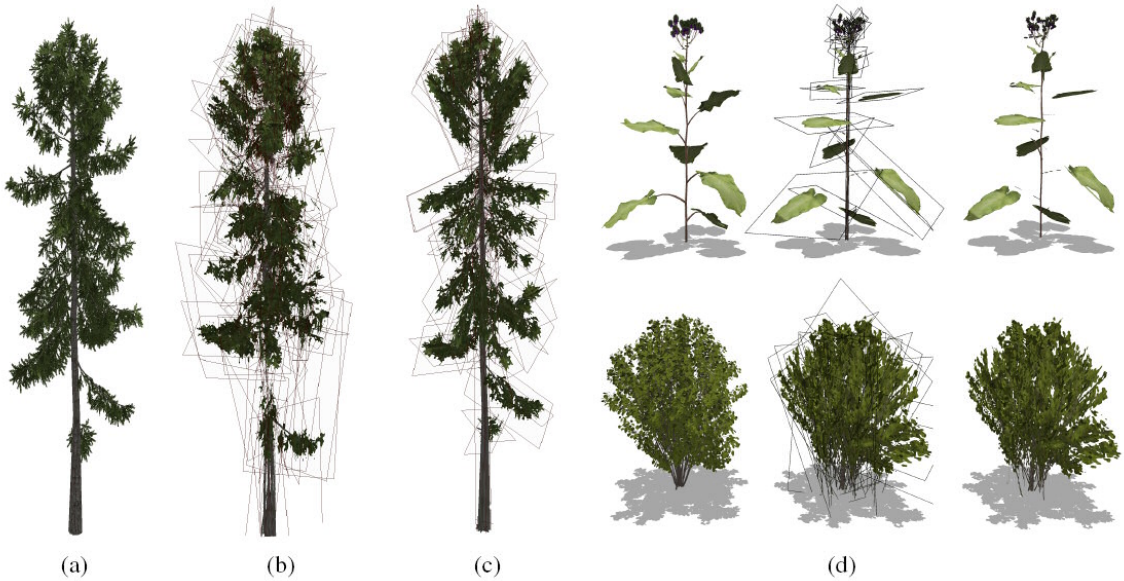

Figure 26: Tree approximation using algorithmically placed texture planes (billboard clouds): (a) original model; (b) billboard approximation using k-means algorithm; (c) approximation using improved clustering and hierarchial model information; (d) approximations of other plants using billboards. From [BCF+05]

3.4.2 Multiple levels of detail

For far away plants, a simple billboard or two crossed planes will look natural, the plant is so small on the screen that the lack of depth in it is not noticeable. At medium distances, such simple impostors start to look unnatural, and some higher detail approximation of the real plant could be used instead, such as the multi plane approximations. At very close distances, the real geometry of the plant may be needed to achieve the best realism. This switching of representations for an object based on the distance to it is called multiple Level Of Detail (LOD). Switching between different detail levels is not a seamless operation however, and may cause visual 'pops' in the object appearance, or 'LOD waves', as objects at a specific distance from the camera change between two different levels of detail. These effects may be reduced to some degree by cross fading between the two representations at different levels of detail, one representation becomes more transparent until it disappears, and at the same time the other representation become more visible at the same place until it is solid.

A single tree can also contain multiple levels of detail, branches and foliage parts on the backside of the tree can be rendered at a lower level of detail, while foliage and branches closest to the viewer can be rendered at a higher level of detail. As the closer foliage occludes much of the foliage on the other side of the tree, this produces relatively good results and allows faster rendering of trees.

The branch system and foliage in a tree can use different kinds of level of detail representations. The branch system can be reduced in detail by simply progressively eliminating the smaller branches, as they become too thin to be visible at increasing distances. The foliage can consist of many groups of smaller billboards at a close range, and cross faded to fewer larger billboards with increasing distance. At some far distance the whole tree can be represented by a single impostor consisting of a few textured planes [SK04].

For real time rendering applications a fully detailed tree model with geometry for individual leaves is too slow to render with current graphics hardware, and a simpler model with impostor based foliage can be used even for high level of detail tree representations.

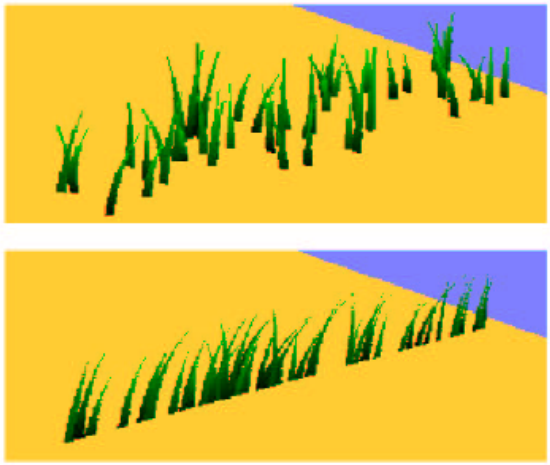

Figure 27: Two different levels of detail for a part of a grass field. In the upper picture (used close to the viewer) each grass plant is rendered as a separate 3D model. In the lower picture (used at longer distances from the viewer) the grass models have been rendered to a textured polygon, allowing much more efficient rendering. As the texture has been created by rendering the 3D models to it at the correct positions, it is easy to do a smooth transition from individual plants to a textured polygon impostor and back. From [PC01].

3.4.3 Rendering dense fields

We will run into problems if we want to represent something like a dense forest or field of grass a as individual plants.

For very far away forests or fields, we can simply use a suitable ground texture, and not render the individual plants at all. But at medium ranges, modeling individual plants can be too performance costly on current graphics hardware. A solution is to render several plant models to textures, and use the texture on impostor polygons facing the user. When the user gets closer to the polygons, polygon based plant models are inserted instead of the impostor polygon *(see Figure 27)*.

Depending on the structure of the plants, the impostor polygons could be either vertical (for tall grasses, trees, etc.), or several horizontal layers (for ground cover plants), or some mixture of these. The plants could even be animated using a simple spring system and force fields, to simulate the effects of wind [PC01].

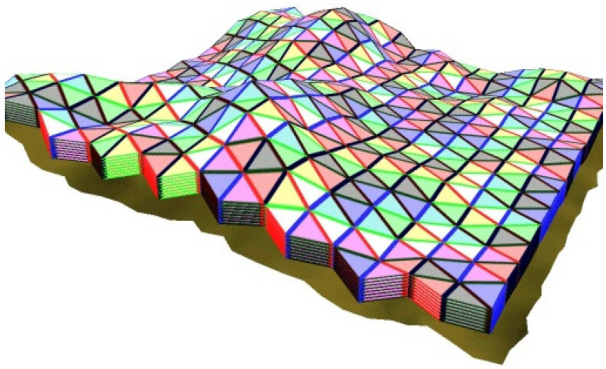

Figure 28: The layout of the textures making up the forest. Several texture layers parallel to the ground shape are used. The textures are pre-rendered for the different heights in the forest, and tiled using Wang tiling to create an arbitrary large seamless but homogeneous forest cover. From [DN04].

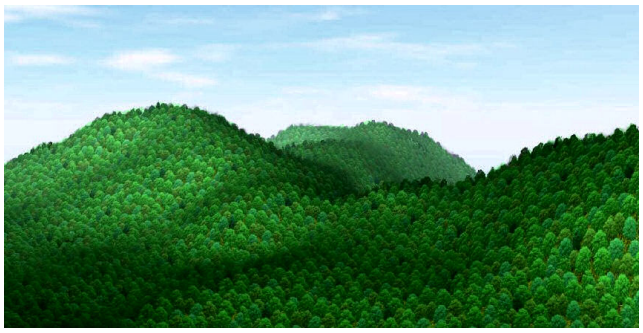

Figure 29: Thirty thousand trees rendered in real time, using layers of textures *(see Figure 28)*. From [DN04].

Volumetric textures are an extension of this approach. The ground is divided into triangular or square patches of some size (usually matching the terrain polygons). For each patch, the parts of the plants closest to the ground are rendered on one texture, the slightly higher parts on another texture, more higher parts on yet another texture, and so on. These textures are then rendered above the patch of ground, at the same tilt as the ground, but at different heights *(see Figure 28)* [DN04]. This gives good looking plants even at medium ranges, and has the advantage that many plants can be rendered on a single texture, arbitrarily dense, without increasing the rendering cost *(see Figure 29)*. The disadvantages are firstly that if seen directly or almost from the side, one can see the individual planes and through them. To counter this more planes can be used to make this worst case viewing angle smaller and to give more depth to the forest. This again leads to the second disadvantage, which is a prohibitively high memory consumption (and slow pre-rendering phase) if each part of the forest should be unique, showing exactly the plants growing at that position. To solve this, one can define a few squares with random plant placement, and repeat those squares over the whole forest. That means that there is less room to make each location unique, although squares with different tree densities could be used based on forest density at a given point. Wang tiling can be used to reduce repetitive

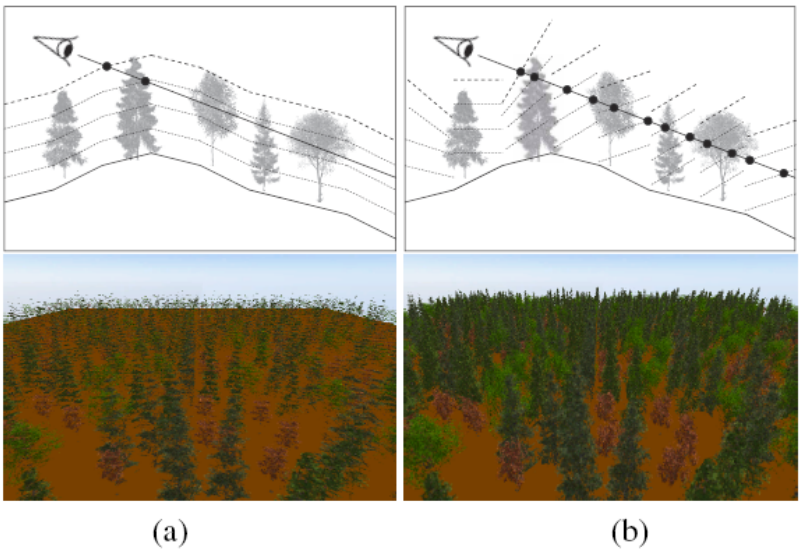

Figure 30: Slanting the slices to improve the view at small incidence angles: (a) No slanting, the layers follow the terrain shape; (b) Slices turned towards the viewer. From [BCF+05].

tiling artifacts [CSHD03], or edged textures can be used to implement transitions between different types of forest [LN03].

An improvement to the volumetric forest algorithm is to tilt each plane some amount towards the viewer, to avoid the case of seeing a plane directly from the side. This way the forest will appear more thick with less texture planes used. It also distorts the volumetric texture a bit, but in the case of trees in a forest the distortion is not very noticeable *(see Figure 30)* [BCF+05].

A further improvement is a level of detail approach, where the number of rendered planes is decreased when the distance to the camera increases. This can be done by simply merging the textures of planes together to create lower level of detail planes. By using cross fading this level of detail transition can be made very smooth and unnoticeable *(see Figure 31)* [BCF+05].

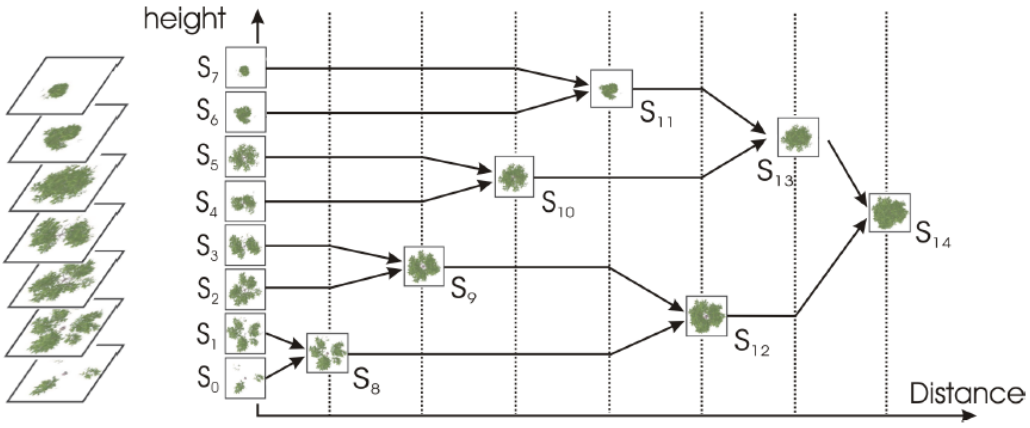

Figure 31: Level of detail for slices. Distant areas are rendered with fewer slices that contain the merged details of the more detailed slices. From [BCF+05].

4 Buildings

Buildings are an important part of many virtual landscape applications. They have less complex surfaces than plants, yet they present their own challenges for landscape generation and rendering.

4.1 Building generation

There is a lot of research on generating buildings in cities from satellite photos [KZGP06], multiple aerial photos from different angles [BZ00], and some research on generating buildings from architectural building floorplans [hOWkYyC05] and ground level photographs [LD06], but they all concentrate on creating 3D models for existing real-world buildings. There is also a significant amount of research about creating procedural cities [GPSL03] [PM01], but that research concentrates on generating building models (mostly high rise buildings) for outside view only, without any internal structure. There is very little research on generating virtual three dimensional houses with internal rooms from scratch.

4.1.1 Building parametrization

In order to make generated buildings more unique and better fitting into their environment, it should be possible to parametrize the architectural style, usage, and history of a building to some degree. The architectural style determines the surface materials (textures) used for the building, as well as what kind of decoration is used, the roof style (flat, sloping, gabled, etc) and some general properties about how the building is laid out (how symmetrical it is for example). The usage specifies what functions the building serves, who lives there, what kind of businesses are located in it, and what kind of needs they all have for places to work, store things in, eat, sleep, and relax. These needs drive the partition of the building lot into rooms with specific roles. The history of a building tells how old it is, what it has been used for before, and how well it has been maintained, whether it has been renovated, and so on. The history is not as essential as the architectural style and functions for the appearance of a building.

A building can be abstracted to consisting of a number of rooms connected to each other. Each room is allocated for one or more usages. Usages have different requirements regarding room size, furniture in the room, and access to rooms with related functions. For example, a workshop might need a workbench, a place to store tools, and easy access to a rooms with storage space used to store raw materials and produced goods. A given room layout can be evaluated based on how well the requirements of the usages are met, and thus how well the building serves the needs of its inhabitants. This evaluation function can be used when searching for a good room layout in a building generation algorithm.

4.1.2 Room connection graph

One of the few existing publications on house generation is [Mar04], where Jess Martin describes a framework for generation of buildings with interior rooms. The framework uses a three phased approach, where first a room connection graph is generated, then the rough room layout is done by placing out the graph nodes in space, and finally walls and doors are created around the room nodes. The result of the first step is a undirected graph with rooms at the nodes and connections (doors) along the edges. The result of the second step is the same graph,

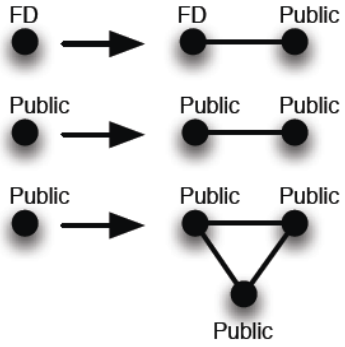

Figure 32: A part of the room connection graph creation rules. From [Mar04].

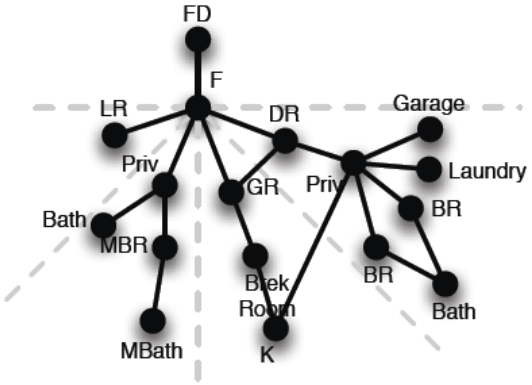

Figure 33: A radial room layout. A floorplan starts with a Front Door (FD), with various public rooms and private rooms connecting to it. From [Mar04].

but with coordinates in 2D space (or 3D space for multi-story buildings) for the rooms. The output of the final step is a floorplan with the walls, doors, and room types marked in *(see Figure 34)*. The steps are independent from each other, so different algorithms could be used for each. For creating the connection graph he uses a set of graph node replacement rules, governed by constraints on minimum and maximum numbers of different room types *(see Figure 32)*, and for the room layout a half-arc radial layout algorithm with the front door as the center *(see Figure 33)*.

Jess Martin also introduces a separation of public and private rooms based on Christopher Alexanders architectural design pattern Intimacy Gradient [AIS77, page 610]. Public rooms (such as living rooms, hallways, kitchens) serve as connection points to doors to the outside, other public rooms, and private rooms (such as working rooms, libraries, and bedrooms). Private rooms only have one entrance to a public room, although they may sometimes connect to other private rooms.

This approach separates the building generation process into a set of simple steps, but it has a few shortcomings however. It treats determining number and functions of rooms, laying out the rooms, and building the walls between the rooms as separate sequential steps, and lacks any feedback from a later step to an earlier step. This reduces the possibility that constraints found during the layout or wall

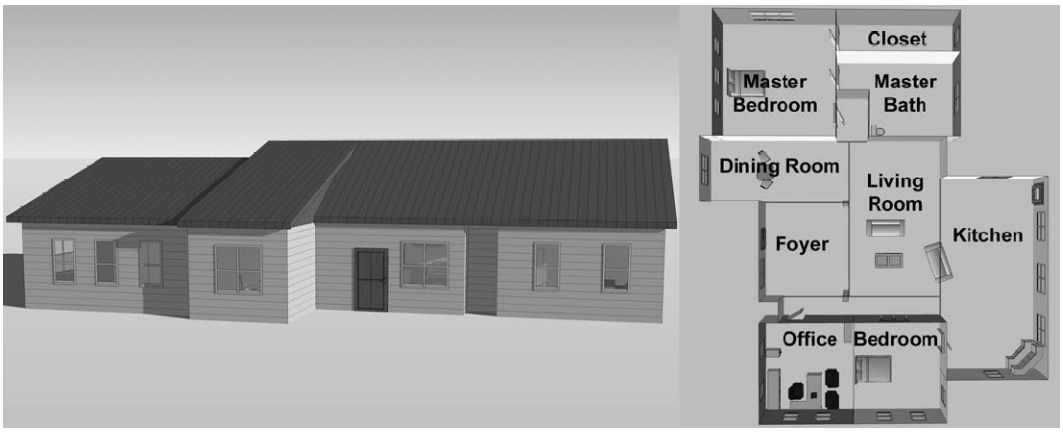

Figure 34: A rendering of a building floorplan created using a room connection graph. From [Mar06].

creation phase could be used to alter the amount and types of rooms in the house. As presented it also focuses on creating just one type of buildings (modern American suburban homes), with affluence (wealth) and building lot size as the only parameters.

4.1.3 Area subdivision buildings

A contrasting approach to using a room connection graph is to recursively divide the available space into rooms. At least in cities, buildings must be constrained to a building lot defined during city generation, so a subdivision approach is more natural than a growth process in this case.

Floors above the first can be created on top of the floorplan of the floor below it. A room normally resides on one floor only, except for stairwells and high halls.

Gardens and yards are treated as rooms also, making the algorithms simpler, and also following the Positive Outdoor Space and Outdoor Room patterns suggested by Christopher Alexander [AIS77] of designing outside space like rooms.

To find the right subdivisions for creating a good floor plan, a simulated annealing algorithm seems natural. It needs a set of change operations (such as split room and merge rooms), and a set of evaluation functions for calculating how well a given floorplan follows the architectural style parameters. Simulated an-

nealing uses a temperature variable that starts out as one and goes to zero over a number of iterations. Each iteration a number of possible modified versions of the floor plan are generated using the change operations, and scored using the evaluation functions according to how well they meet the style parameters. When the temperature parameter is high, one of these floor plans is selected randomly as the basis of the next iteration, but when the temperature cools down, the floorplans with higher score are preferred more, until at zero temperature the best floorplan is always selected. This ensures that the search algorithm has less chance of getting stuck in a local maximum, and will tend to find more optimal solutions.

The floorplan subdivision algorithm starts with a set of style parameters and an initial single room, covering the whole building lot and containing all the usages required of the building.

Evaluation functions measure different properties of a floorplan. The architectural style parameters specify a target value for each evaluation function. The score of a floorplan is calculated by measuring the difference between the evaluation function values and the target values, the closer they match the higher the score will be. The style parameters can also specify the relative importance of different evaluation functions, the more important ones get more influence on the final score. By using different style parameters, buildings with different styles can be generated.

The change operations and evaluation functions can be divided into a number of categories, based on which aspects of a house they are concerned with. Some categories are room shape, connectivity, and usages.

Room shape is modified by two basic operations, Split Room which splits a room along a dividing line into two new rooms, and Join Room which joins two adjacent rooms into a new room. Room shape is measured by the following evaluation functions: Room Size Adequateness measures how well the rooms can contain the furniture needed by their usages; Room Squareness measures how square or elongated the room is *(see Figures 35 and 36)*; Room Size Uniformity measures how similar the rooms in the building are in size; Wall Smoothness measures the amount of corners in the walls of the rooms.

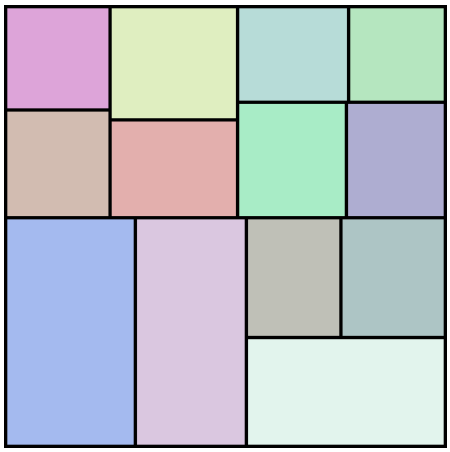

Figure 35: A floorplan generated with the room shape style set to prefer square rooms (using the modified room splitting function presented below). Rooms are randomly colored in order of addition.

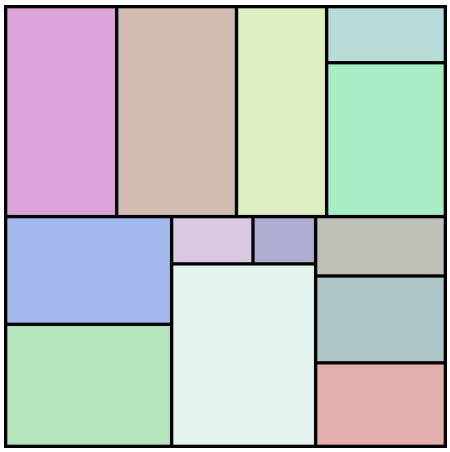

Figure 36: A floorplan generated using parameters that favor rooms whose side lengths are in the golden ratio (approximately 1:1.62), otherwise with the same algorithm as in Figure 35.

Connectivity is changed by the Add and Remove Door operations, and evaluated by Accessibility (whether the room is possible to reach from the front door by going through rooms and doors), as well as Privacy (the number of doors in a room) and Seclusion (the minimum number of rooms needed to pass through to reach the front door).

Usages of a building, such as eating, sleeping, or working, may also specify the preferred style of the room they are located in, using a similar set of style parameters as requirements, with a preferred value and weight for each evaluation function. An Usage Suitability evaluation function evaluates how well the requirements usages have are met by the rooms they are placed in. Usages can be moved around with a Move Usage operation, which moves some usages from a room to another randomly chosen room.

Additional operations and evaluation functions can be added as needed, some likely categories could be furniture arrangement, light and shadow, view, routes of traffic, symmetry, ceiling height, and roof styles. Formalization could be attempted for many of the architectural patterns presented by Christopher Alexander [AIS77] by creating suitable evaluation functions and change operations.

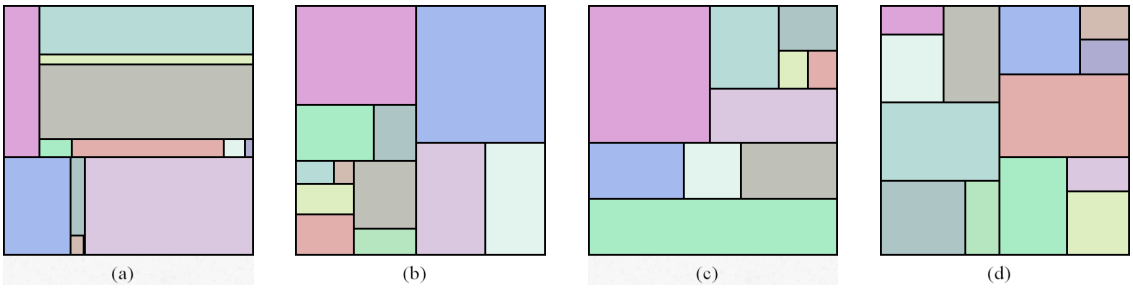

Figure 37: The floorplan in (a) was obtained by running the original subdivision algorithm a dozen iterations with no evaluation functions. Many rooms are much too narrow or small. For the second floorplan (b) a squareness evaluation function was added, with a style parameter that encouraged more square rooms. In the third floorplan (c) two additional evaluation functions were added to severely discourage too small rooms, and to encourage uniform room size. As can be seen, there are still some small and elongated rooms present. In the final floorplan (d) the search space was altered to contain less too thin or too small rooms by modifying the split room function to pick one of the largest rooms for splitting each iteration, and to split it across the long side at approximately the middle.

The advantage of this algorithm is that building style and possible usages are separated from the algorithm, and instead given as parameters. This should allow the algorithm to generate a more wide variety of different kinds of buildings. The algorithm is also iteratively refining, which means that it can be interrupted at any time and still produce a house containing all the specified usages. A disadvantage is that iterative level of detail refining is not possible; change operations may modify the outer shape of the building at any iteration, so all required iterations have to be run before the outer appearance of the building can be generated. This causes problems for city rendering where thousands of buildings can be visible at the same time.

In practice this algorithm does not perform so well because the search space is just too large, the simulated annealing search is not able to find very good buildings in it with few enough iterations. I implemented the basic algorithm, as well as some of the shape evaluation functions and the split room operation. Even with evaluation functions that encourage square rooms and uniformly sized rooms, and discourage thin rooms or too small rooms, the algorithm produces results such as Figure 37 (c). Part of the reason is that without a join rooms function implemented, the algorithm has no way to backtrack once it has introduced some bad room, other than dividing it into smaller rooms.

Another reason is that the split rooms function chooses the room to split, the split axis, and the split location completely randomly. To generate a floorplan with roughly square rooms with uniform size, the split operation should select one of the biggest rooms to split, should split it across the longer side, and should split the room near the center. Using a split function like this, the floorplan in Figure 37 (d) was obtained. What has been done is that part of the style has been moved into the change operation. This narrows down the search space, and allows a better floorplan to be found with fewer iterations, as the randomly generated ones are already of better quality.

This leads to a another algorithm, which first evaluates the building using different evaluation functions, and then picks one of the change operations that can best address the found problems (differences between measured floorplan properties and the style targets). The change operation may be parametrized by the architectural style, for example if the preferred style is to have rooms of uniform size the split operation can split rooms at the middle, but if the architectural style requires some variation in the room size it can split the room into two different sized rooms. The way this algorithm operates resembles refactoring source code – find problems in the building, then apply a modification that preserves usages of the building but improves its adherence to the desired architectural style. This new building refactoring algorithm has a much more focused search space, and should lead to better buildings faster. It also maintains the advantages of being parameterizable, and being iteratively refining, allowing as many iterations as desired. It also generates the room shapes, connections, and usages at the same time, allowing constraints in one of these areas to be solved by changes in another area. It remains untested other than with simple room shape evaluation and change functions however.

The performance of the implemented building generation algorithm was fairly good for the floorplans shown, with 12 iterations and 100 different candidates generated at each iteration the algorithm used 180 milliseconds on an IBM ThinkPad T43, running at 782 MHz and with 2GB memory. However, because only the split function is implemented, the number of iterations was very low. With other operations, such as the join operation, as well as connectivity and usage operations and other possible operations implemented, the number of needed iterations will be much higher, to make sure that all of the operations can be used to find a good floorplan. To simulate this situation, I measured the performance with different

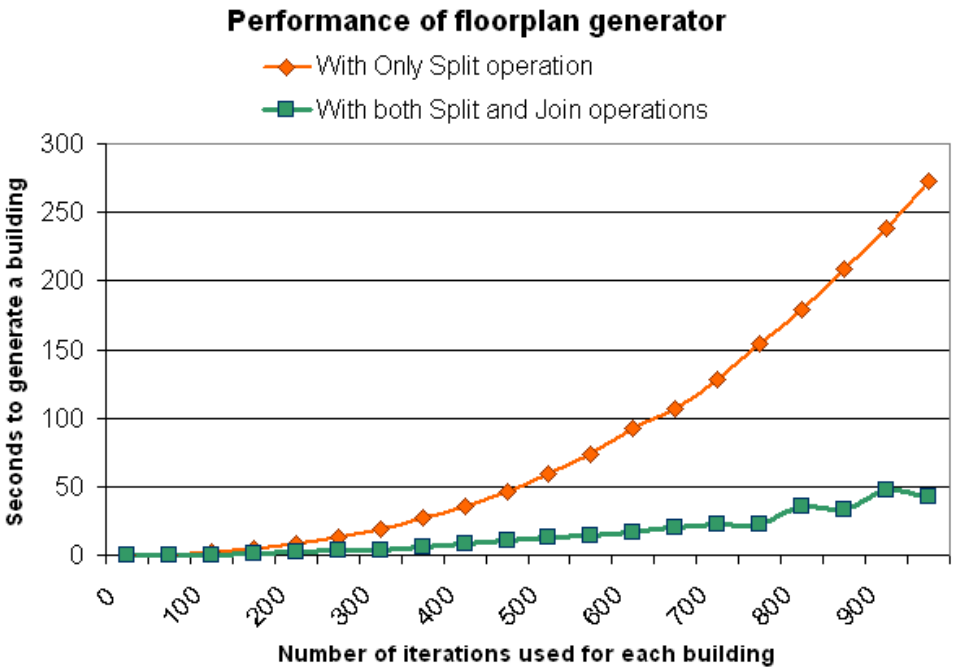

Figure 38: Performance of the area subdivision based floorplan generation algorithm. In the case of using only the split room operation the execution time rises exponentially with the number of iterations used to generate a building, as each split creates a new room. With a join rooms operation added, the number of rooms stays more constant, and the time cost rises linearly with the number of iterations. The experiment was performed on a IBM ThinkPad T43, running at 1.86 Ghz with 2GB memory.

numbers of iterations. When using only split operations, the number of rooms will be roughly equal to the number of iterations used to generate the house. For high numbers of iterations this will lead both to unrealistic houses, as well as a significant slowdown, as the algorithm slows down when the number of rooms increases. To better simulate the performance of the algorithm in a typical case, I implemented a simple join rooms function, and evaluated the floorplans with a certain minimum room size as a scoring function. In this case, the performance of the algorithm was roughly linear to the number of iterations, and the number of rooms stayed around 20-30 *(see Figure 38)*.

Figure 39: A typical view of a a tile based roguelike game with ASCII graphics (symbol explanations added to the right). This type of game often uses randomly generated maps to increase replay value. From [Rog06].

4.1.4 Buildings on a grid

A regular grid is a popular base for buildings. It allows pre-modeled building blocks to be used for walls, doors, floor sections, roofs, furniture, and so on. It can also simplify the building generation, by reducing the search space for floorplans, and simplifying furniture placement. Grids used in this way usually range from around a square meter (enough space for a person to stand) to a few dozen square meters (enough for small rooms or sections of rooms).

Grid based maps have been used for a long time in computer games. Some of these games use various random maze or cave system generators *(see Figure 39)*, but their applicability for normal house generation is limited.

There can also be bigger pre-designed sections, consisting of an area of several tiles shaped into rooms or parts of a building. The UFO Enemy Unknown [Mic93] game used this approach to generate random bases and outdoor landscapes *(see Figure 40 and 41)*. Each section has standardized edges, so that it fits together seamlessly with other sections placed next to it. Base sections will have doors to the four cardinal directions, city sections will have streets crossing the section edge, and outdoor sections just have level ground with grass growing along the edges. By randomly combining different sections together a larger area can be generated. The advantage of this approach is simplicity. The disadvantage

Figure 40: An overview map of a base, showing the different sections it is composed of. Screenshot from the game UFO [Mic93].

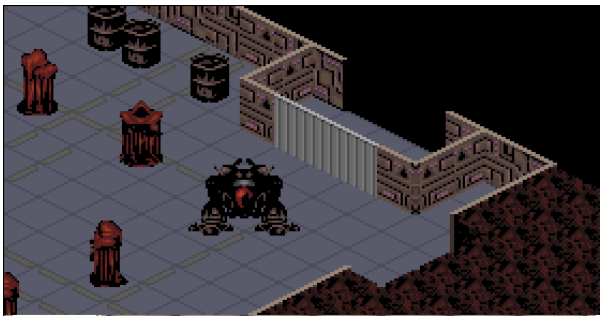

Figure 41: A tactical view from the same base, showing one of the hangars under alien invasion. Note the isometric tile graphics used to render the floorplan. Screenshot from the game UFO [Mic93].

is repeatability, the user starts to recognize the different pre-modeled sections. Parametrization is also lacking, as the sections are fixed.

The area subdivision building generation algorithm presented above can be applied to grid based floor plans also, with some modification to make the room shape altering operations adhere to the grid.

A limitation of tile based buildings is also that they require a regular grid to work on, so building lots in a city need to align with or contain that grid. This may constrain the types of city generation algorithms that can be used, and produce somewhat regular buildings, with walls primarily along grid edges. A way to partially overcome this could be to use house-specific grids, which may be placed in any orientation. This way houses are aligned to a square grid, but the city generation algorithm is free to place building lots in any way, as long as they are rectangular with side lengths that are multiples of the grid size.

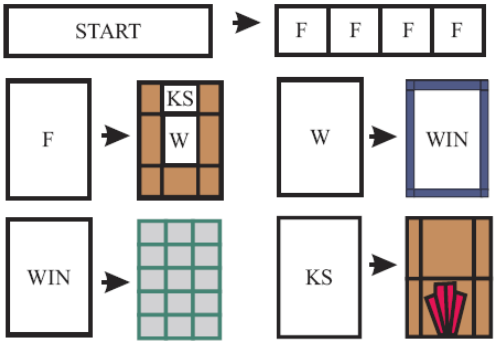

Figure 42: A set of rules for the split grammar. White areas are non-terminal nodes and colored areas are final nodes. From [WWSR03].

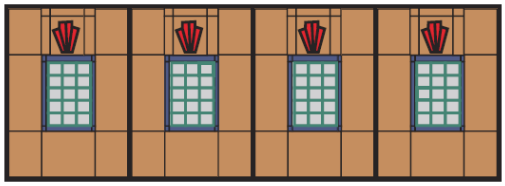

Figure 43: The result of applying the rules to the left. From [WWSR03].

4.1.5 Facades

The outside appearance of buildings has been well researched, mostly in combination with algorithms for generating house shapes for inclusion in generated or modeled cities. In [WWSR03] Wonka et. al. introduce a split grammar for generating building facades containing horizontal and vertical regularly repeating patterns *(see Figures 42, 43, and 44)*. If floorplans exist prior to creating the outside appearance though, the facade generation algorithms will need to get data such as window and door placement as inputs, instead of generating them with grammar rules.

4.1.6 Roofs

Regardless of how the floorplan for a building is generated, the roof for it can be calculated based on a polygon outlining the area to be roofed. Flat roofs are more or less trivial to create. Sloping roofs can be generated by finding the skeleton (center-lines) of the polygon defining the roof outline, using the skeleton as the top edges of the roof, and letting the sides slope down or form gables [LD03] *(see Figure 45)*.

Figure 44: A house facade generated with split grammar. From [WWSR03].

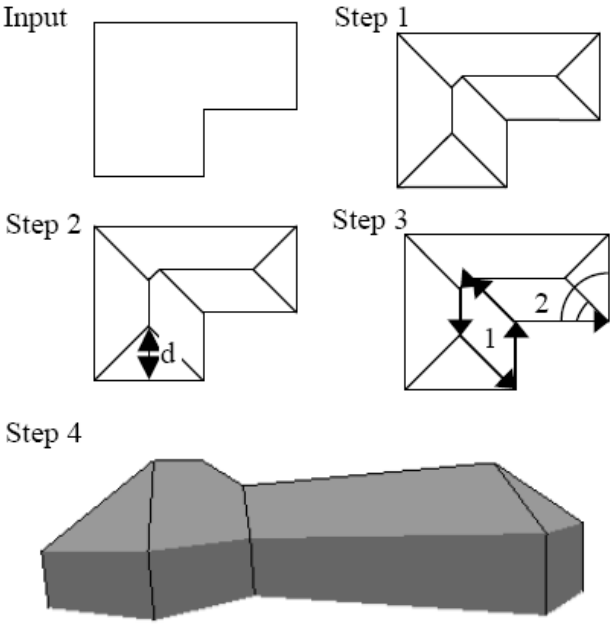

Figure 45: Generating a roof. First a skeleton is created for the given floorplan outline, then the polygons making up the roof are extracted, and finally the roof is raised along the skeleton to a height that is related to the distance from the skeleton to the edge of the roof. From [LD03].

4.2 Rendering building interiors

Building interiors can be quite complex, but the enclosed space inside rooms provides a convenient way to reduce the complexity when rendering buildings. When viewing a room from inside, there is only a need to render the interior of that room, and any parts of adjacent rooms or the outside that is visible through openings such as doors and windows. There are two prominent techniques for optimizing this calculation; using Binary Space Partitioning (BSP) algorithms, or using a Portal based rendering system. The Brute Force approach of rendering all of the content of the building is usable if the interiors of buildings are very simple or non-existent.

4.2.1 Binary space partitioning

In Binary Space Partitioning the building (or more generally 3D model) is analyzed in advance, and a binary partitioning structure is created for it [FKN80]. When rendering a view of the model, the BSP structure can be used to quickly determine which polygons are visible from the current viewpoint, and only render them.

BSP based techniques have the advantages of fast performance when rendering, but the disadvantages of a slow pre-computation step that makes fast dynamic building generation problematic.

4.2.2 Portal based rendering

Portal based rendering systems divide a building into a set of rooms, where each room can have portals to other rooms as parts of their walls [Dat05]. When rendering a view inside a room, the current room is rendered, as well as visible parts of other rooms seen through the portals.

Portal based techniques have the advantage of allowing easy and fast creation of rooms and portals, so they seem more suitable for on-the-fly generated buildings.

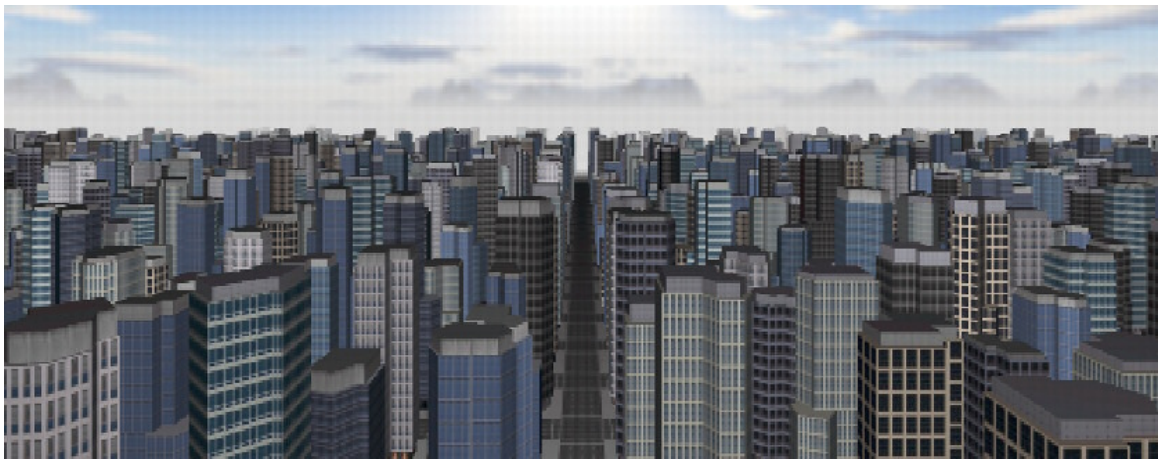

Figure 46: A pseudo infinite, homogeneous city consisting of skyscrapers. The lack of macro scale variation makes this city appear unrealistic. From [GPSL03].

5 Cities

A city is more than just several buildings placed together. A city consists of infrastructure – roads – that connect all the buildings in the city with each other as well as the rest of the world. The roads form an interconnected graph of lines of varying width, depending on the amount of traffic the road is designed to carry. The areas between the roads are blocks, which can be divided up into lots. Buildings are placed on lots, but lots can also be used for other things, such as market squares and parks.

Just generating an endless maze of roads and buildings such as in [GPSL03], does not produce a very realistic city *(see Figure 46)*. The city generation algorithm presented by Y. Parish and P. Müller [PM01] introduces local variation in the city, by utilizing externally provided elevation and population density maps for determining road patterns and building types *(see Figures 47 and 46)*. Elevation and population density maps could be procedurally generated instead of manually created, enabling automatic city generation. However, their algorithm does not have any concept of differentiated districts or local centers.

In architecture literature a city is usually treated as a hierarchical structure that can be divided into successively smaller areas, from the city as a whole to individual buildings. Christopher Alexander presents the architectural city design patterns Community of 6 million, Community of 7000, Neighborhood, House

Figure 47: A road network (on the right) generated automatically from water, elevation and population density maps (on the left). From [PM01].

cluster, and Building [AIS77]. Alexander suggests that each Neighborhood and Community of 7000 should have their own typical profile of residents, as well as a local centers and well defined boundaries.

5.1 Pattern based city generation

To generate a hierarchy of areas with distinct boundaries, a natural approach is applying divide and conquer – first the city is divided into large suburbs, they are divided into smaller local neighborhoods, the neighborhoods are divided into blocks and the blocks into building lots. Roads are placed along the edges of the subdivisions – larger roads between larger areas, and smaller roads between smaller areas. In addition, roads could be created directly to the centers of areas to provide access to services located there. If there are any natural boundaries such as rivers, they are used as dividers also. Each top level area is then assigned an architectural style based on the properties of the city, and sub areas are assigned modified versions of the architectural styles of its parent area. Landmarks are placed near the center of areas, with bigger areas getting more prominent landmarks.

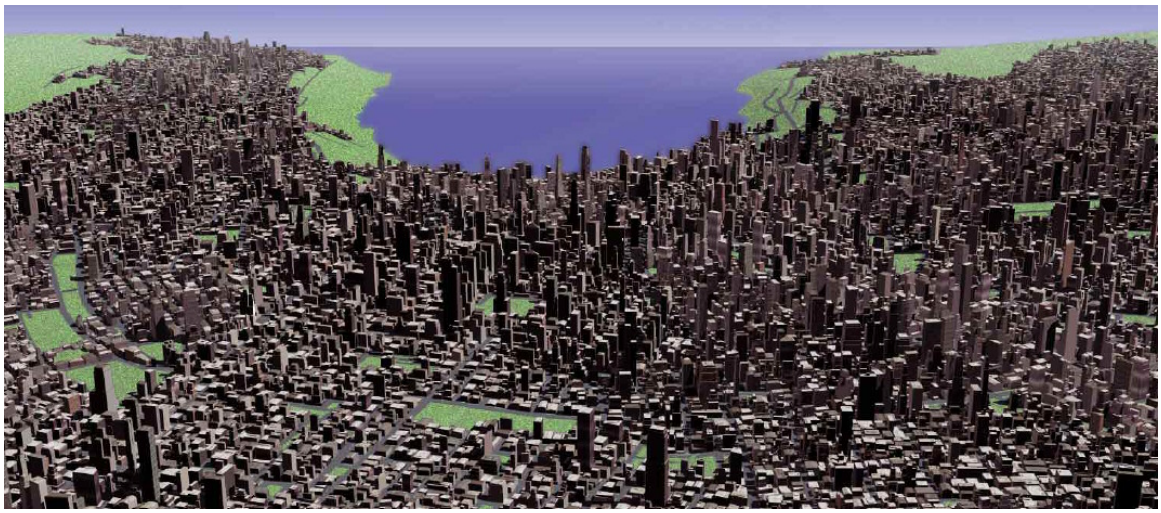

Figure 48: A virtual city modeled using the data from Figure 47. This city has variation and appears more natural than the one in Figure 46.

The style of an area can affect things such as the density of contained sub areas both near the center and at the edges of the area, the pattern of the road network (regular on a grid, irregular with varying road angles), the width and type of roads, the population density and average height of buildings, the architectural styles of the buildings, and so on. The type of population inhabiting each area is also a natural part of its style. Each city contains a mix of different subcultures. As the city is subdivided into areas, the subcultures are distributed to the different areas, with a tendency to concentrate a specific kind of subculture together in a specific area. Natural examples of this from real cities are the Chinatowns of bigger international cities, but also university campuses, business centers, and industrial parks have an own kind of subculture. The subculture can affect the various style parameters of the area.

The architectural design patterns presented by Alexander were mostly collected by examining existing urban and architectural patterns that worked well, and identifying the features that made them work well. To model well designed cities we could build in some of these urban design patterns in the city generation algorithm.

The City-Country Fingers pattern strives to give residents of the city easier access to the countryside. It does so by placing areas at the edge of the city in long parallel fingers that extend into the countryside, with a road at the middle of the finger.

This pattern can be adapted to the divide and conquer city generation approach for example by enlarging borders between areas that are at the edge of the city and placing countryside into those borders. This way the countryside helps form a natural boundary between different areas as well. Another approach would be to replace some areas at the edge of the city by countryside. The tendency to form city-country fingers can be parameterized, and may vary between different areas.

Local Services is a rather logical pattern, it basically states that there should be local services inside each area (usually near the center), for each service that there is enough demand for in the area. Less requested services would then be placed at the centers of bigger areas. The types of services needed in a given area depends on what kinds of people from different subcultures that live or work there.

Alexander also proposes locating heavier industry areas near the borders of larger areas, where they will have easy access to the major transportation routes, and will not disturb the residential areas further inside the area.

These pattern gives us a way to determine what kinds of shops, industry, public buildings, market squares, and parks should be present in an area, and where to locate them inside the area.

The divide and conquer approach does not necessarily generate a very realistic road network, and the change of the styles at the edges of major areas can be somewhat abrupt, but it seems to hold some promise for creating a varying and interesting city. It might be possible to merge it with existing road network algorithms, such as the one presented by Y. Parish and P. Müller [PM01].

5.2 Rendering cities

A large city can contain a huge number of buildings. Rendering them all with a brute force approach might cause the speed of the visualization to drop intolerably low. Fortunately, when walking at street level in a city, the nearby buildings typically hide most other buildings behind them (unlike a forest where tree trunks are relatively narrow and the foliage is partially transparent). Precomputing this occlusion is explored in [WWS00], although this will not work di-

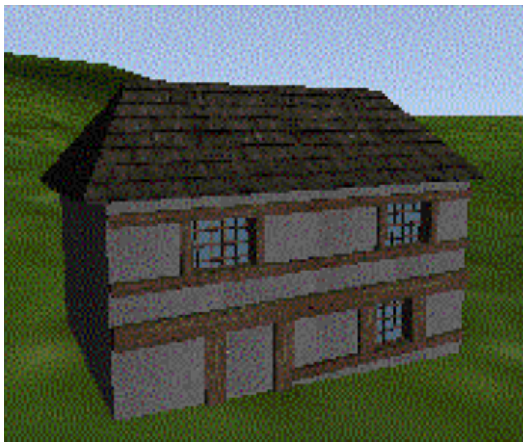

Figure 49: A building with surface details such as windows rendered with geometry. From [WWAD01].

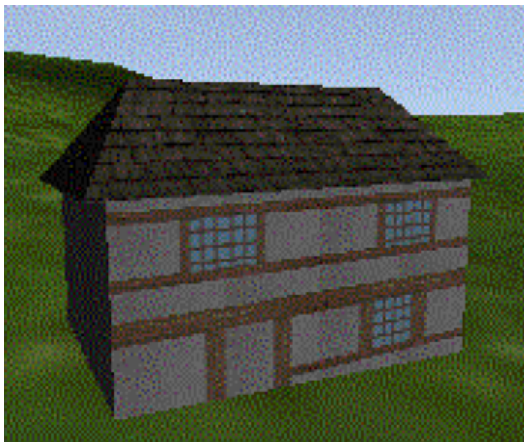

Figure 50: The same building with the surface details rendered to the texture, to reduce the amount of triangles to draw. From [WWAD01].

rectly for on-the-fly generated cities that do not allow expensive pre-computation steps.

Building rendering can also use multiple levels of detail, for example so that smaller details are rendered to the building texture when seen from far away, but rendered as normal geometry from close up *(see Figures 49 and 50)*. Interiors seen through windows and open doors need only be rendered when they are close to the viewer (during the day the outdoor light level is much higher than the indoor light level, so windows and open doors can be rendered with solid black when seen from a distance[1]). Distant buildings can be rendered to billboards and updated only when the viewer has moved enough to change the viewing angle to them noticeably *(see Figure 51)* [MTF03a].

[1]During the night they could be approximated with yellow if the room is illuminated, although this might look a bit less realistic.

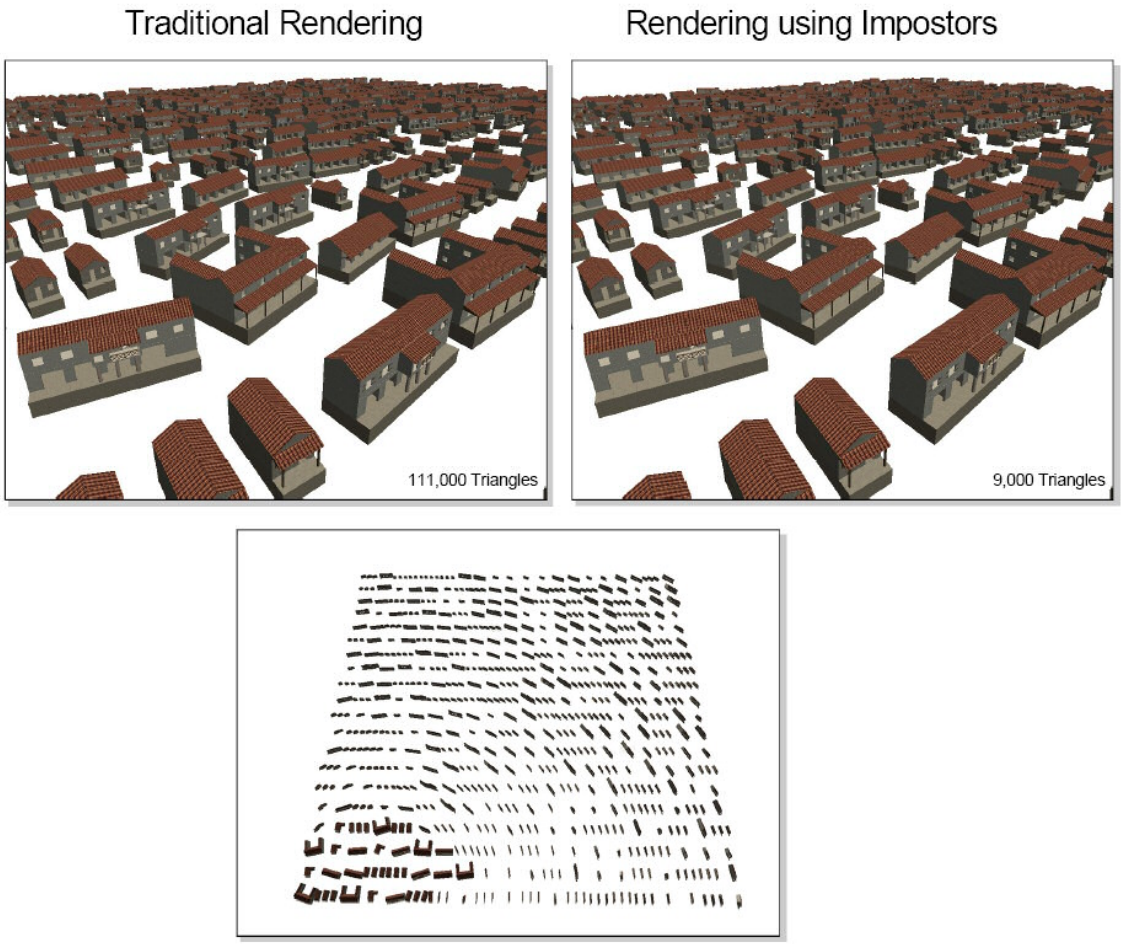

Figure 51: Using impostors to speed up city rendering: In the upper left picture real geometry is used for the buildings; in the upper right picture more distant buildings have been rendered to a billboard. In the lower picture that shows the upper right scene from above, the real geometry and the billboard impostors are clearly visible. From [MTF03a].

6 The sky

The backdrop for a landscape is the sky. The sky contains clouds, the sky background color (the atmosphere illuminated by the sun), and celestial bodies such as the sun, moon, and stars. The clouds form, change, and dissipate over time, the sky changes color depending on the time of the day, and celestial bodies move with the Earth's rotation and the time of the year.

6.1 Clouds

6.1.1 Cloud types

There are many types of clouds, but the most common types are the various convection clouds (cumulus clouds), uniform layer clouds (stratus clouds), thunderclouds (cumulonimbus), and wispy high altitude clouds (cirrus clouds) [Clo06]. Stratus and cumulus clouds can form at low, medium or high altitudes (medium to high altitude cumulus clouds form wavy or dune-like patterns). Cumulonimbus stretches from low altitudes up to high altitudes in an anvil shape. Cirrus clouds occur at high altitudes only.

Convection clouds (cumulus clouds) form when moist air is heated at the ground level and rises up, while cold air sinks down around it. This process is called convection and can be observed when any fluid in a gravity field is heated at the bottom (e.g. when boiling water). The fluid forms into regular convection cells where warm fluid rises at the center and cooled fluid flows down along the edges of the cells. Convection clouds are thus usually relatively circular, and evenly placed over the sky. They have flat bottoms, and cauliflower shaped tops. Winds can stretch the convection cells and clouds into more oval shapes *(see Figure 52)*. Typically convection clouds start forming in the morning when the sun starts heating the ground, grow during the day, and eventually loose their typical shape and merge together in larger congestion clouds.

Stratus and nimbostratus clouds form when moist air is cooled down below its condensation point and forms into clouds. This usually happens at weather

Figure 52: Photo of low altitude cumulus clouds. From [Clo06].

Figure 53: A nimbostratus cloud. This is a mostly formless dark gray stratus cloud that usually brings rain. From [Clo06].

Figure 54: High altitude cirrus clouds. From [Clo06].

fronts, but can also happen when wind pushes warm air up a mountain slope. These clouds do not have the cell structure typical for convection clouds, instead they are mostly uniform with diffuse edges, and often spread out over large areas *(see Figure 53)*.

Cirrus clouds are high altitude clouds consisting of ice particles. The characteristic hairlike filaments of cirrus clouds are rains of heavier ice crystals, often carried by high altitude winds to form long wisps *(see Figure 54)*.

6.1.2 Cloud rendering

A very simple way to render a sky with clouds is to pre-render the clouds on six square textures, arranged in a box around the camera (called a skybox). The box is moved but not rotated with the camera. If correct perspective is used when rendering the textures for the six sides of the skybox, the result is the appearance of a continuous sky (and distant ground) around the camera. This approach works well if the sky does not have to change over time or depending on the camera position. A simple animation effect can be achieved by cross blending-between two different sets of sky textures over time, but this does not work that well for fast moving clouds if generating the view of the sky for the skybox takes too long.

A sky plane or dome is often used for moving clouds. In this approach the sky is covered with one or more textured planes (or half spheres in the case of domes),

Figure 55: Clouds from [HL01] rendered as billboard impostors.

each of which is covered with a cloud texture (clouds painted on a transparent background). By changing the texture coordinates of the planes, the textures can be made to scroll across the sky. By using different scales and moving the textures at slightly different speeds a parallax like effect can be achieved, simulating slower moving high clouds and faster moving lower clouds. The drawback is that the textures are still two dimensional, so the clouds will look quite thin. Pixel shader effects like normal mapping might be used on more recent graphics cards to create an illusion of thickness. For low hanging uniform cloud cover this is not a problem though.

To model individual clouds with volume, a particle system approach can be used, where each particle is a billboard with a soft edged small cloud part on it. For convection clouds, the particles are arranged to form a cloud with a flat bottom, and a cauliflower shaped top. The particles in the cloud could be moved slowly over time, to simulate the shape-changing behavior of clouds. The problem with this approach is performance. A single cloud with fifty or so billboard particles renders fast, but if we want to fill the sky from horizon to horizon with these kinds of clouds, we run out of rendering power quickly. A solution is to render all the particles making up a cloud to a single impostor billboard *(see Figure 55)*. Distant clouds can be simply rendered to a skybox texture, and only updated when lighting conditions change significantly, or the camera has moved significantly. Clouds can also use a level of detail approach for the particles in them (somewhat similar to the foliage of plants), when approaching a cloud each single larger par-

Figure 56: Flying through clouds above the sea. Note the two different layers of clouds, the low altitude cumulus clouds are rendered as billboards, while the higher altostratus clouds are rendered to a sky dome or skybox. From [HL01].

ticle can be replaced by multiple smaller particles. This way far away clouds can be represented with less particles and detail, and close by clouds with more particles and detail, further optimizing performance. Some of these techniques have been used in [HL01] to render realistic clouds for flight simulators *(see Figure 56)*.

The different cloud types lends themselves naturally to one of the three cloud rendering algorithms presented above. Low cumulus clouds and large cumulonimbus clouds can be rendered as particle systems, low stratus clouds and medium level altostratus and altocumulus can be rendered to a plane or dome and moved across the sky, and cirrus and other high altitude clouds can be rendered directly to a skybox and kept stationary. This way resources are used optimally, and different types of clouds can be rendered at the same time, resulting in a more natural looking sky.

Correct lighting is crucial in making clouds appear natural. When light enters a cloud, some of it is reflected back from the water droplets, while some goes through the water droplets and gets scattered in the forward direction, some part gets absorbed, and some part does not hit any water droplets (if the cloud is thin). This results in a somewhat complicated and computationally demanding lighting calculation *(see Figure 57)* [HL01].

Self shadowing among clouds is another way to improve the realism of especially sunsets and sunrises, when there are many shadows cast by clouds on other clouds. Shadows for thin clouds at medium or high altitudes are so light that they do not need to be applied, but low altitude cumulus clouds cast shadows both on themselves, other clouds, and the ground.

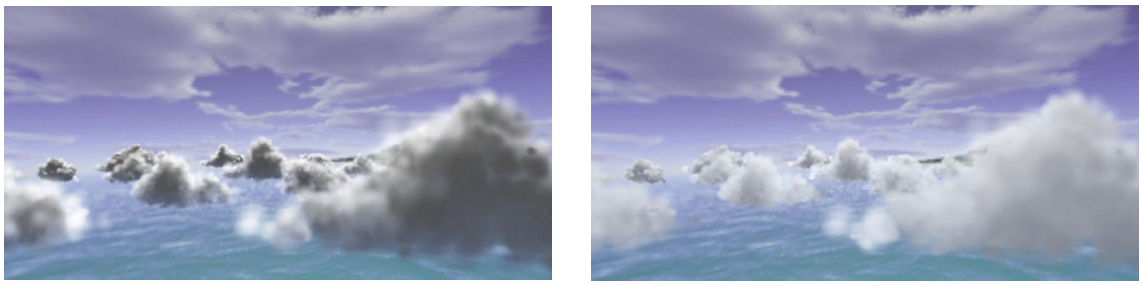

Figure 57: Light scattering in clouds. In the left image only single scattering is used, in which a light beam from the sun to the camera hits one water droplet on the way through the cloud. In the right picture multiple scattering is used, where each light beam from the sun to the camera reflects of several water droplets in the cloud. The result is that the cloud appears more bright (like it does in reality). Multiple scattering is more resource intensive to calculate, although some approximations can be used. From [HL01].

6.2 Weather and climate

Just rendering a lot of different kinds of clouds on the sky can produce a chaotic impression. To achieve a feeling of realism, there should be some system guiding what kind of clouds appear together and after each other. Unlike the supercomputers trying to predict weather from current measurements and historical data, we only need to generate weather that has some appearance of realism, it does not need to exactly match the way the real weather works.

6.2.1 Modeling weather

The main events changing the weather at a given location could be modeled as warm and cold fronts moving past, and changes depending on the time of the day. Other than these the weather would be more or less constant, except for slow shifts as the seasons change. It is a simplistic model, but could be enough for providing a sufficiently natural weather for a landscape.

What effects a weather front or daily cycle has on the local clouds, wind, and temperature depends on the local climate. The climate could specify average temperature, humidity, dominating wind directions and strengths, and instability of the weather (how often cold or warm fronts arrive, and how strong they are). The

climate depends on the distance from the equator, the proximity of large bodies of water, the ground type and amount of vegetation, and how mountainous the area is. The climate can also change depending on the time of the year. The climate could be calculated at a rough level from the world map, or alternatively be specified in the ecotopes to give world designers better control of it.

An incoming front changes the temperature, and possibly wind direction and humidity. A warm front arriving will rise above the local cooler air, first seen as cirrus clouds, later visible as medium altitude stratus clouds that grow denser over time, and can result in rain depending on the difference in temperature and the humidity of the warm air. A cold front pushes the local warmer air upwards more abruptly, causing heavy rains and possibly thunder if the temperature difference is large enough and the warm air is humid enough.

Daily changes in the weather include temperature changes (the sun warms the ground during the day at some rate specific to the type of the ground), winds resulting from different rates of warming at different places, and convection clouds building up from humid air that is warmed by the ground and rises up to form clouds. An example of winds resulting from different rates of warming and cooling are the morning and evening breezes in coastal areas from and to the sea, as the sea and land warm and cool at different rates. In mountainous areas similar morning and evening wind effects can be observed. An example of convection cloud forming in temperate climate zones during summer are small cumulus humilis clouds that form in the morning, grow during the day, and melt together in a larger irregular cloud cover during the afternoon. In equatorial climate zones the effect is more radical; hot and humid air from over the sea rises up during the morning and builds large rainclouds, resulting in heavy rains during the afternoon.

Clouds also form because of the terrain topology and winds that push warm and humid air upwards along a mountain slope.

6.2.2 Rendering weather

A simple model for wind is to have a fixed wind field for the whole (visible) landscape, specifying the direction and speed of the wind. To give some variation and realism to cloud movements, the wind direction and speed could be different for the ground level and clouds at different altitudes. On the ground the wind can be visualized as vegetation that bends in the wind (using a spring system such as in [PC01]), and as light objects such as leaves or snow being driven by the wind. The wind can have some variation value that determines how often the direction and speed changes (while still averaging to a specified direction and strength). The drawback to this simple wind model is that different wind directions and strengths at the same altitude cannot be observed at the same time, but this might not be very noticeable in practice.

Fronts could be modeled as segmented lines moving across the landscape with the wind, causing formation of various clouds at different positions in front and behind them. As a front passes by overhead, the wind strength increases at ground level and the wind becomes more turbulent (both the strength and direction randomly varies more with time). Clouds are generated for the visible area by fronts, by the daily cloud formation cycle, and by topology effects such as a wind pushing moist air up a mountain slope. Clouds could be modeled as a cloud node with a location, radius, altitude, type (cumulus, stratus, cirrus, cumulonimbus), and be parametrized with thickness, amount of rain, and coherence of the shape (for cumulus clouds). Low altitude clouds would be rendered as billboards, and medium and high altitude clouds as images added to a sky plane or dome texture. The clouds could move with the wind, and have their parameters change over their lifetime according to some pattern determined when they were formed.

Rain (or snow) is usually rendered as a particle system, with falling droplets or lines. In addition, rain decreases the saturation and blurs the outlines of distant objects. This can be achieved with a gray fog or pixel shaders. Rain seen from afar could be rendered simply as a textured half transparent polygon under the raincloud.

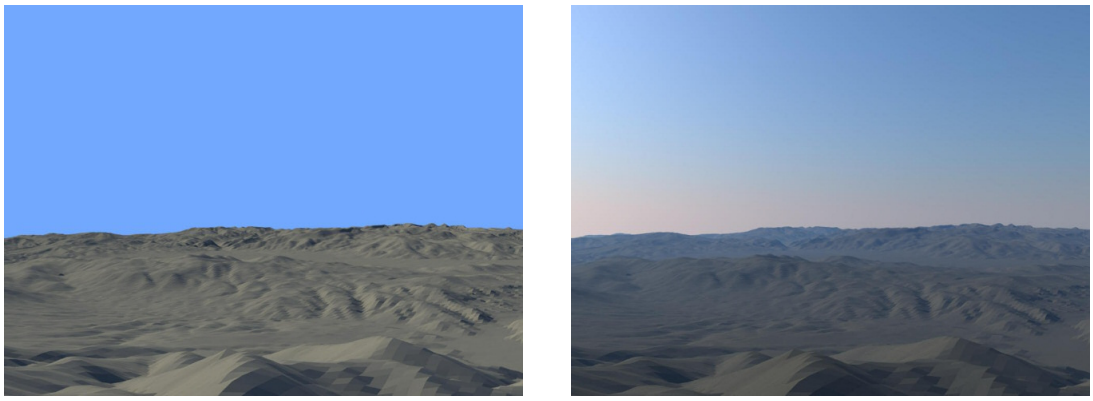

Figure 58: A sky and ground rendered without (to the left) and with (to the right) the analytic model for daylight rendering developed by A. Preetham et. al. From [PSS99]

6.3 Atmosphere rendering

The atmosphere seems like it should be easy to render, just a blend from a dark blue color at zenith to a lighter blue at the horizon. This is a good approximation, but if you strive for higher realism it gets more complicated. First there are concentrations of dust, varying with the height and reddish in color. Then water vapor, light blue in color and with decreasing concentration at increasing height. And the gas mix making up the atmosphere, blue in color. These can be modeled as density fields, with some specific density at ground level, and a density decreasing non-linearly with the height. Sunlight scatters in different ways when passing through the atmosphere, dust, and water vapor, resulting in a slight halo around the sun, and also changes to other parts of the sky. In addition to determining the color of the sky, the atmosphere also affects the appearance of the ground – distant mountains will typically be bluer and less saturated. This is called aerial perspective. A. Preetham et. al. developed an algorithm for efficiently calculating the appearance of both the sky and the aerial perspective based on these effects [PSS99] *(see Figure 58)*.

6.4 Celestial bodies

The sun can be rendered as a billboard texture. Lens flares and an over exposition effect can be used to give an impression of its brightness. The over exposition effect simply fades the whole screen towards white when the camera is pointed

almost directly towards the sun.

The moon can be rendered as a billboard texture, or a lit sphere if phases of the moon should be simulated.

Stars can be faded in at night, starting with the brightest ones. A star database could be used if an earth like night sky is desired, otherwise stars could be generated randomly. However, randomly generated stars should not be uniformly placed, because stars are hierarchically clustered in reality, and a uniform distribution looks very unnatural. The number of bright (nearby) stars is also much smaller than the number of less bright (more distant) stars. Perlin turbulence could perhaps be used as a density field for the stars across the night sky.

7 Conclusions and future work

7.1 Developing the proposed algorithms

I have presented a number of existing algorithms and some ideas for new algorithms for generating and rendering landscapes. The next step is to implement an actual landscape generation and rendering engine using some of these algorithms. Figure 59 shows a screenshot from my current landscape engine work in progress. It will be available as the SkyCastle project on Sourceforge [Häg06]. However, most of the new ideas I presented need some further development first.

Cumulative uneven addition can be used to produce a fractal terrain, by iteratively adding a noise map with decreasing amplitude to a height map. This can be further developed by instead creating the height map by adding together several layers of random noise maps with different scales and amplitudes. In this case the result and algorithm will be similar to Perlin noise, at a fraction of the cost (as huge chunks of noise is precomputed in the noise maps), although the algorithm is not very similar to the original cumulative deposit algorithm anymore. What kind of results can be achieved by using other addition maps than random noise is still to be investigated. The hypothesis that eroded hill slope addition maps might produce a hilly, eroded looking terrain is somewhat questionable, although it could be worth testing.

Generating rivers and lakes to a landscape based on drainage basins and the connections between them seems to hold some promise. Using drainage basins for large scale erosion and ground shaping might also be worth looking into.

Height modifying textures added to the ground texture (e.g. for creating raised roads with ditches) is definitely a promising concept, and should be implemented and demonstrated.

The lighting of plants was not investigated. In practice the lighting of a forest scenery is one of the factors that brings most realism to it. Computing how foliage shadows itself and other foliage is complicated, but some approximations might be possible to use. The color of leaves also change based on the reflection angle of the light, and on how much light is coming through the leaves. O. Franzke and

O. Deussen have developed a model for realistically rendering the appearance of plant leaves in different lighting conditions [FD03]. Lighting for billboard foliage can be (and has been) approximated by pre-rendering the foliage impostor with different directions of incoming light, and blending between the two closest matching impostors when the foliage is rendered in a given lighting.

The animation of plants was not discussed either, if they move in the wind it adds some realism to a landscape, but makes it harder to use impostors and to duplicate static geometry using vertex buffers.

The building generation algorithm that subdivides a building lot into rooms was started, but not finished. To create proper floorplans doors and room usages need to be added also. The performance and quality of the simulated annealing algorithm is questionable when using many different change operations, and the iterative building refactoring algorithm needs to be tested in practice. How to define the boundary between the house and the garden / yard might also be a bit hard, although one solution could be to make the rooms touching the building lot edges into garden 'rooms' and penalizing houses with too irregular outer wall shapes.

Many of Christopher Alexanders design patterns could give inspiration for both building and city generation. However, they might not always be very easy to formalize, and there could be other fundamental aspects of houses that are not captured in those patterns.

The suggested weather simulation algorithm could also be implemented and tested in practice.

7.2 Landscape editor

To allow landscape designers to efficiently design landscapes an easy to use but powerful editor application is critical. It should allow randomly generating a landscape, and changing it by painting in different ecotypes at a large scale, or going in and doing detailed changes such as manually placing trees or designing buildings. Existing greyscale and color images should be possible to use for defining the elevation or ecotype distribution for some area of the landscape. It should have separate editors for building styles, cities, plant species, and ecotypes, and allow import and export of these, so that they could be reused in different landscapes. The landscape should be saveable in some expandable xml format (perhaps contained in a zipped package that includes any image files that were used) that the landscape rendering engine could read, and that third party tools also could easily read, modify, or generate.

7.3 Adding life

After the landscape engine and editor are completed, I would like to focus on adding animal and intelligent life to the landscape.

Population biology uses the concept of habitat, and simulates the population of different animals in it by calculating fertility, mortality from starvation, age and predation, and migration to and from other habitats. This could be a good model for keeping track of animals in the landscape. A simpler model could just be to have a certain density of different kinds of animals in the landscape depending on the ecotype. Either way, once we know what animals are present in an area, we can randomly generate animals around a viewer that walks through the landscape, using the density of each species.

Intelligent life is a more complicated case, because the behavior patterns are more intricate and the behavior of a society as a whole is emergent from the behavior of the individuals in it. There is a relatively smooth scale of simulation quality though, from single-minded people that strictly follow simple daily routines, to an idea-based society where all kinds of organizations and innovations are developed and the society can undergo revolutionary paradigm shifts in addition

to smooth evolution. There are two main approaches for implementing this, top-down and ground-up. There is also some possibility for hybrid approaches that try to combine parts of each approach to get the best features of both.

The top-down approach tries to build various models of the society, and then move individuals according to the models. An example is the existing research on crowd behavior, which models the movement of crowds of people on the streets and in buildings [MT97] [SGC04]. This approach scales well to large societies, as there is no need to model things such as the crowd movement in cities which are not visible to an observer, because they have no lasting effects on anything else. The drawback is that to try to capture all the emergent behavior patterns of a society, more and more models need to be constructed. Economy, politics, culture, science, government, and land development are all quite different from each other, and potentially need own simulation models. But they can be developed separately and added one after another, to slowly approximate the desired level of sophistication.

The bottom-up approach starts from the people in the society, models their behavior, and lets the society form naturally in an emergent way. This approach can also be implemented in steps, from simple agents that just follow defined daily routines and avoid running over each other in the streets, to agents that create, evaluate, spread, and act on ideas, and are able to create or join organizations to collaborate towards some common goals. The problem with this approach is that it does not scale well to simulation of large societies because it needs to store some amount of information for each agent – they can not be generated on the fly as needed such as in the top-down model. But the advantage is that when the agents are sophisticated enough, society will happen by itself.

7.4 Emotional landscapes

Landscapes in computer games, movies, and illustrations can also be used to convey emotions and feelings to the user, in a similar way to background music [Fre03]. Examples of emotions that could be transferred by landscapes are tranquility, gloominess, anxiety, and excitement. Emotions may be location specific or situation specific.

Location specific emotions can be transferred by a suitable choice of ecotope (dark or light forests), and architecture (Gothic style and functionalism convey quite different feelings, as well as closed and labyrinthine versus open and well-lit buildings). Items with strong symbolical meanings may be used to alter the mood of a place. The lighting of a place also affects its mood.

When conveying situation specific emotions the static parts of the landscape can not be changed. However, there are various more dynamic aspects of any landscape that can be manipulated to transfer a given emotion. The weather affects our emotions strongly, the same landscape in sunlight or rain will feel very different. The visible inhabitants of the place can also be selected on the fly, songbirds and crows have different impacts on the mood of a place. It is even possible to change the way the vegetation is rendered to suit the targeted emotion, as slightly withered and dry with windblown leaves, or lush and flowering with flower petals falling from trees.

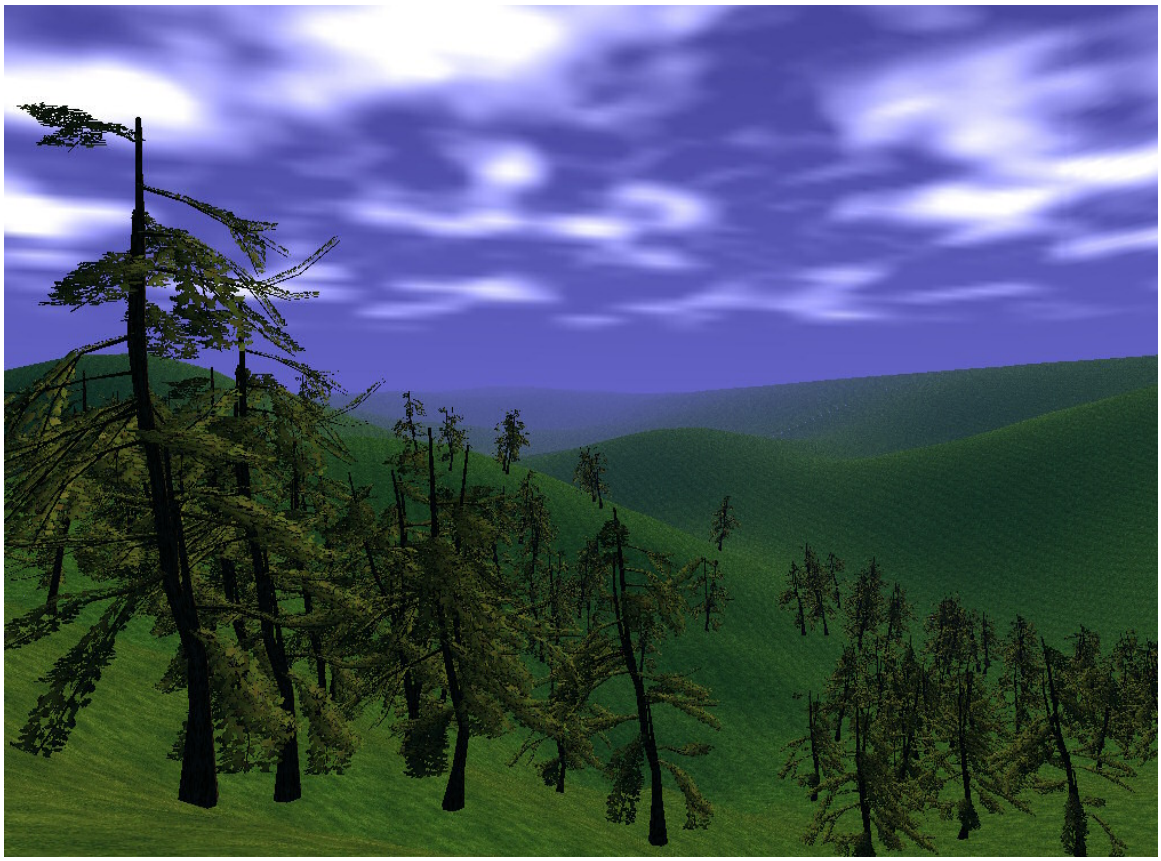

Figure 59: A screenshot showing a landscape with procedural trees and Perlin noise generated terrain and clouds. From the landscape engine being developed by the author [Häg06].

References

[AH05] Arul Asirvatham and Hugues Hoppe. Terrain rendering using gpu-based geometry clipmaps. In *GPU Gems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional, 2005.

[AIS77] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language : Towns, Buildings, Construction*. Oxford University Press, USA, 1977.

[BCF+05] Stephan Behrendt, Carsten Colditz, Oliver Franzke, Johannes Kopf, and Oliver Deussen. Realistic real-time rendering of landscapes using billboard clouds. *Computer Graphics Forum*, 24:507 – 516, 2005.

[Bur06] Carl Burke. Captain barcode's terrain synthesis pages, 2006. http://www.geocities.com/Area51/6902/terrain.html.

[BZ00] C. Baillard and A. Zisserman. A plane-sweep strategy for the 3d reconstruction of buildings from multiple images. In *19th ISPRS Congress and Exhibition*, 2000.

[Car80] Loren C. Carpenter. Computer rendering of fractal curves and surfaces. In *Proceedings of the 7th annual Conference on Computer graphics and interactive techniques*, page 109. ACM Press, 1980.

[CCDH05] C. Colditz, L. Coconu, O. Deussen, and H.-C. Hege. Real-time rendering of complex photorealistic landscapes using hybrid level-of-detail approaches. *Trends in Real-Time Landscape Visualization and Participation*, pages 97 – 106, 2005.

[Clo06] Wikipedia article on cloud types, 2006. http://en.wikipedia.org/wiki/Cloud_types.

[CSHD03] M. Cohen, J. Shade, S. Hiller, and O. Deussen. Wang tiles for image and texture generation. In *ACM Transactions on Graphics (Siggraph'03 Conference proceedings) 22*, pages 287 – 294, 2003.

[Dat05] Member-Amitava Datta. A new technique for rendering complex portals. *IEEE Transactions on Visualization and Computer Graphics*, 11:81 – 90, 2005.

[DCSD02] Oliver Deussen, Carsten Colditz, Marc Stamminger, and George Drettakis. Interactive visualization of complex plant ecosystems. In *Proceedings of the conference on Visualization '02*, pages 219 – 226, 2002.

[DDSD03] Xavier Decoret, Fredo Durand, Francois X. Sillion, and Julie Dorsey. Billboard clouds for extreme model simplification. *ACM Transactions on Graphics*, 22:689 – 696, 2003.

[DHL+98] Oliver Deussen, Pat Hanrahan, Bernd Lintermann, Radomír Měch, Matt Pharr, and Przemyslaw Prusinkiewicz. Realistic modeling and rendering of plant ecosystems. *Computer Graphics*, 32(Annual Conference Series):275 – 286, 1998.

[DN04] Philippe Decaudin and Fabrice Neyret. Rendering forest scenes in real-time. In *Rendering Techniques (Eurographics Symposium on Rendering - EGSR)*, pages 93 – 102, 2004.

[DWS+97] Mark A. Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Miller, Charles Aldrich, and Mark B. Mineev-Weinstein. ROAMing terrain: real-time optimally adapting meshes. In *IEEE Visualization 97*, pages 81 – 88, 1997.

[EMP+03] David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley. *Texturing and Modeling - A Procedural Approach (third edition)*. Morgan Kaufmann Publishers, 2003.

[FD03] O. Franzke and O. Deussen. Accurate graphical representation of plant leaves. *Plant growth modelling and its applications*, 2003.

[FFC82] Alain Fournier, Don Fussell, and Loren Carpenter. Computer rendering of stochastic models. *Communications of the ACM*, 25:371 – 384, 1982.

[FKN80] Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor. On visible surface generation by a priori tree structures. In *SIGGRAPH '80: Proceedings of the 7th Annual Conference on Computer Graphics and*

Interactive Techniques, pages 124 – 133, New York, NY, USA, 1980. ACM Press.

[Fre03] David Freeman. *Creating Emotion in Games: The Craft and Art of Emotioneering*. New Riders Games, 2003.

[GPSL03] S. Greuter, J. Parker, N. Stewart, and G. Leach. Real-time procedural generation of 'pseudo infinite' cities. *Graphite 2003 - International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia*, pages 87 – 94, 2003.

[Ham01] Johan Hammes. Modeling of ecosystems as a data source for real-time terrain rendering. In *Proceedings of the First International Symposium on Digital Earth Moving*, pages 98 – 111. Springer-Verlag London, UK, 2001.

[Häg06] Hans Häggström. The skycastle project, 2006. http://skycastle.sourceforge.net/.

[HL01] Mark J. Harris and Anselmo Lastra. Real-time cloud rendering. In *Eurographics 2001 Proceedings*, volume 20, 2001.

[hOWkYyC05] Siu hang Or, Kin-Hong Wong, Ying kin Yu, and Michael Ming yuan Chang. Highly automatic approach to architectural floorplan image understanding model generation. *Proceedings of 10th Fall Workshop Vision, Modeling, and Visualization*, pages 25 – 32, 2005.

[Jak00] Aleks Jakulin. Interactive vegetation rendering with slicing and blending. In *Proc. Eurographics 2000 (Short Presentations)*. Eurographics, 2000.

[KZGP06] S. Kocaman, L. Zhang, A. Gruen, and D. Poli. 3d city modeling from high-resolution satellite images, 2006. ISPRS Workshop on Topographic Mapping from Space.

[LD98] Bernd Lintermann and Oliver Deussen. A modelling method and user interface for creating plants. *Computer Graphics Forum*, 17:73, 1998.

[LD99] Bernd Lintermann and Oliver Deussen. Interactive modeling of plants. *IEEE Comput. Graph. Appl.*, 19:56 – 65, 1999.

[LD03] R. G. Laycock and A. M. Day. Automatically generating roof models from building footprints. In *The 11-th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision.*, volume 11, 2003.

[LD06] R.G. Laycock and A.M. Day. Image registration in a coarse three dimensional virtual environment. *Computer Graphics Forum*, 25:69 – 82, March 2006.

[Lew87] J. P. Lewis. Generalized stochastic subdivision. *ACM Transactions on Graphics*, 6:167 – 190, 1987.

[LH04] Frank Losasso and Hugues Hoppe. Geometry clipmaps: terrain rendering using nested regular grids. *ACM Transactions on Graphics*, 23:769 – 776, 2004.

[LN03] Sylvain Lefebvre and Fabrice Neyret. Pattern based procedural textures. In *SI3D '03: Proceedings of the 2003 symposium on Interactive 3D graphics*, pages 203 – 212, New York, NY, USA, 2003. ACM Press.

[LP02] Brendan Lane and Przemyslaw Prusinkiewicz. Generating spatial distributions for multilevel models of plant communities. *Proceedings of Graphics Interface 2002*, pages 69 – 80, 2002.

[Man82] Benoit B. Mandelbrot. *The Fractal Geometry of Nature*. W. H. Freeman et Co., 1982.

[Mar04] Jess Martin. The algorithmic beauty of buildings. Master's thesis, Trinity University, 2004.

[Mar06] Jess Martin. Procedural house generation: A method for dynamically generating floor plans. *Symposium on Interactive 3D Graphics and Games*, 2006.

[McA99] Michael McAllister. A watershed algorithm for triangulated terrains. In *CCCG*, 1999.

[Mic93] Microprose. X-com: Ufo enemy unknown, 1993.

[Mil86] Gavin S P Miller. The definition and rendering of terrain maps. In *Proceedings of the 13th annual Conference on Computer graphics and interactive techniques*, pages 39 – 48, 1986.

[MKM89] F. Kenton Musgrave, Craig E. Kolb, and Robert S. Mace. The synthesis and rendering of eroded fractal terrains. *Computer Graphics*, 23:41 – 50, 1989.

[MME98] A. Murta, J. Miller, and S. Embley. Managing complexity and feature placement in planetary terrain synthesis. In *In WSCG '98 Proceedings, Plzen, Czech Republic*, pages 276 – 282, 1998.

[MP96] Radomir Mech and Przemyslaw Prusinkiewicz. Visual models of plants interacting with their environment. In *SIGGRAPH*, pages 397 – 410, 1996.

[MQS+98] R. P. C. Morgan, J. N. Quinton, R. E. Smith, G. Govers, J. W. A. Poesen, K. Auerswald, G. Chisci, D. Torri, and M. E. Styczen. The european soil erosion model (eurosem): a dynamic approach for predicting sediment transport from fields and small catchments. *Earth Surface Processes and Landforms*, 23:527 – 544, 1998.

[MT97] S.R. Musse and D. Thalmann. A model of human crowd behavior: Group inter-relationship and collision detection analysis. In *Proc. Workshop Computer Animation and Simulation of Eurographics '97*, 1997.

[MTF03a] Stephan Mantler, Robert F. Tobler, and Anton L. Fuhrmann. The charismatic project: New approaches to modelling and rendering of urban environments. Technical report, Institute of Computer-Graphics, TU Braunschweig, 2003.

[MTF03b] Stephan Mantler, Robert F. Tobler, and Anton L. Fuhrmann. The state of the art in realtime rendering of vegetation. Technical report, VRVis Research Center for Virtual Reality and Visualization, Vienna, Austria, 2003.

[Ost01] Victor Ostromoukhov. A simple and efficient error-diffusion algorithm. In *SIGGRAPH '01: Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, pages 567 – 572, New York, NY, USA, 2001. ACM Press.

[PC01] Frank Perbet and Maric-Paule Cani. Animating prairies in real-time. In *Proceedings of the 2001 symposium on Interactive 3D graphics*, pages 103 – 110. ACM Press, 2001.

[Per02] Ken Perlin. Improving noise. In *SIGGRAPH '02: Proceedings of the 29th annual Conference on Computer graphics and interactive techniques*, pages 681 – 682, New York, NY, USA, 2002. ACM Press.

[Per06] Ken Perlin. Improved noise reference implementation, 2006. http://mrl.nyu.edu/∼perlin/noise/.

[PH93] Przemyslaw Prusinkiewicz and Mark Hammel. A fractal model of mountains with rivers. In *Proceeding of Graphics Interface*, pages 174 – 180, 1993.

[PJM94] Przemyslaw Prusinkiewicz, Mark James, and Radomír Měch. Synthetic topiary. *Computer Graphics*, 28(Annual Conference Series):351 – 358, 1994.

[PL90] Prusinkiewicz Przemyslaw and Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag, 1990.

[PM01] Y. I. H. Parish and P. Müller. Procedural modeling of cities. *SIGGRAPH'01*, pages 301 – 308, 2001.

[PS88] Heinz-Otto Peitgen and Dietmar Saupe, editors. *The Science of Fractal Images*. Springer-Verlag New York, Inc, 1988.

[PSS99] A. J. Preetham, Peter Shirley, and Brian E. Smits. A practical analytic model for daylight. In Alyn Rockwood, editor, *Siggraph 1999, Computer Grahics Procedings*, pages 91 – 100, Los Angeles, 1999. Addison Wesley Longman.

[RM00] Roerdink and Meijster. The watershed transform: Definitions, algorithms and parallelization strategies. *FUNDINF: Fundamenta Informatica*, 41, 2000.

[Rog06] Wikipedia article on roguelike games, 2006. http://en.wikipedia.org/wiki/Roguelike.

[Sch97] Gernot Schaufler. Nailboards: A rendering primitive for image caching in dynamic scenes. In Julie Dorsey and Philipp Slusallek, editors, *Eurographics Rendering Workshop 1997*, pages 151 – 162, New York City, NY, 1997. Springer Wien.

[SGC04] M. Sung, M. Gleicher, and S. Chenney. Scalable behaviors for crowd simulation. In *Computer Graphics Forum*, volume 23, pages 519 – 528, 2004.

[SK04] Gabor Szijártó and József Koloszár. Real-time hardware accelerated rendering of forests at human scale. In *WSCG*, pages 443 – 450, 2004.

[SLDD93] Jonathan W. Silvertown, Jonathan Lovett-Doust, and Jon Lovett Doust. *Introduction to Plant Population Biology*. Blackwell Science, 1993.

[Sof06] Planetside Software. Terragen, 2006. http://www.planetside.co.uk/terragen/.

[SS02] Alan Strahler and Arthur Strahler. *Physical geography : science and systems of the human environment (second edition)*. John Wiley, 2002.

[TR97] Demetri Terzopoulos and Tamer F. Rabie. Animat vision: Active vision in artificial animals. *Videre: Journal of Computer Vision Research*, 1:2 – 19, 1997.

[WP95] Jason Weber and Joseph Penn. Creation and rendering of realistic trees. In *SIGGRAPH '95: Proceedings of the 22nd annual Conference on Computer graphics and interactive techniques*, pages 119 – 128, New York, NY, USA, 1995. ACM Press.

[WWAD01] J. Willmott, L. I. Wright, D. B. Arnold, and A. M. Day. Rendering of large and complex urban environments for real time heritage reconstructions. In *VAST '01: Proceedings of the 2001 Conference on Virtual Reality, Archeology, and Cultural Heritage*, pages 111 – 120, New York, NY, USA, 2001. ACM Press.

[WWS00] Peter Wonka, Michael Wimmer, and Dieter Schmalstieg. Visibility preprocessing with occluder fusion for urban walkthroughs.

In *Proceedings of the Eurographics Workshop on Rendering 2000*, pages 71 – 82, 2000.

[WWSR03] Peter Wonka, Michael Wimmer, François Sillion, and William Ribarsky. Instant architecture. *ACM Transactions on Graphics*, 22:669 – 677, 2003. Proceeding.