Faculty of Mathematics and Natural Sciences

Department of
Mathematics and
Computing Science

# Scalability of Terrain Visualization in Large Virtual Environments

Esger G.N. Abbink

**Advisors:**

*dr. J.B.T.M. Roerdink*

Department of Mathematics and Computing Science

University of Groningen

*drs. M.J.B. van Delden*

*drs. M. Wierda*

Virtual Environments, Systems, and Consultancy

Zeegse

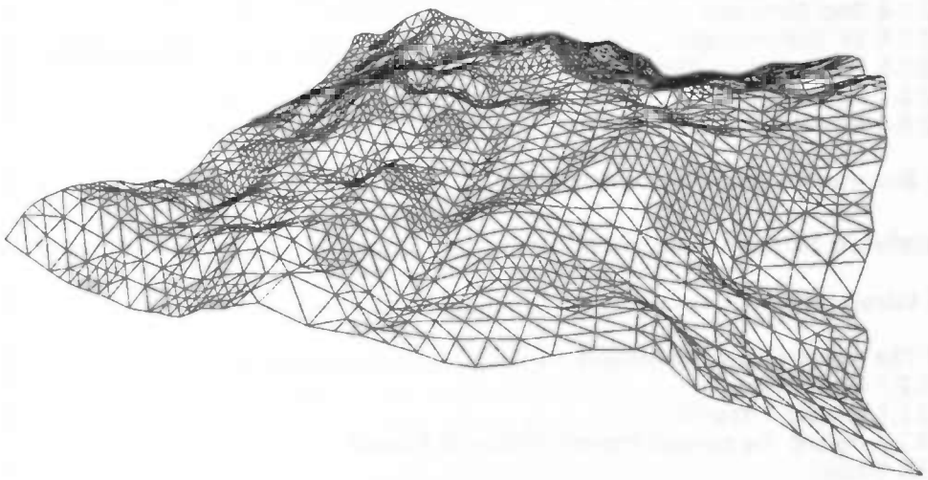


June, 2000



RuG

# Scalability of terrain visualization in large Virtual Environments



Esger G.N. Abbink

Department of Mathematics and Computing Science
Graduate specialization High Performance Computing and Imaging
Studentnumber 0831425

Gerkesklooster/ Zeegse, April 2000
Rapport vesc WR-2000/3

Table of Contents

# 1. Introduction

Only recently one is able to build dedicated low cost PC's for simulation or visualization. The cost/effectiveness has been substantially improved by the fact that 3D accelerators (a piece of hardware which implements part of a graphics pipeline) marketed as toys for gamers are in fact high-end graphics hardware with all the required professional features for high end visualization. Such a "game" card is used increasingly by professionals, which may be not so surprising when you compare the performance of these 'gaming cards' with the professional systems of a few years ago. Together with the increased computing power of a PC and other developments, for instance the usability of OpenGL on Windows, the PC has become a viable platform for simulation and visualization.

As a consequence applications which before required a workstation or even more powerful hardware, can now be run on a PC at fairly low costs. Of course the capabilities of top non-PC systems are increasing too and their performance is still way beyond the capabilities of a PC. To conclude one may say that the use and development of visualization and/or simulation systems has become more accessible. Also it is concluded that, that depending on aim and budget, there is a choice from an increasing range of computing hardware.

Unfortunately, choosing a different hardware setup has currently a major drawback: in many cases it will be necessary to rework or even rewrite large parts of the software to make use of the increased (or decreased) computing power. A software system optimized for a low-end hardware target platform will not be directly usable on a high-end Silicon Graphics (SG) multiprocessor system graphics computer. For example, when porting a simulation software system from a Pentium PC to a SG graphics workstation chances are that the Pentium computer will still have a better performance measured in Frames per Second (FPS) than the SG. This may be due to system specific optimizations, leaving the SG not using its full hardware potential. This problem not only occurs when moving between low and high-end computing hardware but also between generations of both low and high-end machines.

Changing the aim of a simulator is another common cause in making software obsolete, leading to the necessity of having to rewrite simulator software. For example, a flight simulator may have been optimally designed and built for the visualization of airplanes. As an effect it may render the architecture obsolete when it is required to simulate helicopters as well, resulting in a loss of investment. Since software development is an expensive and/or time-consuming affair it is preferable to avoid this problem in the first place. Required, then, is a software architecture that runs as optimal as possible on different hardware platforms and is capable of accommodating simulations and/or visualizations with different purposes (flight simulation, driving simulation, walkthrough, etcetera). Such an architecture may be called a multipurpose, scalable simulation and visualization system.

This study attempts to make a start with the design of such an architecture by making some first steps, leading to the implementation of a part of such an architecture:

1. An analysis and description is given of the various functions and concepts that may be found in the literature on different types of simulators;

2. The results of the study are put into work by testing the concept of scalability. Effectively, the visualization of a large scale terrain is designed as a scalable

subsystem of a simulation architecture. A piece of software is made that runs on various PC's, with respect to hardware, and that will adapt or can be adapted parametrically by hand to the available computing resources on that particular computing platform.

3. The technical/practical application of the routine is tested on a broad set of PC's.

The study is concluded with a discussion of the results of 2 and 3 ending in some recommendations for future research and software design or implementation.

This study has taken place at Virtual Environments Systems & Consultancy (VESC), a company seated in Zeegse. VESC engages itself mainly by doing real-time 3D visualization projects and has delivered several different systems. Internal tutor was dr. J.B.T.M. Roerdink (University of Groningen), external advisors were drs. M. Van Delden and drs. M. Wierda (VESC).

# 2. Virtual Environments

## *2.1 Starting points*

To be able to design a virtual environment system, first it has to be established what the desired features are. In this study the most central desired feature is **scalability**, interpreted broadly. The system should be able to scale in visual quality and speed. Also it should support both small and (very) large databases/environments at multiple resolutions. The study is limited to these aspects strictly to avoid a combinatoric explosion: one could add audio, multi-user capabilities, issues related to force feedback, dynamic physical simulation, multi-computer and/or multiprocessor hardware architectures Before addressing the scalable visualization of variable terrain databases, the process and conditions of Virtual Environment design in general are discussed.

## *2.2 Means-end analysis*

One of the basic issues is the intended purpose of the system. In our case the intended purpose for and use of the new system are not fixed or specifically defined. On the contrary, the goal is to make the system as widely usable as possible for as many applications. There are many applications for virtual environment systems. And each application and purpose has its own set of requirements. Making an in-depth summary of all applications and their respective features, specifications and requirements is nearly impossible and would require one to be an expert in all those fields. It is questionable, however, whether the resulting list would be of much value. Filled with technicalities it would probably devaluate with every appearance of new technologies or applications. Also the technologies and algorithms themselves are not the main point of our interest, much more important is to know *why* they are used. We first need to look at virtual environment systems on a higher level than that of an architecture or a collection of techniques. As a starting point to look at applications / simulators the cognitive psychological approach to man-machine interaction is taken.

## *2.3 Man-in-the-loop*

The first thing we can establish is that we are talking about systems designed to have a human subject do, undergo or experience something in an interactive environment. Or still better, we are talking about "man-in-the-loop" systems. And we further restrict ourselves to those systems where the "man-in-the-loop" controls some kind of object (vehicle, airplane, etc.) moving through a simulated environment (where hereafter "simulator" is mentioned, these restrictions are implied). Following from this specification we can split the system under design in three parts: the human subject, the simulation itself and the interface between the two. The first part, the human subject, is taken as is. The second and third part are the components that are to be designed and subsequently built.

The interface is the communication medium between the human subject and the noticeable part of the simulation, and is a combination of in- and outputs (or channels). It not only makes sense but is required to use the in- and outputs of a human subject *in the same way* as he or she would use them in real life. For example, if a subject in a driving simulator approaches a red traffic light the simulator could use a variety of ways to make the subject aware of this fact: an auditory or haptic signal, a spoken message or even a visual signal. But the only way to make the experience as close to reality as possible is to let the subject see a red traffic light where one would expect a light, correctly colored and lit. The same goes for the other senses and also for the construction and operation of the controls (to continue with the same example, the subject should steer the car with a steer-

ing wheel and not with a mouse or keypad). Figure 1 shows the various (possible) channels of an interface.

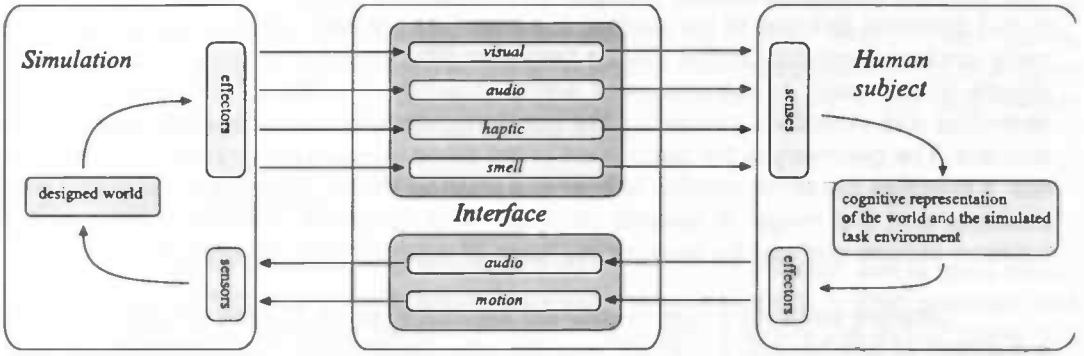



figure 1 - "Man in the loop" simulator

Interesting is that each channel need not be fully used to achieve a particular level of realism sufficient for a specific goal and it is often even possible to omit one or more channels completely. This is the main reason that such a wide range of usable simulators exists within most, if not all, application fields. For instance, for the visual display one could use a computer screen, a 360 degree projection system or a stereo head mounted display. All of these provide visual input for a human subject, but they differ in the levels of "reality" they can achieve. Similar examples can be found for the other channels. Since these channels (other than visual) offer an excellent opportunity to implement scalability they should be borne in mind. A listing of the various channels and how they may be filled in can be seen in figure 2.

**Human inputs** ← **Interface outputs**

| Channel | sensors | Interface output devices | characteristics | device driving hard- & software | "Databases" |
|---|---|---|---|---|---|
| visual | eyes | - computer monitor<br>- projection system<br>- HMD<br>- direct laser projection<br>- 3D visual (e.g. holographic)<br>- gauges / meters | - resolution<br>- viewable area size<br>- colordepth<br>- refresh / framerate<br>- mono / stereo<br>- fov<br>- luminance / contrast | - 2D / 3D pipeline<br>- DA / AD / relais boards | - 3D models<br>- textures<br>- scenegraph |
| audio | ears<br>body | - speakers<br>- subwoofer<br>- headphone | - frequency response / range<br>- dynamic range<br>- # separate channels<br>- # voices / channel | - sound synthesizer / mixing boards<br>- amplifiers<br>- Environmental Audio pipe<br>- speech synthesis | - recorded samples<br>- synthesized samples<br>- human operator |
| haptic | tactile<br>kinesthetic<br>force<br>vestibular<br>warmth / heat | - control loading<br>- motion system<br>- subwoofer | - # DOF<br>- operational range<br>- amount of force<br>- resolution / precision<br>- signal / noise (smoothness) | - control / operator system | - physical properties<br>- physical model |
| smell | nose | - micro nozzles | | | |

**Human outputs** → **Interface inputs**

| Channel | effectors | Interface input devices | characteristics | device driving hard- & software | "Databases" |
|---|---|---|---|---|---|
| audio | voice | - microphone(s) | - noise level<br>- frequency response | - voice recognition system | - vocabulaire |
| motion | hands<br>feet<br>head<br>body | - "limb operated controls" (steer, joystick, buttons, switches, mouse, pedals etc.)<br>- trackers | - operational range<br>- precision<br>- # DOF | - serial / parallel / digital / analog ports | - DOF points in / between objects |

figure 2 - Input / Output channels

The third part is the simulation. The simulation process continually evaluates the input it gets. Input is generated by the subject (delivered by the interface), the dynamic objects and everything else present in the simulation world. This input is used in conjunction with the current state of simulation world and objects to generate a new state for world and objects. Also the simulation process feeds the data into the interface needed for the output channels directed to the subject. Conceptually we can split the simulation in three parts: content, geometry and dynamics (see also Ellis, 1991). The content comprises the objects (either static or dynamic) and actors in a virtual world. Objects and actors are described and stored via properties like position, orientation, acceleration, texture, color etcetera. The geometry is the description of the world in terms of dimensionality and metrics. It specifies the terms needed to specify a position vector, the rules to order and relate positions and the range of allowed positions. The dynamics describe the interactions between objects and can be seen as the "laws of nature" of the simulation.

Different simulators feature different simulation parts, but the differences are not a question of one having "a content" and the other having not. The difference lies in the fact that one simulation may have a content consisting of, among others, 100 objects with highly complex and detailed behavior while an other might use just ten actors with very basic behavior. So the two main points of difference, also for the dynamics and geometry, are *complexity* and *size*. Both are dimensions that should be taken into account when designing a scalable system.

## *2.4 Hardware*

Until now we only have spoken about a simulation system on a conceptual level. Eventually however we want an implemented system that "works". But a running system implies the presence of hardware to run the simulator software. It also implies that a choice has to be made: what hardware will be used? There are many different systems and platforms available, ranging from a top-of-the-line Silicon Graphics Infinite Reality to a basic PC without a 3D graphics accelerator. Often budget concerns are the primary reason to choose a particular system/platform, but in addition personal preferences of developers play a role.

In this analysis we explicitly do not want to commit ourselves to a specific platform and be locked to it afterwards. Instead we want a system capable of performing well on a variety of hardware and platforms. Therefore we have to know what the differences are between the various hardware systems and platforms. There are too many systems to make an exhaustive comparison. Like we did above the comparison needs to be lifted to a conceptual level.

Basically any computer consists of the following components: a central processor, temporary storage, permanent storage, input devices and output devices. These components have many faces and also have rather different features. The main differences are in terms of speed and size (processor type/clock, memory size/access time, pixel fillrate, texture resolution, polygon count etcetera). Apart from increasing speed by using a faster processor it also is possible to use multiple processors, either within a single cabinet (a SMP or MPP system) or within multiple cabinets (distributed processing).

Unfortunately the "similarity" of the differences does not imply that all hardware can be operated or "driven" the same way by software. On the contrary, every piece of hardware or computer system component has a degree of uniqueness, which can become a very large obstacle if it is not taken into account early enough when porting a piece of software from one platform to another.

## 2.5 Real-world simulators

Leaving the theoretical approach we will now take a different perspective and see how existing (real-world) simulators fit the analysis. To do this most simulator types will be briefly discussed and if appropriate we will try to give a general filling in of the channels (a channel will be specified by either the devices used, or by "quality" where quality stands for realism). In addition we will take a closer look at the characteristics of the scenes or "theaters of action", in particular the terrain, of the different simulators. In the next chapter a technique for terrain visualization will be discussed.

### 2.5.1 Flight Simulators

This widely known type of simulator may cost millions of Euros and is used extensively by both commercial airlines and military airforce organizations to train, evaluate and maintain their pilots. In many cases a simulator is available before the actual plane has been built and is used to test the design. The use of these simulators as trainer imposes several important requirements: the physical model and the environment need to be of sufficient fidelity, the system must allow for orchestrated scenarios and real-time operator intervention, plus that it must be possible to record various streams of data in the simulator during a session for the purpose of later evaluation and/or debriefing.

| Channel | Human input | Human output |
|---|---|---|
| visual | overall high quality display (high resolution, large field of view projection system, anti-aliased, high framerate etc.), realistic operating environment (mockup of cockpit) | |
| audio | realistic radio-traffic (done by operator) | voice |
| haptic | loaded controls (stick etc.), 6 DOF motion system | stick, buttons, switches etc. |

figure 3 - Flight simulator channels

**Scene & terrain:**

Typically, because of the nature (speed, altitude) of the simulated vehicle, these simulators feature very large terrains in terms of geometric size (several hundreds or even thousands of square kilometers) but also in the distances at which terrain is still visible (and visualized). Landscape and or object detailing is not done extensively. Usually only the major landmarks are included and buildings are low detail. This is less true for low-altitude trainers involving helicopters or other low-flying aircraft. For the most part surface following is not important, it becomes important only when in a take-off or landing situation.

### 2.5.2 Driving Simulators

This is a very broad category, both in terms of functionality and costs Although the first association with 'driving' is 'a car' a driving simulator may have many moving and controllable vehicles such as trucks, busses, racecars, firetrucks, ambulances while militaries use various tank and other armoured vehicle simulators. The main purpose of these simulators is to train personnel, the second studying environment and task-dependent human operator behaviour.

For a vehicle driving simulator it is important that the simulated environment is real-time controllable (an operator can influence the behaviour (of part) of the simulation environment at run-time) or at least orchestratable (an operator can set up the simulation environment for specific behaviour to appear but has no influence during run-time). Performing specific repeatable experiments like having a car cross a red light or performing an emergency stop in front of the simulated car is impossible without direct or indirect control over the scene and the dynamic objects. In the case of a training simulator it must be possible to create the driving context or traffic situation in which you want to place the trainee(s) given his or hers current state of capabilities and knowledge.

| Channel | Human input | Human output |
|---|---|---|
| visual | low- to high quality / realism | |
| audio | none to high quality / realism | |
| haptic | loaded controls (steer, pedals) and motion system | steer, pedals, keyboard |

figure 4 - Driving simulator channels

**Scene & terrain:**

Depending of the aim of the system the scenes are small to medium sized, depicting areas of several square kilometers. Due to the short viewing distance, objects and terrain detailing should be high. Because of the type of vehicle accurate terrain/surface following is necessary. Generally, since the viewpoint is close above the terrain, it is not possible to view very far.

### *2.5.3 Rail vehicle simulators*

Used in both stand-alone and hybrid configurations, these kind of simulators are in use at railroad companies, tram companies and similar. The hybrid configurations connect one or more 'standard' simulators with a simulation of the railway operator station. The main use of these simulators is training personnel. The objectives differ per site. Some simulators are only used for route-memorizing, others are mainly used to train personnel in emergency situations while others are used to train basic train control operations (like coupling train-units or braking). Especially in a simulator used for the latter an environment as realistic as possible (a high fidelity) is required. In these cases the application of a motion-base is more rule than exception (and more compelling as well). Route-memorizing on the other hand is however mainly audio-visual oriented and a highly accurate physics model of the train-unit is not necessary. Again, depending on the objective scenario control is needed.

| Channel | Human input | Human output |
|---|---|---|
| visual | low- to high quality / realism | |
| audio | none to high quality / realism | |
| haptic | loaded controls (steer, pedals) and motion system | steer, pedals, keyboard |

figure 5 - Train simulator channels

**Scene & terrain:**

For the most part these simulators are similar to driving simulators but with two differences. The first is the possibly larger size of the scene. The second is unique for these type of simulators: the driver can only go forward (and possibly) backwards on a predestined track. The view from any position on the track is known upfront, apart from dynamic objects, given way for the use of fairly simple but highly effective scenedata reduction algorithms.

### *2.5.4 Ship Simulator*

This is a less common type of simulator. Most are large scale and use high-end hardware like motion platforms and 360 degree projection screens. Operated by both naval military departments and commercial vessel companies they are mainly used for training personnel. The simulated ships range from trawlers and destroyers to super-tankers. Given the possibly major consequences of error and the small margins of error in controlling these ships (in particular when dealing with massive tankers) the physics model of these simulations must be extremely accurate. A major difference with other types of simulations is the importance of the environmental forces of nature. In most simulators an accurate natural environment is not necessary, but here an accurate modeling of very complex natural phenomena is mandatory with respect to water and weather such as currents, flows, tides and visibility with respect to rain, bough spray, wind (-effects), etcetera.

| Channel | Human input | Human output |
|---|---|---|
| visual | high quality display | |
| audio | realistic radio-traffic (done by operator) | voice |
| haptic | motion system, loaded controls | joysticks, buttons, switches etc. |

figure 6 - Ship simulator channels

**Scene & terrain:**

The scenes used in these simulators are either open waters (sea) or special situations (harbor, sluices) where specific skills are trained. Depending on the aim of the simulator scenes may look very simple and monotone, a flat water surface, or very high detail (the movement and currents of the water are visualized as realistically as possible). If there is a shore, generally detailing will be low. In the second case more detailing is done, but mostly requirements are relatively low and as such often only the minimum needed is done.

### *2.5.5 VE Walkthrough*

The most numerous form of a simulator of this type is as PC game (the first-person shooter) and includes titles like "Doom", "Quake" and numerous others. Other applications also exist. Architects and contractors can use a virtual three dimensional representation of a not yet existing (re-) building, instead of the effortful interpretation of construction schemes and artist drawings to have their customers make better founded decisions during the design process, all before a stone is laid. Complete houses, buildings and building complexes can be previewed like this. Another slowly emerging use is the three dimensional 'virtual walkthrough' of a specific site or building (university, city, lab etcetera),

gallery or museum, in some cases for use over the Internet, as a successor of or alternative to traditional two-dimensional representations (Internet website, brochure, slide presentations).

| Channel | Human input | Human output |
|---|---|---|
| visual | low- to medium quality | |
| audio | medium- to high quality | |
| haptic | | keyboard, mouse, joystick |

figure 7 - Walkthrough channels

**Scene & terrain:**

Caused by both technological limitations and limited application content, most of the scenes for these systems are indoors in the form of interconnecting rooms. However, due to technological advances, outdoor scenes are also used increasingly. The scenes may be highly detailed, but because of view blocking walls it is not possible to look very far.

### *2.5.6 Entertainment Simulators*

Mostly, these applications are not real fully functional simulators. The visiting and paying subjects are to believe that they are in some kind of moving vehicle (often a carriage of some sort) but they cannot control it, the traveled route is preprogrammed. In these systems there is no input from human to machine (simulation), they are entirely focussed on output to the subjects. The aim is to give them via this stimulation a fun or exciting ride.

| Channel | Human input | Human output |
|---|---|---|
| visual | medium- to high quality | |
| audio | medium- to high quality | |
| haptic | motion system | |

figure 8 - Entertainment simulator channels

**Scene & terrain:**

Most of the time these systems use a different visualization approach than the simulators discussed above. In those cases the image was computed real-time, in this case the image (or rather the series of images) are precomputed. The complexity of the scene can thus be far higher than would be the case when the images would be computed real-time.

### *2.5.7 Telepresence*

The restriction we made earlier was that we were only interested in systems where a human subject controls a moving object while it moves through a simulated environment. The mainstream of simulators that satisfy this specification are vehicle (car, truck, train, airplane etcetera) simulators, which are therefore our main interest. In all

these simulators the human subject is supposed to be seated in the simulated vehicle and most of them portray a realistic task-environment. The concept of realism is interpreted as that the scene and actions must be believable thus constituting a real-world scene. However they are not the only ones fitting the restrictions we made. A wide variety of systems that allow the operator to "get into" environments where that would be impossible in the real world (robot control, human body, microscopic worlds, hazardous environments) exist also.

Many of these systems are not really "virtual" in the sense that they do not portray a situation that does not exist (at the same time) in the real world, instead the controlled robot does exist and moves around somewhere. The quality requirements for the various channels can not be specified in a single table for all systems in this category due to their large differences. One thing sets them apart from the other systems mentioned above though. They in general do not have to realistically mimic a real world situation from the viewpoint of a human, because in the real world a human does not have or even cannot have a viewpoint in that specific situation/environment. Because of this they do not necessarily have a need for a high level of quality/realism.

**Scene & terrain:**

The scenes used in these systems vary strongly. Generally however the content of the visualization is symbolic and not an as-realistic-as-possible image.

### *2.5.8 Virtual GIS*

In itself not a simulator, a virtual GIS (Geographic Information System) system is used for displaying a different kind of (high level) information (very large terrain databases for instance). As such it can be used as an educational or planning tool. But it is also possible to combine the use of a GIS with a vehicle simulator. Take for instance military exercises where tank- and aircraft simulators are coupled with GIS systems where commanders give orders to several (tank) platoons and/or flight squadrons.

**Scene & terrain:**

The common characteristic of these systems is that the size of the area or terrain database is huge. Depending on application only non-perspective views of very large pieces of terrain (satellite photos for example) or local perspective views or even a combination are possible.

## *2.6 Summary & practical continuation*

If we evaluate the different simulators and their characteristics we may draw the following conclusions. We can see that most simulator types do not necessitate a specific quality for a channel, instead most seem to allow a range of possibilities. Above that, the actual implementation of a particular simulator system is dependent on the task to be accomplished and foremost the available budget. As a result, even when the constraints with respect tot ergonomics, psychology and human perception may be precise and firm, wide ranges of a single simulator type may exist. For example, one may have a budget driving simulator (with a TV screen, a game steeringwheel plus pedals and a chair) but also a high fidelity (scientific measurement) simulator (with 16 high-end graphics pipes and a huge motion system. This means a system capable of performing all/most of those tasks at different budgets will have to be customizable to allow for these differences. We can also see that most of the time when one channel is of a particular quality, the other

channels that are used are of matching quality, so as to not disturb the experience by too great a difference of realism presented to the subject.

Another observation is that apparently not all channels are of equal importance. This is not only true for a particular simulator: independent of simulator type it is possible to make a general ordering based on importance. Not really surprising, the visual (input to human) channel is, as a rule, the most important. Together with being the most important it is also the most demanding in terms of hardware resources (computing power, memory, display devices etc.) and requires also considerable effort for content production. This is demonstrated in practice by the large research and development efforts in this area and the costs of high-end visualization hardware. The second part of this study will focus on implementing a specific part of the visual channel.

# 3. Scalable terrain visualization

## *3.1 Introduction*

The visualization of large pieces of terrain by mesh reduction, triangulation, Level Of Detailing (LOD) etcetera is a challenging technical problem for which several solutions exist. Most are based on some form of model simplification: a piece of terrain can be shown using a number of different geometry sets. This is called a multiresolution terrain (MT). Very few of these are suited for real-time applications and even fewer exist that will run acceptably on anything other than high-end hardware. Also in many cases the available literature (for example Puppo, 97 and Lindstrom, 96B) deals only with such a multiresolution terrain algorithm as is and does not provide additional information (on, for instance, texturing the terrain, or integrating it with an all-purpose visualization toolkit).

The subject of this study will be the (partial) design and implementation of a MT algorithm including features such as texture mapping which is intended to run on both low-end and high-end hardware with OpenGL support. This will be achieved by a parameterization of the software to make it adjustable to available features and performance of the hardware. The implementation will be based on an existing algorithm (used in the sample implementation of Sharp, 99B). This choice was made because from the sample implementation a promising base performance was observed and the algorithm and mathematical objects have some very practical features (see also section 3.3). After a performance evaluation on different platforms to assess performance and identify bottle-necks, optimizations will be proposed.

The implementation and other programming work has been done on a Windows NT based computer using Visual C++ (a programming environment) and 3D Studio MAX (a 3D modeling program). The implementation will be added to a development snapshot of 4Space, the VESC 3D visualization toolkit. The available test computers range from high-end Intergraph workstations to off-the-shelf PC's equipped with a (game) 3D accelerator.

## *3.2 The sample implementation*

### *3.2.1 Introduction*

The sample implementation (SI) mentioned above is a program containing an example implementation of the central differencing algorithm discussed in Sharp, 99B. More information about central differencing in general can be found in Watt and Watt's "Advanced Animation and Rendering Techniques" (Addison-Wesley 1992). For an explanation of the central differencing algorithm in the form it is used here see section 3.3.5.

### *3.2.2 Getting it to work*

The SI was obtained from the Internet webpage of the author (http://www.cs.dartmouth.edu/~bsharp/gdmag) and contained the source, a dataset as well as pre-compiled binaries. Unfortunately both MS Visual C++ 5 and 6 were unable to compile this source. Assistance of the author was to no avail. The pre-compiled binaries did work however and showed good framerates while displaying a multi-textured patch landscape on a PentiumII-266Mhz computer equipped with a 3D accelerator card based on the Nvidia TNT chipset (a game card).

### 3.2.3 Porting the sample implementation to 4Space

Because of several reasons it was deemed impractical to port or transfer the source of the SI to 4Space. First of all the fact that the SI did not compile. But mainly the fact is that it was set up as a one-trick program. It shows a patch landscape and to accomplish that does its own GL calls, loads textures itself etcetera. Also these subparts of the program are very much weaved together. The goal was however a general approach integrated in a general 3D toolkit. This does imply a new implementation was needed and that that implementation had to be built from scratch.

## 3.3 Design

### 3.3.1 Goal

The design constraint of the Multiresolution Terrain to be implemented is that it should be a "general approach integratable in a general 3D toolkit". The toolkit is in this case 4Space.

### 3.3.2 Requirements

The practical requirement is a scalable terrain visualization extension to 4Space with as few limitations as possible. The extension should be platform independent (although at this moment 4Space runs only on Microsoft Windows, it is developed to be platform independent and other platforms may become available in the future). To allow the extension to adapt or be adapted to the capabilities of different target systems the extension should be parameterized with respect to controlling image quality and geometric complexity. To allow for simulator-self-adaptation it must be possible to change parameter values at run-time.

A set of concrete requirements for the functionality of the extension can be deduced:

- It must function transparently in 4Space.
- It must be parameterized.
- It must react correctly to changes of parameter values at run-time.
- The landscape it represents must be deformable (creation of bomb craters, tracks) at run-time.

Added to these top level functionality requirements some more technical ones are added:

- The displayed landscape must be a relatively close approximation of the mathematical form.
- The displayed landscape should look realistic. To accomplish that (multi-)texturing, smooth shading and/or vertex coloring have to be supported.
- The extension will have to be usable in a production environment. As a consequence the extension (or a yet to be built converter utility) has to be able to build its internal representation from an external source. In other words, importing terrain data from other formats has to be possible.

### 3.3.3 Terrain representation

As basic representation of the terrain the Bicubic Bézier Patch (BBP) is selected since BBP's have particular properties that prove rather practical for real-time visualization. A BBP surface lies within the convex hull that is defined by the control points of the

patch and interpolates its four corner control points. A terrain surface is built up from interconnected BBP's, the number of which is limited only by the available memory at run-time. Along shared edges the corresponding control points of two BBP's have the same coordinates (the patches share these control points). This composed surface is inherently (because of the shared controls) $C^0$ continuous while $C^1$ and $G^1$ (less strong as $C^1$ as the tangent vectors only need to have the same direction, not the same size) continuity are easily achievable. Other bicubic patches (i.e. Hermite or B-Spline) easily achieve $C^0$, $C^1$ and even higher order continuities also, the higher continuities are not necessarily needed though. More important, other bicubic patch types often lack either one or even both of the first two properties (interpolate corners, convex hull) mentioned above. Also they are generally more complex and more computationally intensive.

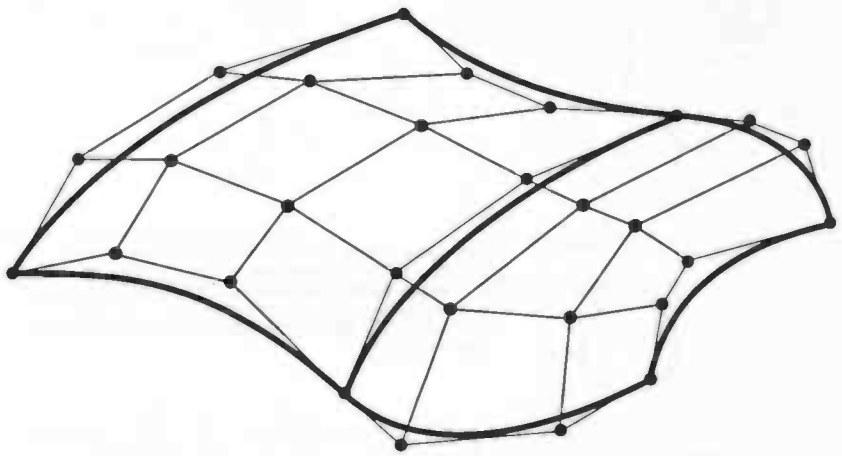


figure 9 - Two interconnected Bézier patches

Using BBP's for landscape creation has a practical advantage: many 3D modeling programs support some form(s) of bicubic patch to create surfaces. These may be NURBS, Bézier or other surfaces. Important is that it is always possible to substitute one form of bicubic patch with (multiple) instances of another form (see also figure 10).

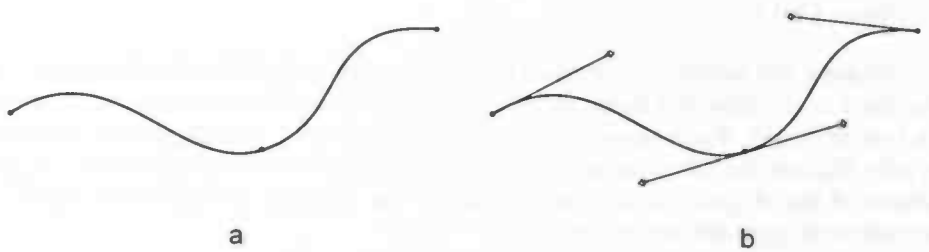



figure 10 - a) curve (no Bézier) b) two Bézier curves describing the same geometry

It is inferred from above that it will be possible to export and if necessary convert the terrain surface in the form of bicubic patches from a modeler to 4Space. However, standard export filters will write polygon geometry and not the actual patch descriptions (the control points i.e.). In the case of 3D Studio MAX a possible solution is to write an exporter plug-in that actually accesses the mathematical surface descriptions and writes them interpretably to file. Modeler programs that do not allow access to these descriptions (or do not allow for plug-ins) as well as terrain models/datasets that come in other forms

than bicubic patches require a different approach. This could include surface fitting or interpolation algorithms to generate, as required result, interconnected BBP's.

### *3.3.4 Basic setup*

We have a set of Bézier control points describing a terrain and a 3D engine (4Space) that can draw polygons The extension which will be added has to be the bridge between these two. As input it has a set of control points, and as output it has to deliver a 4Space compliant geometry so it can be rendered.

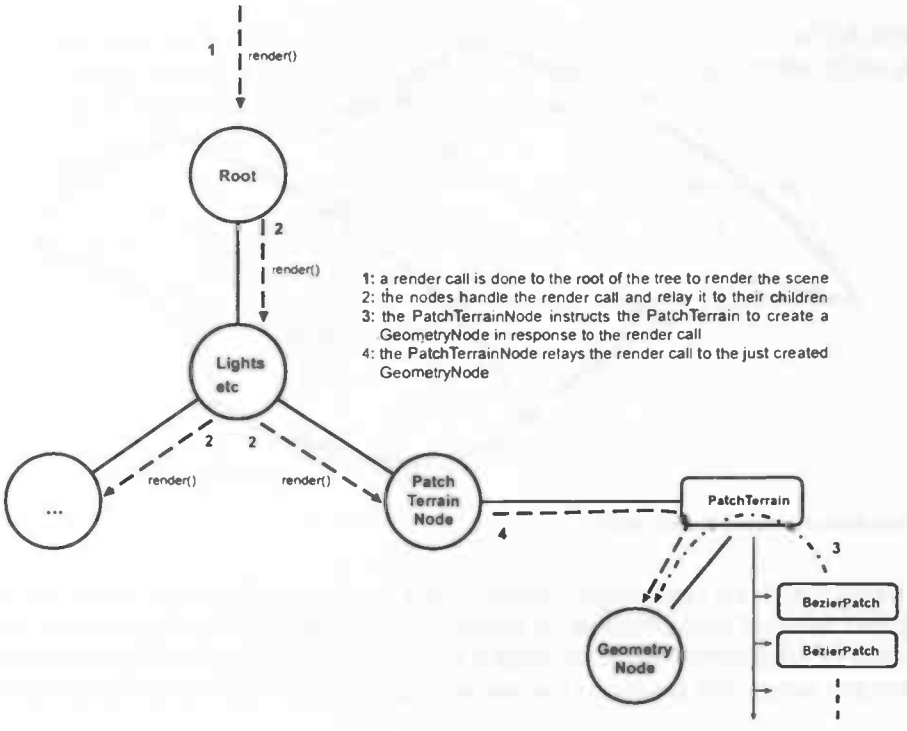



figure 11 - 4Space & MT structure

Because the extension will have to function properly and transparently in 4Space it is important to consider the basic structure of 4Space. In 4Space a scene is represented as a tree of nodes. Each node represents an object, light, transformation etcetera. To comply with 4Space the patch terrain has to be integrated in this tree structure. The generic structure of the 4Space node-tree allows for the creation of new node types, so a *patchterrain* node type will be introduced.

This new node will be implemented as the C++ class *cFSPatchTerrainNode*. It has "a patchterrain" as its most important member and will act as middle-man between 4Space and the also new *cPatchTerrain* class. The *cPatchTerrain* class contains a geometry node and a number of instances of a third new class *cBezierPatch*. The instances of this *cBezierPatch* class are the individual patches that compose the landscape. The node is used for rendering and will be discussed in section 3.4.2. The resulting structure can be seen in figure 11.

### 3.3.5 Tessellation algorithm

As can be seen in figure 11 the *cPatchTerrain* class creates geometry from its patches. Actually it asks its patches (instances of *cBezierPatch*) to tessellate themselves. Tessellation is a process where from the mathematical description of a curved surface, triangles are generated to create an approximation of this surface to a specified degree.

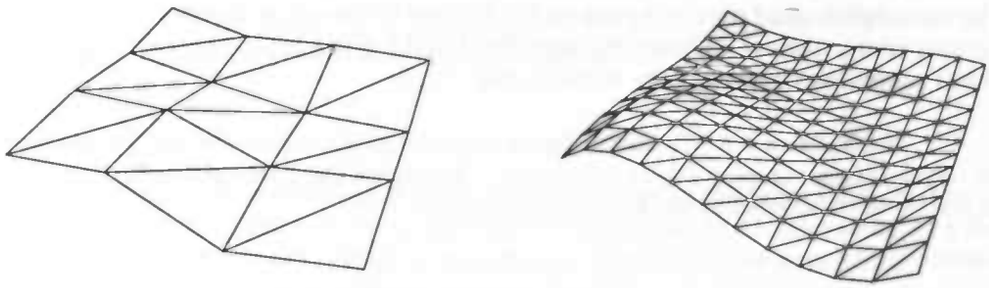


figure 12 - surface S tessellated to a degree **X** surface S tessellated to degree **X + n**

The mathematical description of the terrain is the set of control points and the mathematical formula for a bicubic bezier patch (BBP):

Eq. 1: $$P(u,v) = \sum_{i=0}^{3} \sum_{j=0}^{3} p_{ij} B_i^3(u) B_j^3(v)$$

$p_{ij}$ as the controls of patch P
and $B_i$ and $B_j$ as the Bernstein Basis functions

This formula gives for every valid (u, v) the corresponding point on the surface in Cartesian coordinates. The Bernstein functions look like this:

Eq. 2: $$B_i^n(u) = \binom{n}{i} u^i (1-u)^{n-i} \quad \text{for } 0 \leq i \leq n$$

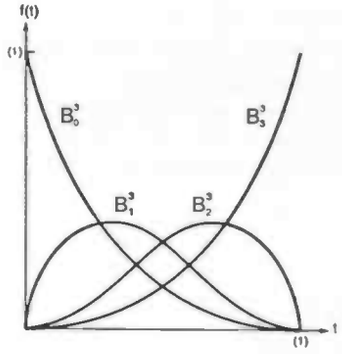



figure 13 - Bernstein Basis functions

Triangles are created between points (vertices) in 3D space. By looking at both equations it can be determined that a simple uniform grid of which the points are calculated by using equation 1 is not practical. This would be very expensive computationally because of the Bernstein basis functions making equation 1 cubic in both u and v. Furthermore using a uniform grid would result in horrendous amounts of triangles for landscapes of any but the tiniest (unusable) sizes. This approach is useless since any real-time 3D render engine will choke in the numbers of triangles. To overcome high amounts of triangles the quality of the approximation could be decreased. However the (compulsory) low triangle-count terrain will be visually lacking in quality. The solution is to use high amounts of triangles only if the curvature of the surface requires it and use less elsewhere. The tessellation used here is based on the *Central Differencing* algorithm. To explain the working of the *Central Differencing* algorithm (CDA), it is put to work on the one-dimensional counterpart of a BBP, the Bézier curve.

The basis of the CDA is the Taylor polynomial. A function f(x) can be approximated near a value v using a Taylor polynomial. The higher the order of this Taylor polynomial, the closer it resembles the function f(x) near v.

Eq. 3: $$f(v + dv) = \sum_{i=0}^{\infty} \frac{dv^i}{i!} f^i(v)$$

with:
$f^i$ = the i'th derivative

Eq. 4: $$C(v) = \sum_{i=0}^{3} p_i B_i^3(v)$$

Equation 4 is the mathematical formula of a Bézier curve. The Bernstein bases cause equation 4 to be cubic. So for a Taylor approximation of equation 4 every term after the fourth will be zero. This results in:

Eq. 5: $$C(v + dv) = \sum_{i=0}^{3} \frac{dv^i}{i!} C^i(v)$$

$$= C(v) + dv\, C'(v) + \frac{dv^2}{2} C''(v) + \frac{dv^3}{6} C'''(v)$$

This equation will prove practical for central differencing. Central differencing is an approximation that works by finding the midpoint between two (end)points on a curve and then the same method is applied recursively to the new points. Now we take C(v) as the midpoint and add up equation 5: C (v + dv) and 6: C (v - dv).

Eq. 6: $$C(v - dv) = C(v) - dv\,C'(v) + \frac{dv^2}{2} C''(v) - \frac{dv^3}{6} C'''(v)$$

Eq. 7: $$C(v + dv) + C(v - dv) = 2C(v) + dv^2 C''(v)$$

Eq. 8: $$C(v) = \frac{C(v + dv) + C(v - dv)}{2} - \frac{dv^2 C''(v)}{2}$$

The first term in equation 8 simply is the average of the two endpoints. The second looks more tricky but it is the second derivative of a cubic function which is a linear function. Therefore this also simply is an average, this time of the second derivatives at the endpoints. Substituting all this into equation 8 gives:

Eq. 9: $$C(v) = \frac{C(v + dv) + C(v - dv)}{2} - \frac{dv^2 (C''(v + dv) + C''(v - dv))}{4}$$

Using equation 9 it is possible to compute the midpoint C(v) between two points on a curve if the (Cartesian coordinates of the) points and the second derivatives in those points are known. For central differencing to work, this has to be repeatable. In other words to compute midpoints between the corners and the new point, for the new point these two pieces of information, the coordinates of this point and the second derivative in this point, need to be known. Fortunately we already have both. The coordinates of the midpoint are known because they were just computed. For the second derivative we look again at equation 8: the second term was the average of the second derivatives in the endpoints. And because the second derivative was a linear function this is the second derivative in the midpoint.

The equations above allow for recursing and new midpoints to be computed a lot faster than from the general curve function (equation 4). This is however just one part of the problem. There still has to be a way to limit the number of curve segments generated. Instead of just recursing to an arbitrary preset level, the algorithm should base the decision to recurse upon the amount of local curvature. The Taylor polynomial provides a solution here. If we look at equation 8 the midpoint consists of the average of the endpoints and the average of the second derivatives of the endpoints. This means that it is the second term of equation 8 which determines how far away the midpoint is from the midpoint of the line through the endpoints. The size of the second term is as such a direct measure of the local curvature and therefore we can use it in the decision whether to recurse or not.

a) Curve b) Uniform Tesselation c) central differencing

figure 14

Now we know how to tessellate a curve we need to extend that approach to make it work for patches. To make recursion work the patch has to be subdivided. A logical approach is to compute the midpoints of the edges and the center point, which results in four similarly shaped smaller pieces. Let us take a patch P, then P (u, v) is the patch surface for u and v in the range [0, 1]. If we now take P (0, v), and similarly P (1, v), P (u, 0) and P (u, 1), that gives us the four edges of the patch. The edges are one-dimensional functions in u or v, in other words: curves (defined by the particular edge control points). So the midpoints of the edges are computable straightforwardly using one-dimensional central differencing. Before we turn to the center point, we need to compute some additional values for the midpoints M1, M2, M3 and M4.

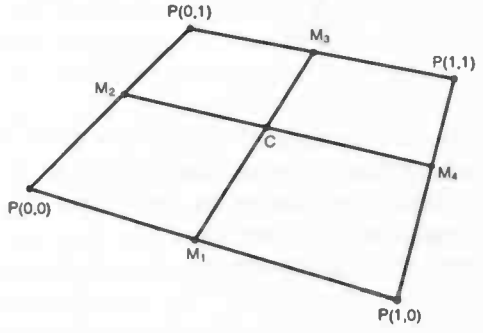



figure 15 - Patch P subdivided (M1, M2, M3 and M4 are midpoints, C is the center point)

We need to know the second order partial derivatives at the midpoints. Otherwise one-dimensional central differencing cannot be used again in both directions for the recursion step to work. More specifically: the second partial derivative with respect to v at the midpoints of P (u, 0) and P (u, 1) and the second partial derivative with respect to u at the midpoints of P (0, v) and P (1, v). We are already able to compute the other partials. To do this we go back to equation 1 and take the second order partial derivative with respect to v.

$$\text{Eq. 10:} \quad \frac{\partial^2 P(u,v)}{\partial v^2} = \sum_{i=0}^{3} \sum_{j=0}^{3} p_{ij} B_i^3(u) \frac{\partial^2 B_j^3(v)}{\partial v^2}$$

Since both basis functions are cubic, the partial derivative is a cubic function in u. This function can be interpolated in u using the one-dimensional central differencing algorithm. For this to work we need to know the partial second order mixed derivatives and

(see equation 9) derivatives in the endpoints. So, to compute the necessary information in the midpoints we need the following information at the corners:

Expr. 1: $P(u,v)$

Expr. 2: $\frac{\partial^2 P(u,v)}{\partial u^2}$

Expr. 3: $\frac{\partial^2 P(u,v)}{\partial v^2}$

Expr. 4: $\frac{\partial^4 P(u,v)}{\partial u^2 \partial v^2} = \frac{\partial^4 P(u,v)}{\partial v^2 \partial u^2}$

Using these and central differencing provides the same information in the midpoints. (Note that the fourth partial derivatives are linear in u and v and can simply be interpolated from the corners.)

The computation of the center point is yet unresolved. It is unclear how we could compute it from the corners. But again one-dimensional central differencing provides a solution, for we have computed the midpoints. The center point is the midpoint of both the curve between the u-midpoints and the curve between the v-midpoints. Using the same technique as before we can compute the expressions 1 - 4 for the center, just as we already have for the other points. Now we have enough information for recursion and do the same thing for the four smaller squares. Obviously the algorithm is not allowed to recurse infinitely, so a way to determine when it should stop recursing is required. This subject will be dealt with in detail later in this text (see section 3.4.2). In any case, the same mechanism used earlier for curves can be used for these surfaces.

### *3.3.6 Implementation approach*

The proposed extension of 4Space is a fairly complex one. Therefore, due to limited time, for this study only a partial implementation will be done. This means some things will be left to be done at a later date. The initial implementation will limit the form of the terrain to a rectangle, with the same number of patches on each side. As a third limitation just one instance (one node) of a patch landscape will be supported. Non-essential support functions required by 4Space, load/save etc, will also be left unimplemented. Terrain deformation will be limited to control point changes. Finally the terrain will only be flat-shaded in the first implementation. This does not mean however there is no regard for these features during implementation. Everything will be set up to allow for later extension with these features as much as possible.

The core that is implemented is an extension that allows 4Space to render in real-time a flat-shaded landscape described by a square of interconnected BBP's.

## 3.4 Implementation

### 3.4.1 Terrain data generation

Before a visualization can be implemented and run, there has to be something to show. Because import and export lies beyond the scope of this study a shortcut is taken. Terrain data will be generated by a utility program "TerrainGen". This program first creates an evenly spaced grid of control points in x and y direction. The number of control points is specified, as well as the length and width of the grid. Secondly each control point is assigned a random z-value (height) recursively. This z-value is not fully random but mediated by the range of z-values of neighbouring control points. Finally the z-values of the control points are corrected to make the composed landscape C1 continuous. This generates a smooth (rolling) landscape.

Subsequently the generated control points are saved to a file named "terrain.ptd". This file will be read by the "MRterrain" program (see also appendix A3). This is a front-end program using the 4Space engine which will set up a scene containing a patchterrain and instructs it to load its control points from the file "terrain.ptd". Also the program sets up an interface to interactively change parameters of the patches.

### 3.4.2 Implementing Central Differencing

In section 3.3.5 it is described how to tessellate a Bézier patch surface if given enough information about its corners. To implement this approach first the *cPolynomial*, the derived *cBezierBasis* and the *cBinomial* classes are constructed. These are used to compute and represent (the Bernstein bases of) equations 1 and 2.

As import of the terrain data has been handled in section 3.4.1 and the new classes above are able to handle the equations of section 3.3.5 all seems ready to implement the last part: the tessellation process. Unfortunately there are still three problems which have not yet been (fully) addressed: stopping the tessellation recursion, data reduction and cracks.

**Recursion**

Initially, the corners of each patch are computed. To compute additional points the above described central differencing algorithm is used recursively. To stop the recursion a few mechanisms are available. The most obvious and easiest is to limit the recursion depth. The second method is discussed in section 3.3.5. In essence it is using the last term of equation 9. As this is a measure for local curvature, it is possible to specify a threshold value. If this curvature term for a particular point is smaller than this threshold the point is not generated and recursion stops. Note that the value of this threshold determines the amount of deviation in the approximation: the threshold is a control for the balance between quality of the landscape and rendering speed by controlling the number of triangles in use.

**Data-reduction**

If the portion of the patch in question is very far away from the camera position, (the camera which generates a view on the virtual landscape) it is not necessary to use a lot of triangles for this patch, even if it has a high amount of local curvature (and thus is requesting lots of triangles). Furthermore, usually a large part of the terrain lies outside the view volume of the camera. The patches residing in this part need not be tessellated at

all, given the already large burden of computational costs. The data reduction has the following steps:

1. determine if the point we are about to compute will be visible. If so, the curvature term is computed and we proceed to step 2, otherwise stop.

2. if the curvature term is above the threshold proceed to step 3, if not stop;

3. if the size of the curvature vector (the vector from the midpoint to the actual surface point), computed and expressed relative to screensize is larger than a "visibility threshold", then proceed to step 4, otherwise stop.

4. tessellate using central differencing and do a recursion step;

As a consequence, portions far away (and thus occupying few pixels) are tessellated with lesser detail than the portions of the terrain closeby (which occupy a lot of pixels). A second consequence is that a recursion step is only done if the curvature is actually visible from the camera location. Both effects can be seen in figures 16 and 17.

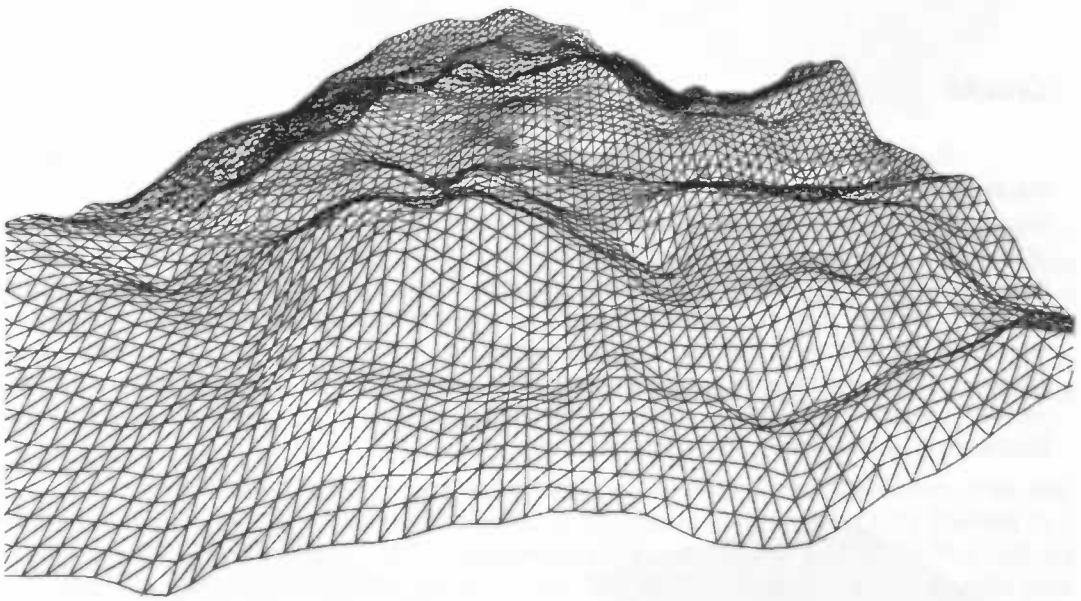


figure 16 - A landscape tessellated using uniform tessellation (8281 triangles)

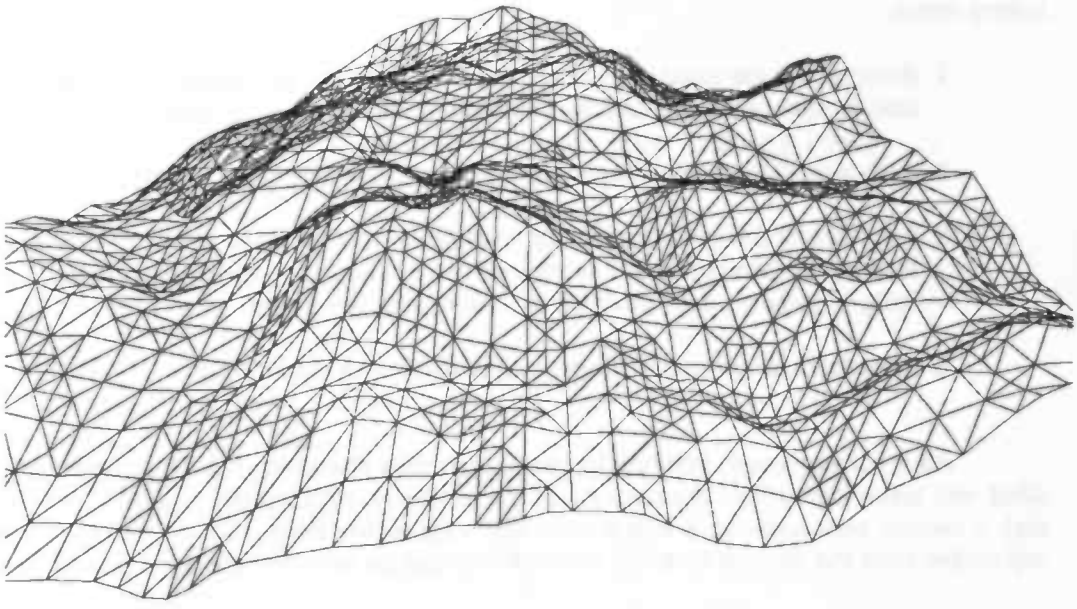


figure 17 - landscape tessellated using view based tessellation (2646 triangles)

## *Cracks*

Because different portions of the terrain are tessellated to different depths, the resulting vertices of neighbouring patches or squares (see below) may not coincide, resulting in the occurrence of cracks in the terrain (see figure 18). These cracks occur where two different depth squares are adjacent, because one of the joining edges has more vertices. This leads to a so called T-junction if the extra vertex is on the line between the previous and next point. Because of imprecise floating point arithmetic this may still lead to a small hole in the surface when drawing. If the extra vertex is not on the line between the previous and next point (which will most often be the case because of curvature) a permanent hole or crack is created because the two surfaces do not connect (completely) along their joining edges.

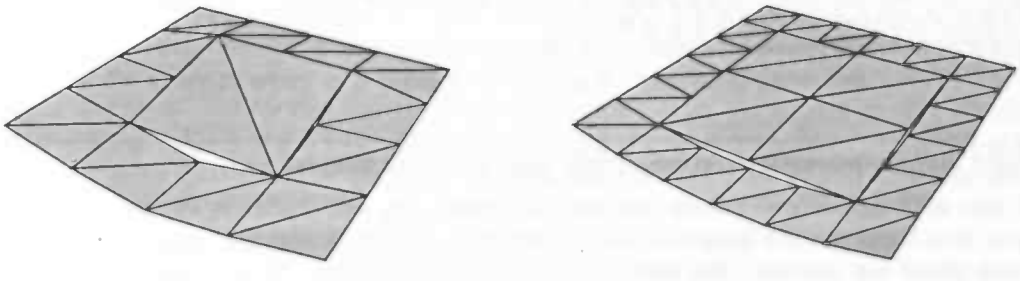


figure 18 - different levels of tessellation causing cracks

Cracks will not be filled by simply adding triangles. The cracks may have a wide variety of complex forms rendering the filling method too complicated and unpractical. A second disadvantage of this method is that it causes irregularities or discontinuities in the

surface (see figure 19). Instead the cracks which may appear will be limited to one form only. For this form a standard and simple method to close it can be used (see figure 20).

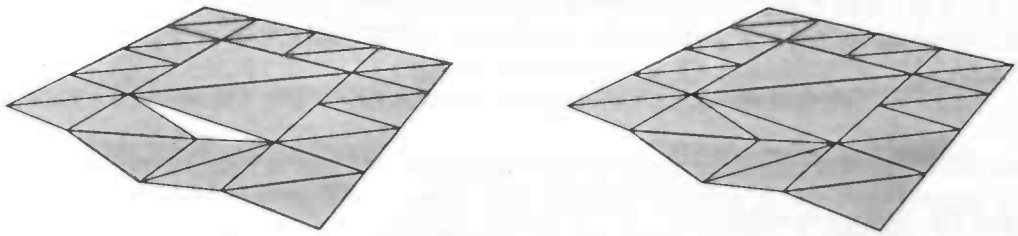


figure 19 - a crack is fixed by "filling" it, causing an irregularity in the terrain

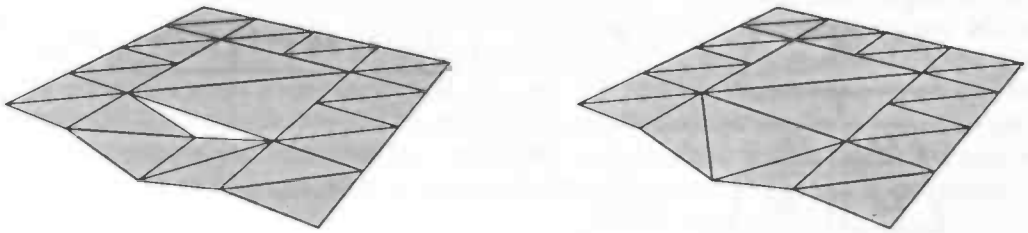


figure 20 - a crack is fixed by adjusting the bordering triangles

Instead of allowing the terrain to tessellate to the desired depth, the depth of the neighbours (up to four) will also be considered: recursion will only be allowed if the depth difference between two squares will not become larger than one. This will limit the form of cracks that can appear to be only of the form shown in the figure above. If a potential crack is created the new vertex causing the crack is marked. When a later recursion step causes the crack to disappear the same vertex will be unmarked. After vertex generation the cracks will be fixed if a marked vertex is found.

To do all this efficiently the patches will be tessellated simultaneously (this way cracks along patch edges can be addressed in the same manner as cracks internal to a patch). A queue (see section 3.4.4.1) is used to store records containing the four corners and the depth of a square. This queue is initialized with the corners of every patch (see figure 21a). One by one the records (or squares) are removed from the queue. When a square is tessellated to one depth level higher the square is subdivided, the four new squares are appended to the queue (see figure 21b) and the generated points are stored. This prevents any work being done twice: any square will only occur once in the queue. Also only the squares (and thus the triangles) that will actually be used are generated, so no time is lost in generating unused points. Tessellation is complete when the queue is empty.

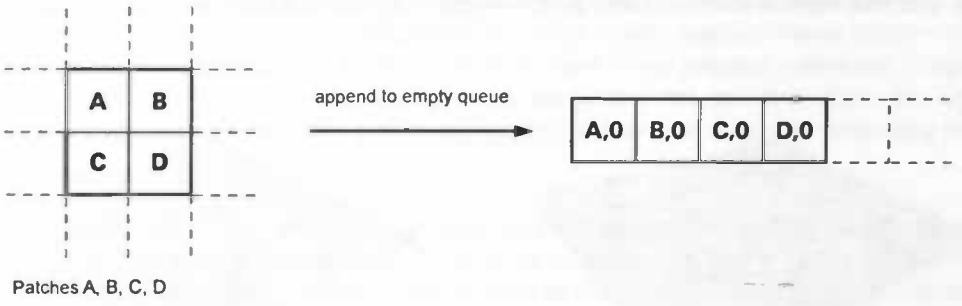



figure 21a - Queue initialization

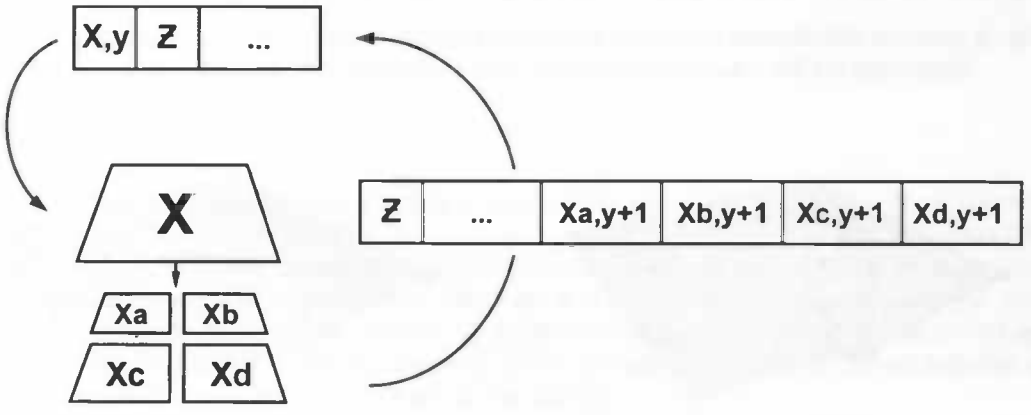



figure 21b - Recursion step

### *3.4.2 Interfacing 4Space*

When the tessellation routine stops it has emptied the queue and generated vertices. From these vertices a 4Space geometry node has to be built and subsequently 4Space can be asked to draw the node. This step will be dealt with in a very straightforward (inefficient) fashion since it is beyond the scope of this study's practical part.

For the generation of the geometry node a recursive approach is used. For each patch it is determined whether a centerpoint exists. If there is no generated center, two triangles are generated for this square. If the center does exist, the square (patch) is subdivided and the centers of the four smaller squares are checked. However if one of the sides is marked to be collapsed (for crack fixing) three triangles are created and recursion takes place only for two squares. The generated triangles are added to a queue. This way, for every patch a queue is generated. From these queues triangles are extracted and corresponding normals are computed. The triangles combined with normals are finally added to a *GeometryNode*. Any information 4Space requires, bounding volumes for example, is computed. The *Render* call to *cFSPatchTerrainNode* that initiated the creation of the tessellated terrain is passed on to the *GeometryNode* which is drawn, finally, by 4Space.

### 3.4.3 Memory management

One of the largest problems during implementation was the very high upper bound on main memory usage. In fact, if the maximum recursion depth is increased, this upper bound quickly becomes too large for even supercomputers to handle. For example, if we have a landscape of 16 by 16 patches and a maximum recursion depth of 8 the total amount of memory allocated for the storage of all vertices and additional information exceeds 1 Gigabyte, for 32 by 32 patches and a depth of 10 it jumps to 60 Gigabytes and even with a depth of 5 this landscape still requires more than 60 Megabytes.

While we have a very bad worst case the average case memory demands are of a few magnitudes smaller. This means that instead of using a two-dimensional array (which allocates room for all possible points) we have to take a different approach. Required is a storage system that can access data indexed by (x, y) tuples with a very fast access time. At the same time it should not allocate (a lot) more room than needed for the points actually used. Also it is not allowed to keep allocating large amounts of new memory if large parts of the currently allocated memory aren't used anymore.

Elaborating on the above: the storage system should only allocate room for index (a, b) if that position is actually used. Or put in an other way, the system should allocate room when position (a, b) is first accessed since generally before that moment it is not known whether or not data will be placed at (a, b). Furthermore, because of the large number of accesses done each frame both unused and used points need to have very fast accessibility. This means that use of multiplications, divisions, allocations and other time-consuming actions should be kept to a minimum in the access function. As a last requirement it not only must be possible to add points quickly but also to delete them quickly.

At first sight a solution would be to allocate a two-dimensional array of pointers. This would allow for a (very) fast and time-constant access function. If a pointer is NULL the point is not in use, if not NULL, the point is allocated and can be used. This would still require enormous amounts of memory just for the pointers and is thus infeasible. Hashing the indices would be another option but computing a hash function is costly. Also running out of space would require resizing the table. This is not a O(1) operation and would require more and more time after a few consecutive resizes.

The approach used here uses (red-black) trees to store indices. The top tree stores nodes that are indexed by x and store a reference to another tree which stores the y-indices for that particular x. The y-tree nodes reference the actual data. This provides an access function of O(lg n) (where n is number of nodes in the tree), but the average case does not cost more than a few comparisons. To prevent frequent allocation, nodes and data are stored in dynamic lists. Room in the lists is pre-allocated. Resizing is O(1) and consists of appending (pre-allocated) fragments retrieved from a pool. Only the pool needs to allocate new memory when needed.

Garbage collection or freeing up memory is an entire problem in itself and several solutions exist. A reasonable approach to start with would be to look at the nodes' valid values. These store the last frame number the nodes were used. If a node stores a very old frame number it generally is not very likely that this node will be used again anytime soon. So we can delete these nodes from the tree. The main problem with this approach is that we have to check all nodes every now and then. This will cause a longer frame time and if it takes too long a hiccup.

Three concepts need to be implemented: the red-black tree(s), the dynamic list and the pool. The lists consist of linked blocks or *fragments* of fixed size which store a

number of data-entries and some book-keeping to track usage. If the list runs out of space it can request a new fragment from an assigned fragment pool. The fragment pool stores a fixed number of references to allocated fragments. Upon request these references are handed out to the lists that are assigned to it. If there is no unused reference (fragment) available new fragments are allocated and all references are updated. Both the list and the pool are implemented using a C++ template class so the lists can store any kind of data.

Because two different types of data must be stored the red-black trees are also implemented using a template class. The implementation of this class is standard and straightforward except that a dynamic list is used to store the nodes. These structures and classes will be explained in detail in section 3.4.4.

### *3.4.4 Dynamic data storages*

In this section the various classes used to support the tessellation process by providing (dynamic) storage are discussed. For every class the use will be explained.

#### *3.4.4.1 Queue*

The *cQueue* class implements a queue. The queue can be used to store any kind of data. The tessellation process uses it to store squares while tessellating the terrain (see also section 3.4.1). By storing the indices of the first and last queue entries both *append* and *remove* are fast O(1) operations. Because a priori there is no knowledge of the maximum number of squares that need to be stored the queue may not be limited in storage space. This is accomplished by breaking up its storage in fragments. When needed an extra fragment is allocated and added to the queue.

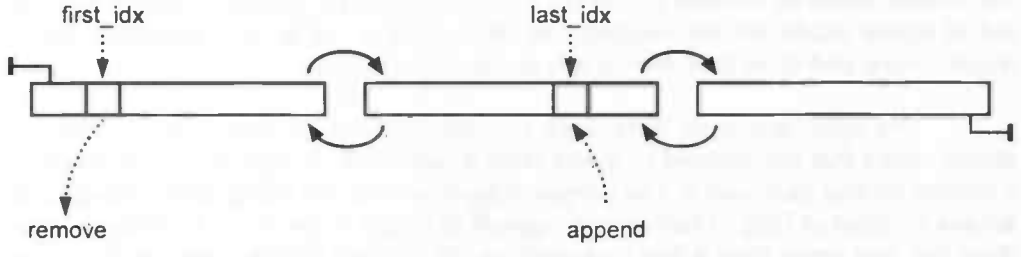



figure 22a - a cQueue consisting of three fragments

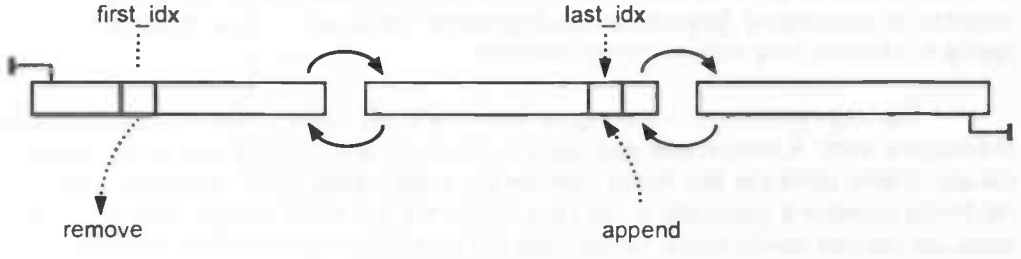



figure 22b - a Remove and an Append have been performed

### 3.4.4.2 Dynamic List & Pool

The *cDynamicList class* is mainly used to support the dynamic storage class described in section 3.4.4.4. In addition it is used to store triangles during the export of geometry to 4Space. In both cases accesses need to be fast and the maximum number of elements to be stored is unknown. To allow for both, the list is broken up in fragments (the *cDynListFragment class*), just like the queue above. Because many lists will be used simultaneously the lists are not allowed to allocate new fragments themselves. To keep allocation time to a minimum this is handed over to a pool, an instance of the cPool class, which keeps a number of unused fragments around for lists that may want them.

A fragment has a fixed length and is able to store a certain number of elements. For every fragment the index of the first free entry is stored. Combined with keeping track of the first fragment with at least one free entry this allows for fast O(1) addition to the list. Note that no ordering is enforced, elements are stored in the order they were added. Likewise, removal is O(1).

Access is not O(1) but is O(n) where n is the number of fragments in the list. Because access is done by index in the total available entries (not only the ones that are used) it is straightforward to compute the number of the fragment which holds the required element. This fragment is accessed by starting at the first fragment and taking the next fragment until the desired one has been reached. Now the element can be accessed by a simple array access.

Although both the *cPool* and the *cDynamicList class* have a destructor, the *cDynListFragment* class does not, because the only data members that (possibly) need destruction are elements that were added to the list the fragment belongs to. These elements are of unknown type, so it is left to the calling function or program to clean up its own data.

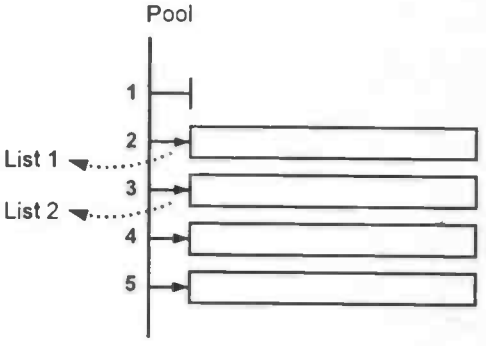
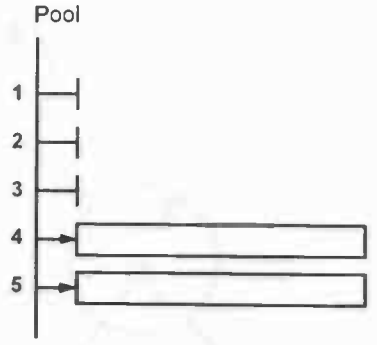



figure 23 - a Pool handing out two references, its first fragment has already been handed out




the first three fragments have been handed out

### 3.4.4.3 Red-Black Tree

This *cRBTree* class is used by the dynamic storage class described in section 3.4.4.4. It implements a standard red-black tree (see also Cormen, 1990) and its associated operations (insert, delete and access). The nodes of the tree are stored using a dynamic list (see section 3.4.4.2). The tree is able to store any kind of data in its nodes. The pool to be used by the dynamic list is passed on by the constructor of the tree. A second pool is used for the data members of the nodes. If a new node is inserted a pre-allo-

cated pointer is requested from this pool. Also, functions must be supplied to initialize or destroy the data member of the node when added to or removed from the tree. A tree access is O(n log m), where n is the number of fragments of the dynamic list and m the number of nodes.

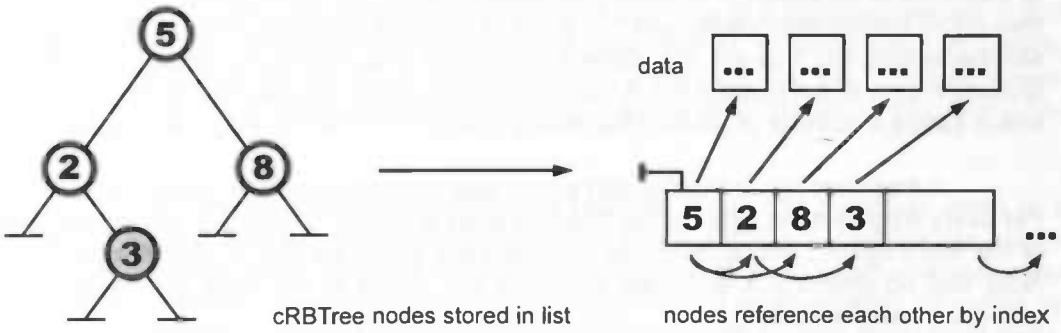



figure 24

### *3.4.4.4 Dynamic Storage*

This class uses both the *cRBTree* and *cDynamicList* classes to provide a dynamic storage. Any kind of data can be stored as long as it can be indexed by a (x, y) tuple with x and y being elements of *Z*. It is especially suited to store large two-dimensional sparse arrays memory-efficiently since only the entries containing data are actually stored. The *cBezierPatch* class uses a dynamic storage to store information for every generated vertex.

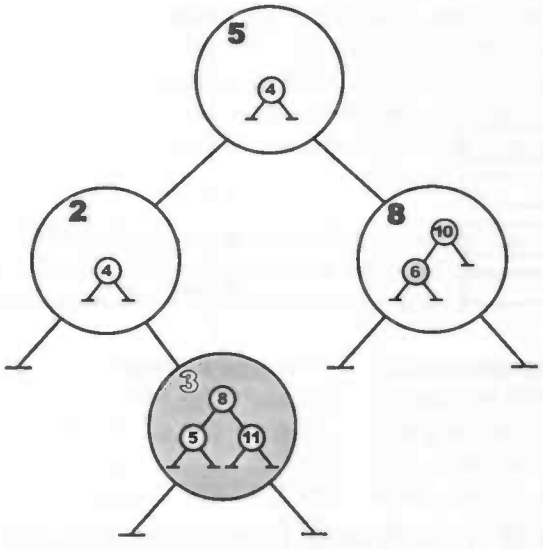



figure 25 - cDynStorage structure

The storage is constructed using a *cRBTree* to store x-values. In every node of this x-tree an y-tree is stored. The nodes of the y-trees store the data. Therefore a single access of a (x, y) position consists of two red-black tree accesses and is thus O(n log m).

### *3.4.4.5 Evaluating performance*

Since initially an implementation was done using arrays instead of the storages described above a comparison could be made. The implementation using dynamic storages proved to be somewhat slower than the one using arrays, which was not unexpected. Using the Intel VTune Performance Analyzer 4.0 hotspots (frequently called functions etc.) were found. These were inspected, minimized and optimized for speed. After this the differences between the two versions were marginal or very small.

# 4. Testing

## *4.1 Test setup*

To test the performance and scaling capabilities of the implementation a benchmark was constructed. This benchmark consists of four runs of a modified version of "MR terrain". For each run parameters are specified (see table 1), of which only the visibility threshold value (see section 3.4.2 "Data Reduction") is varied. For the curvature threshold (see section 3.4.2 "Recursion") a sufficiently small value is chosen to allow for a fine tessellation.

The terrain used consists of 16 by 16 patches. A maximal recursion depth of 8 allows for a fairly close approximation (a maximum of 256 x 256 x 2 triangles per patch). Combined with the chosen value for the nonlinear deviation threshold non-viewbased tessellation is infeasible since this would result in a geometry of approximately 30 million triangles. The visibility threshold is used to keep triangle numbers within acceptable limits. A fifth run was proposed but it generated in excess of 50.000 on-screen triangles and exceeded as such a 4Space limitation. Therefore run 5 was discontinued.

| Runnr | max. recursion depth | nonlinear deviation threshold | visibility threshold |
|---|---|---|---|
| 1 | 8 | 1.0 | 0.01 |
| 2 | 8 | 1.0 | 0.0025 |
| 3 | 8 | 1.0 | 0.000625 |
| 4 | 8 | 1.0 | 0.000156 |

table 1 - Benchmark run parameters

To ensure that any test-run is reproducible the camera travels a pre-destined path. Camera advancement is invoked by the framecounter and not by time. Therefore all benchmarked computers generate the same set of frames for the same run. Prior to testing (close to) optimal values for fragment and pool sizes were selected (experimentation revealed that changing sizes within reasonable values does not have a large impact on performance). Each run produces three files: the first contains global timing data, the second lists the number of triangles generated for every frame and the last lists all frame times.

This benchmark was executed on a number of different computers. A selection was made to be able to observe the effect of various possible performance bottlenecks. The selected computers can be found in table 2.

| **nr.** | **cpu** | **memory** | **hardware graphics pipe** | **Operating** | **System** |
|---|---|---|---|---|---|
| 1 | Pentium II 266 | 128 Mb | Nvidia TNT | Windows | NT 4.0 |
| 2a | Dual Pentium III 450 | 256 Mb | Intergraph XV25 | Windows | NT 4.0 |
| 2b | Dual Pentium III 450 | 256 Mb | Nvidia GeForce256 | Windows | NT 4.0 |
| 3a | Dual Pentium III 500 | 256 Mb | Matrox Millenium G200 | Windows | NT 4.0 |
| 3b | Dual Pentium III 500 | 256 Mb | Intergraph Wildcat 4000 | Windows | NT 4.0 |
| 4 | Dual Pentium III 750 | 512 Mb | Intergraph Wildcat 4110 VIO | Windows | NT 4.0 |
| 5 | Athlon 550 | 128 Mb | Nvidia TNT | Windows | 98 |
| 6 | Dual Pentium III 300 | 256 Mb | Intergraph Wildcat 4105 | Windows | NT 4.0 |

table 2 - The test computers

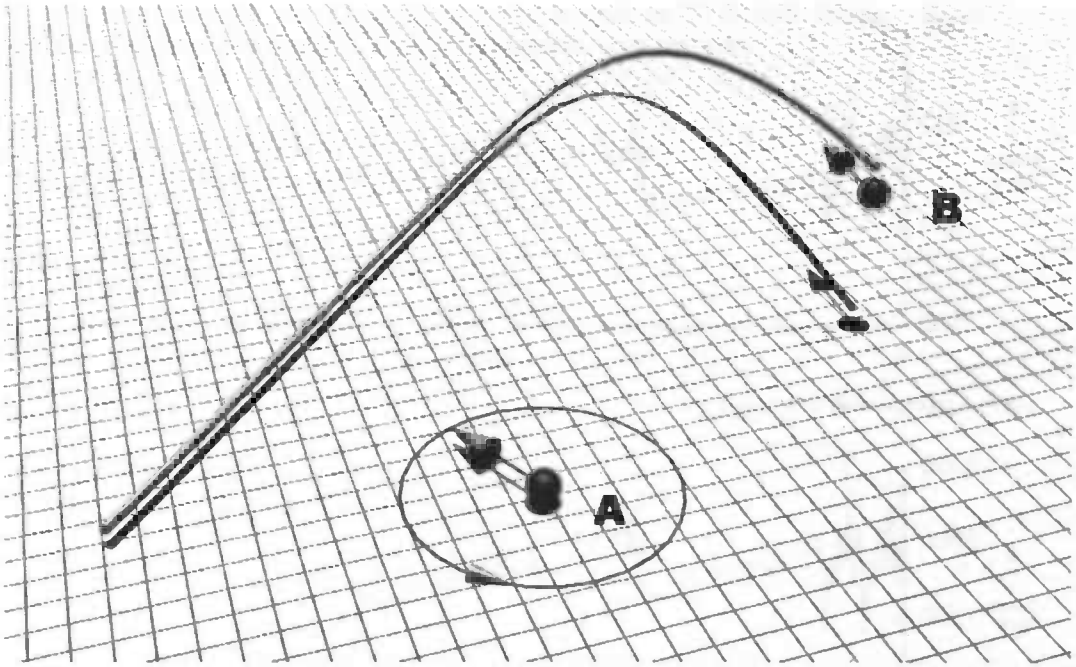
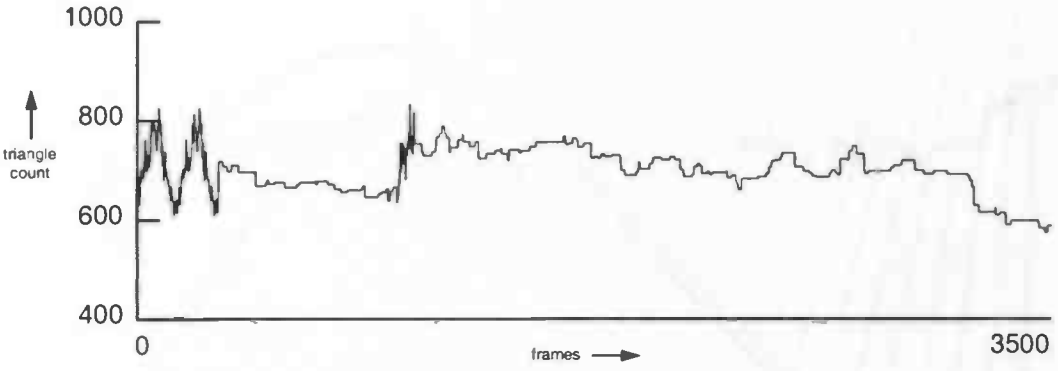


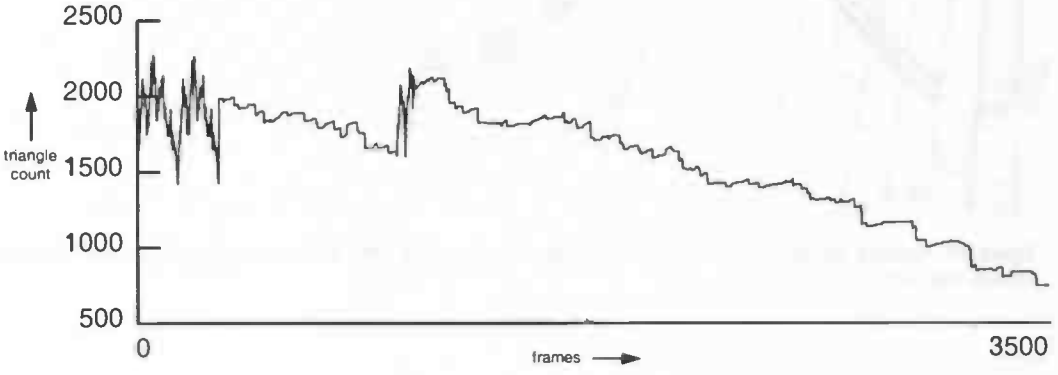

figure 26 - camera movement, A: the camera rotates two times, B: after a jump to a new location the camera follows the path

## *4.2 Results*

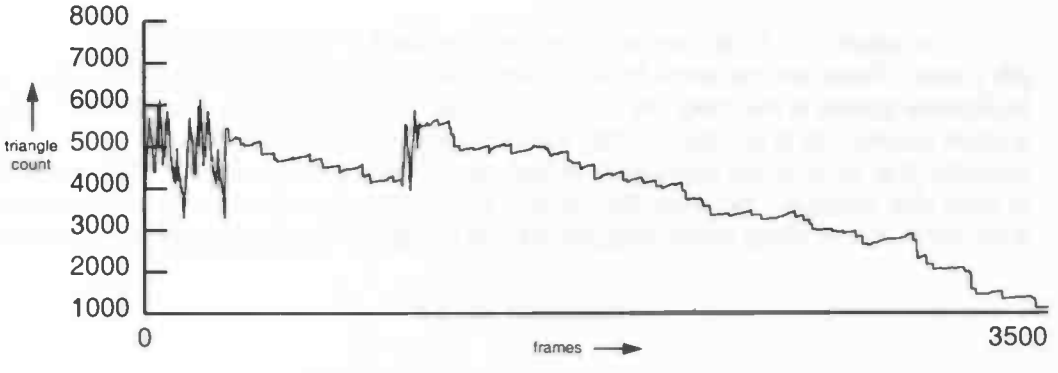
In graph 1 to 4 the triangle counts with respect to frame number of the four runs are shown. These are the same for every test computer. Something that can be observed from these graphs is that they are similarly shaped. Although triangle counts differ strongly (from around 700 to around 17,000), the basic shape of the four graphs is identical. This indicates that varying the visibility threshold value has a scaling effect. This is according to what was expected, because the visibility threshold is (unconditionally) used always if a recursion step is being done. This means that changing its value has effect everywhere.



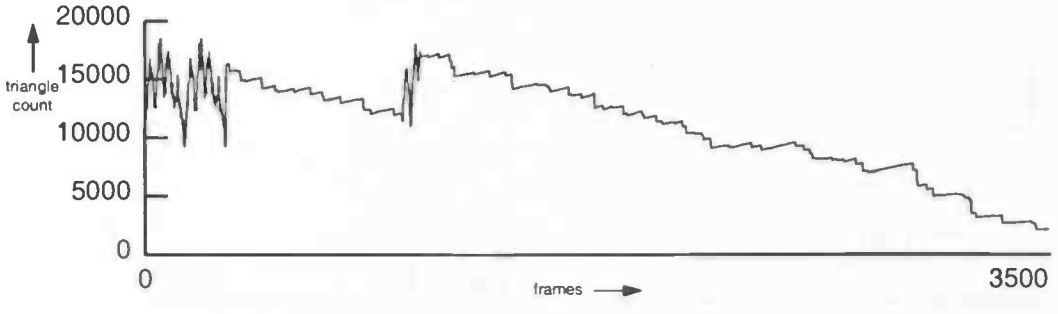

*graph 1 - triangle count of test run 1*




*graph 2 - triangle count of test run 2*

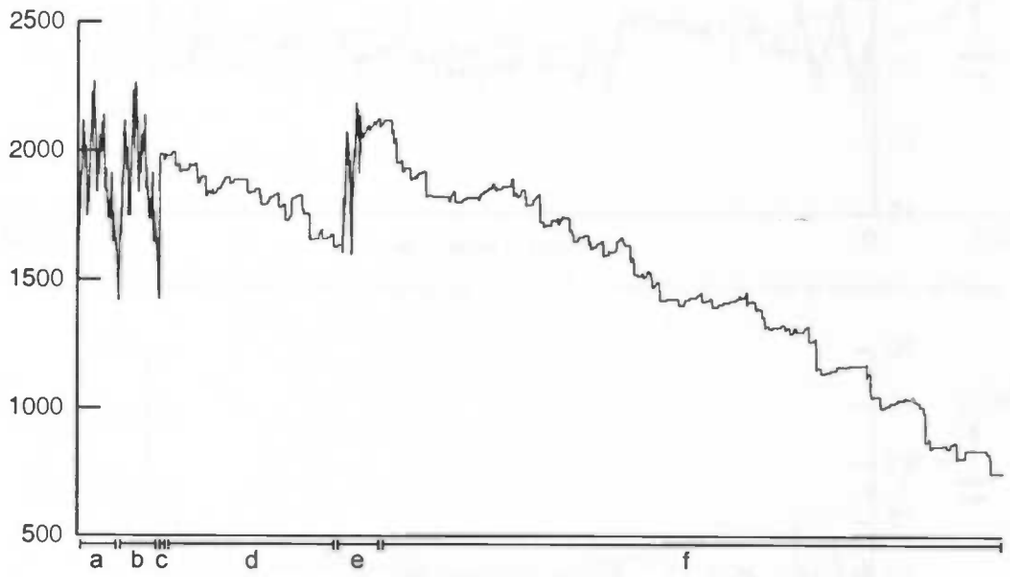



*graph 3 - triangle count of test run 3*




*graph 4 - triangle count of test run 4*

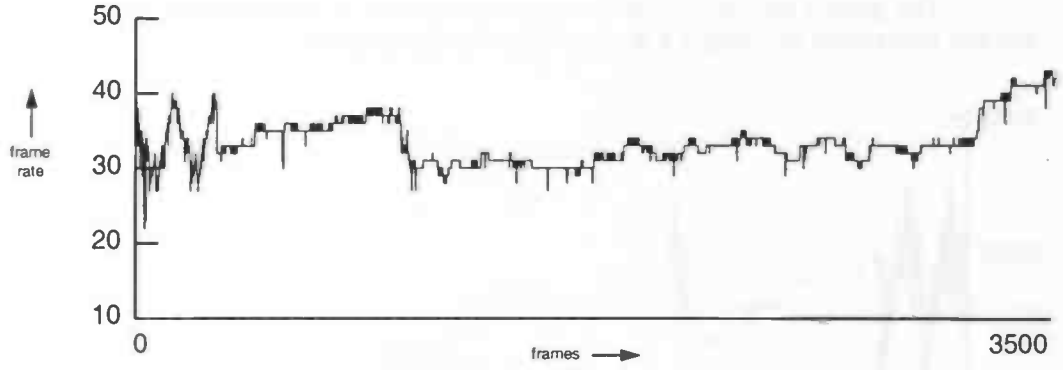
The graphs above have a few irregular regions. To explain these camera movement and graph 4 are combined to get graph 5:



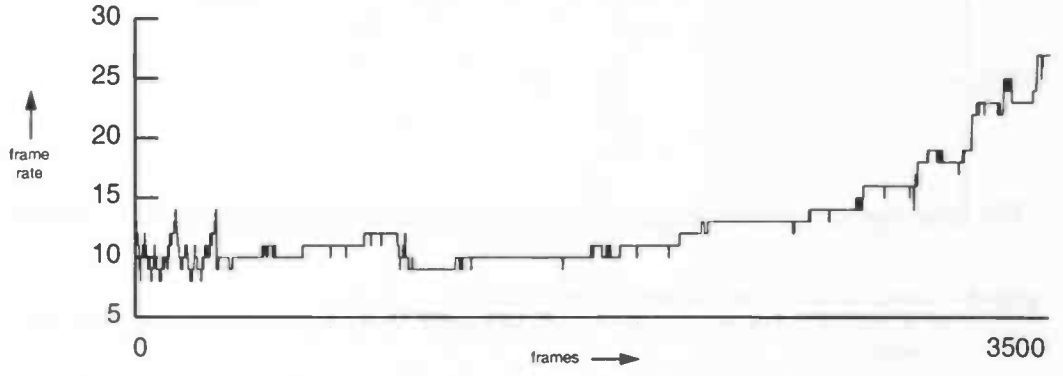

graph 5 - camera movement with respect to triangle count
(a, b: camera rotates 360 o, c: jump to another location, d: moving forward, e: rotate 150 o, f: moving forward)

Graph 5 shows that large changes in the triangle count result from significant changes of the camera view (note that a "free camera" is used: view angle is constant and there is no fixed look-at point). During the first two peaks the camera rotates itself, which changes the terrain in view from a small (camera is close to a slope) to a much larger piece and back. After this the triangle count suddenly jumps to a much higher value, this coincides with an instant jump of the camera to another location. From this location a large part of the terrain is visible. The camera is moving forward now, which shrinks the portion that is visible. This explains the slow but steady decrease in number of visible triangles. When the camera stops moving forward it makes another rotation (about 150 degrees) and descends. This causes the last peak in the graph. It then moves forward again while more and more of the terrain goes out of view.

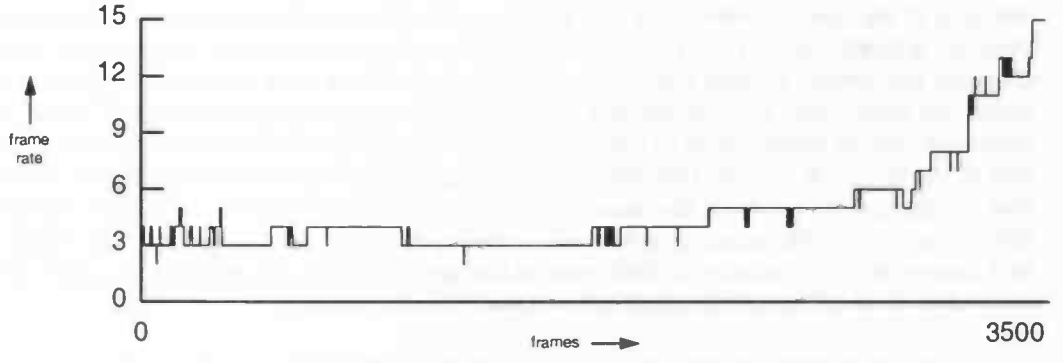
In the following graphs (6 - 13) the results of the tests are shown for each computer. The graphs show the frame rate during the test-runs.



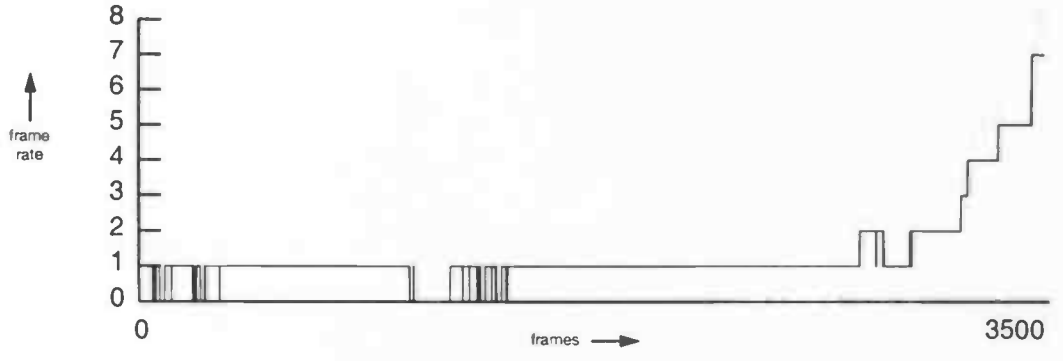

*graph 6a - framerate of test-run 1 on computer nr. 1*



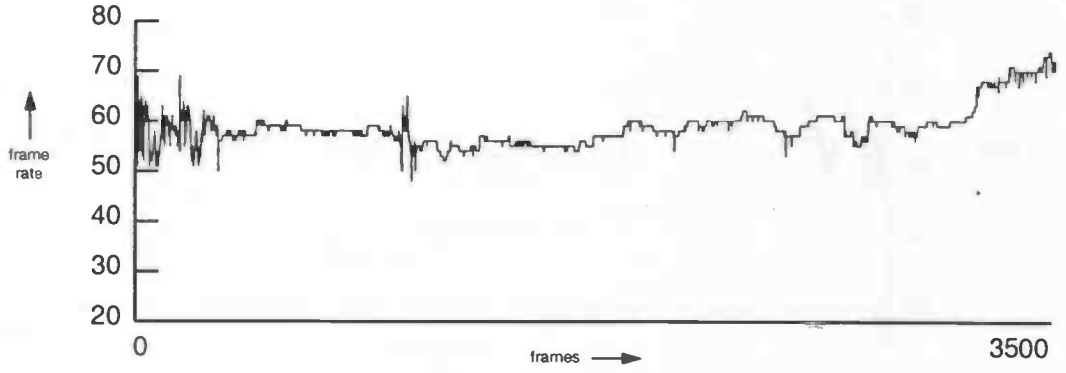

*graph 6b - framerate of test-run 2 on computer nr. 1*



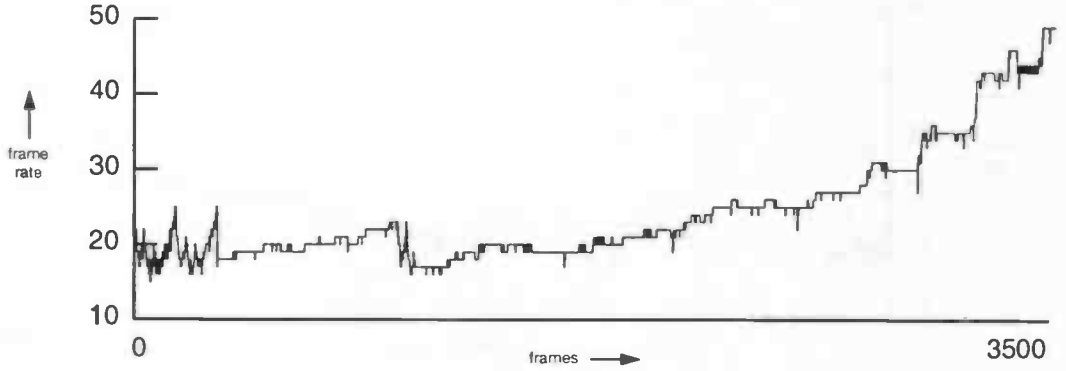

*graph 6c - framerate of test-run 3 on computer nr. 1*



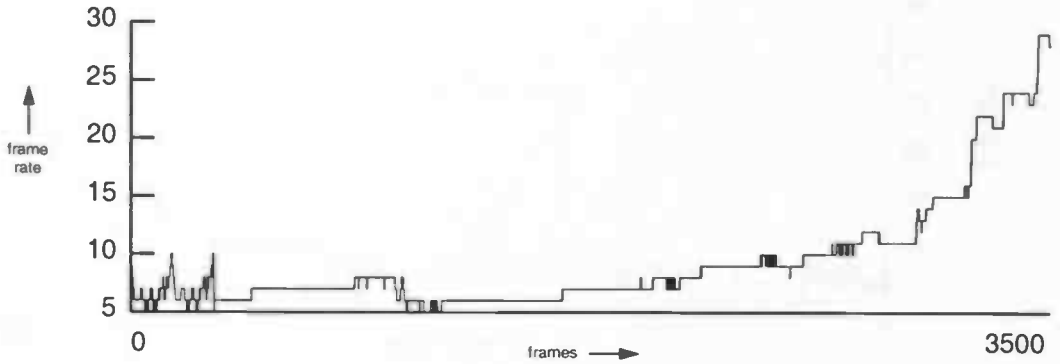

*graph 6d - framerate of test-run 4 on computer nr. 1*



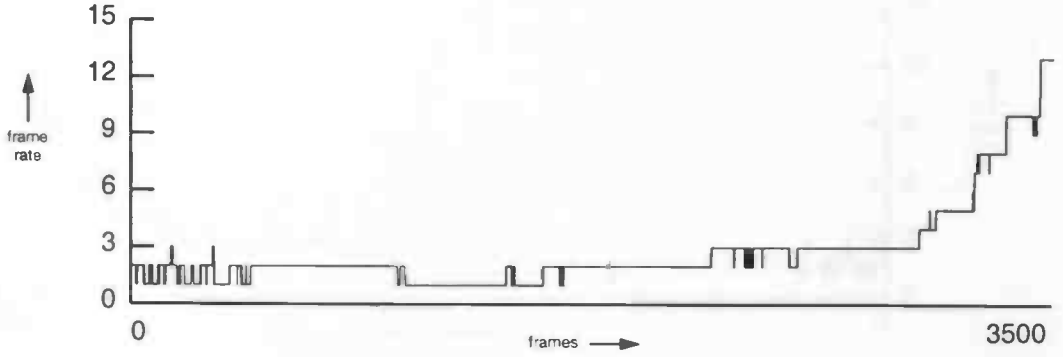

*graph 7a - framerate of test-run 1 on computer nr. 2a*



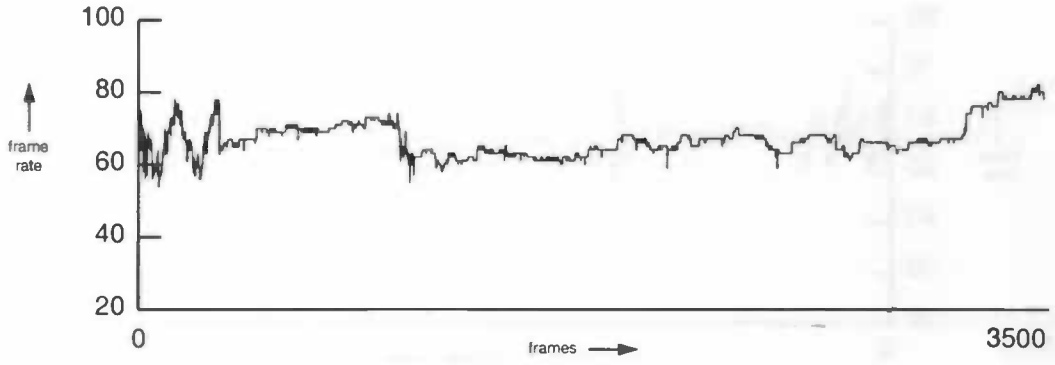

*graph 7b - framerate of test-run 2 on computer nr. 2a*



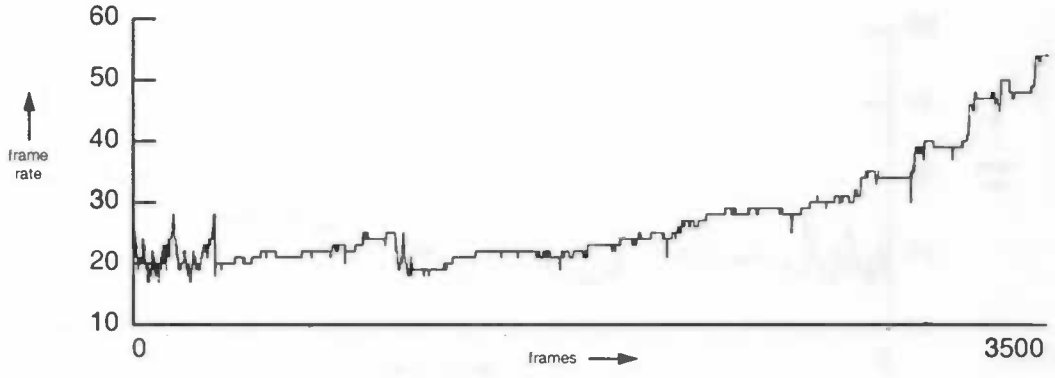

*graph 7c - framerate of test-run 3 on computer nr. 2a*



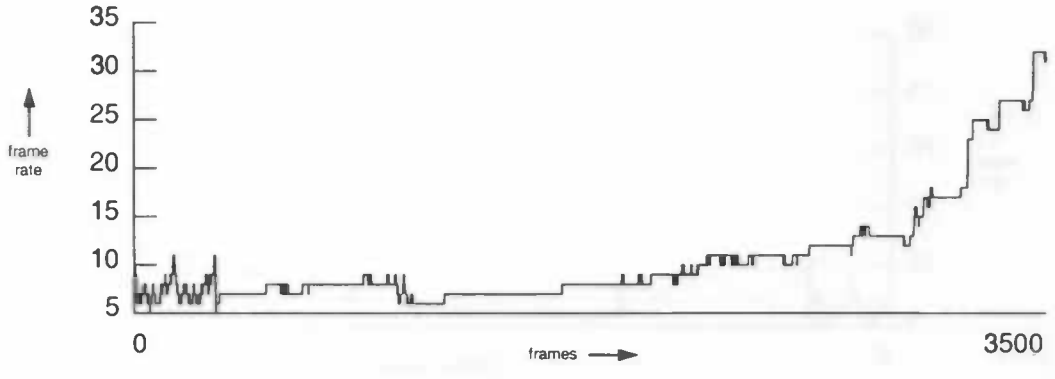

*graph 7d - framerate of test-run 4 on computer nr. 2a*



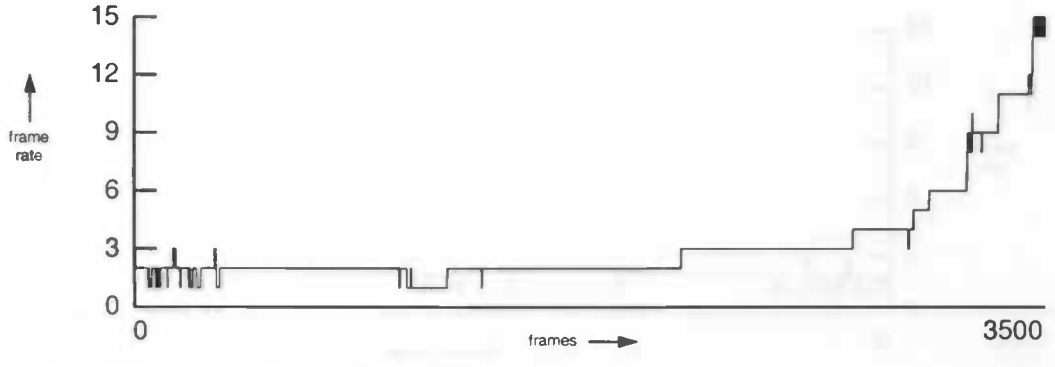

*graph 8a - framerate of test-run 1 on computer nr. 2b*



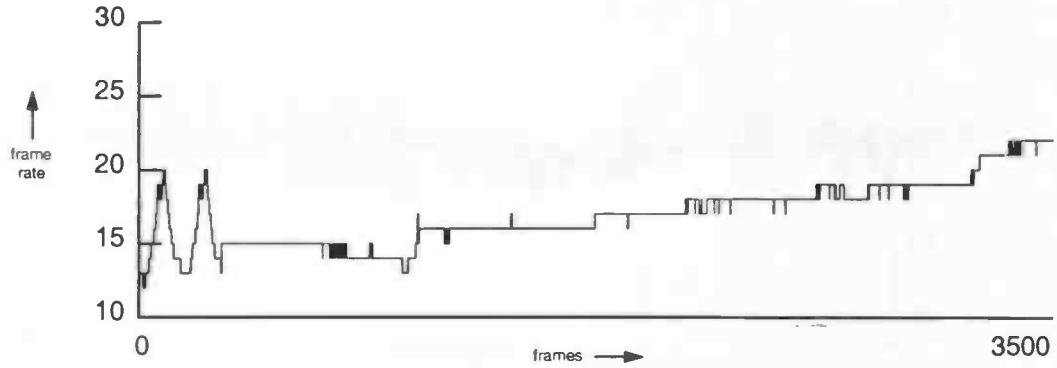

*graph 8b - framerate of test-run 2 on computer nr. 2b*



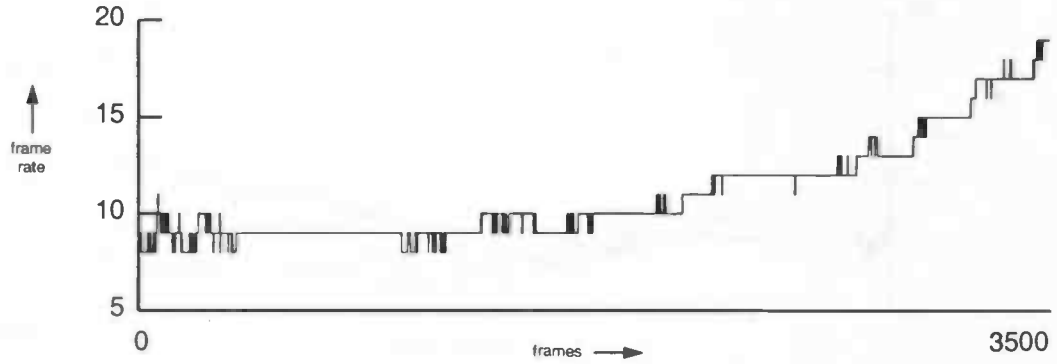

*graph 8c - framerate of test-run 3 on computer nr. 2b*



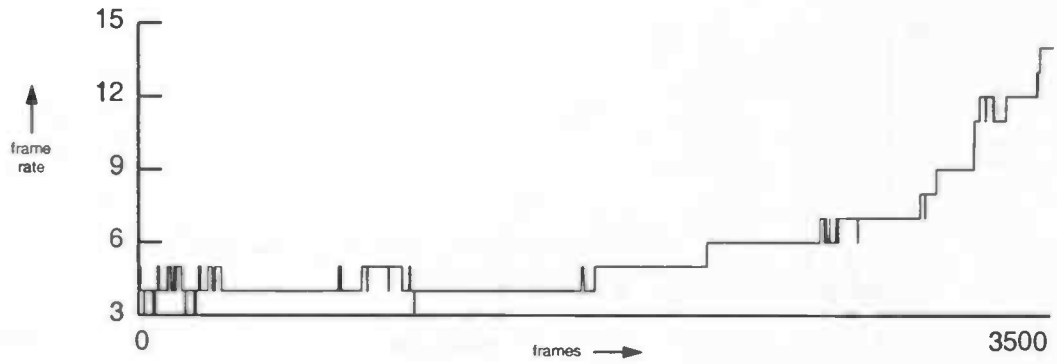

*graph 8d - framerate of test-run 4 on computer nr. 2b*



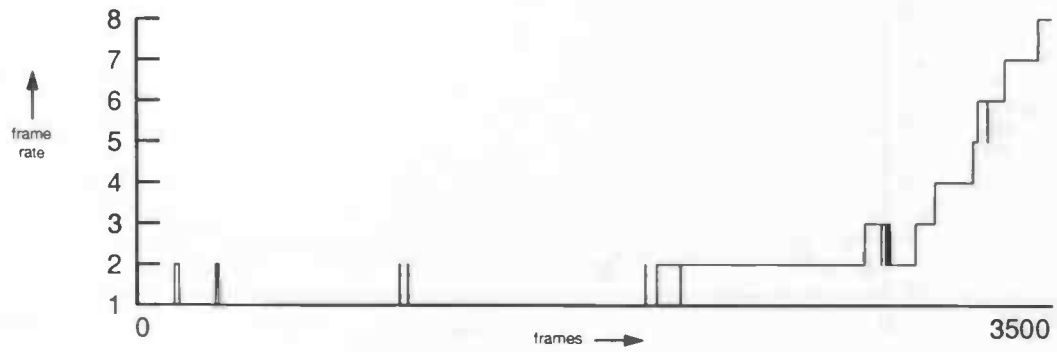

*graph 9a - framerate of test-run 1 on computer nr. 3a*



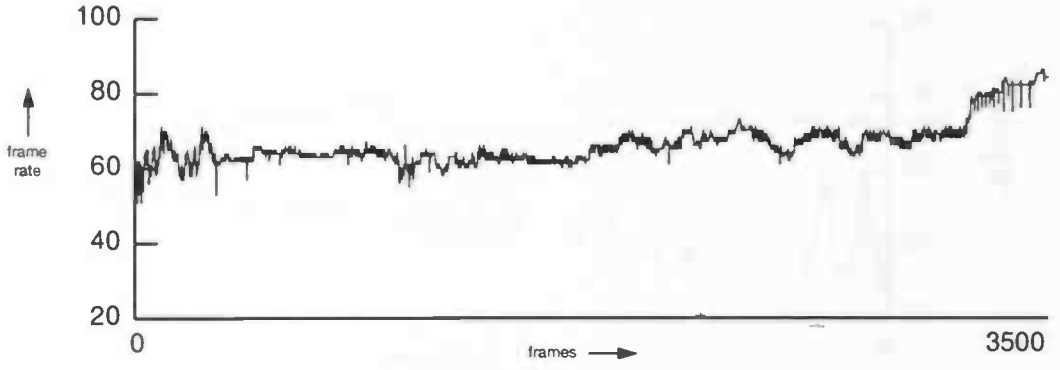

*graph 9b - framerate of test-run 2 on computer nr. 3a*



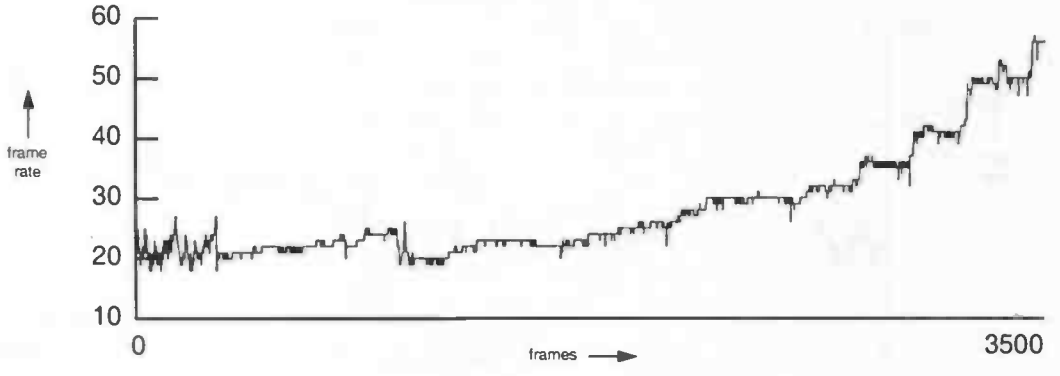

*graph 9c - framerate of test-run 3 on computer nr. 3a*



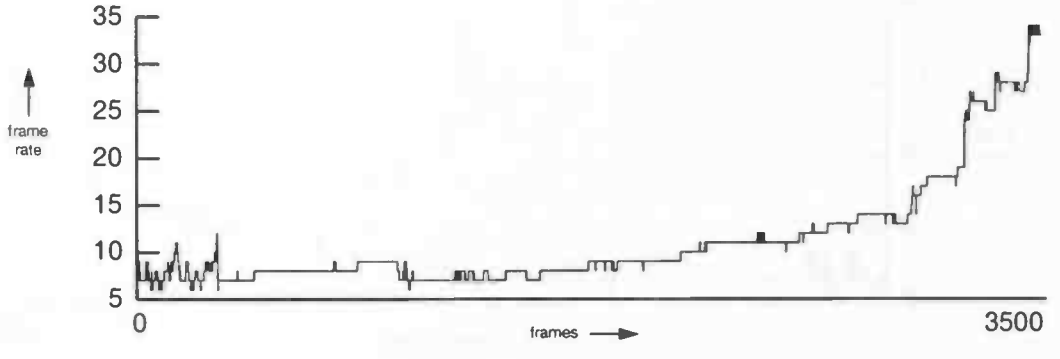

*graph 9d - framerate of test-run 4 on computer nr. 3a*



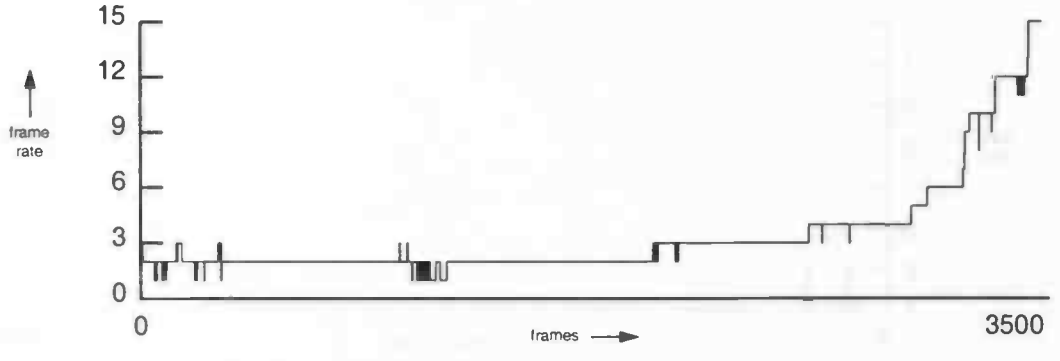

*graph 10a - framerate of test-run 1 on computer nr. 3b*



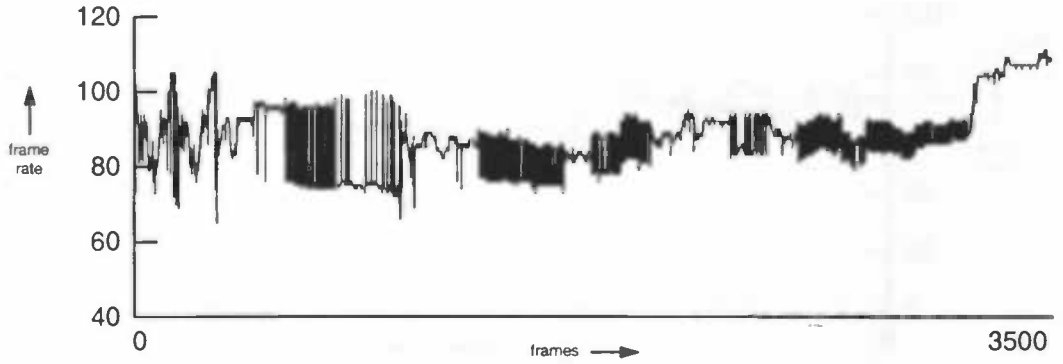

*graph 10b - framerate of test-run 2 on computer nr. 3b*



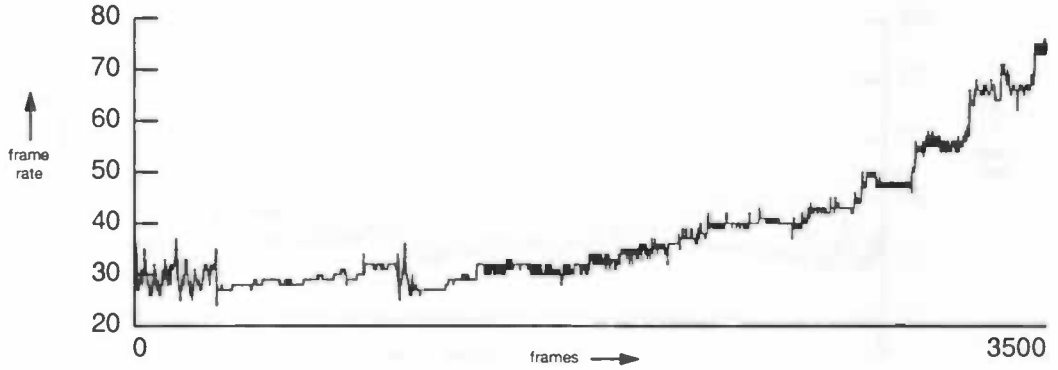

*graph 10c - framerate of test-run 3 on computer nr. 3b*



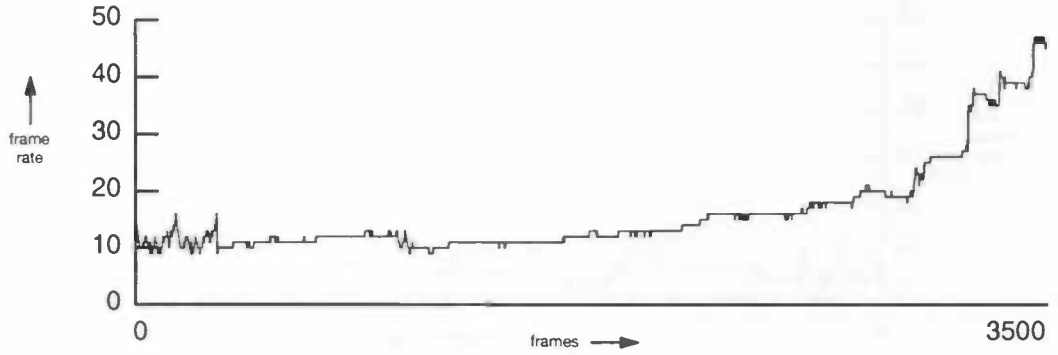

*graph 10d - framerate of test-run 4 on computer nr. 3b*



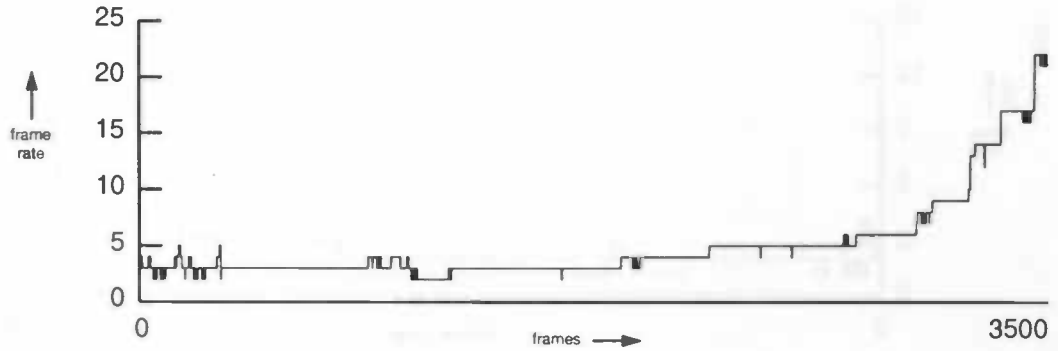

*graph 11a - framerate of test-run 1 on computer nr. 4*



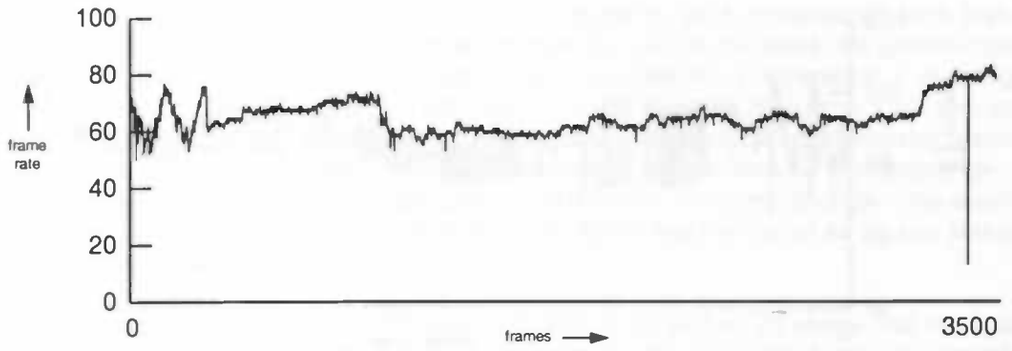

*graph 11b - framerate of test-run 2 on computer nr. 4*



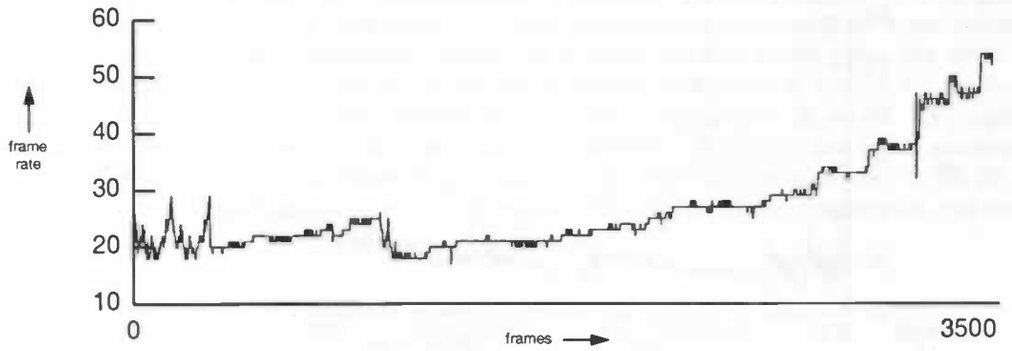

*graph 11c - framerate of test-run 3 on computer nr. 4*



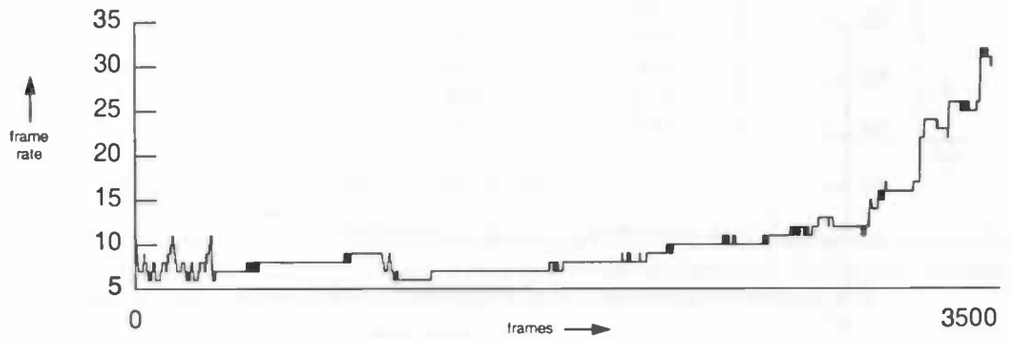

*graph 11d - framerate of test-run 4 on computer nr. 4*



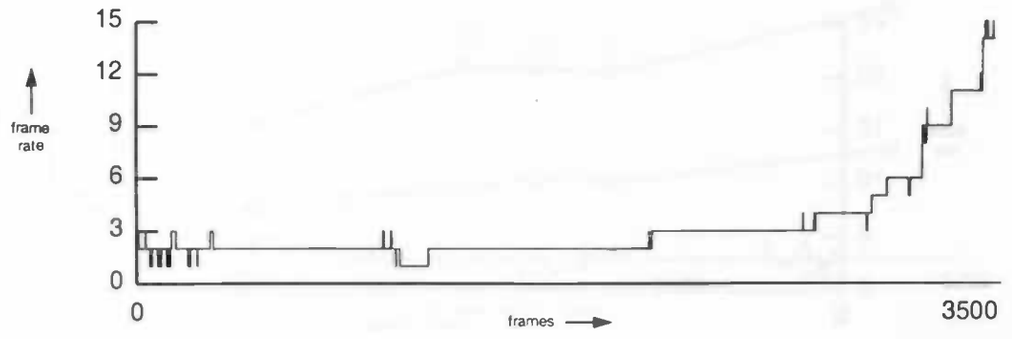

*graph 12a - framerate of test-run 1 on computer nr. 5*



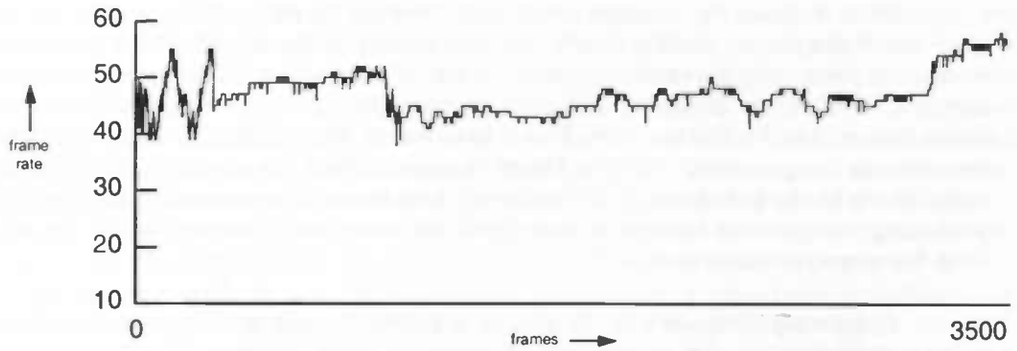

*graph 12b - framerate of test-run 2 on computer nr. 5*



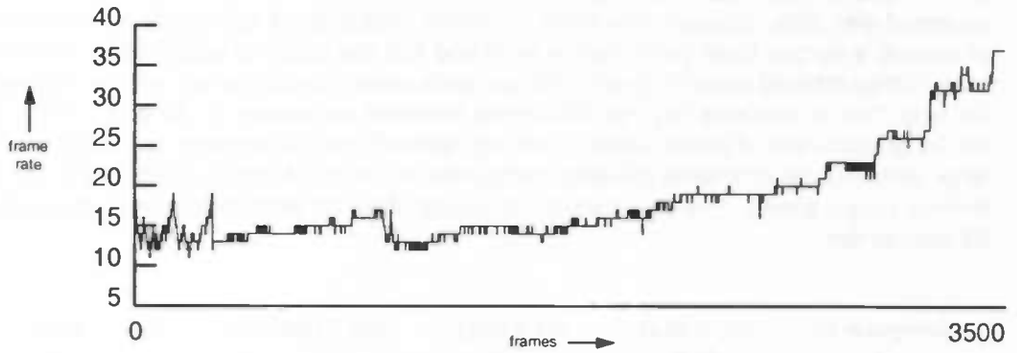

*graph 12c - framerate of test-run 3 on computer nr. 5*



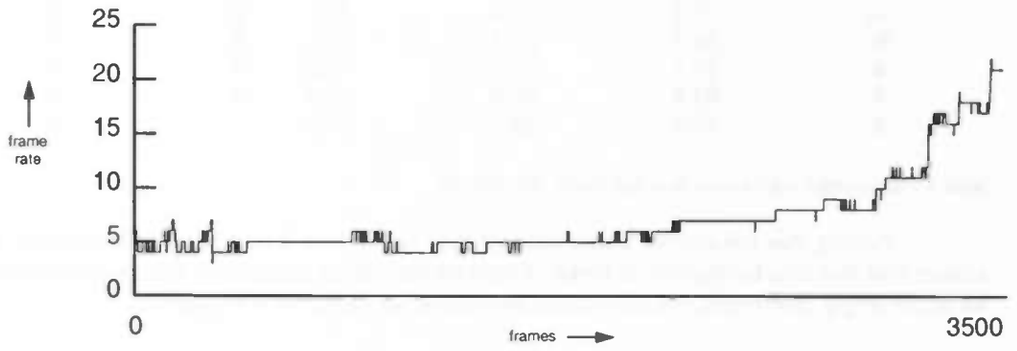

*graph 12d - framerate of test-run 4 on computer nr. 5*



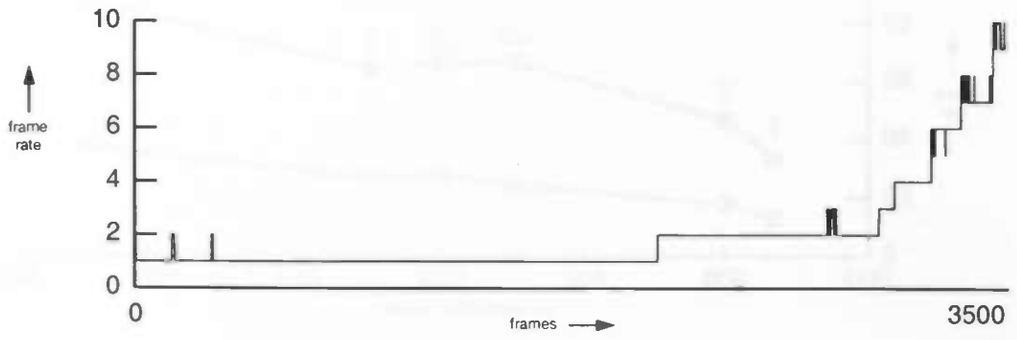

*graph 13a - framerate of test-run 1 on computer nr. 6*




*graph 13b - framerate of test-run 2 on computer nr. 6*




*graph 13c - framerate of test-run 3 on computer nr. 6*




*graph 13d - framerate of test-run 4 on computer nr. 6*

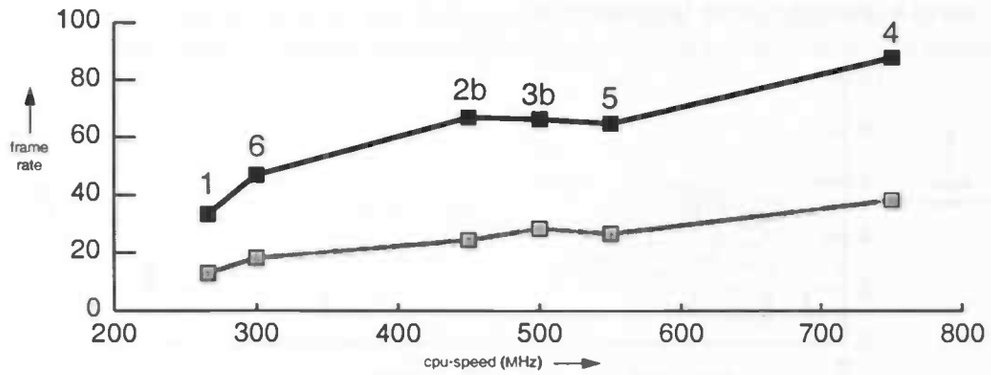
Table 3 shows the average frame rate achieved for each computer on each test-run. From these results and the graphs two bottlenecks can be identified: the general cpu processing power and the rendering power of the 3d accelerator. For instance, comparing computer nr. 1 and nr. 3a shows that although nr. 3a has a much faster cpu it still scores worse than nr. 1 in the first two runs. This is because nr. 1 contains a 3d accelerator which can process the generated triangles faster. However, when the amount of triangles generated starts to rise (run 3 and 4), computer nr. 3a is faster. The amount of cpu-time spent generating triangles has become a more dominant factor and the cpu of nr. 3a has about twice the speed as the one of nr. 1.

Comparing computers nr. 3b with nr. 5 and nr. 2a with nr. 2b shows that the cpu is the main bottleneck. If a certain level of rendering power is present, further increasing this power does not increase performance accordingly. Between computers nr. 2a, 2b, 3b and 5 there is little difference in performance. In contrast, the 3d accelerators they are equipped with differ strongly (see table 2). These accelerators apparently all accomplish or exceed a certain base performance level and it is the cpu that would make the difference (if they differed more in speed): the cpu limits system performance, not the 3D graphics card. This is confirmed by the differences between computers nr. 3b and 4. Although the 3d accelerators of these systems are not identical their differences are small, so any large performance difference between computers nr. 3b and 4 can be attributed to the difference in cpu-speed. This also means the results of nr. 3b were limited by its cpu, not its 3d accelerator.

| Computer nr. | run 1 (fps) | run 2 (fps) | run 3 (fps) | run | 4 | (fps) |
|---|---|---|---|---|---|---|
| 1 | 33.4 | 12.9 | 4.8 | 1 | . | 4 |
| 2a | 58.8 | 24.4 | 9.4 | 2 | . | 8 |
| 2b | 66.9 | 27.2 | 10.7 | 3 | . | 2 |
| 3a | 17.0 | 11.0 | 5.7 | 2 | . | 0 |
| 3b | 66.3 | 28.4 | 11.3 | 3 | . | 4 |
| 4 | 87.7 | 38.2 | 16.0 | 5 | . | 1 |
| 5 | 64.8 | 26.6 | 10.4 | 3 | . | 3 |
| 6 | 47.0 | 18.2 | 7.0 | 1 | . | 9 |

table 3 - Benchmark run results: average frame rate per run

Putting the measured framerates of test-run 1 and 2 in a graph (see graph 13) shows that the cpu-bottleneck is linear: if cpu performance increases, frame rate increases accordingly, permitted the 3d accelerator does not become a bottleneck.




graph 14 - frame rate in relation to cpu-speed (black: testrun 1, gray: testrun 2)

Graph 14 also shows a small drop in performance at some point. Although the computer (nr. 5) has a cpu with a higher clock speed it performs slightly worse than is expected. This can be attributed to a number of possible causes. This particular cpu is of another make and model than the others in graph 14. However since this type is known to be an excellent performer on floating-point operations, which are used quite heavily by the benchmark, this is not likely to be the cause of the low performance. More likely the 3d accelerator it is equipped with has begun to form a bottleneck. It is of the same model as used in computer nr. 1 and the slowest that is used (apart from nr. 3a, but that computer is not included in graph 14). Judging from the results of both computer nr. 1 and nr. 6 this 3d accelerator is a bottleneck, since nr. 6 shows a considerable performance increase with only a marginal difference of cpu-speed. Another possible cause is the operating system. All computers were running Windows NT 4.0 during the benchmark except nr. 5, on which only Windows 98 was available.

# 5. Remaining topics

The implementation discussed so far is not complete, the remaining topics will be discussed subsequently (see also section 3.3.4).

## *5.1 Functionality*

Although 4Space is now able to render a terrain constructed from Bézier patches, the extension is not yet really useful. Because the terrain is only flat-shaded it does not look realistic and there is a visually very distracting popping effect when detail levels change (caused by changing colors due to flat-shading when different triangles are created from one frame to the next). To remedy both problems the terrain should be smoothly shaded and textured. A simple form of texture mapping can be implemented quite easily if one texture per patch is used. In that case the P(u, v) coordinates can be used as texture coordinates. Rendering shades into vertex colors can also further increase visual realism of the terrain.

Another point of attention is terrain creation or import. Instead of using a random landscape it must be possible to render a specific (be it real or imaginary) terrain. Therefore the relationship (export and import of models) between the modeler in use (i.e. 3D Studio MAX), its features, and the MT module will have to be investigated. Terrain data sets are a second possible source of models (i.e. GIS subsets or DEM files). For both it will be necessary to develop a conversion to a format which 4Space can handle.

A simulation that uses the proposed and implemented landscape will require the presence of diverse objects: stationary, mobile or animated. Some issues which currently have not been considered then appear in the integration of landscape and objects. Two examples are the following. Where terrain and other scenery (-objects) collide visual and other problems may occur due to the varying spatial resolution of the various detail levels of the terrain. For instance: buildings on slopes may become (partly) buried in the terrain or floating in the air at different terrain detail levels. The second example is the requirement of tunnels in some environments. These problems can be resolved in two ways: the first is to create two fixed openings in the patchterrain and create the tunnel with another object. The second is to make the tunnel part of the terrain (which does imply the u, v coordinates of the patch control points are no longer in a plane).

## *5.2 Efficiency*

Chapter 4 showed that the current implementation partially achieves acceptable frame rates. However, only on the low(er) complexity runs. Fortunately there is ample room for optimization. Currently the following representations are used when the patches are rendered:

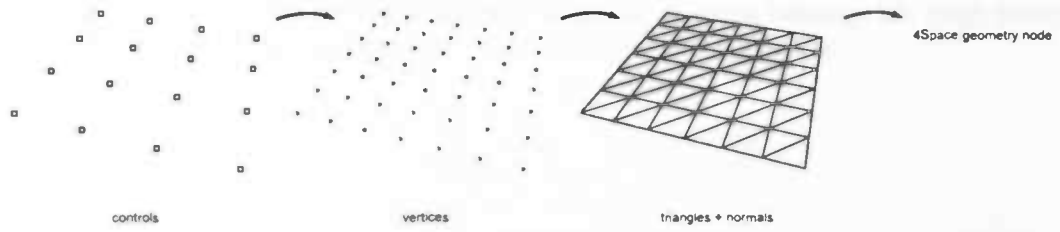



figure 27

The terrain is initially represented by its control points. From the controls tessellation creates a set of vertices. Triangles are created from the vertices and the triangles are inserted in a *GeometryNode* (see figure 27).

There are several ways to increase frame rate. First of all, the current implementation is rather simple and straightforward: each triangle is generated independently. No use is made of triangle strips or vertex sharing. In addition, it probably is possible to skip a full stage completely and go directly from vertices to 4Space geometry. Also a (special) operational mode is possible where instead of a general 4Space geometry node a more efficient form of geometry (i.e. vertex arrays) is used.

Another method to speed things up is to decrease the number of unnecessarily created triangles. If the distance is large enough it is conceivable that two triangles are enough for a 2 by 2 square of Bézier patches. In the current implementation any visible patch always gets at least two triangles. This means the initial steps should be about determining the curvature of multiple patches before a particular patch is tessellated. Furthermore, all patches in the camera view frustum are treated as visible and get tessellated, even if they are occluded or only their downside is visible. Using occlusion and the equivalent of backface culling to eliminate these patches to enter the tessellation process may result in a considerable performance improvement (especially in mountainous landscape environments). Finally, a situation is conceivable where the only terrain visible is just a (small) part of a single patch. In that case, just generating the visible triangles and not all triangles for that patch is more efficient.

Reusing vertex data is another means of preserving time. Instead of computing all vertices and geometry each frame it is conceivable to compute only those portions that were not computed during the previous frames. The remaining vertices can be reused from these frames if the terrain has not changed shape. Depending on the speed of the camera movement and amount of change in the terrain this should result in substantially less computational costs per frame.

Currently no effort is made to actively control the number of triangles created. If framerates are too low, it is conceivable to let the program autonomously adjust its parameters to restore a definable required frame rate. Also no effort is made to limit the variation in the number of triangles created. Due to local curvature a particular area may require far too many triangles while the rest of the terrain is optimal. Possibly a mechanism to put an upper (and lower) bound on the number of triangles can be added.

A lot of effort has already been put in making the implementation memory-efficient, but this topic still requires attention. First of all, memory that is no longer used must be freed (i.e. clearing storage entries). Some kind of garbage collection will be necessary to sustain long-term use of large terrains. Also there is redundancy in the system: since each patch independently stores its own data using its own *DynStorage*, the vertices of shared edges are stored twice (once for each patch). If one larger *DynStorage* is used for storing data for all patches this can be prevented. However, this does introduce problems concerning the derivatives because these are not necessarily the same for both patches sharing an edge.

Another optimization is to allow for differently sized patches in a terrain instead of using a uniform grid of patches. This allows for local high degree curvature using small patches while larger patches are used elsewhere. This prevents the use of high numbers of small patches where they are not necessary. However, the use of differently sized patches introduces non-trivial problems concerning continuity.

The last optimization that will be mentioned here, is to make use of more than one processor. Multi-cpu (SMP) computers, especially those with two cpu's are becoming more and more common. The current implementation uses one cpu, even if more are present. The algorithm presented here may be suitable for parallelization because of two aspects. First, the steps of the tessellation and geometry creation process are small and repetitive. Secondly, during most if not all steps only local data is used.

## *5.3 Loose ends*

There are some remaining issues: the integration with 4Space must be completed. Proper load and save functions for the 4Space FSO file format need to be added. Also some other functions need to be implemented correctly (like bounding volume/sphere computation). In addition there are limitations in 4Space that have not been dealt with yet (i.e. the number of polygons in a 4Space geoset is bound by a maximum). The implementation of the extension discussed so far could use a cleanup. This will remove shortcuts and limitations which are present due to time constraints (allowing for multiple instances of patch terrains, non-rectangular shape of the terrain etcetera).

At least for the following two issues corrections are necessary. The first is due to the tessellation method. If the terrain has a particular shape an aliasing effect can occur. The tessellation stops early and causes the generated triangles to deviate far too much from the actual shape (see figure 28). Since the terrain is two-dimensional this does not happen as often as in the case of a one-dimensional curve because the three other edges of a square/patch can force subdivision of such an edge.

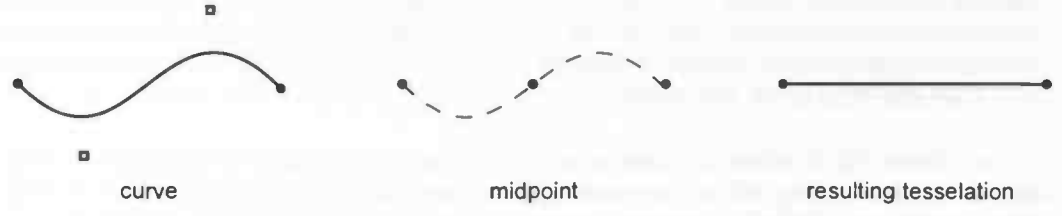



figure 27

The second issue is an artificial limitation which should be removed. Currently each patch is always tessellated to at least depth 0 (which results in two triangles). This causes two problems. As the terrain grows in size (the number of patches increases) the number of unnecessary created triangles grows and eventually the system will choke on them, limiting the size of the terrain and slowing down the visualization. Secondly, because each patch is tessellated to depth 0, the patches that are visible border on edges of depth 0. This results in an odd visual effect in which nearby terrain loses detail because of the crack fixing routine. It creates a border area in which patches are every time tessellated one step further than their neighbours to build a bridge between the (high detail) part of the terrain in view and the (low detail) rest (see also figure 29).

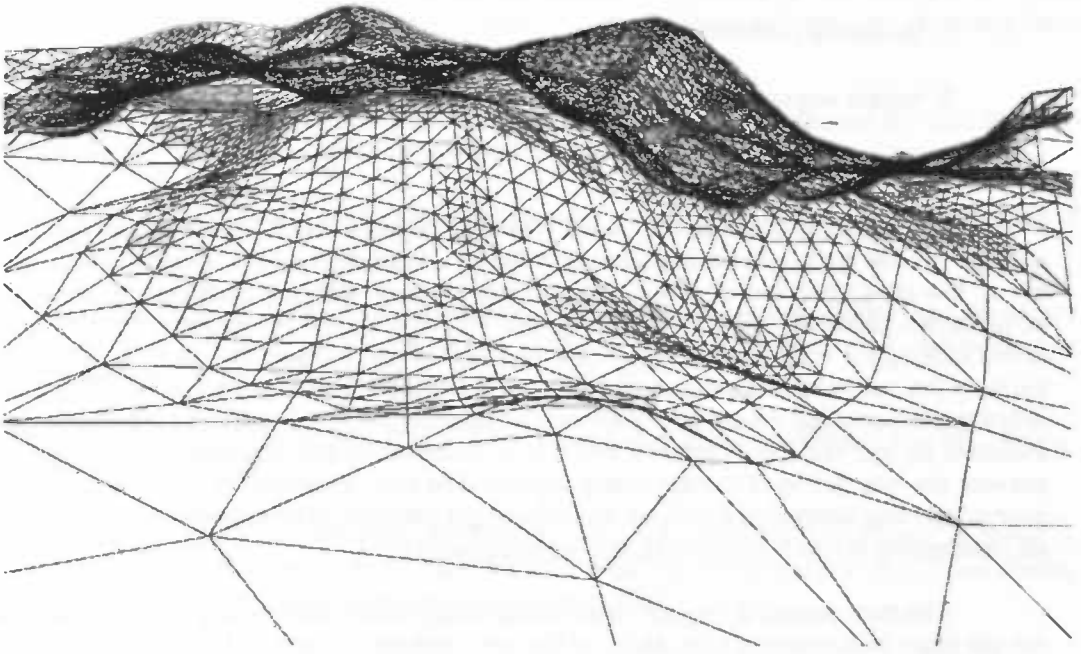


figure 29 - Landscape tessellated with a very small visibility threshold to emphasize the loss of detail near the edges of the view volume.

## 6. Conclusion

The goal of this study was the implementation of a scalable method for visualizing large terrains. Additionally a front-end was created in a way such that benchmark testing would be possible to mark system-component effects as well as scalability effects. This scalability has three aspects:

1. hardware performance

2. terrain approximation quality (detail level)

3. terrain dimension

The implemented algorithm combines the first two aspects in such a way that from a single source terrain description many different visualizations with respect to the number of triangles used are possible. This allows for very different hardware to show the same terrain, albeit at varying levels of quality. This was shown in chapter 4. Performance is still lacking for both low-end hardware and high-detail parameter settings, but implementing the optimizations proposed in chapter 5 should improve this situation. Also in the entire implementation no specific platform is assumed. Although 4Space is currently only available for the Windows platform, there is no limitation in the implementation which will prevent the retargeting of the extended 4Space. The only requirement is that a C++ compiler supporting templates exists for the new target platform (if templates are not supported, retargeting is much more work but not impossible).

The third aspect is explained as the capability of supporting large terrains and has not yet been implemented fully. Much of the work has been done but additional programming is required. Firstly an optimization is required for the tessellation of invisible patches (see section 5.3), a procedure that is fairly straightforward. A second addition is more complex and consists of the implementation of a garbage collector (see section 5.2).

The goal has thus only been partly achieved. And even if the additional work described above is finished, there is still a lot of work to do. Bringing the implementation up to production standards concerning speed and features will require implementing the bulk of the proposed additions and optimizations (see chapter 5) which will require considerable time. But, the current results and the kind of optimizations possible indicate that if this work is finished there could be a significant increase of speed. The proposed additions should cause the cpu to be a less dominant factor regarding performance while visual quality can be raised considerably (for instance, the last two test-runs could then be running at acceptable frame rates).

Looking ahead, the technology is there to visualize large surfaces described by patches while maintaining a good standard of visual quality concerning geometric shape. Real benefits of this technology are that the visual quality of the terrain or surface can be raised quickly if hardware capabilities keep increasing and that visual quality can be maintained at any zoom level. It is a point of further study whether this scalability will be maintained if the terrain is textured fully and in detail. It is also worth investigating if the technique used here can be applied more generally and be used for other objects. Especially interesting is the case of objects which change shape frequently (i.e. a waving flag or a walking person) for which, if they are constructed from Bézier patches, only the controls have to be adjusted and not their entire geometry.

# A1 Literature


ARPA, "WRM Entity Flight Specification", version 1 draft 10, 1994

M.A. Bassiouni, M. Chiu, "Performance and Reliability Analysis of Relevance Filtering for Scalable Distributed Interactive Simulation", ACM Transactions on Modeling and Computer Simulation, Vol. 7, No. 3, pp.293 - 331, 1997

S.L. Berg, S.H. Grigsby, "Improving the fidelity of distributed simulations through environmental effects", 13th Workshop on Standards for DIS, 1995

B.S. Blau, "Object Oriented Terrain Databases for Visual Simulators", thesis, University of Central Florida, 1990

W. Blumenow, G. Spanellis, B. Dwolatzky, "The Process Agent Model and Message Passing in a Distributed Processing VR System", ACM Symposium on Virtual Reality Software and Technology, Lausanne Switzerland, 1997

T.H. Cormen, C.E. Leiserson, R.L. Rivest, "Introduction to Algorithms", The MIT Press, Second printing 1990

T.P. Das, G. Singh, A. Mitchell, P. Senthil Kumar, K. McGee, "NetEffect: A Network Architecture for Large-scale Multi-user Virtual Worlds", ACM Symposium on Virtual Reality Software and Technology, Lausanne Switzerland, 1997

S.R. Ellis, "Nature and origins of virtual environments: a bibliographical essay", Computing Systems in Engineering, Vol. 2, No. 4, pp. 321 - 347, 1991

S.R. Ellis, "What are virtual environments?", IEEE Computer Graphics & Applications, 1994

M. Froumentin, E. Varlet, "Dynamic implicit surface tesselation", ACM Symposium on Virtual Reality Software and Technology, Lausanne Switzerland, 1997

D. Fullford, "Distributed Interactive Simulation: It's Past, Present, and Future", Proceedings to the 1996 Winter Simulation Conference, Arlington USA, 1996

S. Gumhold, W. Straßer, "Real Time Compression of Triangle Mesh Connectivity", SIGGRAPH 98 Conference Proceedings, pp. 133 - 140, 1998

L. Kobbelt, S. Campagna, J. Vorsatz, H. Seidel, "Interactive Multi-Resolution Modeling on Arbitrary Meshes", SIGGRAPH 98 Conference Proceedings, pp. 105 - 114, 1998

E. Lantz, "The Future of Virtual Reality: Head Mounted Displays Versus Spatially Immersive Displays", SIGGRAPH 96 Conference Proceedings, pp. 485 - 486, 1996

P. Lindstrom, D. Koller, W. Ribarsky, L.F. Hodges, A. Op den Bosch, N. Faust, "An Integrated Global GIS and Visual Simulation System", GVU Technical Report 97-07, Georgia Institute of Technology, 1997

P. Lindstrom, D. Koller, W. Ribarsky, L.F. Hodges, N. Faust, G.A. Turner, "Real-Time, Continuous Level of Detail Rendering of Height Fields", SIGGRAPH 96 Conference

Proceedings, pp. 109 - 118, 1996

P. Lindstrom, D. Koller, W. Ribarsky, L.F. Hodges, N. Faust, G.A. Turner, "Level-of-Detail Management for Real-time Rendering of Phototextured Terrain", GVU Technical Report 95-06, 1995

R. Macredie, S.J.E. Taylor, X. Yu, R. Keeble, "Virtual Reality and Simulation: an overview", Proceedings of the 1996 Winter Simulation Conference, 1996

W.R. Mark, S.C. Randolph, M. Finch, J.M. Van Verth, R.M. Taylor II, "Adding Force Feedback to Graphics Systems: Issues and Solutions", SIGGRAPH 96 Conference Proceedings, pp. 447 - 452, 1996

E. Mascarenhas, V. Rego, J. Sang, "DISplay: A system for visual-interaction in distributed simulations", Proceedings of the 1995 Winter Simulation Conference, 1995

S. Nishimura, T.L. Kunii, "VC-1: A Scalable Graphics Computer with Virtual Local Frame Buffers", SIGGRAPH 96 Conference Proceedings, pp. 365 - 372, 1996

J. O'Rourke, "Computational Geometry Column 33", SIGACT News 29(2), Issue #107 pp. 14-16, 1998

I. Oswalt, "Technology trends in military simulation", Proceedings of the 1995 Winter Simulation Conference, 1995

I. Poupyrev, S. Weghorst, M. Billinghurst, T. Ichikawa, "A Framework and Testbed for Studying Manipulation Techniques for Immersive VR", ACM Symposium on Virtual Reality Software and Technology, Lausanne Switzerland, 1997

E. Puppo, R. Scopigno, "Simplification, LOD and Multiresolution Principles and Applications", Eurographics '97 Vol. 16 No. 3, 1997

P.F. Reynolds, JR. Natrajan, A. Natrajan, "Consistency Maintenance in Multiresolution Simulations", ACM Transactions on Modeling and Computer Simulation, Vol. 7, No. 3, pp 368 - 392, 1997

G.A. Schavione, S. Sureshchandran, K.C. Hardis, "Terrain Database Interoperability Issues in Training with Distributed Interactive Simulation", ACM Transactions on Modeling and Computer Simulation, Vol. 7, No. 3, pp. 332 - 367, 1997

D. Schmalstieg, M. Gervautz, "Modeling and Rendering of Outdoor Scenes for Distributed Virtual Environments", ACM Symposium on Virtual Reality Software and Technology, Lausanne Switzerland, 1997

Brian Sharp, "Implementing Curved Surface Geometry", Game Developer, Vol. 6, No. 6, pp. 42 - 53, 1999

Brian Sharp, "Optimizing Curved Surface Geometry", Game Developer, Vol. 6, No. 7, pp. 40 - 48, 1999

G. Smith, J. Mariani, "Using Subjective Views to Enhance 3D Applications", ACM Symposium on Virtual Reality Software and Technology, Lausanne Switzerland, 1997

A. Steed, "Efficient Navigation Around Complex Virtual Environments", ACM Symposium on Virtual Reality Software and Technology, Lausanne Switzerland, 1997

O. Sudarsky, C. Gotsman, "Output-Sensitive Rendering and Communication in Dynamic Virtual Environment", ACM Symposium on Virtual Reality Software and Technology, Lausanne Switzerland, 1997

C. Ware, G. Franck, "Evaluating Stereo and Motion Cues for Visualizing Information Nets in Three Dimensions", ACM Transactions on Graphics, Vol. 15, No. 2, pp. 121 - 140, 1996

B. Watson, N. Walker, L.F. Hodges, "Managing Level of Detail through Head-Tracked Peripheral Degradation: A Model and Resulting Design Principles", ACM Symposium on Virtual Reality Software and Technology, Lausanne Switzerland, 1997

B. Watson, N. Walker, W. Ribarsky, V. Spaulding, "The Effects of Variation of System Responsiveness on User Performance in Virtual Environments", 1996

B. Watson, N. Walker, L.F. Hodges, A. Worden, "Managing Level of Detail through Peripheral Degradation: Effects on Search Performance with a Head-Mounted Display", ACM Transactions on Computer-Human Interaction, Vol. 4, No. 4, pp. 323 - 346, 1997

J. Wu, C. Duh, M. Ouhyoung, "Head Motion and Latency Compensation on Localization of 3D Sound in Virtual Reality", ACM Symposium on Virtual Reality Software and Technology, Lausanne Switzerland, 1997

# A2 Glossary

**3dfx (Inc.)**

A company which started a revolution when it delivered performing 3D hardware rasterizers for the consumer market.

**3D accelerator**

A piece of hardware which efficiently implements part of a 3D graphics pipeline

**3D Studio MAX**

A software program by which 3D models can be built and animated

**Cpu**

The central component of a computer (cpu: central processing unit), by which computation is done.

**$C^n$ continuity**

A function is said to be $C^n$ continuous, if its n-th derivative is continuous

**Crack**

A hole in a polygonal surface caused by imprecise connections between polygons.

**Distributed Interactive Simulation (DIS)**

A standard for simulators designed to ensure interoperability.

**Distributed Processing**

The use of multiple interconnected computers to solve a single problem.

**$G^1$ Continuity**

If two curves are joined in the endpoints and their tangents are of similar direction but not necessarily of the same size the resulting compounded curve is called $G^1$ continuous.

**Geometry Accelerator**

A part of a 3D accelerator which can perform matrix transformations and/or triangle setup.

**Graphics pipeline**

The steps involved in rendering an image of a 3D scene (typically: database ( geometry calculations ( triangle setup ( rasterization ( frame buffer).

**Head Mounted Display (HMD)**

A device which is worn on the head providing (stereo) image by small displays (e.g. lcd) in front of the eyes.

**Projection system**

A setup using beamers to project an image onto a screen.

**Massively Parallel Processing (MPP)**

The use of very large numbers of relatively simple or inexpensive processors in a single computer to solve a single problem.

**Motion system**

A construction which can move a platform in a number of angles and speeds to suggest movement to a person in a simulator on the platform.

**Nvidia**

A company specialized in delivering 3d accelerators for the consumer market.

**OpenGL**

A cross-platform graphics API.

**Pixelfillrate**

The amount of pixels which can be handled by the rasterizer in a certain amount of time.

**Platform**

The environment for a piece of software, consisting of both hardware and software (cpu architecture, memory model, Operating System etcetera).

**Plug-in**

A piece of custom software which can be added to a program by a third party without the need for recompiling that program.

**Polygon**

A shape in 2D or 3D space bounded by a series of points or vertices that lie in a plane. From these complex 3D models are constructed.

**Polygon count**

The number of polygons of a 3D scene, model or camera view of a scene.

**Rasterizer**

A part of a 3D accelerator which does the transformation from 3D triangles to 2D pixels. A simple 3D accelerator consists of only this part.

**Silicon Graphics (SGI)**

A leading company in the field of computer graphics.

**Symmetric Multi Processing (SMP)**

The use of a small number of (equivalent) powerful processors in a single computer to solve a single problem.

**Texture**

A bitmap image that is draped onto 3D geometry.

# A3 MRterrain

The MRterrain program was constructed using the extended 4Space. It loads a terrain dataset from a file and displays the terrain. It provides controls to rotate and translate the camera or the terrain. Also provided is a parameter settings window to change the tessellation parameters real-time. Below are some screenshots of MRterrain (all screenshots are from the same session).

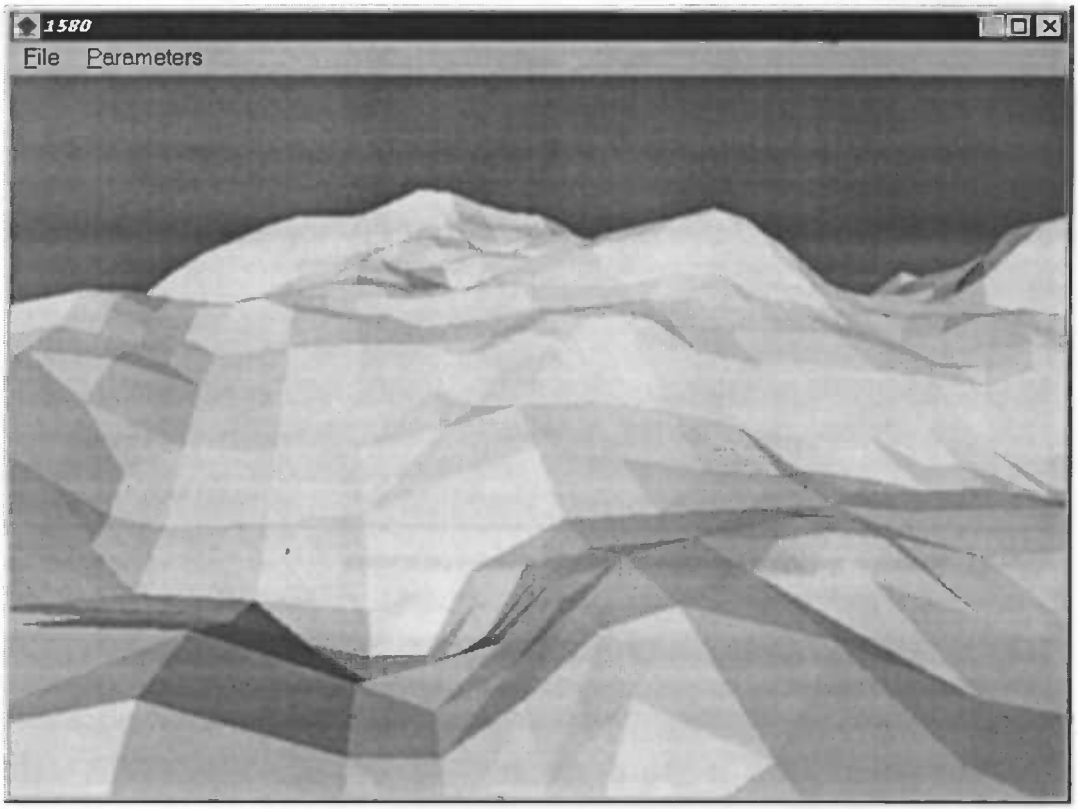



figure 30 - MRterrain displaying a particular terrain, the number displayed in the window bar is the triangle count

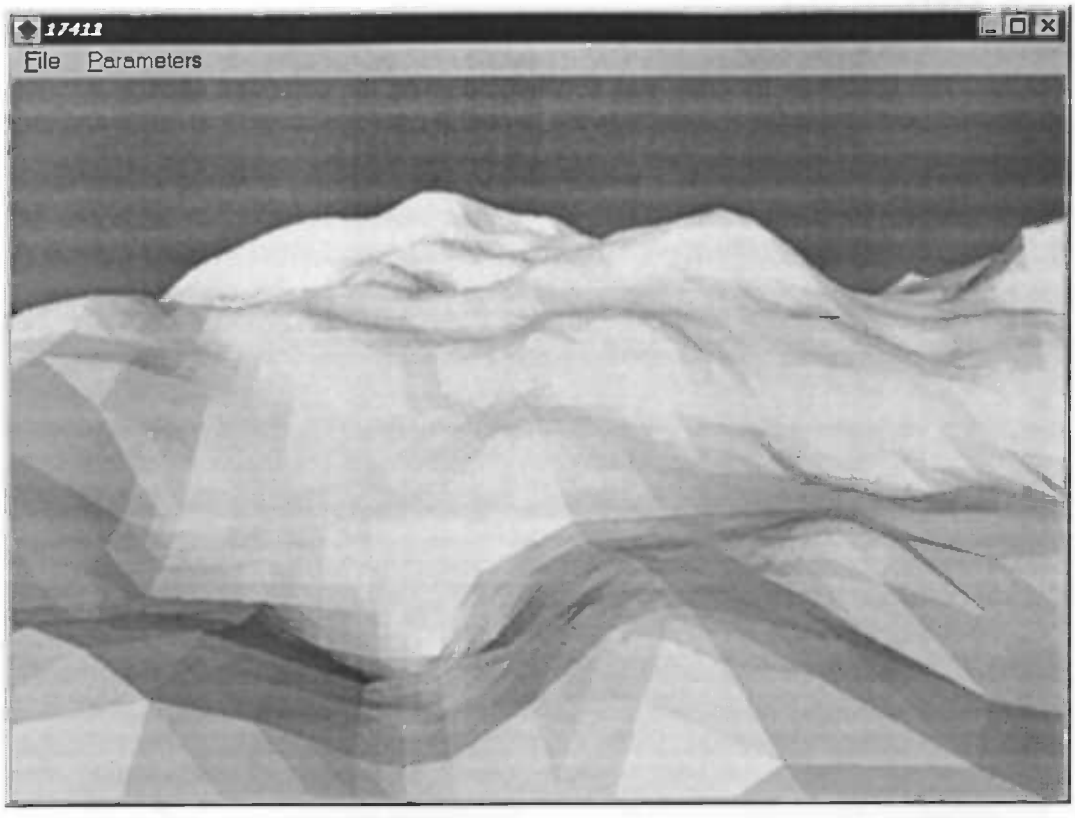



figure 31 - MRterrain displaying the same terrain with different parameters

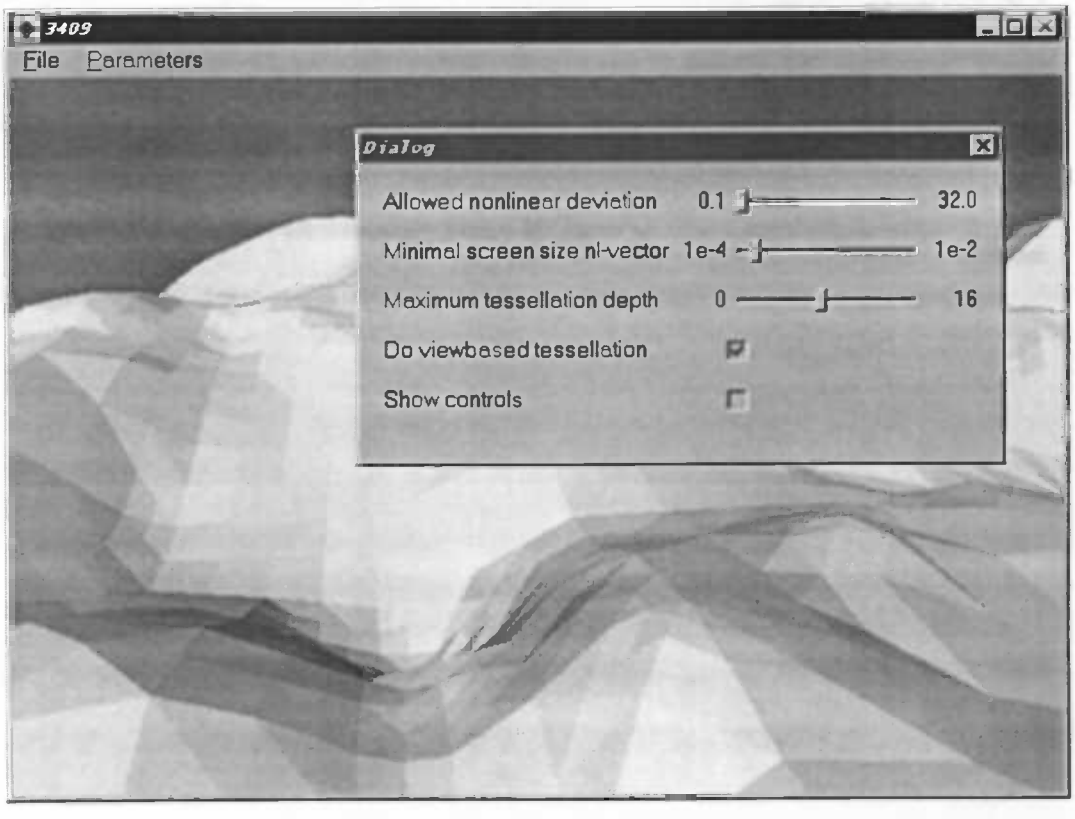



figure 32 - the parameter settings window