Development and Application of Computer Graphics Techniques for the Visualization of Large Geo-Related Data-Sets

–

Entwicklung und Anwendung computergraphischer Ansätze zur Visualisierung großer geographischer Datensätze

Dissertation
der Fakultät für Informatik
der Eberhard-Karls-Universität in Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)

vorgelegt von
Dipl.-Inform. Tobias Hüttner
aus Gäufelden

Tübingen
1999

Tag der mündlichen Qualifikation: 7. Juli 1999
Dekan: Professor Dr. Klaus-Jörn Lange
1. Berichterstatter: Professor Dr. Wolfgang Straßer
2. Berichterstatter: Professor Dr. Andreas Zell

Zusammenfassung

Ziel dieser Arbeit war es, Algorithmen zu entwickeln und zu verbessern, die es gestatten, große geographische und andere geo-bezogene Datensätze mithilfe computergraphischer Techniken visualisieren zu können.

Im ersten Kapitel der Arbeit wird kurz diese Aufgabenstellung beschrieben. Anschließend werden die einzelnen zur Verfügung stehenden Datenarten und Datenquellen spezifiziert. Weiterhin wird ein Abriß der historischen Entwicklung der Kartographie und des Vermessungswesens gegeben und notwendige Definitionen wie Koordinatensysteme erklärt.

Ein Haupteil dieser Arbeit war die Entwicklung neuer kamera-adaptiver Datenstrukturen für digitale Höhenmodelle und Rasterbilder.

Im zweiten Kapitel wird zunächst ein neuartiges Multiresolutionmodell für Höhenfelder definiert. Dieses Modell braucht nur sehr wenig zusätzlichen Speicherplatz und ist geeignet, interaktive Update-Raten zu gewährleisten. Es verwendet einen beobachter-abhängigen Pixelfehler um das zunächst beobachter-unabhängige Multiresolutionmodell an die aktuelle Betrachterposition anzupassen. Dieser Anpassungsvorgang nutzt die perspektivische Verkürzung, um die Menge darzustellender Daten drastisch zu reduzieren. Weiterhin kann dieses Modell auch für andere Datenarten, zum Beispiel CAD-Modelle, verwendet werden.

Kapitel drei diskutiert Ansätze zur schnellen Bestimmung sichtbarer und verdeckter Teile einer computergraphischen Szene, um die Bewegung in großen und ausgedehnten Szenen wie Stadtmodellen und Gebäuden zu beschleunigen. Hierzu wurde ein neuartiger Algorithmus entwickelt, der nur den Bildspeicher des Graphiksystems nutzt, um diese Fragen zu beantworten. Da der Bildspeicher integraler Bestandteil jeder Graphikhardware ist, kann dieser Ansatz auf eine große Zahl von Computersystemen übertragen werden. Dieser Ansatz kann sehr einfach mit einer Hardware-Implementierung realisiert werden und speichert dabei die zum Zeitpunkt der Rasterisierung bekannte Verdeckungsinformation mithilfe spezieller Zählereinheiten.

In Kapitel vier werden einige Problemstellungen im Zusammenhang mit Texture Mapping diskutiert. Zunächst wird eine neue beobachterabhängige Datenstruktur für Texturdaten beschrieben. Dieser Ansatz erlaubt es, auf einfache Art und Weise, sehr große Texturen zu verwenden, die weit größer als der zur Verfügung stehenden Texturspeicher sein können. Wiederum folgt aus der Ausnutzung perspektivischer Verkürzung, daß nur ein kleiner Teil der Bilddaten für eine korrekte Darstellung der Textur notwendig ist. Als nächstes wird ein Verfahren zur Textur-Selektion beschrieben, das es erlaubt, ein beliebiges Quadrilateral innerhalb eines Bildes als Textur zu verwenden. Das Kapitel wird mit der Beschreibung eines neuartigen Verfahrens zur Texturfilterung abgeschloßen. Dieser Filter liefert weit bessere Ergebnisse als die bisherigen Verfahren, läßt sich aber weiterhin in Hardware realisieren.

Die meisten dieser Algorithmen und Verfahren wurden in ein interaktives System zur Geländevisualisierung integriert, das den Projektnamen *FlyAway* hat und in Kapitel fünf beschrieben wird. Dieses System kann verschiedene Arten von Geodaten visualisieren. Ein wesentliches Entwicklungsziel war es, die Portierung auf verschiedene Plattformen zu ermöglichen. Dies ist gelungen und *FlyAway* wird heute sowohl in der Unix Welt wie auch auf PC-Plattformen verwendet.

Contents

Chapter 1

Introduction

In this thesis, several approaches and developments are described, which arose from the question, if and how currently available midrange graphic systems are suitable for use in the area of geographical, geological, and other terrestrial visualization. Today, there exist many interesting and important data sets of different dimensions and scientific content.

The size of these data sets can be arbitrarily large, since modern mapping- and survey technology produces with the help of modern scanning and raster procedures very detailed measurements describing our globe. In contrast to this, graphical visualization capabilities have developed much slower. A simple example explains this discrepancy:

> The federal state of Baden-Württemberg, a part of the Federal Republic Of Germany, extends over an area of approximately 40.000km^2. It is covered with a so called digital elevation model, which is a rectangular grid having a height measurement stored at each grid point. This height value is measured in meters over sea level with a grid cell size of 50m. About 16 millions of measurements are contained in this grid. If these points are visualized using classical computer graphics techniques, they produce about 32 millions of triangles, describing the surface of the landscape. Modern graphic systems are not able to deal with such large amounts of data. A graphic workstation is able to visualize approximately 2 millions of triangles per second (*Hewlett Packard Kayak FX 4*), graphic supercomputers are capable of crunching 10-20 millions of triangles (*Silicon Graphics Infinite Reality*). If we now propose a frame rate of only 5 frames per second to have an interactive feeling during motion, this means that the system has to visualize these 32 millions of triangles 5 times per second. This results in 160 millions of triangles per second. The described workstation would be overwhelmed with this amount by 8000 percent, since this system would need 16 seconds for one visualization of the triangle data given the optimal case (enough memory, thus no interaction with the hard disk to reload data).

The rest of this chapter describes shortly which data sets and data sources are available today. In the following chapters, some ideas and approaches will be described to resolve some of these divergences.

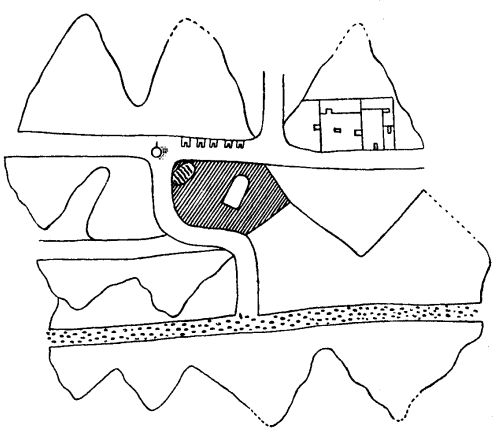


Figure 1.1: Nubic gold mine map, see [45].

1.1 Survey and Cartography

The following description of the above topics is an outline of [61] and [45], since these references are standards in this field.

The science of *Survey* concerns itself with measuring the surface of the earth and builds models for representing the measured data. The results are digital elevation models and all kinds of maps.

Cartography or Map Making is the discipline of collecting, storing, processing, and visualizing terrestrial data with the help of maps. One especially important part of map making is the question of how to get a three-dimensional earth model with the help of a projection onto a two dimensional map.

1.1.1 History of Map Making

Early Map Making Approaches

Some historical sources describe the existence of cartographic activities in Babylon, China, Greece, and Rome during the high time of these cultures. Despite this knowledge, only a few maps have survived until today.

The oldest map seems to be a map made in Babylon 3800 b.C. which is engraved into a piece of clay. It shows the north of Mesopotamia. A very old and important map is also the nubic gold mine map which was drawn on papyrus. (1300 b.C., see figure 1.1).

For the Greece scientists, cartography was the question of how to define a model of the earth. The first earth maps depicted our planet as being a flat tile, surrounded by the seas. The impression of the earth being a sphere was developed by the school of Pythagoras (approximately 500 b.C.) and was made popular by Aristotle's proof (approximately 350 b.C.). He argued in this proof that since the shadow of the earth is always a circle during a moon eclipse then the earth must be spherical to cause such a shadow. Erastothenes measured in 200 b.C. the radius of this sphere for the first time by measuring the so called zenith angle of the sun at two different places at noon time.

With this knowledge, new approaches in map making were possible. The method of map making by using sets of parallel latitude or longitude lines was developed. Dikarchos (350-290 b.C.) showed in his maps only one line going from east to west. But Erastostenes (276-195 b.C.) already used a set of parallel lines. Hipparch (190-125 b.C.) subdivided the equator in 360 degrees and developed the stereographic and the orthographic projections. Marinus of Tyros (approximately 100 a.C.) developed projections using cylinders, Ptolemäus (87-150 a.C.) added conic projections.

The Romans had a completely different opinion in using and making maps. They didn't use maps to document new geographic results as the people in Greece, but to declare the borders, the roads or the counties of their empire. Therefore they only used existing techniques from their time and contributed no overall improvements. Their so called "Itinaria" were not correctly scaled and stored only coarse geographic relations. They were used as road maps for military purposes, and later for documenting trading roads.

The only antique globe and therefore the oldest globe in existence today, the globe of Farnese in Neapels, is a sky globe and made as a Roman copy of Greek work which is dated at 100 b.C. .

Medieval Cartography

The Islamic cultures used and improved the geographic knowledge of Greece. In Europe, the church dominated all scientific developments which were mostly done in monasteries. The earth maps developed there should not only show real geographic properties, but should also demonstrate and explain the content of the Bible. The earth is mostly depicted as a flat, circular disk (so called wheel map), directing with east to the upper side of the map. The continents are ordered in the form of the letter **T** with Asia above the horizontal dash of the **T**, Europe to its left side below the dash and Africa to the right of Europe.

Mountains were shown in such maps in a very schematic way using side or bird-eye views, eventually also as a ribbon with ornaments or integrated small drawings.

This lasted until the late Middle Ages, when the improved level of geographic knowledge freed the cartography of their religious bindings imposed by the church. Then some new earth maps with improved accuracy were made, for example the earth map of Genua (1457) or the circular "mappa mundi" of the munch Fra Mauro (1459), having a diameter of 7 meters.

Change of Cartography up to now

Two important events had a major impact on the development of cartography during the 15th and 16th century: the geographic discoveries made and the advent of the printing process. The discovery of new continents introduced a huge amount of new geographic knowledge, in the same time new map material was necessary for new expeditions. The printing process using encarved wood or copper plates replaced the expensive manual copying which was also prone to introducing copying failures into the maps. This was the beginning of the use of mass produced maps for navigation.

The effort being spent on collecting and processing geographic information resulted in a great growth of cartography. Real cartographic centers were founded, first

in Italy, Spain, and Portugal and later on in the Netherlands and in Germany. The main interest was in land and sea maps, and also the first globes and regional maps were produced. The demand for a geometrically correct representation, which was imposed by the sailors and other nautical people, resulted in the first intensive use of map grids and the development of new mapping projections.

A cartographic highlight was the life and work of Gerhard Kremer, called ”Mercator” (1512-1594 a.C.) who lived in the German town Duisburg. After a great number of regional maps and globes, in 1569 he produced a famous world map, intended to be used on the sea. This map was produced in a new projection technique developed by Meractor and this projection is used to this day to produce the map grid of most sea maps.

The usage as a source of information, but also the wish to represent knowledge and wealth, led to an increase in the production of globe models. One of the most famous producers was V. Coronelli who produced around 1700 many small globes and also some huge ones with diameters between two and four meters. The interwoven development of geography, business trading and discoveries was responsible, as already described, for an increasing demand and also an increasing production of earth and sea maps, but also for maps describing regional topics of smaller areas. The cartographic techniques of those days were not able to produce topographic measurements as we know them today.

Therefore, the first versions of regional oriented maps were based on coarse geographic orientations, the evaluation of travel times and scheme-like approaches. Later on, compass, measurement ropes and footstep counting were used as additional input. The determination of a geographic position with respect to a system of longitude and latitude circles was still not known. The measurements tried to depict the countryside along roads or trading routes with a more or less accurate scheme. The cartographic details consisted of the most important roads, rivers and towns which where often drawn in an arbitrary projection not related to the map itself. Mountains and other landscape variations were drawn with the help of side or bird-eye views with increasing detail. The mapmakers tried to get a more realistic look of their mountain drawings by using shading as an artistic means to get a more intuitive feeling of steepness and angles (see figure 1.2).

The further development of cartography was determined by the improvement of topographic scanning methods and by the slowly happening conversion from image based ways of map making to using more abstract paradigms. The topographic works were heavily influenced by two new developments: the first triangulation, used in 1617 by Willibrord Snellius in the Netherlands (more well known because of his optical laws for refraction) and by the invention of the so called measuring table, a device for accurately doing measurements. This table was invented by Johannes Prätorius about 1600 in Zürich in Switzerland.

Given more detailed regional measurements, it was possible to use more and more an exact orthographic view (parallel projection) from above for drawing topographic details. The birdseye like town views (the most famous ones were produced by Matthäus Merian in his ”Topographien” - about 1640) were replaced by the exact geometric outline of roads and buildings which are much more suitable for their use in maps. Small objects were for the first time substituted by symbols, so called signatures. One example of this is the city map of the German town Esslingen, which was

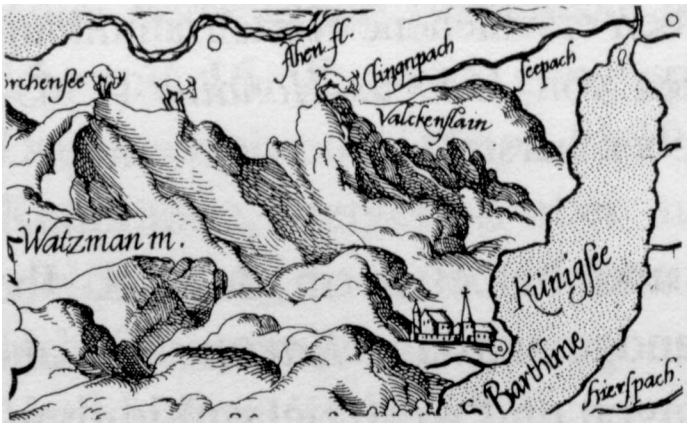


Figure 1.2: Side view in an individual way (example of Apians ”Great map of Bayern” (mountain part of Germany)), see [45].

drawn by the well known German map maker and mathematician Tobias Mayer at the age of only sixteen (1739 - see figure 1.3).

The side view of mountains was not very suitable, since much of the countryside was obscured by itself in these kinds of projections. Therefore, in this area of map making, new projections were applied and were named as ”Kavalier”, or military, or half perspective mostly due to military reasons. The Kavalier perspective is a parallel projection where the projection direction and the normal of the projection plane form an angle of 45 degree. With this, lines perpendicular to the view plane are not shortened and have the original size (see figure 1.4 and [60] in chapter three).

Finally, the orthographic view from above was used in the same way as it is for producing city maps and this has not changed since then.

The usage of triangulation for the survey of a whole country as the standard procedure began, after French scientists had used it successfully in their numerous approaches to measure the exact degree of longitude (between 1669 and 1741). Cassini covered France after 1750 with a mesh consisting out of 2000 triangles and used this as the foundation for a nationwide survey. Furthermore, instruments for high precision angle measurements were developed which are absolutely crucial for using triangulation. As an example, the sextant of the English scientist Hadley (about 1750) and the ”full circle mirror” of Tobias Mayer (1754) shall be cited (see also figure 1.5).

The development of these approaches was also emphasized, since for military operations a comprising topographic mapping of high quality and meaningful content is very valuable. By the middle of the 19th century contour lines for coding the height above sea level and the angle of hill slopes were replacing the so called ”Schraffen” technique, which had dithered the hills with small pen strokes according to their slope. To be able to evaluate these contour lines at a large number of measuring points in short time, new height measurement tools and techniques were developed using different principles like barometric air pressure evaluation or trigonometric calculus.

The triangulation procedure is still used today for topographic and geographic survey. The survey itself is stored with the help of the so called ”fixed point fields” (see figure 1.6).

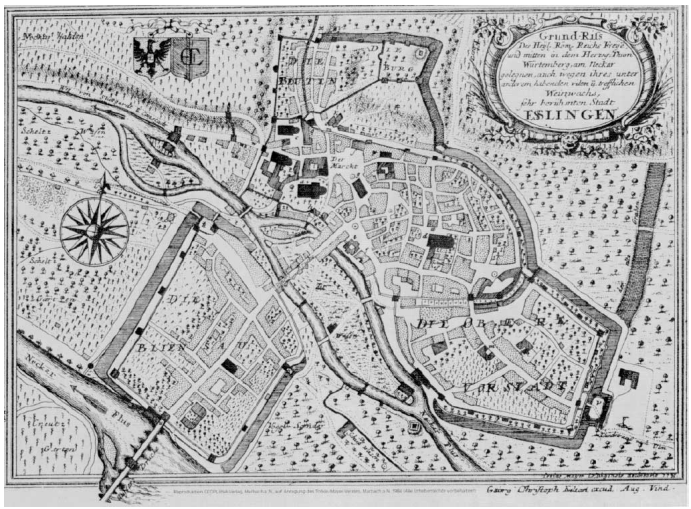


Figure 1.3: City map of Esslingen by Tobias Mayer, see [45].

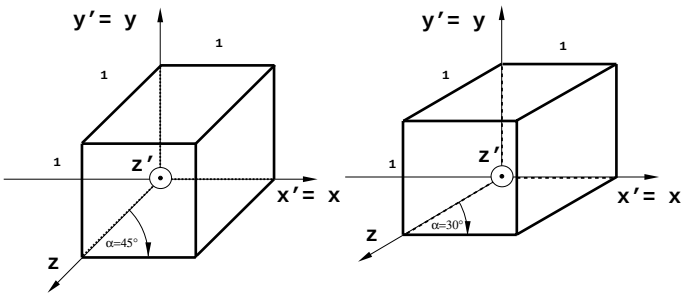


Figure 1.4: Kavalier perspective, see [60].

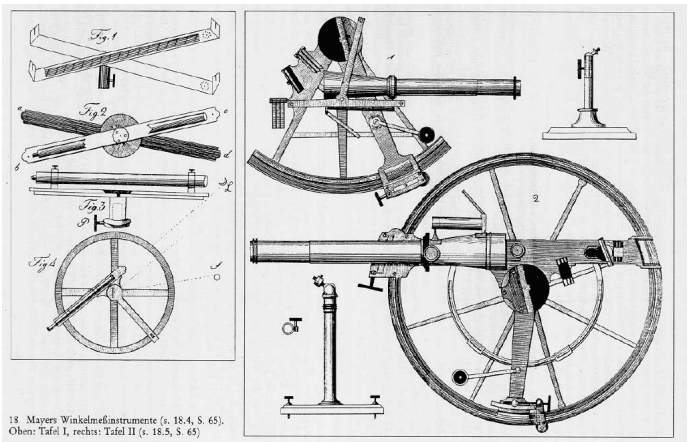


Figure 1.5: Historic measurement tools, here some old versions of angle measurement tools in the form of some sextants and so called ”full circle mirrors”, see [45].

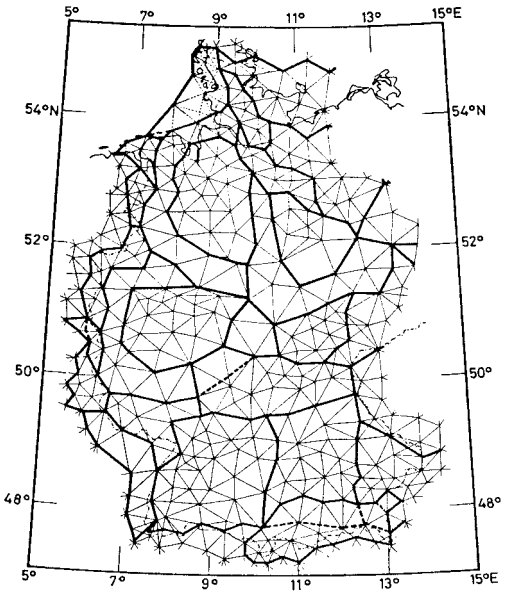


Figure 1.6: German major fixed point field (1969), see [61].

The fixed points were often realized with the help of the tip of a tower or a similar noticeable point in the landscape. Today, usually artificial markers are set since they can be placed more appropriately and with higher precision. The point set itself is sorted hierarchically and builds up a hierarchy of triangulations with decreasing triangle size. In Germany, the triangle size of 30-50km for the first level is reduced down to approximately 1-3km for the finest level.

Other topographic fix point fields store further information. There exists, for example, a special fix point field which stores height values above sea level.

Also in the area of cartographic reproduction techniques, several improvements were made which were mainly due to developing the printing process further and further. In 1796, Alois Senefelder invented the lithography, which enabled the map makers to produce maps faster and with higher reproduction numbers. Photographic procedures and the autotypic rasterization enabled a quick reproduction process from the original model to the reprint and also to reproduce colored maps. Rotation printing presses and the 1904 invented offset printing were the necessary technical developments for an industrial map production process.

The youngest revolution in the cartography and map making discipline is dominated by the massive use of computer systems during the whole process of map making. This technological development introduces the greatest methodic change in cartography and map making up to now.

1.1.2 Survey

In the discipline of survey, three-dimensional models and two-dimensionally accurate schemes of the earth's surface or parts of it shall be produced. To achieve this, the choice of a suitable coordinate system is of great importance. The user has to decide,

whether to use a Cartesian coordinate system addressing the three-dimensional space, or a coordinate system on the earth's surface, using a longitude/latitude scheme, is more suitable. The first ones are used in modern navigation and survey technologies, for example *GPS (Global Positioning System* - will be described shortly), the latter ones are used for cartographic projections and map making. It is of special interest to define a suitable three-dimensional model of the earth which can be used as the basis for all these applications.

Sphere, Ellipsoid, Geoid - a Model of the Earth

As mentioned before, the scientists of Greece were already aware of the earth being a sphere due to their experiments and observations. Newton introduced in 1670 the laws of gravitation which caused some uncertainty about the earth's spherical form. The gravitational force on the earth's surface is a composition of the earth's attractional force due to gravitation directed towards the earth's center and the centrifugal force caused by the rotation of the earth which has the opposite direction. The latter one is stronger at the earth's equator and gets smaller when coming towards the poles. Due to the centrifugal forces, the earth is squeezed in the north-south direction producing an ellipsoidal shape in three-dimensional space. With the increasing accuracy of measurement technology, these ellipsoids and their parameters (major and minor axes or alternatively a flattening factor) where defined again and again, often for special parts of the earth's surface or for only one country. In figure 1.7, the most important definitions are summarized. The *WGS84* ellipsoid (World Geodetic System, defined in 1984) is the actual reference ellipsoid for most technical uses like the GPS system. It is remarkable, that the flattening of the ellipsoidal earth models compared to a sphere model with radius 6371km having nearly the same volume and surface area is approximately 20km at the poles. Also the choice of a suitable ellipsoidal data set, a so-called geodetic datum, is crucial for survey and cartographic purposes, since otherwise deviations of hundreds of meters can result in a map due to choosing the wrong geodetic datum (see figure 1.8).

Even using ellipsoids as earth models are not that perfect in resembling the earth as they seem to be. Due to density variations in the earth's surface and topographically caused mass collections (mountains), scientists today tend to deform the ellipsoid's surface locally with the help of an offset surface. The resulting surface, called geoid, can be thought of as the average water level which would result if the seas could flow beneath the continents with the help of a connected pipe system (see figure 1.9 and 1.10 for the offset surface). This offset surface can be produced with the help of physical simulations and the known density distributions of the earth's surface. The geoid's surface is then the base of all local height measurements, since it defines the local sea level. Usually, these offset surfaces are rather coarse and therefore the deviation of geoid and ellipsoid is normally smaller than 50m.

1.1.3 Map Projections and Coordinate Systems for Map Making

Map projections project a mapping grid superimposed on a three-dimensional earth model onto a plane. The result of this projection should be a suitable base for producing maps or for building planar digital elevation models. To achieve this, a variety

Selected Reference Ellipsoids

Ellipse	Semi-Major Axis (meters)	1/Flattening
Airy 1830	6377563.396	299.3249646
Bessel 1841	6377397.155	299.1528128
Clarke 1866	6378206.4	294.9786982
Clarke 1880	6378249.145	293.465
Everest 1830	6377276.345	300.8017
Fischer 1960 (Mercury)	6378166.0	298.3
Fischer 1968	6378150.0	298.3
G R S 1967	6378160.0	298.247167427
G R S 1975	6378140.0	298.257
G R S 1980	6378137.0	298.257222101
Hough 1956	6378270.0	297.0
International	6378388.0	297.0
Krassovsky 1940	6378245.0	298.3
South American 1969	6378160.0	298.25
WGS 60	6378165.0	298.3
WGS 66	6378145.0	298.25
WGS 72	6378135.0	298.26
WGS 84	6378137.0	298.257223563

Peter H. Dana 9/1/94

Figure 1.7: Different ellipsoids (flattening factor f: measure for the squeezing of the earth in north-south direction, $f = \frac{(a-b)}{a}$ and a is the major, b the minor axis of the ellipsoid).

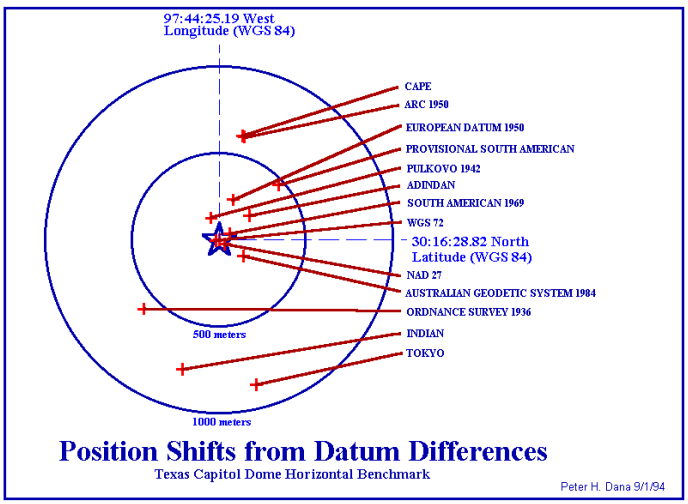


Figure 1.8: Deviations due to using different geodetic datums.

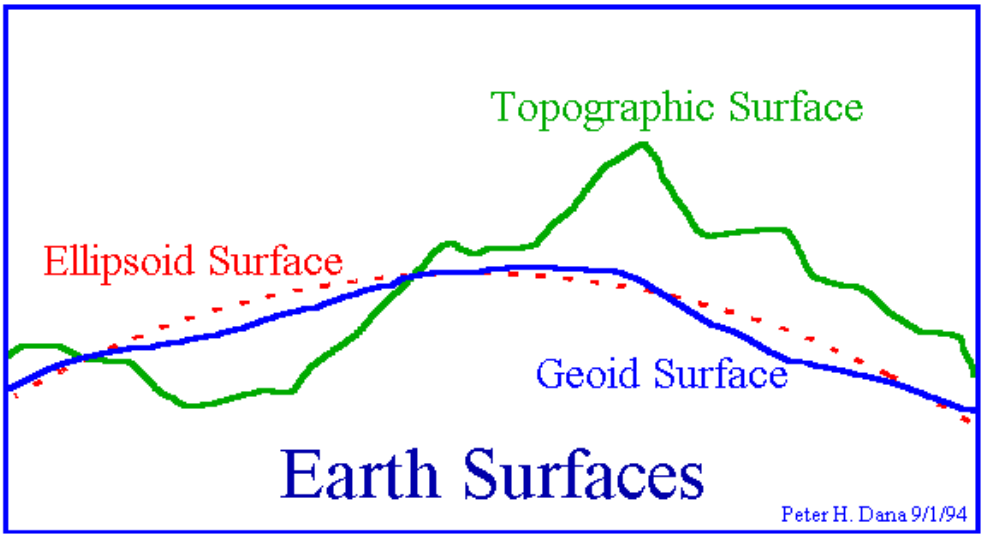


Figure 1.9: Ellipsoid and geoid.

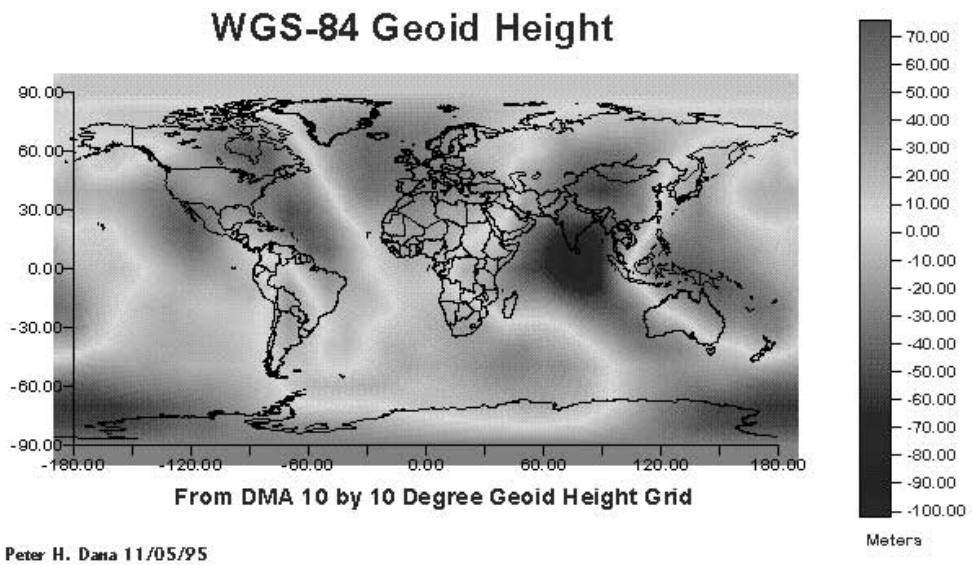


Figure 1.10: Geoid Offset Surface for the WGS84 Ellipsoid.

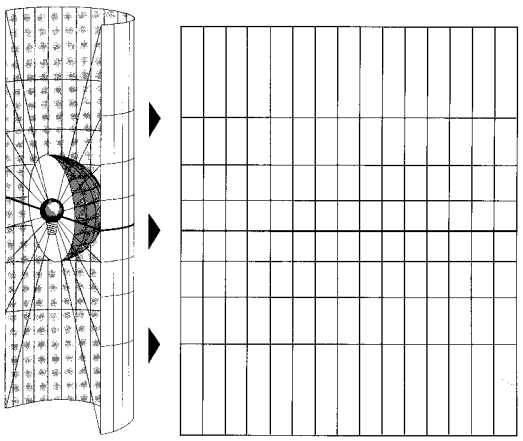


Figure 1.11: Map projection with the help of a light source, see [92].

of map projections were defined, each having different properties useful for special application areas. In the following section only basic principles of doing such projections and the most widely used map coordinate system (UTM - Universal Transversal Mercator) will be summarized.

Properties of Map Projections

Whether one treats the earth as a sphere, an ellipsoid or as a geoid, its three-dimensional surface must be transformed to create a flat map sheet. This transformation, usually achieved through utilizing a mathematical function, is commonly referred to as a *map projection.* One easy way to understand how map projections alter spatial properties is to visualize the projection as a light shining through the Earth onto a surface, called the projection surface (see figure 1.11).

The projection of a map involves the use of coordinates as defined by the projection formulas. These formulas transform an input coverage into an output coverage and have different properties and invariants.

The following enumeration briefly describes the most important properties of map projections:

- Conformal Projections: Conformal projections preserve local shape due to conserving the angles between lines. The drawback is that area enclosed by a series of arcs may be greatly distorted in the process. No map projection can preserve shapes of larger regions due to the flattening imposed on the three dimensional earth surface.

- Equal-area Projections: Equal-area projections preserve the area of displayed features. To do this, the properties of shape, angle, scale, or any combination of these may be distorted. Thus, in such projections the meridians and parallels may not intersect at right angles. In some instances, especially maps of smaller regions, it will not be obvious that the shape has been distorted, and distinguishing an equal-area projection from a conformal one may prove difficult unless

documented or measured.

- Equidistant Projections: Equidistant maps preserve the distances between certain points. Scale is not maintained correctly by any projection throughout an entire map; however, there are, in most cases, one or more lines on a map along which scale is maintained correctly.

- Direction Preserving Projections: The shortest route between two points on a curved surface such as the Earth is along the spherical equivalent of a straight line on a flat surface; that is, the great circle on which the two points lie. Direction preserving or *azimutal* projections are used to rectify some of the great-circle arcs, giving the directions or azimuths of all points on the map correctly with respect to the center. There are projections of this type that are also conformal, or equal-area, or equidistant.

Projection types

As already defined, a map projection is a mathematical expression that systematically projects locations from the surface of a sphere or an ellipsoid/geoid to the representative position on a planar surface.

Because maps are flat, many map projections are made onto geometric shapes that can be flattened without stretching their surfaces. Common examples of shapes that meet this criterion are cones, cylinders, and planes. Actually, cylinders and planes are limited forms of a cone.

The first step in projecting from one surface to another is to create one or more points of contact. Each point is called a *point of tangency*. As illustrated below, a planar projection is tangential to the globe at one point only. Tangential cones and cylinders contact the globe along a line. If the surface of projection intersects the globe instead of merely touching its surface, the resultant projection conceptually involves a secant calculation rather than a tangential calculation. Whether the contact is a tangent or a secant, the location is significant because it defines the point of lines of zero distortion. This line of true scale is often referred to as a *standard line*. In general, projection distortion increases with distance from the point of contact.

Many common map projections can be classified according to the projection surface used for each: conic, cylindrical and planar. In figures 1.13, 1.12, and 1.14 the respective approaches are briefly depicted.

The UTM system

The so called *international ellipsoid* WGS84 (see 1.7) is the basis for the *Universal Transversal Mercator* projection (UTM). The UTM is a transversal cylindrical projection.

This system covers the earth between 84th degree latitude in the north and 80th degree in the south with 60 so called *meridian stripes* each having an extension of 6 degree longitude. It was originally defined for making military maps for the U.S. forces and NATO, but is nowadays also used for civil purposes. To reduce the stretching of length at the vertical stripe borders, the middle meridian is scaled by a factor of 0.9996.

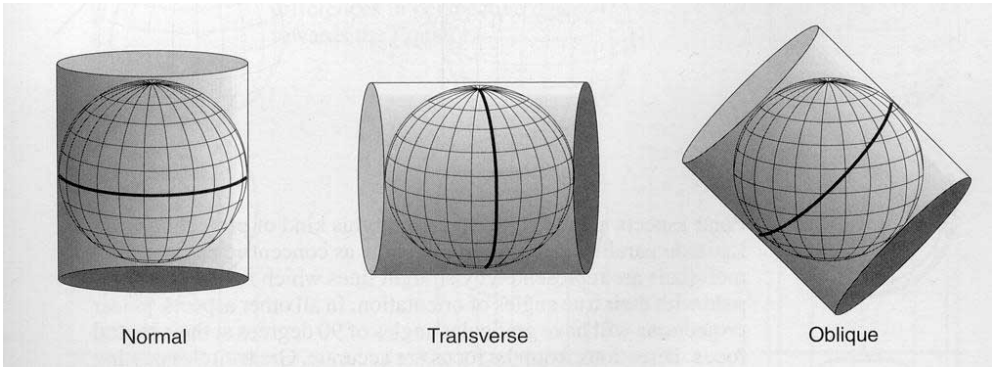


Figure 1.12: Cylindrical projections, see [92].

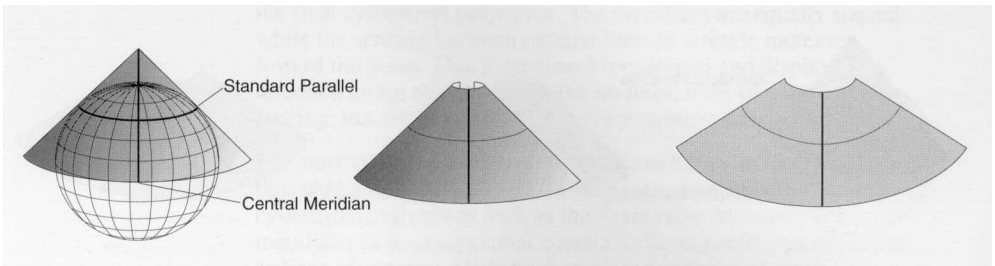
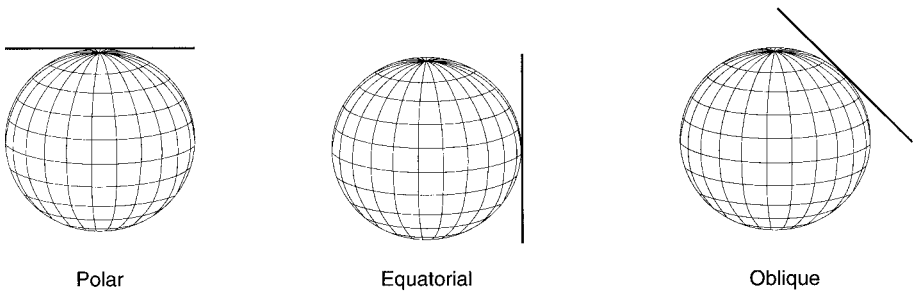


Figure 1.13: Conic projections, see [92].



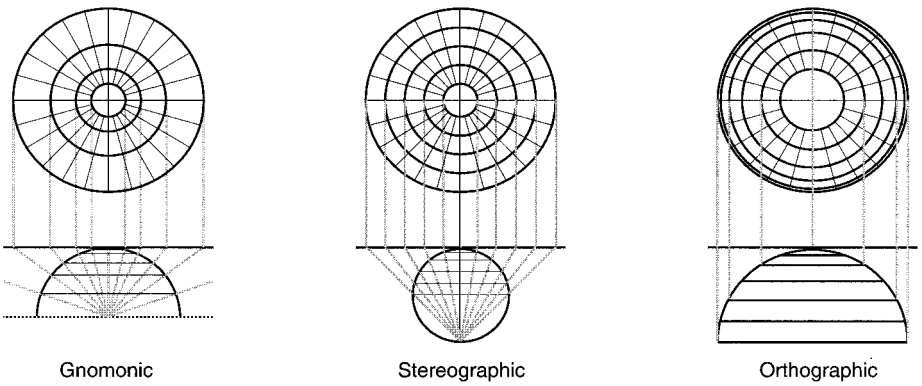


Figure 1.14: Planar projections, see [92]. The bottom figures show, how the placement of the projection center affects the projection properties.

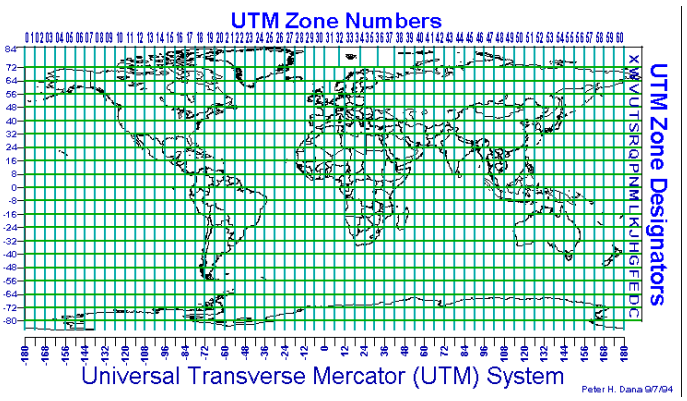


Figure 1.15: Scheme of all UTM zones.

Most European countries today use conformal cylindrical projection systems. One famous exception is Switzerland, where an oblique cylindrical projection is used for map making having the origin and the tangent circle going through a point in the capital Bern.

In figure 1.15, the scheme of the UTM zones is depicted. Figure 1.16 shows one single zone.

The Gauß-Krüger system

The famous German mathematician *C.F. Gauß* developed a projection which utilized ellipsoids for transversal cylindrical projections and not only spheres. This projection was established by Krüger between 1912 and 1919 in the German cartographic community and therefore in Germany the UTM coordinates are called Gauß-Krüger coordinates.

A system of meridian stripes was established in 1927 initially on the basis of the so called *Bessel* ellipsoid. Today, the WGS84 ellipsoid is used for this coordinate system

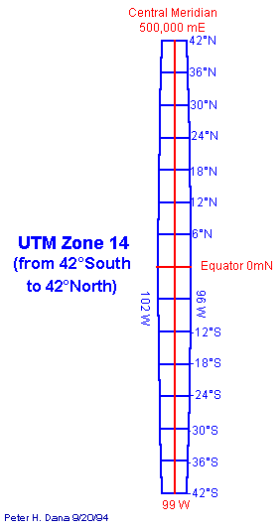


Figure 1.16: Single UTM zone 14.

(see figure 1.7). The meridian stripes for the system have an extension of 3 degrees in longitude and are defined along the *main or middle meridians* 6th,9th,12th, and 15th degrees of longitude.

For a coordinate pair (x, y), the value x is called the *Right* value, and the value y is called the *Up* value. The right value is defined by adding the distance from the main meridian to 500,000m to prevent the right value from becoming negative. Finally, the value of the main meridian divided by 3 is added. The up value simply measures the distance to the equator in meters.

A value of $(3,600,000.00m, 5,000,000.00m)$ describes a point, lying in the meridian stripe around the 9th degree 100km to the east of the middle meridian and 5,000 km to the north of the equator.

1.1.4 Modern Survey Technologies - GPS

One of the most exciting new survey technologies is *Global Positioning System - GPS* introduced at the end of the eighties.

Since the fifties scientists have been able to launch satellites with rockets and to position them on stationary orbits around our globe. These satellites are used for many purposes, transmitting signals, taking pictures and also for navigation and cartographic purposes.

The first navigation system was the *Transit Navigation System* of the U.S. Navy, built up during the fifties. It consisted of five satellites which could be used for position determination using the Doppler frequency shift of signals sent by the satellites. This Doppler shift occurs because the satellites have a defined velocity as a moving signal source. Out of the known orbit parameters and with the measured frequency shift, the position of the receiver could be calculated.

The next generation of navigation systems, GPS and GLONASS in the former Sowjet Union influence area, rely on detecting differences in time stamps compared to

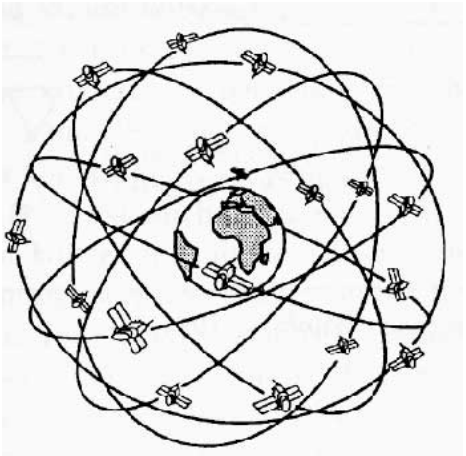


Figure 1.17: Schematic of the GPS satellite constellation, see [61].

a global system time.

The system itself consists of 24 satellites (3 are held as spare devices in orbit), see figure 1.17. 4 satellites share a common orbit, 6 of these orbits are placed in space in such a way, that at any time at almost every point on the earth's surface there are at least three satellites visible. Each satellite has an atomic clock built in and all the clocks of the satellites are held in synchronization ensuring a global GPS system time.

The satellites are constantly sending data records consisting of their position in space and the current GPS system time. The user has a GPS receiver, which has also a built-in clock which is synchronized with the GPS system time. The distance between a satellite and the user can be determined by detecting the time shift between the time the user receives the GPS signal of a satellite and the time which is contained in the signal itself as data. Furthermore, the data sent by the satellite contains the satellite's position in space in the WGS84 coordinate system. If a user on earth can receive at least three of these signals, he can determine his position in 3D uniquely as the intersection of three spheres in space (see figure 1.18). Normally, only electronic clocks with crystal oscillators are built into the GPS receivers for common users. With the help of a forth satellite, the inaccuracy due to frequency shifts in these low cost clocks can be compensated. The signals of the GPS satellites contain additional information to compensate other time shifts due to the different atmospheric layers. The satellite's signals sent down to earth are refracted between these layers. This can increase the time shift measured by the user.

The accuracy of the GPS position measurements depends on several factors. First of all, it is very important that the viewable satellites are suitably placed. This means, the angles between them should be maximal. The best situation is having a satellite in the zenith and the three others in 120 degree angles positioned at the horizon. With this constellation, the maximal deviation of normal GPS measurements is 16m (PPS - precise positioning service). Due to the military origin of the GPS system, the data sent by the satellites has noise added to decrease the accuracy of the system for normal users (AS - anti spoofing). Only special receivers are able to decrypt these signals to allow the use of the full PPS accuracy. Normal receivers can only use the standard

precision service (SPS), which allows an accuracy higher than 100m. For normal purposes like car navigation this is sufficient, but using GPS for survey purposes requires more accuracy.

This can be achieved in different ways:

- *Relative Positioning*: The coordinates of a new point are determined with the coordinates of a known point and two GPS measurements, one at the new and one at the known point. Combining the two measurements with a subtraction results in an offset vector, which can be used to calculate the new position from the known one. The subtraction eliminates systematic errors from the two GPS measurements.

- *Repetitive Measuring*: With the help of long measurement cycles, errors can be reduced with the help of statistical methods. Measurement times can increase to several hours.

- *Modeling of Atmospheric Errors*: The better the atmospheric influences are modeled in the equation systems used for calculating, the more accurate the results will be. Therefore, additional atmospheric information and data can be used besides the data already available in the satellite's signals.

With these approaches, the accuracy of the measurements can be increased to be higher than 2mm. Using such systems, a fast, but very precise survey can be realized, since the survey reduces itself to establishing the GPS receivers and collecting and processing the data. Furthermore, some survey tasks are only possible using this very high precision of GPS. One example is the long time measurement used to control the potential damage of huge structures under high pressure like bridges or dams.

1.2 Data Sources and Data Acquisition

This section describes briefly, where to get data and what kinds of data are currently available as a raw material and input for "virtual geography".

1.2.1 Remote Sensing

Nowadays, flying at different heights and with various devices one can collect geo-related information of the landscape. The devices range from low altitude planes and balloons to satellites in orbit either positioned stationary or dynamic. This is described by the term remote sensing, a discipline driven also by military intelligence, for which such data is crucial.

Digital Elevation Models

When modern raster scanners scan the earth's surface, different kinds of data of all parts of the frequency spectrum are produced. Infrared frequencies can for example provide valuable hints on the health of forest trees. Some sensors carried on planes or satellites are able to scan the earth's surface in an active way. For this, they use laser or radar scanners (SAR: synthetic aperture radar). The result of this scanning is a regular

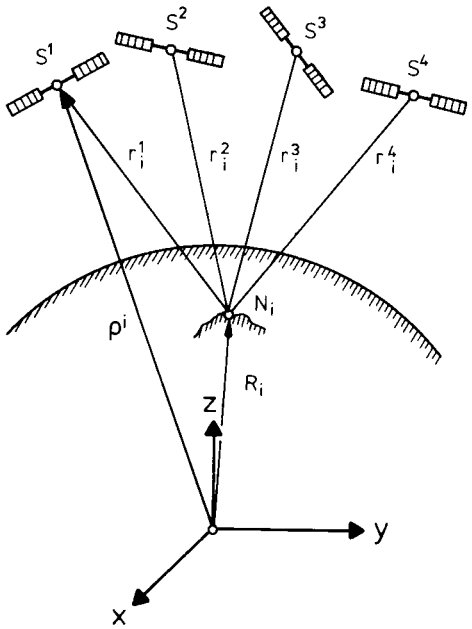


Figure 1.18: Determination of a user's position from 3 GPS satellites, see [61].

rectangle grid with height values at the corner points. These grids are called digital elevation model (DEM) or digital height model (DHM) (see figure 1.19).

Such a DEM can also be digitized from contour height lines already stored in topographic maps, see figure 1.20.

These DEMs are nowadays available of most parts of the earth. Their resolution ranges from 1km down to 30m. Some of these data sets are public domain. For example the *USGS* (U.S. Geological Survey) offers two models:

- USGS 7.5 minute DEM:
  - 30m x 30m resolution
  - coordinates are projected and are given in UTM (universal transversal mercator)
  - maximum error in height is +/- 15m
  - available for the whole USA
- USGS 1 Degree DEM, see figure 1.21:
  - 90m x 90m resolution (3 arc seconds in square)
  - public domain - download via *ftp*
  - coordinates are defined in a 3D system (WGS72 or WGS84), not projected
  - maximum error +/- 30m in 90
  - available for the whole USA
  - originally created by the DoD (Department of Defense)

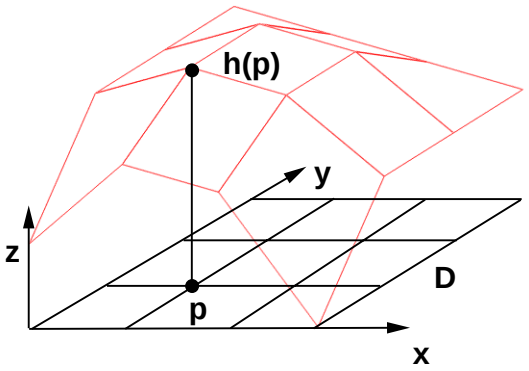


Figure 1.19: Digital Elevation Model.

Aerial Photography

Photography by airplanes flying over the terrain provides us with high-resolution images, which are made with a camera respecting central projection and taking the picture not perpendicular to the earth's surface. To use these photos and to combine them with map data, they have to be geo-coded, that means converted to a common coordinate system, for example Gauß-Krüger, and to an orthographic projection.

This can be achieved with the help of a DEM. The picture is virtually reprojected back onto the DEM from the point where it was taken in a central perspective projection. Then it is resampled from a new point of view with an orthographic projection (see figure 1.22). This task can easily be performed with the help of projective textures (see chapter 3).

The resulting orthographic, geo-coded images (see 1.23) can have very high ground resolutions. They are taken in regular intervals by the mapping authorities for documentation purposes and to improve existing maps.

Satellites and their Usage

Several satellite systems of different nations are permanently observing the earth. Usually, they perform a multi-spectral scanning and often the data is available via ftp or http. These satellites are also used for purposes like weather forecasts or flood prevention. Well-known systems are *SPOT* and the *Landsat* system of *NASA*. These satellites are launched in regular intervals since 1972 and circle around the earth in sun synchronous, polar orbits. Their most important sensor, the *Thematic Mapper*, has 7 spectral channels and a pixel resolution of 30m x 30m. The produced data is stored and distributed via national services like the *USGS* in the US or the *ESA* and the *DLR* in Europe, see figures 1.24, 1.25, and 1.26.

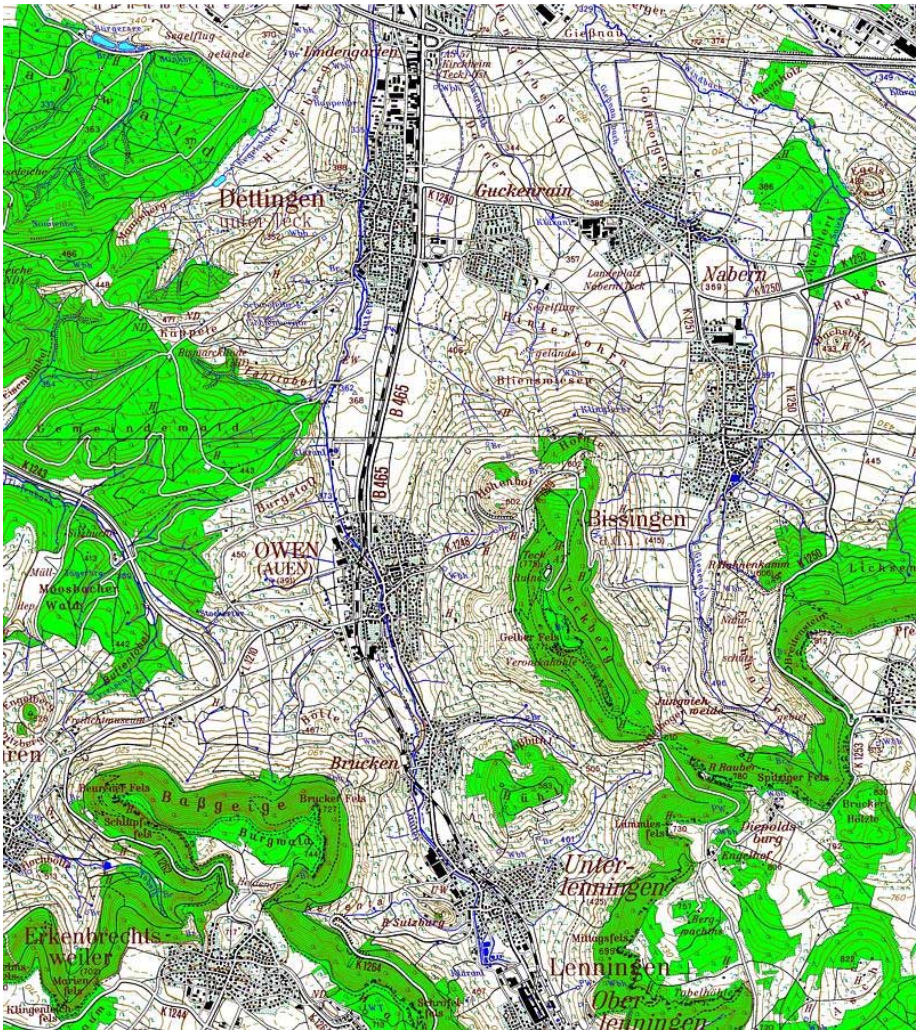


Figure 1.20: Topographic map with contour lines, see [3].

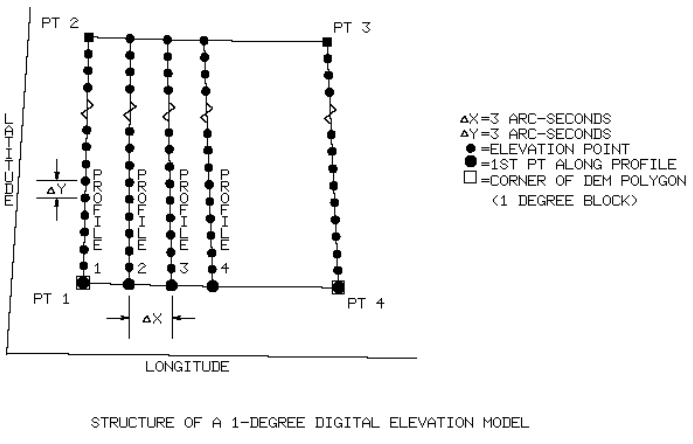


Figure 1.21: Structure of a *USGS* 1 degree digital elevation model, see [102].

Figure 1.22: Conversion of central projected aerial photos to orthographic, geo-coded images, see also [39]. On the upper left, the original image is shown, which is projected in the lower image with a central projection on a digital elevation model (DEM). From this, the orthographic, geo-coded image on the upper right is produced with an orthographic projection.

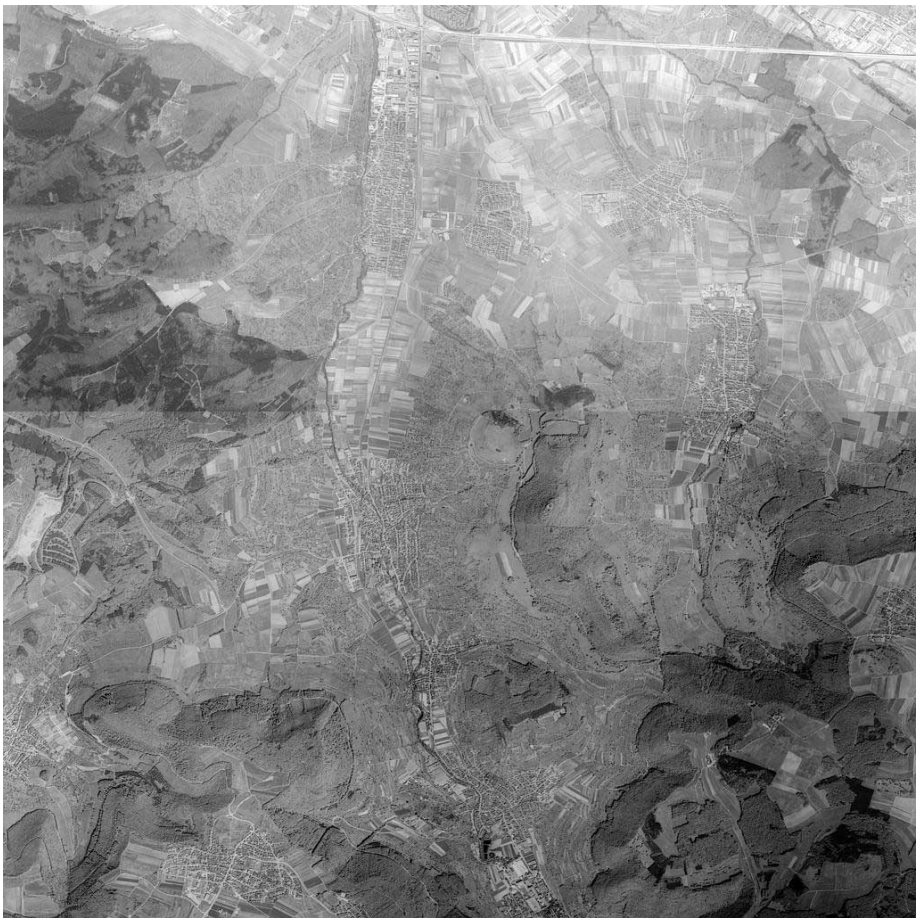


Figure 1.23: Orthographic aerial photography, having an original ground resolution of 25cm, see [3].

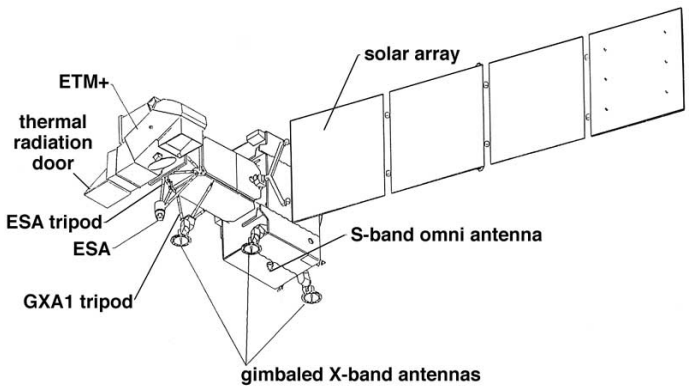


Figure 1.24: Scheme of the *Landsat* 7 satellite, see also [83].

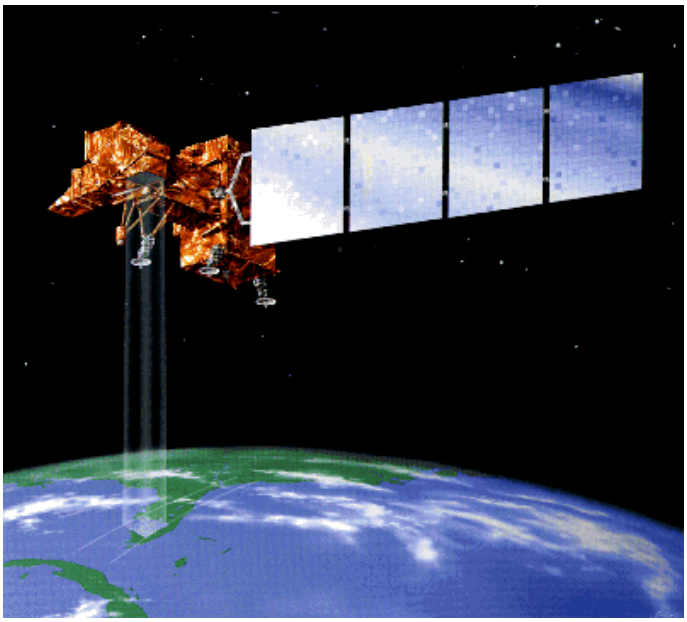


Figure 1.25: Landsat satellite in space, see [83].

Landsat Program - System Summary

System	Launch (End of service)	I(s)	Resolution (meters)	Communications	Alt. Km	R Days	D Mbps
Landsat 1	7/23/72 (1/6/78)	RBV MSS	80 80	Direct downlink with recorders	917	18	15
Landsat 2	1/22/75 (2/25/82)	RBV MSS	80 80	Direct downlink with recorders	917	18	15
Landsat 3	3/5/78 (3/31/83)	RBV MSS	30 80	Direct downlink with recorders	917	18	15
Landsat 4*	7/16/82	MSS TM	80 30	Direct downlink TDRSS	705	16	85
Landsat 5	3/1/84	MSS TM	80 30	Direct downlink TDRSS**	705	16	85
Landsat 6	10/5/93 (10/5/93)	ETM	15 (pan) 30 (ms)	Direct downlink with recorders	705	16	85
Landsat 7	12/98(est.)	ETM+	15 (pan) 30 (ms)	Direct downlink with recorders (solid state)	705	16	150

I(s) = Instrument(s)
R = Revisit interval
D = Data rate
*TM data transmission failed in August, 1993.
** Current data transmission by direct downlink only. No recording capability.

Figure 1.26: Table describing the properties of the *Landsat* satellites, see [83].

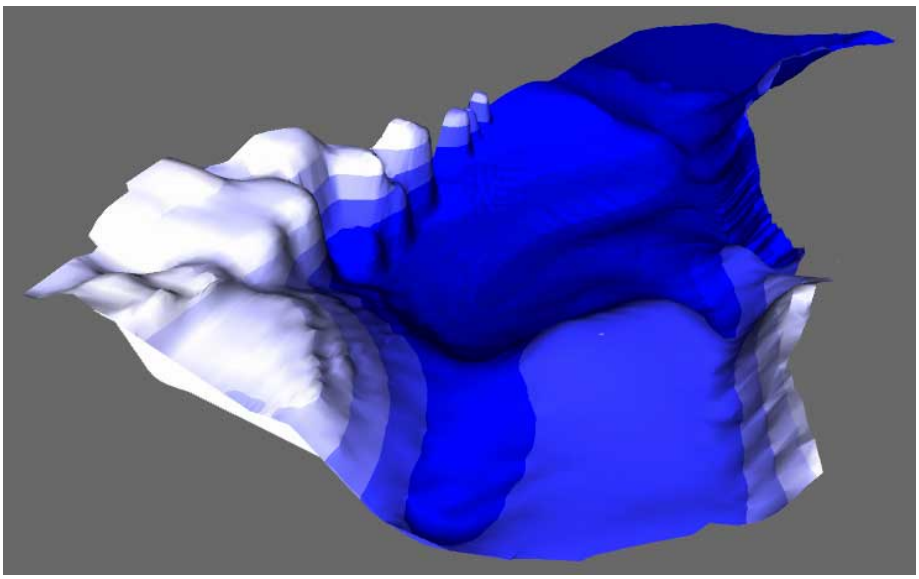


Figure 1.27: An isosurface extracted from a ground water simulation.

1.2.2 Other Data Sources

Several other data sources provide geo-related information, which gives additional content to the already mentioned data.

Data from Simulation and other Scientific Processes

Due to advent of modern computers in all scientific areas, complex processes in the natural environment can be modeled with the help of computers. Geo-related data results are available especially from geological or atmospherical simulations, see figure 1.27 and figure 1.28. But also new measurement technologies like seismic resonance measurements enable the scientists to produce data representing structures in the earth or the atmosphere, which have to be visualized to be understood and interpreted correctly.

CAD and Other Artificial Data

In the context of geo-related information, ”virtual” data, which is not measured or simulated, but planned artificially plays an important role in visualization. These data sets range from three-dimensional CAD models of houses to models of whole bridges, dams or freeways. The visualization of such models is able to show the impact of the planned constructions on the natural environment in which they shall be integrated, see figure 1.29 and figure 1.30.

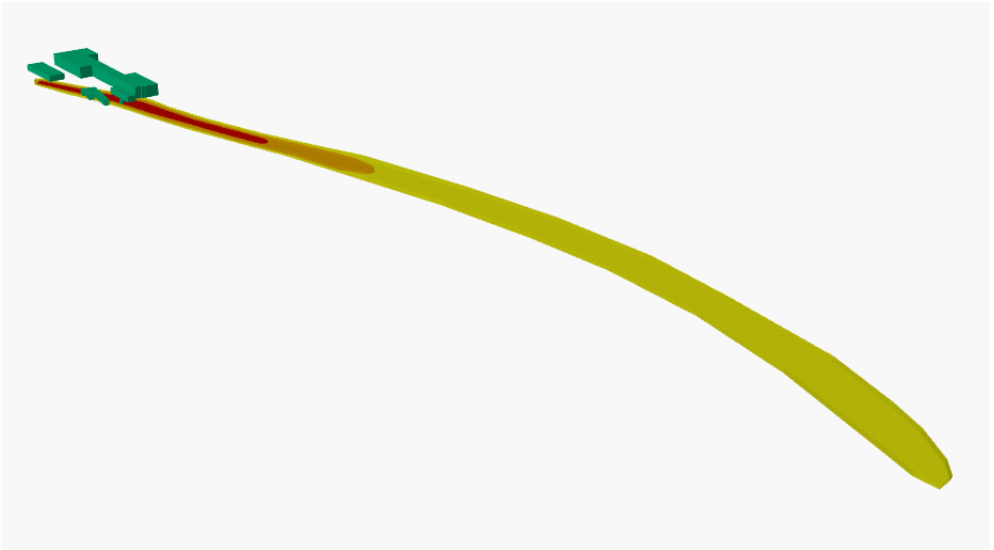


Figure 1.28: Three dimensional model of ground water pollution. The building is a model of the factory where this pollution occurred.

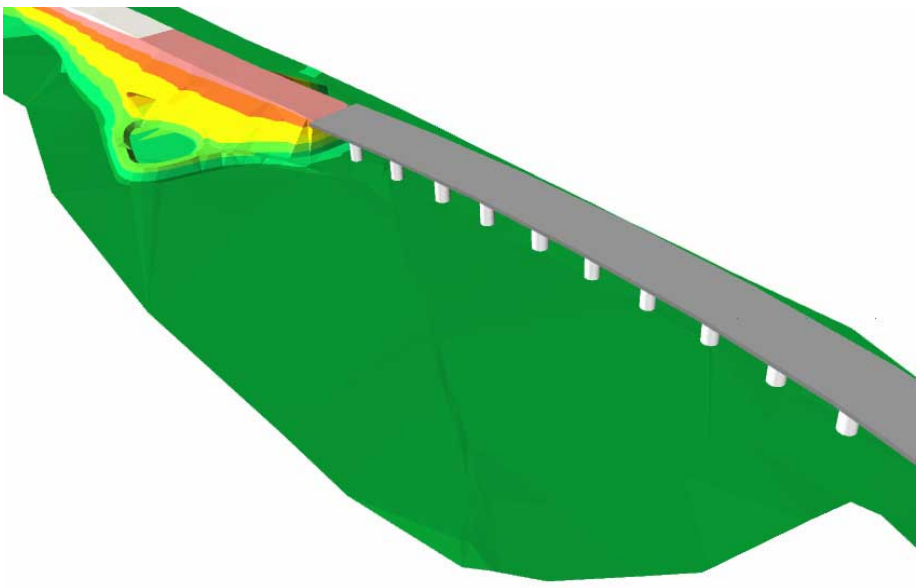


Figure 1.29: Model of a freeway.

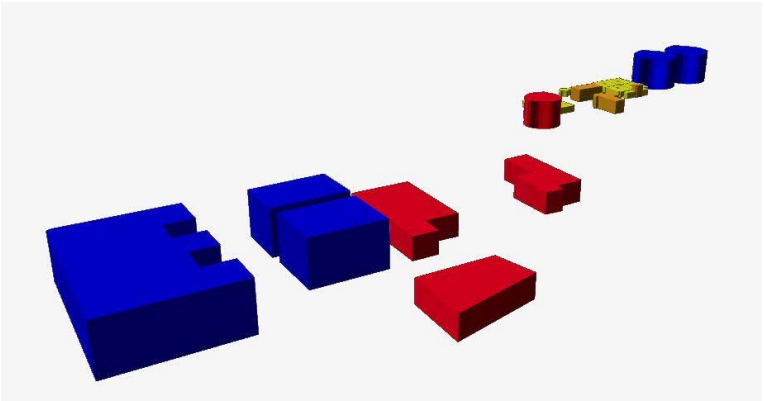


Figure 1.30: Model of a planned industrial area.

1.3 Motivation For The Following Chapters

In this chapter, the available geo-related data-sources were described. Most of their data-sets can not be visualized directly due to their size and special algorithms are necessary to perform this task.

In chapter two, a multiresolution algorithm is described that defines a very storage efficient multiresolution model for storing and reconstructing terrain and landscape surfaces. In the introduction of this chapter was already described, that the regular sampling of terrain surfaces can produce a huge amount of sampling data, see 1. Multiresolution techniques utilize the data reduction possible by exploiting perspective foreshortening to adopt such huge data sets to a certain viewer position. This adaptation process decreases the amount of data that has to be visualized significantly.

Another approach to reduce the amount of data processed and visualized is the fast determination of scene parts that are either outside the visible area, called view frustum, or inside but occluded and therefore also invisible. Imagine for example a landscape consisting of valleys and hills that obscure each other. Another example is a walkthrough of a extended architectural model like a city which is also a geo-related data set. Many houses are possibly visible in the view frustum, but they obscure each other and therefore, only a small part of the scene geometry is needed for the visualization. Chapter three deals with culling algorithms, that are quickly able to detect the visible parts of extended scenes or complex models. A new occlusion culling algorithm is defined, that does not rely on a special machine architecture. Furthermore, this algorithm can be implemented very nicely in hardware.

Texture mapping is a very popular way of visualizing information on surfaces. Unfortunately, very large textures like aerial photographs or maps are too big to be used directly as texture maps. Considering aerial photography, textures of a size of several Gigabytes (GB) are possible, since such photographs are taken with pixel resolutions of up to 25 centimeters, which means one pixel equals 25 centimeters of the original landscape. With this resolution, a terrain of 10km $\times$ 10km would require a texture of approximately 1.5 GB to texture it (40000 $\times$ 40000 pixels). Since the whole state of Baden-Württemberg (approximately 40000 km^2, one of the federal states of Germany)

is represented in pictures of this resolution, one can imagine what order of magnitude of texture data is common today. The handling of data sets in this Gigabyte-area is even on modern workstations a rather resource consuming problem, since these machines are able to handle operations on Megabytes, but not Gigabytes of data. Therefore, again reduction mechanisms offer the solution to enable these moderate equipped machines to use such data sets. Chapter four describes a camera adaptive texture data structure that performs this task.

Furthermore, an algorithm for the hardware accelerated selection of an arbitrary quadrilateral part of a texture is described. Such selections have to be performed for example when using maps for texturing, since they are usually distorted due to cartographic mapping.

Finally, a new texture filtering approach is described. Texture filtering is necessary during rasterization to ensure a correct texture mapping on all parts of a perspectively projected surface. The new filter uses precalculated weighting masks to perform the filtering operation. With this, a much better quality than using available filters can be achieved without giving up the possibility of an easy hardware integration.

Chapter 2

Geometric Multiresolution Models for Terrain Modeling

2.1 What is Multiresolution?

One essential key idea of data reduction is the paradigm of *adaptivity*. This means, that only parts of the available data are processed and the amount of processed data is dynamically adapted depending on a criterion which governs this adaptation process. One kind of criterions that allows adaptation is to exploit a viewer's position in a computer graphics scene. Depending on the distance to the viewer, different *levels of detail* (LOD) of the data can be used, since in the background almost no detail is necessary due to perspective foreshortening. Data structures, that store such LOD descriptions and are able to reconstruct and eventually combine different levels, are called *Multiresolution Models.*

Camera adaptive data structures and *Multiresolution Models* must have two basic properties to be useful and effective:

- Real time update must be possible.
- The criterion should apply no heuristics to govern the adaptation process but use instead exact calculations relying on known constants like screen resolution.

The advantages of adaptivity can be explained with the following example which describes the data set *Grand-Canyon-East* from the *USGS* (US Geological Survey):

- 120km x 120km landscape
- unzipped: 9839616 bytes
- gzipped : 1687238 bytes
- regular grid with 1453252 points, 2906504 triangles

As already depicted in the introduction in chapter 1, these *DEMs* (digital elevation models) are too huge to be rendered interactively. If the above DEM is converted into a multiresolution model as described in the following sections, approximately 15000 to 60000 triangles are necessary to render the model, depending on the actual position of the camera. Though, an interactive handling of this data set can be realized.

2.2 Description of Already Existing Multiresolution Approaches for Terrain Modeling

Triangular meshes are currently the most widely-used representation of terrain models in computer graphics. Planar polygons and triangles in particular are standard rendering primitives of common graphics workstations that can rapidly render polygons. Terrain data is usually reconstructed by photogrammetric modeling techniques and other acquisition techniques which produce many more triangles than necessary to visualize the terrain. The literature is rich on algorithms for reducing the number of triangles, by removing redundant vertices such that the simplified mesh satisfies some error tolerance [48].

In applications which require high fidelity visualization of a terrain, millions of triangles are needed. Even with recent progress in graphics hardware performance, it is impossible to achieve real-time rendering rates of such a large number of triangles. Efficient use of the number of triangles that are fed into the geometric pipeline is an important consideration [96]. Clipping out triangles which fall beyond the viewing frustum is necessary, as well as the use of hierarchical levels-of-detail (LOD) [24, 29, 112]. Low levels of a LOD hierarchy approximate the model more coarsely with fewer details, while higher levels contain finer details. The appropriate level is rendered according to the distance of the terrain section from the view point. Since coarse approximations contain fewer triangles, a large terrain area can be rendered faster with no visible penalty.

Usually the levels-of-detail are generated off-line in a preprocessing stage [22, 28, 112, 20, 95]. Since in general the distance of the observer to the terrain is less than the extends of the terrain itself, different levels-of-detail are necessary in different areas of the terrain. Such a multiresolution representation of the terrain should maintain spatial continuity, that is, the combination of different levels of detail should be seamless and leave no gaps or holes. In addition, triangles with sharp angles (slivers) should be avoided in all levels of the hierarchy. Usually, such triangles can induce numerical problems either when triangulating or when using the resulting mesh for other purposes. Thus, a Delaunay terrain triangulation [22, 30] is considered to be a guaranteed-quality mesh since it maximizes the minimal angles of its triangles. De Berg and Dobrindt [20] proposed a hierarchy of levels of detail that uses Delaunay triangulation at each level. Their method allows the different levels of detail to be combined in the same scene. Cohen-Or and Levanoni [12] have modified their method to form a tree structure which enables a top-down traversal of the Delaunay hierarchy with a fast culling mechanism. The major drawback with this approach is the usage of an explicit hierarchy that has to be stored and this hierarchy is approximately four times the size of the input vertices. Furthermore, the approximation error requirements for combining the different levels of detail are not calculated by exploiting perspective foreshortening and are therefore heuristics.

Hoppe [51] introduced another multiresolution scheme, the *progressive meshes*, which can also be used for terrain modeling (but not for Delaunay terrain triangulations). The basic idea of this approach is to simplify the original meshes by successive edge collapse transformations. The edge collapse transformation unifies two adjacent vertices into a single vertex and the two adjacent faces vanish in the process. The origi-

nal mesh can be restored by a sequence of vertex split operations (the reverse operation of edge collapse). For a vertex split operation only the vertex to be split and pointers to two of its neighboring vertices are necessary. In this way the whole hierarchy of intermediate levels of detail can be stored in a space-efficient way. As reviewed in [71], a selective refinement cannot be combined directly with the mesh compression described in the paper, since during selective refinement the whole topology information is needed which increases the amount of storage considerably. Furthermore, as Hoppe pointed out, a less restrictive hierarchy allows triangle foldings and weakens the control of the approximation error. This approach was tailored to the special needs of terrain surfaces in [52], where the storage requirements could be reduced by refining the data structure which separates the data into a static and dynamic part. The latter encodes the connectivity of the so called active mesh. The selective refinement may still produce, as pointed out by Hoppe, thin and near-degenerated triangles. Such triangles are prone to numerical problems which can occur during rendering or when the extracted mesh shall be used for other purposes.

Puppo [23] describes a general model for the multiresolution decomposition of planar domains into triangles. His method is based on a collection of fragments of plane triangulations arranged into a partially ordered set. Different levels of detail can be obtained by combining different fragments of the model. A similar method is used by Cignoni et al. [9]. In both approaches the topology of the hierarchy is stored explicitly, with no data compression. Since typical terrain models are extremely large, data compression is vital to enable storage of the model in the main memory of a workstation.

In the approach proposed by Lindstrom et al. [95], the original mesh is simplified in an on-line process during the rendering stage. In each simplification step, a viewer-dependent screen-space error resulting from the simplification is computed. This screen-space error controls the adaptation of the multiresolution data structure. It depicts, what error can be allowed at a certain position in the landscape between original data set and reconstruction while respecting after perspective projection a deviation of a user defined pixel number. Despite the fact that errors may accumulate the authors claim that for empirical data this effect is negligible. Interactive frame rates can be achieved by a compact and efficient quadtree. The simplicity of this data structure however has a drawback. To approximate an arbitrary straight line the quadtree has to be subdivided along the line up to the maximum level. For example, if the terrain contains small cliffs, just higher than the allowed error, numerous redundant triangles are generated.

In the next section, a new multiresolution Delaunay triangulation will be presented that can be generated on-the-fly by an incremental algorithm. As opposed to the data structures proposed in the above approaches, on-line multiresolution triangulation avoids the storage requirements of the hierarchy and the explicit determination of the number of levels. This approach was first presented in [68, 67] and extended in [66]. The terrain triangulation is updated dynamically as a function of the camera position and camera parameters, and is thus called a *view-dependent triangulation*. The proposed method inserts and deletes vertices based on an incremental Delaunay triangulation of points in the plane [73]. In [67] a *bottom up strategy* is used, where similarly to Puppo [95], the computation of the view-dependent triangulation starts with the coarsest triangulation. Then, for each triangle a screen-space error is computed.

If this error is larger than one pixel, an additional point of the Delaunay hierarchy is inserted to refine the proximity of the triangle. This process is repeated until the screen-space error of all triangles is small enough and therefore visually negligible. Although in a realistic fly-through there are regions where the triangulation doesn't change, this approach necessarily requires all the triangles to be checked.

Dynamic triangulation is not a new concept. Independent from this work described in [68, 67], other authors, see for example [24, 95, 95, 93], have used a view-dependent triangulation. However, they do not guarantee a geometric approximation error, and their approximation to the optimal triangulation is crude. One rather new approach uses a restricted quadtree data structure with fixed level of details (LOD) for the quadtree tiles, see [110]. To prevent cracks between the quadtree tiles, so called *seams* are precalculated that can bridge the detail gap between tiles of different LOD. One drawback of this approach can be visual artifacts due to toggling from one LOD to another. To prevent this, the algorithm has to switch very conservatively to the next LOD. Unfortunately, [110] does not adress this question of when to switch correctly to the next level. Therefore, the models available in this format show all significant 'popping' artifacts.

2.3 A new Multiresolution Model for Parametric Surfaces The Multiresolution Delaunay Approach

2.3.1 The Multiresolution Delaunay Approach and its Application to Terrain Modeling

In this Section, a view-dependent multiresolution triangulation algorithm is presented for a real-time flythrough. The triangulation of the terrain is generated incrementally on-the-fly during the rendering time. It will be shown, that since the view changes smoothly only a few incremental modifications are required to update the triangulation to a new view. The resulting triangles form a multiresolution Delaunay triangulation which satisfies a predetermined view-dependent error tolerance. The presented method provides a guaranteed-quality mesh since it has control over the global geometric approximation error of the multiresolution view-dependent triangulation. This approach will be called the *Multiresolution Delaunay* approach.

2.3.2 A View-dependent Error Function

A terrain is described mathematically by a bivariate elevation function $h : D \subset \mathbf{R}^2 \rightarrow \mathbb{R}$ defined over a rectangular domain D in the XY plane. This function is sampled at a finite set of points $P = \{p_1, \ldots, p_n\} \subset D$, forming the vertices of a triangulation approximation. This piecewise linear representation is known as a *triangulated irregular network* (TIN). A simple and common measure of the approximation error between the elevation function h and the approximating TIN is the maximum vertical distance between the two representations. In the following, this error is called *geometric approximation error*. The maximum geometric approximation error that cannot be perceived by an observer is called *allowable error*. This error is view-dependent and a function of the camera position and camera parameters. For its computation not

only the distance between camera position and a location in the XY-plane has to be considered but also the height of the terrain at that position. In a number of previous approaches the terrain topology was ignored. Due to the allowable error in pixel space, a geometric approximation error ϵ can be used whose projection onto the screen is smaller than one pixel (see also [95, 68]). Denoting the pixel size on the viewing plane by τ, the relation between ϵ and τ is then expressed by the following calculation (see the Figures in 2.1, 2.2, and 1.19):

$$\frac{\tau}{r} = \frac{f}{f + d_v} \Longrightarrow r = \tau \, \frac{f + d_v}{f} \tag{2.1}$$

$$\frac{s}{\epsilon} = \cos(90 - \beta + \alpha) = \sin(\beta - \alpha) \tag{2.2}$$

$$\frac{s}{r} = \cos(\alpha) \Longrightarrow s = \cos(\alpha)\, r \tag{2.3}$$

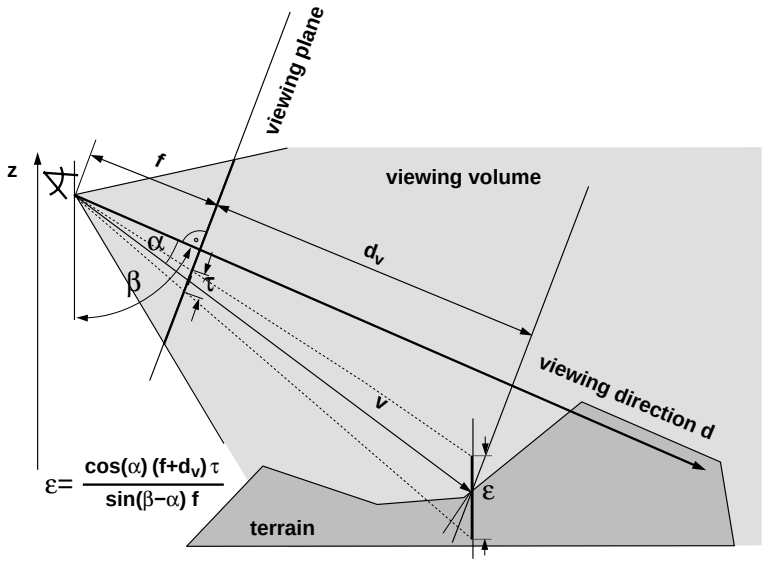


Figure 2.1: The maximum allowed error tolerance ϵ for the approximation of the topographic surface depends on the distance between the surface and the observer, on the angle α between the viewing direction d and the vector v between a surface point and observer, on the pitch angle β, and on the pixel size τ in the viewing plane.

This leads to

$$\epsilon \; = \; \frac{cos(\alpha)\; (f + d_v)\; \tau}{sin(\beta - \alpha)\; f}. \tag{2.4}$$

(α denotes the angle between the viewing direction d and v (as in Figure 2.1), the pitch angle is β, f is the focal distance, and d_v the distance between a terrain point p and the viewing plane, see Figure 1.19)

This expression defines the allowable approximation error for each sample point $p_i \in P$ as a function of its height $h(p_i)$, the camera position, and camera parameters, see Figure 1.19.

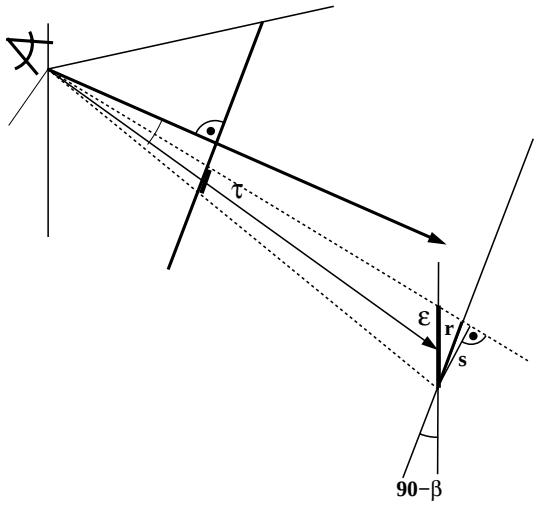


Figure 2.2: Schematic of the angles used.

2.3.3 The Algorithm

In a preprocessing step, a view independent approximation of the terrain is computed, yielding a Delaunay triangulation of the entire set of points. This approach was first described by [33]. Afterwards, the "history" of the triangulation is used by the on-line algorithm to compute the view-dependent triangulations.

The Off-Line View Independent Triangulation

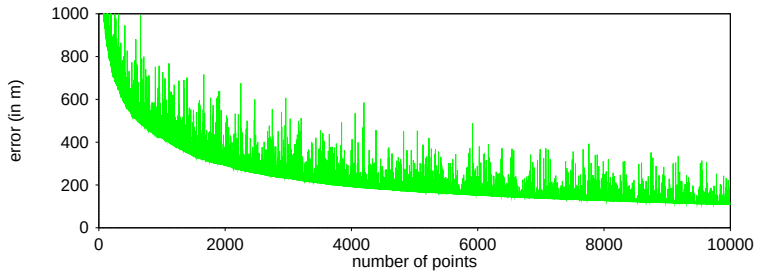


Figure 2.3: The insertion of a single point during the insertion process does not necessarily cause a decrease in the global geometric approximation error.

Let $\epsilon \geq 0$ be a global tolerance value for the whole data set, P be a finite set of points in the XY-plane, and h be the terrain elevation function, see Figure 1.19. Starting point is an initial constrained Delaunay triangulation, denoted by Σ_θ (a constrained Delaunay triangulation is a special form of Delaunay triangulation where edges can be marked as unremovable during point insertion or removal). The borders of this first triangulation Σ_0 of an arbitrary initial set of vertices are such constraints and remain static throughout the on-line process. This initial set of vertices contains at least the four corner points of the rectangular array. It may also include vertices and edges that

should be present in all the levels of details. For example, ridges of mountains, coasts of rivers or the borders of highways. The initial static triangulation is refined by an iterative insertion of new points, one at a time. The insertion is based on an incremental Delaunay algorithm in the XY domain [33, 73]. At each iteration, the point p which improves the maximum global geometric approximation error is inserted as a new point and the triangulation is updated accordingly. This error is measured between the actual refined triangulation and h. After the insertion of m points, the corresponding triangulation is denominated as Σ_m, indicating that it contains m additional vertices. The refinement process continues until the global geometric approximation error is less than the predefined ϵ. The corresponding final triangulation is called Σ_N. As illustrated in Figure 2.3, the insertion of a single point during the insertion process does not necessarily cause a decrease in the approximation error. However, the convergence of the method guarantees that the approximation improves after some additional vertices have been inserted [1].

The N inserted points transforming Σ_0 into Σ_N are stored in a sorted list L ordered by the insertion time. In addition, for each point p_n in the list the maximum geometric approximation error of the reduced triangulation Σ_n is recorded at the insertion time. Thus, each point p_n corresponds to a global geometric approximation error ϵ_n. In other words, if all vertices of the list L up to a certain point p_n are inserted into the initial triangulation, the global geometric approximation error of Σ_n is ϵ_n. Let $G_\epsilon(p_n)$ be the function which maps a given point p_n to its associated global geometric approximation error ϵ_n.

Given the initial Delaunay triangulation Σ_0 and the sequence of points $(p_1, p_2, \ldots, p_N)$, all the unique Delaunay triangulations $\Sigma_1, \ldots, \Sigma_N$ can be reconstructed via incremental point insertions. Therefore, the multiresolution representation of a terrain requires only an initial Delaunay triangulation Σ_0 of the domain in the XY-plane and the sequence of points transforming Σ_0 into the model Σ_N at full resolution. The topology of the triangulation is given implicitly by the use of the Delaunay triangulation and does not have to be stored explicitly. This leads to a massive reduction in the storage costs of the multiresolution model. It is important to point out that, with respect to the storage requirements, the storage cost is superior to all other techniques described above, including the progressive meshes. These techniques may considerably reduce the storage requirements for the representation of the topology, but they still have to store it explicitly.

During the on-line process the geometric error of each point needs to be translated to its allowable view-dependent error by Eq. 2.4. One way to achieve this is to compute the allowable error for each triangle in the current triangulation Σ_i and to refine the triangulation locally in the neighborhood of the triangle. This approach is used in [67] and in [95]. However, this necessarily requires that for every frame, even for small camera movements, all the triangles have to be considered. This causes many redundant checks since for small camera changes the allowable error of the vast majority of the triangles does not change.

To reduce the number of dispensable checks and to accelerate the computation of the allowable error the points $p_1, \ldots, p_n$ are partitioned into cells. The minimum and

[1] A promising approach to smoothen the error curve in Figure 2.3 is to replace the insertion algorithm by a removal algorithm [64]. This will also reduce the variation of the frame to frame update times.

maximum elevation values of all points in a cell define a bounding box which is used to compute an allowable error ϵ for the whole cell, (see Figure 2.4). As depicted in Eq. 2.4, ϵ at a position p in the direction of the height z depends on the given focal size f, on the angle β, the difference between the angles α and β, and the distance to the viewing plane. For $\alpha \equiv \beta$, the allowable error is unbounded. Increasing α while keeping constant the distance d_v to the viewing plane, decreases the allowable error. Therefore, to find an estimate for the representative error of the bounding box, the distance d_v of the cell's bounding box to the viewing plane and the maximum angle $\alpha(p_i) - \beta$ of all corner points p_i of the bounding box are needed. The distance d_v and the maximum angle $\alpha - \beta$ are computed by projecting the corner points p_i of the bounding box to the viewing plane and taking the minimum of the distances between the original points p_i and the projected ones and the maximum of the angle $\alpha(p_i) - \beta$, respectively. Note that since all bounding boxes have the same orientation, the same corner of the bounding boxes has the minimum distance to the viewing plane. As a byproduct, partitioning into cells also accelerates the access to the points and offers a fast culling mechanism, see also 3.1.11.

The number of cells depends on the local density of the points and also on the bandwidth parameters of the graphics system. Furthermore, hierarchically organized cells, see 3.1.11, can be used that are much better suited for using culling algorithms. For the Grand-Canyon example described later in this Chapter, a cell grid of 70×70 cells was used, whereas for the smaller 'Teck' data-set 25×25 cells were sufficient on small systems like PCs or SGI O_2. This constant has to be determined empirically, since it depends on many internal system parameters. This is no drawback, since a reconfiguration can be done at runtime and the multiresolution model itself has not to be recalculated. Furthermore, such a reconfiguration is a very fast process even for large models since it is only a sorting of points into cells.

The preprocessing stage can be summarized by the following steps:

1. Define an initial triangulation Σ_0.

2. Start with Σ_0 and insert, one at a time, the point improving the maximum geometric error. The point and its associated error are stored in the list L. This continues until the maximum geometric approximation error of the reduced triangulation is zero, or another prescribed user-defined error.

3. Choose an appropriate regular grid (see above) which bounds all the sample points $p \in P$. Distribute all the points $p \in P$ into the grid cells according to their XY coordinates.

4. Sort the points in each grid cell according to their corresponding global geometric approximation error and store them in local lists.

The On-Line View-Dependent Triangulation

The computation of the view-dependent triangulation is based on the ordered list L, the view-independent global geometric approximation errors, and the view-dependent global geometric approximation errors of the grid cells. For a given frame, first, the view-dependent global geometric approximation errors for the grid cells are computed.

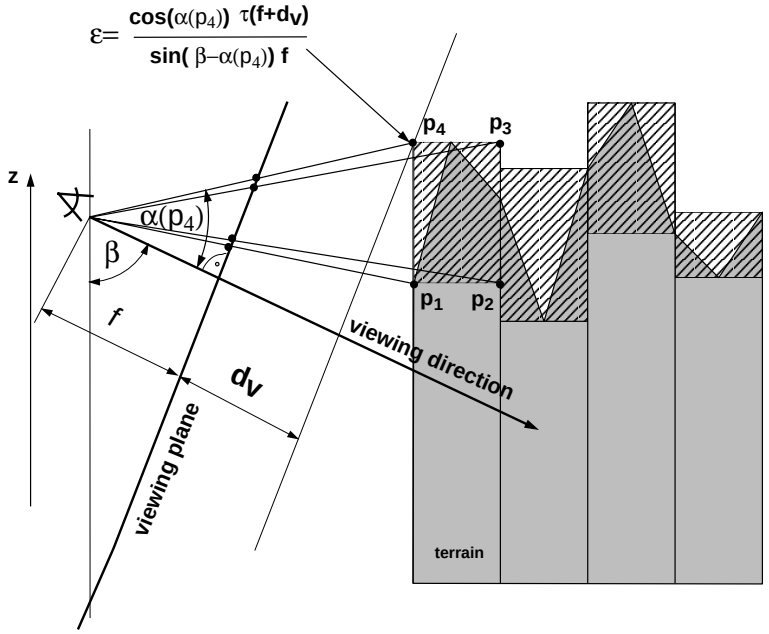


Figure 2.4: The elevation values define a bounding box for each cell. Using the smallest distance d_v to the viewing plane and the maximum angle $\beta - \alpha$ the allowable error ϵ for the cell is given by $\frac{cos(\alpha(p_4))\ (f+d_v)\ \tau}{sin(\beta-\alpha(p_4))\ f}$.

Then in the second step, the points of each grid cell which have a corresponding global geometric approximation error that exceeds the view-dependent error of the cell itself are inserted into the current constrained Delaunay triangulation. This can be done very efficiently since the points in each grid cell are sorted according to their global geometric approximation error. In practice, in most of the cells, no further points are inserted into the triangulation.

Unfortunately, the resulting constrained Delaunay triangulation $\mathcal{T}$ may contain triangles that do not belong to any of the triangulations $\Sigma_0, \ldots, \Sigma_m$. Since in the preprocessing step only for triangles of the intermediate triangulations $\Sigma_0, \ldots, \Sigma_m$ a global geometric approximation error was computed, the geometric approximation in the area of these new triangles is unknown. Note that in the worst case the amount of this error is in the same order of magnitude as the height of the terrain. Therefore, in a following *correction step* further points of the list L are inserted into the triangulation until all triangles of the resulting triangulation belong to one of the triangulations $\Sigma_0, \ldots, \Sigma_N$.

This further insertion step is based on a special observation: Let $\Delta(p_i, p_j, p_k)$ be a triangle of the triangulation $\mathcal{T}$. If the circumcircle of $\Delta(p_i, p_j, p_k)$ does not contain any points with corresponding insertion index less than $m = \max(i, j, k)$, then the triangle is contained in the intermediate triangulation Σ_m and approximates the terrain surface up to an approximation error ϵ_m, (see Figure 2.5).

Proof: Since none of the points already contained in the constrained Delaunay triangulation $\mathcal{T}$ with corresponding insertion index greater than $m = \max(i, j, k)$ belongs to the circumcircle of triangle $\Delta(p_i, p_j, p_k)$, their removal from $\mathcal{T}$ will not influence the triangle itself. Furthermore, due to the assumption that none of the points

with corresponding insertion index less than m belongs to the circumcircle of triangle $\Delta(p_i, p_j, p_k)$, inserting all these points with corresponding insertion index less than m will also not influence the triangle. Therefore, this triangle has to belong to the unique constrained Delaunay triangulation containing all points up to p_m, i.e., to Σ_m. Due to the first observation above this triangulation approximates the initial triangulation with an approximation error less than or equal to ϵ_m.

The above implies that the circumcircle of each new triangle has to be checked to insert more points if needed. It also shows that a new triangle can cause points to be inserted only in its proximity. In practice this locality leads to a small number of vertices (about 20 percent of all inserted vertices, see table 2.1) inserted during the correction step. Furthermore, practical experience shows that the correction step is not absolutely necessary during motion. Instead it is sufficient to utilize it when coming again to a stop.

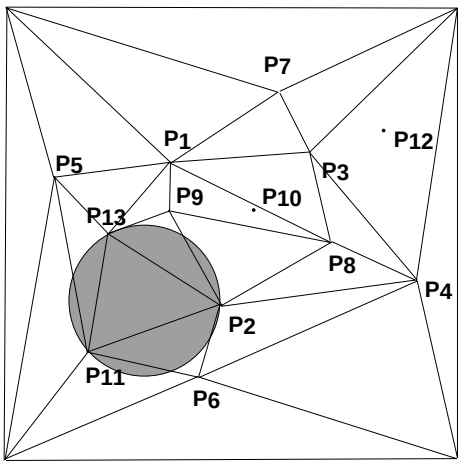


Figure 2.5: Since the circumcircle of the triangle $\Delta(p_{13}, p_{11}, p_2)$ does not contain any other point with corresponding insertion index less than 13, the triangle is contained in Σ_{13} and it approximates the terrain surface up to an approximation error ϵ_{13}, see [68].

The on-line view-dependent triangulation is based on the above and can be stated as follows:

1. For each grid cell C_j compute the maximum allowed approximation error ϵ_j by Equation 2.4. This defines a threshold for each point. During error calculation, some sort of *View Frustum Culling* (see 3.1.6) is already performed. Boxes outside the view frustum have no projection covering a part of the camera window on the viewing plane. They get an infinite error and are excluded from the update process. This culling can be accelerated with a hierachically organized bounding box structure.

2. Remove from the current constrained Delaunay triangulation all points $p_k \in C_j$ with $G_\epsilon(p_k) < \epsilon_j$.

3. Insert into the constrained Delaunay triangulation all points $p_k \in C_j$ with $G_\epsilon(p_k) > \epsilon_j$.

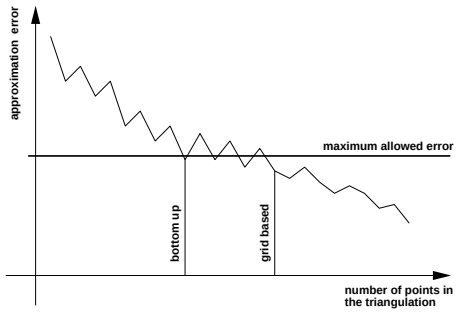


Figure 2.6: In a bottom up strategy the insertion process stops regardless of whether further insertion of points would increase or decrease the geometric approximation error. Thus, it achieves the minimum number of required points. In this approach all points of a grid cell whose geometric approximation error exceeds the allowable error computed for the cell itself are inserted.

4. Start with the point p_m which has the greatest index m among all corner points of the new triangles generated during the second and third step. Check if the triangles adjacent to p_m contain points in their circumcircle that have an insertion index less than m. If such points are found, insert all points p_l that are adjacent to p_m with respect to the triangulation Σ_m into the current triangulation. This guarantees that all triangles adjacent to p_m will belong to the triangulation Σ_m.

Note that the global geometric approximation error is independent of the view direction. Furthermore, this global geometric approximation error is also a local error: Before inserting the point p_n into the triangulation the maximum global geometric approximation error ϵ_{n-1} was caused by a point in the interior of the influenced area of p_n. The influenced area of p_n consists of the set of triangles which are deleted and replaced by a new set of triangles during the insertion of p_n.

Although this approach accelerates the update process between two consecutive frames compared with the bottom up strategy described in [67], it may happen that by this algorithm a small number of points are inserted into the resulting triangulation that are not necessary to guarantee the view-dependent allowable error. This is because the insertion of a single point during the insertion process in the processing stage does not necessarily cause a decrease in the global geometric approximation error. Using a bottom up strategy [67, 95], the insertion of points stops regardless of whether further insertion of points would increase or decrease the geometric approximation error. In this approach in each grid cell all the points whose geometric approximation error exceeds the geometric approximation error of the cell itself are inserted (see Figure 2.6).

The algorithm is advantageous for real-time rendering due to the frame-to-frame coherence in the temporal domain. Only a very small number of vertices has to be inserted and removed from frame-to-frame, and the triangulation can be updated incrementally on-the-fly (see Section 2.3.5).

2.3.4 Real-Time Performance

Temporal Continuity

Real-time rendering of levels of detail causes a noticeable temporal aliasing when the transition between different levels is sharp. Therefore, the transition between different triangulations has to be handled in a way that appears to be smooth, so that the temporal alias effect will not be noticed [11]. Hoppe [51] coined the term *geomorph* for the continuous transition between different geometries. In his method the transition between two triangulations is a split of a vertex and its continuous transformation to an edge, or an edge collapse into a vertex. This elegant solution cannot be adopted for the incremental insertion of a vertex into a Delaunay triangulation. In [12], a method based on object blending visually softens the transition between two levels of the Delaunay triangulation. The transition between two Delaunay triangulations consists of a series of edge collapse transformations. The method can also handle the insertion and removal of multiple vertices simultaneously.

Minimum Frame Rate

In visual applications there is always a need to balance the imaging quality and the frame rate. In interactive on-line systems one is required to maintain a user-specified minimal frame rate. In [35, 77] algorithms were proposed to adjust the image quality adaptively by choosing the level-of-detail and rendering algorithm according to its estimated rendering cost.

Since this multiresolution triangulation is incremental, in most cases the mesh generation time is insignificant in comparison to the rendering time. If the available rendering hardware is not fast enough to render a given triangulation in real-time, it is easy to adjust the error tolerance and trade the mesh accuracy for a coarser and "lighter" triangulation.

However, there is a need to guarantee that the new frame requires only a small incremental update of the mesh. For most practical flying trajectories the image footprint does not change much, unless there is a sharp rotational transition of the viewing direction (the yaw angle). A fast rotation of the yaw angle causes the image footprint to cover large areas which need to be refined for the new frame. The new areas can be displayed coarsely for a few frames until the mesh is updated. Instead it is preferable to maintain a wider footprint around the viewer location which can "absorb" fast rotational changes of the viewing direction [13]. Although the footprint is wider, not all its triangles are fed into the graphics pipeline, but only those which are in the viewing frustum. Maintaining a wider footprint increases the storage cost of the dynamic mesh, but this size is usually insignificant.

Storage and Traffic

A real-time flythrough has also to deal with other important issues, i.e., storage space and traffic overhead. In most real-world applications the terrain size is very large and cannot be loaded entirely in the main memory. Moreover, the access time to and from the memory may become a bottleneck unless some culling mechanism is employed which avoids redundant data movement.

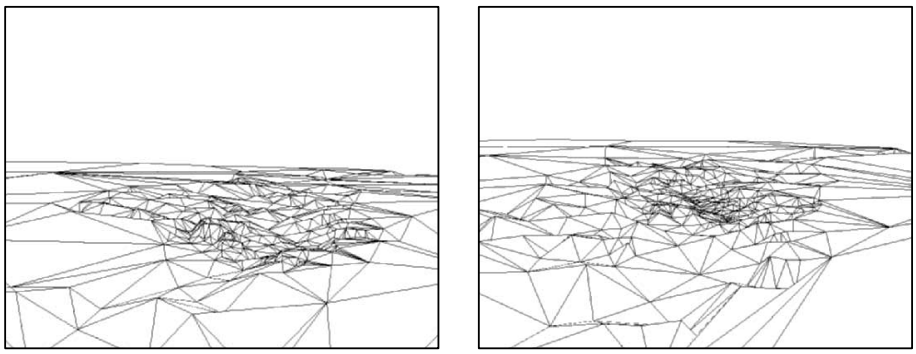


Figure 2.7: Camera dependent TINs of the Grand Canyon area. In the right picture, the viewpoint has moved in the direction of the canyon.

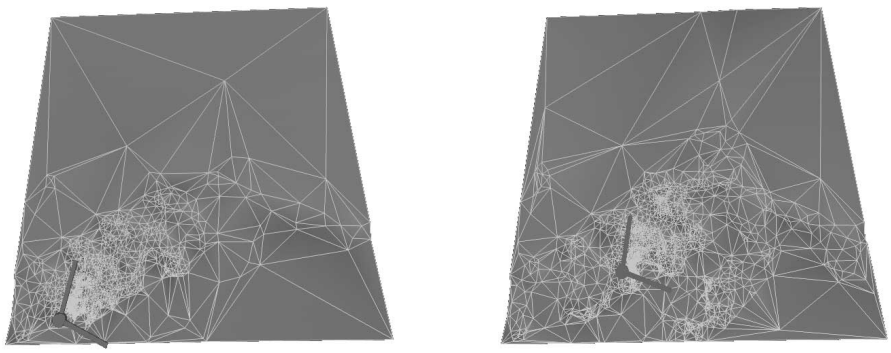


Figure 2.8: Camera dependent TINs with different camera positions.

In the off-line stage the terrain data is converted to a linear list of vertices and error values. As introduced above, it avoids the explicit storage of the mesh topology, and the data storage is thus compact. Assuming the list is stored on the disk and the dynamic mesh in main memory, the traffic between the disk and the memory is slight. Note that the access to the vertices in the list is fast due to the regular partition of the domain into cells. Since each cell contains a relatively short list the search is fast. Moreover, since the list is sorted by the error values, adjacent accesses in time are likely to be close along the list. Thus, for each list, a pointer is maintained to the most recently accessed vertex to further reduce the search.

2.3.5 Results

As a first test data-set, the elevation data of the Grand Canyon area was used. The original data set contains 1.440K vertices. Starting with an initial triangulation of four corner vertices of the original rectangular data, the rest of the sampled points were inserted as described above up to a geometric error of 5 meters. This results in an initial triangulation of about 490K vertices, and about 950K triangles.

For the flythrough, a velocity of about 700 km/h was assumed. At this velocity

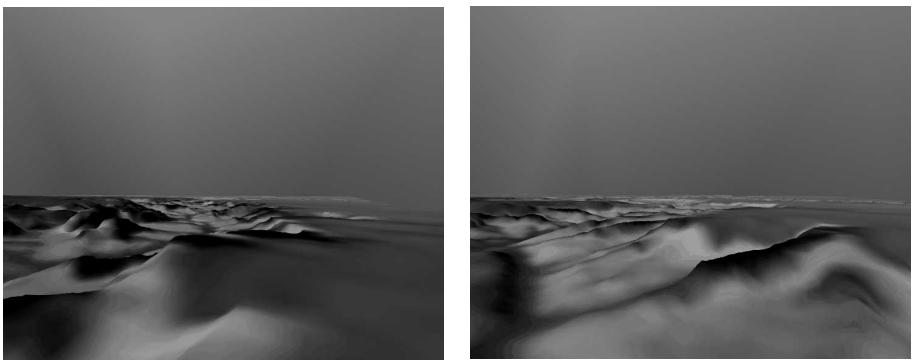


Figure 2.9: Camera dependent TINs with different camera positions.

image resolution (pixel)	200 x 200	400 x 400	720 x 576
vertices inserted per frame	2.5	3.5	10
vertices removed per frame	4	5	12
vertices inserted(correction step) per frame	1	1.5	2.5
triangles	6200	20500	42500
insertion (msec)	17	22	65
removal (msec)	20	30	50
correction step (msec)	6	14	40

Table 2.1: Update statistics for a part of the flight over the Grand Canyon area with about 700 km/h. All numbers presented are averaged for the whole flight.

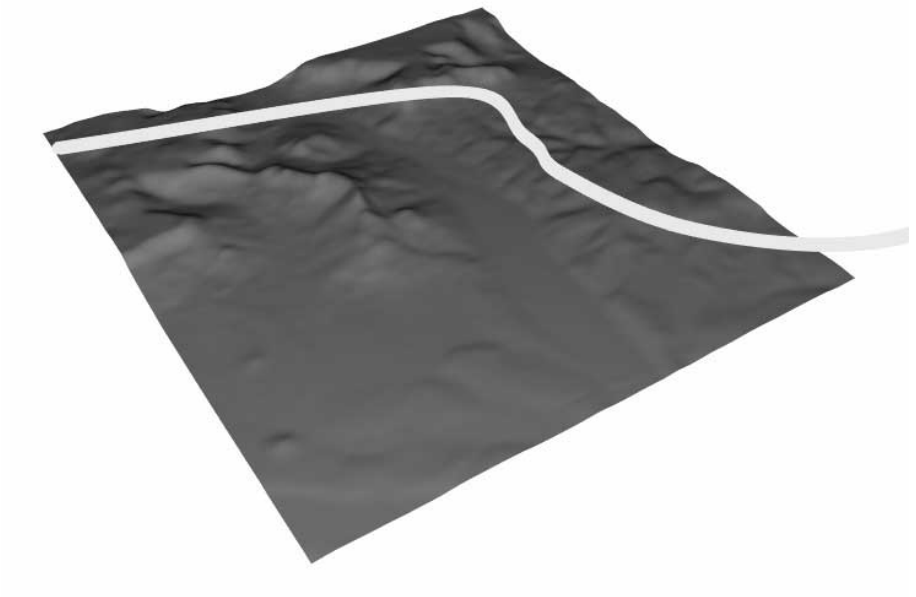


Figure 2.10: The flight path over the Teck-area in southern Germany.

it would take about 13 minutes to fly from one corner to the opposite corner of the terrain area. To realize a frame rate of 25 frames/sec at this velocity, the triangulation has to be updated every 7.6 meters. The focal size of the camera is 50 mm, which is equivalent to a camera with standard objective. The height of the flight over the terrain was between 700m and 5000m. Different levels of approximation are shown in Figure 2.7.

The two images in Figure 2.8 show the camera over the adaptive TIN of the Grand Canyon area. The eyepoint is visualized with a sphere, the two bars mark the boundaries of the viewing volume. Note that due to the dependencies in the hierarchy between different levels of detail in the multiresolution triangulation, there is still an excess of some triangles behind the camera.

The two images in Figure 2.9 show the Grand Canyon area with a simple texture calculated based on the height field from the same positions as in Figures 2.7 and 2.8.

As a second test data set, data of a small area in southern Germany was used. The data was provided by the 'Landesvermessungsamt, Baden-Württemberg, Germany'. The original data set contains about 30K vertices and 60K triangles. Figure 2.10 shows the path of the fly. It is 15.7 km long and the flight time with a velocity of 700 km/h is about 1 min 18 sec. Figure 2.11 shows the time needed to adapt the triangulation from one frame to the next one. It is clearly visible, that the update is nearly always possible with rates between 25 and 50 Hz.

For this data-set, aerial orthographic photographs are available. For the number of polygons achieved by the algorithm real time texturing becomes feasible. Figure 2.12 shows two example shots with texturing. Further considerations concerning texturing will be discussed in Chapter 4.

The frame-to-frame update rate for the TIN depends on the velocity of the camera,

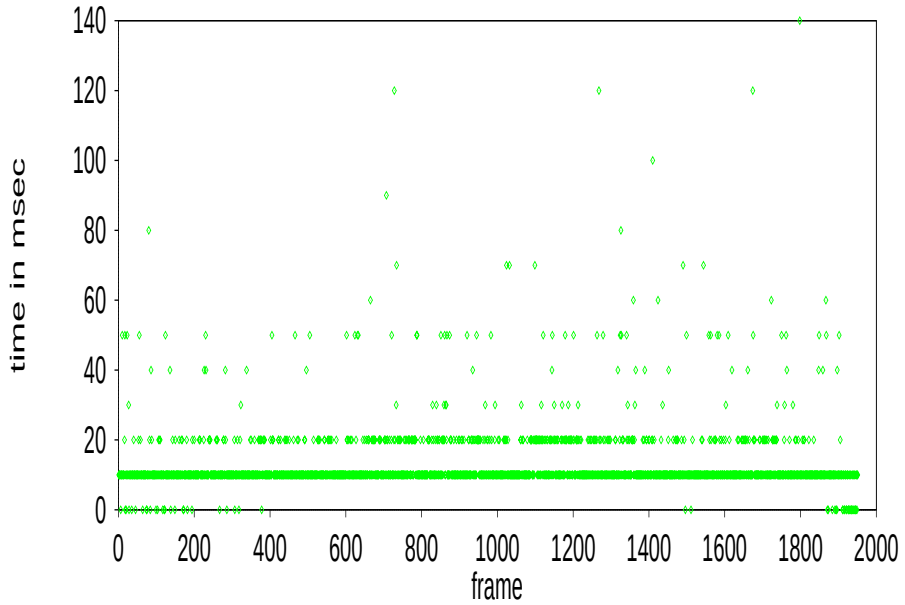


Figure 2.11: Time needed to adapt the triangulation from one frame to the next one during the flight over the Teck-area. The flight path is shown in Figure 2.10.

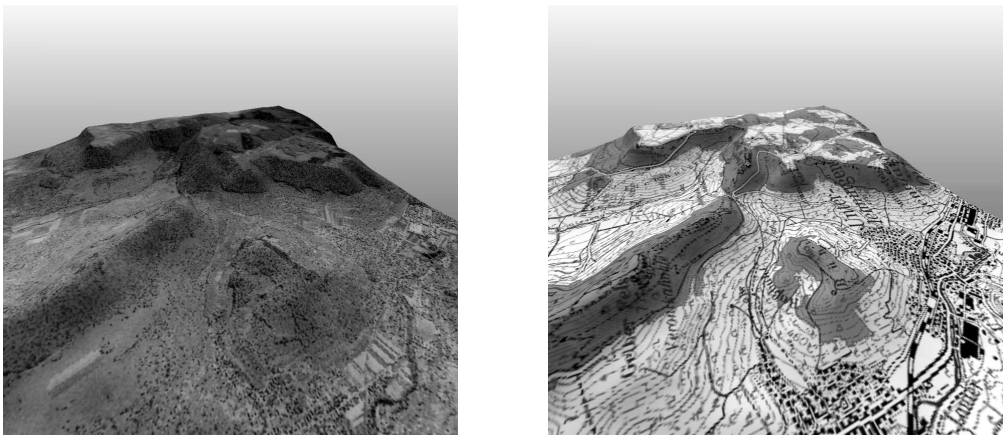


Figure 2.12: Images of the Teck-area. Left textured with an aerial photograph and on the right with a topographical map.

the distance between the camera and the terrain, the focal size, and the resolution of the image. In Table 2.1 the average data for the frame-to-frame update time and the number of inserted and removed vertices between two frames are listed for a small part of the flight at a height of about 1900 meters over the terrain. The update times were measured on an SGI O_2 with a 175MHz R10000 Processor. At an image resolution of 200x200, an update rate of 25 updates/sec (without rendering) can be achieved. At this resolution, only about 6000 triangles are necessary to represent the terrain. Assuming a frame rate of 25 frames/sec only 150000 triangles/sec should be rendered. Such a rendering performance is already available on small workstations or even on low cost PCs.

2.3.6 Comparison with Other Approaches

It is very difficult to compare the performance measurements of the different publications, since each author has his own data-sets that are usually not public domain and there exists no set of benchmarking data for heightfields and terrain. Furthermore, older publications are often not usable since the machines used there to produce the results can not be compared to the actual much faster computers. Therefore, two recent publications, [89] and [52], were used as comparison candidates.

It can be clearly stated that the Multiresolution Delaunay approach is equivalent in performance to [89] for small data sets up to 50,000 triangles. Unfortunately, no values of memory consumption or the application to bigger models are given in [89].

For another recent approach described in [52], it is much harder to make a comparison. First of all, this approach uses a varying error tolerance for the screen space error up to several pixels whereas the approach described in this chapter maintains and respects the screen space error criterion. Interestingly, also a part of the Grand-Canyon area was used for performance measurements. First of all, the data-set used was pre-simplified from 16,777,216 to 1,453,154 triangles before calculating the multiresolution model. To store the multi-resolution model, [52] needs 50 MB whereas the Multiresolution Delaunay approach stores a multiresolution model with 2,800,000 triangles of the Grand-Canyon area in 23 MB. Therefore, a much better storage efficiency can be claimed. To compare performance is nearly impossible. [52] uses not only a variable screen space error up to four pixels but the results were calculated on a SGI Octane MXE 195 MHz having a much higher internal system bandwidth compared to the O_2. Both algorithms realize approximately 20-30 frames per second for the Grand Canyon, but there is not enough data available for a detailed comparison.

Looking at a completely different approach, in [13] a heightfield was converted to a highly oversampled voxel data-set and visualized with a ray-casting algorithm on a special parallel hardware prototype. This approach needs 32 processors to achieve a framerate of 11-17 frames per second but can not be realized on current graphics systems. Furthermore, the original sampling of the terrain data-set was not mentioned in the publication , where a 55km$\times$80km piece of landscape was visualized. Sampling this with a widely used 50m grid, one gets a height field consisting of approximately 1,700,000 sample points or approximately 3,400,000 triangles. Therefore, the data-set dimensions can be compared to the Grand Canyon example used in this chapter. The voxel data-set itself needs 17.6 GB of disk storage. Together with the additional needed hierarchy levels, a total amount of 23.5 GB storage is needed for the model. This is a

tremendous amount of memory compared to the storage needs of the Multiresolution Delaunay approach.

Some older approaches that use also Delaunay triangulations are interesting candidates for a comparison. In [21], a Delaunay triangulation was used to combine different levels of detail (LOD). Unfortunately, no correct screen space error was evaluated and no performance measurements for dynamic updates during a flythrough were measured. Therefore, this paper can not be compared to the approaches cited above or the Multiresolution Delaunay approach. Nevertheless, it was one of the first papers that showed the usefulness of triangulations for terrain modeling.

In [12], this approach was improved to prevent aliasing due to the transition between different levels of detail while evaluating the precalculated levels of a Delaunay hierarchy. Again, no correct screen space error was calculated and the paper claims a framerate of approximately 10 frames by having three distinct levels of detail and adopting a mesh containing 5,000 triangles in average.

2.4 Further applications of the Multiresolution Delaunay approach

2.4.1 Approximation of NURBS Surfaces

Trimmed NURBS surfaces are a very popular way to represent the surfaces of objects in current available CAD-systems. Examples of such surfaces are ship hulls, the outer surfaces of airplanes, or the interior of cars. For visualization purposes, the surfaces are approximated by triangles. Even for small parts like the door of a car, thousands of triangles are needed for such approximations. The IRIS Inventor^{TM} built-in approximation tool e.g. uses 434,107 triangles to visualize the car door of Figure 2.13.

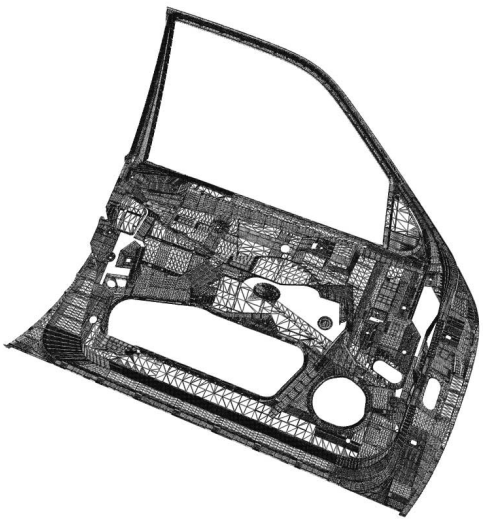


Figure 2.13: Triangle net of a car door approximation generated by IRIS Inventor^{TM}.

This approximation is independent from the viewing parameters and, therefore, independent from the door size in pixel space. The real-time visualization of a whole car in such a way is difficult even on high-performance graphics workstations.

In [69] is described how a CAD-model given as a set of trimmed NURBS patches can be approximated by a position dependent discrete multiresolution model resulting from an application of the Multiresolution Delaunay Approach described in the last section.

This discrete model consists of a coarse initial triangle mesh, a set of vertices, and a rule on how to insert these vertices into the initial triangulation in order to get different levels of detail (LOD). The main idea is to compute different LODs, instead of using special data structures to store them. This way storage capacity and storage access time are traded for computing power. In addition, only a few points of the actual triangle mesh have to be removed or inserted in order to switch between different LODs.

The algorithm is easy to implement and fast enough to achieve real-time updates for the visualization of common CAD models. Examples from real data illustrate the

power of this approach.

CAD models of a car door and a car felly were used as test cases. The original data set of the door consists of 446 trimmed NURBS patches and is approximated by IRIX Inventor^{TM} by a triangle net of 434,107 triangles, see Figure 2.13. The car felly containing 36 trimmed NURBS patches is approximated by IRIX Inventor^{TM} by 164,586 triangles.

The insertion time for one vertex is $\approx$ 0.45 msec and the time for removing one vertex $\approx$ 0.35 msec on a Silicon Graphics Workstation with R8000 processor and 150 MHz, see tables 2.2 and 2.3, which means real-time operation in practice.

Car door:

number of vertices	number of triangles	approximation error in mm
16,133	19,817	0.1
9,848	12,318	0.5
6,870	7,962	1.0
3,652	3,254	4.0
Inventor^{TM}:		
unknown	434,107	unknown

Car felly:

number of vertices	number of triangles	approximation error in mm
48,524	88,671	0.1
47,018	75,106	0.5
40,658	86,828	1.0
11,135	19,513	4.0
Inventor^{TM}:		
unknown	164,586	unknown

Table 2.2: Number of vertices and triangles needed to achieve the corresponding approximation error of the car door and the car felly.

User-defined Approximation Error

The following Figures 2.14-2.16 show three different levels of a car door in the resolutions of 0.1mm, 1.0mm and 4.0mm.

approximation error in mm	number vertices to insert	update time in msec
$4.0 \rightarrow 1.0$	3,218	1,320
$1.0 \rightarrow 0.5$	2,978	1,260
$0.5 \rightarrow 0.1$	6,285	2,940

approximation error in mm	number vertices to remove	update time in msec
$0.1 \rightarrow 0.5$	6,285	2,230
$0.5 \rightarrow 1.0$	2,978	1,020
$1.0 \rightarrow 4.0$	3,218	1,130

Table 2.3: The update time for changing the approximation error of the car door by inserting vertices (top table) or by removing vertices (bottom table). The resulting time for inserting one vertex is ≈ 0.45 msec and ≈ 0.35 msec for removing one vertex.

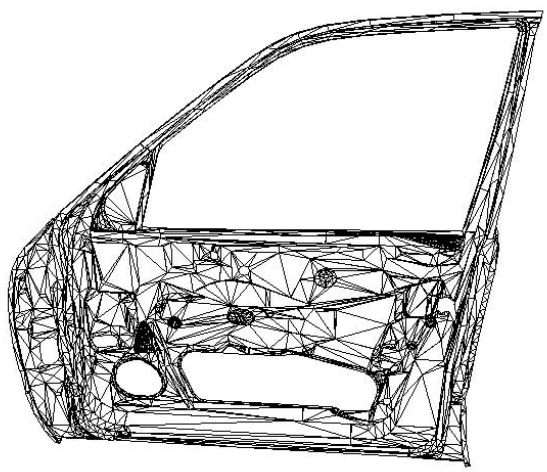


Figure 2.14: Triangle mesh of a car door with approximation error 4.0mm with 3,254 triangles .

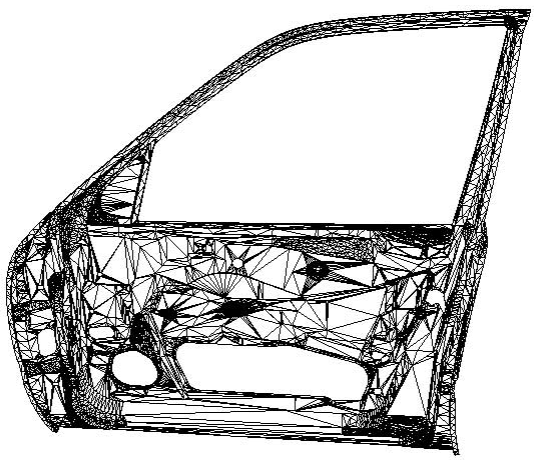


Figure 2.15: Triangle mesh of a car door with approximation error 1.0mm with 7,962 triangles .

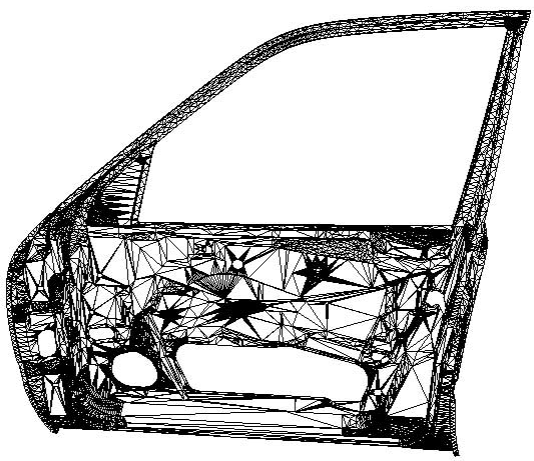


Figure 2.16: Triangle mesh of a car door with approximation error 0.1mm with 19,817 triangles .

Approximation Error Dependent on Viewing Parameters

The following pictures 2.17, 2.18 and 2.19 show a car felly in different distances from the observer. It has a length and height of 0.41m and a depth of 0.25m.

Figure 2.17: Car felly in a distance of 0.5 m from the observer with image size 150x150 and the corresponding wireframes with 14,489 points and 25,661 triangles.

Figure 2.18: Car felly in a distance of 0.5 m from the observer with image size 81x81 and the corresponding wireframes with 7,344 points and 12,606 triangles.

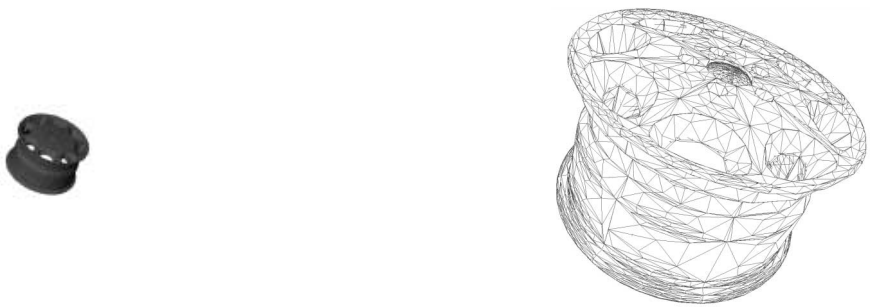


Figure 2.19: Car felly in a distance of 0.5 m from the observer with image size 47x47 and the corresponding wireframes with 4,035 points and 6,641 triangles.

2.4.2 Approximation of Radiosity Functions on Surfaces

Global illumination approaches are essential for getting a realistic impression of a virtual reality (VR) scene to increase the user's acceptance. These approaches try to simulate the lighting distribution in a scene by approximating it with the help of a simulation model. The models are constructed with the help of the physical laws governing the distribution of light, for example reflection and refraction principles. Two well-known approaches are the raytracing approach and the radiosity approach.

One problem of global illumination in VR scenes is the costly calculation of the illumination solution due to the great amount of data that has to be handled. To calculate and visualize the illumination solution in a radiosity approach, the tessellation of the geometry is usually not sufficient and has to be refined to represent illumination changes steadily.

The Multiresolution Delaunay approach can be used to reduce the amount of tessellation data to be able to visualize these solutions in real-time. Radiosity methods are widely used for realistic walkthroughs of static architectural scenes, since the radiosity solution is viewer independent. It is therefore important to maintain interactivity also on small systems. This can be achieved with adaptive data structures like the Multiresolution Delaunay approach.

Current research efforts try to enable radiosity methods to be able to handle dynamic scenes, where objects or light sources can move around. Non static scenes require a fast update possibility of the radiosity solution to maintain interactivity, see [32]. These approaches are not yet fully applicable and in the following only static scenes will be discussed.

In [70] is described, how a remeshing of the radiosity meshes can be performed by using the Multiresolution Delaunay approach. The result of this remeshing is an adaptive data structure, which has the advantages of viewer adaptivity (see the previous sections) and the possibility of incremental transmission, see Section 2.5.

Due to strict bandwidth limitations, approaches for a distribution of VR applications over the Internet are difficult to realize, see [90]. In this context, factors like scene size and the ability to transmit the data incrementally gain importance. The Multiresolution Delaunay approach is very well suited for this, see Section (2.5).

Data Reduction for Radiosity Meshes

Usually, triangle meshes are utilized during the solving process of the radiosity equation. If piecewise constant basis functions approximate the radiosity function, the triangle mesh used for the calculation can be directly used for the visualization.

The generation of these triangle meshes can be done with different approaches:

- Regular Subdivision:
  Regular subdivision schemes aren't adaptive at all. Therefore the surface is sampled with the sampling resolution imposed by the finest detail of the radiosity function. This oversampling can produce great amounts of additional, but unneeded triangle data.

- Adaptive Subdivision:
  With adaptive subdivision, only surface areas, where changes of the the radiosity

function occur, are tessellated with more triangles. These tessellation schemes have to be able to cope with fine structures, since otherwise the problem of figure 2.20 can occur, where fine structures are lost due to a coarse initial triangulation.

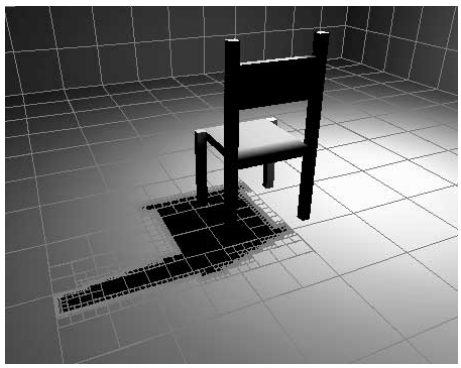


Figure 2.20: The shadow of the chair's right leg is not represented correctly.

- Hierarchical Subdivision:
  Hierarchical subdivision schemes usually utilize quadtrees. Quadtrees can be easily adapted to a needed resolution. Naive ways of triangulating a quadtree would result in T-vertices, which can produce cracks and other artifacts like holes during rendering.

  To prevent this, so-called *restricted* quadtrees are used. The subdivision depth of cells being neighbors are restricted in these quadtrees to one level. Therefore, only a restricted number of possible subdivisions exist which can be triangulated with a set of fixed schemes, see [107] and [5]. In figure 2.21, one disadvantage of restricted quadtrees is shown. Small structures tend to extend themselves due to the subdivision limitation for adjacent cells over a greater area. They impose a finer triangulations on areas, where this resolution is not needed.

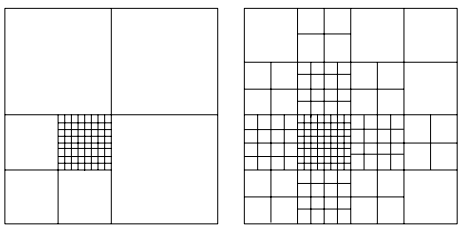


Figure 2.21: On the left side, a standard quadtree is shown, whereas the quadtree on the right side is restricted.

The following sections will describe, how a remeshing of the resulting triangle mesh can be done by using the Multiresolution Delaunay approach. The result of this remeshing is an adaptive data structures, which has the advantages of viewer adaptivity (see the previous sections) and the possibility of incremental transmission, see Section 2.5.

Definition of the Radiosity Function

Let $F: \Omega \longrightarrow \mathbf{R}^3$ be a parameterized surface patch with its parameter space Ω. The radiosity function

$$B: \Omega \longrightarrow \mathbf{R}^d,$$

is a vector valued function. The dimension d depends on the number of color channels. Usually, $d = 3$ is utilized for the three channels red, green, and blue.

By using the Multiresolution Delaunay approach, the radiosity function B can be approximated on the patch F by a piecewise linear function $B: \Omega \longrightarrow \mathbf{R}^d$ with the following properties:

- The global approximation error is smaller than a user defined error limit ϵ.
- The geometry of patch F is also well approximated by the piecewise linear approximation F'.
- A fast change between different approximation qualities is possible.

Definition of the Approximation Error

As depicted in the last paragraph, the approximating triangle mesh has to respect on one hand the surface geometry of F and on the other hand, the radiosity function B shall be well approximated.

Let

$$G(u, v) = (F(u, v), B(u, v))$$

be a function combining F and B.

$$G: \Omega \longrightarrow \mathbf{R}^{d+3}$$

is approximated by a function

$$G': \Omega \longrightarrow \mathbf{R}^{d+3}$$

with

$$G'(u, v) = (F'(u, v), B'(u, v))$$

By using RGB color space, an approximation error can be defined by a modified supreme norm, where the color channels red, green, and blue are weighted by their human's eye sensitivity:

$$\begin{aligned} ||G - G' \quad || = \\ \sup_{(u,v)\in\Omega} (\\ & |F_x(u, v) - F'_x(u, v)| + \\ & |F_y(u, v) - F'_y(u, v)| + \\ & |F_z(u, v) - F'_z(u, v)| + \\ & 0.3|B_r(u, v) - B'_r(u, v)| + \\ & 0.59|B_g(u, v) - B'_g(u, v)| + \\ & 0.11|B_b(u, v) - B'_b(u, v)| \\ &) \end{aligned}$$

Having these definitions, the algorithm described in Section (2.3.3) will produce a multiresolution model of the radiosity function and will respect the geometry of the patch.

Figures 2.22-2.32 show some test scenes, where the effect of using an adaptive data reduction paradigm can be clearly seen.

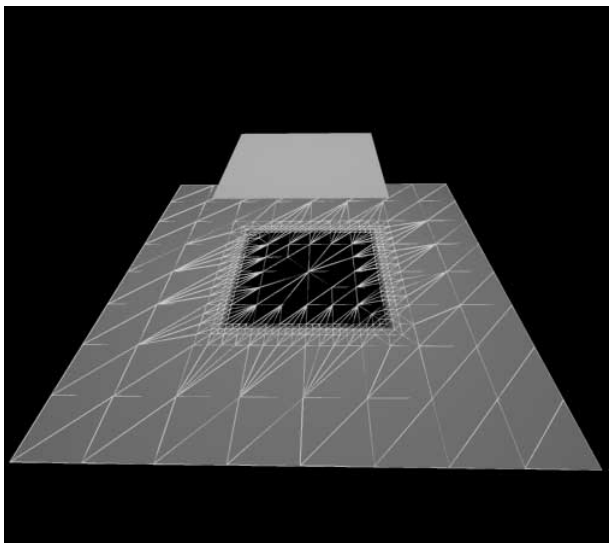


Figure 2.22: Test scene with point light source and sharp shadows on the floor. The floor surface was adaptively triangulated during the radiosity calculation. The triangulation contains 836 triangles. By using a regular grid and the resolution needed to sample the shadows correctly, a triangle mesh with 8192 triangles is produced.

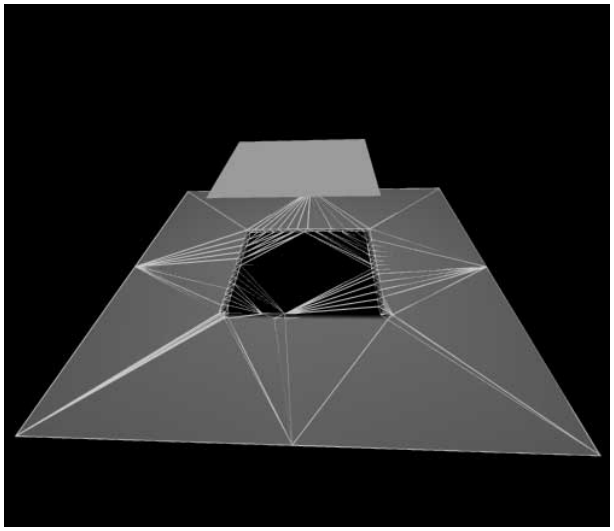


Figure 2.23: The same test scene as in figure 2.22 contains after data reduction only 234 triangles. There is no loss of visual quality visible.

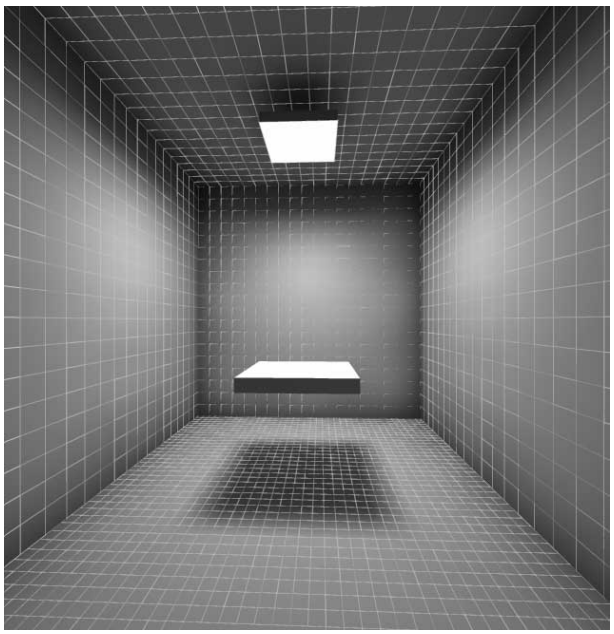


Figure 2.24: Test scene with an area light source and smooth shadow transitions. The regular triangulation contains 9760 triangles.

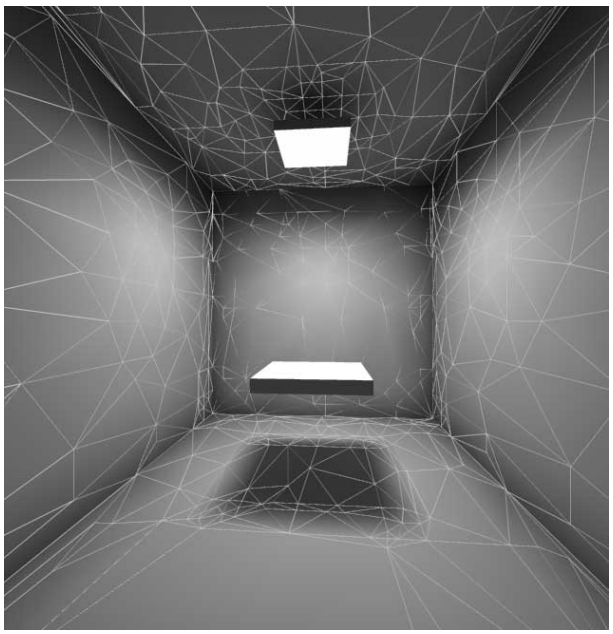


Figure 2.25: After the reduction step of the meshes shown in figure 2.24 with an approximation error of $\epsilon = 0.05$ (R, G, B normalized to 1), there remain 1187 triangles. Again, no loss of visual quality can be determined.

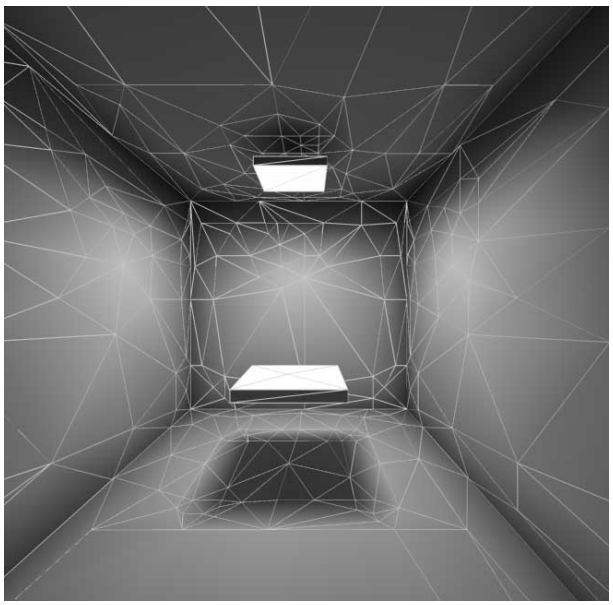


Figure 2.26: Using an approximation error of $\epsilon = 0.1$ (R, G, B normalized to 1), only 571 triangles are needed and the reduction rate is nearly twice the one of figure 2.25. The visual quality is a little bit reduced due to slight machband effects.

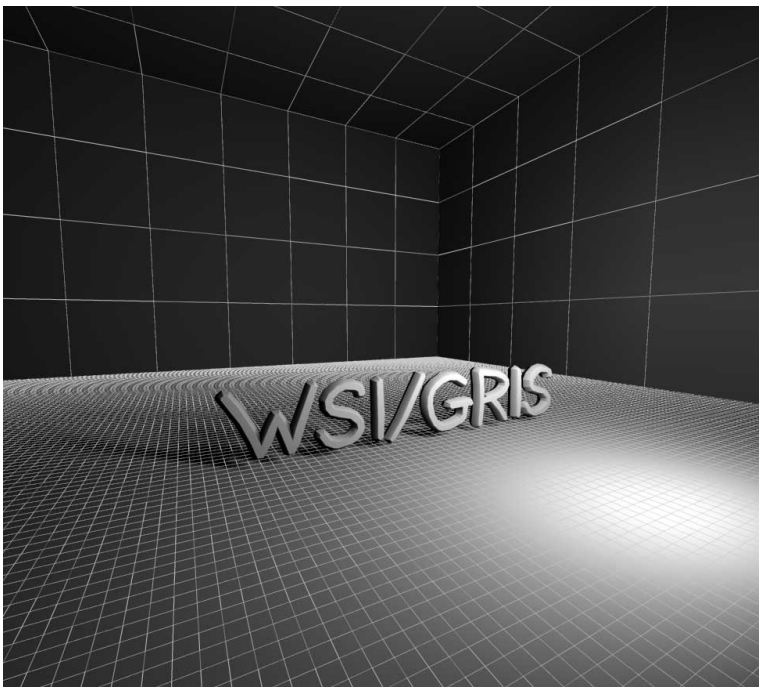


Figure 2.27: More complex radiosity scene, regular subdivisions are used for the radiosity meshes.

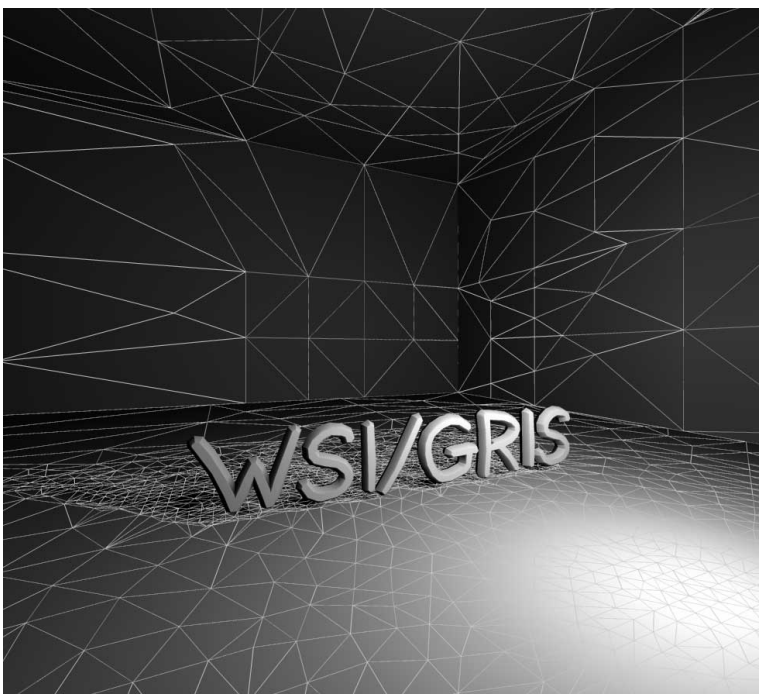


Figure 2.28: The radiosity meshes are reduced by the Multiresolution Delaunay approach.

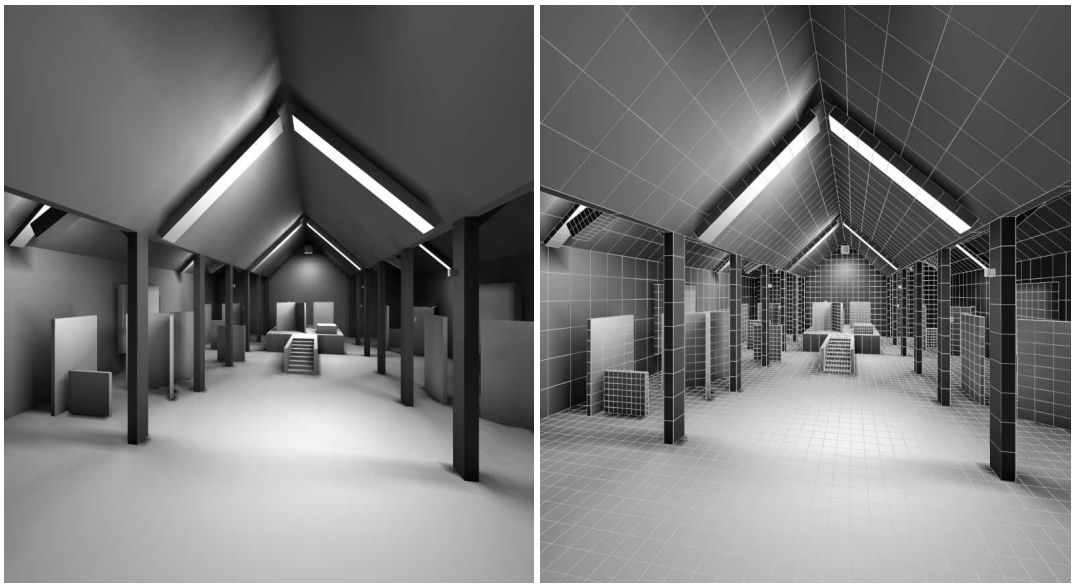


Figure 2.29: Room scene with meshing after radiosity-calculation

Figure 2.30: Remeshing with $\epsilon = 0.075$, reduction 44%

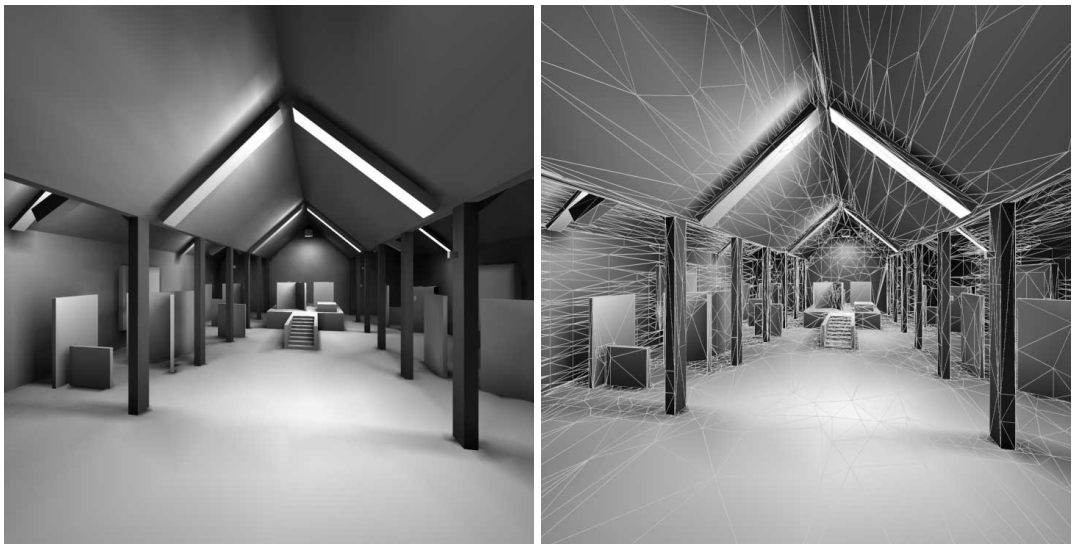


Figure 2.31: Remeshing with $\epsilon = 0.1$, reduction 52%. Please note, that up to now, no visual difference to the original scene is visible.

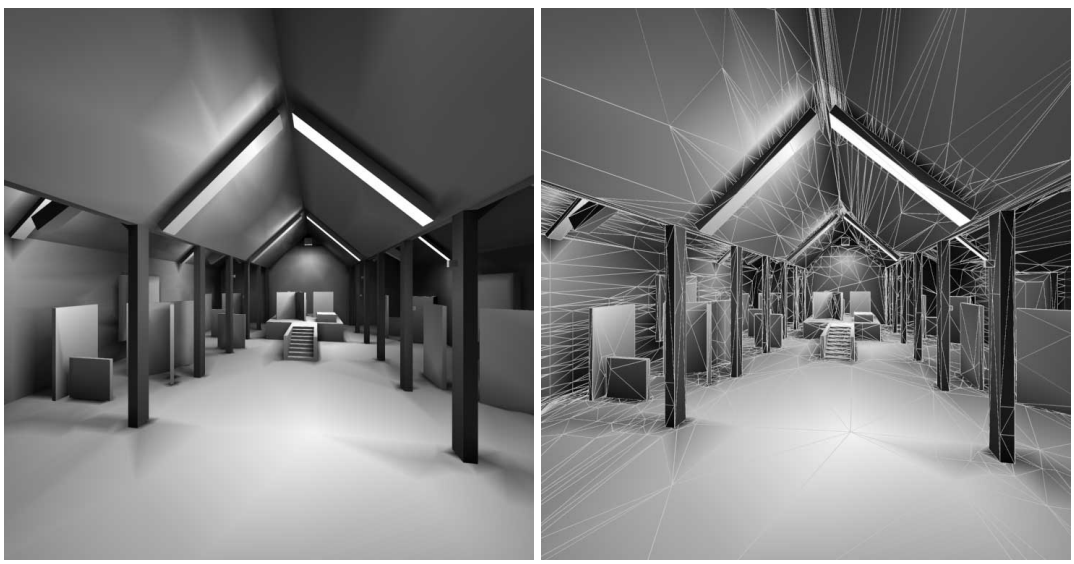


Figure 2.32: Remeshing with $\epsilon = 0.25$, reduction 71%

2.5 Multiresolution Models in Client Server Environments

The current available network bandwidth means, that for the next future a transmission of a complete multiresolution model over the internet will still take some time. Therefore, it is more suitable to transmit the model incrementally and to integrate this process in a WWW standard like VRML. There are two possible ways to create a multiresolution extension in VRML2.0.

First, one can define so-called *PROTO* nodes. They rely completely on VRML2.0, the aspect of sharing the data is not possible and nodes are transferred as an entity. The only thing which could be used to transfer data incrementally is the so-called *SCRIPT* node, which can contain a Java-like program for execution on the client system. But implementing the complete algorithm necessary to update the multiresolution model is not very fast, since script-nodes are realized in Java-Script which is an interpreted language . Furthermore, the whole communication over the web would be handled by the Java scripts.

Next there is the concept of *Living-Worlds*, which is a proposal of *Microsoft* to extend VRML2.0. The structures introduced there would be very suitable to support the multiresolution model with all the needed information. A so-called *SharedObject* node, the *pilot*, can distribute its replications, called *drones*, to several clients. The pilot instance holds one mesh structure for every drone, which is used to determine the needed update information. The drones themselves have an own mesh structure which is updated by insertion and removal of the point-information received from the pilot. The geometry information of the pilot is therefore not duplicated to all the drones and each drone has only the needed amount of information as a copy. A drone can evaluate its actual camera position and send this information to the pilot instance of the node. The pilot instance in turn determines the needed information for updating the multiresolution model of the drone. This is done with the pilot inserting additional points into the mesh structure using the Multiresolution Delaunay Approach. Then these points are transmitted to the drone, which updates its data structures too. This is necessary, since the drone doesn't have all points of the entire model and constructs its mesh of a subset of points. Furthermore, unnecessary points and triangles of the drone mesh and the pilot mesh are eliminated by the drone and the pilot independently, since the drone can evaluate also the camera or the avatar-position to decide which information isn't necessary any more. So the drone mesh can't grow bigger and bigger.

Therefore, the Living-Worlds concept would be a good foundation to implement a distributed multiresolution model for parameterized surfaces. The key idea for this is to save bandwidth, memory and rendering-power on the drone system by reconstructing the topology from the geometry with the use of the Multiresolution Delaunay Approach.

Chapter 3

Visibility Algorithms

In the previous Chapter, geometrical scene adaptation methods were described that allow a reduction of a scene's complexity by reducing the scene geometry.

Another paradigm of scene adaptation is the determination of the visible and invisible parts of a scene. These parts don't have to be rendered, since they will not contribute anything to the rendered result.

Usually, only a small part of a large and extended scene is visible. The rendering time for such a scene can be reduced significantly, if it is possible to figure out invisible parts more quickly than rendering them. It is inevitable to partition the scene by some sort of hierarchy for these fast visibility queries. Therefore, also some methods for scene organization will be described in this chapter.

The determination of the parts of a scene not contributing to the rendering result is called *Culling*. Generally, culling approaches don't change the geometric scene complexity as the reduction approaches do. Of course, both kinds of adaptation approaches can be combined. A simple form of culling is *Back Face Culling*, where back facing polygons are excluded from rendering. The decision if a polygon is backfacing can be made without putting it in the graphics pipeline by using the actual viewer position and the polygon's normal. Another well known culling paradigm is the so-called *View Frustum Culling*, where the parts of the scene contained in the view-frustum are determined and all other parts are then discarded for rendering.

Bothering about culling is important for the visualization of geo-related data-sets. In Chapter one and two, it was already depicted, how huge geographic data sets can be. Especially for reconstructing city models, a great amount of additional complexity has to be spend for getting the houses and buildings of such a city to resemble the original objects. Intelligent and fast culling algorithms enable the user to spend additional complexity while maintaining a certain frametime in order to have an interactive feeling.

3.1 A New Approach for Occlusion Culling

View frustum culling enables the user to cut away all parts of the scene not being contained in the view-frustum of the virtual camera. Nevertheless, a great amount of data in the view-frustum can be invisible due to occlusion by other structures in front of it. This occlusion usually is controlled with a *Hidden Surface* algorithm, for example a

z-buffer, see [31]. Therefore, rendering time can again be saved by fast determination of occluded parts of the scene. This special kind of culling is called *Occlusion Culling*.

3.1.1 OpenGL-assisted Occlusion Culling

One of the major graphics APIs (application programming interface) for implementing computer graphics software is currently *OpenGL*, which was introduced in the mid-nineties by *Silicon Graphics*. Today, it is an independent standard and its improvements and versions are controlled by an architectural review board of major companies and developers.

In the following Sections, an OpenGL-assisted occlusion culling algorithm will be presented to improve the rendering performance of large polygonal models. The algorithm itself is also usable on a graphic system which does not have OpenGL capabilities. Since all major suppliers of graphics hardware and operating systems are supporting OpenGL nowadays, the algorithm is described in an OpenGL environment.

Using a combination of OpenGL-assisted view-frustum culling, hierarchical model-space partitioning, and OpenGL based occlusion culling, a significantly better performance on general polygonal models can be achieved compared to previous approaches. In contrast to these approaches, only common OpenGL features are exploited and therefore the algorithm is also well suited for low-end OpenGL graphics hardware. To show the applicability of the algorithm on low-end graphics workstations, all measurements were performed on a SGI O_2/R10000 workstation.

Furthermore, an extension to the OpenGL rendering pipeline is proposed to add features for improved and fast general occlusion culling.

3.1.2 Introduction

Hidden-line-removal and visibility determination are among the classic topics in computer graphics [31]. A large variety of algorithms are known to solve these visibility problems, including the z-buffer approach [100, 8], the painter algorithm [31], and many more.

Recently, visibility and occlusion culling had been of special interest for walkthroughs of architectural scenes [1, 105, 76] and rendering of large polygonal models [50, 36].

Unfortunately, these approaches are limited to cave-like scenes [50], require not commonly available hardware support [43], or do not provide interactive rendering (more than 10 frames/second) of large models on mid-range graphics hardware [114].

The following algorithm, described in [57], is suitable for general occlusion queries. In a pre-process, the model is subdivided into a sloppy n-ary space-partitioning-tree (snSP-tree). In contrast to ordinary partitioning-trees, like the BSP-tree [34], the subdivision is not a precise one; snSP-tree sibling nodes may not be disjunct. This is to prevent large numbers of small fractured polygons, which can cause numerical problems and an increase of the rendering load. Since all non-sloppy tree structures can be also stored in a snSP-tree, the algorithm uses a very flexible data structure.

During the actual occlusion culling, the OpenGL selection buffer is used to implement a view-frustum culling of the nodes of the subdivision tree. Thereafter, the

bounding volumes of the remaining nodes of the snSP-tree are rendered into an implementation of a *virtual occlusion buffer* to determine the non-occluded nodes. Finally, the polygons of the snSP-nodes [114] considered not occluded are rendered into the framebuffer.

Overall, the algorithm's features are:

- **Portability:** Only basic OpenGL-functionality is used for the implementation of the algorithm. No additional hardware support, such as for texture-mapping [114], or special occlusion queries are necessary. Even low-end OpenGL supporting PC graphics hardware is able to use the proposed occlusion culling scheme.

- **Adaptability:** Due to the use of the OpenGL rendering pipeline, the presented algorithm adapts easily to any OpenGL graphics card. Features that are not supported in hardware can be disabled, or they are realized in software by the OpenGL implementation.

- **Generality:** No assumptions of the scene topology or restrictions on the scene polygons are made.

- **Significant Culling:** Although high culling performance is always a trade-off between culling efficiency and speed efficiency, the algorithm obtains high culling performance, while keeping good rendering performance.

- **Well-balanced Culling:** Different computer systems introduce different rendering and CPU performance. The presented algorithm provides an adaptive balancing scheme for culling and rendering load.

- **Non-conservatism:** Due to some optimizations, the algorithm provides a non-conservative approach to occlusion culling. In most cases, this results in no visual impact.

3.1.3 Related Work

There are several papers which provide a survey of visibility and occlusion culling algorithms. In [114], Zhang provides a brief recent overview with some comparisons. Brechner surveys methods for interactive walkthroughs [7]. Occlusion algorithms for flight simulations are surveyed in [81].

Early approaches are based on culling hierarchical subdivision blocks of scenes to the view-frustum [36]. Although this is a simple but effective scheme for close-ups, this approach is less suited for scenes that are densely occluded, but lie completely within the view-frustum.

In architectural model databases, the scene is usually subdivided into cells, where each cell is associated with a room of the building. For each potential view point of the cells, the potential visible set (PVS) is computed to determine the visibility. Several approaches have been proposed in [1, 105, 76]. However, it appears that the cell subdivision scheme is not suitable for general polygonal scenes without room-like subdivision. Therefore, these approaches are of no apparent importance for general occlusion culling problems.

Several algorithms have been proposed in computational geometry. A brief overview can be found in [38]. Coorg and Teller proposed two object space culling algorithms. In [16], a conservative and simplified version of an

aspect graph is presented. By establishing visibility changes in the neighborhood of single occluders using hierarchical data structures, the number of events in the aspect graph is significantly reduced.

Secondly, by combining a shadow-frustum-like occlusion test of hierarchical subdivision blocks (i.e., octree blocks), the number of occlusion queries is reduced [17]. However, both algorithms are neither suited for dense occluded scenes with rather small occluders (resulting in a large increase of queries), nor for dynamic scenes.

Cohen-Or et al. proposed a method called ϵ-Visibility-Culling for distributed client/server walkthroughs [14, 15, 10]. Computing the shadow-frusta for a series of local view points and an occluder permits visibility queries on the local client. However, the algorithm does not seem to scale for very highly occluded scenes.

In [53], an occluder database - a subset of the scene database - is selected. During the occlusion culling, the shadow-frusta of the occluders are computed and a scene hierarchy is culled against these shadow-frusta. Overall, the surveyed computational geometry-based visibility approaches only deal with convex occluders, which limits their practical use severely.

In 1993, Greene et al. proposed the hierarchical z-buffer algorithm [43, 42, 40], where a simplified version for anti-aliasing is used in [42]. After subdividing the scene into an octree, each of the octants is culled against the view-frustum as proposed in [36]. Thereafter, the silhouettes of the remaining octants are scan-converted into the framebuffer to check if these blocks are occluded. If they are not occluded, their content is assumed to be not occluded too; if they are occluded, nothing of their content can be visible. The occlusion query itself is performed by checking a z-value-image-pyramid for changes. Unfortunately, this query is not supported by common graphics hardware and therefore becomes an expensive operation. However, one can consider this algorithm as the inspiring origin of the approach presented in Section 3.1.6.

In [41], Greene presented a hierarchical polygon tiling approach using coverage masks. This algorithm improves the occlusion query of a hierarchical z-buffer, due to the two-dimensional character of the tiling. However, the main contribution of this algorithm is an anti-aliasing method, as the algorithm has advantages for very high-resolution images. The strict front-to-back order traversal of the polygons - necessary for the coverage masks - needs some data structure overhead. Building a hierarchy of an octree of BSP-trees limits the application of this algorithm to static scenes.

Naylor presented an algorithm, based on a 3D BSP-tree for the representation of the scene, a 2D BSP-tree as image representation, and an algorithm to project the 3D BSP-tree subdivided scene into the 2D BSP-tree image [84].

Hong et al. proposed a fusion between the hierarchical z-buffer algorithm [43] and the PVS-algorithm in [76] for special applications. In this z-buffer-assisted occlusion culling algorithm, a human colon is first subdivided into a tube of cells in a pre-process. Thereafter, the occlusion is determined on-the-fly by checking the connecting portals between these colon cells, exploiting the z-buffer and temporal coherence to obtain high culling performance [50]. Unfortunately, this approach is closely connected to the special tube-like topology of the colon and therefore, is not suited for general occlusion culling problems.

In [113], a voxel-based occlusion culling algorithm is presented. After classifying the scene on a grid of samples of the data-set as void-cells, solid-cells and data-cells, the occlusion is determined in a pre-process for each potential view point. Presumably, this algorithm achieves good results for cave-like scenes, but has a high memory and processing overhead for sparse scenes like the forest scene of Section 3.1.9. Therefore, this algorithm is not suited for a general occlusion culling algorithm.

In [114], occlusion culling using hierarchical occlusion maps was presented Similar to [53], an occluder database is selected from the scene database. Using these occluders, bounding boxes of the potential occludees of the scene database are tested for overlaps, using the image hierarchy of the projected occluders. Strategies for dynamic scenes are presented in [101] and [114]. Sudarsky and Gotsman propose in [101] a fast update of a hierarchical data structure. Zhang [114] et al. suggest using each object of a scene as an occluder in the hierarchical occlusion maps algorithm.

3.1.4 Scene Organization

In general, subdivision schemes for general polygonal models are difficult to derive. This results in individual solutions for different data-sets. Hong et al. [50] use a technique which subdivides a voxel-based colon data-set along its skeleton. The size of the different subdivision entities depends on the number of voxels belonging to this entity. In [99], Snyder and Lengyel proposed that the designer of the scene needs to provide the subdivision. Similarly, Zhang et al. used a pre-defined scene database [114]. The most general approach is to subdivide a polygonal model into regular spatial subdivision schemes, such as BSP-trees [34, 84, 41] or Octrees [40, 43]. While these subdivision schemes produce good results on polygonal models extracted by the Marching Cubes algorithm [75] from uniform grid volume data-sets, which provide a "natural" subdivision on Marching Cubes cell base, these schemes run into numerous problems on general models. If a polygon of the model lies on a subdivision boundary, it must be split into several parts, in order to produce a disjunct representation of the bounding volumes. Unfortunately, this procedure increases the number of small and narrow polygons tremendously.

3.1.5 Sloppy N-ary Space Partitioning Trees

In this approach, the use of a sloppy n-ary Space Partitioning tree (snSP-tree) is proposed. While the leaf nodes of the tree contain the actual geometry of the model, the upper nodes only represent the bounding volume of their child nodes in the subtree. The sloppiness is given by the sloppy partitioning of the model, where the bounding volumes of tree nodes of the same tree level may not be disjunct. Therefore, any given model can be stored in such a tree. No re-triangulation is necessary, due to a missing strict subdivision border for the polygons. Nevertheless, polygons which expand over the entire model, such as floors, should be subdivided into smaller polygons to ensure a well balanced tree.

Since the tree does not rely on a particular number of child nodes, like i.e., an Octree, the tree will be called *sloppy n-ary Space Partitioning tree* or *snSP-tree*. Unfortunately, using a snSP-tree as subdivision data structure does not solve the actual

subdivision problem. However, it removes some of the limitations of other subdivision schemes.

Generating Subdivision Hierarchies

Generally, a polygonal scene can be subdivided into smaller parts, where this subdivision can be either hierarchical or non-hierarchical. Each part of this subdivision is a *subdivision entity*. If information at different multiresolution levels are required, usually a hierarchical organization is chosen, where different subdivision entities are combined into one parent entity which contains the whole information of the associated subdivision entities, or only information with less detail (a lower level-of-detail). This subdivision can be represented as a tree which is called *subdivision tree* or *subdivision graph*.

Generating hierarchical subdivisions of very large models can be done in numerous ways. In this Section, two research algorithms will be presented and compared with a commercial tool. All three approaches generate a subdivision hierarchy starting from an arbitrary model. The first, *D-BVS*, was developed by the author and Jens Einighammer, *p-HBVO* was developed by Gordon Müller at the University of Braunschweig, and *SGI Optimizer* (see [59]) is part of SGI's OpenGL Optimizer toolkit. All approaches subdivide general polygonal models.

While good results can be achieved using scenes where additional information is available (i.e., medical scanner data (octree or BSP subdivision), or pre-subdivided scenes), the subdivision performance of the available algorithms for general models remains improveable. Consequently, an automatic subdivision scheme of general scenes is a field of the current research.

Dimension-oriented Bounding Volume Subdivision (D-BVS) The goal of the volume oriented D-BVS subdivision algorithm is to generate evenly-sized, cube-shaped bounding boxes, hence minimizing the area of the screen projection of these bounding boxes. This goal is approached by splitting the bounding boxes multiple times in the largest dimension. The size of the bounding boxes and the associated sub-models is controlled by user-specified parameters, such as the minimal number of polygons, see [27].

Starting with the root subdivision entity — which contains the whole model — the associated bounding volume is split n_{split} times along its largest dimension such that each fraction is approximately of the same size as the second largest dimension. Therefore, the trees generated are sloppy snSP-trees. If cube like volumes have to be divided, this is done one time in the dimension producing the lowest number of new polygons.

$$n_{split} = \frac{\text{largest bounding volume dimension}}{\text{second largest bounding volume dimension}} \tag{3.1}$$

This process continues recursively until the termination criteria are met. These criteria give lower bounds for the number of polygons of the subdivision entities or the size of the dimensions of the associated bounding boxes. This is necessary to avoid undersized subdivision entities which increase the occlusion culling overhead and do not improve the cull rate any more. Occasionally, the split-operation of the subdivision process splits a polygon which lies across the subdivision boundary into two new

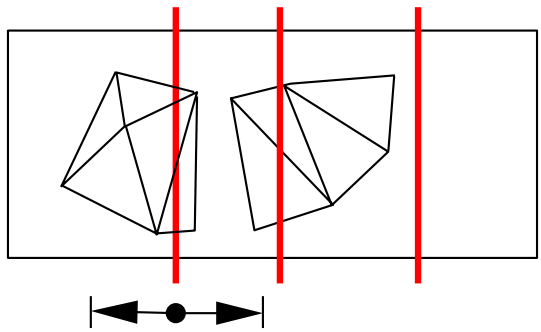


Figure 3.1: Moving subdivision planes to reduce polygon splits. The other planes are already calculated.

polygons (this is usually not the case for uniform grid data-sets). This can tremendously increase the number of polygons and frequently, these polygons are small and narrow and are therefore prone to numerical problems. In order to compensate this, two techniques are applied. First, the subdivision plane is moved along the subdivision dimension so as to reduce the number of additional polygons (Fig. 3.1). The direction and value of the movement is controlled by user-specified parameters. Nevertheless, very large polygons can not be handled by moving the subdivision planes, because they cover several high-level subdivision entities (i.e., a single polygon factory hall floor can be divided in this case into a huge amount of triangles). To avoid the generation of unnecessary additional geometry, the second technique *pulls up* these polygons into a leaf node close to the tree root. The pull-up is controlled by a user parameter that specifies the number of subdivision planes a polygon may intersect. If it intersects more planes, the pull-up is performed. The associated geometry is no longer affected by split operations in lower tree levels.

In general, this algorithm can handle all types of polygonal scenes without producing significantly more polygons. It optimizes shape and size of the subdivision entities, hence their bounding boxes. However, the polygon load is not evenly distributed to the subdivision entities, possibly resulting in a less balanced subdivision tree.

Polygon-based Hierarchical Bounding Volume Optimization (p-HBVO) The polygon-based Hierarchical-Bounding-Volume-Optimization (*p-HBVO*) method recursively subdivides a set of polygons into two subdivision entities. Instead of arbitrarily selecting possible subdivision planes, these planes are induced by the barycenters of the polygons (triangles). By evaluating a cost-function (see equation 3.2), an optimal subdivision plane is established. At each subdivision level, the individual polygons are assigned to exactly one subdivision entity of that level. Consequently, no polygons are split, hence no new polygons are generated by this method.

Starting from the root node, at each subdivision step, the polygons are sorted along all coordinate axes, where the barycenter of each polygon serves as sorting key. Based on these three ordered lists, the potential subdivision planes are evaluated along each axis for each entry in the respective list by splitting the sorted list of polygons into a *left* and *right* part. Please note, that no polygons are split as in the previous approach. In-

stead, the polygons are only reordered. In contrast to pre-defined subdivision planes of the median cut scheme [62], a cost function is evaluated for each possible subdivision plane — defined by the entries in the lists — which approximates the costs of rendering the polygons of one of the two subdivision entities, generated by the respective subdivision plane. By minimizing this cost-function over all possible subdivision planes, an optimal subdivision plane is obtained that generates two new subdivision entities; one contains all *left*−polygons, the other one contains all *right*−polygons. The subdivision process terminates when either the number of polygons or the subdivision depth exceeds one of the two pre-defined parameters: *Max_Triangles_Per_Subdivision_Entity* or *Max_Subdivision_Depth*. These parameters are specified by the user and supplied at the start of the subdivision process. It can be again defined, that polygons extending across the whole subdivided volume can be pulled up in higher level nodes.
In most cases, the cost function is identical to one which has already been successfully applied in ray tracing environments [82]. Adopting this cost function is possible since the objective is the same; both algorithms traverse the scene graph in a similar way to determine visibility. The costs of a subdivision entity H, with children H_{left} and H_{right}, is given by:

$$C_H(axis) \quad = \quad \frac{S(H_{left})}{S(H)} \cdot |H_{left}| + \frac{S(H_{right})}{S(H)} \cdot |H_{right}| \tag{3.2}$$

where

- $|H|$ is the number of polygons within hierarchy H,
- $S(H)$ the surface area of the bounding box associated to sub-scene H, and
- $axis \in \{X, Y, Z\}$.

In the current implementation, the algorithm generates non-sloppy binary trees, but it can be easily extended to generate also sloppy snSP-trees. Overall, this algorithm generates well balanced subdivision trees with respect to their polygon load. Furthermore, polygons of individual objects are detected and clustered together.

SGI Optimizer (SGI) SGI's OpenGL Optimizer is a C++ toolkit for CAD applications that provides scene graph functionality for handling and visualization of large polygonal scenes. It includes mechanisms for subdivision of databases as well as for tessellation, simplification, and others.

Optimizer (depicted in the following sections as "SGI"), which is part of the toolkit, provides functionality for subdivision of model databases. Since the algorithm which is used in this product is not described in a publication, one can only guess from the results obtained what methods are utilized.

All the examples produced with this algorithm were of a regular, non-sloppy tree structure. The subdivision method realized in SGI seems to be similar to the construction of an octree; each subdivision entity is split into eight equally sized subdivision entities. This process is repeated recursively, until a certain threshold criteria for the iterated subdivisions is reached.
Octree-based spatial subdivision is a simple and efficient subdivision scheme. However, the SGI subdivision mechanism subdivides space not by simply bisecting edges

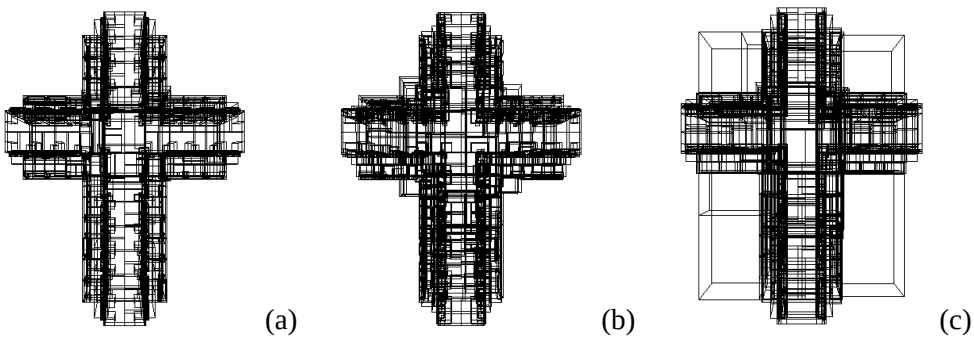


Figure 3.2: Cathedral data set — subdivided by (a) p-HBVO, (b) D-BVS, and (c) SGI; the arts and pillars of the cathedral are well detected by p-HBVO and D-BVS. SGI only used a regular spatial subdivision.

of a cube, as in an octree, but by choosing subdivision planes so that the rendering loads of the resulting parts are similar. As a result, the amount of geometry in the subdivision on each sides of the cutting plane is approximately the same. Polygons which are split due to the subdivision are distributed to the respective subdivision entities.
The main parameters that can be used to control the subdivision are hints for the lowest and highest amount of triangles (tri_{min}, tri_{max}) in each subdivision entity at the leaf-level of the subdivision hierarchy. However, the subdivision algorithm only tries to meet these criteria but is not bound to it. Note, that this tool usually produces triangle strips to achieve better rendering performance.

In general, SGI generates subdivision hierarchies with a well-balanced polygon load. However, the bounding boxes of the subdivision entities are less suited for occlusion culling applications, because the cost function determining the subdivision entities is obviously not optimized with respect to the volume of the bounding boxes. It can be observed that the right-most branch of the subdivision tree frequently contained large subsets (bounding box volume size) of the model, even in the lower tree levels. This is a clear disadvantage compared to the two algorithms described above, since this leads to bounding boxes in high levels of the hierarchy that can contain a great amount of empty space since they don't approximate the geometry tightly. For the occlusion culling approach described in this chapter, such bounding volume hierarchies are not very suitable.

The interior of a gothic cathedral, designed with a CAD system, was used as a test data-set. Occlusion is limited to small parts of the model, because a large share of the polygons is visible from most view points within the model. Figure 3.2 shows a very fine granular subdivision of the cathedral model. Especially the p-HBVO approach (a) adapts very nicely to the structures of the model, such as pillars and arcs. In contrast, the subdivision generated by SGI (c) introduces very large bounding boxes, which do not adapt properly to the actual geometry.

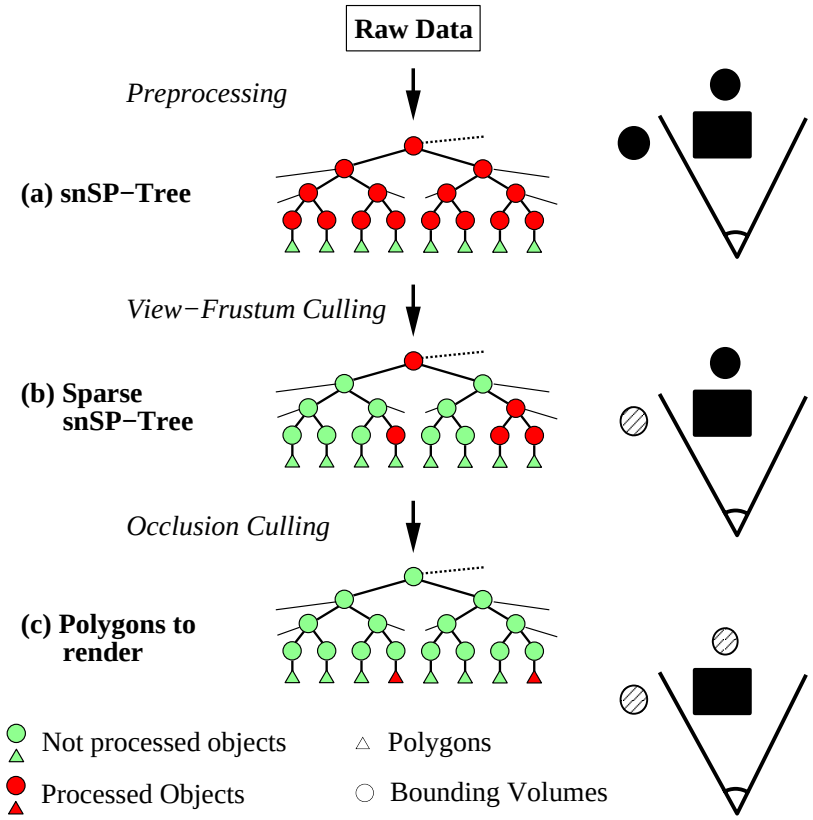


Figure 3.3: Survey of the basic algorithm (the schematic on the right shows the effects of the different culling steps as they are applied to the scene).

3.1.6 The Culling Algorithm

In this Section, a novel solution to the occlusion problem is presented. The algorithm is based on core OpenGL functionality and utilizes the available capabilities of OpenGL to check for occlusion. The basic strategy is to use a hierarchical spatial subdivision of the model and cull all occluded subdivision nodes. As mentioned earlier, a sloppy n-ary Space Partitioning tree (snSP-tree) as a hierarchical representation of a scene is assumed, which is generated once per scene in a pre-processing step.

For each frame to be rendered, view-frustum culling and occlusion culling are performed. Figure 3.3 schematically illustrates the pipeline of the culling algorithm. These individual steps are described in detail in the following Sections.

View-Frustum Culling

In contrast to other approaches, OpenGL is used to perform the view-frustum culling step. In detail, the *OpenGL selection mode* is able to detect whether a bounding volume interferes with the view-frustum. Therefore, the polygonal representation of

the bounding volume (as convex hull) is transformed and clipped. Once a bounding volume intersects the view-frustum, it is tested whether the bounding volume resides entirely within the view-frustum. In this case, all subtrees of the bounding volume can be marked as not occluded. Otherwise, the testing is continued recursively until the child nodes of the bounding volume hierarchy are reached.

In rare cases, the bounding volumes can completely contain the view-frustum - resulting in a invisible bounding volume representation. This can be prevented by testing if the view point lies within the bounding volume, or if the bounding volume lies in between the near plane of the view-frustum and the view point.

As a result of the view-frustum culling step, leaves are tagged *possibly not occluded*, if they are not culled by the view-frustum culling, or *definitely occluded*.

Occlusion Culling

The task of an occlusion culling algorithm is to determine occlusion of objects in a model. A *virtual occlusion buffer* is defined, which is mapped onto the OpenGL framebuffer to detect a possible contribution of any object to the framebuffer. In the implementation of the algorithm on a SGI O_2, the stencil buffer is used for this purpose[1]. Intentionally, the stencil buffer is used for advanced rendering techniques, like multi-pass rendering.

To test occlusion of a node, the triangles of its bounding volume are sent to the OpenGL pipeline. The z-buffer test is performed during the scan-conversion of the triangles, and the output of this test is redirected into the virtual occlusion buffer. Occluded bounding volumes will not contribute to the z-buffer and hence, will not cause any trace or *footprint* in the virtual occlusion buffer.

Although reading the virtual occlusion buffer is fairly fast, it is the most costly single operation of the algorithm. This is mainly due to the time consumed for the setup getting a buffer content out of the OpenGL pipeline. For models subdivided into thousands of bounding volumes, this can lead to a less efficient operation. Furthermore, large bounding boxes require many read operations. Therefore, a progressive occlusion test was implemented which reads spans of pixels from the virtual occlusion buffer using a double interleaving scheme, as illustrated in Figure 3.4.

Although, the setup time for sampling[2] small spans of the virtual occlusion buffer increases the time per sample, spans of ten pixels achieved an almost similar speed-up as sampling entire lines of the virtual occlusion buffer. The compromised setup time for sampling small spans is amortized by the higher probability of finding footprints, due to the addition of the in y direction interleaved scheme.

During motion, iterative sampling enables low culling costs without producing visible artifacts in the scenes used for testing. Once the movement stops, the buffer will be read progressively until all values are tested. Basically, every $sampling$th horizontal line is read from the buffer, where the y-offset is incremented by $\frac{sampling}{2}$

[1] Other buffers could be used as well but the stencil buffer, as an integer buffer, is often the least used buffer and has on many OpenGL implementations, for example on a SGI O_2, an empirically measured better read performance than the other buffers.

[2] Basically, this scheme implements a *sampling* of the virtual occlusion buffer, where $\frac{1}{sampling}$th of each bounding box is read in each iteration. In other words, the algorithm needs *sampling* iterations to fully read the entire bounding box.

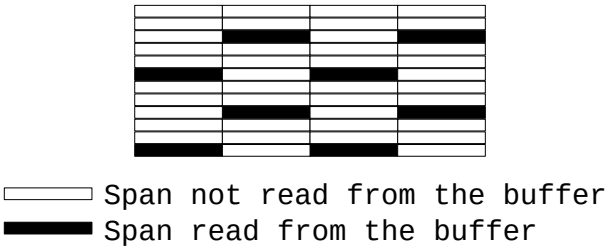


Figure 3.4: Progressive sampling of the virtual occlusion buffer using a sampling value of 6. Hence, after six sampling iterations, the correct occlusion information will be retrieved.

for every second column of spans. For the purpose of illustration, Figure 3.4 uses a sampling factor of six, which is smaller than in the actual implementation.

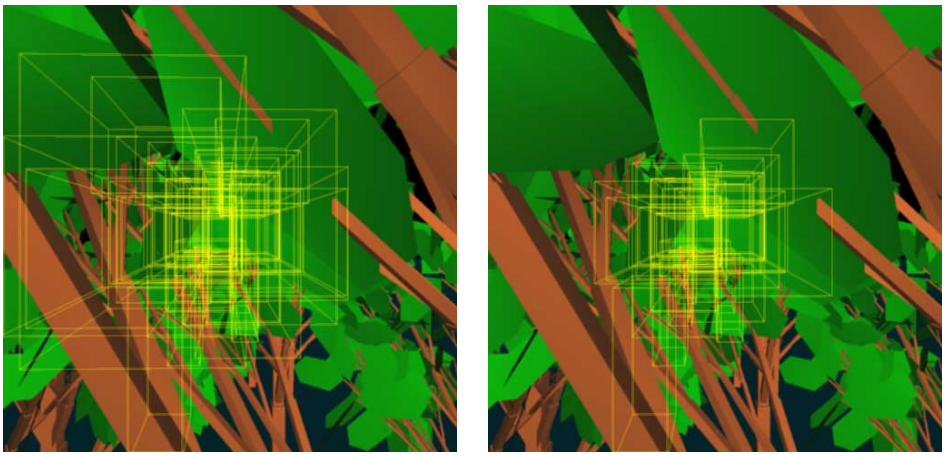
Please note that sampling introduces a non-conservatism into the approach. In some cases, bounding boxes are considered occluded, although they are not fully occluded. However, due to orientation and shape, the actual geometry is usually smaller than their bounding boxes. After performing some measurements, it turned out that a sampling factor of ten is sufficient without compromising image quality for the scenes that were used for testing. Nevertheless, this is a sampling problem and the parameter is dependent on the scene and has to be chosen accordingly. Therefore, the sampling value can be adjusted adaptively in the implementation.

3.1.7 Adaptive Culling

For complex models with deep visibility [3], many almost occluded objects contribute only a few pixels to the final image. Knowing whether an object is not occluded does not introduce a measure of the quantity of contribution. To cull objects which are almost occluded and therefore, are barely noticeable, adaptive culling is an alternative to approximate culling defined in [114]. The approach of [114] utilizes the inherent property of hierarchical occlusion maps, where the combined occluder projections are available at different levels of detail. With this, a threshold value can be defined for whole groups of pixels to declare them as opaque even if some pixels are not definitely covered by an occluder. In contrast to this approach, *adaptive culling* is described in this section. It is much more scalable since it can operate with different criteria based on exact pixel counting whereas approximate culling can only use a fixed threshold.

Each object which generates a footprint on the virtual occlusion buffer needs to be evaluated. Therefore, the number of footprints of the object on the virtual occlusion buffer is counted. The distance of the object from the view plane, the size of its 2D bounding box relative to the view plane, and the number of footprints are used for the visibility consideration. In other words, the percentage of footprints relative to the size of the object is calculated and weighted by the distance of the object.

[3] In scenes with a deep visibility, many objects in the background are visible, due to the sparse scene geometry. An example for such a scene is the forest scene in Figure 3.12.



(a) (b)

Figure 3.5: Alley of trees - bounding volumes of culled objects are marked yellow: (a) Adaptive culling (94% culled). (b) Occlusion culling (88% culled).

$$Adap_{cull}(Obj) = \frac{SizeOf2DBoundingBox(Obj)}{SizeOfViewplane} * \frac{Dist(Eye) + Dist(Obj)}{Dist(Eye)} \quad (3.3)$$

where Sizeof2DBoundingBox(*Obj*) returns the number of pixels of the screen projection of the bounding box, SizeOfViewplane returns the number of pixels of the view plane, Dist(*Eye*) returns the distance between view plane and view point, and Dist(Obj) returns the minimal distance between the *Obj* and the view plane.

For each potentially not occluded object, Equation 3.3 is evaluated. If $Adap_{cull}(Obj)$ is smaller than a user defined threshold, the object is considered as occluded.

Different strategies for dealing with almost occluded objects are possible. First, as mentioned in the previous section, the actual geometry is usually smaller than the associated bounding box. A partially not occluded bounding box does not necessarily mean that the associated geometry is not occluded. Therefore, culling of the object may not have any visual impact. Secondly, if a small fraction of the actual geometry might be not occluded, one will probably not see any detail. Consequently, a lower level of detail representation of this geometry can be used.

In [114] is also mentioned, that a non-conservative culling strategy imposes the problem of aliasing in the form of flickering or blinking dependent on the scene it is applied to. This is also true for adaptive culling. On the other hand, adaptive culling is able to reduce the amount of rendered geometry further. The animation calculated for Figure 3.12 was very suitable for adaptive culling. No aliasing was visible due to adaptive culling but the average culling rate could be increased by 11 percent.

Figure 3.5 shows a result of the adaptive culling mode compared to the standard

occlusion culling mode of the algorithm[4].

3.1.8 Further Optimizations

Several optimizations of the proposed approach are possible. In this Section, a few of them will be discussed.

Interleaved Culling: Clearly, the subdivision tree representing a model can be too deep to efficiently test every bounding volume for occlusion. In the worst case, each leave could contain a single polygon. This is circumvented by generating well balanced trees, holding sufficient polygons in each leaf. Additionally, the view-frustum culling step and occlusion culling step are dynamically interleaved to exploit culling coherence - an already occluded bounding volume of a tree node does not require any further culling test for its child nodes.

Cost-adaptive Culling: To obtain a good ratio between the time spent for rendering and the time spent for culling, one needs to ensure that only a reasonable fraction of the rendering time is spent on culling. Therefore, a factor $P_{graphics}$, which represents the percentage spent on culling, is defined. This factor is hardware dependent and needs to be determined empirically. On the SGI O_2, one can determine $P_{graphics} = \frac{1}{3}$ as a reasonably good factor.

The cull depth adapts dynamically in order to meet the time budget for culling. This budget is calculated using Equation 3.4, where T_{render} is the absolute amount of time spent for rendering the previous frame.

$$T_{culling} = T_{render} * P_{graphics} \tag{3.4}$$

Once the accumulated culling time of the current frame exceeds $T_{culling}$, the remaining nodes are simply culled against the view-frustum and sent to the rendering pipeline. Furthermore, once a node is detected to be entirely within the view-frustum, all leaves of this node can directly be sent to the rendering pipeline without further view-frustum culling the nodes in between.

Depth Ordered Culling: Front-to-back, or depth sorted order of the occlusion tests provides a good heuristic for fast filling of the virtual occlusion buffer. Therefore, it is important to process objects in depth sorted order. The z_{min} and z_{max} values for each bounding volume are returned by the view-frustum test for free. The bounding volumes interfering within the view-frustum are sorted by their z_{min} value into a *DepthHeap*.

The occlusion culling step tags each node as *not occluded* or *possibly occluded*. During motion, those tags have to be generated for every frame. As soon as the camera stops, only bounding volumes in the previous iteration determined as possibly occluded are progressively refined. Nodes earlier marked not occluded will stay not occluded and can therefore be skipped. The leaf nodes which contain the actual geometry are directly sent to the rendering pipeline. This scheme changes once a bounding volume is determined to be not occluded which has previously been marked as possibly occluded. In this case, occlusion culling has to be performed for all following

[4] A threshold of 100 (0.02% of view plane) was used on a view plane of 650 by 650 pixels. Average distance to the objects was 10; their bounding box projection size was on average 423 pixels; the view point was located 0.002 behind the view plane.

nodes in the *DepthHeap*, due to the changed occlusion in the image. As mentioned earlier, this change of occlusion did not happen in the experiments using a sampling factor of 10.

Overall Refined Algorithm: To integrate these additional features, the basic algorithm is modified. The *DepthHeap* is initialized containing at least two uppermost tree nodes which do intersect with the view-frustum. Unless the time budget is not entirely consumed, the head element of the *DepthHeap* is transferred to the cull test. In an interleaved manner, view-frustum culling and occlusion culling for a single frame are performed as described in the following pseudo-code.

```
InitDepthHeap();
while (UsedTime < Budget){
  Node = DepthHeap->getHead();
  if (node == LEAF)
     render(Node->polygons);
  else if (OccTest(Node) == NOT OCCLUDED)
     forall (children(Node))
        if (ViewFrustumTest(child) == NOT OCCLUDED)
           DepthHeap->add(child);
  }
```

One advantage of this interleaved culling scheme is the reduced cost for sorting. For the cathedral scene, which is a well balanced snSP-Tree of twelve tree levels, an average list length of 8 and a maximum of 17 possible not occluded boundary volumes can be measured.

3.1.9 Performance Analysis of the Algorithm

scene	#triangles	#objects	#triangles /object
cathedrals	3,334,104	8	416,763
city	1,056,280	300	3521
forest	452,981	12 + 1	28,500 + 110,981
garbage	5,331,146	2,500	about 2,100

Table 3.1: Model sizes.

The algorithm was examined by processing four different scenes. One architectural scene of a 3D array of gothic cathedrals, a city scene, a forest scene to demonstrate adaptive culling, and - similar to [114] - the content of a virtual garbage can of rather small objects.

The cathedral scene was processed with SGI Optimizer to utilize its triangle-stripping capabilities and then manually reordered to create a better bounding volume hierarchy, whereas the city model and the forest model were artificial scenes constructed and triangle stripped with a CAD tool. In this Section, the performance of the algorithm applied on the test scenes will be discussed which is described in Table 3.1. Note, the achieved percentage of the model culled depends on the granularity of the

snSP-tree. The more the individual objects of a scene are subdivided, the higher is the potential culling performance. Nevertheless, a higher culling performance does not imply a higher rendering performance. While a culling rate of up to 99% of many scenes is possible, the overall rendering performance would drop in most cases due to the culling costs. All measurements were performed rendering with a framebuffer size of 650×650 pixels on a SGI O_2 workstation with 256 MB of memory and one 175 MHz R10000 CPU.

Cathedral Scene

In this scene, eight gothic cathedrals are aligned on a $2 \times 2 \times 2$ grid, where each cathedral consists of 416,763 polygons (Figure 3.6).

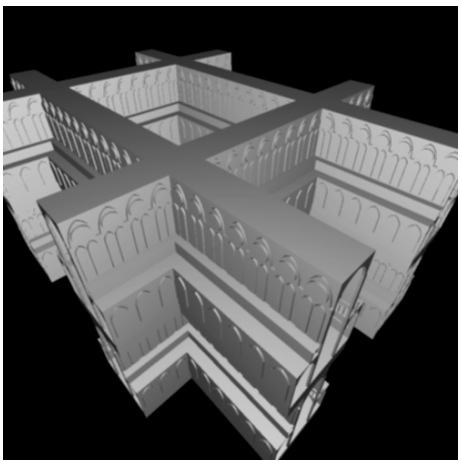


Figure 3.6: Overview of the cathedral scene.

According to Section 3.1.6, two different culling phases are performed: First, a view-frustum culling (VFC); secondly, an occlusion culling. Figure 3.8 shows framerate and percentage of model culled by the algorithm on the cathedral scene for a sequence of about 100 frames. For the performance tests, three different modes were measured: **Direct rendering** (DR) - without any culling, **view-frustum culling** only (VFC), and **view-frustum and occlusion culling** (V+O).

The view-frustum only mode culls only small portions of the eight-cathedral model; for most view points the other cathedrals are still within the view-frustum. However, occlusion culling is far more successful. Up to 65% of the model is culled away. Due to occlusion culling, an average speed-up of seven was obtained (Figure 3.7).

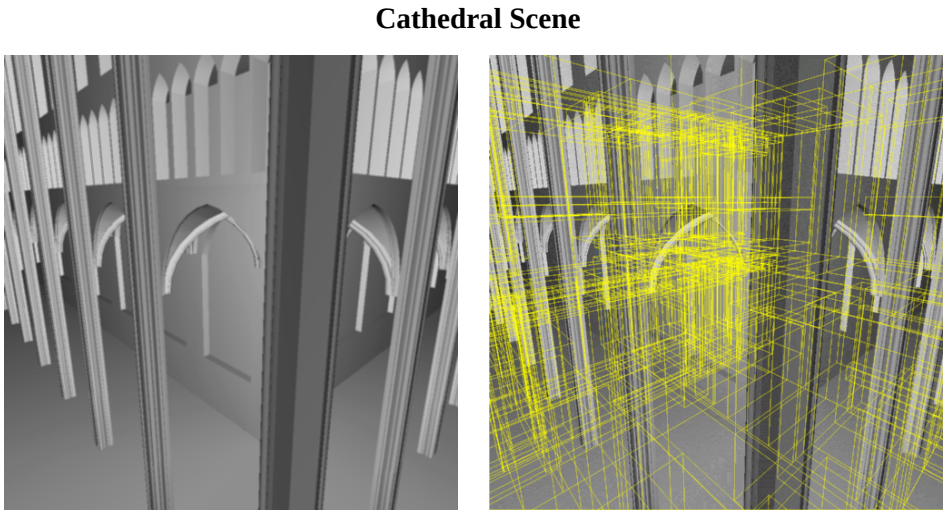


Figure 3.7: (a) Interior view of cathedral. (b) Bounding volumes of culled objects are marked in yellow.

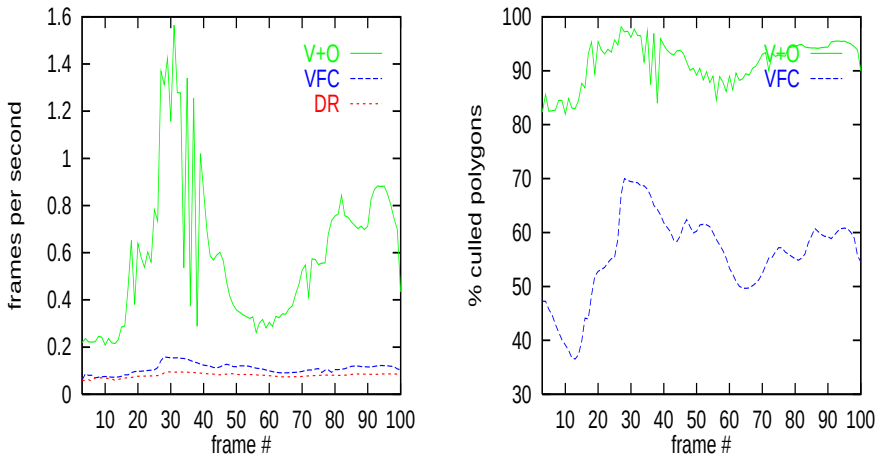


Figure 3.8: Framerate and percentage of model culled: V+O denotes view-frustum culling and occlusion culling, VFC denotes view-frustum culling only, and DR denotes direct rendering without any culling.

City Model

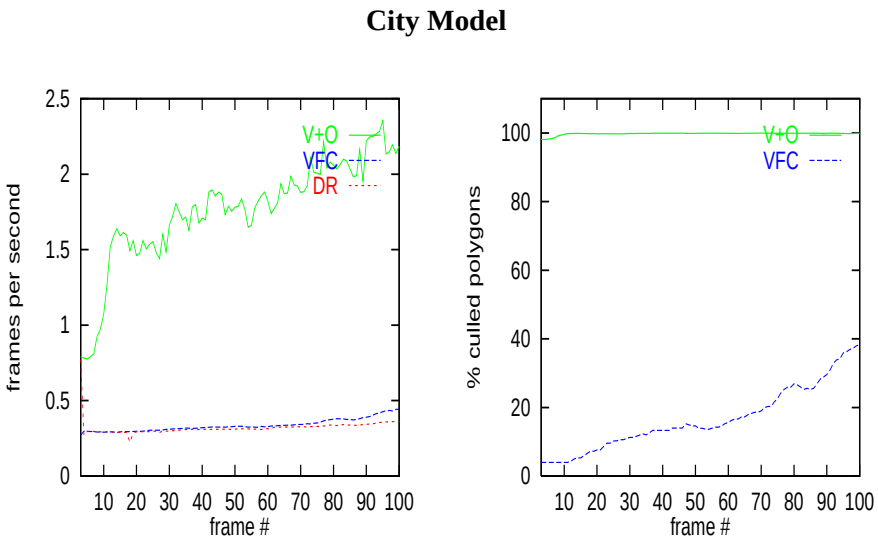
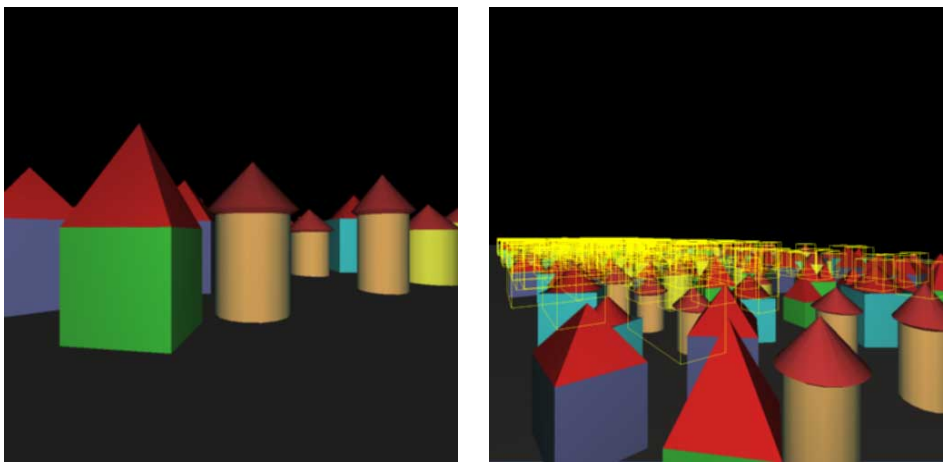


Figure 3.9: Framerate and percentage of model culled: V+O denotes view-frustum culling and occlusion culling, VFC denotes view-frustum culling only, and DR denotes direct rendering without any culling.

The city model is constructed out of three-hundred buildings. Each building contains some interior furniture (Figure 3.10).

Figure 3.9 shows framerate and percentage of model culled of the city model. Three culling modes were measured while rendering a sequence of 100 frames: **Direct rendering** (DR) - no culling, **view-frustum culling only** (VFC), and **view-frustum and occlusion culling** (V+O). While the view point is moving near the ground of the scene, 99.8% of the geometry is culled using the proposed culling scheme. Only 3.9% up to 39.9% of the geometry is culled due to view-frustum culling, where the remaining geometry is culled due to the occlusion culling algorithm. On average, a framerate of almost two frames per second was achieved, which represents a speed-up of about eight against view-frustum culling only.

City Model



(a) (b)

Figure 3.10: City model is rendered using V+O culling: (a) Visitor's perspective. (b) Bird's perspective of visitor's view - all yellow bounding volumes are not rendered due to occlusion culling.

Forest Scene

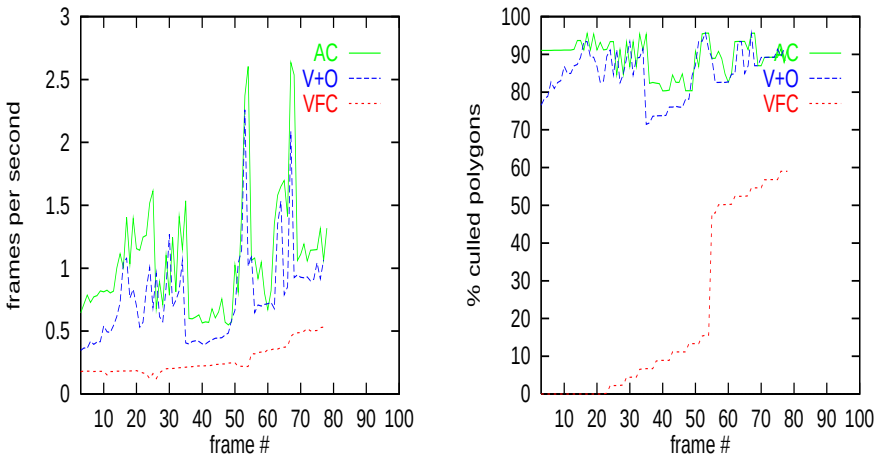


Figure 3.11: Framerate and percentage of model culled: V+O denotes view-frustum culling and occlusion culling, VFC denotes view-frustum culling only, and AC denotes adaptive culling.

The forest scene consists of 12 “tree with leaves” objects - each consists of 28,500 polygons - and one model of ”Castle del Monte” of 110,981 polygons behind the trees (Figure 3.12). The scattered, yet dense occluded structure of the leaf trees has special demands for an occlusion culling algorithm. Depending on the subdivision of those trees, higher additional culling was achieved due to adaptive culling; Figure 3.11 shows an average additional reduction of 11% of the geometry using adaptive culling (AC), compared to the usual V+O culling of the algorithm.

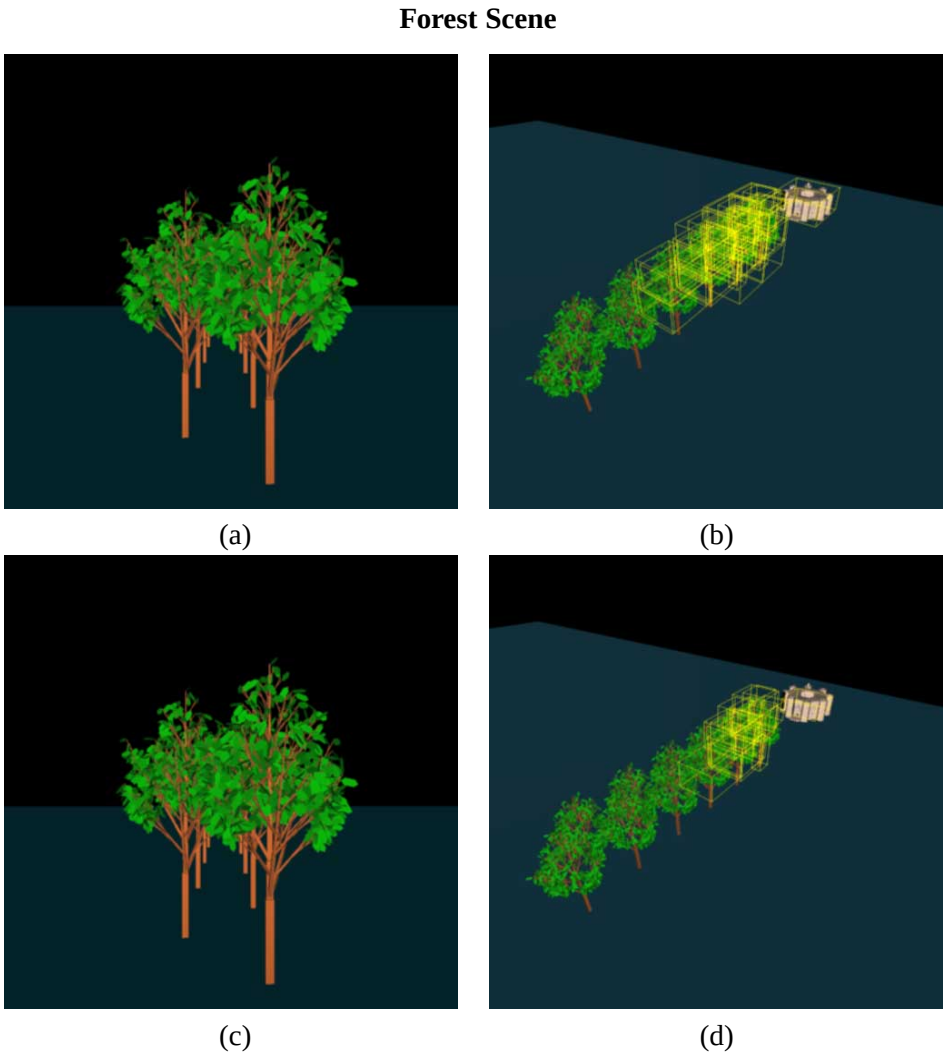
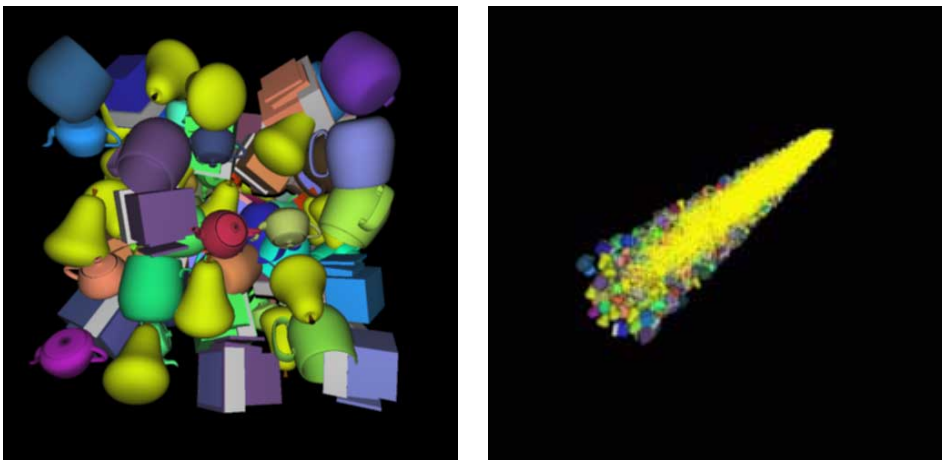


Figure 3.12: The forest scene is rendered using adaptive culling which culled 88% of the structure: (a) Front view. (b) Overview - all culled bounding volumes are marked yellow. (c) The forest scene is rendered using V+O culling which culled 77% of the structure. (d) Overview - all culled bounding volumes are marked yellow.

Virtual Garbage Can Scene

To cull dynamic scenes, a special mode can be used. Only the leafs of the snSP-tree are used to check the occlusion. The performance of this mode is shown with a scene of the content of a virtual garbage can (Figure 3.13). 2,500 independent, potentially moving objects of an average size of about 2,100 polygons are contained in the scene. 96% of the total 5,331,146 polygons are culled. The average obtained speed-up is still larger than seven.

Virtual Garbage Can Scene



(a) (b)

Figure 3.13: (a) Front view. (b) Bird's perspective of front view - all yellow bounding volumes are not rendered due to occlusion culling. Direct rendering took more than 28 seconds, while rendering using the proposed culling algorithm took less than four seconds.

Different Subdivision Strategies

In the tables 3.2 and 3.3, the impact of the different scene subdivision strategies described in Section 3.1.5 on the performance can be seen. These measurements were performed on a HP B180/fx4 graphics workstation and a special HP occlusion culling hardware was used for the occlusion culling test (see [49, 98]) to measure only the influence of the different subdivision algorithms on the culling results. This time, the hierarchies of SGI's Optimizer were not tuned but used as they were produced by the tool. It can be clearly seen, that a scene subdivision in a more optimal sense can improve the performance of the occlusion culling algorithm by up to four frames per second. Furthermore, the polygons in the subdivision nodes were not stripped as in the performance measurements above. This was necessary, since otherwise SGI's Optimizer would have had an unfair advantage over the other two algorithms since they have currently no stripping capabilities.

Approach:	p-HBVO	D-BVS	SGI
#Subdiv. nodes	9	59	51
#Leaf nodes	10	67	52
Time for vfc [s]	0.0021	0.0077	0.0089
Time for occ [s]	0.0041	0.0322	0.027
Time for ren [s]	0.1361	0.1239	0.1426
Render rate [%]	30.0	25.9	30.1
Frame rate [fps]	12.4	8.8	7.8
Render rate of vfc only [%]	33.1	25.9	32.2
Frame rate of vfc only[fps]	12.4	10.8	9.5

Table 3.2: Cathedral data-set: subdivision granularity, average render rates, frame rates, and time consumed by occlusion culling based rendering.

Comparison with other approaches

After all, these speed-ups are equivalent or much better than the ones presented in [114], see Table 3.4. The problem of having no well defined set of test data sets is the same as it was already described in Chapter two. The measurements presented in [114] are not very detailed and also the models are only roughly the same. The cathedral scene used in this chapter is five times bigger than the model used in [114]. Some algorithms, for example the hierarchical z-buffer approach described in [43] or the hierarchical coverage mask approach of [41] use extremely large models that generate very nice speed-ups but the absolute frame times are far away from being interactive. Unfortunately was not tested, how this algorithms behave when used for models with the sizes mentioned in this chapter or used in [114]. Other algorithms are not implemented at all and therefore exist no results.

Approach:	p-HBVO	D-BVS	SGI
#Subdiv nodes	2722	266	420
#Leaf nodes	2723	495	420
Time for vfc [s]	0.0189	0.0111	0.0124
Time for occ [s]	0.0541	0.0318	0.0332
Time for ren [s]	0.0073	0.0831	0.0787
Render rate [%]	0.1	4.1	3.6
Frame rate [fps]	14.0	10.9	11.8
Render rate of vfc only [%]	47.0	50.7	50.4
Frame rate of vfc only [fps]	1.0	1.4	1.4

Table 3.3: City data-set: Subdivision granularity, average render rates, frame rates, and time consumed by occlusion culling based rendering.

scene	#triangles	culling	speed-up	speed-up of [114]
cathedrals	3,334,104	91.3%	4.2	$\sim$2
city	1,056,280	99.8%	4.8	$\sim$3
forest AC	452,981	89.0%	3.8	not
forest V+C	452,981	84.7%	2.6	known
garbage	5,331,146	96.0%	7	$\sim$5

Table 3.4: Average performance of OpenGL-assisted Occlusion Culling algorithm (OOC) compared to view-frustum only culling. The forest scene reflects comparison of adaptive culling (AC) and V+O culling to view-frustum culling.

3.1.10 Limitations

Most occlusion culling algorithms focus on fast determination if rendering of bounding volumes would change the content of the framebuffer, hence the associated geometry would be not occluded [43, 50, 114, 76]. In this approach, a virtual occlusion buffer was introduced which contains the occlusion information. Still, similar to the other approaches, this information has to be read out of the rendering pipeline and searched for changes.

The measurements were performed on a low-end graphics workstation, the SGI O_2. Similar to PC graphics cards, the O_2 performs parts of the rendering pipeline using the CPU. Only operations associated with the rasterization are executed by special purpose graphics chips. This leads to a fast accessibility of the framebuffer. However, this is not true on highly distributed and interleaved graphics subsystems, like the InfiniteReality graphics of SGI. In these cases, the setup-time for reading the framebuffer is significantly larger, thus limiting the performance of the occlusion culling algorithm. Consequently, the scheme works well on low- and mid-end graphics hardware (i.e., HP fx4, SGI O_2, SE, SSE, MXE, and several PC graphic cards).

On highly distributed or highly interleaved graphics systems, the overall speedup

can be less significant. On these systems, the parameters like the progressive sampling factor *sampling* and the cost-adaptive culling $T_{culling}$ have to be chosen appropriately to resemble the system's capabilities[5]. These factors are very different for the different systems, since high performance graphic systems are very proprietary. Furthermore the choice of the *virtual occlusion buffer* can be essential. Eventually, the stencil buffer is no longer appropriate as on low-end systems and instead another buffer, for example the alpha buffer, has to be used.

3.1.11 Using Occlusion Culling with the Multi Resolution Delaunay Approach

The algorithm presented above is ideally suited to be connected with the Multiresolution Delaunay Approach described in Chapter two, see 2.3.

In Section 2.3.3 is depicted, how the data is sorted into a bounding box grid in a preprocessing step before adopting the multiresolution model. These bounding boxes are suitable bounding volumes for performing view frustum and occlusion culling, since they contain the sample points and the reconstructed surface in a tight way. The occlusion information determined for the bounding boxes can then be applied directly to the triangles that form the landscape surface. Triangles, that are fully included within an occluded bounding volume can be excluded from rendering. Triangles that are bigger than a bounding volume have to be processed. This decision can be made in the first stage of the rendering pipeline and needs no interaction with the graphic subsystem. It is therefore very fast and will not introduce synchronization problems.

The bounding boxes need not necessarily being aligned in a regular grid structure. Firstly, they can be organized hierarchically to form a snSP-tree. With this, also triangles included in more than one bounding volume can be marked as occluded.

Furthermore, the bounding volumes can be constructed in a way that they represent the form of the actual landscape. It makes sense to divide a hill from a valley by building special bounding boxes for each of them. With this, hills can obscure parts of the landscape behind of them. It is even possible to process only the front side of a bigger hill by dividing it up into a set of bounding volumes.

When viewing landscape scenes, the performance gain that can be realized by using occlusion culling is very much dependent on the scene. In rather flat scenes, most culling will be done by view frustum culling. On contrast to that, mountains and hills that obscure things behind them can introduce excellent occluders into a scene that make occlusion culling feasible and useful. When using cost-adaptive culling, see 3.1.8, these effects can be brought into account by choosing the coefficients $T_{culling}$ and T_{render} accordingly. In flat scenes, small values of $T_{culling}$ have to be used that can be increased depending on the structure of the landscape. The information for this adaptation is inherently contained in the dimensions of the bounding volumes that comprise the surface points. Using this information, an automatic adaptation scheme is possible that reduces the amount of time spent on culling and increases it again when the viewer turns to parts of the terrain with hills and potentially more occlusion.

[5] An Infinite Reality system is able to read a buffer of 500×500 pixel 10 times as fast as an O_2 (0.00322s vs. 0.0254s). When changing to 100×100 pixels, the O_2 needs only 2.1294e-05s compared to 0.0001762s.

3.2 Embedding Occlusion Queries in the OpenGL pipeline

There are many limiting factors of current OpenGL that hinder real-time occlusion culling for large scenes on the base of this API. Probably most important is the lack of a distinct occlusion culling stage in the rendering pipeline and a well defined set of OpenGL commands that allow the usage of this functionality.

The determination of whether a subdivision node is occluded depends very much on the actual implementation of the virtual occlusion buffer. In the test implementation used for the measurements, the stencil buffer was used for this purpose. This buffer operates with integer values and its read performance is therefore better than the one of color buffers that store float values. Right now, the relevant part of the stencil buffer is checked in a special interleaved mode (see Section 3.1.6) for the identifier of the subdivision node. In many cases, a node is completely occluded. If so, it takes a long time to establish its occlusion state. Therefore, extensions to OpenGL are necessary to implement occlusion culling in basically two ways:

- **Footprint flag.** Most effort is spent checking the buffer for a modification since the last action. Adding a modification flag to the framebuffer would improve the performance tremendously.
- **Footprint counter.** Adaptive culling requires a measurement of how much of an object is not occluded, i.e. a building through a hole in a wall. The number of modified footprints of the virtual occlusion buffer could be such a measure. Extending OpenGL by this feature would simplify this task greatly.

Generally, the strategy for occlusion-driven rendering of a given hierarchically subdivided scene is based on three steps. For each subdivision entity, first the entity (e.g. an octree block) is rendered in a special occlusion mode, which does not affect the content of the framebuffer, similar to the OpenGL selection mode. Second, the occlusion of the individual subdivision entity is established by using the occlusion extension. Finally, depending on the occlusion information, the actual graphic primitives, which are represented by the not occluded subdivision entity are rendered into the framebuffer.

Please note, for the correct computation of occlusion, backface culling must be enabled, since otherwise hits can be counted twice. Furthermore, the necessary counting of pixels of the subdivision entities is only correct, if the bounding volumes used are convex.

In this Section, an extension to the OpenGL API is described. Basically, three features are provided by the extension.

1. **Non-Occlusion Hit Counter (NOHC)**. This is used to quantify all not occluded pixels of the scan-converted subdivision entity. This provides simple analysis of the non-occlusion hits: how many, on which area of the viewport (this is called a occlusion tile).
2. **Projection Hit Counter (PHC)**. This counts the number of pixels of the projection of the object to be rendered. Projection hits together with non-occlusion hits can provide information about the number of pixels that belong to the projection of an object and are not occluded.

Further discussion on the use of the PHC can be found in Section 3.3.2.

3. **Multiple Occlusion Tiles.** The complete viewport can be limited to smaller portions, or refined into a hierarchy of tiles. Alternatively to run a hierarchy of occlusion tests, multiple occlusion tiles can split the area of interest into a multiresolution non-occlusion hit representation, e.g. a quadtree-like representation of occlusion in a given scene (see Fig 3.14).

   As another application of multiple occlusion tiles, visibility of portals in a PVS (potential visible set) approach can be determined [76].

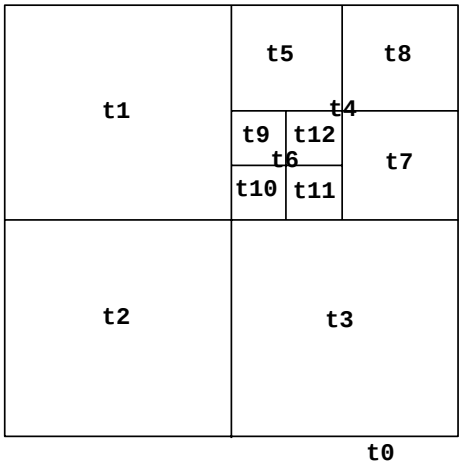


Figure 3.14: Quadtree of occlusion tiles $t_0..t_{12}$ are used.

3.2.1 OpenGL Commands for Occlusion Queries

In order to exploit hardware extensions as proposed in Section 3.3 within OpenGL, the OpenGL API has to be extended. Basically, this extension takes place in three different ways, see also [4]:

Dual-use of already existing OpenGL calls

```
void glScissor( GLint x, Glint y,
                GLint width, Glint height)
```

To specify the occlusion tile, which limits the viewport for the occlusion test, the `glScissor` call is used. Within the viewport, only the tile, starting at x, y with width $width$ and height $height$ is considered for the occlusion test. This command is used to limit a test to the neighborhood of a certain area. By default, the whole viewport is used as occlusion test tile.

Adding new OpenGL calls

```
void glScissors( GLint numTiles,
                 GLint *tiles)
```

In contrast to `glScissor`, `glScissors` specifies multiple tiles as occlusion test tiles. Depending on the occlusion hardware below, the various occlusion test tiles are distributed to different Occlusion Engines (see Section 3.3).

The parameter $numTiles$ and $tiles$ specify the number of tiles and a pointer to an array of $numTiles$ tile specifications. Each of these array entries contains $x, y, width, height$ of one tile.

```
void glOcclusionBuffers( GLsizei *sizes,
                         GLuint **buffers)
```

Similar to the `glSelectBuffer` call of OpenGL, buffers for non-occlusion hits are specified occlusion tile-wise. All non-occlusion hits are stored into the occlusion buffers $buffers$ of the sizes specified in $sizes$. Minimum size of each occlusion buffer is eight, due to the minimal requirements of the `GL_BRIEF_OCCLUSION` mode, which is introduced in the next paragraph.

Adding new parameters to existing OpenGL calls

```
void glGet(...)
```

`GL_MAX_OCCLUSION_TILES` returns the maximal number of occlusion tiles. This information is important in case multiple occlusion tiles are used.

```
GLint glRenderMode(GLenum mode)
```

- `GL_BRIEF_OCCLUSION` is used to specify a fast occlusion mode. In this mode, the number of non-occluded hits and the number of projection hits are returned. Furthermore, to provide information on position and size of the various not occluded pixels, X_{min}, X_{max}, Y_{min}, and Y_{max} of the screen bounding box, and Z_{min}, and Z_{max} as minimal and maximal depth values of the non-occlusion hits are returned.

- `GL_VERBOSE_OCCLUSION`. In addition to the features of the `GL_BRIEF_OCCLUSION` mode, a list of the actual not occluded pixels of the occlusion tiles is returned, up to the maximum size of the occlusion buffer, specified with `glOcclusionBuffer()`.

If `glRenderMode(GL_RENDER)` is called, the respective occlusion information is returned into the buffers specified with `glOcclusionBuffers`. The syntax depends on the previous occlusion mode and enumerates the information tile-wise. If buffer overflows occur, the number of the respective tile buffers is returned. However, the buffers are still set with non-occlusion hits up to its maximum size - which is specified by `glOcclusionBuffers` - and terminates with a -1 entry. Consequently, some occlusion measure up to a user controllable limit is returned, without completely computing the potential costly occlusion information.

Note, similar to the `GL_SELECT` mode, all occlusion render modes do not change the content of the framebuffer,

3.3 Hardware-assisted Occlusion Culling

For interactive rendering of large polygonal objects, fast visibility queries are necessary to quickly decide whether polygonal objects are visible and need to be rendered. None of the numerous published algorithms provide visibility performance for interactive rendering of very large models.

In this Section, the hardware extension proposed in [4] for fast occlusion queries is described. Added after the depth test stage of the OpenGL rendering pipeline, the extension provides fast queries to establish occlusion of polygonal objects. Furthermore, the hardware aspects of this proposal are discussed and possible implementations on two different graphics architectures are presented.

In addition, *Hewlett Packard* developed an Occlusion Flag extension to OpenGL for occlusion queries, see [49, 98]. This flag is already included in their recent fx4 and fx6 graphics accelerators. Similar to the hierarchical z-buffer approach, graphic primitives, which represent a more complex geometry, are rendered within an occlusion test mode to determine their visibility. Depending on the result, all underlying geometry is rendered or skipped. The Hewlett-Packard approach is limited, since only binary visibility decisions can be made. There is no way to quantify the percentage of visibility of a certain bounding volume and the geometry it encloses. The extension explained in this Section provides answers also for these more elaborated visibility queries. Measurements showed that using this flag achieves a significant higher framerate compared to view-frustum only approaches. So far, this is the only available implementation of occlusion culling hardware and it underlines the growing importance of occlusion culling for high-performance rendering of large models.

The implementation of the extension to the OpenGL API proposed in Section 3.2 does require a few modifications within the OpenGL pipeline. To delineate these modifications, first a brief overview of the OpenGL pipeline will be given.

OpenGL rendering pipeline

OpenGL processes graphic data using a pipeline of several distinct stages [111]. In Figure 3.15, an abstract, high-level block diagram of this pipeline is given. Commands enter from the left and proceed through what can be thought of as functional units for the specific operations. Some commands specify the geometry of objects, while others control how the objects are processed during the various processing stages.

OpenGL operates in two modes. In **immediate mode**, all commands are executed directly when they are stated. Alternatively, a **Display List** can be used, where commands are compiled and stored for later execution.

In contrast to objects specified by vertices, parametric curves and surfaces are approximated by the **Evaluator** unit. Polynomial commands are evaluated to generate a vertex based description of the objects.

During the next stage, **Per Vertex Operations** and **Primitive Assembly**, OpenGL processes geometric primitives. These are points, line segments, and polygons, all of

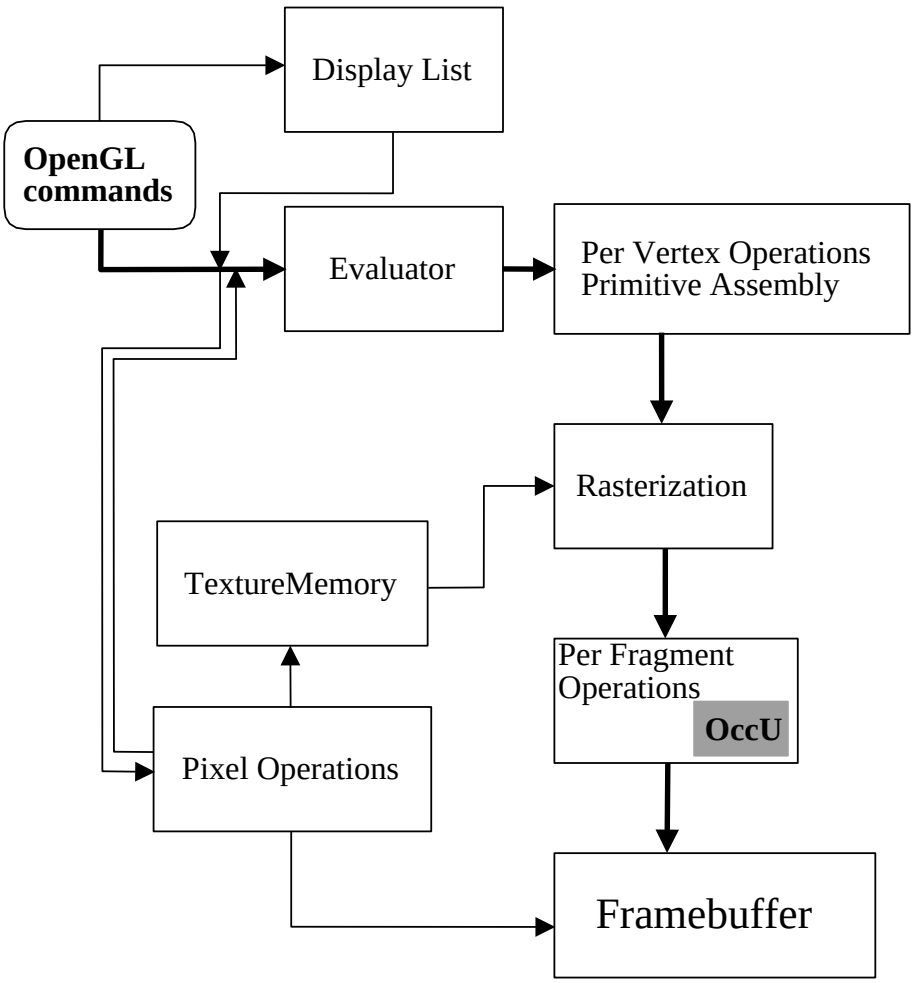


Figure 3.15: Schematic of the OpenGL rendering pipeline. OccU denotes where to fit in the proposed Occlusion Unit.

which are described by vertices. The vertices of the primitives are transformed and illuminated. Furthermore, the primitives are clipped to the viewport in preparation for the next stage.

The **Rasterization** unit produces framebuffer addresses for rasterizing of the primitives. It interpolates associated values using two-dimensional descriptions of points, line segments, or polygons. The resulting fragments are then fed into the last stage, the **Per Fragment Operations**.

This stage performs the final operations on the data before the fragments are stored as pixels in the framebuffer. Since the framebuffer update depends on some conditions, some tests which evaluate arriving and previously stored z-values (for z-buffering) have to be carried out. Also, blending of incoming pixel colors with stored colors, as well as masking and other logical operations on pixel values are done in this stage of the pipeline.

Input can be in the form of pixels rather than vertices to describe two dimensional

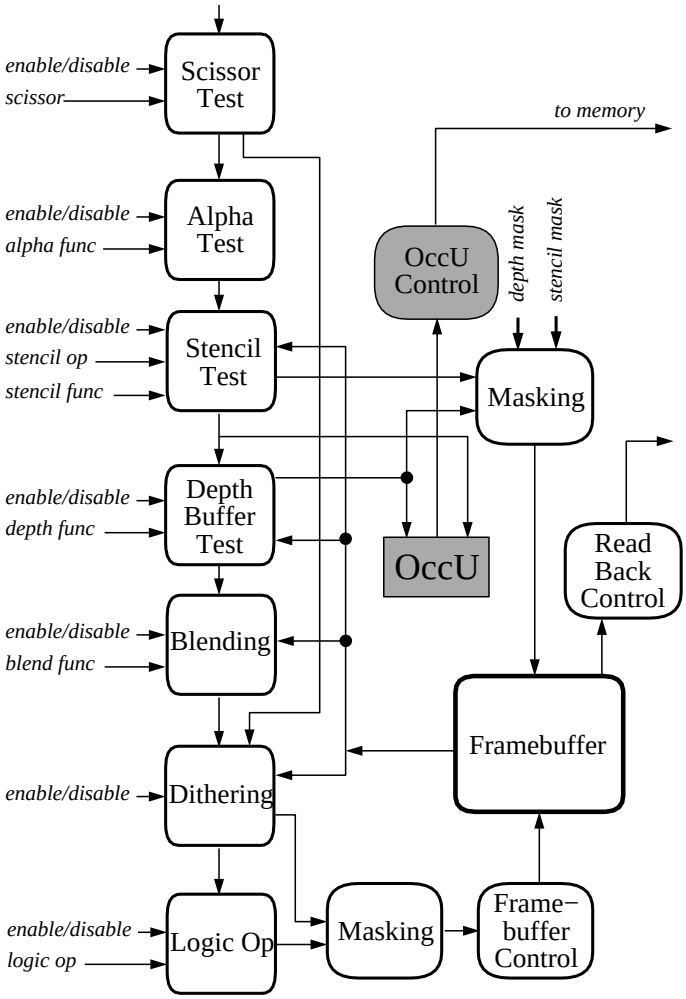


Figure 3.16: Per Fragment Operations and Framebuffer.

image data. This data skips the first stage of processing described above. Instead, it processes data as pixels in the **Pixel Operations** stage. The resulting pixels of this stage are either stored in **Texture Memory**, for use in the **Rasterization** stage, or merged directly into the **Framebuffer** just as if they were generated from geometric data.

The New Occlusion Unit

Many per fragment operations exist in the current OpenGL rendering pipeline. Some of the most important are scissoring, alpha test, stencil test, and depth buffer test, as shown in Figure 3.16.

Testing for occlusion is a ”per fragment” operation since every pixel has to be tested. Therefore, the Occlusion Unit is part of the functional **Per Fragment Operations** block as illustrated in Figure 3.15.

It will be differentiated between the **Occlusion Unit (OccU)**, which is logically

responsible for the overall occlusion, and the **Occlusion Engine**, which is the actual implementation of the Occlusion Unit. In order to accelerate the processing of multiple tiles, the Occlusion Engine can be replicated within the Occlusion Unit. All Occlusion Engines of the Occlusion Unit are synchronized at the Occlusion Control (**OccU Control**).

To connect the proposed Occlusion Unit, it is provided with the x, y screen space address of the fragment, its depth value z, and the write enable signal of the depth buffer test, which is used to write and update the framebuffer with the fragment which is closer than the so far stored fragment. Therefore, the Occlusion Unit is placed behind the **Depth Buffer Test** unit, as it is demonstrated in Figure 3.16.

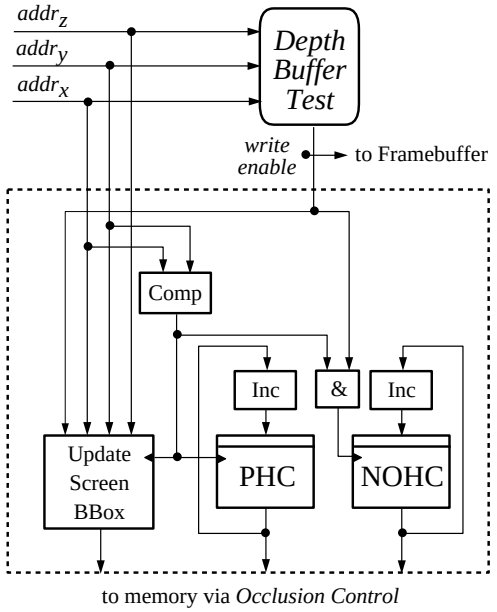


Figure 3.17: Schematical description of one Occlusion Engine.

The Occlusion Unit tests the x, y screen space address of the fragment against the user defined occlusion tile. If the fragment resides within the tile, the projection hit counter (PHC) is incremented. Further, the non-occlusion hit counter (NOHC) is increased, if the depth buffer test was successful, which signifies that the fragment contributes to the framebuffer. To trigger the increment of the non-occlusion hit counter, an AND operation is used. Besides increasing hit counters, it is tested whether the screen bounding box defined by the already found non-occlusion hits is increased due to the newly found hit. So far, the list of hits has yet not been updated. As long as the number of hits is smaller than the provided entries of the list, the x, y coordinates of the fragments are stored in the occlusion buffers which resides in main memory. To send data from the Occlusion Unit to the main memory, the OccU Control is introduced. This unit operates similar to **Selection Control** of the OpenGL selection mode. Its purpose is to synchronize memory access of the Occlusion Unit in case that multiple Occlusion Engines detect non-occlusion hits.

A schematic overview of an Occlusion Engine is given in Figure 3.17. Note, the Occlusion Engine shown in this Figure illustrates the schematic structure necessary to

test for a user defined occlusion tile. Since the user can instantiate multiple tiles, e.g. a tile hierarchy, the Occlusion Engine has to be capable of updating all by the user instantiated tiles. This can be accelerated by assigning the tiles to multiple Occlusion Engines, using a round robin strategy.

3.3.1 Implementing the Occlusion Unit on two different Architectures

In this Section, the integration of the proposal in two existing architectures is investigated. This is done for two well known and described architectures of Silicon Graphics (see [79, 63]).

The SGI O_2 is an example for a medium performance graphics pipeline, which is comparable to many current PC graphics accelerators. It has a single rasterizing unit and a monolithic framebuffer. In Figure 3.18, the details of the **Memory and Rendering Engine** of the SGI O_2 are shown.

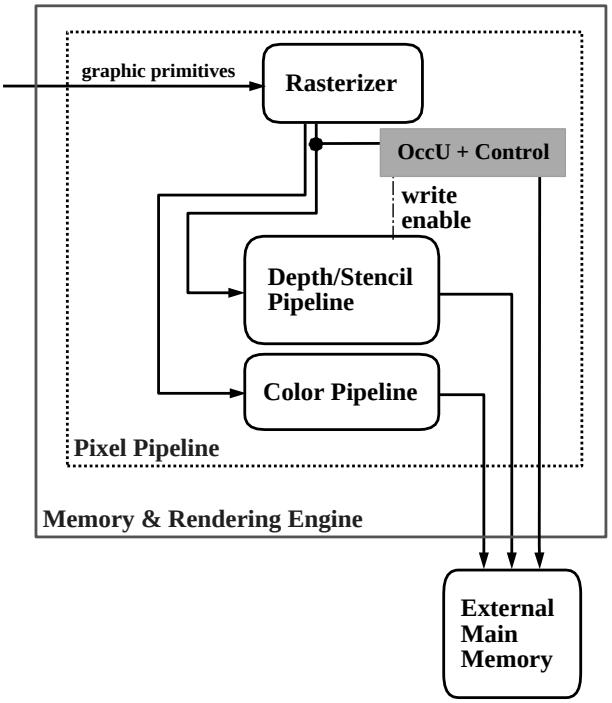


Figure 3.18: Memory and Rendering Engine of a SGI O_2 including the Occlusion Unit.

This unit is connected to previous pipeline stages and to the main memory of the system. Its main part is the **Pixel Pipeline**, which performs all OpenGL rasterization, texturing, and per-fragment operations. Since no dedicated framebuffer is used, this implementation of the OpenGL pipeline uses an extensive pre-fetching algorithm to hide memory latency. The framebuffer itself is located in the main memory of the system.

To integrate the proposed extension, the Occlusion Unit is placed into the Pixel Pipeline, where all the information necessary for the occlusion test is present. Address

information is provided by the **Rasterizer**, while the write enable signal of the depth buffer test is provided by the **Depth/Stencil Pipeline**. Each Occlusion Engine which processes multiple tiles introduces additional cycles for each further tile.

In contrast to the SGI O_2, the InfiniteReality system is used as exponent for a high end graphics system. Its pipeline has a highly parallel architecture, containing multiple rasterizing units and an interleaved and distributed framebuffer [79], as it is illustrated in Figure 3.19.

The pixel operating part of the system is composed of the so called **Raster Memory Boards**. Each board has one rasterizer, called **Fragment Generator**, and an interleaved framebuffer which is accessible via special interfaces, the **Image Engines**.

Since the extension computes occlusion on a pixel basis, the Occlusion Units need to be integrated at the Image Engine level. In all Occlusion Units, Occlusion Engines are configured in the same way as their respective occlusion tile of the viewport. Consequently, each Occlusion Engine handles only hits of the part of the framebuffer to which its Image Engines belong to. In order to optimize occlusion performance, it is desirable to have an Occlusion Engine for each occlusion tile.

The evaluation process for an occlusion test has to respect the distributed nature of the system. Therefore, a two stage synchronization process is proposed. First, the hits of one Raster Memory Board are synchronized locally. Thereafter, the result of the different boards are merged to form one occlusion report. During the latter synchronization process, detected non-occlusion hits which belong to the occlusion tile which is partitioned between different Raster Memory Boards or different Image Engines needs to be merged to form a single occlusion report for this tile. This process can be either implemented in hardware or software.

The integration of the Occlusion Unit has been shown for two different graphic architectures. For a rather simple system as the SGI O_2, the Occlusion Unit can easily be integrated into the Pixel Pipeline. Although the integration into a InfiniteReality system is much more complicated, it is still feasible and not more difficult than the organization of the Image Engines themselves. Nevertheless, some latency will be introduced, due to necessary synchronization.

3.3.2 Further Applications

Support for Collision Detection

In virtual environments, it is a non trivial task to detect collisions of the user with objects. The process is highly demanding, since it requires a collision test with all objects. In [25], raytracing is used to compute an analytical solution for this problem. In *VRML*, one standard API for describing VR scenes, collision can be detected by introducing collision nodes. Each time the user changes its position, a collision test is applied to all collision nodes. Due to the time consuming collision test, the frame rate drops and people tend to switch the collision detection mode off.

The proposed extension for OpenGL can be used for this purpose to a certain degree. For a given position of the user, an image of the scene is generated. A mostly valid assumption is that the user changes its position and direction incrementally. Hence, in case the user heads in viewing direction - e.g. straight, straight-left - a

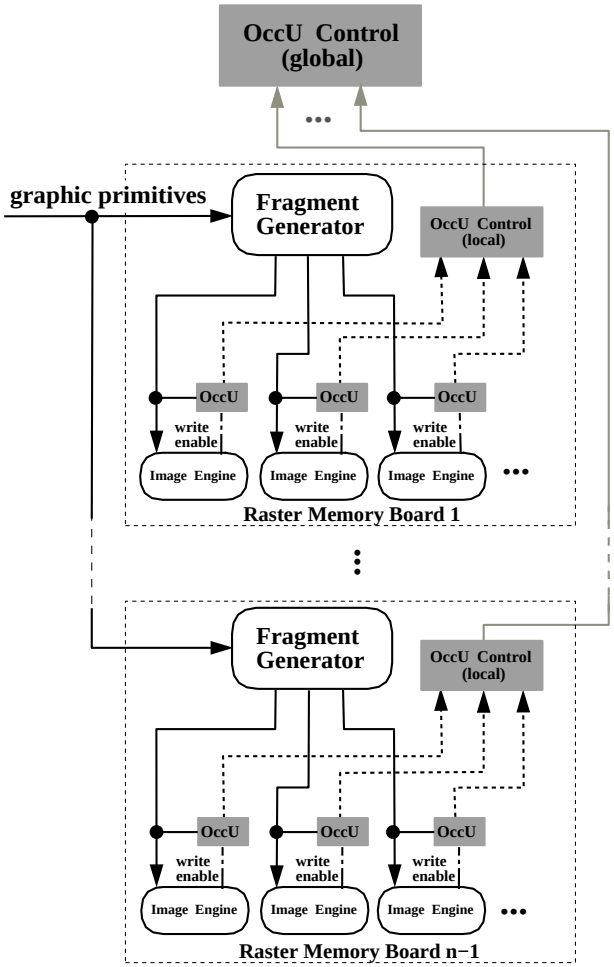


Figure 3.19: Schematic for implementing the Occlusion Unit on an InfiniteReality system.

subdivision of screen space can be determined which represents the possible collision areas within the viewport. This is illustrated in Figure 3.20.

To check whether a certain step will cause a collision, a customized view-frustum covering the check area of the screen is rendered in `GL_BRIEF_OCCLUSION` mode. The far plane of this view-frustum depends on the step size of the user, near plane is identical to the view plane. Information whether a step can be taken without causing a collision is indicated by the non-occlusion hits and the projection hits. If the number of non-occlusion hits is different from the number of projection hits, some pixels of the customized view-frustum are occluded, which means that a collision can be expected. In contrast to the occlusion test, backface culling must be disabled, in order to detect intersections with the backfacing polygons of the view-frustum. To get more detailed information, screen space can be subdivided further.

So far, collision detection can only be indicated and depends on the given viewport. Unfortunately, testing backward stepping does require two-pass rendering since no

straight− up−left	straight−up	straight− up−right
straight−left	straight	straight−right
straight− down−left	straight−down	straight− down−right

Figure 3.20: Subdivision of screen space into areas which correspond to the heading direction of the user.

image of the scene behind the user is available in the framebuffer.

Support for Ray Casting

One of the results of the `GL_BRIEF_OCCLUSION` and `GL_VERBOSE_OCCLUSION` modes is a list of not occluded pixels of the tested subdivision entity. Besides usual statements on occlusion or non-occlusion of that entity, this information can be used to accelerate ray casting in a mixed volume graphics and polygon model. Considering a hierarchical ray casting approach, the subdivision entities are rendered in occlusion mode. All computed non-occlusion hits for this entity mark pixels which are not occluded. Consequently, these pixels are image plane parts to cast rays through, because the content represented by the subdivision entity may be still visible. In other words, all pixels without an non-occlusion hit are not visible, and therefore, rays casted through those pixels of the image plane have no contribution.

Chapter 4

Texture Mapping

Texture mapping is a popular way of enhancing the realism of three dimensional scenes without increasing their polygonal complexity. Therefore, texture mapping is very important for generating high quality renderings of geo-related data-sets.

Furthermore, some kinds of geo-related data are available as two dimensional raster data (for example maps, see Chapter one). This data can be visualized in a three-dimensional scene by using texture mapping.

In the rest of this chapter, some texture mapping related problems and possible solutions will be discussed. First, a camera adaptive data structure for sparse texture data loading will be explained. Next, an approach for the specific selection of parts of an image for texturing will be described. The concluding topic will be a new method for texture filtering.

4.1 Adaptive Texture Data Structures

System memory or dedicated texture memory is restricted by technical or economical constraints. On an O_2 from *Silicon Graphics*, the texture size of a single texture is limited to a resolution of 1024 $\times$ 1024 texels (texture pixels are called *texels*). Since in the field of GIS-applications textures can be maps with 10000 $\times$ 10000 texels, the texturing power of graphic workstations can't be directly used to render these textures on the terrain (see also 1.3).

In the area of texture mapping, filtering techniques like *MIPmapping* have been developed to adapt the texel-size used in object-space to the pixel-resolution of the projected texels in screen-space. With this filtering paradigm, a hierarchy of images is generated from the original texture, the so called *MIPmap pyramid*, see [109]. Many publications describe how to use texture mapping to enhance realism, see [44], [47]. Only a few authors mention the usage of hierarchically organized texture data to save memory and bandwidth (see [86]). Some architectures, for example the *Infinite Reality* graphics system of SGI, are already capable of using larger texture maps than the above mentioned size of 1024 $\times$ 1024 texels. This architecture uses so called *Clip-Maps* to adapt the texture resolution to the actual camera position of the viewer, see[103]. The problem of having not enough memory to store the entire texture in main or texture memory results in a selection paradigm which reduces the amount of memory necessary. With *Clip-Mapping*, this problem is solved by clipping the MIPmap pyramid on

its finest levels. Levels that exceed a texel size of 2048 $\times$ 2048 are clipped to a viewer dependent part of the MIPmap level. This is possible since even in the finest level of a MIPmap, only a part of the map corresponding to screen resolution must be accessible. In [80], it is assumed that the size of the clipped area is 2048 $\times$ 2048 texels used on a 1024 $\times$ 1024 pixel screen with trilinear MIPmap textures. This assumption holds since 1024 $\times$ 1024 pixels are nowadays a normal screen resolution. This approach requires a regular update of the clipped MIPmap levels, if the viewer changes his position. SGI has solved this problem with a clever update procedure which does not allocate new memory but reuses the old memory by overwriting the no longer necessary parts of the level with new information (*toroidal loading*). The texture access is realized with offset registers, which can be set accordingly to allow the placement of the coordinate origin arbitrarily in the MIPmap level.

As a result, the problem of using very large textures on a broad range of modern graphic workstations has not been solved convincingly. The next sections will propose a new a framework to support very large textures.

4.1.1 Texture Tiling and MIPmap Precalculation

In contrast to *Clip-Mapping*, this approach suggests a tiling of the whole texture in a set of MIPmap pyramids, which are precalculated and stored on disk. Not all levels are stored, since smaller levels can be recalculated more quickly. This depends on the hardware of the graphic system and its power to calculate the missing levels at loading time (e.g. 64 $\times$ 64 texels are used on a SGI O_2 as a limit).

Each single pyramid can use at its finest level the maximal resolution *OpenGL* offers for a single texture. The image is subdivided into an array of texture tiles, after scaling it to a size that can be divided into tile sizes of a power of two in both dimensions. The size of a single tile is the biggest tile size possible that increases the image size in the scaling phase in a minimal way. This means that even if a tile size of 1024 $\times$ 1024 texels would be possible, also a tile size of 1024 $\times$ 512 texels may be chosen if with this tile size the overall scaling of the whole image can be reduced. This structure is called *MIPmap pyramid grid (MP-Grid)*, see Figure 4.1.

Each of these MIPmap pyramids offers now the possibility to adapt to the viewer's position by only loading those levels of the map which are necessary. The question of which levels are necessary can be answered with the three dimensional bounding box of the polygons which shall be textured with this pyramid. The projection of this bounding box onto the viewing plane of the viewer's camera depicts the maximum texel number necessary to texture the content of this bounding box correctly. Rounded up to a power of two the maximum of the dimensions can be used to determine the maximum level of this MIPmap pyramid which has to be accessible in memory (see fig. 4.2). This is a conservative estimation of the number of levels needed, since even worst-case polygons like P which are parallel to the viewing plane can be textured correctly. With this, the whole *MP-Grid* can easily be adapted to a changing viewer position.

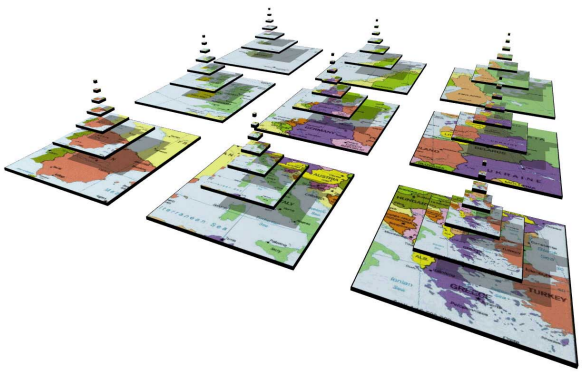


Figure 4.1: The MIPmap pyramid grid (MP-Grid).

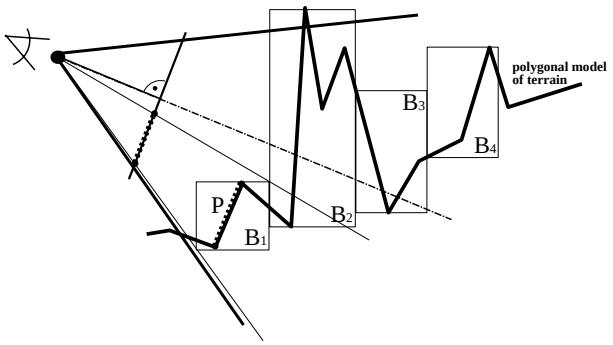


Figure 4.2: The bounding box B_1 depicts with its projection the needed MIPmap level for the texture tile used to texture the polygons of B_1.

4.1.2 Clip-Map Versus MP-Grid

One problem of the *Clip-Map* approach is the adaptation of the *Clip-Map* to the current viewer's position. To do this, one must find a so called *clip center* for placing the clipping window in texture space. Finding a good center can be difficult and expensive. To solve this correctly, one polygon of all polygons to be textured has to be determined that contains the point of minimal distance to the viewing plane on one of its edges. The center is placed at the corresponding point in texture space. If the center is not placed in this way, it can happen that parts of polygons do not find the needed resolution for texturing because they are nearer to the viewing plane than the center of the *Clip-Map*. The selected map levels with their fixed resolution of 2048 $\times$ 2048 texels may then eventually not contain the needed texels. Also in the case of terrain textures, simple approaches like projecting the center of the *Clip-Map* directly beneath the viewer result in wasting half of the finest texture levels, since they lie in the viewers back and are only useful, if the viewer turns without moving. Especially when the viewer looks from a higher altitude with a small pitch this simple approach is not sufficient, since then the center of the *Clip-Map* must be moved in the viewing direction.

In Figure 4.3 and Figure 4.4 is shown, that *Clip-Maps* produce for some special situations very bad results. The left image of Figure 4.3, a tiled earth texture, is used in Figure 4.4 to texture a globe. The north pole can be seen from the chosen viewpoint. Therefore, all the texture tiles that are marked yellow in the right image are invisible. It is not possible to solve this texturing problem correctly with *Clip-Mapping*, since each choice of a center will injure some part of the pole region due to the fixed resolution of the clipped levels. In this case, the finest resolution has to be maintained over the whole pole region which is impossible when this finest level is clipped to a fixed number of texels .

All these considerations can be solved with the approach of *MIPmap pyramid grids* presented above more directly and either conservatively or with a controlled approximation error. This means, that not all of the MIPmap levels have to be loaded if other considerations like time requirements do not allow this. Nevertheless, the algorithm always has the possibility to determine very quickly the needed levels.

Another advantage is the sharing of the proposed *MP-Grid* structure between more than one viewer. With a *Clip-Map*, only the non clipped levels of the texture can be shared between two or more viewers, since the clipped levels of the fixed size of 2048 $\times$ 2048 are viewer dependent. Imagine two planes which meet each other while flying over a terrain. With this approach, the whole *MP-Grid* can be shared between the viewers. The grid is locally refined for each viewer, but this refinement can be reused by the other viewer if he changes his position. In the case of two *Clip-Maps*, information which has been present in the *Clip-Map* of the first plane is reloaded in the *Clip-Map* of the second while it has already been in texture memory.

Furthermore, if occlusion culling for a set of polygons and their bounding box is available (see [4]), the *MP-Grid* can easily exploit this. If the bounding box is occluded, the adaptation of the corresponding *MP-Grid* pyramid is not necessary and the occupied texture memory can be freed if it is needed for other pyramids. This integration of occlusion culling assisted texture management is not possible with the *Clip-Map* approach.

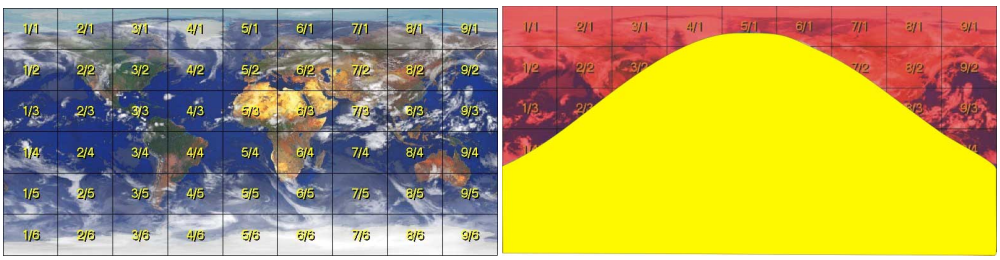


Figure 4.3: Tiled earth texture.

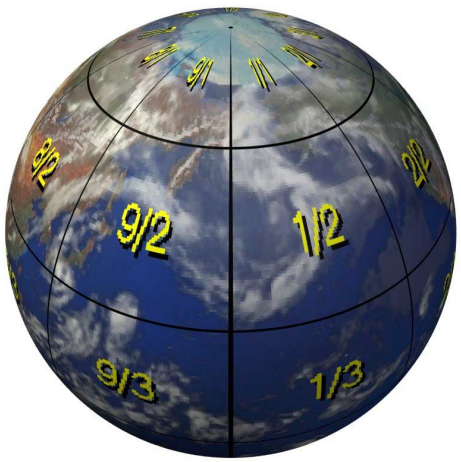


Figure 4.4: The left image of 4.3 is used to texture a globe.

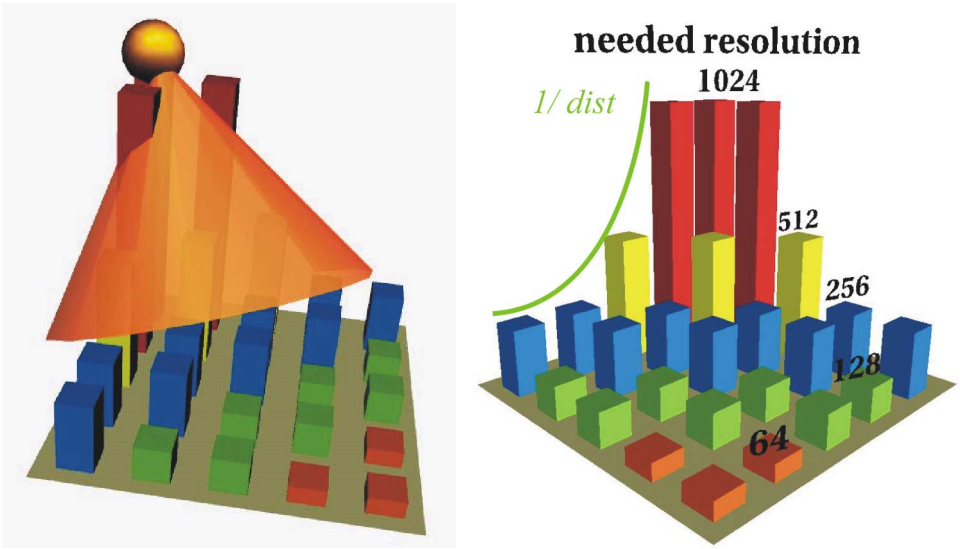


Figure 4.5: *MP-Grid* example: in x and y directions are the extents of the terrain drawn, in z direction is the needed maximal texture resolution for the single pyramids depicted.

As a last consideration, the needed texture memory for texturing with a *MP-Grid* will be compared to the one needed for a *Clip-Map.* A flat area of terrain of the size 5000m $\times$ 5000m is used with the viewer sitting at $(0, 0, 1000)$ and looking towards $(2500, 2500, 0)$. The focal length of the camera is 50 mm and the camera has a pixel resolution of 1024 $\times$ 1024 pixels in screen-space and an angle of the field of view of 45 degrees. A texture of 5120 $\times$ 5120 texels is used which can be subdivided in a *MP-Grid* of 5 $\times$ 5 pyramids with a resolution of 1024 $\times$ 1024 texels in the finest level. Figure 4.5 shows the resulting adaptation of the *MP-Grid* to this viewer position. The total amount of memory needed for the adapted *MP-Grid* is 6,198,605 bytes for a 8 bit grey scale picture. The original size of the MIPmap is 34,952,550 bytes for this 8 bit grey scale picture. For constructing a *Clip-Map,* the same image first has to be converted to a 8192 $\times$ 8192 sized image to have sizes of a power of two. The alternative of reducing the image size to 4096 $\times$ 4096 texels would result in a loss of quality. Then the clipped MIPmap levels must be constructed covering 2048 $\times$ 2048, 4096 $\times$ 4096 and 8192 $\times$ 8192 texels of the original texture. Each of these clipped levels is 2048 $\times$ 2048 texels in size. The non clipped levels below 1024 $\times$ 1024 texels sum together with these three maps up to 13,981,013 bytes. Since the *Clip-Map* cannot handle this picture size of 5120 $\times$ 5120 texels efficiently, the rescaling to a power of two consumes memory while the *MP-Grid* approach can handle such sizes more economically and only half of the memory is needed. Such arguments are important in the field of commercially sold graphic adapters for PC's or game-stations, since in these machines texture memory is a rather expensive resource. The image size of this example (5120 $\times$ 5120 texels) is a size which is already widely in use in GIS-systems (see also the result section at the end). Larger images increase the advantage of *MP-Grids* with respect to memory usage even more because of the scaling problem to powers of two. Furthermore, *Clip-Maps* must be of a square size in the current implementation. If the texture data has not a square size, it must be rescaled. This rescaling process consumes additional memory, otherwise information is lost due to rescaling the data to the smaller side of the rectangle.

4.1.3 Fast Rendering of MP-Grids with OpenGL

The *MP-Grid* can be used directly to address the texture mechanism of OpenGL in its current version 1.1 (see [85]). The *MP-Grid* shall texture polygonal models. Without loss of generality, the models are assumed to be triangle meshes. Otherwise, a triangulation can be generated in a preprocessing step. In OpenGL, textures are described with so called *texture objects.* Furthermore, OpenGL uses a so called *current texture* to texture the primitives. This *current texture* is declared with selecting one of the texture objects. A texture object is a structure which contains all MIPmap levels of a texture and allows the user to change the current texture fast, since only pointers to the texture data in memory have to be changed. A texture object is used in this approach to represent directly one MIPmap pyramid of the *MP-Grid.* Before drawing triangles, the *MP-Grid* is adapted to the current viewer's position. For each MIPmap pyramid, a maximum level is calculated as described above. Depending on the already loaded levels, the pyramid has to be refined or memory can be freed by deleting levels. It cannot always be guaranteed that the texture data requested for triangle rendering will be available at the needed level of detail. In this case, the texturing process must

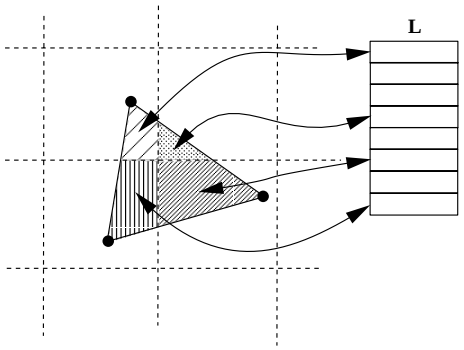


Figure 4.6: Once the triangles are inserted in the list *L*, they are divided and drawn.

use the best resolution of texture data available while the needed data is loaded in parallel. Furthermore, the user can control the tile size of the *MP-Grid.* If it turns out, that with a specific tile size the peak bandwidth of the system is overtaxed, the tile size can be reduced to adapt to the system's capabilities. The algorithm to draw a triangle mesh with a *MP-Grid* has two steps, since the currently available OpenGL API is used. In the first step, each triangle is classified to find out which cells of the *MP-Grid* are covered by this triangle. For this, a rasterization algorithm is used, well known as *Pineda* algorithm (see also [91]). After the classification, there exists a list *L* which holds for each grid cell all the covering triangles (see figure 4.6). In the second step, the triangles are drawn. First the texture object belonging to the grid cell is selected to be the current texture. Next, the triangles are subdivided to find out which part of them is exactly covered by the current texture. For this, the GLU utilities integrated in OpenGL are used, which have a so called *tessellator* that is able to tessellate polygons in 2D and 3D. With this tessellator, the tessellation is calculated in 2D texture space. The tessellation is then mapped with the help of barycentric coordinates determined during this tessellation into 3D object space. Furthermore, when using the tessellator of OpenGL, hardware support can be exploited whenever the OpenGL implementation is able to use this. The tessellation process can be accelerated by using OpenGL display lists to store the tessellation results. Once a triangle is tessellated, the tessellation results are stored in a temporary set of OpenGL display lists, one list for each covered *MP-Grid* tile. As you can see in 4.1, the rendering speed with this caching and using a *MP-Grid* is only 4 times slower than using standard *OpenGL* texture resolution. The described approach, first classifying and then filling, makes sense. Otherwise the current texture would have to be changed for each tessellated part of a triangle. This would be much more costly than classifying the triangles since each new selection of a current texture results in a reset of the texturing unit.

4.1.4 Enhancing Hardware for MP-Grids

This approach using *MP-Grids* can be integrated into a rendering hardware with little extensions (see figure 4.7). First of all, a register, the current texture register (CTR), must be integrated into the texturing unit, that holds for each level λ of the MIPmap pyramids a pointer into a three dimensional pointer table (all MIPmap pyra-

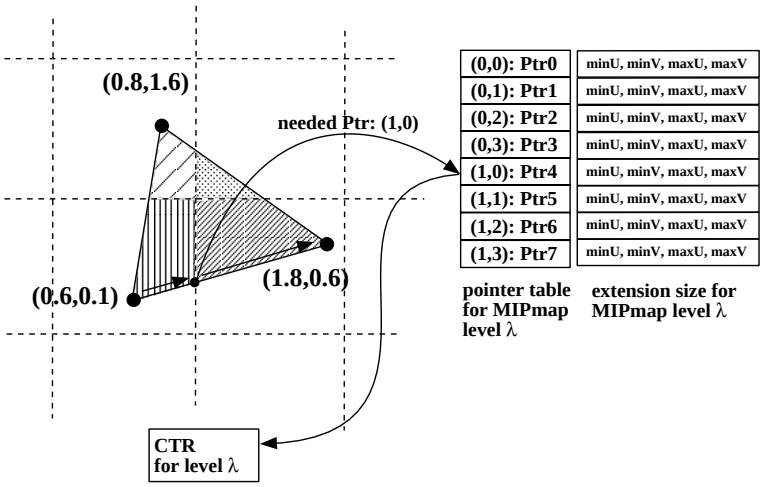


Figure 4.7: Scheme of the needed hardware extension to support *MP-Grids.*

mids are assumed to have the same number of layers). This table is addressed with the level denominator λ and two indices identifying uniquely the single pyramids of the *MP-Grid.* The texture coordinates are no longer in the range [0..1] $\times$ [0..1], they are now defined with a *MP-Grid* of n $\times$ m pyramids in the range [0..n] $\times$ [0..m]. The texture coordinates contain the information of crossing the border between two adjacent MIPmap pyramids during texturing. If such an event is detected by the logic interpolating the texture coordinates, the pointer to the now needed pyramid level is loaded into the CTR for the level λ by using the truncated texture coordinates as indices into the table. If not only the pointer, but also an extension size is stored in the table, MIPmap levels of different MIPmap pyramids but the same level λ can be merged together. With this, lower levels of the *MP-Grid* which comprise only a few texels, can be stored at one position in texture memory which prevents gaps in allocation. The interpolation logic now loads the extension sizes into the comparators used for detecting the border crossing and uses this information for its decision when to change to another pointer. This can prevent ”pointer trashing”, if triangles have to be textured with low levels of MIPmap pyramids comprising only a few texels. Furthermore, it can be economical not to remove such low levels from texture memory, since they are small in size and they are always needed for consistent MIPmap pyramids. With a maximum of 11 MIPmap levels, sufficient for 1024x1024 tiles, a 100x100 *MP-Grid* would need $11 * 100 * 100 * (4 + 4) = 880,000$bytes $= 860$kB for storing the table.

The texture data for the MIPmap pyramids used in the *MP-Grid* could be supplied by the user via an extension to the already available OpenGL texturing commands.

4.1.5 Results

The pictures 4.8, 4.9, 4.10, and 4.11 show the usage of high resolution textures in the system *FlyAway* used for GIS purposes and described in Chapter five.

For performance measurements, triangle meshes of different fixed triangle counts and *MP-Grids* with different tile sizes were used. All pictures and timings were pro-

duced on a low end graphics workstation SGI O_2 with 256MB RAM and a 175 MHz R10000 processor. The *MP-Grid* pyramids had all 1024x1024 texels in their finest level and the texel-size was 24 bit. Screen resolution for all the measurements was 1024×1024 texels.

For pictures 4.8 and 4.9, the original texture image had a size of 4096×4096 texels. Pictures 4.10 and 4.11 result from a 3200×3600 map image which was resized to a size of 4096×4096 texels to calculate the *MP-Grid.* In picture 4.8 and 4.10, the textures were reduced to the size of the standard OpenGL texture resolution of 1024×1024 texels on a SGI O_2. In picture 4.9 and 4.11, the full resolution of 4096×4096 texels using a 4×4 *MP-Grid* with 1024×1024 texels tiles was used to texture the terrain.

In table 4.1 and 4.2, some run time results of using *MP-Grids* with the application are summarized. The *Setup* time is needed for intersection and creation of display lists, the *Redraw* time for drawing. The performance results show, that this approach is usable without the proposed hardware extensions and allows us to improve image quality drastically. Little tessellated surfaces (below 100 triangles) can be drawn in real time. With high triangle counts above 5000, currently both, the *MP-Grid* and the standard OpenGL texture resolution of 1024×1024 texels are used. During motion, texturing is done with the standard resolution and the *MP-Grid* is used when coming to a stop.

Δ	4x4 Setup	4x4 Redraw	2x2 Setup	2x2 Redraw	standard resolution
10,000	2.67s	0.44s	1.8s	0.47s	0.13s
5,000	1.44s	0.21s	0.89s	0.23s	0.6s
2,500	0.88s	0.11s	0.63s	0.09s	0.03s
1,000	0.35s	0.04s	0.20s	0.04s	0.01s
500	0.27s	0.03s	0.11s	0.02s	$<$0.001s
2	0.005s	0.005s	0.005s	0.005s	$<$0.001s

Table 4.1: Performance measurement of *MP-Grids.*

Δ	10,000	5,000	2,500	1,000	500
drawn 4x4	9600	4608	2112	880	352
tess. 4x4	1308	1000	847	399	367
drawn 2x2	9950	4940	2400	960	480
tess. 2x2	300	200	251	131	91

Table 4.2: Number of directly drawn and tessellated triangles. Note: triangles can be tessellated more than once, if they are covered by many MIPmap pyramids.

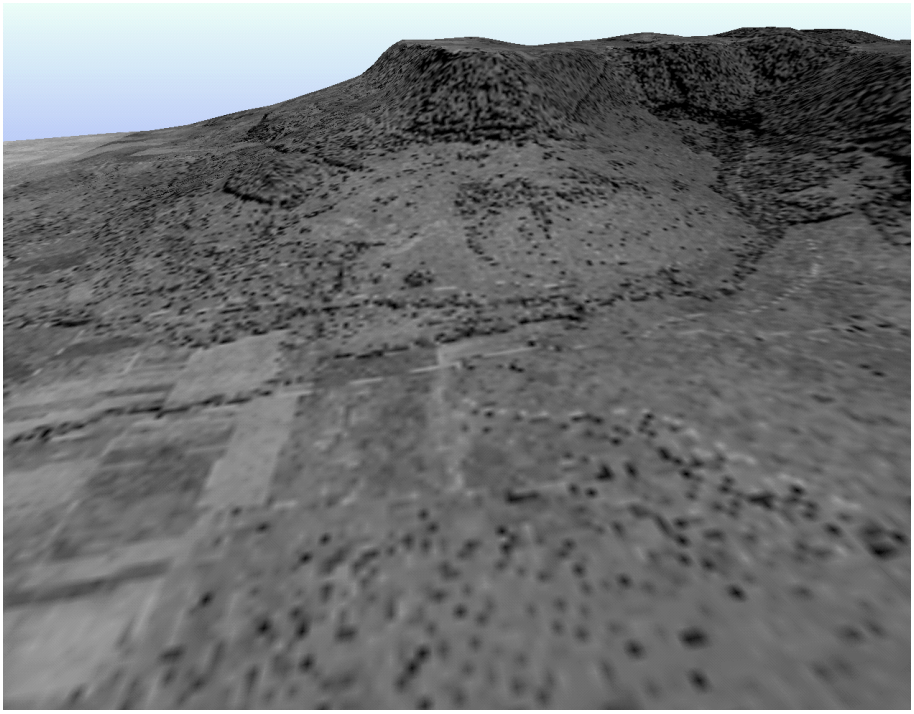


Figure 4.8: 1024 $\times$ 1024 standard resolution

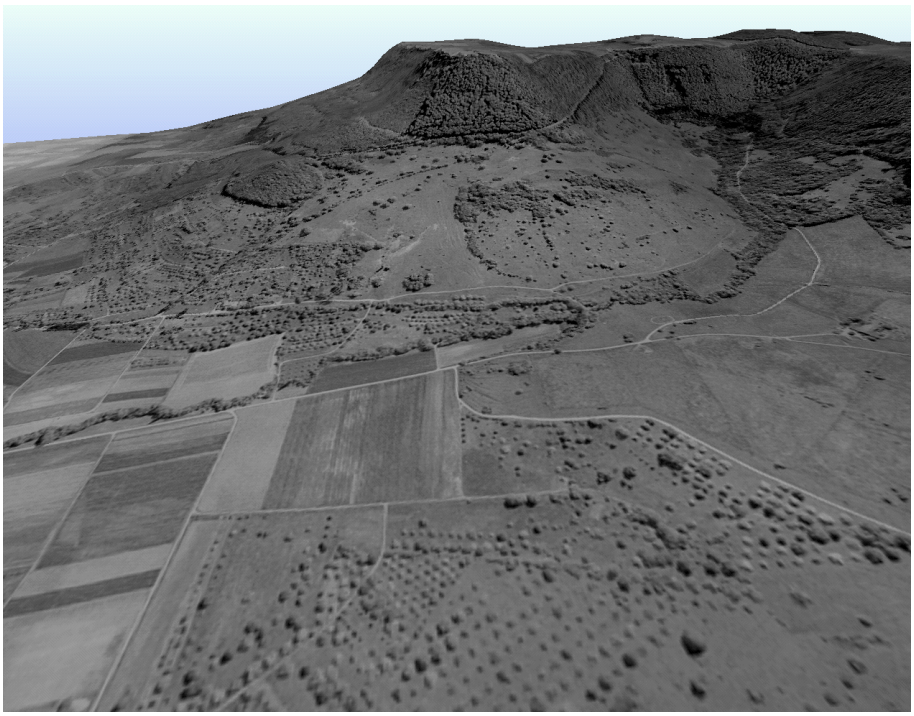


Figure 4.9: 4096 $\times$ 4096 texels using a 4 $\times$ 4 *MP-Grid* with 1024 $\times$ 1024 texels tiles

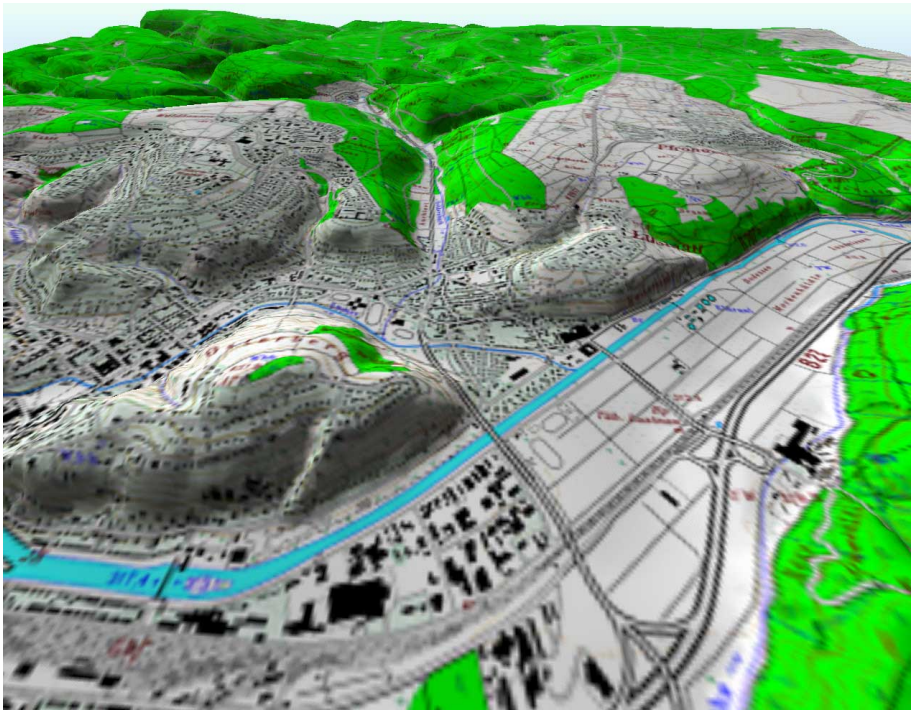


Figure 4.10: 1024 × 1024 standard resolution

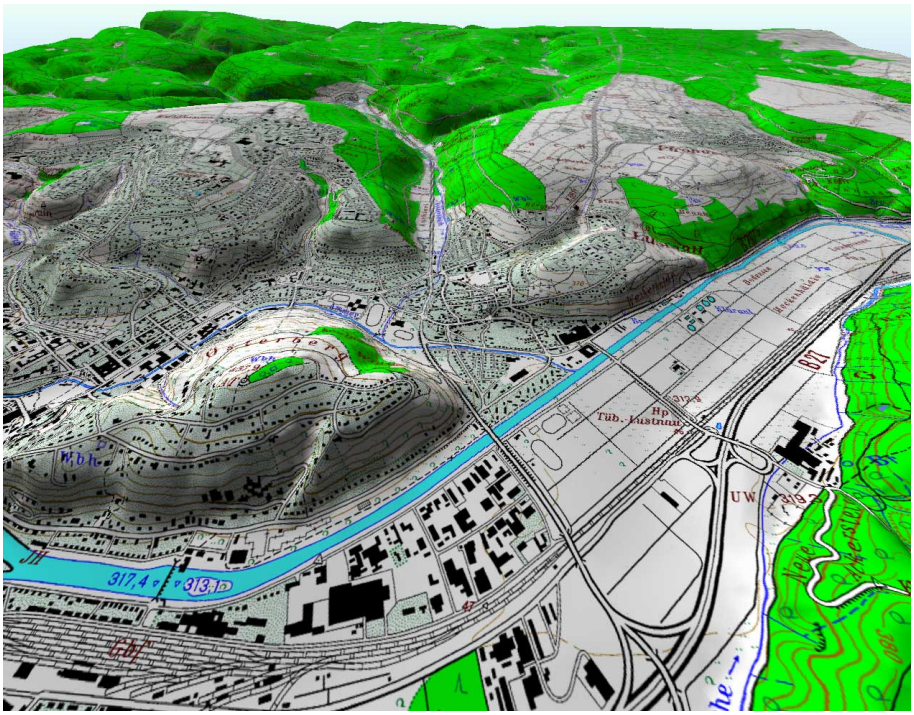


Figure 4.11: 4096 × 4096 texels using a 4 × 4 *MP-Grid* with 1024 × 1024 texels tiles

4.2 Texture Cutting with Projective Texture Mapping

The normal way of selecting a part of an image for texturing is to manipulate texture coordinates accordingly. Unfortunately, the graphics hardware currently available is only able to perform a bilinear interpolation of texture coordinates on rectangles, but not on arbitrary quadrilaterals. In this Section, an approach will be explained which can perform such a bilinear interpolation with the help of a projective mapping on today's hardware. It was developed in a cultural heritage project that was realized together with the University of Padova in Italy. The selection problem is also relevant for digital maps, if they are applied to digital elevation models as textures. Due to the cartographic mapping, maps are usually not aligned to the Cartesian coordinate axis (see Chapter 1).

4.2.1 Virtual Reality for the Conservation of Cultural Heritage

Starting point of this project was the fact, that all over the world rare and ancient books exist in libraries, which could not be made accessible to the public because of their uniqueness and their sensibility. One possibility nowadays available to accomplish this is an exact three-dimensional reproduction with the help of modern computer graphics knowledge and the means of *Virtual Reality*. With this, many people can get access to such a rare and seldom historic piece. Such a reproduction can furthermore be used to document the actual condition of those pieces to be able to detect changes and to serve as a basis for conservation decisions. To build a reproduction of such a book, one has to construct a polygonal model of it which will be textured. This model can be constructed for example with a CAD system or can also result from using a three-dimensional scanner. Constructing the model by hand will usually result in a model having less polygons than a scan, which is advantageous for the rendering speed. Texture mapping is a popular way of enhancing realism of three dimensional scenes without increasing their polygonal complexity. Many publications describe how to use texture mapping to enhance realism [44],[47].

In the next section, some short description of *Andreas Vesalius* will be given, who was a famous medieval doctor and medical researcher. One of his books was used as a test case for this texture cutting approach.

4.2.2 Andreas Vesalius and his Work

The following characterization of Andreas Vesalius is based on [26], [108], and [6]. These sources are available on the WWW and are summarized in this Section.

In [108], following description is given: "Andreas Vesalius (1514-1564) was a Flemish anatomist who founded the sixteenth century heritage of careful observation in anatomy characterized by *refinement of observation*. Vesalius changed the organization of the medical school classroom, bringing the students close to the operating table. He demonstrated that, in many instances, the former anatomists as *Galen* (130-200) and *Mondino de' Luzzi* (1275-1326) were incorrect (the heart, for instance, has four chambers). He conducted his own dissections, and worked from the outside in so as not to damage the cadaver while cutting into it. Vesalius also wrote the first

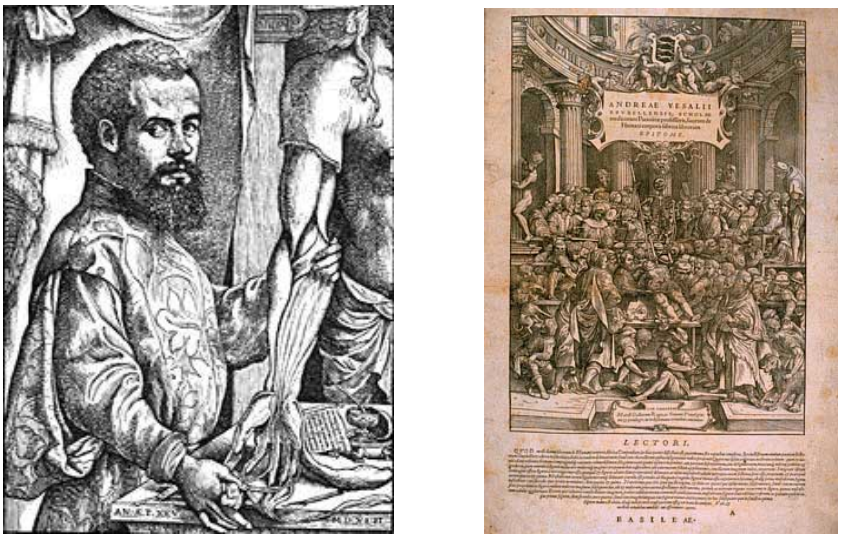


Figure 4.12: Andreas Vesalius (1514-1564) and the titlepage of his main work, the *Epitome*.

anatomically accurate medical textbook, *De Humani Corporis Fabrica* (1543), which was completed with precise illustrations."

He also produced a summary of the *Fabrica*, called *Epitome*, less bulky and less expensive, for the use of medical students and those with little or no anatomical knowledge. In the *Epitome* the illustrations seem to have been considered more important then the text and in consequence the format is even larger than that of the *Fabrica*, where the figures are about six centimeters shorter. The first Latin edition of the *Epitome* was printed by Johannes Oporinus, a friend of Vesalius, in Basel simultaneously with the *Fabrica*.

In [6], the *Epitome* is described in the following way: " The *Epitome* has 12 unnumbered leaves signed A-M; two further unsigned leaves are printed on one side only, containing the figures which were to be cut out and put in place on both the skeleton and the body representing the nervous system.

There exist copies of the *Epitome* that have the superimposed flaps with the anatomical details mounted thus following Vesalius' own instructions on the preparation of the two manikins: "We wish to advise those who obtain unprepared copies, and put them together by their own efforts and industry, on the method of cutting each from the superfluous paper and pasting them on, and then of coloring them according to their ability and desire. In order to provide strength, it will be useful to glue a piece of parchment to the back of the entire sheet so that it may not in vain be divided into as many pieces as there are figures comprising it. " There are only a few copies of the *Epitome* with unmounted flaps, one is located at the institute of medical history of the University of Padova, where Vesalius worked for five years as a research assistant. This book was the basis for this work.

One last remark of [108] summarizes very well the importance of Andreas Vesalius for modern medicine: " Vesalius's careful observation, his emphasis on the active participation of medical students in dissection lectures, and his anatomically accu-

rate textbooks revolutionized the practice of medicine. Through Vesalius's efforts, medicine was now on the road to its modern implementation, although major modifications and leaps of understanding were, of course, necessary to make its practice actually safe for the patient."

An object like the *Epitome* is very sensitive. The only way to reproduce its content is to take photographs of the single pages. Scanning, for example, is impossible, since such books may not be folded up that wide without possible damage. The data basis were 25 slides of all the pages plus slides of front and back cover. These slides were scanned with an resolution of 2.000×3.000 pixels. This results in 6 mega pixels per image or 18 MB storage capacity using truecolor and no compression. The polygonal model of the book itself was produced with standard modeling tools and is stored in a CAD format. The measurements for this model can be taken directly with the help of standard measuring tools. The book model is not static, but can be animated. It is possible to "turn" single pages just as with a real book. The geometry necessary for this task is already contained in the model and activated when it is needed.

4.2.3 Texture Cutting

Due to the generation process of the images, only a part of the images can be used as textures. Usually, it is not possible to fold up and photograph the book in a way, that the pages are aligned exactly in the camera window. Therefore, a part of the image has to be selected for texturing. Texture selection can be easily achieved with manipulating the texture coordinates of a mesh.

There exist many approaches for generating texture coordinates for a given mesh. In [72], a comprehensive summary of techniques and solutions for this problem is given. All these techniques calculate texture coordinates Ω for a set of vertices of a surface, which form a triangulation and therefore a discrete approximation of this surface.

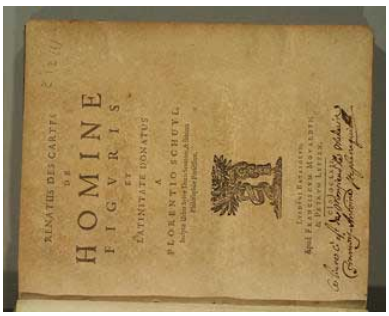


Figure 4.13: Photographed image of a book page.

The selection process is demonstrated in figure 4.13, figure 4.14, and figure 4.15. In figure 4.13, a cover page of a historic book can be seen. Figure 4.14 shows the part of the image which shall be used for texturing as the quadrilateral $[X_0, X_1, X_2, X_3]$. The result can be seen in figure 4.15.

Similar selections have to be made, if digital maps shall be applied to digital elevation models. Due to the cartographic mapping, the map is not aligned to the Cartesian

Figure 4.14: Part of the image selected for texturing.

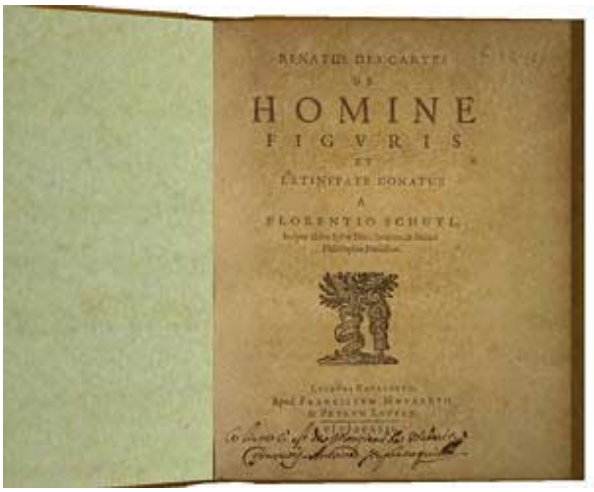


Figure 4.15: Virtual book model with this texture.

coordinate axis (see Chapter 1).

Current rasterization hardware is only able to perform normal linear interpolation along the two Cartesian axis during the rasterization and texture mapping process of a triangle. A projective transformation M can be calculated, which allows a bilinear interpolation on the quadrilateral $[X_0, X_1, X_2, X_3]$. This approach has several advantages:

1. It is very simple to define the quadrilateral $[X_0, X_1, X_2, X_3]$ directly in image coordinates.

2. This approach utilizes the capabilities of modern *OpenGL* graphics hardware, since the four-dimensional texture coordinates are available and can be interpolated linearly during rasterization. Originally, they were defined for projective textures and real-time shadow and lighting effects, but they are also usable in this way. Other, possibly non linear, remappings of texture coordinates during rasterization on a per pixel basis are currently not possible in the available graphics hardware and can only be realized with approaches like ray tracing.

The remapping of the texture coordinates to four dimensions with the help of M can be done in most graphic systems in hardware and therefore on the fly, since the *OpenGL* API has a special four-dimensional transformation matrix for transforming texture coordinates before the rasterization. Therefore, the matrix M can be transmitted to the graphics subsystem to transform the set of texture coordinates Ω of a given mesh temporarily during rasterization into the set Ω_M usable for the texture image M was calculated for.

When the texture changes to another texture image to which the matrix M belongs, M is replaced by M' in the graphics subsystem.

3. Using traditional image processing tools like *Photoshop* for cutting the texture, the image may be resampled up to three times:

   (a) Texture selection and rectification.

   (b) Rescaling for texture filtering during MIPmap calculation.

   (c) Texture mapping during rasterization.

   The number of resamplings can be reduced greatly with this approach to only one resampling process taking place during texture mapping.

   - The first resampling of the selection is not needed due to the transformation M.
   - With the idea explained in figure 4.16, the rescaling of the input image during the MIPmap calculation can be avoided: The right/upper border of the image can be extended to image dimensions of a power of two by clamping the right/upper texel row (see figure 4.16). The filtering of the border texels during MIPmap calculation will be correct, since they are averaged with the clamped extension texels having the same value.

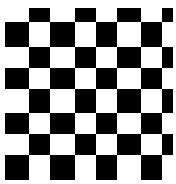


Figure 4.16: Clamping the borders of an image to prevent rescaling during texture filtering.

For the purpose of demonstration, some cartographic image examples are used, since there the blurring due to too much resampling can be demonstrated more clearly. The effect of this can be seen in the figures 4.17, 4.18, and figure 4.19. The original image, a part of a digital map is shown in figure 4.17. A conventional cropping, rectifying, filtering and texture mapping process produces figure 4.18. The mapping with M results in figure 4.19.

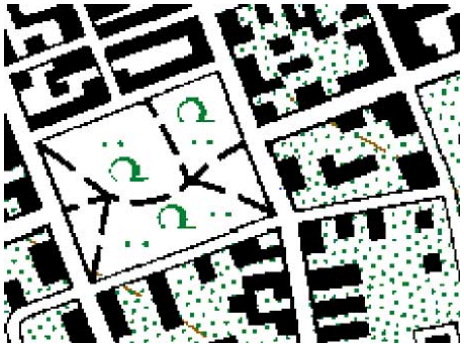


Figure 4.17: Original image for texturing.

Figure 4.18: Three resampling processes during texture mapping.

It can be clearly seen, that figure 4.19 conserves more from the original sharp image than can be realized with using multiple resamplings in figure 4.18. This is no dramatic improvement, but doing high quality texture mapping means also to preserve details in the texture during texture mapping and to avoid things from getting blurry. This is what shall be done with the systems available today to realize the intended visualization goals for the virtual book.

The next paragraph will now explain, how the transformation matrix M can be computed. It is not possible to define an affine two-dimensional mapping A

$$\begin{aligned} A\,[0,\,0] &= [x_0,\,y_0] \\ A\,[1,\,0] &= [x_1,\,y_1] \\ A\,[0,\,1] &= [x_2,\,y_2] \\ A\,[1,\,1] &= [x_3,\,y_3] \end{aligned} \tag{4.1}$$

which transforms bilinearly as depicted in equation (4.1) the input coverage $[0..1] \times [0..1]$ into the output coverage $[X_0, X_1, X_2, X_3]$.

In a four-dimensional projective space, a projective transformation M can be determined for this calculation. The calculation is described in four dimensions due to the four-dimensional matrix in the hardware texturing unit M is loaded into.

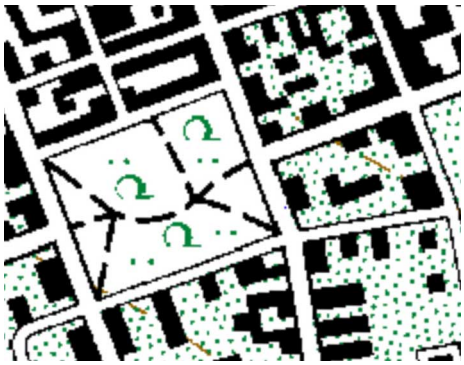


Figure 4.19: Projective texture cutting avoids resampling processes.

First $[X_0, X_1, X_2, X_3]$ is extended to the projective space. This results in

$$\begin{aligned}
\widetilde{X_0} &= [x_0,\ y_0,\ 0,\ 1] \\
\widetilde{X_1} &= [x_1,\ y_1,\ 0,\ 1] \\
\widetilde{X_2} &= [x_2,\ y_2,\ 0,\ 1] \\
\widetilde{X_3} &= [x_3,\ y_3,\ 0,\ 1]\,.
\end{aligned} \tag{4.2}$$

The matrix M has 16 coefficients

$$M = \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \tag{4.3}$$

M can be calculated with the following 4 equations

$$\begin{aligned}
M\,[0,\ 0,\ 0,\ 1] &= \widetilde{X_0} \\
M\,[1,\ 0,\ 0,\ 1] &= \widetilde{X_1} \\
M\,[0,\ 1,\ 0,\ 1] &= \widetilde{X_2} \\
M\,[1,\ 1,\ 0,\ 1] &= \widetilde{X_3}\,.
\end{aligned} \tag{4.4}$$

Some trivial considerations eliminate a couple of coefficients which are set as

$$\begin{aligned}
c &= 0 \\
d &= x_0 \\
g &= 0 \\
h &= y_0 \\
i &= 0 \\
j &= 0 \\
k &= 0
\end{aligned}$$

$$\begin{aligned} l &= 0 \\ o &= 0 \\ p &= 1\,. \end{aligned} \tag{4.5}$$

The other coefficients can be calculated using symbolic linear equation solving and result to

$$\begin{aligned}
a &= \frac{x_0\,x_3\,y_2 + x_0\,x_2\,y_1 - x_0\,x_2\,y_3 - x_0\,x_3\,y_1}{x_1\,y_2 - x_1\,y_3 - x_3\,y_2 - x_2\,y_1 + x_2\,y_3 + x_3\,y_1} + \\
&\quad \frac{-x_2\,x_1\,y_0 + x_1\,x_3\,y_0 - x_1\,x_3\,y_2 + x_2\,x_1\,y_3}{x_1\,y_2 - x_1\,y_3 - x_3\,y_2 - x_2\,y_1 + x_2\,y_3 + x_3\,y_1} \\
b &= -\frac{x_0\,x_1\,y_2 - x_0\,x_1\,y_3 - x_0\,x_3\,y_2 + x_0\,x_3\,y_1}{x_1\,y_2 - x_1\,y_3 - x_3\,y_2 - x_2\,y_1 + x_2\,y_3 + x_3\,y_1} - \\
&\quad \frac{-x_2\,x_1\,y_0 + x_2\,x_3\,y_0 + x_2\,x_1\,y_3 - x_2\,x_3\,y_1}{x_1\,y_2 - x_1\,y_3 - x_3\,y_2 - x_2\,y_1 + x_2\,y_3 + x_3\,y_1} \\
e &= \frac{-y_0\,x_1\,y_2 + y_0\,x_1\,y_3 + y_0\,x_3\,y_2 - y_0\,x_2\,y_3}{x_1\,y_2 - x_1\,y_3 - x_3\,y_2 - x_2\,y_1 + x_2\,y_3 + x_3\,y_1} + \\
&\quad \frac{-y_3\,x_0\,y_1 + y_2\,x_0\,y_1 - y_2\,x_3\,y_1 + y_3\,x_2\,y_1}{x_1\,y_2 - x_1\,y_3 - x_3\,y_2 - x_2\,y_1 + x_2\,y_3 + x_3\,y_1} \\
f &= -\frac{-y_0\,x_1\,y_3 - y_0\,x_2\,y_1 + y_0\,x_2\,y_3 + y_0\,x_3\,y_1}{x_1\,y_2 - x_1\,y_3 - x_3\,y_2 - x_2\,y_1 + x_2\,y_3 + x_3\,y_1} - \\
&\quad \frac{y_2\,x_0\,y_1 - y_2\,x_0\,y_3 + y_2\,x_1\,y_3 - y_2\,x_3\,y_1}{x_1\,y_2 - x_1\,y_3 - x_3\,y_2 - x_2\,y_1 + x_2\,y_3 + x_3\,y_1} \\
m &= \frac{x_2\,y_1 - x_3\,y_1 + x_3\,y_0 + y_2\,x_0}{x_1\,y_2 - x_1\,y_3 - x_3\,y_2 - x_2\,y_1 + x_2\,y_3 + x_3\,y_1} + \\
&\quad \frac{-x_1\,y_2 - y_0\,x_2 - x_0\,y_3 + x_1\,y_3}{x_1\,y_2 - x_1\,y_3 - x_3\,y_2 - x_2\,y_1 + x_2\,y_3 + x_3\,y_1} \\
n &= -\frac{-x_1\,y_0 + x_1\,y_2 + x_3\,y_0 - x_3\,y_2}{x_1\,y_2 - x_1\,y_3 - x_3\,y_2 - x_2\,y_1 + x_2\,y_3 + x_3\,y_1} - \\
&\quad \frac{x_0\,y_1 - x_0\,y_3 - x_2\,y_1 + x_2\,y_3}{x_1\,y_2 - x_1\,y_3 - x_3\,y_2 - x_2\,y_1 + x_2\,y_3 + x_3\,y_1}\,.
\end{aligned} \tag{4.6}$$

All the calculations were made with *Maple*. This has the advantage, that *Maple* can convert its calculations into an optimized C source code which can be directly copied into the *OpenGL* implementation.

4.2.4 Tiled Textures

As already depicted, the approach of Section 4.1 is used to extend the texture capabilities of *OpenGL* to handle arbitrarily large textures.

To use the texture cutting approach from the last section also for *MP-Grids*, a tessellation of the texture in the original Cartesian coordinates has to be produced that respects the tile boundaries. Only in the original Cartesian coordinate space, the proper texture coordinates for the temporary points can be calculated which will then again be transformed by M (see figure 4.20).

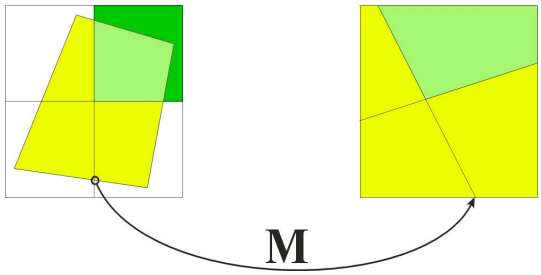


Figure 4.20: A *MP-Grid* with projective texture cutting.

4.2.5 Results

In this Section, some screenshots will be presented in Figure 4.21 - Figure 4.23 produced with the prototype implementation. It is called *showBook* and its only requirement is the existence of an *OpenGL* runtime environment for a specific system. It runs therefore already on a broad range of computer system, ranging from a small laptop up to a SGI Infinite Reality system. The amount of texture which can be displayed depends of course on the actual executing system. On a laptop with 32MB RAM, the number and the resolution of the textures has to be reduced to be able to navigate around the book, since the whole graphic system is emulated by the CPU (Pentium 166). The screenshots were taken on a SGI O_2 with 384 MB RAM and 180 MHz. On this system, the textures for several book pages with the full scanning resolution of 2.000×3.000 pixels can be handled interactively and the pages can be turned in real-time. The application performs even better on modern Pentium II systems running with more than 300 MHz.

Figure 4.21: Bird eye view of the book.

Figure 4.22: Turning a page.

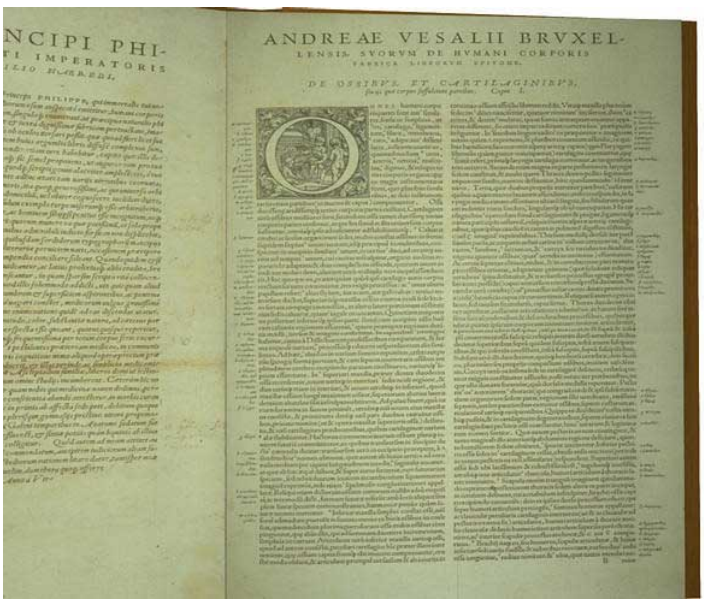


Figure 4.23: First page of the book.

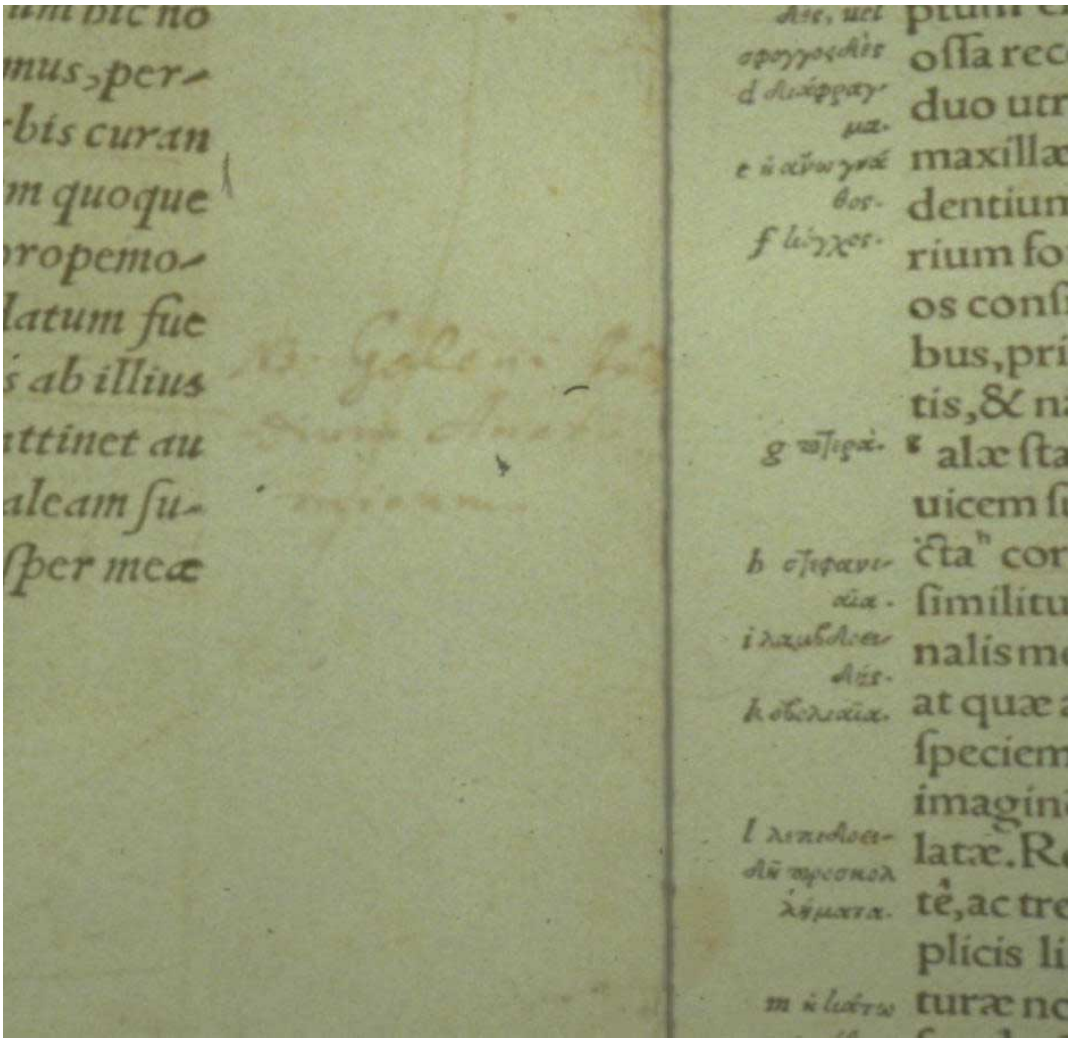


Figure 4.24: Detail of the first page. Someone has written a remark beside the original text.

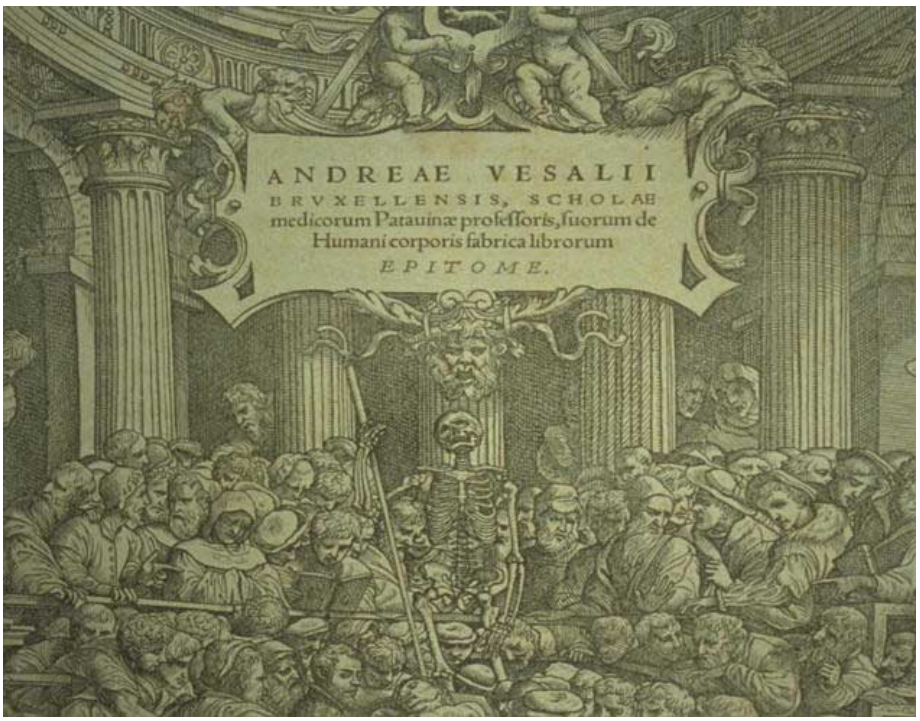


Figure 4.25: Detail of the front page.

Figure 4.26: Going even more into the detail. This is the maximum that can be achieved with the current scanning resolution of 2.000×3.000 pixels.

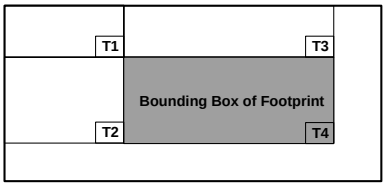


Figure 4.27: Summed-Area Table.

4.3 Texture Filtering

4.3.1 Texture Filtering Approaches

During the rasterization process, mapping images onto objects can be considered as the problem of determining a screen pixel's projection onto a texture image and computing an average value which best approximates the correct pixel color. This projection is usually called *footprint* and the process of finding its color value is known as *texture filtering*. The filtering process is responsible for the visual quality of texture mapping and unfortunately, there exist no convincing solutions up to now. Therefore, the visual impression of a textured object can be very blurry. The following sections (see also [56]) describe a new approach to remedy this which is still feasible for integrating it in hardware.

The notation of [97] was adopted for the following discussion. In real-time environments, where several tens of millions of pixels per second are issued by a fast rasterizing unit, hardware expenses for image mapping become substantial and algorithms must therefore be chosen and adapted carefully. Thus, the straightforward approach of taking the mean of all image texels t inside the footprint for the screen pixel's color $C(x, y)$

$$C(x, y) = \frac{1}{M} \cdot \sum_{i=1}^{M} t_i \,, \tag{4.7}$$

or, more generally, defining a filter kernel h, which is convolved over the image $t(\alpha, \beta)$ (see also [2])

$$C(x, y) \; = \; \int \int \left(h(x - \alpha, y - \beta) \cdot t(\alpha, \beta) \right) d\alpha d\beta \tag{4.8}$$

can be excluded from further discussions due to the long computing times. *Summed-area tables* [19] are an attempt to simplify and speed up the above operation. Instead of the color value, each cell of a summed-area table holds the sum of all values in a certain region, usually the rectangle defined by the position of the cell and the origin as indicated in Figure 4.27.

Given the bounding box of a footprint, $C(x, y)$ is then approximated by accessing the table four times and performing the following operation:

$$C(x, y) = T4 \; - \; T3 \; - \; T2 \; + T1. \tag{4.9}$$

However, since the footprint of a pixel is not rectangular, but can be considered as a quadrilateral in the general case, a potentially large number of texels within the bounding box contributes without reason to the pixel color. Glassner proposes a solution in [37] to incrementally remove rectangles within the bounding box to best approximate the footprint at the cost of increased computing time.

For two reasons, summed-area tables are not well suited for direct hardware implementation:

1. For each pixel, four accesses must be made that can have very different locations, depending on the bounding box of the footprint. This limits the achievable texturing speed.

2. If the color components are 8-bit quantities, a 1024×1024 summed-area table requires entries as wide as $24 + 4$ bits for each color component.

Another approach is to create a set of prefiltered images, which are selected according to the level of detail (the size of the footprint) and used to interpolate the final pixel color. The most common method is to organize these maps as a *MIPmap* as proposed by Williams, see [109] and Figure 4.1. In a MIPmap, the original image is denoted as level 0. In level 1, each entry holds an averaged value and represents the area of 2×2 texels of level 0. This is continued, until the top-level is reached, which has only one entry holding the average color of the whole texture. Thus, in a square MIPmap, level n has one fourth of the size of level $n - 1$.

The shape of the footprint is assumed to be a square of size q^2, where q is suggested in [46] as

$$q = max\left(\sqrt{\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2}, \sqrt{\left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2}\right) . \tag{4.10}$$

In equation (4.10), u and v denote texture coordinates and x and y are screen coordinates.

The MIPmap is accessed by the texture coordinate pair (u, v) of the pixel center and the level λ which in the general case is a function of $\log_2 q$. λ can be expressed with its integer part λ_i and its fractional part λ_f as

$$\begin{aligned} \lambda_i &= \lfloor \log_2 q \rfloor \\ \lambda_f &= \frac{q}{2^{\lambda_i}} - 1 \end{aligned} \tag{4.11}$$

Nearest-neighbor sampling is inadequate due to severe aliasing artifacts. Instead, the levels λ and $\lambda + 1$ are accessed and bilinearly interpolated at (u, v). The final pixel value is linearly interpolated from the result in both levels according to λ_f.

MIPmapping is a reasonable candidate for a hardware implementation due to its regular access pattern. There exist various approaches and architectures (for example [97]) to implement it directly into logic-embedded memories. Due to the high costs of chip development and chip productions, these approaches weren't realized for a broad range of systems. Nevertheless, MIPmapping is the classical filtering approach

used over the last decade in graphics systems and it is nowadays available in nearly every PC graphics card. But the approximation of the footprint with a square limits the MIPmap approach severely and people will try to improve it as graphics systems get more powerful on the one hand and the increasing demand for high quality, but cheap visualization on the other hand.

One filtering approach, called *footprint assembly*, is described in detail in [97]. Its basic idea is the approximation of the projection of the pixel on the texture by a number N of square mipmapped texels. The pixel's deformation is neglected and it is approximated with a parallelogram given by

$$\mathbf{r_1} = \left[\frac{\partial u}{\partial x}, \frac{\partial v}{\partial x}\right] \text{ and } \mathbf{r_2} = \left[\frac{\partial u}{\partial y}, \frac{\partial v}{\partial y}\right] \tag{4.12}$$

The pixel's center $\mathbf{p}$ in the texture map is the intersection point of the diagonals $\mathbf{d_1}$ and $\mathbf{d_2}$ of the parallelogram. The direction $\mathbf{r}$ in which to step from the pixel center to best approximate the footprint is determined from the larger of the two vectors $\mathbf{r_1}$ and $\mathbf{r_2}$ and

$$\begin{aligned} q &= min(|\mathbf{r_1}|, |\mathbf{r_2}|, \mathbf{d_1}, \mathbf{d_2}) \\ N &= \frac{(max(|\mathbf{r_1}|, |\mathbf{r_2}|)}{q}, \end{aligned} \tag{4.13}$$

rounded to the nearest power of two as the number of square mipmapped texture elements for the footprint. A difference vector $\Delta\mathbf{r} = (\Delta u, \Delta v)$ is constructed and a sequence of sample points is generated to cover the footprint.

Footprint assembly is able to produce high quality texture filtering, but it has the drawback of being computationally intensive. Therefore, [97] proposes a hardware for a logic embedded memory device, which can perform this filtering method during memory access.

This approach has been adopted in the *TALISMAN* architecture which uses a weighted anisotropic filtering, see [106].

All of these approaches are difficult and costly to be integrated into current rasterizing hardware, since they either require a great amount of computational power or are based on very special system architectures.

It is the goal of this approach to develop a method, which provides a filter quality comparable with footprint assembly but which is more easily integrated into actual graphics architectures.

4.3.2 Fast Footprint Filtering

Starting from the MIPmap, it can be easily detected that this filtering method wastes texel information by approximating the footprint by a square, where the footprint is an arbitrary quadrilateral.

Improving filtering means to find a trade off between loading more texels to texture a screen pixel and using the loaded texels more efficiently.

The number of texels that can be loaded for real-time filtering is restricted due to strict constants like memory bandwidth or bus width. This limit will be called $\mathcal{M}$.

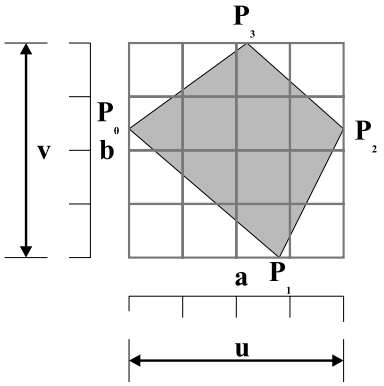


Figure 4.28: Definition of the footprint.

$\mathcal{M}$ has to be respected very strictly, since otherwise system performance will decrease heavily.

4.3.3 Calculating a MIPmap level

The first problem to be solved is depicted in Figure 4.28 which shows a footprint $[P_0, P_1, P_2, P_3]$. Its bounding box has in texture coordinates the extension (u, v). It is now necessary to find a suitable rectangle of $a \times b$ texels from MIPmap level λ to cover the footprint and to respect at the same time the limit $\mathcal{M}$.

The following considerations lead to the calculation of a, b, and λ.

From (u, v), the aspect ratio $f = \frac{u}{v}$ of the bounding box can be determined.

Then one can calculate

$$a \cdot b = \mathcal{M} \text{ and } f = \frac{a}{b} \Rightarrow \frac{a^2}{f} = \mathcal{M} \Rightarrow a = \sqrt{\mathcal{M} \cdot f}\,. \tag{4.14}$$

Furthermore, the following settings are made

$$a' = a \text{ and } b' = \frac{\mathcal{M}}{a}. \tag{4.15}$$

These values can be used to calculate two MIPmap levels m and n for d and b'

$$\frac{u}{2^m} = a', \frac{v}{2^n} = b' \;\Rightarrow\; m = \frac{\log \frac{u}{a'}}{\log 2}\,,\; n = \frac{\log \frac{v}{b'}}{\log 2}\,. \tag{4.16}$$

From this, λ can be determined as

$$\lambda = \lceil \max(m, n) \rceil \tag{4.17}$$

Having λ, the level is known that has to be accessed to get the maximum amount of texture information to cover the footprint and to respect $\mathcal{M}$.

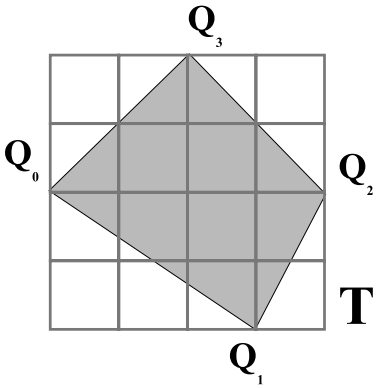


Figure 4.29: Transformation of corner points to integer positions.

4.3.4 Definition of the weighting table

To do a correct filtering, the contributions of the single texel values to the final pixel value have to be calculated. This is done with a precalculated lookup table, since calculating this on the fly would be too expensive.

First, the corner points of the footprint are transformed to the integer positions of the texel grid in level λ. This is shown in Figure 4.29 and generates the quadrilateral $[Q_0, Q_1, Q_2, Q_3]$.

The contribution of a texel to the footprint is now a fixed value. For each possible footprint, a vector ω can be calculated consisting of $\mathcal{M}$ weights which represent the footprint's coverage for each texel. Computing the weights is a preprocessing step and once it is done, the result can be stored in a lookup table. With the help of these weights, a filtering can be performed, since they represent the coverage of the footprint. The texels fetched from memory are stored in a linear array T and the weighting vectors in a *weighting table* W.

The filtered pixel value C can then be calculated as

$$C = \sum_{i=1}^{\mathcal{M}} \left(T[i] \cdot (\omega[i]) \right). \tag{4.18}$$

The weighting vectors allow an easy and efficient computation of the footprint's coverage, but since a footprint is a quadrilateral having four corner points, a huge amount of weighting vectors has to be calculated and stored. For the situation in Figure 4.29, where $\mathcal{M} = 16$ holds, precalculating the weighting vectors for all possible footprints would result in $25 \cdot 24 \cdot 23 \cdot 22 = 303,600$ vectors, since there are 4×4 texels, but only 5×5 possible corner positions.

Each vector has 16 entries and one would have to provide storage for $4,857,600$ weights.

By dividing the quadrilateral $[Q_0, Q_1, Q_2, Q_3]$ into two triangles Δ_1 and Δ_2 and filtering each of them separately, the number of needed weights can be reduced significantly, see Figure 4.30. For the given example, only $25 \cdot 24 \cdot 23 = 13,800$ vectors requiring $220,800$ weights are needed.

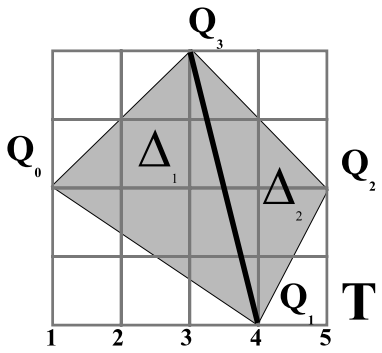


Figure 4.30: Divide footprint for table lookup.

This amount can be reduced even further, if the corners are not transformed to integer positions of the texel grid, but to the mid-points of the texels they lie in. This is not as accurate as using the integer positions, since if a corner point of a texel lies on such an integer position, it can be snapped to up to four possible neighboring mid-points. Therefore, this snapping is no longer a unique solution and one has to use a heuristic to ensure a consistent snapping if a corner point is snapped more than once in successive footprints. Always the mid-point to the top and to the right for corner points on integer positions is chosen. With this, aliasing due to inconsistent mid-point snapping can be prevented and the error is in maximum the half of a texel. In the example from Figure 4.28 are 4×4 mid-point locations possible and this ends up with $16 \cdot 15 \cdot 14 \cdot\ 16\ = 53,760$ weights.

The weighting table W is depicted in Figure 4.31. It is accessed in three stages with a multi-stage lookup structure, one for each corner point. For the example above, the lookup structure consists of $16 + 16 \cdot 15 + 16 \cdot 15 \cdot 14\ = 3616$ pointers for mid-point snapping. In Figure 4.31, a single byte value in the range $[0..255]$ represents the weight that will be linearly scaled to $[0..1]$ during the weighting calculation.

Currently, the corner positions are numbered regularly as depicted in Figure 4.30. Some combinations of corner points can never occur. There lie always more than two corner points on the borders of T respectively on the mid-points of border texels, since T can be oriented this way when covering the footprint in level λ. By exploiting this fact, the two corner points on the border can be placed at 12 respectively 11 different positions in the example above. Therefore, the amount of necessary weights can be again reduced to $12 * 11 * 14 * 16 = 29,568$ values. Especially when $\mathcal{M}$ is small, a whole series of weighting tables can be calculated in advance for all possible bounding boxes with $a \cdot b = \mathcal{M}$. For $\mathcal{M} = 16$, the table size needed is 3036 vectors $*16$ weights $= 48,576$ weights for the table with 4×4 texels. The one for 2×8 needs $139,840$ weights (2×8 and 8×2 are the same due to symmetry). With this, elongated and distorted footprints can be approximated better.

Table 4.3 summarizes the sizes of W and the pointer structures for different values of $\mathcal{M}$. The values are calculated for integer positions (I) and for mid-point snapping (MS). Since in current architectures, $\mathcal{M}$ is realistically restricted to be ≤ 32, there is no space problem with having more than one table, since W and the pointer structures needed to access W have still feasible sizes. But even if $\mathcal{M}$ is increased to 64, approx-

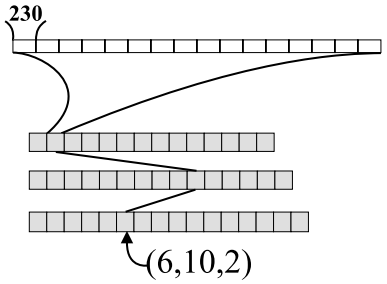


Figure 4.31: Accessing the weighting table.

$\mathcal{M}$	size of W		pointers
8	(MS 2x4)	$8 * 7 * 6$ *8=2,688	400
8	(I 3x5)	12*11*13 *8=13,728	2,955
16	(MS 4x4)	12*11*14 *16=29,568	3,136
16	(MS 2x8)	16*15*14 *16=53,760	3,136
16	(I 5x5)	12*11*23 *16=48,576	14,425
16	(I 3x9)	20*19*23 *16=139,840	18,279
32	(MS 2x16)	32*31*30 *32=952,320	30,784
32	(MS 4x8)	20*19*30 *32=364,800	30,784
32	(I 3x17)	36*35*49 *32=1,975,680	127,551
32	(I 5x9)	24*23*49 *32=865,536	87,165
64	(MS 8x8)	28*27*62 *64=2,999,808	254,080
64	(I 9x9)	32*31*79 *64=5,015,552	518,481

Table 4.3: Sizes of the structures needed for fast footprint filtering.

imately 7 MB of memory are needed to store W and the pointer structures (one pointer is assumed to be 4 byte to address 2^{24} possible values). These sizes of the weighting table and the pointer structure can be handled with current chip technology. Using the lookup table W, two weighting vectors ω_1 and ω_2 belonging to the triangles Δ_1 and Δ_2 can be generated. The filtered pixel value C can now be calculated as

$$C = \sum_{i=1}^{\mathcal{M}} \left(T[i] \cdot (\omega_1[i] + \omega_2[i]) \right). \tag{4.19}$$

4.3.5 Hardware Realization

The algorithm shown above can be realized with standard hardware components and is organized in a pipeline having the following successive stages:

- Determination of the weighting vector
  Here a multi-stage lookup unit is needed consisting of multiplexers and decoders and a ROM for the vectors. The unit converts the indices of the corner vertices into an access to the ROM table. As already depicted, not all combinations of corner indices can occur. This is coded in the structure and saves memory in the

ROM table. Currently, the symmetry of triangles covering the weighting mask is not used to further reduce the number of necessary vectors, since this would mean a reordering of the footprint corners that would need additional hardware. The design shall be stream-line, only consisting of lookups, texel fetches and the final evaluation of the convolution in Equation (4.19). This ensures speed and can be realized more economically.

- Texel Array T
  The values that are read from texture memory are stored in T before they are combined with the weights. The texture memory access itself can be greatly accelerated by using banking and caching techniques, since adjacent footprints have a coherent memory access pattern (see [58]).

- Evaluation of Equation (4.19)
  This evaluation can be performed with the help of a scalar vector multiplication unit and a second vector unit for calculating the sum of a vector's components.

For this approach, no interpolation units are necessary, which are necessary for a good quality MIPmapping. Instead lookup tables and a unit which calculates the final pixel value given in equation (4.19) are used. This hardware effort is comparable to the one needed for MIPmapping and can also deliver a similar performance, since only basic arithmetic functions are used.

4.3.6 Results

The following measurements were produced with a software prototype of the algorithm built into a ray tracing system. Also the other filters, MIPmapping and footprint assembly, were implemented.

To compare the approaches not only visually, but also statistically, Figure 4.42, Figure 4.43 and Figure 4.44, show how the algorithm behaves in selecting MIPmap levels. The pixels are rendered in this ray tracer from the top row to the bottom row. Therefore, the switching between MIPmap levels can be reported, since the test scene consists of a textured, flat plane which is sampled with the ray tracer. In these diagrams, the horizontal direction represents the pixel number as the calculation proceeds. In vertical direction, the used MIPmap level is depicted. It turns out, that the method switches earlier to lower levels compared to MIPmapping, and a bit later than foot print assembly. This is mainly due to the effect explained in Figure 4.32. Rather distorted footprints extend the bounding box as depicted for the left footprint and the new method is therefore forced to switch to a higher MIPmap level, but will still sample the footprint correctly with the help of the weighting vectors. It can be seen, that increasing the tablesize $\mathcal{M}$ reduces this behavior and for $\mathcal{M} = 16$ and $\mathcal{M} = 32$, fast footprint filtering catches up with footprint assembly.

Setting $\mathcal{M}$ to 32 is reasonable, since modern graphics chips like the *Riva TNT* chip produced by *NVidia* don't load any longer only the 8 texels necessary for a trilinear MIPmapping. This special chip supports anisotropic filtering and takes up to 8 bilinear samples from up to two adjacent MIPmap levels and supports anisotropy of up to 2:1. With this, already 32 texels have to be loaded. The current implementation has following features:

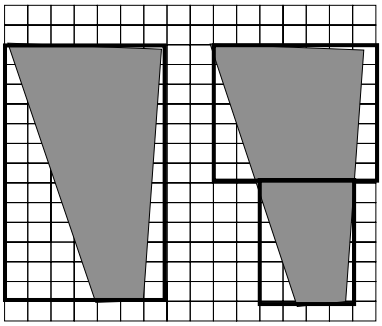


Figure 4.32: Using two arrays for distorted footprints.

- Anisotropic filtering is only necessary for a small amount of footprints with heavy distortions. It is therefore possible, to combine MIPmapping and fast footprint filtering and to use the latter to filter only distorted footprints. This reduces the amount of texture data accessed, since for trilinear MIPmapping, only eight texel values have to be loaded.

- There is currently not a fixed limit $\mathcal{M}$ for texel fetching, but this limit is adopted to the footprint characteristics. If footprints with a difficult shape have to be sampled, the size of $\mathcal{M}$ can be raised up to $2 * \mathcal{M}$ which results in a slower sampling due to two steps of fast footprint filtering, but means also improved sampling quality. Footprints, that are more isotropic, are sampled with smaller tables having less than $\mathcal{M}$ texels or they are filtered with MIPmapping. Furthermore, normal bilinear interpolation is used to access the first level of the MIPmap, if the size of a footprint is smaller than the texel size at the finest level.

  With this, a better sampling quality can be achieved without increasing the overhead as much as fixing $\mathcal{M}$ on a high level.

When analyzing this for Figure 4.35, the following distribution of texture accesses can be measured:

Pixels to be filtered	275,334
Pixels that can be MIPmapped	246,678
Pixels with Fast Footprint Filtering	28,656
Pixels with T between $\mathcal{M}$ and $2 * \mathcal{M}$	8,836

Figures 4.33 - 4.41 show the visual behavior of the algorithm compared to the other two. The images are all calculated with a screen resolution of 600×600 pixels. Setting $\mathcal{M} = 16$ results in an improvement compared to MIPmapping, but is still a little bit lower in quality than footprint assembly. $\mathcal{M} = 32$ reaches the quality of footprint assembly. This can be clearly seen at the checker board pattern, which has a resolution of 1024×1024 and is therefore a little bit blurry in the foreground due to interpolation, since its resolution is not sufficient in the foreground area.

In Figures 4.37 - 4.41, a scene with a map texture having 2048×2048 texels was used. The resolution is sufficient even for the foreground and it turns out, that for such a ”real-world” texture which is no artificial test pattern like the checker board, fast footprint filtering with $\mathcal{M} = 16$ is sufficient to get a comparable result as with footprint assembly. The difference to $\mathcal{M} = 32$ can only be seen in a difference image. Nevertheless, even with fixing $\mathcal{M}$ to eight texels a significant improvement can be achieved compared to MIPmapping in terms of the image being less blurred, see Figure 4.39. Eight texels is the amount of texture information which has to be fetched for the actual trilinear MIPmapping.

It is important to mention the smooth, not visible transition between the MIPmap levels without interpolating between MIPmap levels as it is done using trilinear MIPmapping. This is necessary to prevent aliasing during animation.

The next step will be to enhance the filtering quality further. It would make sense not to access only one MIPmap level, but to sample the footprint with a number of independent and smaller arrays on different levels of the MIPmap. This seems to be especially useful for pixels which have extended footprints. The loading of texels which are not needed but are contained in the loaded texel rectangle can be reduced even further, if T is better adapted to the shape of the footprint, see Figure 4.32. On the other hand, this will cost additional hardware and introduce latency, since the footprint has to be divided temporarily. Doing this is therefore a trade-off decision between:

- the cost of the fast footprint structure dictating how much weighting tables and in which size can be realized
- bandwidth of the texture memory
- additional costs and latency introduced by footprint subdivision.

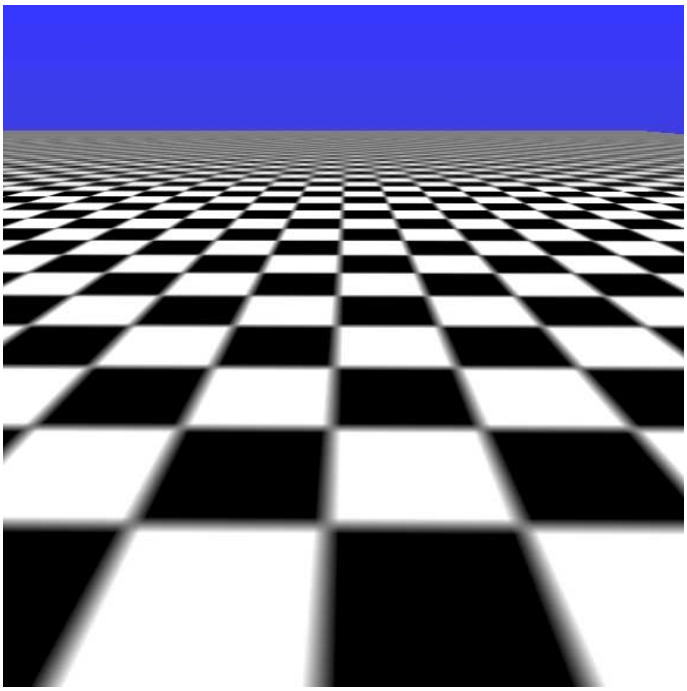


Figure 4.33: MIPmap filtering.

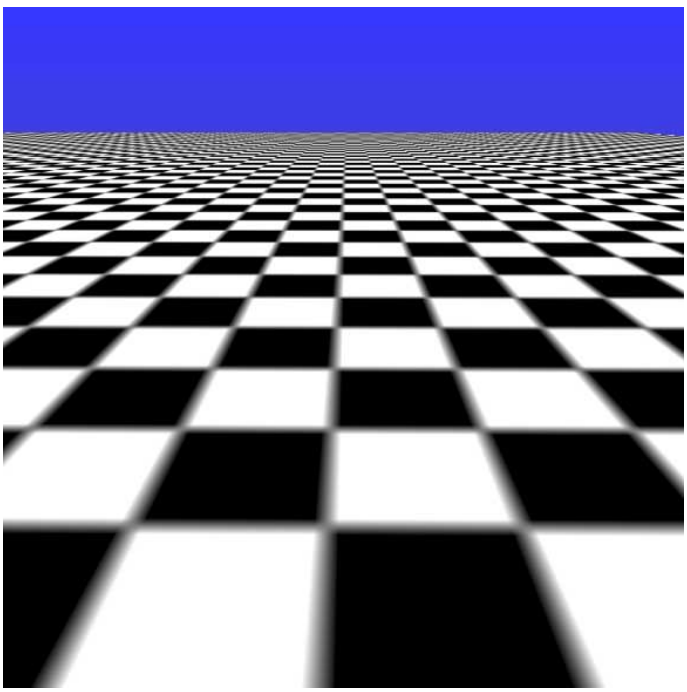


Figure 4.34: Footprint assembly.

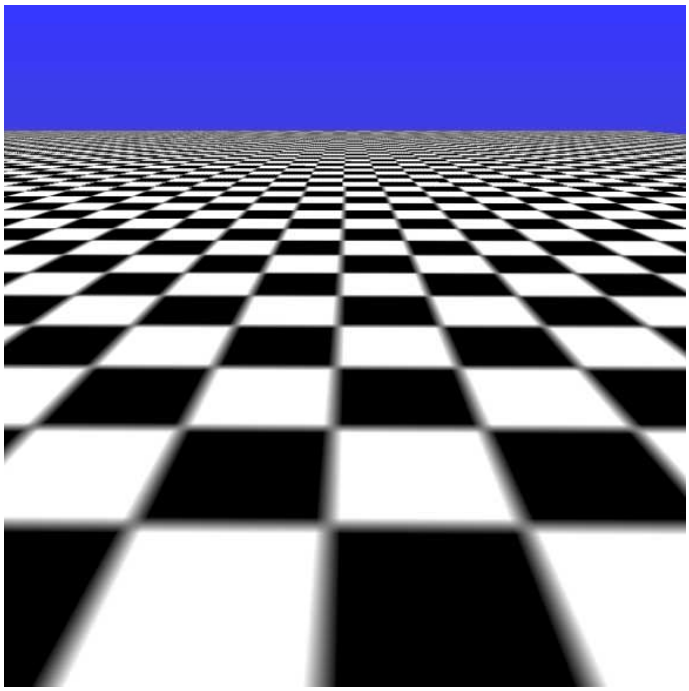


Figure 4.35: Fast footprint filtering using $\mathcal{M} = 16$.

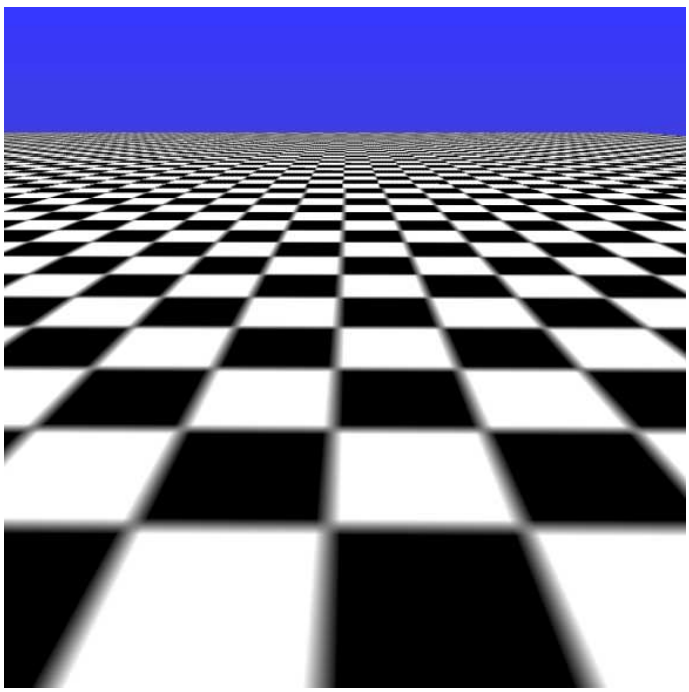


Figure 4.36: Fast footprint filtering using $\mathcal{M} = 32$.

Figure 4.37: MIPmap filtering.

Figure 4.38: Footprint assembly.

Figure 4.39: Fast footprint filtering using $\mathcal{M} = 8$.

Figure 4.40: Fast footprint filtering using $\mathcal{M} = 16$.

Figure 4.41: Fast footprint filtering using $\mathcal{M} = 32$.

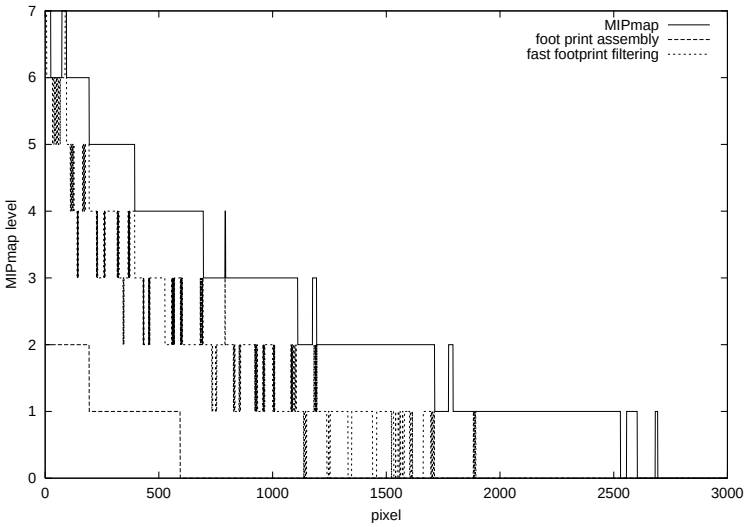


Figure 4.42: Selected MIPmap levels, $\mathcal{M} = 8$.

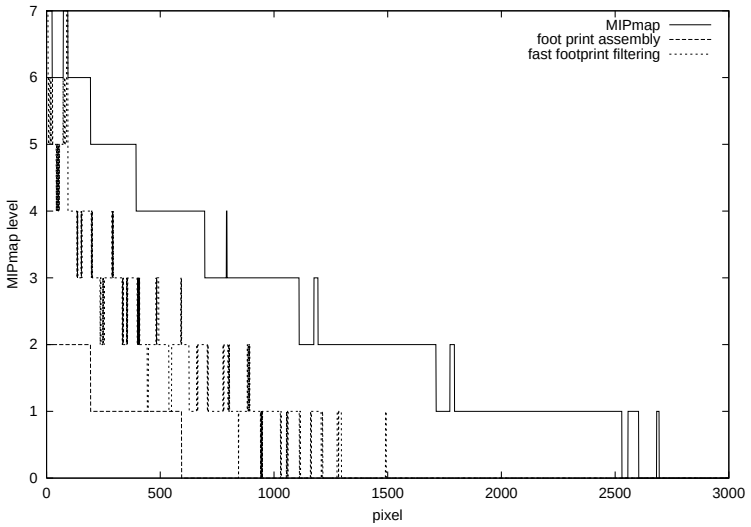


Figure 4.43: Selected MIPmap levels, $\mathcal{M} = 16$.

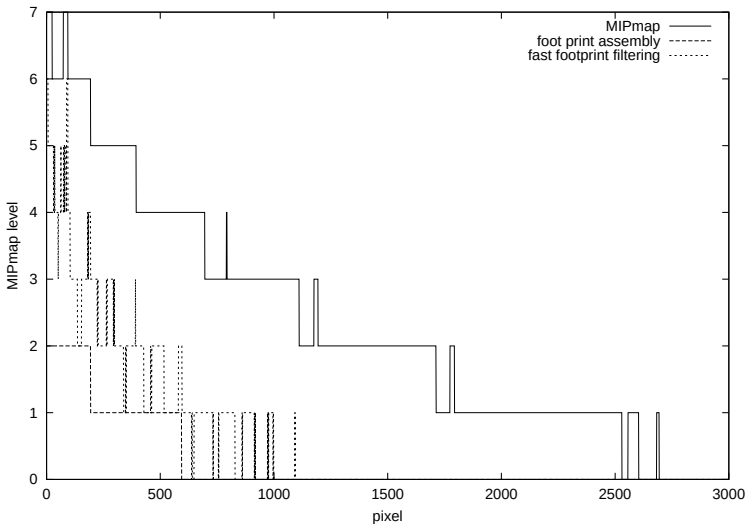


Figure 4.44: Selected MIPmap levels, $\mathcal{M} = 32$.

Chapter 5

FlyAway

5.1 Motivation

Using real geo-related topographic data-sets in a three-dimensional computer-graphics scene is usually known as *Terrain Modeling*. Most of the systems used for *Terrain Modeling* are not designed especially for this purpose, but are for example GIS systems or Multi-Media tools. They are often restricted in their ability to visualize very large data sets interactively but can be very useful for example for creating offline animations. Furthermore, it is difficult to extend them with algorithms like the multiresolution algorithms described in Chapter 2 and 4, since most of them cannot be extended at all or only with a kind of restricted scripting language as it is the case for example for *ArcView/ArcInfo*. Therefore, it was necessary to design an own *Terrain Modeling* tool, since *Terrain Modeling* applications demand interactivity to enable the user to move freely in a three-dimensional terrain environment.

5.2 Description of the System

One of the design goals of this tool was the wish of having huge virtual landscape models under interactive control even if they exceed the machine's graphics power. The only way to realize this with moderate machine power is to integrate *adaptivity* in the model repre-sentation (see the algorithms of Chapters 2 and 4). Perspective foreshortening ensures that the user cannot discriminate small details which are far away. This fact can be exploited and eliminates the rendering of this information with the help of *multiresolution* model representations that can be adapted to the properties of the camera viewing the scene like position, viewing direction or resolution of the image.

Having fast multiresolution algorithms and the idea of their use in *Terrain Modeling*, the next step was to define a project with the goal of bringing together the different types of data in an application. This application, called *FlyAway*, has to integrate the power of modern graphic systems with the possibility of extension to use new features like multiresolution techniques. Furthermore, *FlyAway* is an environment for testing new techniques and advanced rendering topics, since the power of these new approaches needs a verification with real data.

Some other approaches as in [13] use non-polygonal models for representing the

terrain surface. This requires a system able to render non-polygonal information in real-time. Since nearly the whole market of computer graphics hardware is dominated by polygonal render hardware, these approaches are not useful for the concept of *FlyAway* and the idea of having it run on a broad range of computer systems. Nevertheless, such approaches are interesting and very useful for the development of fast volume-rendering hardware, since in the near future, *hybrid rendering* will be the key to high rendering performance. With this, the graphic system can decide if a volumetric or polygonal representation of the data is suitable for the current rendering situation.

5.3 The Render Engine of *FlyAway*

One important idea behind *FlyAway* is the portability of the application to a broad range of computer systems enabled for 3D graphics. Therefore, an universal render engine was designed based on *OpenGL* and implemented in *C++*. As platforms for development are IRIX, Linux and WindowsNT systems in use. All system dependencies are concentrated in libraries that are available in both, the Unix and WindowsNT world.

Currently, the following libraries and formats are used:

1. *ImageMagick*, see [18]:
   This library is used for all image operations like scaling, filtering or converting. The libraries for the different image formats (for example, tiff, gif or jpeg) are used from the different operators of *ImageMagick*. Therefore, approximately 20 different image formats can be written and read for purposes like texturing. The algorithms of *ImageMagick* are implemented in *C* and are made accessible via wrapper classes that encapsulate the *ImageMagick* specific data structures.

2. *Glut, Tcl/Tk, Qt*, see [94], [88], [104]:
   These are toolkits for the system independent management of window functionality. Basic operations like the system dependent creation of physical drawing windows and the handling of user events are covered by these libraries. It is necessary to encapsulate such system specific functionality, since it differs very much between the various operating systems. Especially Windows and Unix are completely incompatible in their window handling, since Unix uses a client-server based window protocol (*X11*) whereas WindowsNT integrates this functionality into the operating system kernel.

   Glut is used for creating simple output windows which are able to handle user interaction. Glut is a public domain library that was initiated by Mark Kilgard and is nowadays a de facto standard in the OpenGL community to build small, portable applications. If used with X11, it supports the X11 input extensions which integrate additional input devices like spaceball, knob box or graphic tablet.

   Tcl/Tk and *Qt* serve as an abstraction layer for window processing with extended functionality. Tcl is a public domain scripting language that is the base of another operation system independent window toolkit which is called Tk. It is already widely used for building graphical user interfaces (GUI) in the Unix

world, but its Windows port has the disadvantage of exhibiting the native Windows 'look and feel' which is quite different from the one of the Unix world. Furthermore, Tcl/Tk has to be programmed in a C-style fashion and due to the scripting kernel, Tcl is not as fast as a compiled solution like Glut or Qt. On the other hand, scripting is useful for a run-time configuration of the application which is much more difficult when using a compiled and linked solution.

This is improved in Qt, a toolkit developed by the Linux community and nowadays used to implement for example the window manager *KDE* which is very widely used in the Linux domain. Qt is object oriented and written completely in C++. Therefore, no encapsulation is necessary as this is the case for Glut and Tcl/Tk and the 'look and feel' problem is solved much more convincingly than in Tcl/Tk.

3. *OpenInventor, VRML2.0*, see [87]:
   Currently, *OpenInventor* and *VRML2.0* are used as input format for three-dimensional data sets. The geological data sets of Figures 5.8 were produced with the GIS system *ArcInfo*, where experimental data was triangulated and exported as polygonal data in *VRML2.0* format. The houses and the freeway of the Figures 5.6 and 5.7 are constructed in the same way from real planning data. OpenInventor is a graphical data format defined by *SGI*(formerly *Silicon Graphics*). With OpenInventor, it is possible to define hierarchical, scene-graph based scene representations (see Chapter 3). Furthermore, behavioral elements like animation engines can be integrated into OpenInventor scenes which can be coupled to a part of the scene graph to produce for example animations like a spinning wheel. OpenInventor was the base for *VRML(virtual reality markup language)*, where special functionality for network transparent graphic scenes was added. Some implementations of OpenInventor exist, from which the version of *TGS* is most widely developed, since it is able to integrate OpenInventor based scene graphs as well as VRML scene graphs in its data structures at the same time.

4. *OpenGL*, see [85]:
   OpenGL is nowadays the most widely used low level graphics API for rendering polygonal data. It was defined by SGI and was first a proprietary language called *GL*(graphics language). Due to its success in the graphics community, it became a standard and its development is now controlled by the OpenGL architectural review board, where major suppliers and companies but also university members try to maintain the standard and to integrate useful and needed functionality. OpenGL implementations are available on nearly every computer system that is able to produce a raster graphic output. OpenGL is programmed in C and has no object oriented functionality. This was one of the main reasons for deciding to build an own rendering engine using object oriented principles and being implemented in C++.

It is one of the design goals of *FlyAway* to reuse existing software packages to build the desired functionality. All of the above components and libraries are either available in source code or they are included in the operating system like the *OpenGL* library. Therefore, *FlyAway* is easily portable to different platforms.

5.3.1 Class Hierarchy

The simplicity of the render engine and the object oriented implementation ensures on one hand an easy extension of the concept to integrate new features. On the other hand, a simple design and a class hierarchy with a moderate depth ensure speed. The render engine is used for the configuration of the *OpenGL* render context and the render widgets, since this is rather complicated and operating system dependent.

The OpenGL render context is a finite state machine, that accepts input in form of OpenGL commands, which can either manipulate the context internal state or trigger some render action to produce output. The render context itself is encapsulated in the *DispOpenGL* class, see Figure 5.1.

Render widgets are structures supplied by the operating system. They are areas on the screen, in which an OpenGL render context can put its output into. As mentioned above, the render engine encapsulates the render widgets by using libraries like *Glut* or *Tcl/Tk*.

The central managing class of the render engine is the *ViewerWindow* class. It controls the render context (*DispOpenGL*). Furthermore it has an instance of a *Viewer*-class (for example an *ExaminerViewer*), that controls the user interaction and is responsible for taking event input coming for example from a mouse and generating three-dimensional camera movements from this. The *Viewer*-class detects user interaction, evaluates it, and hands over the results to the render context via the *Camera*-class.

The render context holds a list of data objects, that will be rendered. They are called drawables and are derived from the abstract *Drawable* base class. Each drawable has to overload a virtual function *draw(void)*. In this function, the OpenGL commands necessary to draw this drawable are concentrated. Therefore, OpenGL commands can occur only at two places in the render engine:

1. The functions of the *DispOpenGL* render context contain the OpenGL calls that configure things that are common for all the drawables.

2. Each drawable issues its specific OpenGL calls. If it changes the common state in its *draw* function, these changes have to be recorded and have to be resolved before the *draw* function is finished.

The render engine supports the concept of *hybrid rendering*. This means, that different kinds of data, like voxels, polygons or images, can be used and rendered based on *OpenGL*. The integration of new data types is realized with the creation of a new *Drawable* class, e. g. a *MeshDrawable* to integrate triangle meshes. Other *Drawables* are available for scenegraph based *OpenInventor* or *VRML* data.

5.4 Further developments and future work

As mentioned earlier, *FlyAway* is an environment for many purposes. Therefore, it is consequently improved and enhanced to integrate new features and functionality. Some of the topics currently being worked on:

- Multi-Threading:
  For the usage of very large terrain models, it will be inevitable to use more than

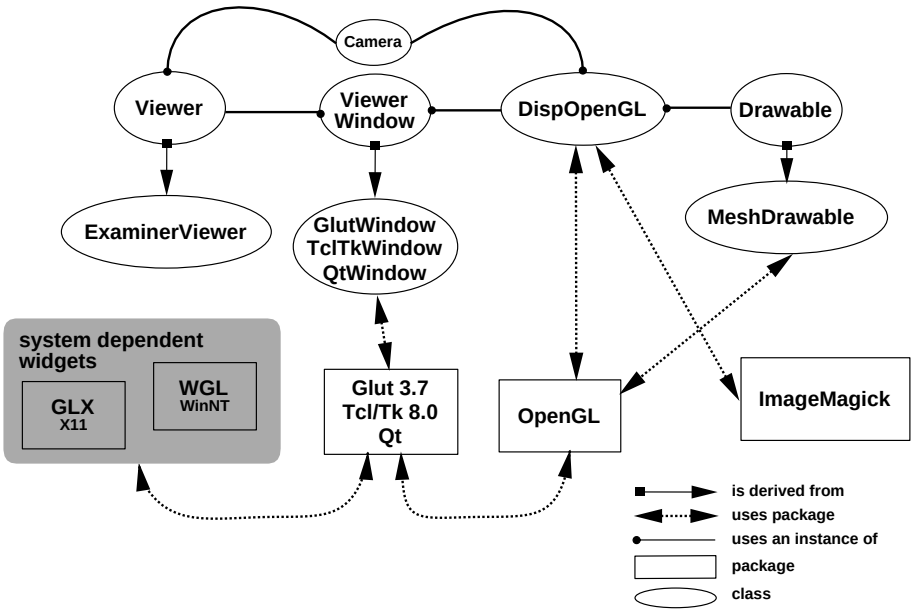


Figure 5.1: *Scheme of the class hierarchy of the render engine*

one processor in parallel. On one hand, independent parts of the landscape can then be updated independently. On the other hand, only multi-threading will ensure that the user-interaction and the update-process can be handled in parallel. It is intended to use some form of thread abstraction layer to guarantee portability. The prototype programmed with *ACE* (adaptive communication environment) looks very promising.

- VR Setup:
  Large scale projection devices are very suitable to make for a group of people an impressive interactive flight-through with *FlyAway* without having a VR-cave. (see Figures 5.2 and 5.3). Currently, a *Barco Baron Virtual Table* is used for this purpose. This device can be rotated by an angle and is therefore suitable to look like a bird on the landscape visible in a stereo projection. The table can also be used in a horizontal position for displaying the landscape model. This position will be suitable to interact with the terrain and things like houses placed on it. The next step will be the integration of additional input devices like grabbing pointers to enhance the possibilities of this environment. With stereoscopic viewing, several users will be able to interact and to place for example models in the virtual landscape. In this context, completely new user interfaces will be necessary to control this interaction.

- Hybrid Rendering:
  One of the main goals of *FlyAway* is to bring together different kinds of data to render them with the common basis of OpenGL. The use of voxel and light-field data are investigated in this context to enhance the optical richness of the scenery. Another topic is to integrate different geological, hydrological, and meteorological visualization techniques into the landscape models without losing

interactivity. To this context belongs also one of the current projects which deals with the visualization of some cave systems in the surroundings of Tübingen.

Figure 5.2: Virtual Table for stereoscopic model display.

Figure 5.3: Cooperative work of two people wearing stereo glasses.

5.5 Results Produced with FlyAway

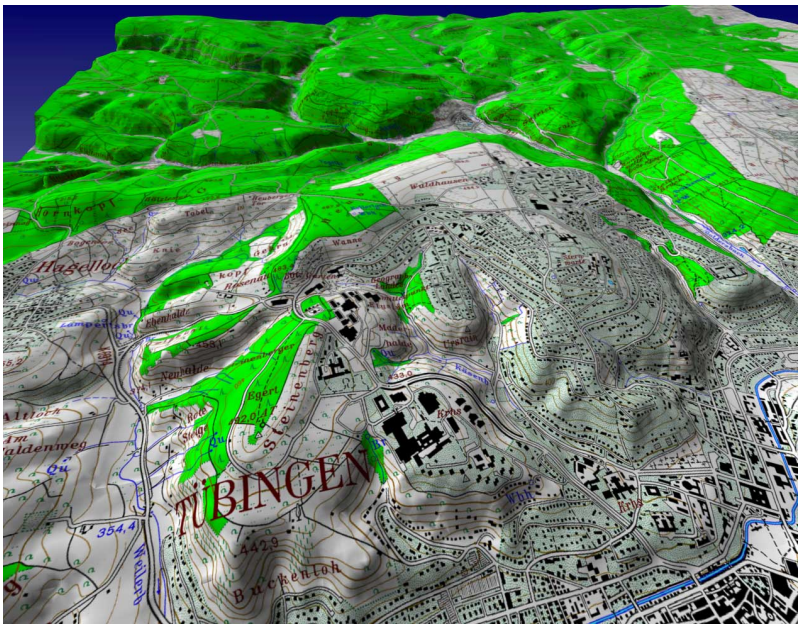


Figure 5.4: Cartographic map of Tübingen. In the center, one can see the buildings of the 'Morgenstelle' university campus.

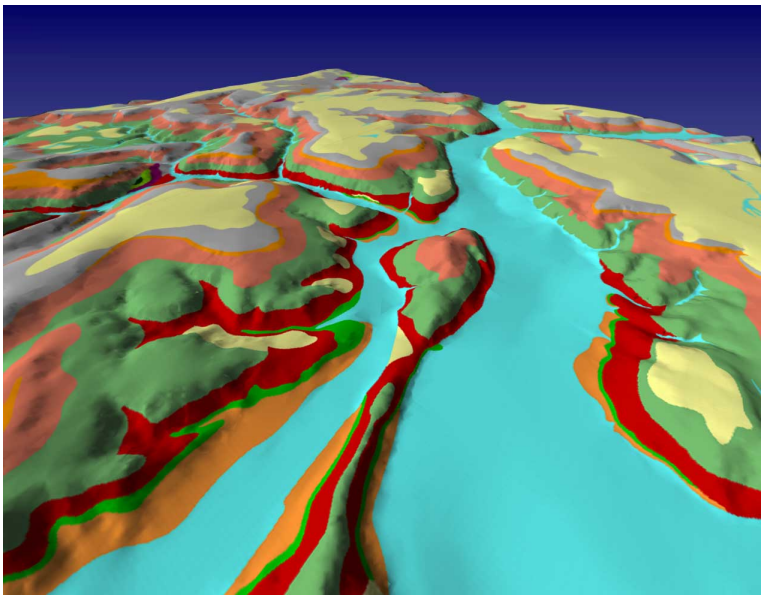


Figure 5.5: Map showing the geological layers of the Tübingen area. The viewer looks in north-east direction, the hill in the centre of the image is the well known Österberg. The color coding depicts the single geological layers.

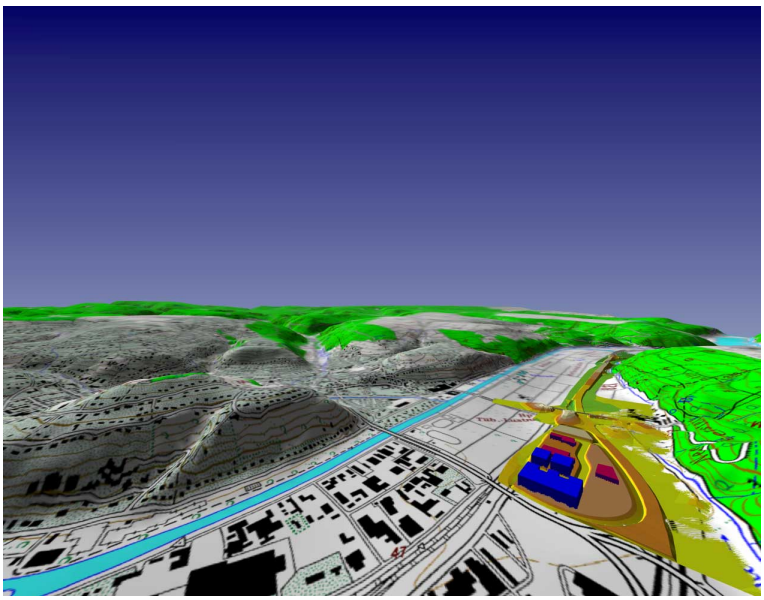


Figure 5.6: Visualization of a new industrial area near Tübingen. The planning data of the buildings was produced with a GIS system and imported into *FlyAway*. This visualization was made approximately 8 months before the building construction began and showed very nicely the visual impact of the planned industrial area on the surroundings.

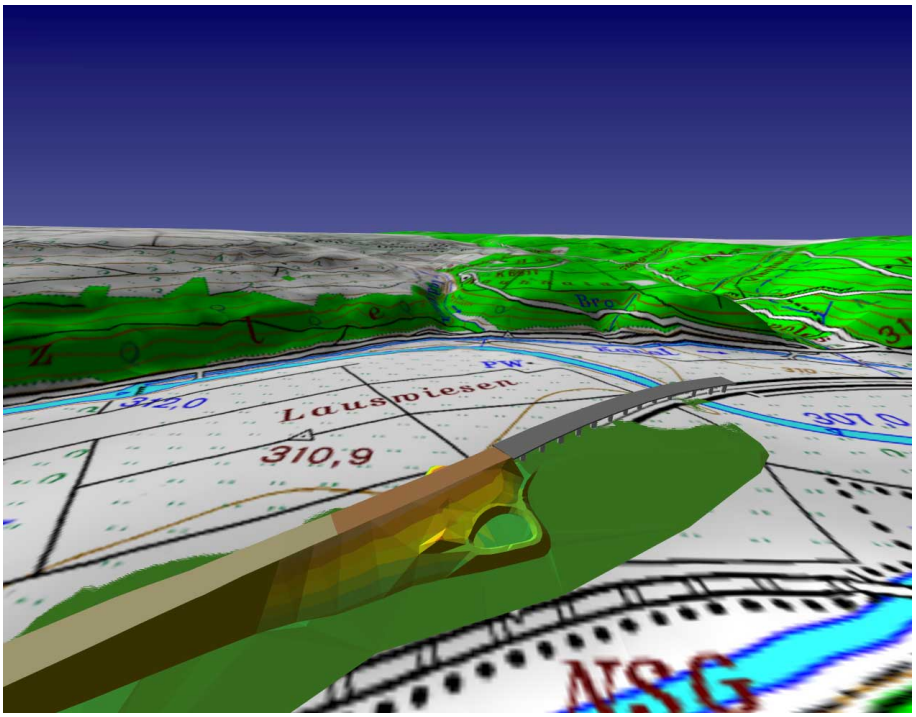


Figure 5.7: Visualization of the freeway constructed from planning data and placed at its correct geographic position.

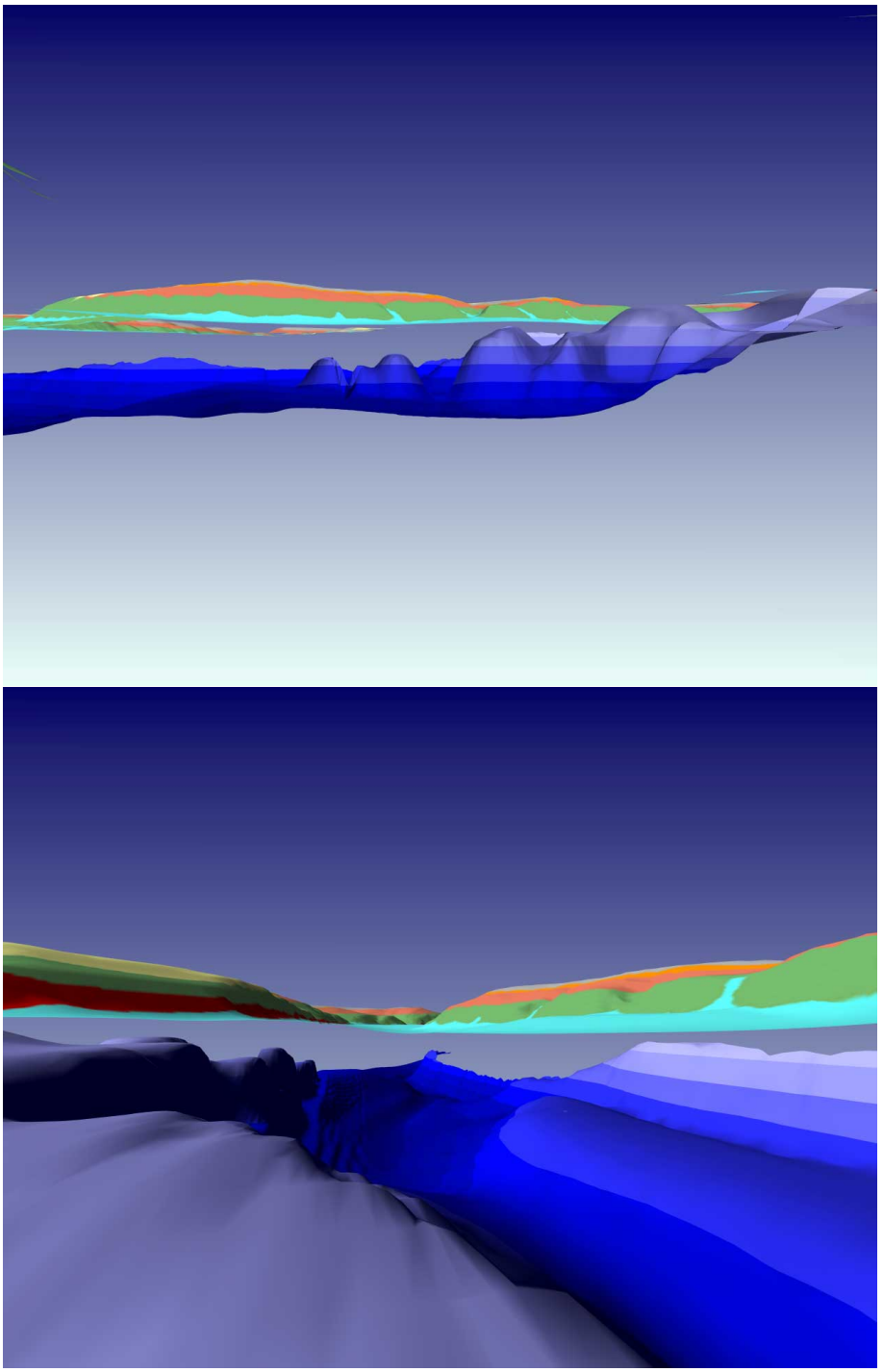


Figure 5.8: Visualization of a geological ground water model beneath the earth's surface. This structure is produced by a geological simulation process and is the isosurface of the flow of the groundwater in the valley of the river Neckar.

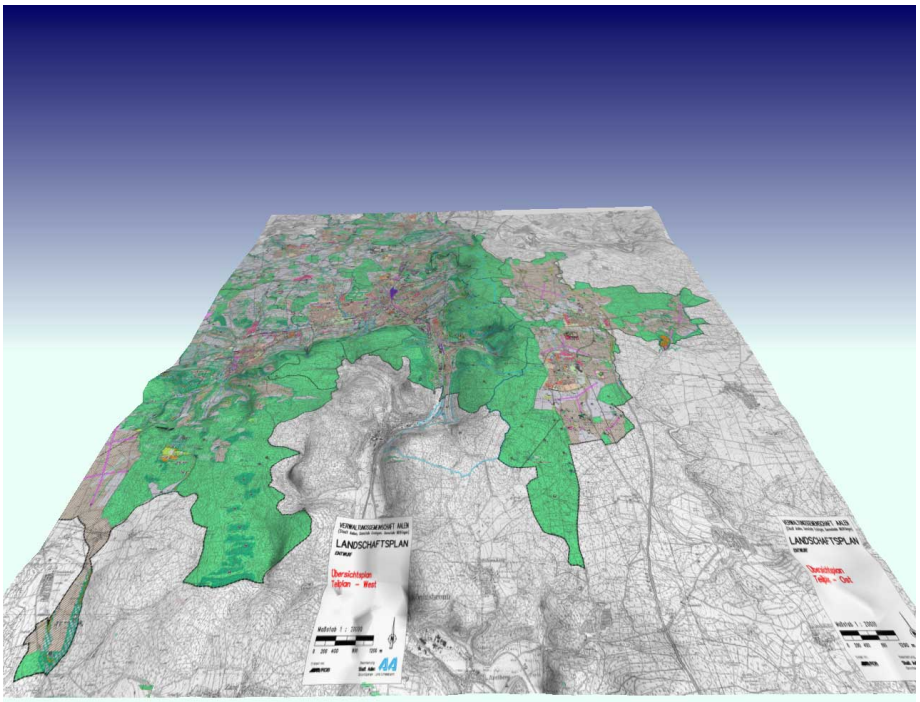


Figure 5.9: Open land management map covering 25km $\times$ 25km of landscape around the city of Aalen in southern Germany.

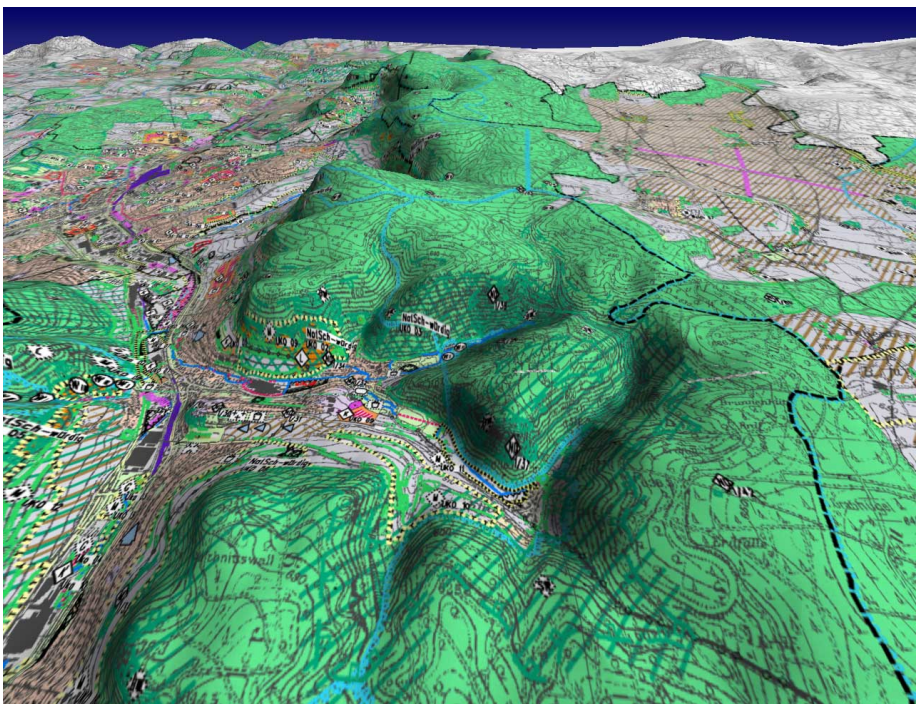


Figure 5.10: Detail of the open land management map.

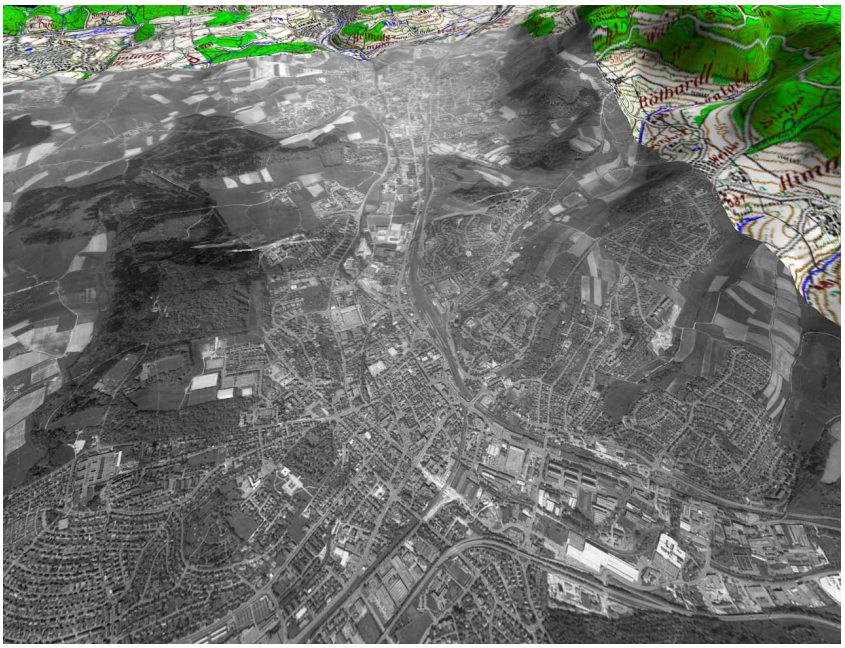


Figure 5.11: Topographic map with superimposed orthographic photograph (see Chapter 1). The orthographic photo has a much higher resolution than the map used as background texture.

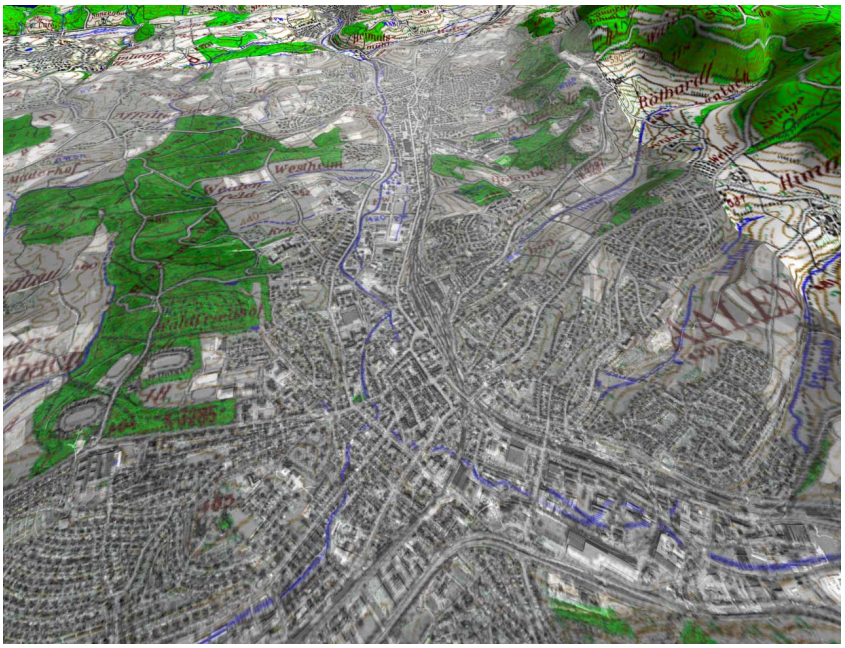


Figure 5.12: Topographic map with superimposed orthographic photograph. The orthographic photo is blended with the map below.

Chapter 6

Summary

The main goal of this work was to develop and to improve algorithms, that allow a better usage of geo-related data-sets in three-dimensional raster graphics.

Chapter one gives an short introduction to the topic and describes the data sources and kinds of data that are available. Furthermore, the historic development of survey and map making are described and the necessary definitions like coordinate systems are explained.

Major topics under investigation were the development of new camera adaptive data-structures for heightfield and raster image data. Therefore, the next chapter describe some new approaches for achieving this.

In the second chapter, the definition of a new multiresolution model for terrain data is given. This multiresolution model needs almost no additional memory overhead and it can be updated at interactive rates. It uses a viewer dependent screen space error to adopt the viewer independent multiresolution model the actual viewer's position. With this, very high reduction rates in polygon count can be achieved. Furthermore, this model can be used for other kinds of data, for example CAD models.

In the next chapter, approaches for the fast determination of visible and occluded scene objects are studied to accelerate the walkthrough of large and extended building and city models. A new occlusion culling algorithm was developed that uses only the framebuffer for culling and can therefore be used on a broad range of computer systems. This approach can be implemented in hardware which is also described in chapter three. In hardware, the algorithm utilizes the occlusion information already available during rasterization by collecting it with the help of special counting units.

In chapter four, some problems related to texture mapping are discussed. A new camera adaptive data-structure for textures, the *MIPmap grid*, was developed to store and to adopt huge texture images beyond the machines scope. This structure is able to respect the machines internal bandwidth and is therefore applicable to a broad range of computers. Furthermore, perspective foreshortening ensure here again that only a small part of the data has to be accessed. Next, a new method for texture cutting is described that allows the selection of an arbitrary quadrilateral of a texture for texturing by using graphics hardware. Some recent work was done in the field of texture filtering, where a new hardware approach was developed that gives much better filtering results than conventional *MIPmapping*.

Most of these algorithms were integrated into an interactive terrain visualization

system, called *FlyAway*. This system can visualize geo-related data-sets and is already used for planning purposes and the visualization of scientific data-sets. One of its main design goals was the easy portability of the system to various platforms. Therefore, it is currently ported to different Unix platforms and to the Windows world.

An application like *FlyAway* may give the public the opportunity to gain new insights into the complex behavior of our natural environment and to participate better in strategic decisions such as global landscape planning.

Bibliography

[1] J. Airey, J. Rohlf, and F. Brooks. Towards image realism with interactive update rates in complex virtual building environments. In *Symposium on Interactive 3D Graphics*, pages 41–50, 1990.

[2] H. C. Andrews and B. R. Hunt. Digital image restoration, 1977.

[3] Landesvermessungsamt Baden-Württemberg. *Digitales Höhenmodell des Landesvermessungsamts Baden-Württemberg, Verwertung genehmigt am 4.4.1997 unter Az.: 4.3/296*.

[4] D. Bartz, M. Meißner, and T. Hüttner. Extending Graphics Hardware for Occlusion Queries in OpenGL. In *Proc. of Eurographics/SIGGRAPH workshop on graphics hardware 1998*, pages 97–104, 1998.

[5] M. Bern, D. Eppstein, and J. Gilbert. Provably good mesh generation. In *Proc. 31st Annu. IEEE Sympos. Found. Comput. Sci.*, pages 231–241, 1990.

[6] Karolinska Institutets Bibliotek. Epitome. *http://www.mic.ki.se/vesalius.html*.

[7] R. Brechner. Interactive walkthroughs of large geometric databases. In *SIGGRAPH'96 course notes*, 1996.

[8] E. Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, University of Utah, 1974.

[9] P. Cignoni, E. Puppo, and R. Scopigno. Representation and visualization of terrain surfaces at variable resolution. In R. Scatenied, editor, *Scientific Visualization 95 (Int. Symp. Proc.)*, pages 50–68. World Scientific, 1995.

[10] D. Cohen-Or, , G. Fibich, D. Halperin, and E. Zadicario. Conservative Visibility and Strong Occlusion for Viewspace Partitioning of Densely Occluded Scenes. In *Proc. of Eurographics'98*, pages 243–253, 1998.

[11] D. Cohen-Or. Exact antialiasing of textured terrain models. *The Visual Computer*, 13(4):184–198, June 1997.

[12] D. Cohen-Or and Y. Levanoni. Temporal continuity of levels of detail in Delaunay triangulated terrain. In Roni Yagel and Gregory M. Nielson, editors, *Proceedings of the Conference on Visualization*, pages 37–42, October 27–November 1 1996.

[13] D. Cohen-Or, E. Rich, U. Lerner, and V. Shenkar. A real-time photo-realistic visual flythrough. *IEEE Transaction on Visualization and Computer Graphics*, 2(3):255–264, September 1996.

[14] D. Cohen-Or and E. Zadicario. On-line Conservative ϵ -Visibility Culling for Client/Server Walkthroughs. In *Late Breaking Hot Topics Proceedings, IEEE Visualization'97*, pages 37–40, 1997.

[15] D. Cohen-Or and E. Zadicario. Visibility Streaming for Network-based Walkthoughs. In *Proc. of Graphics Interface'98*, pages 1–7, 1998.

[16] S. Coorg and S. Teller. Temporally Coherent Conservative Visibility. In *Proc. of the 12th ACM Symposium on Computational Geometry*, 1996.

[17] S. Coorg and S. Teller. Real-Time Occlusion Culling for Models with Large Occluders. In *Proc. of the 1997 ACM Symposium on Interactive 3D Graphics*, pages 83–90, 1997.

[18] J. Cristy. Imakemagick homepage. *http://www.wizards.dupont.com/cristy/*.

[19] F. C. Crow. Summed-area tables for texture mapping. In Hank Christiansen, editor, *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 207–212, July 1984.

[20] M. de Berg and K. Dobrindt. On levels of detail in terrains. In *Proc. 11th Annual ACM Symp. on Computational Geometry*, Vancouver, B.C., June 1995.

[21] M. de Berg and K. T.G. Dobrindt. On levels of detail in terrains. Technical Report UU-CS-1995-12, Department of Computer Science, Utrecht University, April 1995.

[22] L. De Floriani. A pyramidal data structure for triangle-based surface description. *IEEE Computer Graphics and Applications*, 9(2):67–78, March 1989.

[23] L. de Floriani and E. Puppo. Hierarchical triangulation for multiresolution surface description. *ACM Transactions on Graphics*, 14(4):363–411, October 1995.

[24] M. J. DeHaemer, Jr. and M. J. Zyda. Simplification of objects rendered by polygonal approximations. *Computers and Graphics*, 15(2):175–184, 1991.

[25] B. Eberhardt, J. Hahn, and T. Hüttner. Raytracing and collision detection for cloth modelling. *accepted for publication in* Computer and Graphics, 1997.

[26] W. U. Eckart. Plastination. *http://www.uni-heidelberg.de/institute/fak5/igm/g47/eck_plas.htm*.

[27] J. Einighammer. Spatialize. *Studienarbeit am WSI/GRIS WS98/99*, 1999.

[28] John S. Falby, Michael J. Zyda, David R. Pratt, and Randy L. Mackey. NPSNET: Hierarchical data structures for real-time three-dimensional visual simulation. *Computers and Graphics*, 17(1):65–69, January–February 1993.

[29] R. L. Ferguson, R. Economy, W. A. Kelley, and P. P. Ramos. Continuous terrain level of detail for visual simulation. In *Proceedings of the 1990 Image V Conference*, pages 145–151. Image Society, Tempe, AZ, June 1990.

[30] L. De Floriani, P. Marzano, and E. Puppo. Hierarchical terrain models: survey and formalization. In *Proceedings SAC'94*, pages 323–327, Phoenix (AR), March 1994.

[31] J. Foley, A. Van Dam, S. Feiner, and J. Hughes. *Computer Graphics: Principles and Practice*. Addison Wesley, Reading, Mass., 2nd edition, 1996.

[32] D. A. Forsyth, C. Yang, and K. Teo. Efficient radiosity in dynamic environments. In *Fifth Eurographics Workshop on Rendering*, pages 313–323, Darmstadt, Germany, June 1994.

[33] R. J. Fowler and J. J. Little. Automatic extraction of irregular network digital terrain models. *Computer Graphics*, 13(3):199–207, August 1979.

[34] H. Fuchs, Z. Kedem, and B. Naylor. On Visible Surface Generation by a Priori Tree Structures. In *Proc. of ACM SIGGRAPH*, pages 124–133, 1980.

[35] T. A. Funkhouser and C. H. Sequin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. *Computer Graphics*, 27(Annual Conference Series):247–254, 1993.

[36] B. Garlick, D. Baum, and J. Winget. Interactive Viewing of Large Geometric Databases Using Multiprocessor Graphics Workstations. In *SIGGRAPH'90 course notes: Parallel Algorithms and Architectures for 3D Image Generation*, 1990.

[37] A. Glassner. Adaptive precision in texture mapping. In David C. Evans and Russell J. Athay, editors, *Computer Graphics (SIGGRAPH '86 Proceedings)*, volume 20, pages 297–306, August 1986.

[38] J.E. Goodman and J. O'Rourke. *Handbook of Discrete and Computational Geometry*. CRC Press, Boca Raton, 1997.

[39] Silicon Graphics. Performer whitepapers: uav.html. *http://www.sgi.com/software/performer/brew/uav.html*.

[40] N. Greene. *Hierarchical Rendering of Complex Environments*. PhD thesis, Computer and Information Science, University of California, Santa Cruz, 1995.

[41] N. Greene. Hierarchical Polygon Tiling with Coverage Masks. In *Proc. of ACM SIGGRAPH*, pages 65–74, 1996.

[42] N. Greene and M. Kass. Error-bounded Antialiased Rendering of Complex Environments. In *Proc. of ACM SIGGRAPH*, pages 59–66, 1994.

[43] N. Greene, M. Kass, and G. Miller. Hierarchical Z-Buffer Visibility. In *Proc. of ACM SIGGRAPH*, pages 231–238, 1993.

[44] P. Haeberli and M. Segal. Texture mapping as a fundamental drawing primitive. *Proceedings of Fourth Eurographics Workshop on Rendering*, pages 259–266, 1993.

[45] G. Hake. *Kartographie*. de Gruyter Lehrbuch, 1997.

[46] P. S. Heckbert. Texture mapping polygons in perspective. TM 13, NYIT Computer Graphics Lab, April 1983.

[47] P. S. Heckbert. Survey of texture mapping. *IEEE Computer Graphics and Applications*, pages 56–67, Nov. 1986.

[48] P. S. Heckbert and Michael Garland. Survey of polygonal surface simplification algorithms. Technical report, CS Dept., Carnegie Mellon U., to appear.

[49] Hewlett-Packard. Occlusion Test, Preliminary. http://www.opengl.org/Developers/Documentation/Version1.2/ HP-specs/occlusion_test.txt, 1997.

[50] L. Hong, S. Muraki, A. Kaufman, D. Bartz, and T. He. Virtual Voyage: Interactive Navigation in the Human Colon. In *Proc. of ACM SIGGRAPH*, pages 27–34, 1997.

[51] H. Hoppe. Progressive meshes. In *Computer Graphics Proceedings, Annual Conference Series, 1996 (ACM SIGGRAPH '96 Proceedings)*, pages 99–108, 1996.

[52] H. Hoppe. Smooth view-dependent level-of-detail control and its application to terrain rendering. In David Ebert, editor, *Proceedings of Vis'98*, pages 35–42, November 1998.

[53] T. Hudson, D. Manocha, J. Cohen, M. Lin, Kenneth E. Hoff, and H. Zhang. Accelerated Occlusion Culling Using Shadow Frusta. In *Proc. of ACM Symposium on Computational Geometry*, 1997.

[54] T. Hüttner. High resolution textures. *Visualization'98 - Late Breaking Hot Topics Papers*, pages 13–17, November 1998.

[55] T. Hüttner. Viewing landscape in three dimensions. In *EG-Workshop on Visualization in Scientific Computing 98, Proceedings*, 1998.

[56] T. Hüttner. Fast footprint mipmapping. In *to appear in Proc. of Eurographics/SIGGRAPH workshop on graphics hardware 1999*, 1999.

[57] T. Hüttner, M. Meißner, and D. Bartz. Opengl-assisted visibility queries of large polygonal models. Technical Report WSI-98-6, ISSN 0946-3852, Dept. of Computer Science (WSI), University of Tübingen, 1998.

[58] H. Igehy, M. Eldridge, and K. Proudfoot. Prefetching in a texture cache architecture. In *Eurographics/SIGGRAPH Hardware Workshop '98 Proceedings)*, 1998.

[59] Silicon Graphics Inc. Optimizer Manual. In *Manuals*, 1997.

[60] W. Straßer J. Encarnação and R. Klein. *Graphische Datenverarbeitung I*. R. Oldenbourg Verlag, 1996.

[61] H. Kahmen. *Vermessungskunde*. de Gruyter Lehrbuch, Berlin, 1997.

[62] T. L. Kay and J. T. Kajiya. Ray Tracing Complex Scenes. In *Proc. of ACM SIGGRAPH*, pages 269–78, 1986.

[63] M. Kilgard. Realizing opengl: Two implementations of one architecture. In *1997 EG/SIGGRAPH Workshop on Graphics Hardware*, 1997.

[64] R. Klein. Multiresolution representations for surfaces meshes based on the vertex decimation method. *Computers & Graphics*, 22(1), January 1998.

[65] R. Klein, D. Cohen-Or, and T. Hüttner. Incremental view-dependent multilevel triangulation of terrain. In *Proceedings of the Fifth Pacific Conference on Computer Graphics and Applications, Seoul, Korea*, pages 127–136. IEEE Computer Society, Los Alamitos, California, October 1997.

[66] R. Klein, D. Cohen-Or, and T. Hüttner. Incremental view-dependent multiresolution triangulation of terrain. *The Journal of Visualization and Computer Animation*, 9:129–143, August 1998.

[67] R. Klein and T. Hüttner. Generation of multiresolution models from CAD - data. In W. Strasser, R. Klein, and R. Rau, editors, *Theory and Practice of Geometric Modeling 96, (Blaubeuren II)*. Springer, October 1996.

[68] R. Klein and T. Hüttner. Simple camera dependent approximation of terrain surfaces for fast visualization and animation. *Visualization'96 - Late Breaking Hot Topics Papers*, pages 20–25, November 1996.

[69] R. Klein, T. Hüttner, and J. Krämer. Viewing parameter dependent approximation of nurbs-models for fast visualization and animation using a discrete multiresolution representation. In *Workshop 3D Bildanalyse, Erlangen-Nuremberg*. infix, November 1996.

[70] R. Klein, R. Sonntag, and T.Hüttner. Datenreduktion von Radiositynetzen zum Einsatz globaler Beleuchtung in Anwendungen der virtuellen Realität. In *Workshop MVD'95, Bad Honnef / Bonn*. infix, November 1995.

[71] Reinhard Klein and Stefan Gumhold. Data compression of multiresolution surfaces. Technical Report WSI-1997-17, WSI/GRIS, 1997.

[72] B. Levy and J.L. Mallet. Non-distorted texture mapping for sheared triangulated meshes. In *Computer Graphics (SIGGRAPH '98 Proceedings)*, July 1998.

[73] D. Lischinski. Incremental Delaunay triangulations. In P. S. Heckbert, editor, *Graphics Gems IV*, pages 47–59. Academic Press, 1994.

[74] G. Lörcher and T. Hüttner. *FlyAway* homepage. *http://www.gris.uni-tuebingen.de/~flyaway*.

[75] W. Lorensen and H. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. In *Proc. of ACM SIGGRAPH*, pages 163–169, 1987.

[76] D. Luebke and C. Georges. Portals and Mirrors: Simple, Fast Evaluation of Potentially Visible Sets. In *Proc. of ACM Interactive 3D Graphics Conference*, 1995.

[77] P. W. C. Maciel and P. Shirley. Visual navigation of large environments using textured clusters. In Pat Hanrahan and Jim Winget, editors, *1995 Symposium on Interactive 3D Graphics*, pages 95–102. ACM SIGGRAPH, April 1995. ISBN 0-89791-736-7.

[78] M. Meißner, T. Hüttner, W. Blochinger, and A. Weber. Parallel direct volume rendering on pc networks. In *PDPTA'98 Proceedings*, 1998.

[79] J. Montrym, D. Baum, D. Dignam, and C. Migdal. Infinitereality: A real-time graphics system. In *Proc. of ACM SIGGRAPH*, pages 293–302, 1997.

[80] John S. Montrym, Daniel R. Baum, David L. Dignam, and Christopher J. Migdal. Inifinite reality: A real-time graphics system. In *Computer Graphics (SIGGRAPH '97 Proceedings)*, 1997.

[81] C. Mueller. Architectures of Image Generators for Flight Simulators. Technical Report TR95-015, Department of Computer Science, University of North Carolina, Chapel Hill, 1995.

[82] G. Müller and D. Fellner. Hybrid Scene Structuring with Application to Ray Tracing. In *Proc. of ICVC'99 (to appear)*, 1999.

[83] NASA. Landsat homepage. *http://geo.arc.nasa.gov/sge/landsat/lpsum.html*.

[84] B. Naylor. Partitioning Tree Image Representation and Generation From 3D Geometric Models. In *Proc. of Graphics Interface'92*, pages 201–212, 1992.

[85] J. Neider, T. Davis, and M. Woo. *OpenGL Programming Guide*. Addison-Wesley, 1993.

[86] E. Ofek, E. Shilat, A. Rappoport, and M. Werman. Multiresolution textures from image sequences. *IEEE Computer Graphics and Applications*, pages 18–29, March 1997.

[87] Open Inventor Architecture Group. *Open Inventor C++ Reference Manual: The Official Reference Document for Open Systems*. Addison-Wesley, Reading, MA, USA, 1994.

[88] Santa Cruz Operation. Tcl/tk homepage. *http://www.sco.com/Technology/tcl/Tcl.html*.

[89] R. Pajarola. Large scale terrain visualization using the restricted quadtree triangulation. In David Ebert, editor, *Proceedings of Vis'98*, pages 19–26, November 1998.

[90] M. Pesce. Background on the virtual reality modeling language. http://www.sgi.com/Products/WebFORCE/WebSpace/VRMLBackgrounder.html, March 1995.

[91] J. Pineda. A parallel algorithm for polygon rasterization. In John Dill, editor, *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 17–20, August 1988.

[92] John P.Snyder. *Mapping Projections*. ESRI - ISBN 1-879102-28-5, 1995.

[93] B. Rabinovich and C. Gotsman. Visualization of large terrains in resource-limited computing environments. In *IEEE Visualization '97*, pages 95–102, October 1997.

[94] N. Robins. Glut homepage. *http://www.cs.utah.edu/~narobins/glut.html*.

[95] J. Rohlf and J. Helman. IRIS Performer: A high performance multiprocessing toolkit for real-time 3D graphics. *Computer Graphics*, 28(Annual Conference Series):381–394, 1994.

[96] L. Scarlatos and T. Pavlidis. Hierarchical triangulation using cartographic coherence. *Computer Vision, Graphics, and Image Processing. Graphical Models and Image Processing*, 54(2):147–161, March 1992.

[97] A. Schilling, G. Knittel, and W. Straßer. Texram: A smart memory for texturing. *IEEE Computer Graphics & Applications*, 16(3):32–41, May 1996.

[98] N. Scott, D. Olsen, and E. Gannett. An Overview of the VISUALIZE fx Graphics Accelerator Hardware. *The Hewlett-Packard Journal*, (May):28–34, 1998.

[99] J. Snyder and J. Lengyel. Visibility Sorting and Compositing Without Splitting for Image Layer Decomposition. In *Proc. of ACM SIGGRAPH*, pages 231–242, 1998.

[100] W. Straßer. *Schnelle Kurven- und Flächendarstellung auf graphischen Sichtgeräten*. PhD thesis, Technische Universität Berlin, 1974.

[101] O. Sudarsky and C. Gotsman. Output-Sensitive Visibility Algorithms for Dynamic Scenes with Applications to Virtual Reality. In *Proc. of Eurographics'96 conference*, pages 249–258, 1996.

[102] U.S. Geological Survey. *Data Users Guide*. 1993.

[103] C. C. Tanner, C. J. Migdal, and M. T. Jones. The clipmap: A virtual mipmap. In *Computer Graphics (SIGGRAPH '98 Proceedings)*, 1998.

[104] Troll Tech. Qt homepage. *http://www.troll.no/*.

[105] S. Teller and C.H. Sequin. Visibility Pre-processing for Interactive Walkthroughs. In *Proc. of ACM SIGGRAPH*, pages 61–69, 1991.

[106] J. Torborg and J. Kajiya. Talisman: Commodity Real-time 3D graphics for the PC. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 353–364. ACM SIGGRAPH, Addison Wesley, August 1996. held in New Orleans, Louisiana, 04-09 August 1996.

[107] B. Von Herzen and A. H. Barr. Accurate triangulations of deformed, intersecting surfaces. In Maureen C. Stone, editor, *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21, pages 103–110, July 1987.

[108] E. W. Weisstein. Vesalius. *http://www.astro.virginia.edu/ eww6n/bios/Vesalius.html*.

[109] L. Williams. Pyramidal parametrics. In *Computer Graphics (SIGGRAPH '83 Proceedings)*, pages 1–11, July 1983.

[110] L. R. Willis. Who says you can't teach an old lod new tricks. In *Proceedings of IMAGE 1998*, August 1998.

[111] M. Woo, J. Neider, and T. Davis. *OpenGL Programming Guide*. Addison-Wesley, 2nd edition, 1997.

[112] J. C. Xia and A. Varshney. Dynamic view-dependent simplification for polygonal models. In Holly Rushmeier, editor, *Computer Graphics (SIGGRAPH '96 Proceedings)*, volume 30(4), August 1996.

[113] R. Yagel and W. Ray. Visibility computations for efficient walkthrough of complex environments. *PRESENCE*, pages 1–16, 1996.

[114] H. Zhang, D. Manocha, T. Hudson, and Kenneth E. Hoff. Visibility Culling Using Hierarchical Occlusion Maps. In *Proc. of ACM SIGGRAPH*, pages 77–88, 1997.