

REAL-TIME WALKTHROUGH OF COMPLEX ENVIRONMENTS

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER
ENGINEERING AND INFORMATION SCIENCE
AND THE INSTITUTE OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By

Alper Selçuk

September, 1997

T
385
545
1997

REAL-TIME WALKTHROUGH OF COMPLEX ENVIRONMENTS

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER
ENGINEERING AND INFORMATION SCIENCE
AND THE INSTITUTE OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

Alper Selçuk .

By Alper Selçuk

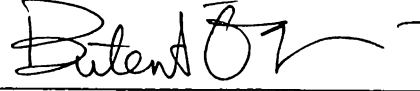
Alper Selçuk

September, 1997

T
385
.S45
1997

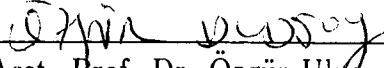
B C38447

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.



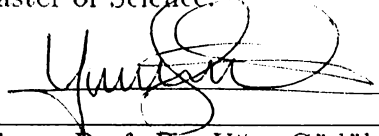
Prof. Dr. Bülent Özgüç (Principal Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.



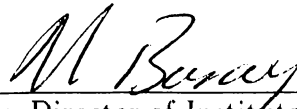
Asst. Prof. Dr. Özgür Ulusoy

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.



Asst. Prof. Dr. Uğur Güdükbay

Approved for the Institute of Engineering and Science:



Prof. Dr. Mehmet Baray, Director of Institute of Engineering and Science

ABSTRACT

REAL-TIME WALKTHROUGH OF COMPLEX ENVIRONMENTS

Alper Selçuk

M.S. in Computer Engineering and Information Science

Supervisor: Prof. Dr. Bülent Özgüç

September, 1997

One of the biggest problems in computer graphics is displaying huge geometric models in interactive frame-rates. Such models exceed limits of best graphics workstations. A lot of work has been done for achieving the required frame-rates in architecture, simulation, computer-aided design and entertainment applications. In this thesis, a survey of methods that enable walkthrough of huge geometric models is done and a system for walkthrough is developed. The system uses hierarchical triangulated geometric models as input. In pre-processing phase, multiresolution models of objects in the scene are created using polygonal simplification techniques. During walkthrough, fast frustum culling based on bounding boxes is performed which eliminates branches of hierarchy that are not visible to camera efficiently. Appropriate level of detail of objects are selected and displayed depending on the distance of the objects to the camera. For far nodes of hierarchy, geometric data in lower levels is ignored and textured bounding box is displayed. The system achieves interactive frame rates for moderate level models, however it is far from being interactive with huge models.

Key words: level-of-detail, visibility culling, geometric simplification, real-time rendering.

ÖZET

KARMAŞIK GEOMETRİK ORTAMLARDA GERÇEK ZAMANDA GEZİNTİ

Alper Selçuk

Bilgisayar ve Enformatik Mühendisliği, Yüksek Lisans

Tez Yöneticisi: Prof. Dr. Bülent Özgüç

Eylül, 1997

Bilgisayar Grafiği alanındaki en büyük problemlerden birisi dev boyuttaki karmaşık geometrik modelleri etkileşimli ve hızlı olarak göstermektir. Bu boyuttaki modeller en iyi grafik iş istasyonlarının bile kapasitesini aşmaktadır. Mimarlık, simulasyon, bilgisayar destekli tasarım ve eğlence alanlarındaki uygulamalarda, gereken etkileşimi sağlayabilmek için pek çok çalışma yapılmıştır. Bu tezde, dev geometrik modellerde gerçek zamanlı gezintiyi sağlayan yöntemler incelenmiş ve bu amaca hizmet eden bir sistem geliştirilmiştir. Sistem üçgenlerden oluşan, hiyerarşik geometrik modeller üzerinde çalışmaktadır. Önce ortamdaki nesnelerin basitleştirilmiş halleri, çokgen sadeleştirme teknikleri kullanılarak oluşturulmakta: gezinti sırasında, görüş alanı dışında kalan nesneler, kapsama kutuları kullanılarak hızlı bir biçimde ayıklanmaktadır. Nesnelerin orjinal ya da basitleştirilmiş halleri, nesnenin kameraya olan uzaklığına göre gösterilmektedir. Çok uzaktaki nesneler için sadece doku kaplanmış kapsama kutusu gösterilmektedir. Sistem orta karmaşıklıkta modeller için gereken etkileşimli hızları sağlamakla birlikte, dev boyuttaki modeller için yavaş kalmaktadır.

Anahtar kelimeler: basamaklı gösterim, görünürlük testi, geometrik basitleştirme, gerçek zamanlı görüntüleme.

ACKNOWLEDGMENTS

I am very grateful to my supervisor, Prof. Dr. Bülent Özgüç for his invaluable guidance and motivating support during this study. I would also like to thank Asst. Prof. Özgür Ulusoy and Asst. Prof. Uğur Güdükbay for their remarks and comments on the thesis.

I would like to thank everybody who has in some way contributed to this study by lending me moral, technical and intellectual support, including Ozan Özhan, Barış Yaman, Esra Taner and Serdar Yağcı.

I would also like to thank SEBIT (Sevgi Education and Information Technologies) and Ahmet Eti for providing me SGI Onyx and helping me prepare my demo video. Fahri Tuncer for his invaluable technical support on SGI Onyx and 3D animators for their help for using Alias/Wavefront deserve special thanks.

I would like to thank my parents and my sister for their moral support and motivation.

Finally, I would like to thank to my wife Aslı. I cannot forget her valuable moral support and hope-giving. I dedicate this thesis to Aslı.

To my wife, Ashi

Contents

1	Introduction	1
2	Human Visual System	4
2.1	Visual Acuity	5
2.2	Field of View	5
2.3	Latency	6
2.4	High Frame Rate	6
2.5	Constant Frame Rate	7
2.6	Temporal Resolution	7
3	Structure of Graphics Pipelines	9
4	Methods for Real-Time Walkthrough	13
4.1	Reducing Geometric Complexity	14
4.1.1	Texture Mapping	14
4.2	Polygonal Simplification	16
4.2.1	Basic Methods	16

4.2.2	Topology Preserving Algorithms	18
4.2.3	Topology Simplifying Algorithms	21
4.3	Geometric Level of Detail	22
4.4	Optimizing Run-Time Rendering	26
4.4.1	Visibility and Occlusion Culling	26
4.4.2	Level of Detail Switching	28
4.4.3	Mode Sorting	28
4.5	Optimizing Graphics Databases	29
4.5.1	Triangle Meshes	29
4.5.2	Smart Caching	30
4.6	Handling Very Large Databases	30
4.6.1	Paging Geometry from Disk	31
4.6.2	Paging Texture from Memory	31
5	The Walkthrough System	33
5.1	Input Model	34
5.2	The Preprocessing Phase	35
5.2.1	Building the Hierarchy	37
5.2.2	Simplification	38
5.3	The Walkthrough Phase	49
5.3.1	Camera Model	49
5.3.2	Frustum Culling	50

5.3.3	Managing Level of Detail	51
6	Results	54
6.1	Geometric Simplification	54
6.2	Level of Detail Management	60
6.3	The Walkthrough System	61
7	Conclusions and Future Research Areas	72
7.1	Conclusions	72
7.2	Future Research Areas	74
A	Input Model	75
B	OpenGL	79
B.1	Transformation Calculations	80
B.2	Triangle Fans and Triangle Strips	81

List of Figures

2.1	Repeating Objects	7
3.1	Principal Graphics Pipeline	9
3.2	Reality Engine Pipeline	11
4.1	Earth-Texture Mapped Sphere	15
4.2	Level of Detail	22
4.3	Hierarchical Organization of a Scene	25
4.4	Triangulated Polygon	29
5.1	Block Diagram of the Walkthrough System	33
5.2	Hierarchical Model of a Cube	34
5.3	Data Required for a Triangle	35
5.4	Hierarchical Model of a Column	36
5.5	Hierarchy List of the Column	37
5.6	Finding Textures of an Intermediate Node	39
5.7	Geometric Optimization Algorithm	40
5.8	Original Surface	41

5.9	Possible Groups of the Original Surface	42
5.10	Ignored Neighboring Triangles	43
5.11	Edge List for a Simple Polygon	44
5.12	Boundary Polygons of Coplanar Groups	46
5.13	Graham Scan Algorithm	48
5.14	Sample Graham Scan Execution	48
5.15	Triangulation Algorithm	49
5.16	Difference Between Original and Triangulated Polygon	49
5.17	The Virtual Camera Model	50
5.18	Level of Detail Management	52
6.1	Polygonal Simplification of a Square	55
6.2	Polygonal Simplification of a Polygon	56
6.3	Polygonal Simplification of a Surface	57
6.4	Polygonal Simplification of a Sphere	58
6.5	Polygonal Simplification of a Column	59
6.6	Level of Detail Management-Original Scene	60
6.7	Level of Detail Management-1	61
6.8	Level of Detail Management-2	62
6.9	Level of Detail Management-3	62
6.10	Level of Detail Management-4	63
6.11	Level of Detail Management-5	63

6.12 Level of Detail Management-6	64
6.13 Level of Detail Management-7	64
6.14 The Parthenon	66
6.15 The Test Scene	66
6.16 The Walkthrough-1	67
6.17 The Walkthrough-2	68
6.18 The Walkthrough-3	69
6.19 The Walkthrough-4	70
A.1 Representation of an Object in OSTF.	76
A.2 The Cube	76
A.3 Representation of the Cube	78
B.1 Stages of Vertex Transformation in OpenGL	80
B.2 Triangle Commands in OpenGL	81

List of Tables

4.1	Rendering Cost	13
6.1	Polygonal Simplification of a Polygon	55
6.2	Polygonal Simplification of a Surface	57
6.3	Polygonal Simplification of a Sphere	58
6.4	Polygonal Simplification of a Column	58
6.5	Centers of Objects in Scene	60
6.6	Polygonal Simplification of a Surface	71

Chapter 1

Introduction

In recent years, the use of computers in design has become extremely important. As the price of computers and peripherals decrease, more and more companies have started to buy powerful workstations for their design process. This trend led many software companies to develop serious and complicated programs for computer-aided design. There are now many CAD programs for mechanical and architectural design. These programs are so good and reliable that Boeing has developed its new aircraft 777 entirely on computer using CATIA CAD software. All 3 million parts of the aircraft were modeled using CATIA. The entire model requires 20 GB of storage and consists of 500 million polygons [1].

Although CAD software helps engineers a lot in designing and viewing individual components, in order to verify the correctness of design, engineers need to view combined shots of their design. Only combined shots are also sometimes not enough, walkthrough of the entire model is required. Considering the size of the model of Boeing 777, only one single camera shot can require days of time to render even if state-of-the-art computers are used.

Architects today also use computers to design and view buildings. The size of such architectural models can also be huge. Customers may want to view the building on computer. Just showing rendered pictures or a film prepared moving a virtual camera inside the building model on a predetermined path

sometimes may not satisfy customer. Customer may want to walk inside the model interactively. This case is different from the Boeing example. Accuracy and correctness are the most vital properties for Boeing. However, for the architectural case, the image quality of the walkthrough is the most important property. By image quality, the quality of the texture, positioning of light sources and the rendering quality is meant. Real-time restrictions can be slightly loosened for the sake of quality. Considering the size of the architectural model, the textures used for walls, floor and furniture and the number of light sources in a building, the power of today's computers are again exceeded.

Flight simulators are used for the training of pilots and astronauts. Flight simulators must show the pilot exactly what would be seen from the cockpit of the airplane. Terrain, sky, sea and many more things should be rendered at the speed of an airplane. The most important property is the realism of the frames generated by the rendering engine. By realism, quality of the frames and simulation of the speed of the airplane is meant. Considering the size and complexity of the training area modeled for use with the flight simulator and the real-time requirements, performance of most powerful computers is brought to a crawl.

The problem for all the systems explained above is how to display more polygons than the computer can render in one second. For explaining the problem further, suppose we have a polygon database of more than one million polygons and for the application displaying ten frames per second is enough. In that case our computer system must be able to display ten million polygons for just one second. Current high-end computer graphics systems can render approximately one million polygons per second. To make things worse, for a realistic looking sequence thirty frames per second is required, which makes for our example polygon database thirty million polygons per second. For virtual reality (VR) applications two displays exist and that means sixty frames per seconds, in other words sixty million polygons per second. In the extreme, a VR display of the entire Boeing 777 would require a display rate of over 30 billion polygons per second.

It is trivial that current computer graphics systems cannot meet required graphics throughput for the above examples. The situation will probably not

change in the future because as the graphics systems evolve and get more powerful, the size of models grow larger and larger. In 1986, 8000 polygons required 3-5 seconds to be rendered on the Vector-General 3300 [2]. Since 1986 computers have evolved and grown into polygon monsters, however models have grown into polygon mountains.

In this thesis, a survey of methods that enable walkthrough of complex environments is done and a system for walkthrough is developed. The system uses hierarchical geometric models as input. In preprocessing phase, multiresolution models of object in the scene are created using polygonal simplification techniques. During walkthrough, fast visibility and frustum culling based on bounding boxes is performed which eliminates objects that are not visible to the camera. Appropriate level of detail of objects are selected and displayed depending on the distance of the object to the camera. The system achieves acceptable frame rates for moderate-weight models.

In chapter 2 capabilities of human visual system is explained. Chapter 3 explains the architecture of today's most powerful and popular graphics engines. The survey of walkthrough methods is given in chapter 4. Details of the walkthrough system developed in this work is explained in chapter 5. Last two chapters 6 and 7 gives results and conclusions.

Chapter 2

Human Visual System

For decreasing the load of graphics systems in complex tasks, we must consider the capabilities of human visual systems. Without any knowledge about this, graphics output requirements will certainly exceed the limits of current computer graphics systems. We should always keep in mind that humans will view generated images, not machines. Therefore for simulating reality in a virtual environment, satisfying only the human visual system is enough. This can reduce the workload of computer graphics systems considerably and enable real-time walkthrough of one-million-polygon models.

Different applications require different kinds of realism, however they can all be categorized into two classes [3].

(i) Perceptual Realism

An image is perceptually realistic, if the image is the same as its virtual form. That means a viewer synthesizes a mental image to that similar synthesized by the virtual camera. For example, if the scene is dark to the camera, the generated image should also look dark. The degree of perceptual realism depends on the kind of rendering technique used for generating the image. Perceptual realism is especially important for architectural and art applications.

(ii) Visceral Realism

An image is viscerally realistic if the viewer believes that everything in the

image is real. The most important way of increasing visceral realism is complexity. Every detail of the model must be considered and made carefully. For entertainment applications and flight simulators visceral realism is vital.

Although human visual system is complex, it is not perfect. We must consider the thresholds of human visual system in order to achieve both kinds of realism with low cost. Below is an explanation of perceptual limits of human visual system.

2.1 Visual Acuity

Visual acuity is commonly measured in terms of the angle subtended at the eye. For reasonably bright objects on axis in normal lighting, the limit is 1 minute of an arc [4].

Visual acuity weakens rapidly as the object moves outside the central 2 degree region, and at 10 degree of off-axis eccentricity [5]. This should be taken into account when deciding on the resolution of the system. For example, displays that generate a high resolution constant pixel density are wasting a lot of pixels wherever the user is not looking.

2.2 Field of View

Field of view is the area that human eye can see. According to [8], each eye has approximately a 150 degrees horizontal and 130 degrees vertical field of view. The horizontal field of view covers 60 degrees towards the nose and 90 degrees to the side. Vertical boundaries are 50 degrees up and 80 degrees down. Without this knowledge, redundant parts of the models will also be rendered which will cost a lot in terms of time. Furthermore, generated frames will not be realistic because of the wider angle. Parts of the model that lie outside the field of view should not be processed by the graphics system.

2.3 Latency

Latency is the time measured from the setting of an input parameter until the corresponding output is observed. There are many factors that affect latency: input devices, software architecture, rendering time, display scan out time. Different portions of a system may have different latencies and different paths through the system architecture may introduce different latencies. For the rendering portion of a graphics system, the latency is the time difference between a value change, such as the camera position, and the display of the last pixel of the corresponding frame by the display device.

Different applications have different latency requirements. For example reports from flight training indicate that the quality of the perceived-reality degrades as the total latency exceeds 100ms [9]. According to other reports [7], tolerable latency range from 40-80ms for driving simulators to 100-150ms for low-maneuverability flight simulation. Excessive latency may cause the trainee to make mistakes.

2.4 High Frame Rate

Frame rate is the measure of the frequency of changing the frames on the screen. Low frame rates make motion choppy. Especially for head-mounted displays where rapid motion is possible, low frame rates can make virtual reality applications useless. The motion of moving objects will not be seen continuous, instead snap shots from the motion will be displayed. User will have the feeling of a jumping object rather than a moving object.

Another problem is the relation between the refresh rate of the display device and the frame rate being used by the system. If the frame rate is smaller than the refresh rate, a user tracking a moving object will observe multiple copies of the object as seen in Figure 2.1. The number of the copies is equal to refresh rate divided by frame rate [5]. To reduce this effect, frame rate should be selected equal to the refresh rate of the display device.

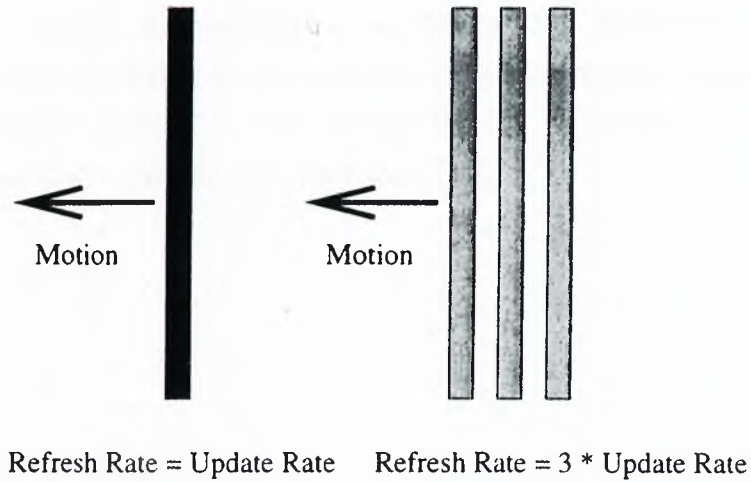


Figure 2.1: Repeating Objects

2.5 Constant Frame Rate

Most applications require a constant frame rate. Changing frame rates will affect the user and lead to misunderstandings. Frame rate variations also cause temporal inaccuracies because the change in frame rate affects the latency. An unexpected increase in latency will cause a frame that is planned to be displayed to be skipped or to be displayed later. This will cause inconsistent motion. In many cases, when a graphics overload occurs, it is better to force a degradation in frame rate so that the change in latency does not occur.

Constant frame rate is vital for immersive and real-time applications. However its priority decreases for design and model applications.

2.6 Temporal Resolution

The peak sensitivity to temporally modulated illumination happens around 10Hz to 25Hz. The frequency increases as the luminance increases. The frequency at which modulation is no longer noticed is called *critical flicker fusion frequency*. It varies between 15Hz to 50Hz [6]. For large bright displays 85Hz or more may be required [7].

Federal Aviation Administration has specified the standards for flight simulators. These standards give metrics for minimum requirements for the concepts explained above. For example, the latency requirement for commercial flight simulators is specified as 100-150ms [10].

Chapter 3

Structure of Graphics Pipelines

There are many computer companies developing and manufacturing high performance graphics workstations with different graphics systems. A taxonomy of graphics systems is given in [4]. In this chapter, the pipelined parallel graphics architectures will be explained. Leading companies such as Silicon Graphics implements this architecture for its latest products. Workstations with this architecture can achieve real-time texture-mapped anti-aliased polygon performance.

The principal form of the pipeline has three stages. It can be seen in Figure 3.1.

The first stage of the pipeline is the CPU. The CPU is not included in the graphics system, it only runs the application and supplies graphics system with input. The input is in the form of data and commands. The responsibility of CPU is to keep the graphics system always full. A program with high CPU overhead or poorly optimized graphics command loops can introduce a bottleneck at the first stage of the pipeline. Architectures allowing multiple

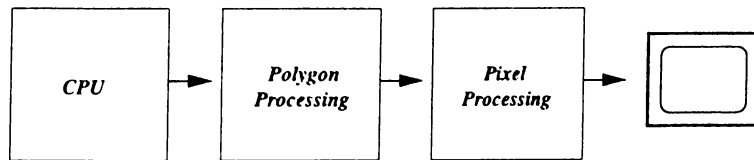


Figure 3.1: Principal Graphics Pipeline

CPU's can reduce the possibility of a bottleneck at the first stage significantly. If there is more CPU power than needed, this can be used to reduce the work for later stages of the system. For example, more advanced and complex culling techniques can be used to reduce the number of polygons that are to be sent to the later stages.

Polygon processing is the second stage of the pipeline. Per-polygon operations such as coordinate transformations, lighting, depth-cueing and clipping are done in this stage. These operations were performed by the CPU in the earlier graphics systems. The latest systems have specially designed geometry engines to perform calculations. Commands and polygons are sent to geometry engines by the CPU. The performance of this stage increases considerably when dealing with long meshes of triangles or packed vertex arrays and few mode changes. Therefore, the commands should be organized to minimize state changes and to batch draw similar objects.

Last stage of the pipeline is the pixel processing. Per-pixel operations such as writing colors into the frame buffer, z-buffering, alpha blending and texturing are done at this stage. The performance of this stage is affected by the number of memory accesses required and hence on the number of pixels and type of pixels. Compared to the old systems that performed these operations using software, new systems have increased the performance of this stage significantly by implementing all the operations using hardware such as depth buffers and texture memories.

The architecture of the graphics systems is important because software should be optimized according to the properties of the architecture and the application. Maximizing frame rate and image quality can become a problem of making the best use of the available stages in the pipeline and avoiding bottlenecks.

For explaining how the pipeline works further, a single triangle will be traced from the beginning of the pipeline to the screen. For the example, Silicon Graphics' Reality Engine¹ is chosen [11] among different graphics engines because of its popularity. The internal structure of Reality Engine is given in

¹Reality Engine is a registered trademark of Silicon Graphics

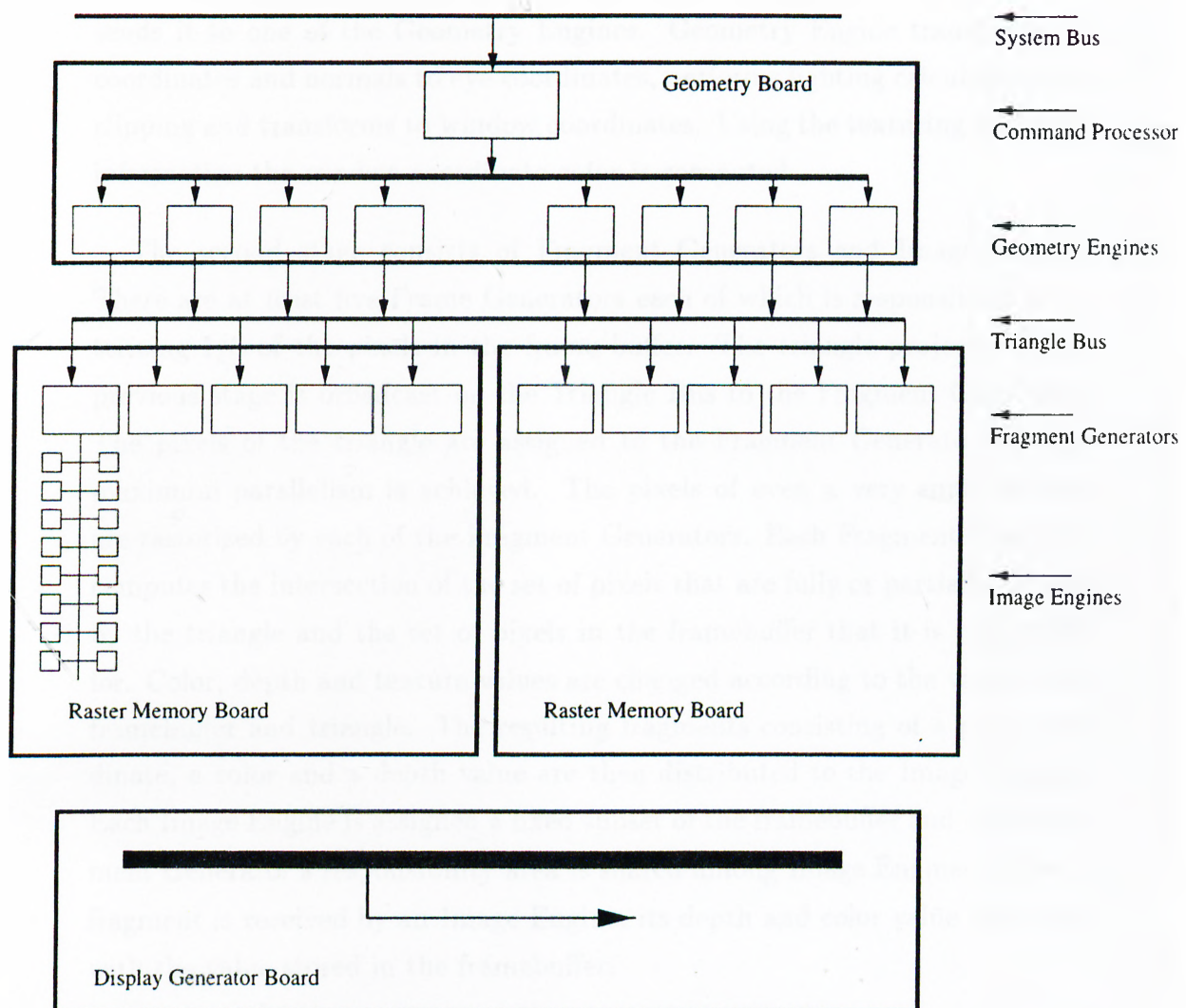


Figure 3.2: Reality Engine Pipeline

figure 3.2.

Reality Engine is a graphics system with pipelined parallel architecture. It implements all the stages explained above except the CPU. As stated in the previous paragraphs CPU is not a part of the graphics system. To reduce the possibility of a CPU bottleneck, high performance architectures such as Reality Engine are fed with more than one CPU.

The per polygon processing stage of the pipeline is implemented by Command Processor and Geometry Engines. All the data and commands describing triangles such as position, color, normal and texture are put to the input FIFO by the CPU. The Command Processor gets input from FIFO, interprets it and

sends it to one of the Geometry Engines. Geometry Engine transforms the coordinates and normals to eye coordinates, performs lighting calculations and clipping and transforms to window coordinates. Using the texturing and alpha information the window coordinate color is computed.

The second stage consists of Fragment Generators and Image Engines. There are at least five Frame Generators each of which is responsible for rasterizing 1/5 of the pixels in the frame buffer. The triangle projected in the previous stage is broadcast on the Triangle Bus to the Fragment Generators. The pixels of the triangle are assigned to the Fragment Generators so that maximum parallelism is achieved. The pixels of even a very small triangle are rasterized by each of the Fragment Generators. Each Fragment Generator computes the intersection of the set of pixels that are fully or partially covered by the triangle and the set of pixels in the framebuffer that it is responsible for. Color, depth and texture values are changed according to the value in the framebuffer and triangle. The resulting fragments consisting of a pixel coordinate, a color and a depth value are then distributed to the Image Engines. Each Image Engine is assigned a fixed subset of the framebuffer and each Fragment Generator's responsibility area is shared among Image Engines. When a fragment is received by an Image Engine, its depth and color value are mixed with the value stored in the framebuffer.

The rendering is finished as soon as the last primitive of the initial triangle is written to the framebuffer by Image Engines.

For applications requiring high graphics performance, knowledge about the graphics system is as important as the power of the workstations and as the efficiency of the algorithms used. Even the most powerful system can become a turtle with inefficient programming. The stages of the graphics pipeline and their relations should be considered when sending polygons to pipeline polygons to the screen.

Chapter 4

Methods for Real-Time Walkthrough

In this chapter, methods that enable real-time walkthrough of complex models will be explained. These methods vary from very simple ideas to very complex algorithms. They can be used together to get combined advantage of each other.

For explaining the benefits of the methods, a simple rendering cost computation is used. This computation is far from being accurate. It only gives a conceptual view of the optimization. The computation is based on the number of triangles T in the scene and on the graphics architecture of the system. Each stage of rendering is assigned a cost. Table 4.1 gives the factors of the computation.

The computation of the rendering cost depends on the implementation of

Symbol	Explanation
$F(T)$	Cost of loading triangles to memory
$X(T)$	Cost of transforming and lighting vertices
$C(T)$	Cost of clipping triangles
$R(T)$	Cost of rasterizing

Table 4.1: Rendering Cost

the graphics pipeline.

For purely software rendering architectures, total cost is simply the sum of all factors: $F(T) + X(T) + C(T) + R(T)$.

For a graphics pipeline explained in chapter 3, the total cost reduces to: $\max(F(T), kX(T) + kC(T), hR(T))$. Since the system has a pipelined architecture, the total cost equals to the slowest stage of the pipeline. k and h are improvement factors because of the hardware implementation of the stages.

Further in the thesis, the rendering cost computation formula will be used to explain the benefits of the acceleration techniques.

4.1 Reducing Geometric Complexity

The main idea of Geometric Complexity Reduction is to get a realistic image without modeling and rendering all the scene. In terms of rendering cost, the aim is to reduce number of triangles T , thus obtain an overall improvement.

4.1.1 Texture Mapping

In the early days of computer graphics, generated images were so smooth that they looked very unrealistic. There were no bumps, scratches or textures on them. For adding realism all the surface details had to be modeled separately which increased complexity.

The introduction of texture mapping [13] solved the problem of complexity. The basic idea is to map a multidimensional image to a multidimensional space for increasing realism. In [14] texture mapping is defined as mapping of a function onto a surface in 3D. The domain of the function can be one, two or three-dimensional and it can be represented by either an array or by a mathematical function. For example a 1D texture can simulate rock-strata, 2D texture can represent waves or surface bumps and a 3D texture can represent clouds, wood or marbles. A survey of texture mapping techniques can be found

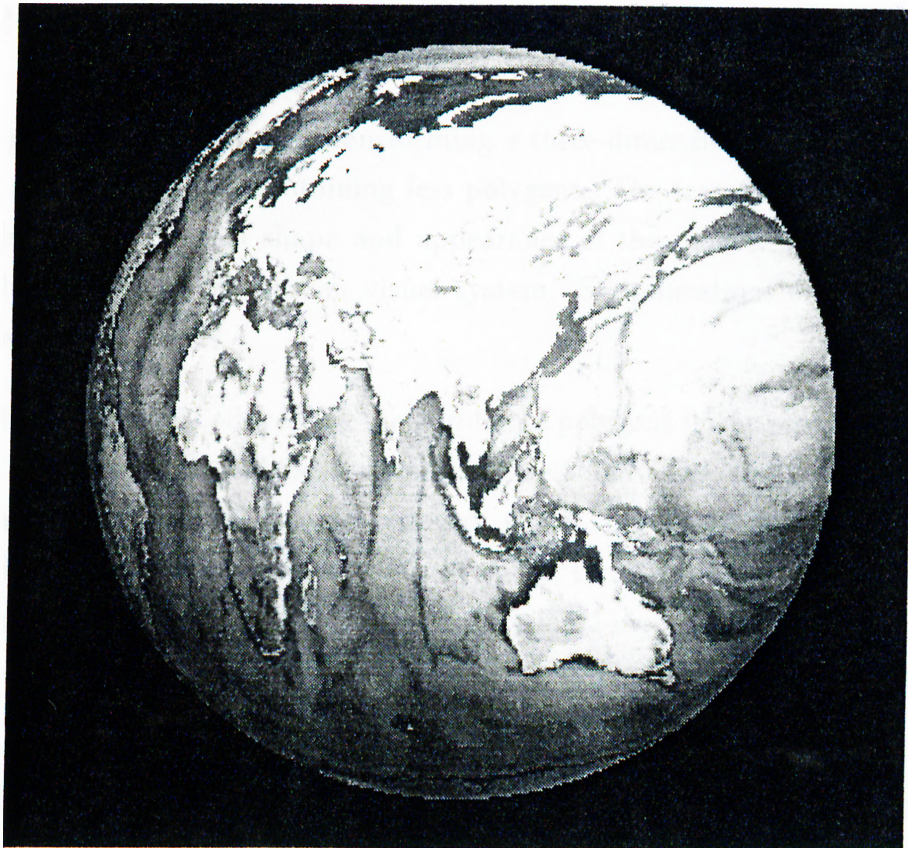


Figure 4.1: Earth-Texture Mapped Sphere

in [14].

The contribution of texture mapping to real-time walkthrough is the increase of realism without additional polygonal complexity. A simple sphere can be shown as earth using an earth texture as shown in Figure 4.1.

Current graphics workstations implement texture mapping on hardware. They have texture memories to store texture and do mapping as a part of their graphics pipeline. For example, using OpenGL and Reality Engine no CPU computations are required for texture mapping. Texture mapping commands of OpenGL are processed by Command Processor of Reality engine, the texture is loaded into texture memory and finally Fragment Generators compute texture coordinates automatically and final image is generated accordingly. One of performance metrics of graphics workstations is the number of textured polygons rendered per second. This also shows the close connection of hardware and texture mapping.

4.2 Polygonal Simplification

Polygonal Simplification is transforming a three-dimensional polygonal model into a simpler version containing less polygons. The transformation tries not to change the original shape and appearance of the model. It makes use of the limitations of the human visual system. Simplifications are usually not perceived by viewers.

Since these methods reduce the number of polygons in the model, rendering load of the system will be reduced considerably. The storage required for storing the model will also be reduced, which simplifies the management of the data from disk to memory. Furthermore, transmission of simplified large models over networks will be faster than original models. If the simplification is good enough, then each stage of the graphics pipeline will have a workload that can be handled in real-time.

In terms of rendering cost formula, the number of triangles T will be reduced, and the overall cost will decrease.

Polygonal Simplification algorithms are categorized into two groups [40]. Algorithms that preserve the topology of the original model and algorithms that does not preserve the topology of the original model. Before explaining some of the popular polygonal simplification algorithms, basic methods will be explained.

4.2.1 Basic Methods

There are three basic methods for polygonal simplification.

Adaptive Subdivision

An adaptive subdivision algorithm starts with a simple base model and recursively subdivides it. Each step adds more detail to the area of subdivision and the basic model approximates the original model. Once the difference between

original and simple model gets smaller than the user-specified error range, the algorithm stops.

These kinds of algorithms have difficulties with selecting the starting simple model. The boundaries of the simple model should resemble the original polygon. At each step of the algorithm a subdivision must be performed. Finding the subdivision to get the best approximation is also another problem. At the end of each step, the difference between the original and simple model are calculated. The calculation may be very complex and time-consuming depending on the complexity of the original model.

The solutions of the problems stated above vary according to the complexity and characteristics of the original model and to the requirements of the application.

Geometry Removal

A geometry removal algorithm starts with the original model and removes faces or vertices from the original model to get a simplified version. The removal process continues until a user-specified error range is reached.

Sampling

This method varies from the other two methods in that it tries to generate a model from scratch. It does not use the vertices of the original model. First a sampling algorithm samples the geometry of the original model by either taking a number of random points from its surface or by overlaying the model with a three-dimensional grid and sampling each box of the grid. The algorithm then tries to create a simple model that fits to the sampled data. The precision of the sampling method determines the success of the simplification. With an inaccurate sampling algorithm the final model will not match the original model.

4.2.2 Topology Preserving Algorithms

Topology Preserving algorithms do not change the local or global topology of the original models. For example, if there is a hole in the original model, the position and shape of the hole is preserved. This restriction sometimes limits the amount of simplification.

Below are topology preserving algorithms that can be used at the preprocessing phase of real-time walkthrough applications explained.

Geometric Optimization

The Geometric Optimization algorithm proposed by Charles Hansen [18] is a geometry removal algorithm which can be applied to any geometric model. The basic idea is to combine coplanar polygons and to obtain an optimized geometry.

The method first tries to create sets of coplanar polygons. It groups neighboring polygons together with roughly the same normal. Each group has a representative normal that can be calculated by averaging all of the normals of the polygons in the group. At each iteration of the algorithm, each polygon's normal is compared to the normal of the neighboring set of polygons. If the normals are close, then the polygon is added to the group. At the end of this process, the groups of polygons are nearly coplanar sets.

Next, the algorithm tries to find the boundaries of sets. For doing this, a segment list for each set is created by adding all the edges of the polygons in the set. Duplicate segments are removed from the list by sorting segments according to the endpoints. For each set, a boundary polygon is formed by linking segments that share end points. At the end, boundary polygons are triangulated.

The algorithm can be applied to any three dimensional model. The success of the algorithm depends on the curvature of the model. Models with high curvature will have a few coplanar sets and the simplification ratio will be low.

User can specify a threshold value for coplanar groups. The threshold affects the amount of simplification. Different threshold values can be used to generate multiresolution models.

A Data Reduction Scheme for Triangulated Surfaces

The Data Reduction algorithm proposed by Bernd Hamann [19] is a geometry removal algorithm which can be applied to any triangulated geometric model. The basic idea is the same as Hansen's method, simplifying regions of low curvature.

First, a weight for each triangle in the model is calculated. The weight is the average of the local curvature with neighboring triangles. To calculate the local curvature the interior angles with adjacent faces are used. At each iteration of the algorithm the lightest triangle is found and replaced with a single point. The coordinates of the point is determined according to the neighboring triangles. All the neighbors are then removed and the hole is filled by combining the vertices at the boundaries of the hole with the point. At the end of the iteration, weights of the newly formed triangles are calculated.

User can specify the number of vertices to be removed from the original model. Therefore, this algorithm can be used to create multiple levels of detail.

Re-tiling Polygonal Surfaces

The algorithm proposed by Greg Turk [20] is different from the algorithms explained so far, because it combines sampling and geometry removal.

The algorithm randomly distributes user-specified number of vertices over the surface of the model. For obtaining a better distribution of vertices, repulsion forces between vertices are used. If two vertices are close to each other, they repel each other and moved over the surface of the model. Thus, the positions of the randomly distributed vertices are refined.

The next step is constructing a new model from the distributed vertices.

This is accomplished by triangulating polygons of the original model with the randomly distributed vertices that lie inside the polygon. The resulting model contains both the original and additional vertices and is very complex. For simplifying the model, the algorithm removes original vertices one by one according to some constraints concerning the preservation of topology. At the end a simplified version of the original model is obtained.

The algorithms explained so far work well on low-curvature models. However, this algorithm can also simplify high-curvature models. High-curvature areas can be defined as low-repulsion areas. This will cause more vertices to be moved to high-curvature areas.

User can specify the number of vertices that are randomly distributed. Therefore, this algorithm can be used to create levels of detail for any object consisting of polygons.

Surfaces: Polygonal Mesh Simplification with Bounded Error

The algorithm proposed by Kalvin and Russel [24] is an adaptive subdivision algorithm.

The algorithm consists of three main steps. The first phase is surface creation phase. A face is selected as the initial surface and adjacent vertices are added to the surface. There are some topology, aspect ratio and error bound requirements for the addition of faces to the surface. When the surface can no longer grow, a new surface is selected and the same operation continues until all polygons are added to surfaces.

The second phase is the border straightening phase. In this phase the jagged faces of the surfaces are straightened by omitting some vertices from the borders.

The algorithm finally triangulates surfaces. The vertices of the final model are a subset of the original model.

There are also other topology preserving methods which can be found in [21], [22]

and [23].

4.2.3 Topology Simplifying Algorithms

Topology Simplifying algorithms do not guarantee the preservation of local or global topology of a model. For example, holes in a model can disappear at the end of simplification. Because of fewer constraints to satisfy, these algorithms have better simplification rates than the topology preserving algorithms. For example, the model of a slice of cheese cannot be simplified by topology preserving algorithms because the holes in the cheese must be preserved.

Topology simplifying algorithms are usually used in applications with real-time constraints. In such application model degradation is acceptable if the model cannot be rendered within a given time.

Multi-resolution 3D Approximations for Rendering Complex Scenes

This simplification algorithm created by Rossignac and Borrel [25] works on any input model. It is a sampling algorithm.

The algorithm works in four main steps: Grading, Clustering, Synthesis and Elimination. The initial step grades each vertex in the original model. It assigns each vertex a weight based upon two perceptually important factors. It considers vertices important that have a high probability of being on the silhouette of an object from an arbitrary viewing angle. The algorithm calculates this factor using the inverse of the maximum angle between any pair of edges adjacent to the vertex. Thus, a vertex at the end of a spike has a high silhouette weighting. The algorithm considers vertices important that bound large faces of the original model. It calculates this factor using the maximum length of all edges adjacent to the vertex. This method calculates a final weighting for each vertex using a linear combination of these two factors.

The algorithm triangulates each face of the original model and then performs clustering that breaks up the bounding box of the model into uniform subboxes

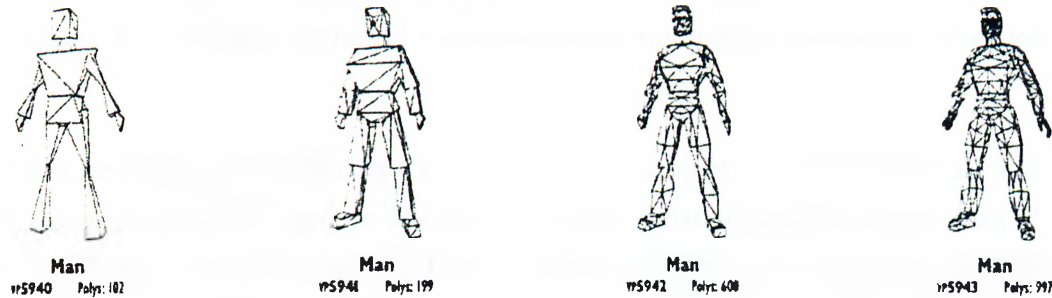


Figure 4.2: Level of Detail

(three-dimensional grid). The algorithm adds each vertex to the vertex list of the subbox that contains the vertex. This method greatly simplifies the model if the subboxes are large and minimally simplifies if the subboxes are small.

Synthesis collapses all the vertices contained in a subbox to the most important vertex in the subbox which is the vertex with highest weight. This step generates some triangles that degenerate to points or edges.

Elimination removes these degeneracies from the model using a simple test of the vertices of each of the triangles.

4.3 Geometric Level of Detail

One of the most efficient and important techniques for managing complex scenes is geometric level of detail (LOD). The main idea of all the LOD techniques is representing objects that do not contribute much to the scene with less primitives.

For example, an object with many polygons that is far away from the viewer requires roughly same amount of time as an object with the same number of polygons that is nearer. The far object will cover a smaller portion of the scene than the near object. The rendering time of objects depends on two factors. The first is the transformation and clipping calculations and the second one is the scan conversion. The first factor depends on the number of primitives and the second factor depends on the area that the object covers in the scene. For

the objects in the example, first and second factors require the same amount of time. The problem is should we spend the same amount of time for the two objects.

For reducing the rendering time of the nearer object, the first factor should be considered, that is the number of primitives constituting the object should be reduced. In other words, T of the total rendering cost formula should be reduced. What is needed are methods for simplifying an object that has been modeled with accessive detail so that arbitrary views can be rendered quickly, ideally with a cost proportional to the number of pixels that the object covers. These methods are called multiresolution models. An example of a multiresolution model¹ is shown in Figure 4.2. The simplest model consist of 102 polygons whereas the most complex model has 977 polygons.

Creation of multiresolution models is quite difficult. Each object in the scene must be modelled with different geometric representations and surface properties. They should be organized in the database so that accessing different representations of the object is simple and not time-consuming.

There are also algorithms that generate multiresolution models automatically. Such algorithms take the detailed model as input and generate simple representations. The algorithms explained in section 4.2 can be used for this purpose. Especially algorithms that let the user specify the simplification ratio can simply be used for generating multiresolution models.

One important requirement for multiresolution models is the preservation of appearance of objects. There are measures for determining the success of the simplifying algorithm [17]. Measures for raster image output of the objects is more important than measures for the topology or geometry. Therefore, an image based error metric is needed. The error metric should measure the difference between an image rendered using the fully detailed model and an image rendered using the multiresolution model. It should take into account that simple representations are used only when the object is far from the viewpoint.

Once the multiresolution model of the scene is prepared, the problem reduces

¹From *Vicupoint Datalabs Summer '96 Catalog*

to managing levels of detail of objects. The management algorithm should consider the time restrictions and image quality restrictions. If the rendering time is exceeded, low-resolution models for far objects should be used.

An adaptive display algorithm that can manage LOD according to quality and time constraints is proposed by Funkhouser and Sequin [26]. In addition to LOD, the algorithm also manages the complexity of the rendering algorithm. The aim is to find the combination of levels of detail and rendering algorithms for all potentially visible objects that produces the best image possible within the target frame rate.

They defined an *object tuple*, (O, L, R) , to be an instance of object O , rendered at level of detail L , with rendering algorithm R . Two heuristics are required for object tuples: $Cost(O, L, R)$ and $Benefit(O, L, R)$. The $Cost$ heuristic estimates the time required to render an object tuple, and the $Benefit$ heuristic estimates the contribution to model perception of a rendered object tuple. S is defined as the set of objects rendered for each frame. The statement of the aim using the above given notation is stated as follows:

Maximize:

$$\sum_S Benefit(O, L, R)$$

Subject to:

$$\sum_S Cost(O, L, R) \leq TargetFrameRate$$

In other words, “do as well as possible in a given amount of time.” The details of the heuristics and optimizations are given in [26].

Other techniques concentrate on the organization of the model. A hierarchical data structure for storing LOD is proposed in [27] and [28] as seen in Figure 4.3. The intermediate nodes of the tree (black nodes) store simplified data of its children. Leaf nodes (white nodes) on the other hand store original data. In [27], the representation of a scene is selected according to the distance of the scene to the viewpoint and the area that the scene covers on the screen.

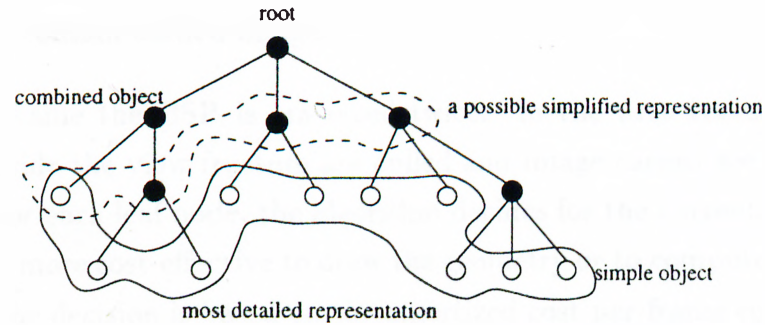


Figure 4.3: Hierarchical Organization of a Scene

If the scene is far away from the viewpoint, the root of the hierarchical organization is rendered. As the scene gets closer, more detailed representations are selected. Figure 4.3 shows two possible representations of a scene. In [28], different search strategies such as depth-first search and best-first search are proposed for selecting the representation.

Algorithms explained so far use geometric simplifications of the original model. Replacing the original model with a texture or a colored cube is another method. In such methods geometric simplification cost is eliminated. Furthermore, the representations are much more simple to render than the representations generated by geometric simplification algorithms.

In [29], a method which replaces cells of the scene obtained by spatial subdivision by colored cubes is proposed. The algorithm in the preprocessing step partitions the scene using octree subdivision. Each cell of octree is assigned a colored cube. Colors of faces of the cube are computed so that it represents general appearance of the geometry contained in the cell as viewed using orthogonal projection perpendicular to the face. During rendering, geometry in near cells are rendered. For far cells that cover a small number of pixels on the screen, the corresponding colored cube is rendered. This interesting method causes some visual artifacts such as discontinuities in solid surfaces. Despite the artifacts, the method accelerates rendering considerably.

Another interesting algorithm [30] makes use of frame-to-frame coherence by caching images of objects rendered in one frame for possible reuse. The algorithm first creates a Binary Space Partitioning-tree (BSP-tree) and partitions the environment. The nodes of the BSP contain geometric primitives

and can also contain cached images.

At each frame the BSP is traversed twice. In the first traversal, nodes that are outside the view frustum are culled and image caches are updated if necessary. For each leaf node, the algorithm decides for the current viewpoint, whether it is more cost-effective to draw the geometry or to compute and cache an image. The decision is based on the amortized cost-per-frame computation given in [30]. For interior nodes, the process is the same, except that instead of considering the cost of drawing the geometry, the cost of drawing the children is considered. After the first traversal of the BSP, BSP is traversed back-to-front again and the scene is rendered by displaying either the geometries or cached images.

Although replacing a cluster of objects with a texture or cube increases performance, angular changes in view position causes the image quality to decrease considerably.

4.4 Optimizing Run-Time Rendering

4.4.1 Visibility and Occlusion Culling

Users in walkthrough applications usually are in the middle of a huge database. They can only see a very small portion of the database. Therefore, sending all the objects in the database to the graphics pipeline would be wasteful.

One trivial optimization to reduce the number of objects to be rendered is culling of the objects to the viewing frustum. This can easily be done on the host CPU and reduce the work of graphics pipeline in great amounts. Clark [32] used an object hierarchy to rapidly cull surfaces that lie outside the viewing frustum.

The method explained in [34] takes care of many aspects of visibility computation and accelerates rendering considerably compared to classical z-buffering.

It processes the scene in terms of object-space coherence, image-space coherence and temporal coherence. An object space octree is used to exploit object-space visibility. Traditional z-buffering is augmented to a Z-pyramid to find image-space coherence. To exploit temporal coherence, the geometry that was visible in the previous frame is used by storing a list of previously visible octree nodes.

More advanced optimizations have also been proposed in [31] and [33]. They take into account large occluders in the database. For example, a user facing the wall of a house will see only the wall of the house but nothing else. In such cases, only the wall of the house should be rendered. Simple algorithms explained in the previous paragraphs would send the wall and the objects that are behind the wall but inside the viewing frustum also to the graphics pipeline. Graphics pipeline will then rasterize all polygons and find out that only the wall is visible.

Occlusion culling techniques are especially important for architectural models. Architectural models generally consists of cells (rooms) and portals (windows, doors, etc.). User in a cell can see only the objects in the cell and objects that are visible through the portals. Methods using this idea can reduce the number of polygons to be rendered to the 1/20 of the original model [33].

Unlike architectural models, terrain models for flight simulators do not contain trivial large occluders. For example, a forest is an occluder in the scene, however it does not block a large portion of the model. Therefore it is not as simple and efficient as in architectural models to use the idea of occlusion culling in terrain models.

Different visibility and occlusion culling algorithms can be used together to eliminate more objects. For example, for a room with no portals, first all the objects outside the room can be eliminated and after that visibility culling can be performed to find out which object to render.

One important concept to consider in visibility determination is the processing power requirements of the culling algorithm. The algorithm should not create a bottleneck in the pipeline. Ironically, the better the algorithm culls

the less time for culling, since the frame rate increases because of the small number of polygons. For example, if an algorithm eliminates nearly all objects in the scene, then the rendering will take a very short time, and the culling algorithm will have much less time for preparing the polygons of the next frame. Therefore, culling algorithms should be customizable according to the CPU and graphics subsystem power.

4.4.2 Level of Detail Switching

Switching between different representations of objects causes visual defects such as popping. Therefore transitions between different representations of objects should be optimized. There are a number of ways to reduce the popping effect.

The first method is called *fading*. Instead of simply switching the models, for a period of time both models are drawn blended together. This reduces the popping, however increases the workload of the graphics system.

The second method is *morphing*. One of the objects is morphed to the other object continuously until they are the same. This method has the best visual effect compared to other techniques, however it is difficult to morph between arbitrary models.

For time-critical applications the popping effect in simple switching can be ignored, because the other two methods require expensive graphics operations.

4.4.3 Mode Sorting

For most graphics workstations changing the mode is very expensive. By mode, the texture and material properties is meant. Therefore, grouping polygons to reduce the number of mode changes is a simple and efficient optimization. It can increase performance incredibly for scenes with many small pieces of textures.

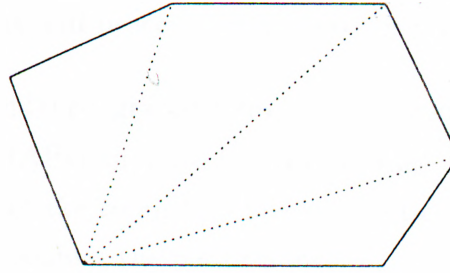


Figure 4.4: Triangulated Polygon

4.5 Optimizing Graphics Databases

4.5.1 Triangle Meshes

Applications for creating geometric models usually are designed to give the designer the easiest way for modeling an object. A designer can use NURBS (Non-Uniform Rational B-Spline curve) [43, 44], surface patches and any arbitrary polygons. On the other hand, graphics engines are usually designed and optimized for rendering triangles. Therefore, triangulating the model as a preprocessing step reduces the rendering time of the scene.

Considering the architecture of reality engine, a significant performance increase can be obtained by generating strips of triangles which allow sharing of vertices of adjacent triangles. A strip of triangles with shared vertices require much less rendering time than the same number of separately processed and rendered triangles.

Other more advanced graphics systems such as Infinite Reality of SGI has ready-to-render arrays. These arrays are filled with vertex normals and vertex data. Vertices in the arrays are accessed using ultra-fast DMA and per vertex transfer overhead is thus reduced.

For triangulated polyhedral objects, the number of vertices is much less than three times the number of triangles in the scene, because of the shared vertices in the scene. This can be seen in Figure 4.4.

For the example, processing each vertex independently will unnecessarily increase $X(T)$ and $F(T)$. There are only six vertices in the scene, however

independent processing will process twelve vertices instead of six.

To reduce the redundancy, graphics engines and associated Application Programmer's Interfaces (API) support triangle meshes. Each vertex is used in conjunction with two of the recently processed vertices to define the next triangle. This approach doubles the performance.

Algorithms minimizing vertex duplication can be found in [15], [16] and [41].

4.5.2 Smart Caching

For reducing the $F(T)$ component of rendering cost, data can be arranged in contiguous memory locations. Also secondary processors for pre-fetching data can be used to reduce delays caused by page-faults. This method becomes especially important when the size of the model exceeds the size of available memory.

A unified memory architecture where all the data is stored, can decrease $F(T)$. All the engines in the system should have direct access to the unified memory. Unlike typical graphics workstations, where data is transferred between RAM and VRAM, all the data should be stored in unified memory. Therefore, the use of unified memory can decrease $F(T)$ to only disk-memory transfer time. Efficient paging can reduce $F(T)$ even to zero.

4.6 Handling Very Large Databases

Another problem with complex models is that the model can be too large to fit into the memory. This problem has two dimensions. The first one is that the size of the model exceeds the RAM of the system and the second one is that the size of texture does not fit into the texture memory of the system. Both problems must be handled separately.

4.6.1 Paging Geometry from Disk

When paging geometry data from disk to the application memory, full I/O bandwidth of the system should be used. This requires structuring the data so that it can be read in large blocks; preferably being transferred using Direct Memory Access (DMA) into the application's address space.

Paging operation should not affect the frame rate of the system. Therefore, it can be performed synchronously between frames. However the amount of data being transferred between frames can be too small to utilize full I/O bandwidth. To avoid this problem in multi-threaded systems, asynchronous loading can be performed by creating a thread for load operation. For pipelined systems such as Reality Engine, paging operations do not affect frame rate if paging operations are pipelined with the graphics operations.

In order to avoid arriving at a point in the scene and not having the correct data to render, the application must predict and adjust timing of loading of data. In very large scenes with long visibilities, it is also important to have the data structured so that low-resolution level-of-detail models and textures can be loaded in without having to read all the resolutions of the model or texture.

4.6.2 Paging Texture from Memory

For achieving fast texture mapping, graphics subsystems have their own textures memories that are used for caching textures. Texture memories are much more expensive than the conventional RAM and maximum available texture memory size is much less than RAM. Therefore, management of texture memory is very critical.

Unlike paging data from disk, paging texture cannot be done asynchronously with rendering because on most graphics architectures texture loading share the same data paths as normal rendering. A fraction of the rendering time must be reserved for texture loading.

Memory management issues such as fragmentation of the already limited

memory should also be considered. The simplest solution to this problem is to load textures with the same size.

The unified memory architecture explained in 4.5.2 solves the problems of texture paging by combining all types of memories in system.

Chapter 5

The Walkthrough System

In this chapter, the walkthrough system developed in this thesis work is explained. The system in the preprocessing phase builds a hierarchy of the scene and generates simplified versions of the objects in the scene. The simplification algorithm is based on removing nearly coplanar triangles from the objects. A number of simplified versions can be produced by the simplification algorithm each with different levels of detail. In the walkthrough phase a virtual camera is moved inside the model. As the camera moves inside the model, the appropriate version of the objects is selected and displayed. The block diagram of the system is given in figure 5.1.

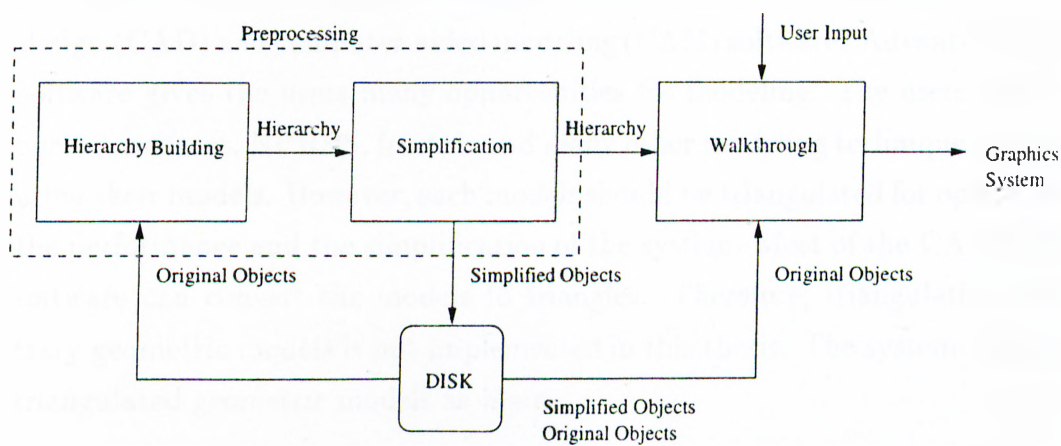


Figure 5.1: Block Diagram of the Walkthrough System

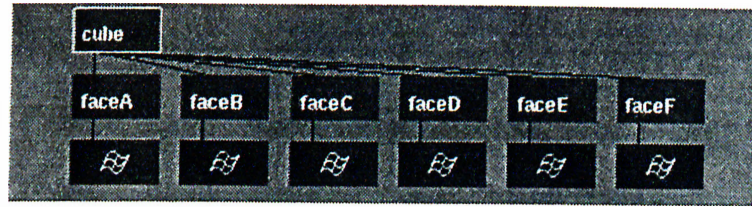


Figure 5.2: Hierarchical Model of a Cube

5.1 Input Model

A complex geometric model of a scene is required by the system. A scene can be modelled in several ways. For example, a simple 3D cube in a scene can be modelled by using its edges which make twelve line segments, by using its faces which make six squares or by using triangles which make twelve triangles. As the object gets more complex such as a sphere, the number of possible representations increases.

For the system, the triangle representation is selected for models. That means all the objects in the scene must consist of triangles. This selection makes the system more appropriate for most of the graphics systems. As explained in section 3, graphics pipelines are optimized for triangulated polygons. In addition, triangle strips can be sent faster than arbitrary polygons to graphics hardware.

Complex geometric models are prepared by using advanced computer-aided design (CAD) and computer-aided modeling (CAM) software. Advanced CAD/CAM software gives the users many opportunities for modeling. The users can use curves, surfaces, NURBS, fractals and many other modeling techniques for creating their models. However, such models should be triangulated for optimizing the performance and the simplification of the system. Most of the CAD/CAM software can convert the models to triangles. Therefore, triangulating arbitrary geometric models is not implemented in this thesis. The system assumes triangulated geometric models as input.

The geometric model should have a hierarchical structure. Objects should consist of other objects and only the simplest objects should have geometric data. By simplest object, the objects in the lowest level of the hierarchy is

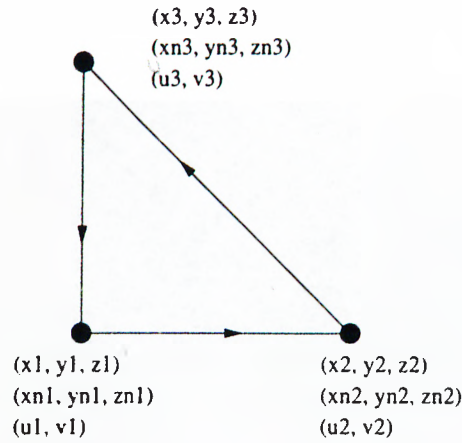


Figure 5.3: Data Required for a Triangle

meant. Hierarchical structure is very suitable for simplification and management purposes. Sample hierarchy of a simple cube is given in Figure 5.2. Faces of the cube contain geometric data. On the other hand, higher levels only contain pointers to lower levels.

For simplifying and drawing a triangle, the system requires 3D coordinates, normal vectors and 2D texture coordinates of the vertices as seen in Figure 5.3. Input model should contain all the data for each vertex in the scene. The ordering of the vertices of the triangles must be counterclockwise and the texture coordinates must be between 0.0 and 1.0.

Object Separated Triangle Format (OSTF) [42] is selected for importing models from CAD/CAM software. OSTF meets all the requirements explained above. It stores objects in hierarchical order. The hierarchy is based on objects. At lowest level of OSTF hierarchy lies triangles. Each triangle has its vertex, normal and texture data. The details of OSTF is given in Appendix A.

5.2 The Preprocessing Phase

In the preprocessing phase, scene data is read from the OSTF file and an object hierarchy is built in memory. Then, simplified versions are built and stored in separate files for later use. Each phase of the preprocessing is explained below.

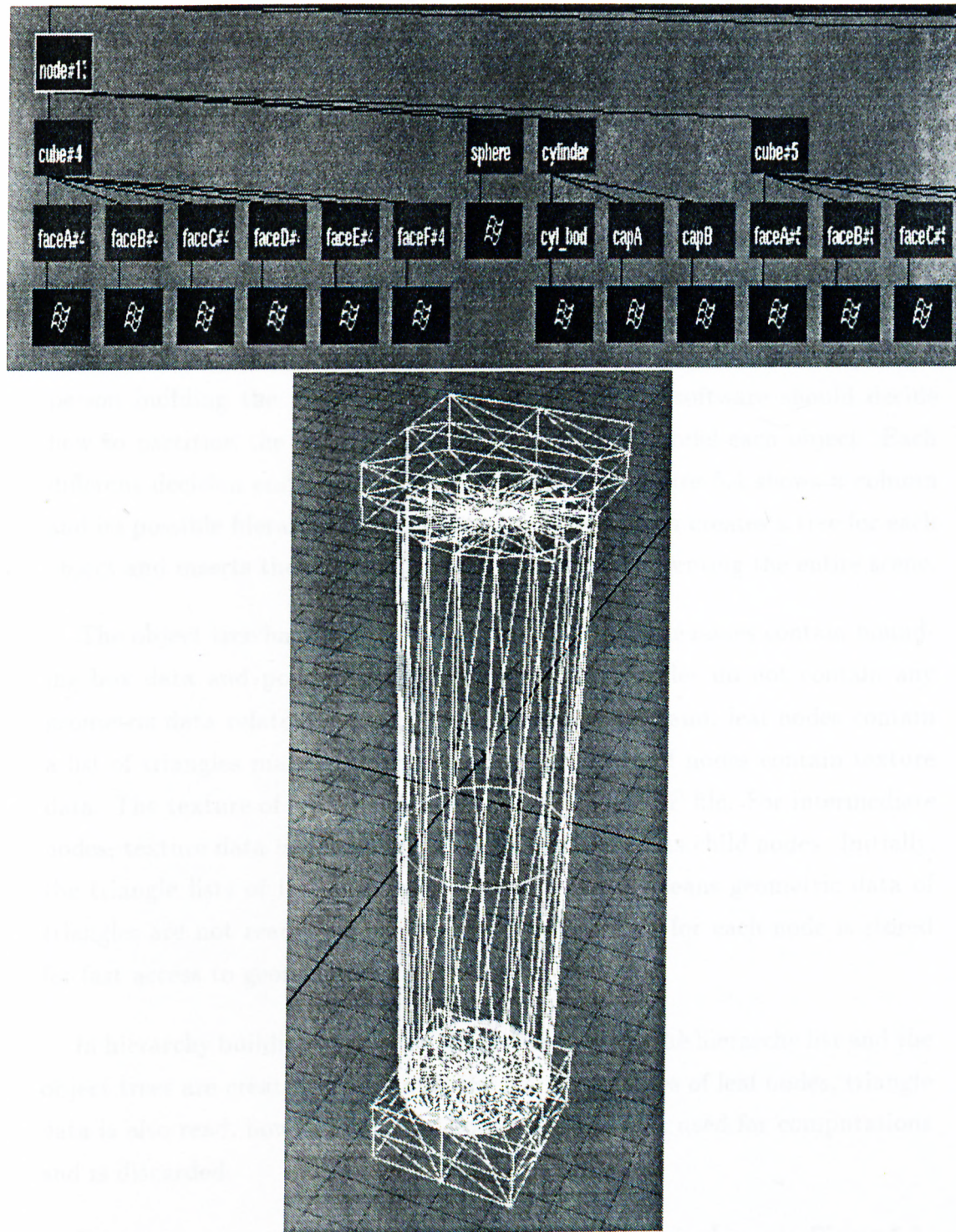


Figure 5.4: Hierarchical Model of a Column

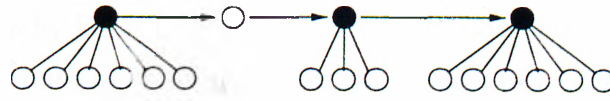


Figure 5.5: Hierarchy List of the Column

5.2.1 Building the Hierarchy

The first step of preprocessing is building the hierarchy of the geometric model in memory. The hierarchy is in the form of a list. Each element of the list is a root of a tree representing an object.

Each object in the scene is represented by a tree in the hierarchy. The person building the model by using the CAD/CAM software should decide how to partition the scene into objects and how to model each object. Each different decision ends with a different hierarchy. Figure 5.4 shows a column and its possible hierarchical representation. The system creates a tree for each object and inserts the root of the tree to the list representing the entire scene.

The object tree has two types of nodes. Intermediate nodes contain bounding box data and pointers to lower levels. These nodes do not contain any geometric data related to the objects. On the other hand, leaf nodes contain a list of triangles making up the object. Both types of nodes contain texture data. The texture of leaf nodes is taken from the OSTF file. For intermediate nodes, texture data is obtained by mixing textures of its child nodes. Initially, the triangle lists of leaf nodes are all empty. That means geometric data of triangles are not read from the file. Only a file offset for each node is stored for fast access to geometric data.

In hierarchy building phase, the OSTF files is read, the hierarchy list and the object trees are created. For finding the bounding boxes of leaf nodes, triangle data is also read, however it is not stored. Instead it is used for computations and is discarded.

The hierarchy list of the column shown in Figure 5.4 is shown in Figure 5.5. Black nodes represent intermediate nodes and white nodes represent leaf nodes. Note that the depth of the hierarchy tree can increase depending on the complexity of model.

After the hierarchy list is built in memory, it is traversed for setting texture and bounding box data of child nodes. Bounding box of an intermediate node is the smallest box that contains all the bounding boxes of its child nodes. Texture data of an intermediate node is found by mixing textures of children. Figure 5.6 shows an example of texture mixing for an object tree with 5 nodes.

5.2.2 Simplification

The second and last phase of preprocessing is simplification of geometric data. In section 4.2, several polygonal simplification techniques are explained. Among those techniques, Geometric Optimization [18] explained in subsection 4.2.2 is selected. The technique is simple to implement and is very effective on planar triangulated surfaces. In addition, the amount of simplification can be adjusted by giving threshold values enabling the generation of multiple levels of detail for a single object.

Figure 5.7 gives the steps of simplification process. Details of each step is given below.

Reading Geometric Data

For each element of the hierarchy list, the tree represented by the element is traversed until leaf nodes. For each leaf node, geometric data of triangles constituting the node are read from the OSTF file. File offset values which are stored in leaf nodes in hierarchy building phase are used for accessing the file. This process can require an enormous amount of memory depending on the size of the scene. For reducing the memory needs, the simplification process handles leaf nodes one by one, which means only triangles of a single node are read and processed at a time from the file.

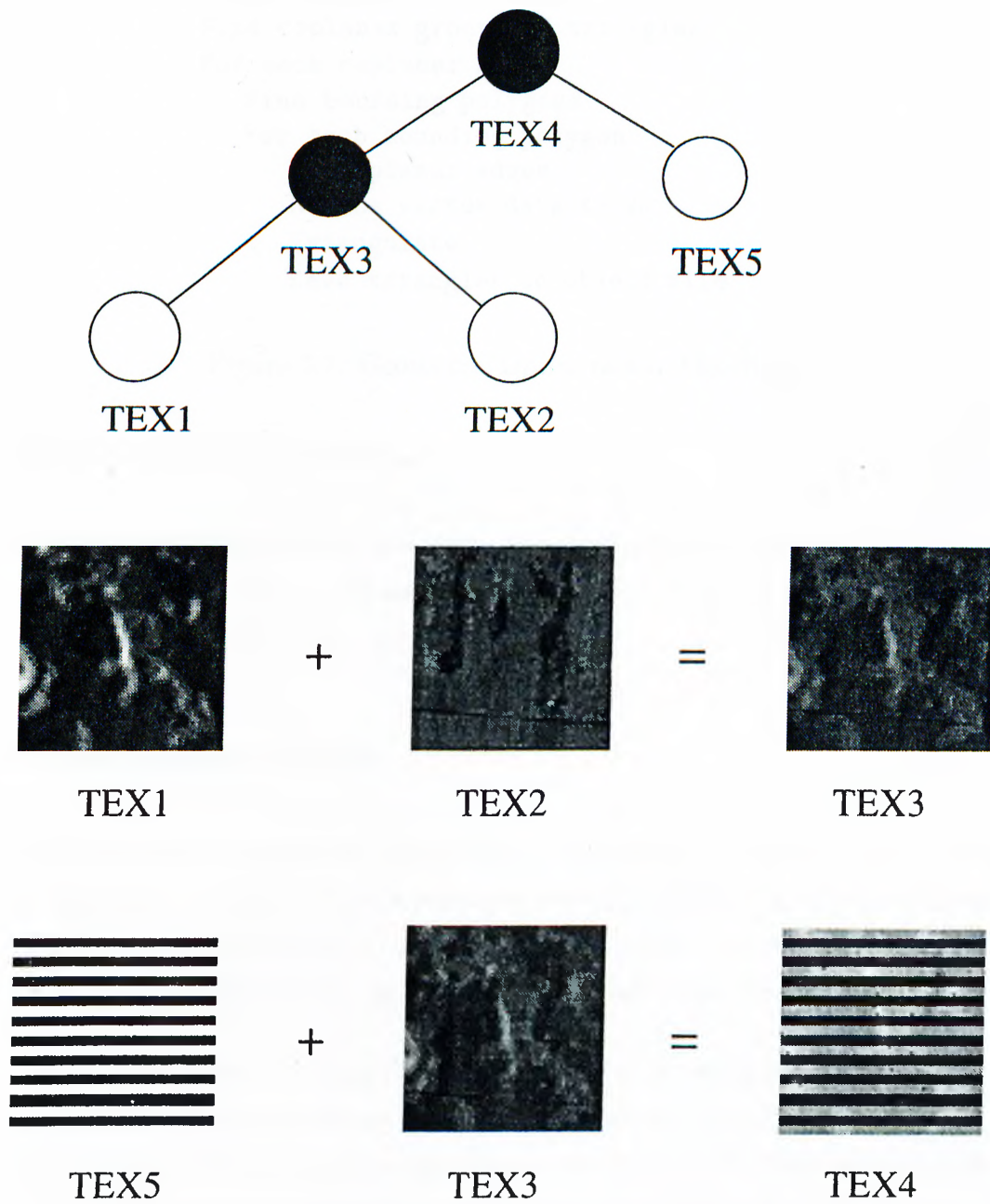


Figure 5.6: Finding Textures of an Intermediate Node

```
For each object tree in the hierarchy list
  For each leaf node of object tree
    Read geometric data of triangles
    Find normals of triangles
    Find coplanar groups of triangles
    For each coplanar group
      Find bounding polygons
      For each bounding polygon
        Omit planar edges
        Reduce vertex data to 2D
        Triangulate
        Save triangles to object file
```

Figure 5.7: Geometric Optimization Algorithm

Finding Triangle Normals

Each triangle of a leaf node has its vertex coordinate and vertex normal data ready in memory. Using this data, the normal vector of each triangle is calculated as explained in [4, 35, 36].

Finding Coplanar Groups

This operation is applied to each leaf node separately. Triangles of the node are grouped according to their normal vectors. Threshold value ϵ_1 for grouping determines which triangles to put to the same group. A small ϵ_1 will cause lots of groups with few triangles and a large ϵ_1 will cause vice versa.

Figure 5.8 shows a triangulated surface. The system given two different thresholds creates two different coplanar groups as shown in Figure 5.9. Triangles which are in the same group are colored similarly. Note that the threshold for the upper scene is smaller compared to the threshold for the lower scene.

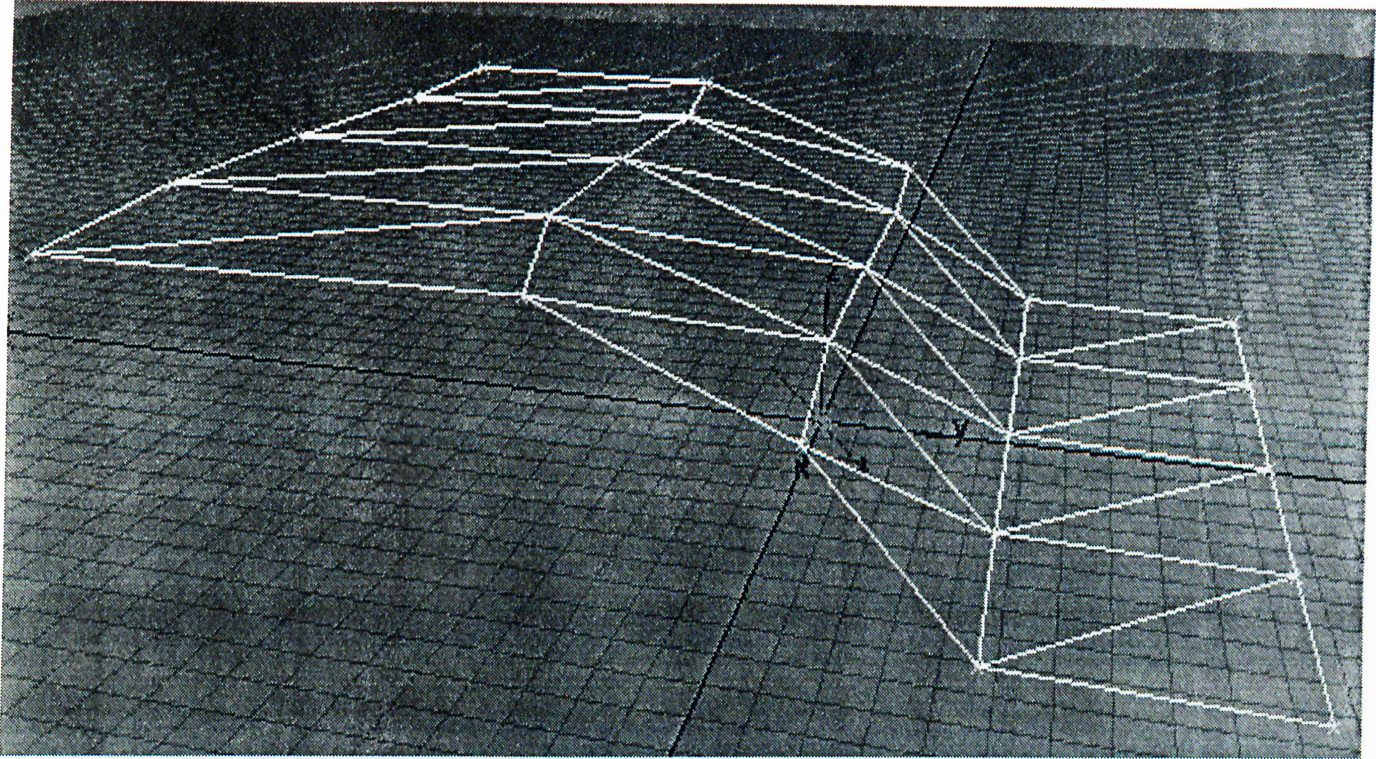


Figure 5.8: Original Surface

Finding Boundary Polygons

For each coplanar group created in the previous step, a boundary polygon is found by omitting shared edges between triangles. Shared edges are determined by using an edge list.

It should be considered that more than one boundary polygon for a coplanar group can be found. This means there can be groups of triangles which have the same normal vector but are not neighbors. An example of this situation can be seen in Figure 5.9. *Plane A* and *Plane D* are in the same coplanar group, however they are not neighbors. For solving such situations, additional data and algorithms are used.

The edge list consists of nodes with midpoint data, group number and triangle list. The midpoint data is used for storing the midpoint of the edge and group number is used for creating multiple boundary polygons and determining sub-groups. Triangle list stores pointers to the triangles sharing the edge. Each triangle also has fields for storing pointers to the edge list elements for

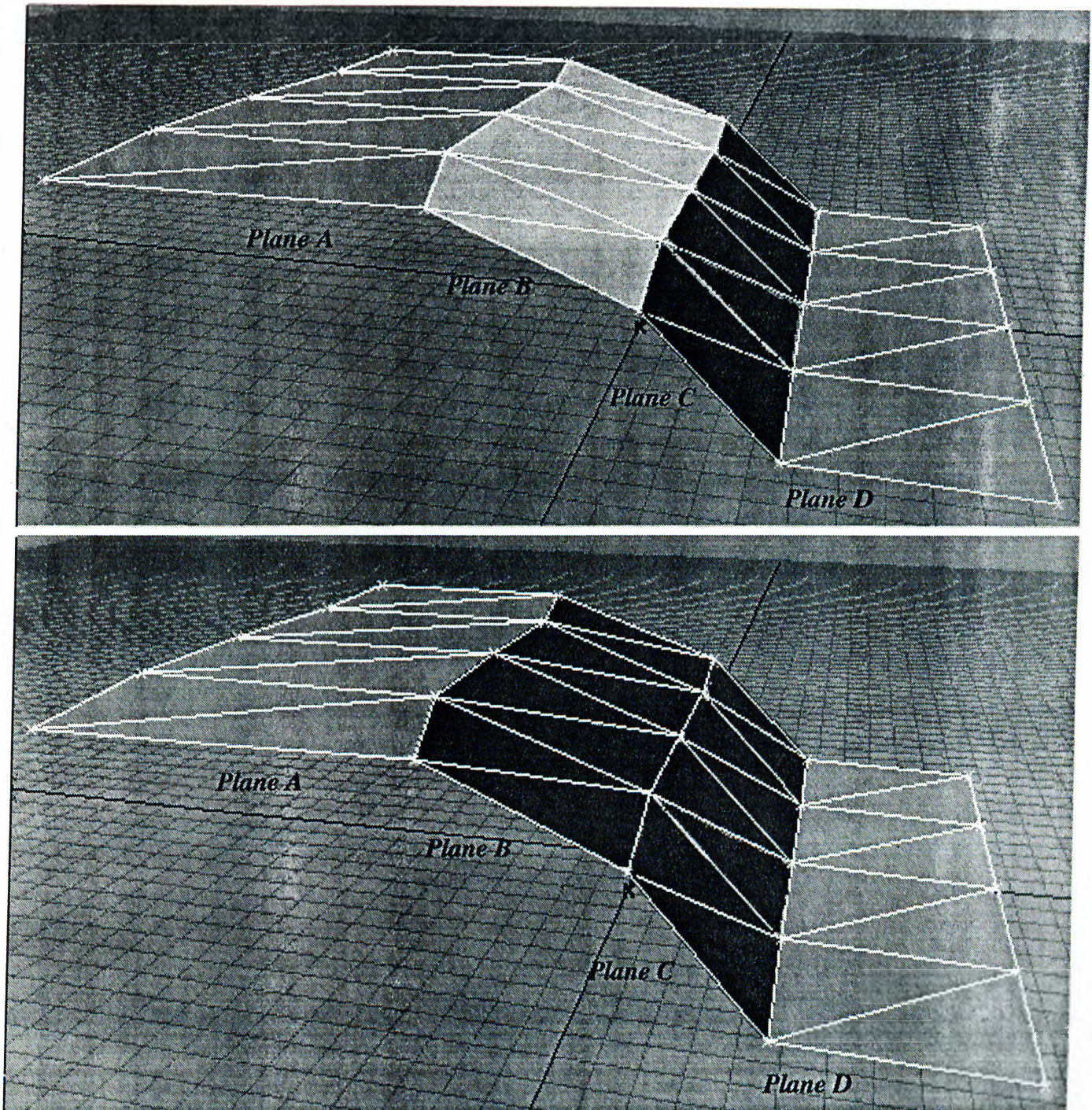


Figure 5.9: Possible Groups of the Original Surface

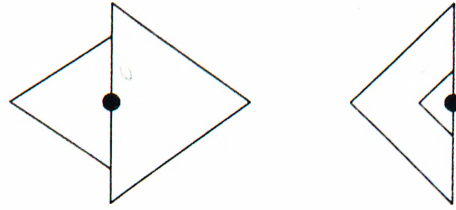


Figure 5.10: Ignored Neighboring Triangles

its edges.

Each vertex of triangles in the coplanar group is added to the edge list. The edge list is sorted by midpoints of the edges. For the insert operation a binary search is performed. At the end of the search, an appropriate place for the new edge is found. If an edge with the same midpoint is already in the list, the edge in the list is marked for deletion and the new edge is discarded. There is no need for checking the end points of edges, only midpoint checking is enough. Consider two edges with same midpoints and different endpoints as shown in Figure 5.10. Such triangulation can cause degradation in image quality and is avoided by CAD/CAM systems. Therefore, existence of such triangles is ignored. In both deletion and insertion cases, the triangle is added to the list of the edge list element and edge list element is added to the list of triangle for later use.

Figure 5.11 shows the edge list of a simple polygon. The polygon consists of two coplanar groups, therefore two edge lists and their corresponding triangle lists are created. *Tri2* and *Tri3* together make a planar group. They have a shared edge E . The node representing edge E is colored, because it is marked for deletion.

After all vertices are inserted to the edge list, a recursive algorithm is executed for finding the sub-groups of the coplanar group. The main idea is assigning sub-group numbers to edges that are on the same bounding polygon. The algorithm starts by giving sub-group number zero to the head of the edge list. All the triangles that are in the list of the edge are then processed. For each triangle, the edges of the triangle are assigned sub-group number zero. This process continues until all the triangles that are in the list of the head edge are processed. At the end of first recursion, all edges that belong to the first

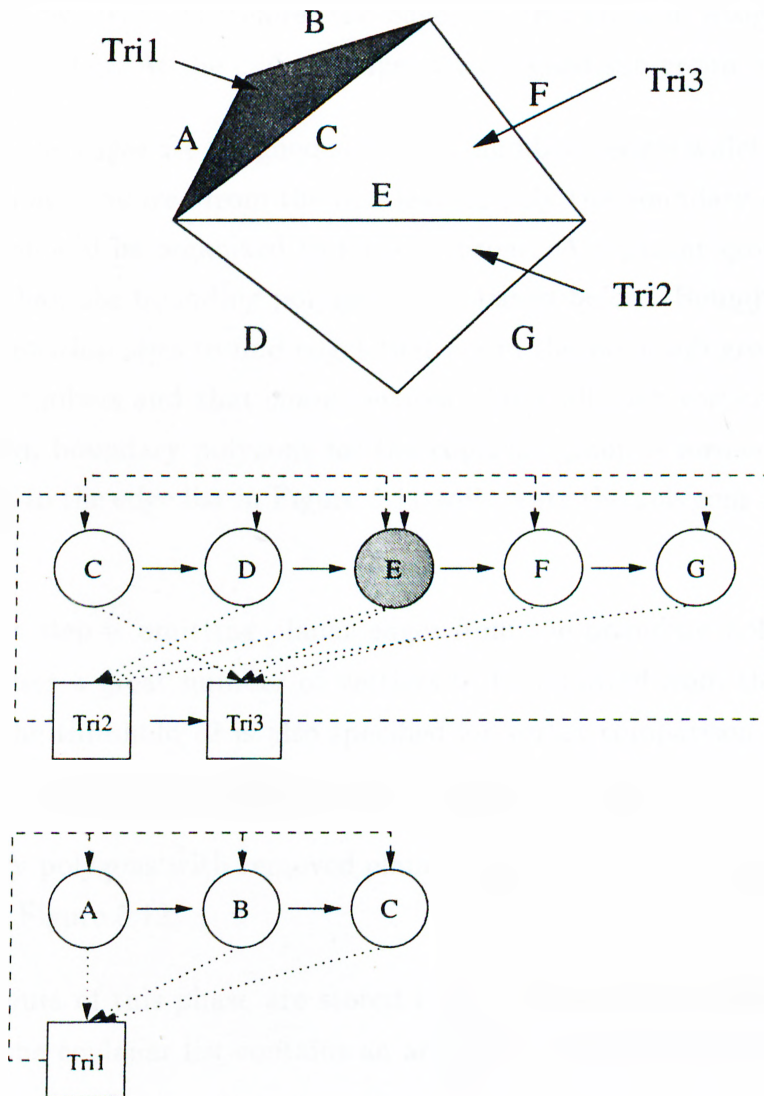


Figure 5.11: Edge List for a Simple Polygon

sub-group are assigned the sub-group number zero. The algorithm continues until all the elements of the edge list are assigned sub-group numbers.

For the example in Figure 5.11, node *C* is assigned sub-group number zero. The method then processes edges of *Tri3* and gives zero to edges *F* and *E*. Since the edge *F* only belongs to *Tri3*, it is finished. On the other hand, edge *E* is shared by *Tri2*. Therefore, the edges of *Tri2* are also assigned zero as sub-group number. At the end, all edges are assigned sub-group number zero.

Once all the edges are assigned sub-group numbers, edges which are marked for deletion are removed from the list, leaving only the boundary edges in the list. They should be organized to form polygons. A coplanar group can contain more than one bounding polygon as explained before. Boundary polygon building algorithm tries to find edges that are in the same sub-group by using the group numbers and that share vertices. After all such vertices are found and grouped, boundary polygons for the coplanar group is formed. Applying the method to the edge list in Figure 5.11 will create the polygons *A*, *B*, *C* and *C*, *F*, *G*, *D*.

The next step is omitting planar edges from the bounding polygon. This method causes a great number of vertices to be removed from the bounding polygon. The threshold ϵ_2 is also specified for vertex comparison. A large ϵ_2 will cause very simple, but inaccurate bounding polygons.

Boundary polygons with removed planar edges of the surfaces in Figure 5.9 is shown in Figure 5.12.

The outputs of this phase are stored in the coplanar list of objects. Each element of the coplanar list contains an array of vertices of its bounding polygons.

Reduce Boundary Polygon to 2D

Boundary polygons created in the previous phase are triangulated for optimizing graphics performance of the system. Depending on the value of threshold ϵ_1 for creating coplanar groups, vertices of bounding polygon may not be on

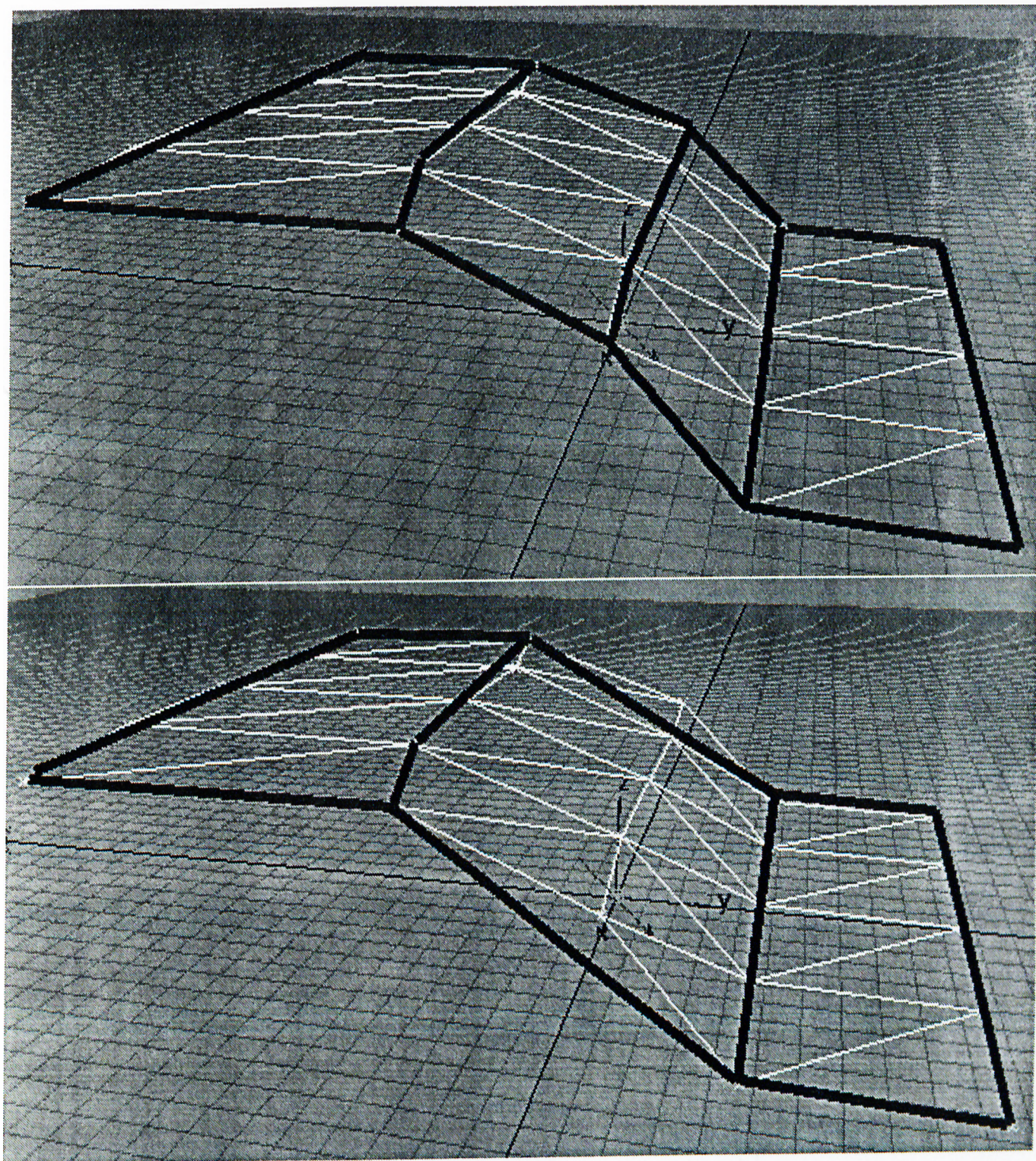


Figure 5.12: Boundary Polygons of Coplanar Groups

the same plane. Therefore, vertices of the bounding polygon are reduced to 2D coordinates. This guarantees that the vertices are on the same plane and simplifies triangulation.

For 3D-2D conversion of vertices, a coordinate system whose z-axis is perpendicular to the plane bounded by the boundary polygon is formed. This is done by selecting three arbitrary vertices from the bounding polygon and finding normal of the plane represented by the three vertices. The normal becomes the z-axis. The vector between the two vertices becomes the x-axis. The cross product of both axes gives the y-axis. After the coordinate system is formed: for each vertex, projection to the x and y axes are found. At the end of this process, each vertex on the boundary polygon has a 2D coordinate. Notice that for very large ϵ_1 values, the space bounded by the boundary polygon can become a volume rather than a plane. In such cases the 2D coordinates generated in this phase will be useless. Therefore, large ϵ_1 values are not used in coplanar grouping phase.

Triangulating the 2D Boundary Polygon

The 2D polygon formed in the previous phase may not be convex. Triangulation algorithms for non-convex polygons such as [39] are complex and have restrictions. Remember that triangulation in the system is a part of the simplification phase and triangles created in this phase will be displayed only when the object is far from the camera. Therefore a fast and simple triangulation algorithm which can cause visual artifacts is implemented. The algorithm first creates the convex hull of bounding polygon and after that triangulates convex polygon.

First the convex hull containing the polygon is found. Graham Scan algorithm is used for that purpose. The algorithm starts by selecting the vertex with the smallest y coordinate. After that, for each vertex of the bounding polygon the angle with respect to the selected vertex is computed. Vertices are sorted using quick-sort by angles. Using the sorted list and an additional stack structure a counterclockwise traversal is searched. The pseudo-code of the algorithm is given in Figure 5.13. Figure 5.14 shows a sample execution of

```

GrahamScan( P, N )
  Find the point with lowest Y and exchange with p[1]
  Sort vertices by angle with respect to p[1]
  push( p[1] )
  push( p[2] )
  for I = 1 to N
    while ccw( top\_next, top, p[I] ) = CW
      pop
    push( p[I] )

```

Figure 5.13: Graham Scan Algorithm

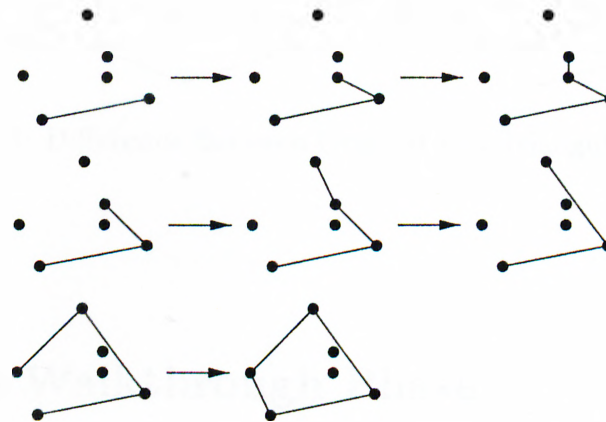


Figure 5.14: Sample Graham Scan Execution

the algorithm.

At the end of Graham Scan algorithm [45] a 2D convex counterclockwise polygon is obtained. The convex polygon is easily triangulated. The first vertex of the convex polygon is selected as base. Using the base, other vertices are grouped to form triangles. Figure 5.15 gives the pseudo-code of the triangulation algorithm. Triangles generated by the algorithm are in form of triangle fans, as seen at the right of Figure 5.16.

The triangulation changes original polygon by omitting vertices that cause concavities. This results in different views of original and triangulated polygons as seen in Figure 5.16. However, changes are usually not noticed by the user because simple versions are only displayed when the object is far from the

```

Triangulate( P, T )
  Select P[1] as base }
  Select P[2] as active}
  for I = 3 to vertexCountOf(P)}
    Form triangle, (base, active, P[I])}
    aSelect P[I] as active}

```

Figure 5.15: Triangulation Algorithm

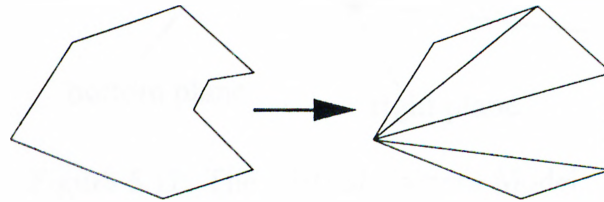


Figure 5.16: Difference Between Original and Triangulated Polygon

camera.

5.3 The Walkthrough Phase

In the walkthrough phase, a virtual camera is moved inside the geometric model interactively. The user can control the direction and position of the camera. As the camera moves, a fast frustum culling is performed to discard that are out of viewing frustum. Appropriate level of detail of objects is selected and displayed according to the distance of the object to the viewpoint. Each part of the walkthrough is explained below.

5.3.1 Camera Model

The virtual camera consists of three elements: direction vector, current position and angle of field of view. All elements of the camera can be changed interactively by the user. Figure 5.17 shows the camera model.

The direction vector is always parallel to x-z plane. That means the movement of the camera is perpendicular to y-axis. As the camera is turned by the

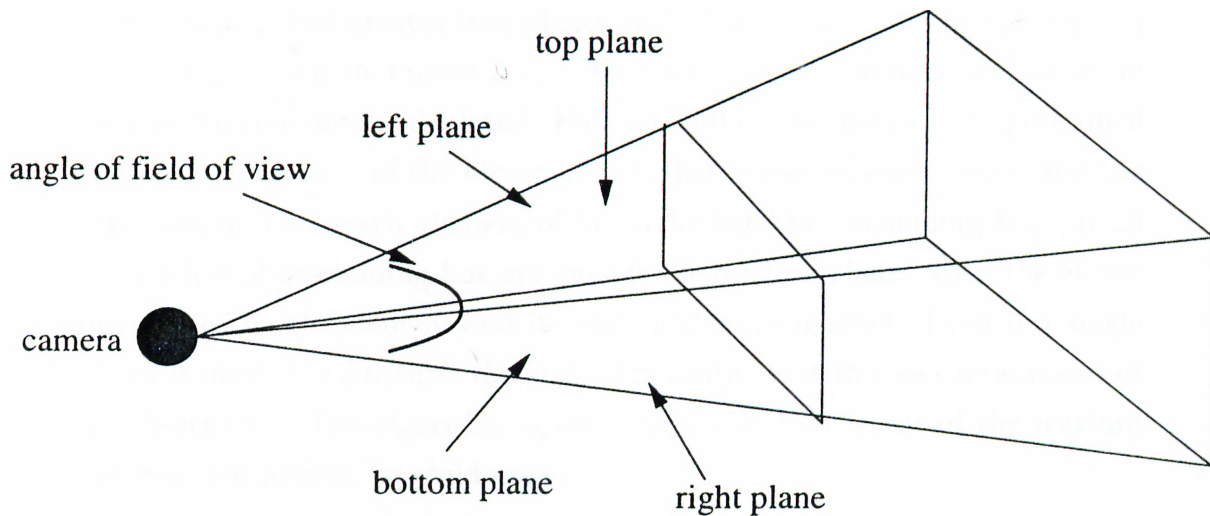


Figure 5.17: The Virtual Camera Model

user, the direction vector changes depending on the direction of rotation. The system takes advantage of OpenGL projection calculations and matrix stack for changing the direction vector. The details of projection calculations are given in Appendix B.

The current position changes as the user moves the camera back and forth. The amount of change depends on the value of the direction vector. For example, if the direction vector is perpendicular to x-y plane, then only the z component of the current position changes. Current position is also used for frustum culling calculations.

The angle of field of view determines how objects are projected. It also affects frustum culling calculations. The value of the angle is selected as 45 degrees, however it can be changed interactively by the user. A new projection matrix is used to re-render the scene as the angle of field of view changes.

5.3.2 Frustum Culling

A viewing frustum is formed using the angle of field of view, current position and direction vector. Objects in the scene are culled against the frustum. Hierarchical structure which is built in preprocessing phase is used for calculations.

The system first creates four planes each of which are sides of the frustum pyramid as shown in Figure 5.17. After each camera movement, planes of viewing frustum are recalculated. Frustum culling calculations are performed first on each element of the hierarchy list, that is root of each object tree. As explained in 5.2.1, each element of hierarchy list has a bounding box. If all the vertices of a bounding box are outside the frustum planes, then the object represented by the element and its child nodes are ignored. Even if a single vertex is inside the frustum, the algorithm continues with lower level nodes of the object tree. The algorithm ignores each node that is out of the frustum and does not process its child nodes.

The explicit frustum culling used in the system introduces overhead when a large amount of objects are in the frustum. However, for cases when explicit frustum culling prunes object trees, its overhead is compensated.

5.3.3 Managing Level of Detail

The system in this phase selects appropriate level of detail of objects in the scene. Only objects that are in the viewing frustum are included in calculations.

As explained in Section 5.2.1, hierarchy list consists of roots of object trees. The intermediate nodes of object tree contain bounding box and texture data. Leaf nodes in addition to the bounding box and texture data contain geometric data.

During walkthrough, each object tree in viewing frustum is traversed. For each node, bounding box data and current camera position is used for calculations. The distance between the camera position and the center of each plane of the bounding box are calculated. The minimum of these six distance values gives an estimate of distance of the node to the viewpoint. If the distance is less than a user-defined threshold, child nodes of the node are processed. For nodes that are far away from the camera, only bounding box of the node is rendered using texture of the node. Children of the node are skipped. This method reduces rendering operation for far objects only to a textured rectangular prism rendering no matter what the geometric complexity of the object

```

For each element of hierarchy list
  If node in viewing frustum
    Find estimate of distance
    If distance greater than threshold
      Render textured bounding box
    Else
      If leaf node
        Select appropriate level of detail
        If representation not in cache
          Load representation to cache
        Render representation
      If intermediate node
        Process children

```

Figure 5.18: Level of Detail Management

is.

Distance calculation for near objects proceeds to the leaf nodes of the object tree. For leaf nodes, more than one threshold value is defined. Threshold values ϵ_1 , ϵ_2 and distance of the leaf node to the camera determine which representation of the object to render. Possible representations of the leaf nodes depend on the simplification phase of the system. In addition to the representations created in simplification phase, textured bounding box of leaf node is also an option for rendering.

Switching between different representations of an object introduces two problems to the system. The first one is the visual defects such as popping and the second one is the loading time of different representations. The solution of popping is blending different representations. It can be implemented by fading in the new representation, while fading out the old representation. For fading, alpha-blending is required, which is an expensive operation. Therefore, popping problem is not handled by the system. Thus, in exchange of performance increase, image quality is degraded.

For solving the disk-loading problem, memory caches are used by the system. There are two caches for each leaf node of an object. If the node is out of the viewing frustum, both caches are empty. For nodes that are in the viewing

frustum, caches contain two different representations of the node. The caching especially helps reducing delay of continuous switching. Consider a leaf node that is d units away from the camera with threshold value $d + \epsilon$. Without caching, for each back and front movement of the camera, loading of geometric data is required. However with caching both representations will be in memory and this will increase performance.

The pseudo-code of level of detail management is given in Figure 5.18.

Chapter 6

Results

In this chapter, results obtained from the system will be given. The results of polygonal simplification, level of detail management and walkthrough are given in separate sections.

6.1 Geometric Simplification

The amount of simplification depends on two thresholds as explained in section 5.2.2. The first threshold ϵ_1 affects the number of polygons in coplanar groups. The second threshold ϵ_2 is used for removing planar edges from boundary polygons. Examples of simplification are examined both in terms of ϵ_1 and ϵ_2 .

The polygonal simplification algorithm gives the best visual and numerical results for planar surfaces. Figure 6.1 shows an extreme example. The square in 6.1(a) consists of 2048 triangles. Since there is only one coplanar group and all vertices except the four corners are planar edges, for any value of ϵ_1 and ϵ_2 , there will be only two triangles at the end of simplification as shown in Figure 6.1(b).

Figure 6.2 shows the effect of ϵ_2 . The polygon is planar, therefore ϵ_1 will not have any effect in simplification process. The original polygon has 10 vertices

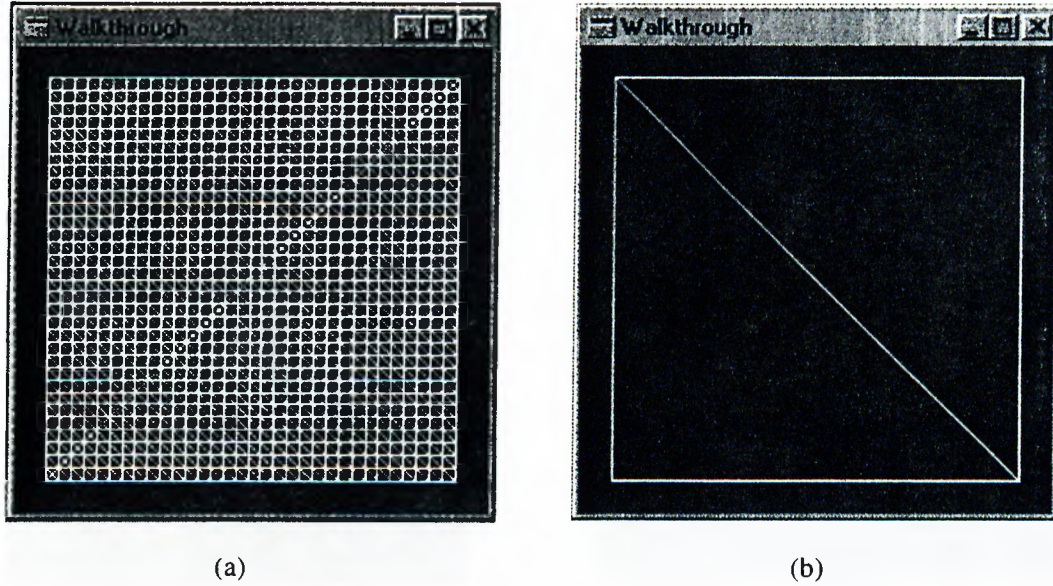
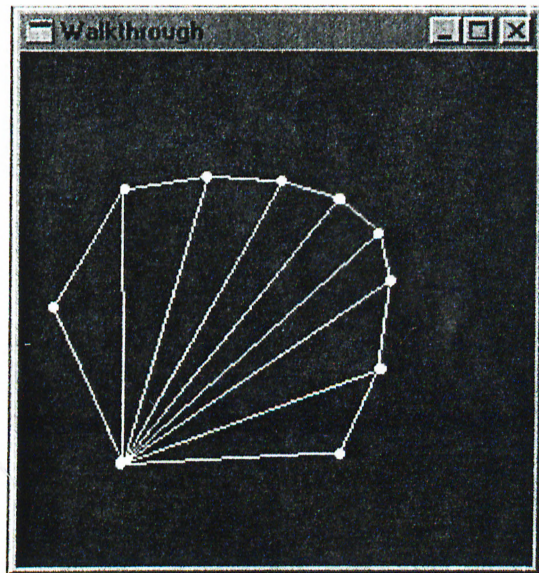


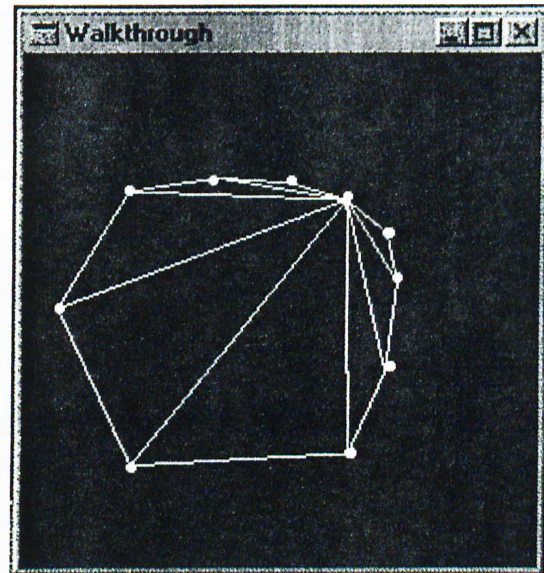
Figure 6.1: Polygonal Simplification of a Square

Figure	ϵ^2	Vertex Count	Triangle Count
6.2(b)	0.0	10	8
6.2(c)	0.5	6	4
6.2(d)	1.0	3	1

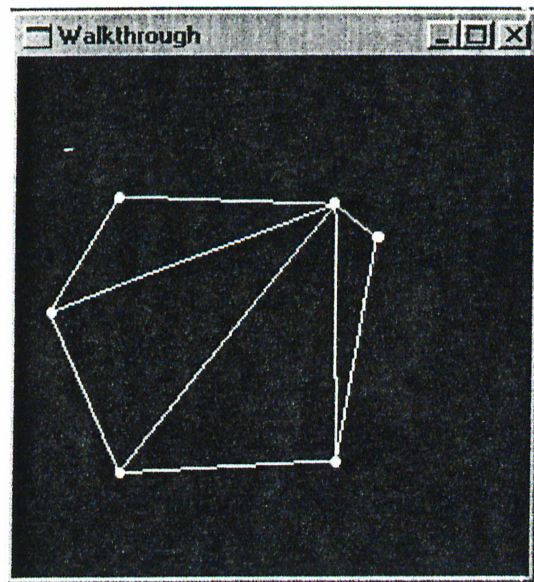
Table 6.1: Polygonal Simplification of a Polygon



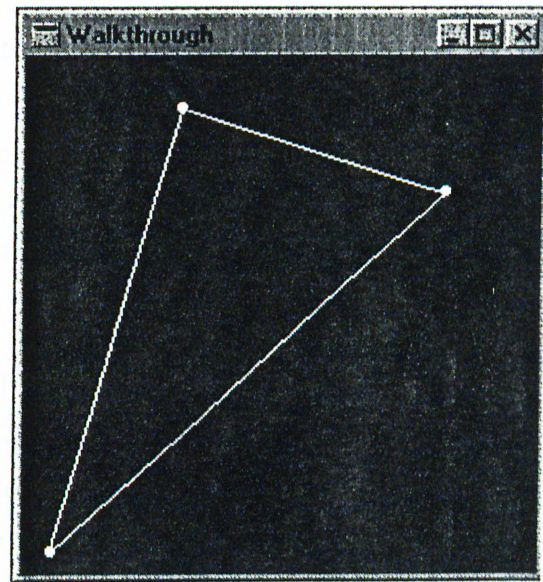
(a)



(b)



(c)



(d)

Figure 6.2: Polygonal Simplification of a Polygon

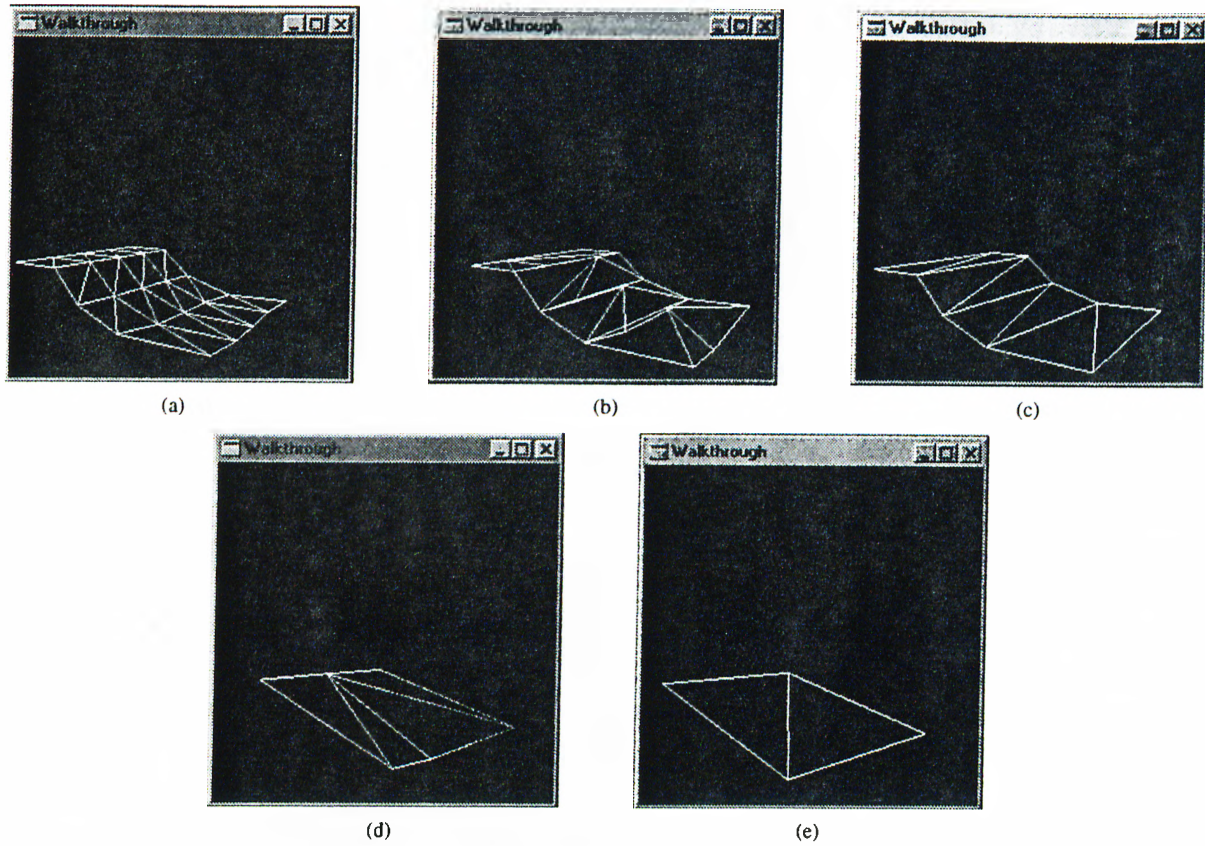


Figure 6.3: Polygonal Simplification of a Surface

and 8 triangles. Table 6.1 gives result of simplification with different ϵ_2 . As it can be seen from Table 6.1, the increase of ϵ_2 results in better simplification, however excessive values degrades the quality of result by merging edges that are not planar.

In Figure 6.3, the effects of both ϵ_1 and ϵ_2 can be seen. Table 6.2 gives the numerical results of simplification with different ϵ 's. The original surface shown in Figure 6.3(a) consists of 32 triangles.

Figure	ϵ_1	ϵ_2	Group Count	Triangle Count
6.3(b)	0.1	0.0	3	17
6.3(c)	0.1	0.3	3	8
6.3(d)	0.7	0.0	1	4
6.3(e)	0.7	0.3	1	2

Table 6.2: Polygonal Simplification of a Surface

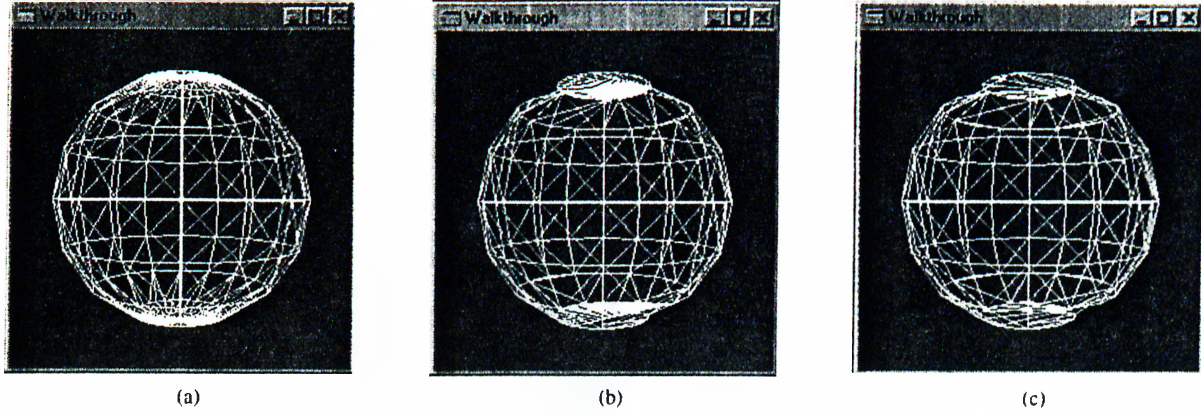


Figure 6.4: Polygonal Simplification of a Sphere

Figure	ϵ_1	ϵ_2	Triangle Count
6.4(b)	0.2	0.0	232
6.4(c)	0.2	0.3	189

Table 6.3: Polygonal Simplification of a Sphere

Polygonal simplification algorithm does not give good visual results on curved surfaces such as a sphere. The positioning of the faces that make a sphere are not appropriate for the algorithm, because of the large difference between normals of neighboring faces. Giving large ϵ_1 values for coplanar grouping solves the problem, however the quality of resulting models reduces. Figure 6.4 shows simplification of a sphere. The original sphere 6.4(a) has 416 triangles. Table 6.3 gives the numerical results of simplification.

For showing the effect of simplification on a real object, the column in Figure 6.5(a) with 3136 triangles is selected. Figure 6.5 shows the visual and Table 6.4 gives numerical results of the simplification.

Figure	ϵ_1	ϵ_2	Triangle Count
6.5(b)	0.1	0.0	1242
6.5(c)	0.1	0.3	415

Table 6.4: Polygonal Simplification of a Column

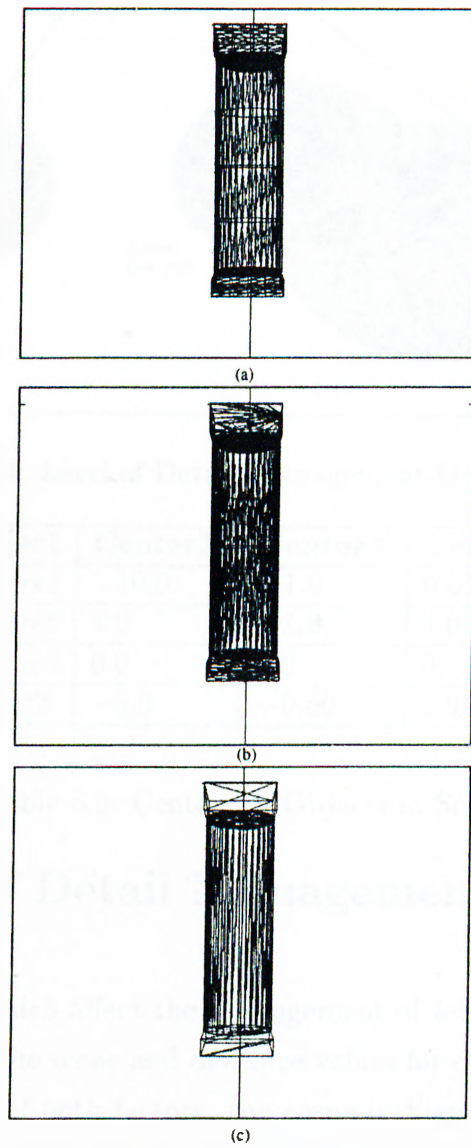


Figure 6.5: Polygonal Simplification of a Column

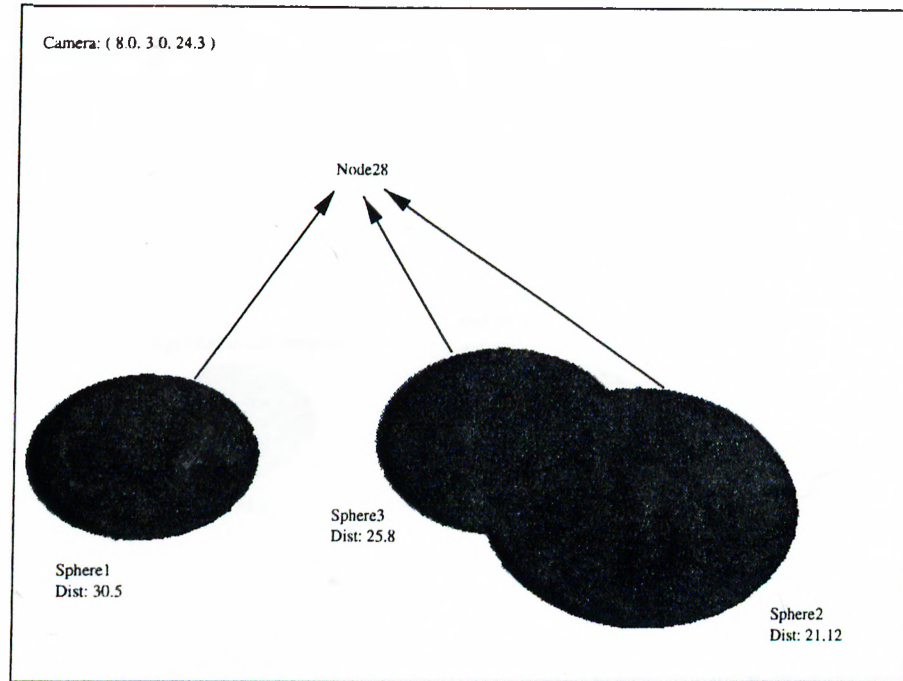


Figure 6.6: Level of Detail Management-Original Scene

Object	CenterX	CenterY	CenterZ
<i>Sphere1</i>	-10.0	-1.0	0.0
<i>Sphere2</i>	4.0	-1.0	4.0
<i>Sphere3</i>	0.0	0.0	0.0
<i>Node28</i>	-3.0	-0.50	2.0

Table 6.5: Centers of Objects in Scene

6.2 Level of Detail Management

The parameters which affect the management of level of detail are the hierarchy of objects in the scene and distance values for changing level of detail. To explain the effect of both factors, the scene in Figure 6.6 is selected. The spheres in the scene are identical except their textures and consist of 960 triangles. *Sphere2* and *sphere3* and *sphere1* are grouped together to form *node28*.

The Figures 6.7, 6.8, 6.9, 6.10, 6.11, 6.12 and 6.13 show level of detail management for different camera positions. The first change in level of detail occurs when the distance of object center to camera is larger than 40.0. If the distance of objects is larger than 80 bounding box representation of the object

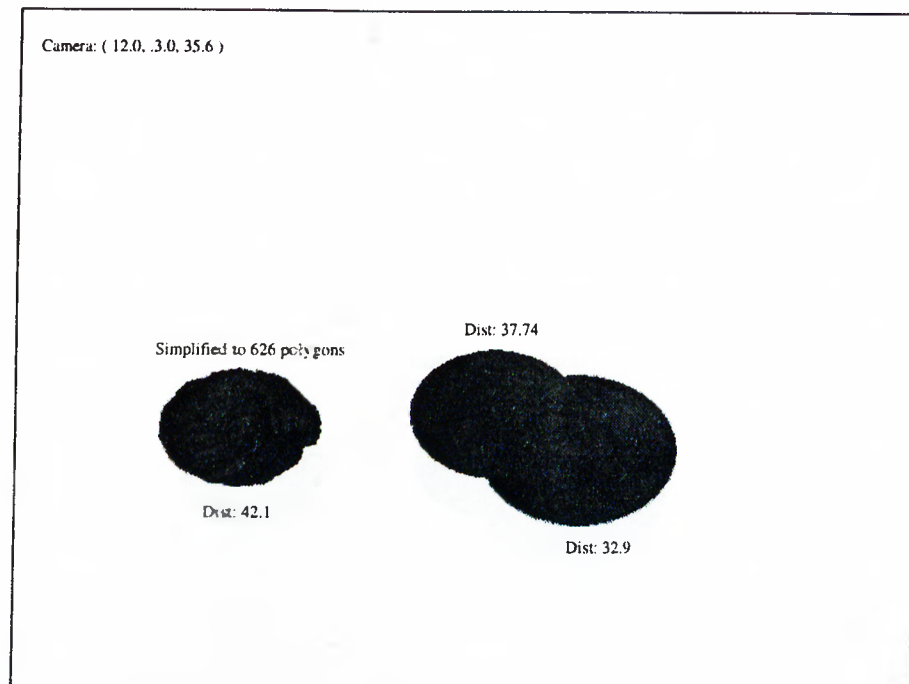


Figure 6.7: Level of Detail Management-1

is displayed. If an intermediate node's distance to camera is larger than 160, instead of child nodes, mixed-textured bounding box of the intermediate node is shown. The only intermediate node in example is *node28*. The camera position and distances of objects are shown on the figures. Figure 6.6 has 960x3 triangles whereas Figure 6.12 has only a bounding box.

6.3 The Walkthrough System

The system has three differences from a straightforward walkthrough application. By straightforward application, a system which directly draws all the polygons in the scene using OpenGL is meant. The walkthrough results of the system will be examined in these three aspects.

The first difference is explicit frustum culling. Although OpenGL makes automatic frustum culling, for faster elimination of vertices that are not visible, a frustum control based on bounding boxes is performed. The frustum is recalculated for each camera movement and each vertex of the bounding boxes

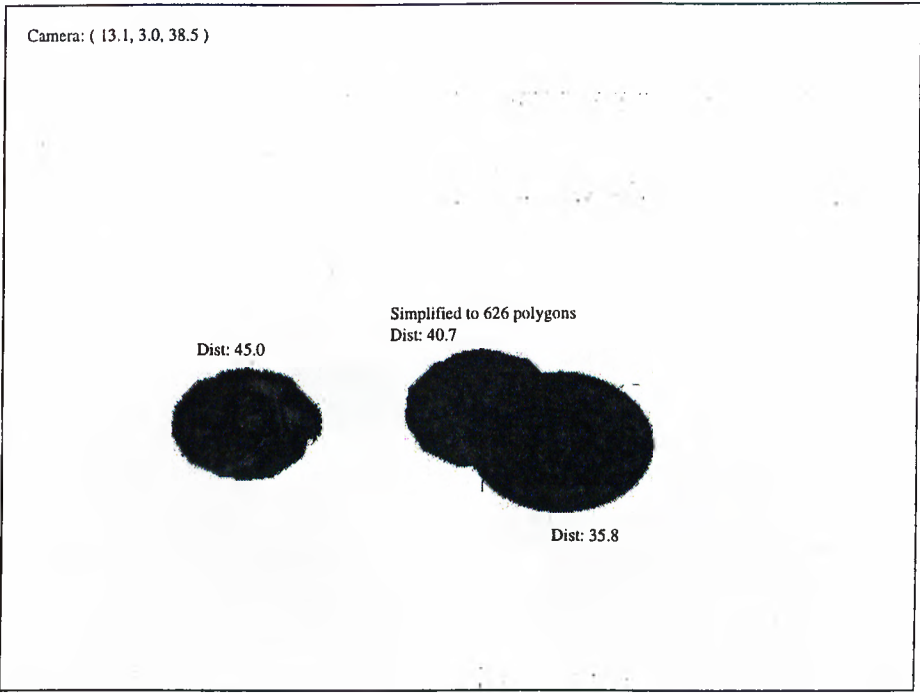


Figure 6.8: Level of Detail Management-2

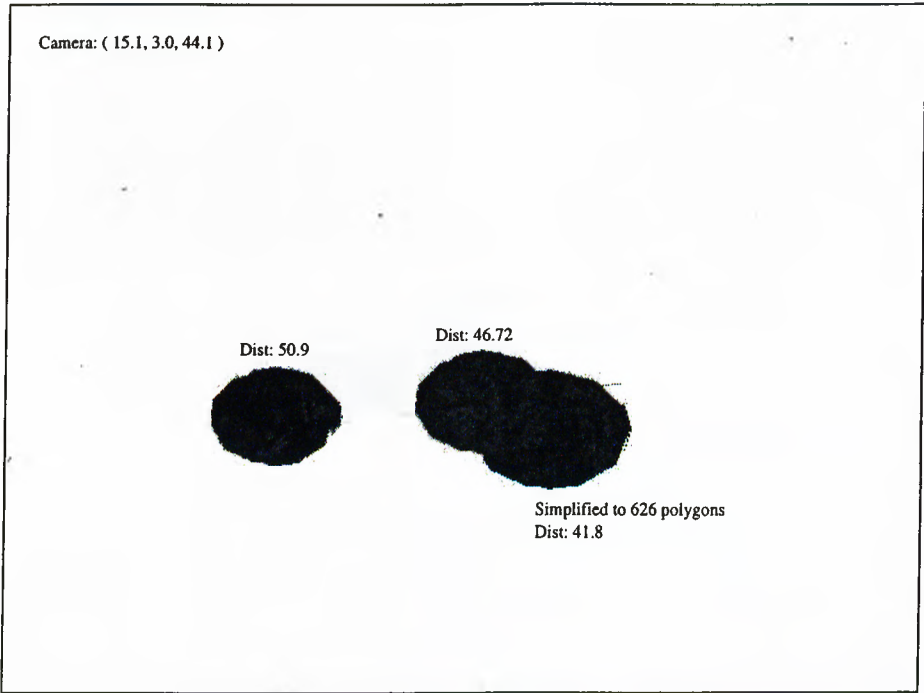


Figure 6.9: Level of Detail Management-3

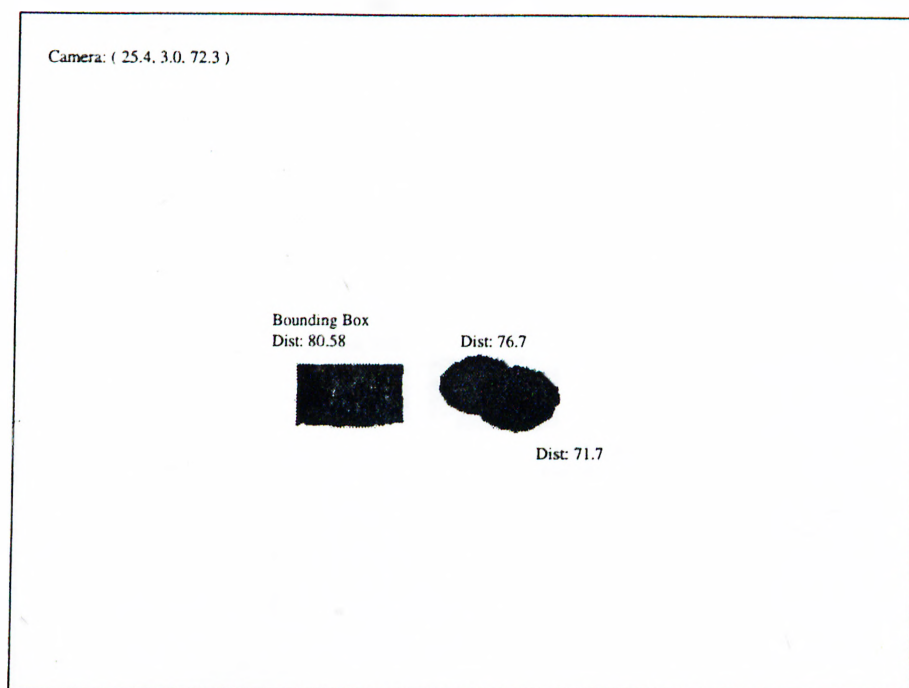


Figure 6.10: Level of Detail Management-4

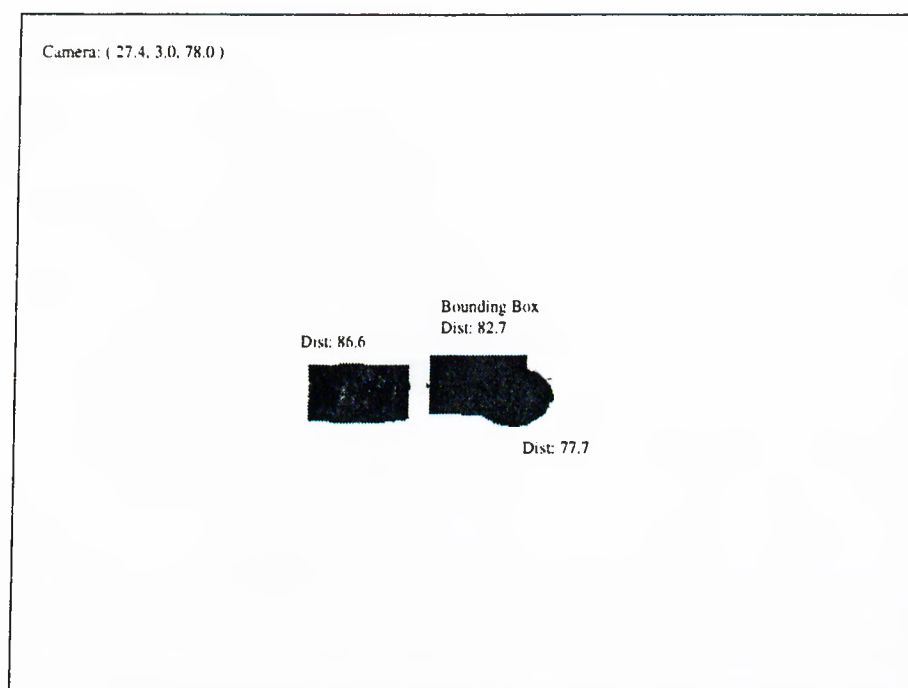


Figure 6.11: Level of Detail Management-5

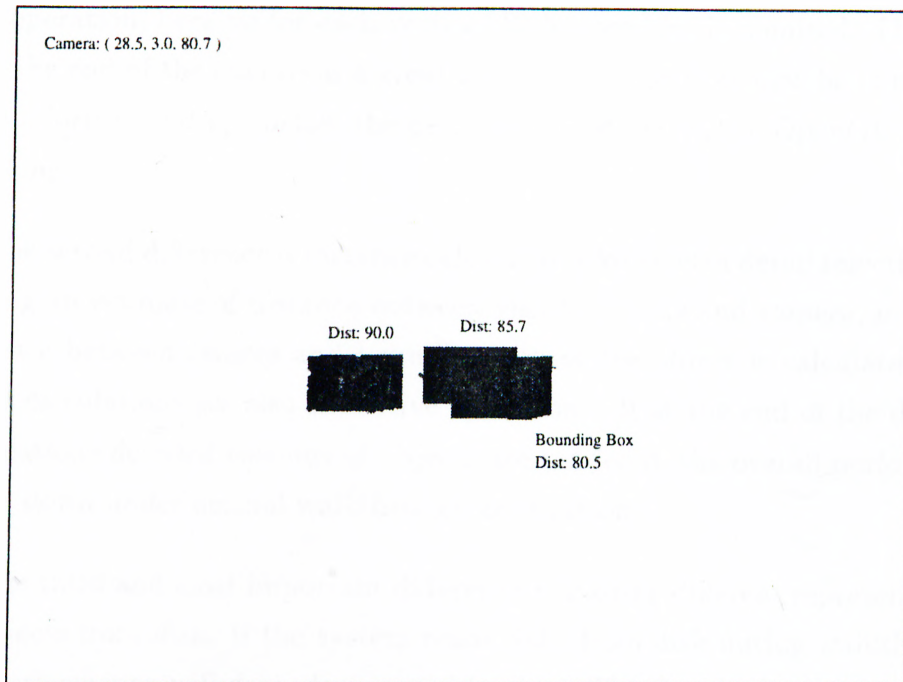


Figure 6.12: Level of Detail Management-6

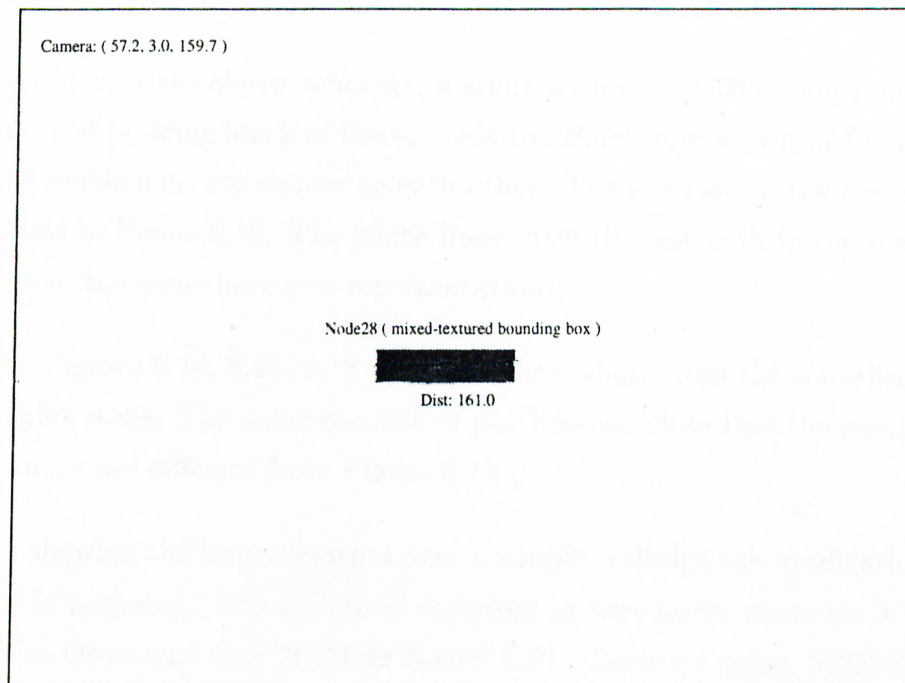


Figure 6.13: Level of Detail Management-7

of the nodes in the hierarchy are controlled for each new frame. It is an expensive operation, because for each vertex 3D dot-product is required. Therefore, if at the end of the operation a great number of objects cannot be eliminated, the performance drops below the performance of automatic OpenGL frustum checking.

The second difference is distance calculations for level of detail selection. For finding an estimate of distance between visible objects and camera, minimum distance between camera and bounding box of the object is calculated. Distance calculations are also expensive operations. If at the end of the distance calculations detailed versions of objects are rendered, the overall performance drops down under normal walkthrough applications.

The third and most important difference is loading different representations of objects from disk. If the system reads data from disk during walkthrough, the performance will drop down considerably. The double-memory-cache reduces the number of disk accesses for two-level-of-detail scenes. However if the number of representations for objects is more than two, for each representation change, a new model should be loaded. The number of memory caches can be increased, which will lead to memory shortage problems.

Considering the above concepts, a scene with over 250000 polygons is prepared. The building block of the scene is the Parthenon shown in Figure 6.14. All the parthenons are similar to each other. The top-view of the test scene is displayed in Figure 6.15. The white lines show the test path in the scene. All objects in the scene have two representations.

The Figures 6.16, 6.17, 6.18 and 6.19 show shots from the walkthrough of a complex scene. The scene consists of parthenons. Note that the positions of parthenons are different from Figure 6.15.

For showing the improvements over a simple walkthrough application, Table 6.6 is prepared. The system is executed in wire-frame mode on a Silicon Graphics Onyx with four 200MHz R4400 CPU, Reality Engine, 512MB RAM and 64MB texture memory. The simple walkthrough system required between 1.48 and 1.53 seconds per frame depending on the position of camera. Table 6.6

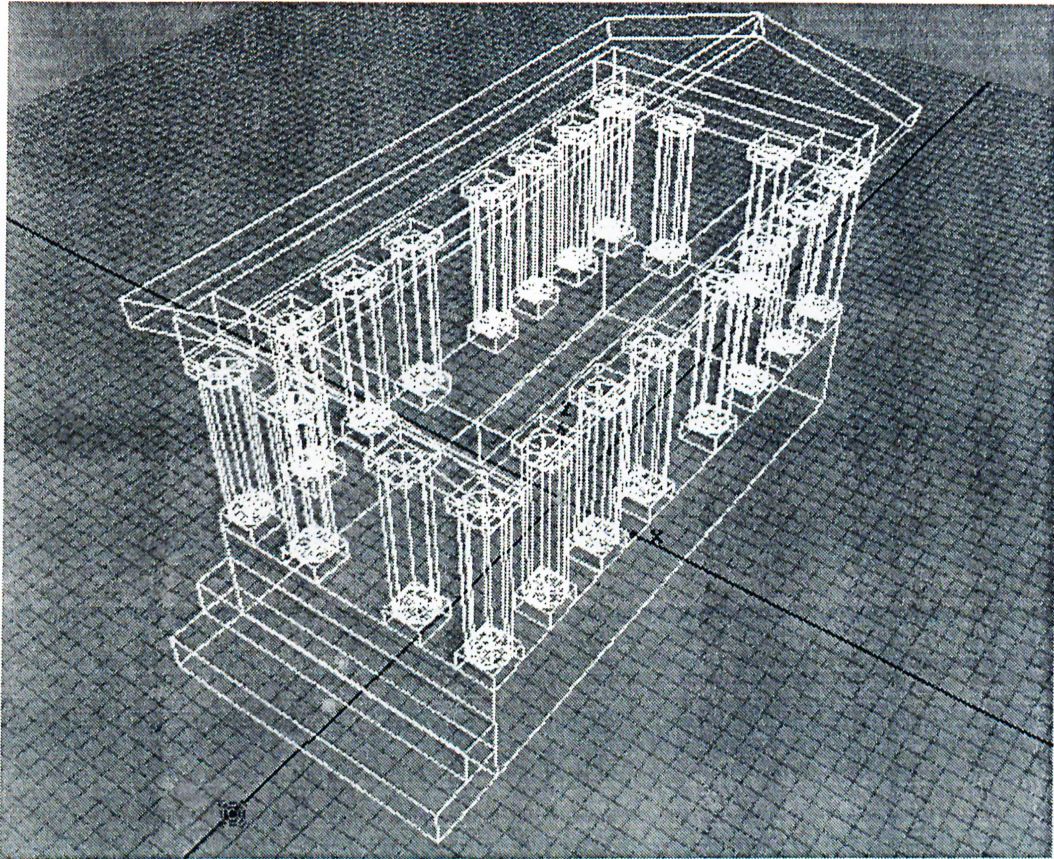


Figure 6.14: The Parthenon

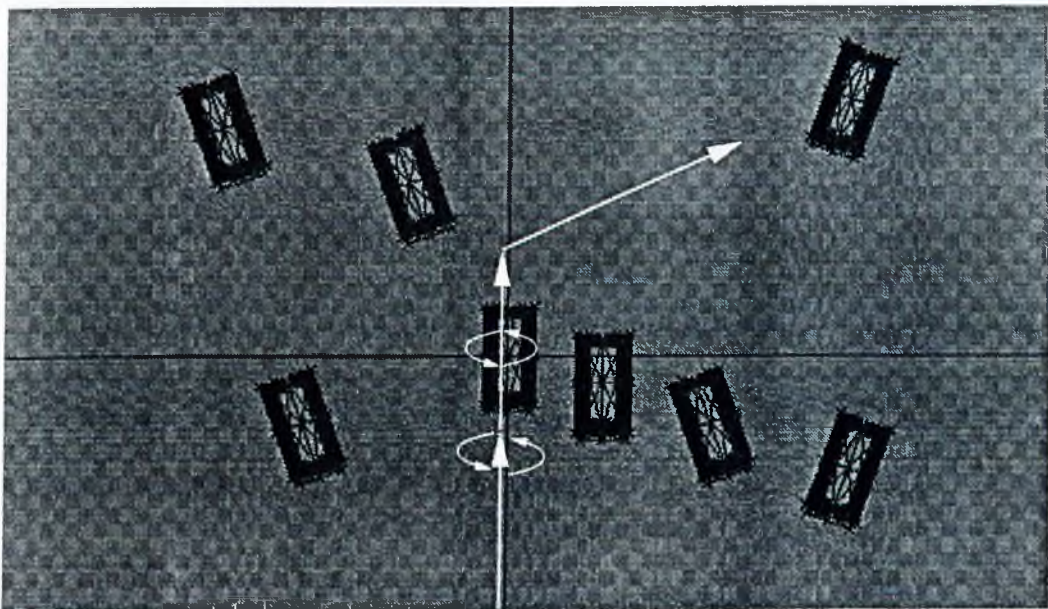


Figure 6.15: The Test Scene

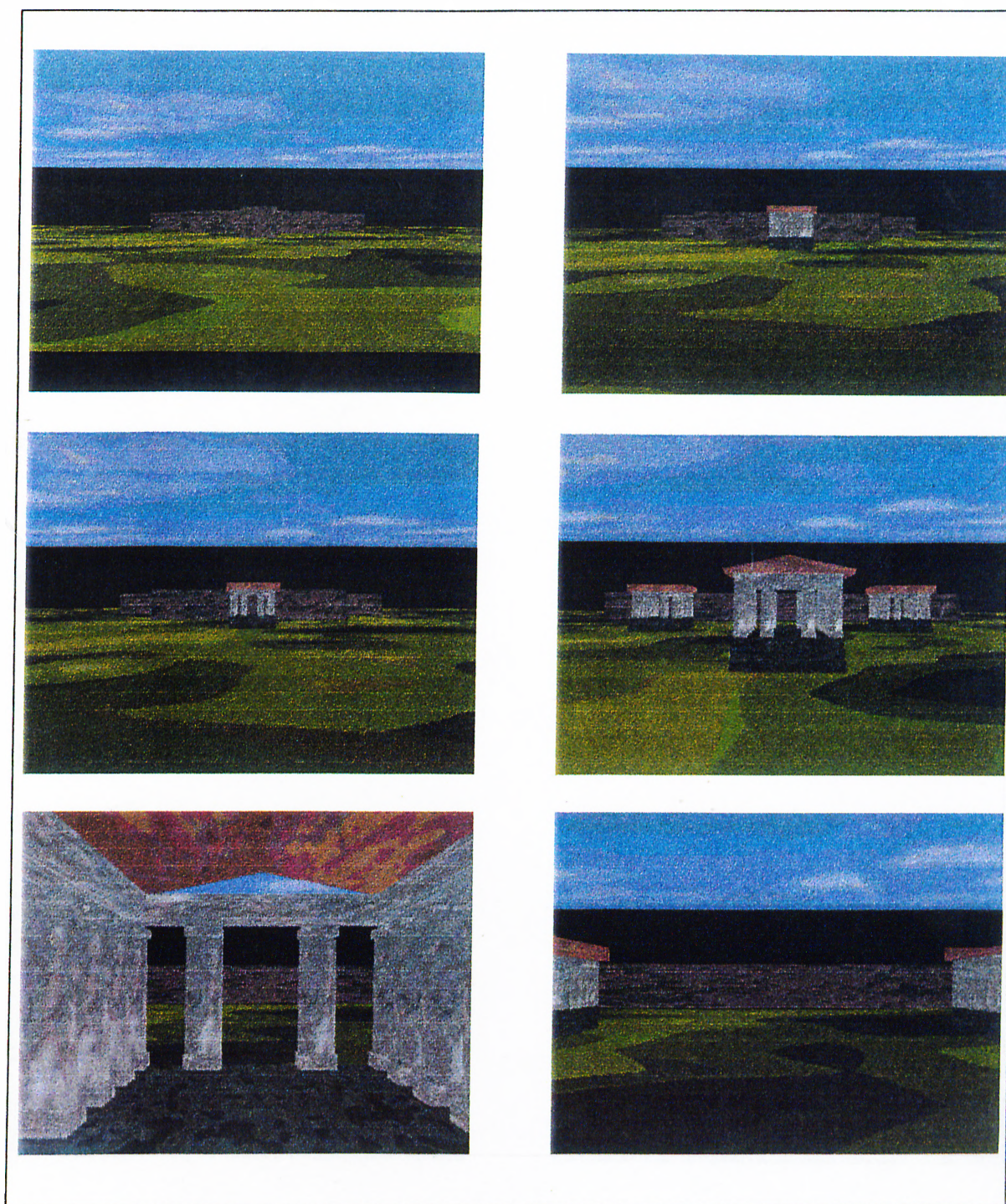


Figure 6.16: The Walkthrough-1

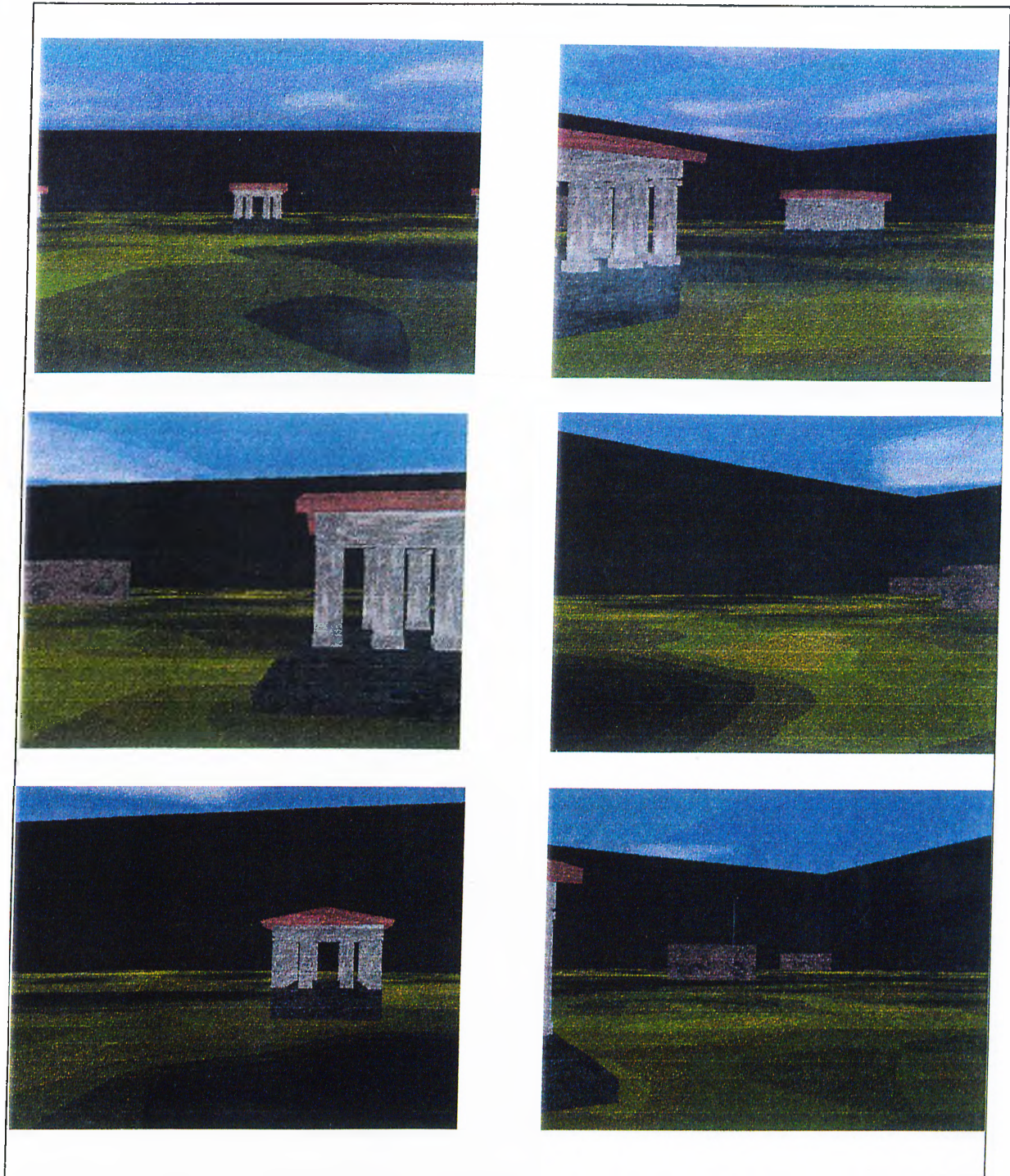


Figure 6.17: The Walkthrough-2

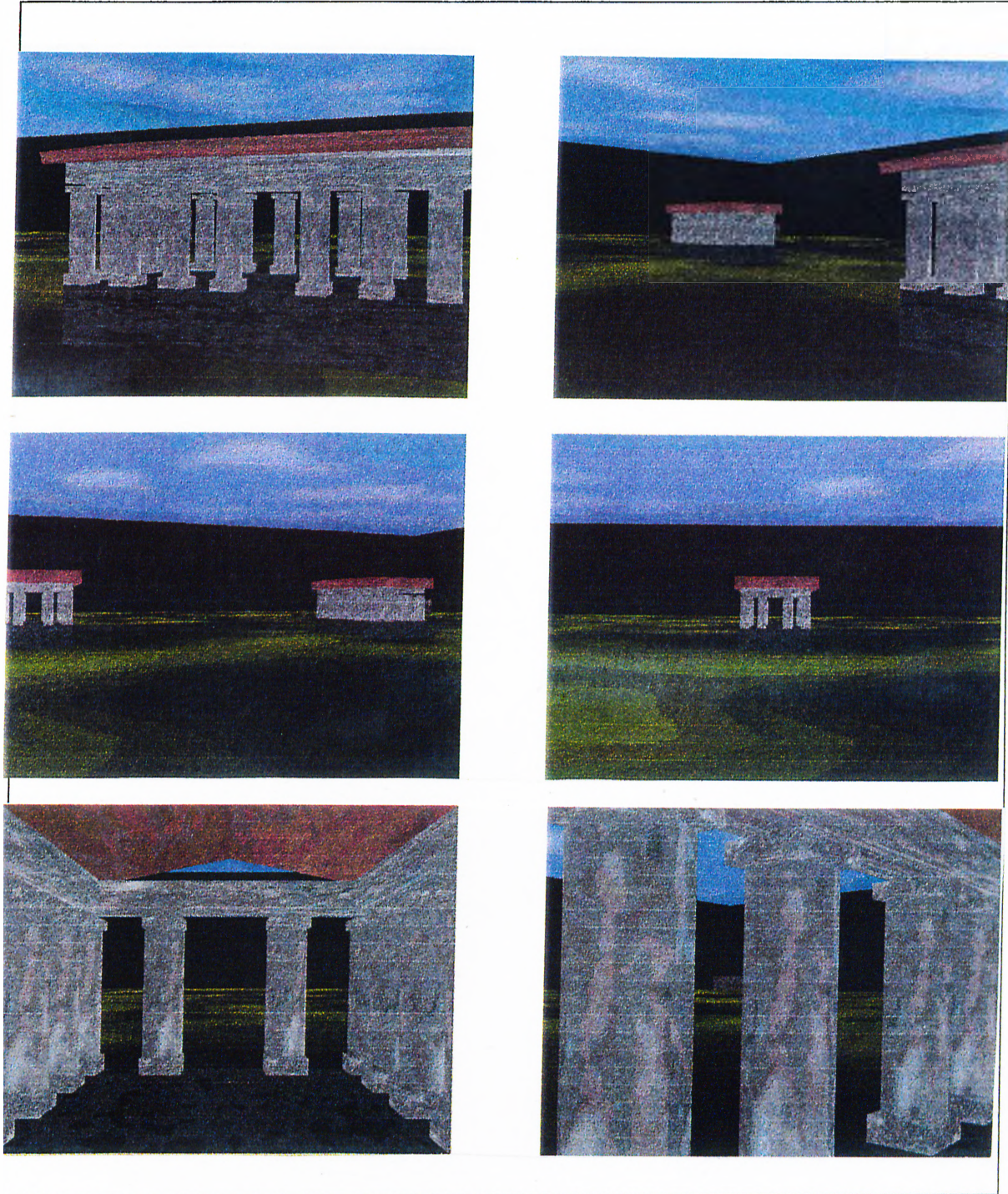


Figure 6.18: The Walkthrough-3

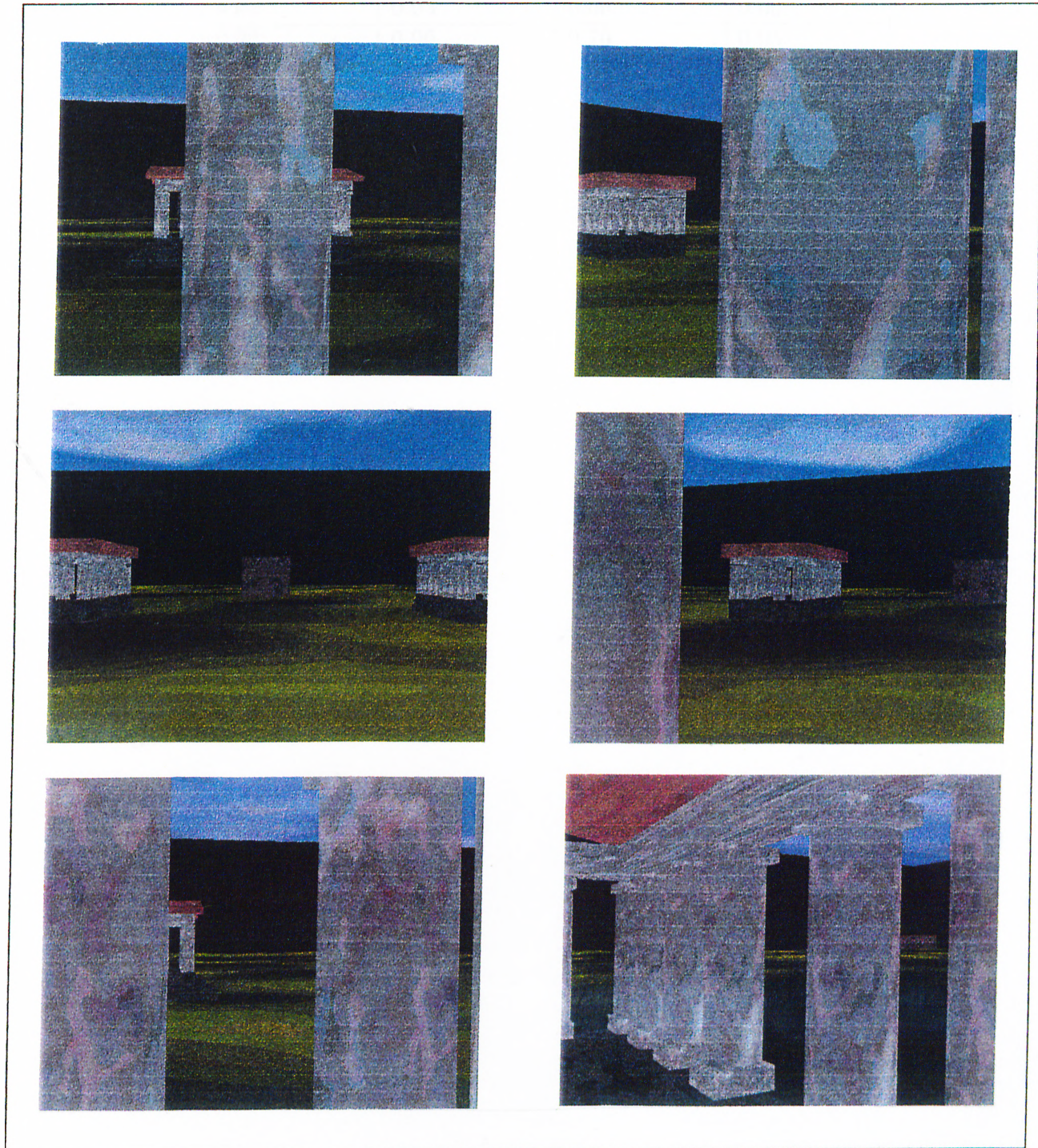


Figure 6.19: The Walkthrough-4

Frustum (s)	Distance (s)	Reading (s)	Display (s)
0.00	0.00	0.00	0.00
0.02	0.03	0.00	0.00
0.02	0.06	0.70	0.05
0.08	0.10	1.82	0.34
0.06	0.08	1.77	0.55
0.06	0.08	0.00	0.45
0.02	0.05	0.00	0.48

Table 6.6: Polygonal Simplification of a Surface

gives the execution times of each step in the system for different camera positions on the path.

The first two rows shows the required times when the camera is far from the scene. Only the higher level nodes of object trees in hierarchy list are tested. Since they are all far away from camera their bounding boxes are displayed. Therefore all values are near zero. As the camera gets closer, lower level nodes are also tested and for some objects simpler representations are loaded from disk to cache (row 3). Rows 4 and 5 show the worst cases of the system. Too many disk accesses are required for these shots. After most of the caches are filled, the execution times remain constant as can be observed from last two rows.

The results of the table show that the display times are improved at worst more than three times compared to the simple system. Depending on the distance of camera, much better results can be obtained. The disadvantage of the system is disk-access times. When many object's caches need to be loaded, the system's performance gets worse than the simple application. One possible solution to that is to load all representation to caches if there is no memory shortage.

Chapter 7

Conclusions and Future Research Areas

In this chapter, a summary of this work and directions for future work will be given.

7.1 Conclusions

In this work, a survey of methods that enable handling of huge geometric models in interactive frame-rates is prepared. A system for walkthrough in complex models is developed using the ideas of the methods.

The need for interactive frame-rates in simulation, CAD/CAM, architecture and entertainment applications led many researchers to this area. The models for such applications exceed the limits of current graphics workstations.

Researchers make use of limitations of human visual system for reducing the number of primitives and the hardware properties of state-of-the-art graphics systems for optimizing the performance.

Methods for walkthrough range from simple ideas to very complex algorithms. Reducing the number of polygons to render without affecting the

image quality is the main idea of these methods. For reducing the number of primitives either geometric simplification algorithms or visibility and occlusion culling algorithms are used. Such algorithms reduce the number of polygons per frame to manageable numbers.

The system developed in this work takes as input triangulated hierarchical models in OSTF format. The object hierarchy is built first in memory. After that simplified versions of objects in the scene are created and saved for later use.

The simplification algorithm is based on removing nearly coplanar triangles from the model. Triangles with nearly same normals are grouped together. For each group a boundary polygon is found. Planar edges are removed from boundary polygon. The remaining vertices of the boundary polygon are triangulated.

The simplification ratio depends on two thresholds. ϵ_1 for determining which triangles to put to same group and ϵ_2 for removing edges from boundary polygon. By changing the values of ϵ_1 and ϵ_2 , different representations with different simplification ratios can be obtained.

The simplification algorithm is very successful on objects with planar surfaces. It reduces the number of triangles considerably without affecting the image quality.

For walkthrough, a virtual camera and a viewing frustum is defined. The user moves the camera inside the model as if he walks in the model. As the camera moves, the nodes in the hierarchy are traversed. Nodes that are out of viewing frustum are discarded. Viewing frustum calculations are based on bounding boxes. For near objects detailed representations are displayed, whereas for far objects simplified versions are shown.

To reduce the model-loading overhead, double-memory cache for storing recently used representations of an object are used.

7.2 Future Research Areas

Since the topic of the thesis is a very popular research area, there are a lot of future directions to improve the performance of the system, both in terms of simplification and frame-rate.

The system has a modular structure. Each module has a certain interface with other modules. Using the advantage of this, any geometric simplification algorithm can replace the current simplification algorithm.

The intermediate nodes in the object hierarchy are represented by textured cubes. The textures of the cubes are obtained by mixing the textures of lower level nodes. Intermediate nodes can be represented by more accurate representations.

The system currently performs only visibility culling. For architectural models, occlusion culling algorithms are very important for reducing the number of vertices. An occlusion culling method can be adopted to the system.

For reducing the number of mode changes when rendering the model, primitives with the same textures can first be grouped together. After that with minimal number of mode changes, primitives can be displayed.

Current double-memory cache system solves the problem of model loading for two-level of details. The caching system can be optimized for more levels of detail by increasing the number of caches.

Appendix A

Input Model

The models used in the system are prepared using Alias/Wavefront PowerAnimator which is an advanced modelling software developed by Silicon Graphics. The program is capable of exporting models in several file formats. Among those file formats OSTF (Object Separated Triangle Format) is selected. OSTF is very suitable for the system and is easy to handle.

In OSTF files, geometric data is stored hierarchically. For each object in the model, there is a section in the file. Each section contains data of the triangles of the objects or data of the child objects of the object. The data stored for each object is given in figure A.1.

If the *count* field of an object is equal to zero, that means the object contains other objects. The definitions of sub-objects are given inside the container object. This structure is very suitable for the system because of the hierarchy. Container objects refer to intermediate nodes and sub-objects refer to the leaf nodes of the object tree in the system.

Per vertex data (normal, coordinate, texture and color) in OSTF also exactly match to the data required per vertex by the system. For displaying a vertex, sending the vertex data without any conversion to OpenGL is enough.

START_TAG	Marks start of an object.
name	Name of the object.
count	Number of triangles.
texturename	Name of the texture file.
nx ny nz	Normal of vertex1.
fx fy fz	Coordinates of vertex1.
cr cg cb	Color of vertex1.
u v	Texture coordinates of vertex1.
nx ny nz	Normal of vertex2.
fx fy fz	Coordinates of vertex2.
cr cg cb	Color of vertex2.
u v	Texture coordinates of vertex2.
nx ny nz	Normal of vertex3.
fx fy fz	Coordinates of vertex3.
cr cg cb	Color of vertex3.
u v	Texture coordinates of vertex3.
etc...	
END_TAG	Marks the end of the object.

Figure A.1: Representation of an Object in OSTF.

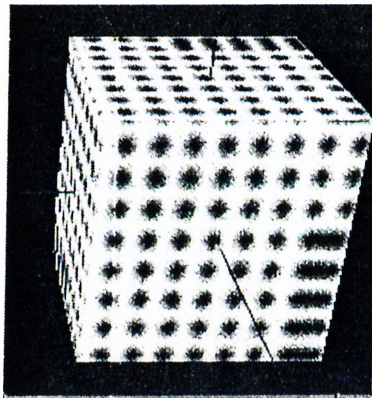


Figure A.2: The Cube

The representation of the cube in figure A.2 is given as an example in figure A.3. Each face of the cube is given as a sub-object and each face consists of two triangles.


```
START_TAG
Cube
0
START_TAG
    face#1
    2
    line.tex
    ( data of 2 triangles )
END_TAG
START_TAG
    face#2
    2
    line.tex
    ( data of 2 triangles )
END_TAG
START_TAG
    face#3
    2
    line.tex
    ( data of 2 triangles )
END_TAG
START_TAG
    face#4
    2
    line.tex
    ( data of 2 triangles )
END_TAG
START_TAG
    face#5
    2
    line.tex
    ( data of 2 triangles )
END_TAG
START_TAG
    face#6
    2
    line.tex
    ( data of 2 triangles )
END_TAG
END_TAG
```

Figure A.3: Representation of the Cube

Appendix B

OpenGL

In the earlier days of computer graphics, each person implemented his own graphics library for rendering a model. Even the simplest transformation calculations were coded for every new project. As the concepts of computer graphics get more mature, people started to write graphics libraries for their own usage. Throughout time those libraries get complex and portable. At the end of this trend companies like Microsoft and Silicon Graphics started research on standardizing graphics libraries. OpenGL is developed by SGI and Direct3D is developed by Microsoft.

In this thesis, OpenGL is selected for implementation. OpenGL is a software interface to graphics hardware. The interface consists of about 120 distinct commands, which are used for specifying objects and operations needed to produce interactive three-dimensional applications [37]. The two reasons for selecting OpenGL for implementation are; its wider usage and its portability. An OpenGL code can be compiled on graphics hardwares for which an OpenGL library exists. Such platforms include Sun and SGI workstations and PCs.

Since OpenGL is a well known library, only the properties that are used for optimizing the system will be explained. More detailed explanations can be found in [37] and [38].

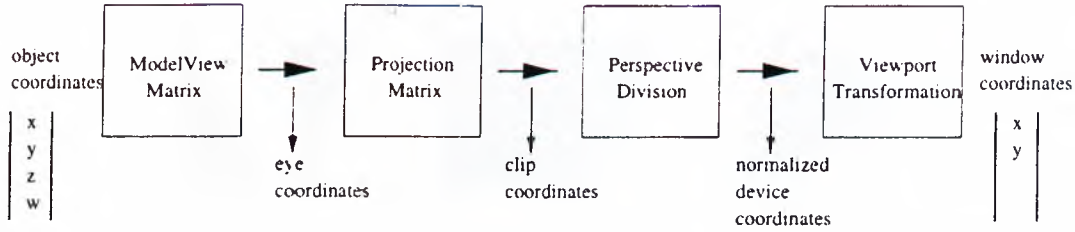


Figure B.1: Stages of Vertex Transformation in OpenGL

B.1 Transformation Calculations

OpenGL uses two matrices for transformation calculations: modelview matrix and projection matrix.

Modelview matrix is used to set the position and orientation of the scene. Projection matrix is used to set the type of projection and to create a viewing frustum for automatic culling. Perspective and ortographic projections are supported by OpenGL. After two matrices are set, each vertex is processed as shown in figure B.1.

Both matrices can be changed any time. For example, for rotating the objects in the scene, multiplying the modelview matrix with the rotation matrix is enough.

OpenGL stores matrices in the form of two stacks. One stack is for projection matrix and the other is for modelview matrix. For each operation a target stack must be specified.

For this work, perspective projection is selected because it generates realistic looking images. Projection matrix of OpenGL is used to set the parameters of perspective projection. The angle of field of view is the only parameter for perspective projection. Also back and front planes of viewing frustum are combined with projection matrix to enable automatic frustum culling of OpenGL.

Modelview matrix is used for camera movement. As a result of each movement, a rotation and translation matrix is formed and multiplied with the modelview matrix to form a new modelview matrix.

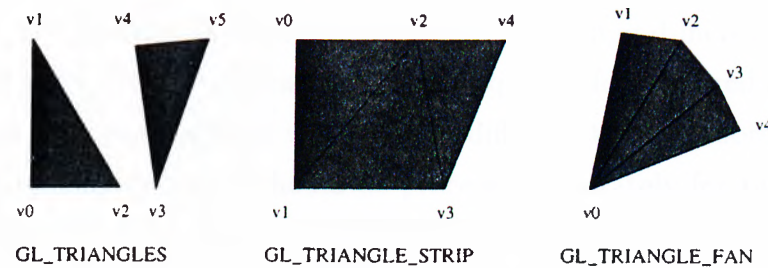


Figure B.2: Triangle Commands in OpenGL

Once both matrices are set for a camera movement, each vertex in the scene is sent to the process shown in figure B.1. As the result, new shot for the new camera position and angle of field of view is obtained on the screen.

B.2 Triangle Fans and Triangle Strips

OpenGL has commands for drawing geometric primitives such as lines, polygons, triangles and quads. Triangle-drawing commands are used in the implementation to optimize the rendering time.

OpenGL has three types of triangle commands. The first one is used for rendering separate triangles as seen at the left of figure B.2. For each triangle, all the three vertices are processed by the graphics engine. Therefore, the first command is the slowest among other triangle commands.

The second triangle command is optimized for drawing triangle strips as shown at the middle of figure B.2. It processes vertices in the following order: $(v0, v1, v2)$, $(v2, v1, v3)$ and $(v2, v3, v4)$. Since there are shared vertices between triangles, the number of processed vertices reduces.

The last triangle command is optimized for drawing triangle fans as shown at the right of figure B.2. It processes vertices in the following order: $(v0, v1, v2)$, $(v0, v2, v3)$ and $(v0, v3, v4)$. The command takes the advantage of shared vertices, therefore it reduces number of processed vertices.

In the system, triangulated simplified versions of objects consists of triangles fans as explained in 5.2.2. Triangle-drawing command for triangle fans is used

for the implementation. Although the use of the command increases the performance of the system, it degrades the image quality for textured and lighted scenes. This is because shared vertices have different texture coordinates and normal vectors. Each vertex should be processed separately for textured and lighted scenes.

Bibliography

- [1] B. Cabral, N. Greene, J. Rossignac and T. Funkhouser. Interactive Walkthrough of Large Geometric Databases. *SIGGRAPH'96 Course Notes*, Course 35, August 1996.
- [2] F.P. Brooks Walkthrough - A Dynamic Graphics System for Simulating Virtual Buildings. *Proceeding of Workshop on Interactive 3D Graphics (Chapel Hill, NC, 23-24 October 1986)*, pp. 9-21.
- [3] K. Chiu and P. Shirley. Rendering. Complexity and Perception. *Proceedings of the 5th Eurographics Rendering Workshop (Darmstadt, 1994)*.
- [4] J.D. Foley, A. van Dam, S.K. Feiner and J.F. Hughes. *Computer Graphics: Principals and Practice*. Addison Wesley, second edition, 1992.
- [5] M. Jones. Designing Real-Time Graphics for Entertainment. *SIGGRAPH 94' Course Notes*. Course, August 1994.
- [6] G. Wyszecki and W.S. Stiles. *Color Science*. John Wiley and Sons, New York, 1982.
- [7] P. Padmos and M. Milders. Checklist for outside-world images of simulators. *Proceedings of ITEC'92, International Training Equipment Conference and Exhibition (Luxembourg, 7-9 April 1992)*, pp. 2-14.
- [8] P. Buser and M. Imbert. *Vision* MIT Press, Cambridge, Massachusetts, 1992.

- [9] F.M. Cardullo. Virtual systems lags: The problem, the cause, the cure. *Proceedings of the 1990 Image V Conference (Phoenix, 19-22 June 1990)*, Image Society, 1990. pp. 31-42.
- [10] Airline Simulator Qualification. Advisory Circular 120-40B, Federal Aviation Administration, May 1989.
- [11] K. Akeley. Reality Engine Graphics. *Proceedings of SIGGRAPH'93 (California, 1-6 August 1993)*, pp. 109-116.
- [12] SGI Onyx Owner's Manual.
- [13] J.F. Blinn and M.E. Newell. Texture and Reflection in Computer Generated Images. *CACM, October, 1976*, pp. 542-547.
- [14] P.S Heckbert. Survey of Texture Mapping. *Computer Graphics and Applications, November 1986*, pp. 56-67.
- [15] M. Deering. Geometry Compression. *Proceedings of SIGGRAPH'95 (August 1995)*, pp. 13-20.
- [16] H. Hoppe. Progressive Meshes. *Proceedings of SIGGRAPH'96 (August 1996)*.
- [17] P.S. Heckbert and M. Garland. Multiresolution Modeling for Fast Rendering. *Proceedings of Interface '94 (Banff, Canada, May, 1994)*, pp. 43-50.
- [18] P. Hinker and C. Hansen. Geometric Optimization. *Proceedings of Visualization (October, 1993)*, pp. 189-195.
- [19] B. Hanamann. A Data Reduction Scheme for Triangulated Surfaces. *Computer Aided Design*, April 1994, pp. 197-214.
- [20] G. Turk. Re-Tiling Polygonal Surfaces. *Proceedings of SIGGRAPH'92 (July 1992)*, pp. 55-64.
- [21] W.J. Schroeder, A.Z. Jonathan and E.L. Lorensen. Decimation of Triangle Meshes. *Proceedings of SIGGRAPH'92 (July 1992)*, pp. 65-70.

- [22] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald and W. Stuetzle. Mesh Optimization. *Proceedings of SIGGRAPH'93 (California, August 1993)*, pp. 19-26.
- [23] A. Varshney. Hierarchical Geometric Approximations. PhD Thesis. Department of Computer Science, University of North Carolina at Chapel Hill, 1994.
- [24] A.D. Kalvin and H.T Russell. Surfaces: Polygonal Mesh Simplification with Bounded Error. Technical Report RC 19808, IBM Research Division. T. J. Watson Research Center, Yorktown Heights, NY 10958. 1994.
- [25] J.R. Rossignac and P. Borrel. Multi-resolution 3D Approximations for Rendering Complex Scenes. Technical Report RC 17697. IBM Research Division, T. J. Watson Research Center. Yorktown Heights, NY 10958, 1992.
- [26] T.A. Funkhouser and C.H. Sequin. Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments. *Proceedings of SIGGRAPH'93 (California, August 1993)*. pp. 247-254.
- [27] S. Belblidia, J.P. Perrin and J.C. Paul. Multi-Resolution Rendering of Architectural Models. July, 1995.
- [28] P.W.C. Macial. Interactive Rendering of Complex 3D Models in Pipelined Graphics Architectures. Technical Report, Department of Computer Science. Indiana University, May 1994.
- [29] B. Chamberlain, T. DeRose, D. Lischinski, D. Salesin, J. Snyder. Fast Rendering of Complex Environments Using a Spatial Hierarchy.
- [30] J. Shade, D. Lischinski, D. Salesin, T. DeRose and J. Snyder. Hierarchical Image Caching for Accelerated Walkthroughs of Complex Environments. Technical Report UW-CSE-96-01-06, Microsoft Research.

- [31] S.J. Teller. Visibility Computations in Denseley Occluded Polygonal Environments. Ph.D. Thesis, Computer Science Division, University of California, Berkeley, 1992.
- [32] J.H. Clark. Hierarchical Geometric Models for Visible Surface Algorithms. *Communications of the ACM*, October 1976, pp. 547-554.
- [33] S.J. Teller and C.H. Sequin. Visibility Preprocessing for Interactive Walkthroughs. *Proceedings of SIGGRAPH'91(August 1991)*, pp. 61-69.
- [34] N. Greene, M. Kass and G. Miller. Hierarchical Z-Buffer Visibility. *Proceedings of SIGGRAPH'93 (California, August 1993)*, pp. 231-236.
- [35] P. Burget and D. Gillies. *Interactive Computer Graphics*. Addison Wesley, 1989.
- [36] D. Hearn and M.P. Baker. *Computer Graphics*. Prentice Hall, 1997.
- [37] T. Davis and M. Woo. *OpenGL Programming Guide: The Official Guide to Learning OpenGL*. Addison Wesley, 1993.
- [38] OpenGL Architecture Review Board. OpenGL Reference Manual: The Official Reference Document for OpenGL. Addison Wesley, 1992.
- [39] R. Seidel. A Simple and Fast Incremental Randomized Algorithm for Computing Trapezoidal Decompositions and Triangulating Polygons. *Computational Geometry: Theory and Applications (January, 1991)*, pp. 51-64.
- [40] C. Erikson. Polygonal Simplification: An Overview. Technical Report TR96-016, Department of Computer Science, UNC-Chapel Hill, 1996.
- [41] M.M. Chow. Optimized Geometry Compression for Real-time Rendering. To appear in *Proceedings of Visualization (1997)*.
- [42] SGI Alias Menu Book.

- [43] R.M. Bartels. *An Introduction to Splines for Use in Computer Graphics and Geometric Modelling*. M. Kaufmann Publishers, Los Altos, Cali, 1988.
- [44] B.A. Barsky. *Computer Graphics and Geometric Modelling Using Beta-splines*. Springer-Verlag, Tokyo, New York, 1987.
- [45] F.P. Preparata. *Computational Geometry: An Introduction*. Springer-Verlag, New York, 1985.