Issues involved in Real-Time Rendering of Virtual Environments

Priya Malhotra

Thesis submitted to the Faculty of Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

Master of Science in Architecture

July 19, 2002
College of Architecture and Urban Studies
Blacksburg, Virginia

Keywords: Real-time rendering, virtual environments, photo-realism, optimization

ISSUES INVOLVED IN REAL-TIME RENDERING OF VIRTUAL ENVIRONMENTS

Priya Malhotra

ABSTRACT

This thesis explores the issues involved in modeling and rendering virtual environments with special emphasis on photo-realistic visualization and optimizing models for real-time applications.

Architectural walk-through systems are expected to give convincingly realistic interactive visualizations of complex virtual environments (Brooks, 1986). This pursued high degree impression of reality enhanced by interactivity, leads the user into a state of immersion, or the suspension of disbelief. The use of these systems ranges from virtual prototyping of building designs, stage and set lighting design, and architectural design reviews where the demands for greater realism and higher frame rates are always increasing. Until recently, the major focus has been on quickly rendering a complex model, rather than on photo-realism. The primary goal was reducing the number of graphics primitives rendered per frame without noticeably degrading image quality.

The aim of this research is to study some of the real-time rendering and illumination techniques, bringing out the limitations and advantages of each. In addition the study investigates the extent of inclusion of standard 3 Dimensional modeling packages in the methodology pipeline, providing architects and designers with some guidelines for photo-realistic visualization and real-time simulation of their models.

This is demonstrated through an example model of Tadao Ando's Church on the Water. A 3D photo-realistic reconstruction and real-time simulation is attempted, using widely available standard tools**. The aim is to develop a methodology for building a compelling, interactive and highly realistic virtual representation**. The whole methodology is based not on proprietary commercial 3D game engines, but on international open standard programming languages and API's, while leaving the user to freely select and use his/her 3D character-modeling package of choice. However, several shortcomings in both hardware and software became apparent. These are described, and a number of recommendations are provided.

ACKNOWLEDGEMENTS

This work would not have been possible without the sincere support and encouragement of many...

I want to express my gratitude to everybody who left 'footprints' in my life. My work would not have been the same without the inspiration and influence of all of you!

I want to express special thanks

...to my family for their support, optimism and positive attitude. I owe them my happiness, trust in life and belief in myself.

...to John Kelso, Fernando das Neves, Joe Gabbard, Jeremy Eccles, Ted Boardman, Dr.Carolina Cruz Niera, Prof. Nadia Magnetat-Thalmann, George Papagiannakis, Dace Campbell, Manu Sporny and many more, for their invaluable insights and painstaking explanations.

...to Matthew Campbell for working with me on this and developing the application that made the real-time visualization a reality.

...finally to my committee members, Prof. Dennis Jones, Dr. R.P Schubert and Jason Lockhart, for all their inspiration and insights they shared along the way. Every one of them was individually supportive of my specific needs and questions during the process. Looking back, I especially enjoyed the encouragement they offered for my ongoing struggles and efforts in 'bringing the computers to their knees'. I owe my committee members the discovery of the non-linear qualities of my work and hence the continuation of my 'thesis'.

The supposed end already became a beginning...

DEDICATION

I dedicate this work to my father…..'my darling angel'.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

1.1 Rationale

Proponents of virtual reality (VR) believe in the idea that architecture is one of the many professions that can benefit and grow with the development of virtual reality technologies. Indeed, VR can aid the profession of architecture in many ways, from enabling “walk-through” presentations to providing a revolutionary medium for architectural design, allowing the design and construction of spatial models which can be simulated and experienced virtually. These worlds, or environments, can describe real or imaginary spaces, architectural design proposals, lost historical structures, urban landscapes, and non-physical virtual locales.

Architecture will also be affected by virtual reality on another level, as it and other simulation technologies become more integrated into our society. Virtual architecture, the design of three-dimensional environments for inhabitation in virtual reality, will soon be realized on a world-wide scale and may even come to replace much of what we know as the architectural representations of today.

In recent years, architectural walk-throughs have surfaced in the design profession as a presentation and design tool. By “walking” through computer simulations of a design, architects and clients alike have been able to visualize and evaluate designs and clarify design intentions long before the project is built in the physical world. The idea that we can now “inhabit” our designs in the computer, even if today only for a few minutes at a time, is an important one, and will be influential in the nature of society in the next millennium.

One of the major problems in a VR walk-through lies in predicting where the user will move in the subsequent instance. It is not obvious how this problem can be resolved. Cracking this problem would yield many benefits, such as the ability to focus the rendering of a scene to a specific area more quickly rather than rendering all other scenes in which the user might not be interested.

VR, however, does not merely promise a new method for presentation and visualization of “completed” design projects. VR will also allow designers and planners to visualize various factors and constraints in design that today are only available in text media. Building codes, zoning ordinances, technical data and properties (lighting, acoustics, HVAC, etc.), and other such design constraints and conditions will become available to designers in intuitive and easy-to-understand visual formats. As such advancements are made in the three-dimensional representations of abstract information, designers will be able to more fully concentrate on the content and quality of their designs.

In its current state, VR technology is in its infancy. The interfaces are clumsy, the concepts are immature, and many of the promises that proponents of VR made are, as yet, just promises. As the technology develops, the interfaces will become less clumsy, and virtual reality will achieve its goal: to become a transparent medium of communication, which will provide total immersion of the senses into an alternate “reality.”

Indeed, there are constraints in the virtual realm, but they are unlike the design constraints of the physical world. No longer constrained by a monetary budget, designers will find themselves limited in materials nonetheless. Limitations on polygons (the building blocks of virtual worlds) and pixels (that which renders the polygons with color or texture), as well as restrictions like bandwidth, disk space, and memory, will become the budgets of virtual environments. Design projects will become limited in their complexity only by the amount of room they take up in memory. While it is possible that these restrictions will be lessened, and may someday be lifted, it is unlikely that we will ever be satisfied with the complexity of our representations in the virtual realm; these limitations will always remain.

The key to effective virtual reality, especially for fields like architecture, is the creation of images that simulate the essence of a moving scene so accurately that the user becomes *emotionally involved* in the action of the simulation. Users want to see the design that will be created, in terms of lighting effects, finishes, materials, layout and construction details. In order to provide a convincingly realistic simulation, we must attempt to make the virtual model as close to the real-world counterpart as possible, by accurately representing the visual complexity of the scene while maintaining a smooth interactive frame rate (greater than 20 frames per second).

1.2 Background

Computer aided design (CAD) applications and scientific visualizations often need user-steered interactive displays (walk-throughs) of very complex environments (Aliaga et al., 1999). Today, a growing number of models are being created whose complexity exceeds the interactive visualization capabilities of current graphics systems. For such projects the design process, and especially the multidisciplinary design review process, benefits greatly from interactive walk-throughs. Many such CAD models contain millions of primitives, and even high-end systems like the SGI® Infinite Reality Engine cannot render them interactively.

For many years now research has been going on to make objects look more realistic on computer displays, developing techniques required to produce realistic scenes at the required speeds and rendering them at nearly interactive rates. Ideally an interactive walk-through needs to maintain a frame rate of at least 20 frames per second to maintain smooth motion. After more than 20 years of research, rendering still remains a partially unsolved, interesting, and challenging topic. The research questions involved are still far from solved, as is the related question of adapting the limited resolution of the computer display to the perceptual variations of human viewers.

1.3 Previous Work

The Walk-through Project, at the University of North Carolina, Chapel Hill, has developed many different systems to create interactive computer graphics that enable a viewer to experience an architectural model by simulating a walk-through of the model. Most of their research involves software efforts to make advances in the following categories: photo-realism, faster display, real application and model building, and handier interfaces.

Hierarchical view frustum culling, fidelity-based level of detail, hierarchical levels of detail, use of portal textures and occlusion culling are some of the techniques developed for faster display. In the area of real-time realistic rendering, work has been done to enhance photo-realism for interactive walk-throughs and using texture-mapped radiosity for efficient rendering. Most of their studies have shown that one must process and render only those primitives, which the user can see from the current viewpoint and render them only to an extent, which can be detected by the user.

Studies by Bastos et al. (1999) present a method for interactive rendering of globally illuminated static scenes. Global illumination is decomposed into view-independent (diffuse) and view-dependent (non-diffuse) components. Their research involves recombining the two components during rendering using hybrid geometry and image-based approach coupled with multi-pass blending techniques. It allows the pre-computing of both components and fast rendering of globally illuminated scenes. The diffuse scene is stored and rendered as a geometric database with an associated pre-computed radiosity solution, using dense meshing and per-vertex colors or a set of radiosity textures, both of which can be rendered on today's typical graphics hardware. This approach focuses on interactive architectural visualization that accounts for view-dependent glossy illumination.

Lischinski and Rappoport (1998) presented an image based technique for rendering non-diffuse synthetic scenes based on layered depth images. Their work involves capturing both the view-independent and view-dependent appearances as images and recombining the two components to render images from any viewpoint, the results are far superior to raytracing with similar quality for shading and reflections.

Environment mapping is another method for providing fast view-dependent illumination. This technique efficiently accounts for mirror like reflections by pre-computing the incoming radiance for a reflector into a world projection texture map (N.Greene, 1986). This technique is usually implemented in hardware in the form of sphere mapping, making it nearly as fast as regular texture mapping (Lischinsky et al., 1998).

Id Software's Quake 3D Game engine pioneered a new surface-based lighting utilizing lightmaps and texture maps (Michael Abrash, 1997). This is the multi-texturing process, where each texture on a polygonal surface is lit according to the surface's lightmap, which is calculated during a pre-processing stage, resulting in a perspective correct and photo-realistic lighting.

Zhukov et al. (1998) proposed various techniques for the use of lightmaps in real-time applications and specifically addressed the problems of sampling and storing lightmaps. For storing lighting information each face in the scene (a polygon) is assigned to a tiny non-rectangular texture called a lightmap element. All lightmap elements are then grouped (packed) into a number of larger rectangular textures (lightmaps) that are used for real-time modulation of ordinary textures of the scene. With this approach, very realistic three dimensional (3D) scenes were rendered in real-time using a custom dedicated lighting tool.

One of the most well known high quality virtual reconstructions and architectural walk-throughs is the one created for the Virtual Notre Dame Project, from Digitalo Studios (DeLeon, 2000). This project focused on an accurate reconstruction of the cathedral in as much detail as possible. They introduced photo-realistic real-time technology in the area of virtual heritage on desktop computers. Their technique deals with presenting a complex environment comprised of textured polygons. For the overall development of the real-time experience, Epic's Unreal commercial 3D game engine was used allowing for high-end visual effects, high performance frame rate and interactivity, showing lighting, shadows and all the texture properties on the fly.

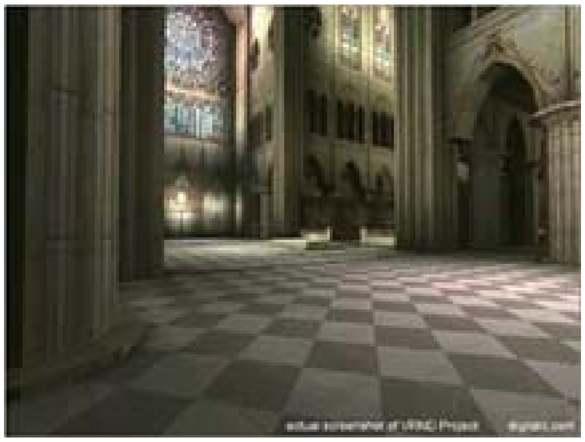


Fig 1-1 Virtual rendition of Notre Dame Cathedral

Much research has been also been done in the field of optimization of models, allowing rendering of complex 3D models at nearly interactive frame rates. The principle for the ideal algorithmic approach is: *'Do not even attempt to render any geometry that the user will not ultimately see*' (D.Aliaga et al., 1999). Traditional approaches to this problem use a hardware graphics pipeline and attempt to minimize the number of polygons sent to the system by extracting visibility information from the model and using this information to cull objects against the viewing frustum and by rendering geometrically coarse representations (levels of detail) of single objects to keep a high frame rate.

Maciel (1995) developed a system that allowed a user to interactively navigate through a complex 3D environment at an approximately constant user-specified frame rate. This technique was based upon the use of imposters.

"*An imposter is drawable representation for an object that takes less time to draw than the true object, but retains the important visual characteristics of the true object. In this context, traditional LODs are just particular cases of imposters*", (Maciel, 1995).

In this way imposters for objects (or the object themselves) could be used near the viewpoint, and cluster imposters could be used when groups of objects were far away.

D.Aliaga et al. (1999), presented a scalable framework for the rapid display of massive models. This included an effective pipeline for integrating multiple rendering acceleration techniques, including visibility culling, geometric levels of detail, and image based approaches. This study described a database representation scheme for massive models, partitioning the model into manageable subsets, which could be fetched into the main memory at run time.

Hoff (1997) worked on increasing 3D game realism, in terms of both scene complexity and speed of animation. His approach was similar to the above in terms of efficiently finding a large portion of the game world that is not visible to the viewer for each frame of the animation and preventing it from being sent to the graphics system. He employed the concept of *conservative visibility* whereby a set of visible polygons is conservatively estimated. Techniques such as view frustum culling, bounding volumes, bounding volume hierarchies, backface culling, dividing the model into cells and portals have been used. By combining these relative simple algorithms, he was able to achieve a high average rate of culling for each frame of animation for scenes with high complexity. The system could be used as a culling stage before applying rendering schemes such as raytracing or radiosity based methods.

The University of California, Berkeley, building walk-through system (Funkhouser et al., 1996) used a hierarchical representation of the model. They used visibility culling and level of details of objects for geometric simplification.

IRIS Performer^{TM} (Rohlf 1994) is a high performance library that uses a hierarchical representation to systematically organize the model into smaller parts, each of which has an associated bounding volume. This data structure can be used to optimize culling and rendering of the model. Many other systems have been developed on top of Performer for interactive display of large environments, including an environment for real-time urban simulation (Jepson 1995).

A much-used approach to simplify geometric complexity is to replace geometry using textures. A texture is a snapshot of the model from a single viewpoint. D.Aliaga et al. (1998) proposed techniques for providing smooth transitions for simplifying large, static, geometric models with texture-based representations (or imposters). Their study focused on developing techniques for providing continuous imagery across borders between geometry and sampled textures at all times, and providing smooth dynamic transitions between geometry and textures, removing the sudden jump when switching between geometry and textures. They achieved this by warping the geometry to match the texture when the eye moved from the texture viewpoint (the viewpoint from which the texture appears perspectively correct). This allowed the texturing hardware to be effectively used. Their method took advantage of the fact that geometry is anyway re-rendered every frame, so by slightly modifying the geometry one is able to use static textures and achieve higher frame rates.

Visualization of architectural spaces requires large complex models with many geometric primitives. Past research has focused on dividing a model into cells (rooms or pre-determined subsections of the model) and portals (doors, windows and other openings). Visibility culling algorithms are used to determine which other cells are visible from a particular viewpoint and view direction. Rendering is thus reduced to the geometry of the visible cells. Exact pre-processing algorithms (Teller, 1991) as well as conservative run-time algorithms (Luebke, 1995) have been developed.

Another technique that further simplifies the rendering process, is by replacing the cells visible through a portal with a texture (Aliaga et al., 1997). The system now needs to render only the geometry of the cell containing the viewpoint and a few texture- mapped polygons. If the viewpoint approaches a portal, the portal texture will return to geometry, allowing the viewpoint to move into the adjacent cell. These 2D texture representations are much faster to render as compared to 3D geometry.

The downside here is that a portal can be viewed from multiple view directions and multiple viewpoints. Since, a single texture only produces a perspectively correct image from one viewpoint, once a viewer changes his position the texture appears incorrect. This problem can be solved by either using image warping or using multiple textures. D.Aliaga et al. (1997) used multiple portal textures for their study. At run-time, the portal texture that most closely represented the geometry visible from the portal was selected. As the viewpoint moved, the textures were continuously switched. This produced a visual jump known as ‘popping’, which can be controlled by increasing the number of textures. This requires more texture memory or the use of a large amount of main memory and copying to texture memory as necessary.

Applying image-based rendering techniques to warp the portal textures to the current viewpoint, thereby achieving smooth transitions, can solve the above problem. Rafferty et al., (1998) used 3D warping of single reference images and 3D warping of layered depth images (LDIs), when using images to replace geometry visible through portals. The first technique involves warping multiple images obtained from different viewpoints. The second technique which uses layered depth images stores multiple samples on one image, that become visible as the image is warped. This study showed that for highest quality using LDIs is best solution to date, for the visibility errors to which 3D image warping is prone. However, conventional textures exhibit the highest performance and can be generated at run-time.

1.4 Motivation and Problem Statement

There are increasing demands of realism in immersive virtual environments, both in terms of scene complexity and speed of rendering detailed sets of 3D polygons with appropriate lighting effects as the camera moves through the model. With this approach, though, the primary challenge is constructing a digital representation for a complex, visually rich, real-world environment. The key to realism is the complexity of the scene both in terms of the geometry of the model and in terms of how the interaction of light in the virtual world simulates its counterpart in a real-world environment. If the rendered images are visually compelling, and they are refreshed quickly enough, the user feels a sense of presence in a virtual world, enabling applications in education, computer-aided design, architectural visualizations, electronic commerce, and entertainment.

Despite recent advances in interactive modeling tools, laser-based range-finders, computer vision techniques, and global illumination algorithms, it remains extremely difficult to construct compelling models with detailed 3D geometry, accurate material reflectance properties, and realistic global illumination effects. Even with tools to create an attractive, credible geometric model, it must still be rendered at interactive frame rates, limiting the number of polygons and shading algorithms that can be used.

The reason that the criteria mentioned above are mutually incompatible is that while an interactive system needs to achieve interactive frame rates, realistic looking models can contain hundreds of millions of polygons, far more than currently available workstations can render in an interactive fashion. A balance between realism and interactivity is required. Today, the fastest commercial systems can effectively draw around 20 million triangles per second, i.e. around one million triangles per frame at 20 frames per second. An Interactive walk-through of a geometric database for an entire university campus, with vegetation, cars, buildings with furniture, etc., is much more than two orders of magnitude out of reach.

All science, engineering and design disciplines are facing the same problem: How to archive, transmit, visualize, and explore the massive datasets resulting from modern computing.

This research seeks to achieve the following:

- Develop an understanding of real-time rendering issues, studying techniques used for photo-realism and optimization of virtual environments.
- To model an architectural environment, for an interactive and highly realistic virtual simulation in real-time. Tadao Ando's, Church on the Water, has been chosen for this project.
- Use the techniques best suited for introducing photo-realism and optimization of this model for real-time simulation.
- In the process, lay down a set of guidelines for architects/designers that would help them efficiently construct their virtual environments for a real-time interactive photo-realistic simulation.

Reasons for selecting 'Church on the Water'

- For its truly challenging lighting model, suitable for the purpose of demonstrating photo-realism.
- The geometric purity of its architectural structure.
- Simplicity of form and materials.

  In Ando's buildings light gives objects existence as objects and connects space and form. This building has a simple inorganic geometry, limited materials, powerful expression, style being reduced to the minimum. It is the interplay of light and dark that reveals forms, bringing richness to the space.
- The approach area is surrounded on four sides by milk white frosted glass, forming 'an enclosure of light'. Here strong, direct sunlight and soft filtered light intermingle. This contrast of light gives this place its solemnity.
- A curving darkened stairway leads to the main chapel, where natural light pervades the space. The glazed wall of the chapel facing the pond can be entirely opened framing the landscape beyond.

1.5 Software Used

The surface models which represent geometry in virtual environments are typically made up of polygons as their most basic element. These planar polygons are commonly three or four-sided, and can be combined to create an infinite number of shapes and volumes.

Most CAD packages generate dozens of different kinds of 3D primitives, and many of these are valid for describing the surface models we wish to render in real-time. The modeling softwares used for this project are AutoCAD Architectural Desktop™ 3.0 and Autodesk® VIZ 4 Almost any CAD package can be used to generate the initial model data, which can be later embellished to create virtual environments, including AutoCAD®, form•Z, 3D Studio VIZ®, 3ds max™, MicroStation, and Alias|Wavefront™.

AutoCAD Architectural Desktop™ 3.0

AutoCAD Architectural Desktop™ 3.0 is a common CAD program, and its DXF file format is an industry standard for describing model geometry. Translations from AutoCAD Architectural Desktop™ to other programs (3D Studio VIZ®, 3ds max™) via DWG, DXF, and 3DS file formats is a simple process. Lots of primitives are available for modeling. Most of the architectural virtual environments can be created using only a few of these: extruded lines and polylines, and 3D faces. With practice almost anything can be represented with these few primitives, at the same time maintaining a low polygon count in the virtual environments.

Autodesk® VIZ 4

Autodesk® VIZ 4 was chosen due to its flexibility in polygonal mesh editing and flexibility to use it in combination with the Lightscape 3.2 radiosity software package and AutoCAD Architectural Desktop™. “Modeless” modeling provides a unified workspace, and surface finishes and lighting systems can be created, mapped, and manipulated on-the-fly. In conjunction with global illumination rendering Autodesk® VIZ 4 also introduces support for physically based lighting. Physically based lighting benefits by making it significantly easier and more intuitive to set up lighting in the scenes.

Lightscape

Neither traditional scan-line renderings nor high-end raytracers account for indirect illumination. Unlike these methods, which try to "fake" realism, Lightscape uses radiosity to simulate how light energy is distributed, reflected and absorbed by every surface.

Because Lightscape works with actual photometric (light energy) values, it can more accurately simulate real-world lighting and materials. With Lightscape’s powerful yet straight-forward lighting interface and global illumination rendering, the process of lighting a scene and obtaining natural looking effects is both intuitive and precise.

Lightscape also accounts for **real-time interactivity.** Using radiosity, the lighting of a scene is pre-calculated and stored as an integral part of the 3D geometry. Lightscape is particularly useful for creating highly realistic real-time virtual environments. It has tools for reducing the geometry of a model by converting radiosity meshes into textures. This technique can be particularly useful for creating an interactive 3D environment for use in applications such as games, websites (VRML), and virtual sets.

CHAPTER 2: PHOTO-REALISM IN VIRTUAL ENVIRONMENTS

2.1 What is Photo-realism?

Photo-realism refers to construction of computer images that in addition to geometry accurately simulate the physics of materials and light. The goal in this study, both for static images and moving scenes in virtual reality applications, is to achieve as efficiently as possible, sufficient accuracy to create ***photo-realistic images.***

At present, the nearest we have come to "photo-realism" in "real-time" in computer graphics requires very powerful computer workstations, such as Silicon Graphics (SGI®). The main additional requirement of VR is that displays be shown in three-dimensions using stereoscopic vision. This requires still greater computer power since double the information is being processed at any given point in time. Photo-realism also requires that images should have a frame-refresh rate suitable for the human eye to view. Also, whilst moving through the "virtual world" both the size and the perspective view of the "virtual buildings" should change. If not, "display mismatch" will make the experience seem less than believable, (McMillan, 1994).

For many years now computer scientists have tried to make objects look more realistic on computer displays. Realism here means to generate images on a computer, which are indistinguishable from real photos. This realism is possible through the use of local and global illumination methods. The basis for a physically correct calculation of the illumination depends on the accuracy of the scene description. This means both the geometry as well as the description of the surfaces and light sources.

2.2 Lighting Algorithms

Lighting (global illumination) in a scene can be categorized into view-independent (diffuse) and view-dependent (non-diffuse) components.

View independent illumination accounts for all effects of light that are not dependent on the viewer's position. This illumination depends only on the configuration of the geometry and the lights and so may be precomputed and stored as a dense mesh with pervertex colors or as texture maps rendered using traditional shading hardware (Cohen et al., 1993) (Diefenbach, 1996).

View-dependent illumination accounts for specular and glossy directional-biased lighting. These effects depend on the viewpoint and are difficult to pre-compute and render accurately.

There are several techniques used for photo-realistic visualization of 3D models, many of which incorporate in real-time, effects such as lighting and reflection, that are commonly only possible in frame by frame rendering. These are discussed below:

Flat Shading

This is the most basic shading technique. A single shading color is determined per polygon, and the entire polygon is either rendered with this color, or the color is added to the texture on the polygon. This technique tends to emphasize the flat nature of the polygons in the scene.

Gouraud Shading

In the area of 3D graphics the most traditional way of implementing polygonal lighting for VR real-time applications is to take the light and color information from each vertex and to linearly interpolate the lighting and color of pixels between the vertices. Each vertex of a scene contains information about its location, its color, and finally about its orientation relative to a light source. ‘Lighting’ means making an actual solid looking object out of a wire frame. This is a fixed shading method, meaning that once the vertex colors have been calculated they will not vary with time-of-day changes such as moving sun angle and sun intensity. It only relies on the RGB values assigned to each vertex and the polygons color will vary in between these RGB values. Most VR systems including Leeds Advanced Driving simulator and arcade games such as SEGA® ‘Street Fighter’, use gouraud shading extensively.

This approach has several disadvantages. It poses serious difficulties in achieving detailed lighting without creating a large number of polygons. The lighting can look

perfectly realistic as long as an object consists of enough polygons or vertices. However, many vertices put a high strain on the 'transform' part of the 3D rendering process and it puts a high strain on the bus, since each vertex has to be sent to the graphics chip. Also, this lighting is not perspective correct, It lacks correct diffuse-specular simulation and photo-realism. For example, it is also impossible to have a highlight in the middle of a large polygon. It is restricted to the diffuse component of the illumination model.
If used correctly, this technique makes objects look round. The technique can not be used in a convincing way if there are multiple light sources in the scene. Another disadvantage is that when the polygon moves or rotates, a static light source does not cause static lighting on the polygon. It also varies according to the viewing angle.

To summarize it can be said that 'Vertex Lighting' or 'Gouraud Shading' is a good and valid technique for objects that consist of many polygons, but its not good enough for objects that consist of only few polygons. How NVIDIA® puts it is " vertex lighting is of limited use in some scenarios because its base unit is the triangle" (Pabst, 2000)

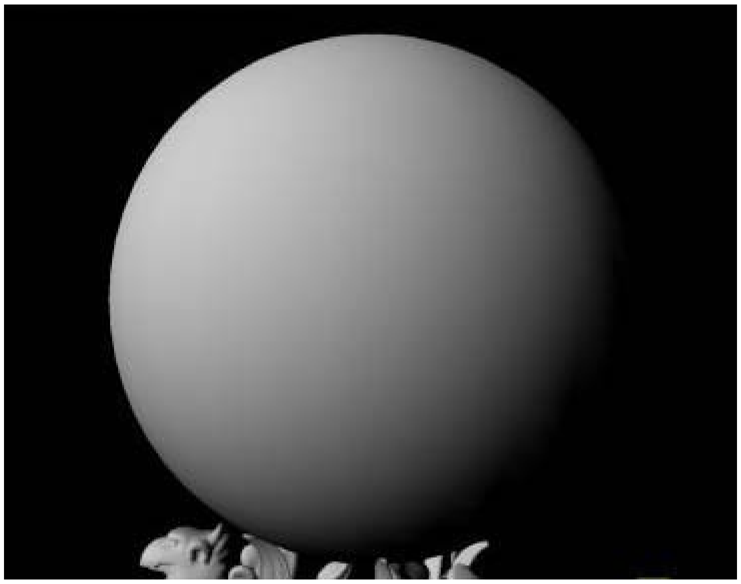


Fig 2-1 A ball with Gouraud Shading

Phong Shading

Phong shading is one of the most realistic techniques for dynamic lighting. It overcomes the limitation of gouraud shading by incorporating specular reflection into the scheme. A texture is attached to every light source. This texture is then projected on every polygon, by using the normals of each polygon vertex as an index in the lightmap. This way, highlights can occur in the middle of a polygon. Also, the lightmap is fully configurable: It can be dithered, smooth, or very sharp in the center. Also, the intensity transition from one polygon to the next is smoother. The primary objective is efficiency of computation rather than for accurate physical simulation. It is very hard however to have directed spotlights with this technique, or to have multiple lights on a single polygon.

Dynamic Lighting

With dynamic light sources, their sphere of illumination is projected into the world and their dynamic light contributions are added to the appropriate lightmaps, which are used to rebuild the affected surfaces.

Local Illumination Algorithms

Local illumination algorithms describe only how individual surfaces reflect or transmit light (Lightscape 3.2, User's Guide). And these are usually applied in real-time 3D graphics shading. Given a description of light arriving at a surface from specific light sources, these mathematical algorithms predict the intensity, spectral character (color) and distribution of light leaving the surface.

Global Illumination Algorithms

For production of more accurate results it is important to take into account not only the light sources themselves, but also how all the surfaces and objects in the environment interact with the light and how this is transferred between the model surfaces. Such rendering algorithms are called global illumination algorithms and include radiosity and raytracing.

Radiosity

The radiosity method is a theoretically rigorous method that provides a solution to diffuse interaction within a closed environment. One interesting aspect of a radiosity solution for a scene is its view-independent nature. This is an important result, because this means that during a walk-through, when the user changes their field of view (FOV) to focus on another view, there is no need to re-compute the form factors and solve the linear radiosity equations. Here, it is only required to render the new FOV based on pre-computed radiosity solution. As the solution is view-independent, it is based on subdividing the environment into discrete patches, or elements, over which the light intensity is constant. Radiosity is most successful in dealing with such manmade environments like interiors of large architectural structures, and has certainly produced the most impressive computer graphics images to date (Watt et al., 1992).

Using software like Lightscape, you can render your lighted model using radiosity techniques. Radiosity computes the illumination of a surface from light shining directly from a source and from indirect light reflected from other surfaces in the environment to achieve very realistic lighting of diffuse-surfaced environments. Unlike view-dependent rendering algorithms like raytracing, radiosity is view-independent and once the lighting of each surface has been calculated, a participant can navigate through a very "realistic" virtual environment in real-time. Using Lightscape, you can add or import point lights, linear lights, and area lights to your model, as well as day-lighting, spotlights, or even customized lamps.

Once the materials have been defined and lights added to the model, Lightscape can be used to process a radiosity solution. This means that the light emitted from the sources in the file is calculated, directly and indirectly from inter-reflections between surfaces in the model. Upon completion of the radiosity solution, the virtual environment is simply a geometric mesh, colored appropriately to simulate the light that it has received. **The radiosity solution has no light sources itself**, but when the file is rendered in real-time, parameters in the file tell the simulation software to use the colors of the mesh and not to add or render the effects of additional light sources. Thus, only with radiosity solutions, is there no explicit light source in the virtual environment. Radiosity works best with indirect lighting and diffuse surfaces.

Raytracing

Raytracing is a method that allows us to create stunning photo-realistic images on a computer. It excels at calculating reflections, refractions and shadows. The first stage of creating a raytraced image is to "describe" what it is that you want to depict in your picture. You may do this using an interactive modeling system, like a CAD package, or by creating a text file that has a programming language-like syntax to describe the elements. Either way, you will be specifying what objects are in your imaginary world, what shape they are, where they are, what color and texture they have and where the light sources are to illuminate them.

Having done all of this, it is fed into the raytracer. The main drawback of raytracing is – it is often very slow and processor intensive. The software actually mathematically models the light rays as they bounce around the virtual world, reflecting, refracting until they end up in the lens of your imaginary camera. This can quite literally involve thousands and millions of floating-point calculations and takes time. Raytracing images can take anything from a few seconds to many days. It's a long process, but the results can make it all worthwhile.

This method, however, omits the most important aspect for creating a true photo-realistic image, that is, diffuse inter-reflections such as 'color-bleeding' effects, eg: colored light bouncing off blue tiles onto the lens in the picture below should result in a blue tint.
Raytracing works best with direct lighting and highly reflective materials.

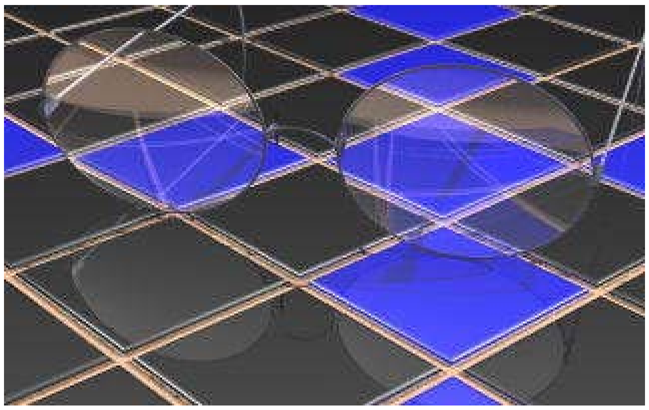


Fig 2-2 Raytraced glasses, take note of the reflections on the lens and the shadows cast by the glasses on the floor.

Lighting Algorithm	**Advantages**	**Disadvantages**
Raytracing	Accurately renders direct illumination, shadows, specular reflections, and transparency effects. Memory Efficient	Computationally expensive. The time required to produce an image is greatly affected by the number of light sources. Process must be repeated for each view (view-dependent). Doesn't account for diffuse inter-reflections.
Radiosity	Calculates diffuse interreflections between surfaces. Provides view-independent solutions for fast display of arbitrary views. Offers immediate visual results.	3D mesh requires more memory than the original surfaces. Surface sampling algorithm is more susceptible to imaging artifacts than ray-tracing. Doesn't account for specular reflections or transparency effects.

Table 2-1 Comparison between Radiosity and Raytracing

CHAPTER 3: REAL-TIME RENDERING

Real-time rendering refers to creating synthetic images fast enough on the computer so that the viewer can interact with a virtual environment. It involves issues of three dimensional rendering, modeling, animation and user interaction. It is the most highly interactive area of computer graphics.

The process of reaction and rendering happens at a fast enough rate, so that the viewer does not see individual images but becomes immersed in a dynamic process.
The rate at which images are displayed is measured in frames per second (fps) or Hertz (Hz). At one frame per second there is little sense of interactivity and the user is aware of the arrival of each new image. At around 6 fps, a sense of interactivity starts to grow. An application displaying at 15 fps is real-time, the user can focus on action and reaction. At about 72 fps and up, differences in the display rate are effectively undetectable. (Moller, 1971)

There is more to real-time rendering than interactivity. Graphics acceleration hardware, which is dedicated to three-dimensional graphics, has become a part of its definition. With recent rapid advances in this market, add-on three dimensional graphics accelerators are becoming standard for home computers.

3.1 The Graphics Rendering Pipeline

The graphics rendering pipeline forms the core of real-time rendering. The main function of the pipeline is to generate, or render, a two-dimensional image, given a virtual camera, three-dimensional objects, light sources, lighting models, textures, and more. The rendering pipeline is thus the underlying structure for real-time rendering. The pipeline can be coarsely divided into three stages called *application, geometry*, and *rasterizer* . This structure is the core- the engine of the rendering pipeline- which is used in real-time computer graphics applications.

The application stage, as the name suggests, is driven by the application and therefore implemented in software. This stage may, for example, contain collision detection, speed up techniques, animations, force feedback, etc. The developer can change the implementation in order to change performance. Here the user can interact with the models in the scene if the application allows that. Say, the user has selected a subset of the scene and moves it (using the mouse). The application stage must then see to it that the model transform for that subset of the model is updated to accommodate for that translation. At the end of the application stage , the primitives of the model are fed into the geometry stage.

The geometry stage deals with transforms, projections, lighting, etc. This stage computes what is to be drawn, how it should be drawn and where it should be drawn.
The model is transformed into several different spaces or coordinate systems. After its vertices and normals are transformed, the model is said to be located in the world coordinates or world space from the model space.

Finally, **the rasterizer stage** draws (renders) an image with the use of the data that the previous stage generated. Given the transformed and projected vertices, colors, and texture coordinates the goal of the rasterizer stage is to assign correct colors to the pixels to render an image correctly. This is the conversion from two-dimensional (2D) vertices in screen space- with a z-value (depth value), a color and possibly a texture coordinate associated with each vertex-into pixels on the screen. For high performance graphics, it is critical that the rasterizer stage be implemented in hardware.

3.2 Texturing

Texturing is a process that takes a surface and modifies its appearance at each location using some image, function, or other dataset. Huge modeling, memory, and speed savings are obtained by combining images and surfaces in this way. Color image texturing also provides a way to use photographic images and animations on surfaces.

The first step in the texture process is obtaining the surface's location and projecting it into parameter space. Projector functions include spherical, cylindrical, box, and planar projections. In real-time applications projector functions are usually applied at the modeling stage, and results of the projection are stored at the vertices. Some methods, such as environment mapping, have specialized projector functions that are evaluated on the fly.

The typical method for using projector functions in real-time systems is by applying (u, v) texture coordinates at each vertex during modeling. To allow each separate texture map to have its own input parameters during a rendering pass, Application Programming Interfaces (APIs) allow multiple pairs of (u, v) values. For image textures, the texture coordinates are used to retrieve texel information from the image. Once the texture values have been retrieved, they maybe used directly or further transformed, and then used to modify one or more surface attributes.

3.2.1 Image Texturing

In image texturing a two dimensional image is effectively glued onto the surface of a polygon and rendered. The texture image size used in hardware accelerators is restricted to $2^m X 2^n$ texels, or sometimes even $2^m X 2^m$ square, where *m* and *n* are non-negative integers. Some graphics accelerators have an upper limit on texture size.

The rendered image appears correct as long as the projected polygon is the same size as the texture. But if the projected square covers more or lesser pixels than the image, the image appears pixellated. Texture antialiasing algorithms are used to preprocess the texture and create data structures that will allow quick approximations of the effect of a set of texels on a pixel.

Mipmapping is the most popular method of antialiasing for textures. It is a process in which the original texture is filtered down repeatedly into smaller images. It is implemented in some form on even the most modest graphics accelerators now produced. When the mipmapping minimization filter is used, the original texture is augmented with a set of smaller versions of the texture before the actual rendering takes place.

Texture caching and compression is important for efficiency of real-time performance. A complex application may require a considerable number of textures. The amount of fast texture memory varies from system to system, but whatever the case there is never enough. Texture caching can be done to strike a balance between speed and minimizing the number of textures in memory at one time. For example, for textured polygons that are initially far away, the application may load only the smaller subtextures in a mipmap, since these are the only levels that will be accessed.

Textures should be kept small, no larger than is necessary to avoid magnification problems. Polygons should be grouped by their use of texture. Another method called tiling involves combining a few smaller (non-repeating) textures into a single larger texture image in order to speed access.

3.2.2 Multi-pass Multitexture Rendering

Multi-pass rendering is a technique that builds up a final rendered image from a 'combination' of the results of a set of separate rendering passes (Watt et al., 2001). Each separate pass modifies the previous result. These rendering passes involve pixel-by-pixel operations and in general they can be exemplified by masking an image with a binary mask image, or blending two images together.

The more passes a renderer must take, the lower is its overall performance. To reduce the number of passes, some graphics accelerators support multitexturing, in which two or more textures are accessed during the same pass. In addition to saving rendering passes, multitexturing actually allows more complex shading than does the application of a single texture per pass. Different texturing methods can be used like alpha mapping, lightmapping, gloss mapping, environment mapping, bump mapping, etc.

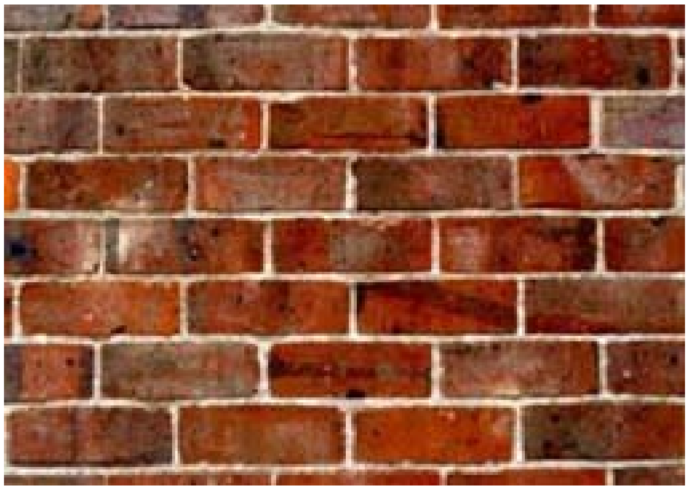
3.2.2.1 Lightmapping

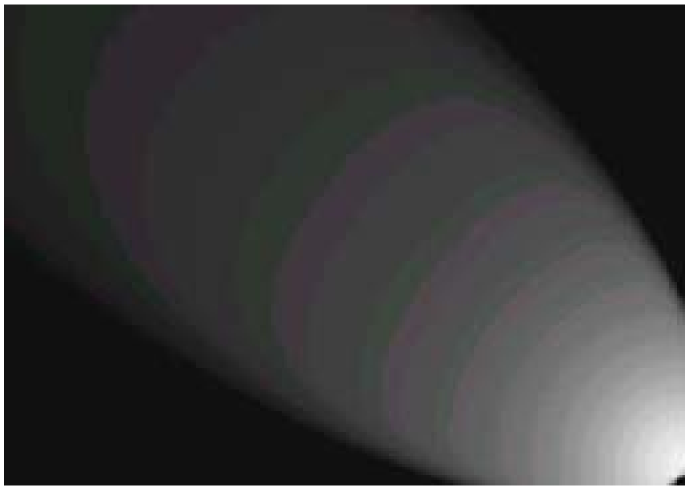
For static lighting in an environment, the diffuse component on any surface remains the same from any angle. Being view independent, this contribution of light on a surface can be captured as textures. The light texture can be multiplied by the surface's material texture during the modeling stage and the single resulting texture can be used. Or a better approach is to use the lightmap in a separate stage. Since the lighting changes slowly across a surface, so the lightmap can be quite low- resolution.

The lightmap textures are modulated into the scene with a second rendering pass in addition to the first pass to render the base surface texture. Multi-pass multitexture rendering is done using a standard texture map and a lightmap.

Lightmaps are an obvious extension to texture maps that enable lighting to be pre-calculated and stored as a two-dimensional texture map (Watt et al., 2001). The reflected light over a sampled surface is cached in such a two-dimensional map. The lightmap is then applied to a material to simulate the effect of a local light source. Like, specular highlights it can be used to improve the appearance of local light sources without resorting to excessive tessellation of the objects in the scene.

Using lightmaps requires a multi-pass algorithm unless the objects being mapped are untextured. A texture simulating the light's effect on the object is created, then applied to one or more objects in the scene. Appropriate texture coordinates are generated, and texture transformations can be used to position the light, and create moving or changing light effects. Multiple light sources can be generated with a combination of more complex texture maps and/or more passes to the algorithm. Lightmaps can often produce satisfactory lighting effects at lower resolutions than normal textures. The use of lightmaps in real-time photo-realistic lighting, was pioneered by both the Quake^{TM} 3D game engine (Abrash, 1997) and by Zhukov et al. (1998)


Wall mapped with a standard brick texture


Lightmap

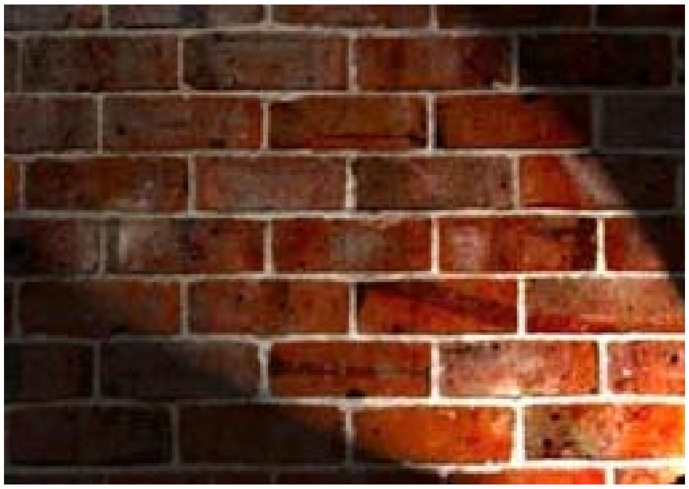

Lightmap applied to the wall using a second rendering pass

Fig 3-1 Lightmapping Technique

The advantages of this approach are:

- Real-time performance speed
- Graphics hardware independence
- Lower algorithm complexity,
- A set-up for an open, realistic lighting model allowing for more real-time effects.
- There is no restriction on the complexity of the technique used for the pre-calculation of the light stored in the lightmap. These maps can either be stored separately from the texture maps, or used to pre-modulate the object's texture map with their lighting information. If the lightmap is kept as a separate entity they are calculated at a far lower resolution than the ordinary texture map to save the required memory storage. An interesting situation that arises from rendering hundreds of textures is that the main area to optimize is now texture memory usage rather than polygon count. The standard 64 MB of graphics cards' texture memory just isn't enough for really nice textures (especially in a large environment). Rendering lighting as textures is also pushing the movement from SGI®s to Linux PCs since PC graphics cards will do multi-texturing (and SGI hardware will not). This enables us to combine a tiled texture map with a non-tiled lightmap on a surface (similar to current games such as Quake™ 3).

  This helps significantly in the area of optimizing texture usage because the main problem with using one non-tiled texture on a surface is that the texture needs to have a high resolution in order to look good up close. Multitexturing allows us to use a lower resolution tiled texture combined with a lower resolution lightmap and yet have a textured surface that appears similar at a distance and more detailed up close.

  Lets say there is a brick wall with a light shining on it. A tiled brick texture is applied to the wall but the lighting will not be tiled- it will be a texture that covers the entire wall. The brick texture can either be combined into the lighting and applied as a single texture or the lighting can be rendered as a lightmap and multi-texturing can be used to apply this and the tiled texture to the wall. The later allows us to maintain a high resolution up close that is obtainable with a tiled texture but not feasible if one is trying to render high resolution textures for every wall in a scene.

- As the lightmap view independent lighting changes more slowly than texture detail, except at shadow edges, any aliasing effects are eliminated with the use of linear texture filtering techniques.

Using Lightmaps has several down sides:

- Lightmaps require an additional rendering pass. GLQuake^{TM}, was pretty much the first game that made heavy use of lightmaps. At the time of GLQuake^{TM}, very few 3D chips had more than one rendering pipeline and the pipelines back then were only able to apply one texture per pass. On top of that the clock of those good old 3D chips was rather low, somewhere in the 50-100 MHz range. The result of GLQuake^{TM} 's use of lightmaps was that only very few 3D chips could run it at acceptable frame rates, simply because the low fill rate of the old 3D chips was halved due to the fact that each pixel had to go through the rendering process twice, once for the texture and once for the lightmap. This situation created the need for 3D chip solutions that could render two textures (a lightmap is nothing else than just a special kind of texture) in one pass. Quantum3D® 's first Obsidian board using the Voodoo1 chip was the first one that could render two textures in one pass. The first mainstream solution with this feature was 3dfx's Voodoo2^{TM} cards, where simply two texelfx2 chips worked in parallel. The first one-chip solution came from NVIDIA® in form of their TNT chip. Today there's hardly any modern 3D chip anymore that would not at least have two pipelines to process two textures in one clock, but still lightmaps mean a performance impact, because those two pipelines could produce two simply textured pixels in one clock, instead of only one that's textured with a texture and lightmap. GeForce^{TM} 2 does not suffer from this issue though, since, each of its rendering pipelines can apply two textures to a pixel in one pass. Thus lightmaps have no performance impact on GeForce^{TM} 2 's rendering engine, although, they require an additional amount of memory and bus bandwidth (Pabst, 2000).

- Lightmaps need to be produced for each surface separately. In the example of the light beam falling on the wall, a lightmap needs to be created for the wall and one for the floor. That's the way its done in games like GLQuake^{TM}, Quake^{TM}2 and Quake^{TM}3 for example.

- Lightmaps cannot give any acceptable impression of depth to an object. With lightmaps one cannot produce bump mapping effects, because the lighting of a flat bump mapped object changes with the viewing angle. Lightmaps cannot represent that; they appear the same from any angle that one looks at them.

3.2.3 Shadows

Shadows can improve 3D modeled representations of our environments by enhancing realism and providing depth and perspective cues. Shadows can be generated by several methods. For simple objects shadows can be modeled as a set of polygons. The disadvantage of this though is, that the shadow has very sharp edges, which doesn't look very realistic. For more complex cases, shadow textures can be projected onto objects in a multipass approach, just as with lightmaps above.

Another approach is to pre-render the lighting into textures on the model; software packages like Lightscape would do this. Here again, the question of a single non-tiled texture map arises (which has the lighting and shadows pre-rendered into it) with issues of resolution and memory usage, and one has to make choices accordingly.

3.2.4 Reflections

Environment mapping (EM), also called reflection mapping is a simple, yet powerful method of generating approximations of reflections in curved surfaces (Moller, 1971).
Environment mapping simulates the results of raytracing. Because , it is performed using texture mapping hardware, it can obtain global reflection and lighting results in real-time. This is essentially the process of pre-computing a texture map and then sampling texels from this texture during the rendering of a model. The texture map is a projection of 3D space to 2D space. There are many ways to project a 3D surface to a 2D surface, like, Open GL spherical mapping, Blinn/Newell latitude mapping, Cube mapping, etc.

3.3 Optimization Techniques- by not drawing what is not seen

In real-time rendering, there are three performance goals: more frames per second, higher resolution, and more (and more realistic) objects in the scene. A speed of 60-72 fps is generally considered enough, and 1600 X 1200 pixels is enough resolution, but there is no upper limit on scene complexity. Hence, speed-up techniques and acceleration schemes will always be needed (Moller, 1971). To reduce the load on the graphics subsystem, the solution lies in efficiently finding a portion of the virtual world that is not visible to the viewer, for each frame, and simply prevent it from being sent to the graphics system.

In Computer Graphics, a 3D scene is organized into a data structure called a scene graph. This is a directed acyclic graph of objects called nodes. It organizes and stores all of the data needed to render a 3D scene. The node objects contain the scene–database information, including geometry, materials, textures, lights, cameras, and anything else necessary to implement display. Rendering of the 3D scene is performed by traversing the graph, selecting the leaf nodes to be drawn, and issuing commands for the information contained in the selected leaf nodes. During the traversal, culling algorithms and filtering techniques can be performed, whereby nodes completely outside the current view are not selected, and need not be rendered.

There are several optimization and filtering techniques for real-time rendering, and have been described below:

View Frustum Culling involves discarding polygons that are outside of the viewer's current field of view. View frustum culling takes place in the application stage, so both the geometry and rasterizer stages significantly benefit from it. Before drawing the entire scene for the current frame, each polygon is tested to see if it lies inside the viewer's field-of-view. Polygons that lie outside the viewing frustum are removed, or culled, from the list of polygons to be drawn. The limitation is that polygons sent to the graphics system may still be invisible to the viewer since they could be behind other polygons.

Bounding-Volumes improve upon the visibility culling algorithm. Often in 3D games and CAD models there is some structure to the world that can be utilized by the view frustum culling system. By forming clusters of polygons that can be culled away together, a much more efficient overlap test can be performed. By testing the bounding volume for overlap, the entire group of polygons can be culled away or accepted. If the volume partially overlaps, then using the previous system, individual testing of each polygon in the bounding volume must be performed.

Bounding-Volume Hierarchies: Recursively forming higher-level groupings formed by clustering neighbouring bounding-volumes into larger volumes until the entire scene is contained in a tree-structure, with the root bounding-volume containing the entire scene (Hoff, 1997). View frustum culling can now be performed in steps, by first testing the root of the tree. Based on whether the root is completely inside or completely outside, all of the children can be accepted or rejected. In case of a partially overlapping root node, the children and their subtrees would have to be tested. This process is more efficient since, many polygons can be culled away with only a few overlap tests.

Backface Culling: The simplest form of occlusion culling is known as backface test, also called backface culling. All polygons whose frontsides face away from the viewer can be culled. These examples should be modeled so that the user can only view the frontside of a polygon, and are quite typical in most games and CAD designs. Each polygon has a surface normal that points in the front facing direction. If the angle between the surface normal and the vector from the viewer to the polygon is less than 90 degrees then the polygon is back facing, and is discarded.

Occlusion Culling: More sophisticated algorithms which determine a polygon mesh being rendered is visible to the camera or not, are known as occlusion culling algorithms. Most occlusion culling algorithms start with a spatial data structure (e.g. an octtree, a data structure that can define a shape in three dimensions) or a hierarchical representation of the scene being drawn and calculate the visibility of a given polygon from the eye's point of view.

The Hierarchical Z-buffer: Ned Greene et al. (1993) developed an algorithm called the hierarchical z-buffer which takes the scene, represented using an octtree, and an image pyramid built using the z-buffer called the z-pyramid. When traversing the octtree, for each node that node's bounding box is tested against the z-pyramid to determine if that box is visible. Within a node, each front-facing polygon is tested against the z-pyramid to determine visibility. For each face, the coarsest z-pyramid cell, which encloses the face's nearest z-value. If the face's z-value is less than the cell's z-value, that face is occluded and testing can stop at this point. For densely occluded scenes, this algorithm is able to often cull faces based on a single depth comparison.

Cells and Portals algorithms work well for large-scale architectural models that have rooms or open spaces separated by doorways or other connecting structures. Each cell or room is separated by some doorway or portal, and from any particular cell, other cells are only visible through a sequence of portals. The approach is to find out the cell the viewer is in. Then, find out which portals leading out of that cell are in the viewer's field of view. Then recursively apply the previous step for each visible cell, finally rendering all of the cells visited. Portal culling methods preprocess the scene in some way, either automatically or by hand.

Levels of Detail: The basic idea of levels of detail or LOD's is to use simpler versions of an object as it gets farther from the viewer. In the simplest type of LOD, the different representations are simply models of the same object containing different numbers of primitives. Typically, the different LOD's of an object are associated with different ranges of distance.

By combining these algorithms, a high average rate of culling can be achieved for each frame of the scene. The system can be used before rendering the scene with raytracing or radiosity based methods.

CHAPTER 4: CASE STUDIES

4.1 Study Examples demonstrating the potential of Real-Time Visualization

A Real-Time Visualization system for Large scale Urban Environments

Using its own unique computer simulation system and methodologies, the University of California, LA (UCLA) Urban Simulation Team is creating a real-time virtual model of the entire Los Angeles Basin. They are exploring the diverse applications for real-time visual simulation in design, urban planning, emergency response and education.

This system is being extended to support a client server capability, which will allow the seamless interactive navigation of the entire Virtual Los Angeles Model, (a model which is projected to reach terabyte size over the next several years) while simultaneously supporting hundreds of remote interactive users.

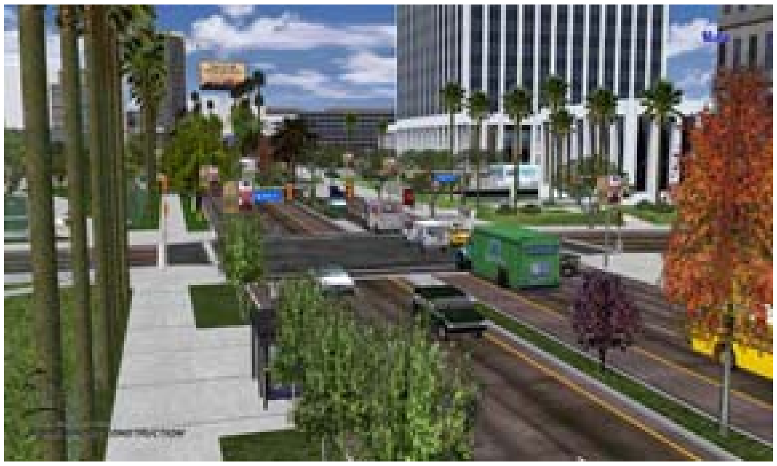


Fig 4-1 The Miracle Mile, Los Angeles

The Urban Simulation Team at UCLA has drawn from technologies developed for military flight simulation and virtual reality to implement a system for efficiently modeling and simulating urban environments. This system combines relatively simple 3-dimensional models with aerial photographs and street level video to create a realistic model of an urban neighborhood, which can then be used for interactive fly, drive and walk-through demonstrations. A separate mode of interaction allows three-dimensional selection of objects in the scene. Once selected, an object can be removed from the scene (simulating, for example, the removal of a building from a lot), or used as a reference to information via a Universal Resource Locator (URL) or World Wide Web (WWW) address. This approach allows the association of a rich (and virtually infinite) assemblage of information with the 3 dimensional graphic entities located within the visual database.

The primary benefit of the model is the ability to navigate the city in real-time, at a rate of about 30 fps. Users can see what a proposed building would look like on its site, change the cladding material, and witness in an instant what 15 years of growth will do to a sapling.

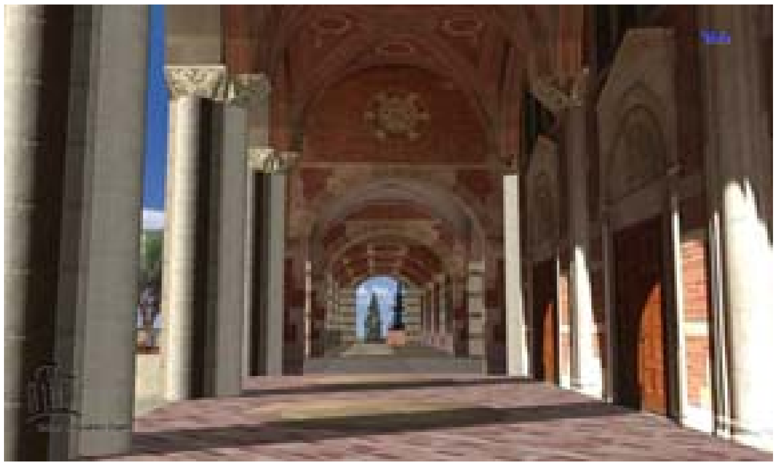


Fig 4-2 Virtual UCLA Campus

The team's urban simulation system has proven to be an extremely useful tool for exploring potential design solutions. It is possible to evaluate alternatives rapidly and in more detail than through more traditional analysis. Results of the planning/design process are illustrated visually, allowing the client or community to view a proposed environment in a realistic fashion and become informed participants in the decision-making process. The strength of the simulation system is the elimination of complex blueprints, charts, and other hard-to-understand traditional representational methods. Instead, viewers can easily ‘place’ themselves within a digitally accurate perspective representation of a proposed development and better assess the project’s impact. To model an urban area, plan view aerial photographs are used as the base image. Streets and blocks are identified, outlined, and inserted into the database. Video images from a street-level survey of the study area are then fed directly into the computer, perspective- and color-corrected. Once all of the image data has been collected, MultiGen Creator™

is used to create the appropriate built-form geometry and apply the textures. MultiGen Creator™ is the primary 3D modeler used by the team. The interface and simulation software is run on SGI® workstations with Reality Engine^{TM} graphics hardware allowing extensive real-time texture mapping.

Historic Reconstruction and Education

Real-time virtual reality offers unique opportunities for recreating and exploring historic environments. Because real-time virtual reality approximates the experience of a building in a manner not feasible until recently, an educator can 'tour' students through a modeled environment with the same freedoms as in the physical world. She can choose to stop, turn, change directions, look up at the ceiling, examine a painting, or look out the window - thereby providing the students with an understanding of the modeled environment unequalled by static two-dimensional representations.

A real-time visual simulation model of the Herodian Temple Mount was developed jointly by the Urban Simulation Team at UCLA and the Israel Antiquities Authority. The reconstruction is based on the recent excavations at the Temple Mount. The purpose is to show the viewer a real-time virtual walk-through of the Herodian Temple Mount as it looked prior to its destruction by Roman troops some 2,000 years ago.

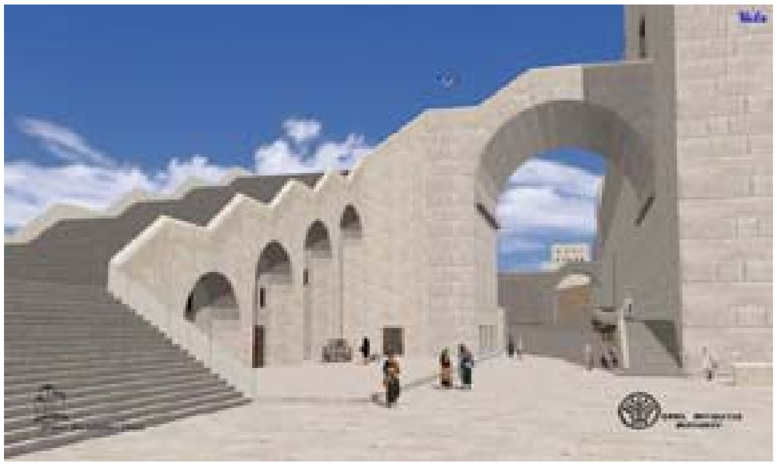


Fig 4-3 Virtual model of Jerusalem’s Temple Mount

‘The 95-megabyte binary file that comprises the virtual temple also works from a computer system housed in a wide-screen theater built into an underground chamber of a seventh-century Umayyad palace' (Eftekhar, 2001). The visualization system takes the group on a free-roaming walk-through of a city no one has seen for nearly 2,000 years—the Jerusalem of Biblical times. Based on questions and interests from each new audience, the spectacular images, the ability to zoom in and out, and the direction of the walk are manipulated on the spot by the host, using SGI® technologies.

The site was built on a digital terrain model using photo textures and the Onyx2 system's massive 256 MB of dedicated texture memory. Actual site photographs were used as a basis for the model that eventually grew to more than 200 individual computer files nested together hierarchically, including more than 300 individual texture maps.

Another example of Virtual heritage reconstructions is the one created for the Virtual Notre-Dame project (VRND), from Digitalo Studios. The primary benefit of VRND was to first introduce photo-realistic real-time technology in the area of desktop computers, through the reconstruction of the Cathedral. Unlike circular panoramas or slow and clunky VRML worlds, VRND is a 'real' VR application with multi-user chatting and virtual tour guides. The entire self-installing viewer and cathedral can be downloaded from their website, making it globally accessible. Internet Explorer supports external applications like Unreal server URL's (unreal://server.com) - automatically launching the world when a user clicks on hypertext links. This launches the world from the VRND website.

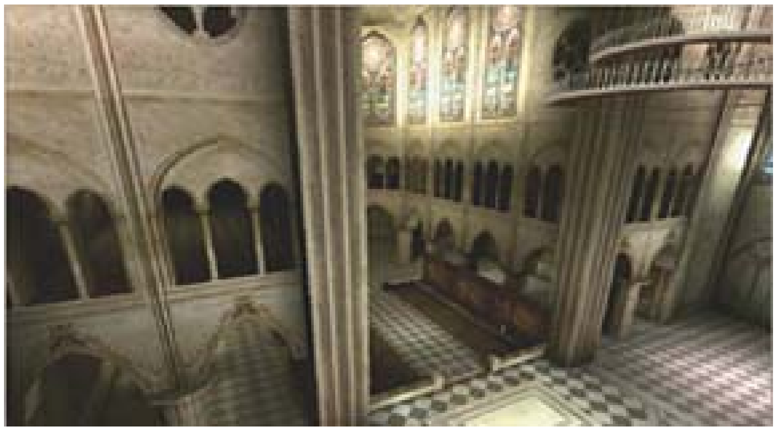


Fig 4-4 Virtual Notre-Dame Project

Available through both CD-ROM and a high speed internet server, allows thousands of visitors, those unable to experience the cathedral in person, to access this reconstruction from anywhere on their computers. The minimum system configuration should be Intel® Pentium® II 233 with 32 MB of memory and 1MB video card. Again, software rendering looks great but running a TNT or TNT2, Voodoo^{TM}2 or 3, or any other accelerator dramatically increases the frame rates and screen resolutions not to mention the colored lighting and special effects.

Virtual Florida Everglades: Virtual World Heritage Project

This project involved the design and development of a richly detailed three-dimensional virtual environment, which could be used to educate the public and promote ecological awareness, especially important for endangered environments like the Florida Wetlands. It was designed for a location-based installation with a large screen and multiple viewers by Victor J. De Leon and H. Robert Berry of Digitalo Studios.

The research concentrated on high-end three-dimensional rendering engines that could be used to create and process the real-time imagery and achieve lifelike atmospheric effects combined with audio, animated creatures, and a unique exhibition and presentation system.

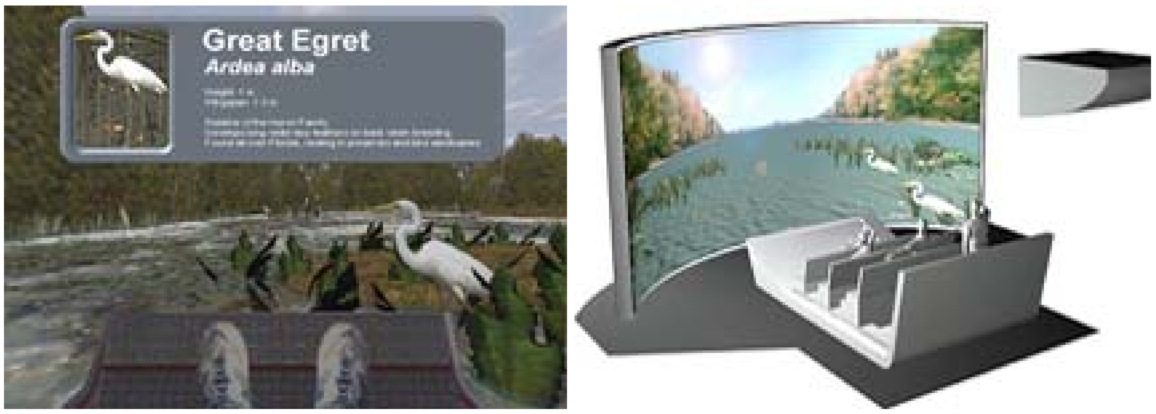


Fig 4-5 Virtual Florida Everglades Project

The museum exhibit was a 140-degree panoramic display system. The simulation would simply run continuously, allowing spectators to grab hold of the joystick and navigate the airboat vehicle throughout the environment. The spectators would be seated on simple bench-type seating arrangement with one main seat in front, controlling the airboat vehicle's path through the simulation via a simple joystick controller.

The Windows® NT/98 platform and Epic's Unreal 3D Engine (using OpenGl®) was used for developing the application. A solution from an industry that has been dealing with heavy polygon processing on low-end equipment- the videogame industry. The final textures were optimized and compressed along with the entire world geometry and lights to produce an unprecedented 45 fps. The minimum system requirements for the final model to run was at least 16 MB of video memory and be powered by two Intel® Pentium® II 400 processors.

4.2 Survey of Available Virtual Reality Toolkits

EON Studio^{TM} is a PC-based 3D interactive software product developed by EON Reality, Inc. It is a complete Graphical User interface (GUI) based tool for developing real-time 3D multimedia applications focused on E-commerce/marketing, E-learning/training and Architecture. The development process includes importing different 3D objects, usually originating from different modeling tools like 3ds max^{TM}, Lightscape etc., or from different CAD systems such as ArchiCAD®, Pro/Engineer®, or CATIA®. Once imported, behaviors can easily be associated with the models through EON's intuitive graphical programming interface, scripting or compiled C++ code. Finally the simulations can be deployed over the Internet or stand-alone on a CD-ROM. Simulations can also be integrated in other tools like Microsoft Powerpointl®, Mocrosoft Word®, Macromedia® Authorware®, Macromedia® Shockwave® Studio, Macromedia® Director®, Visual Basic etc.

EON Studio can import several file formats including, 3D Studio, ArchiCAD, VRML2, DXF, OpenFlight and Lightwave. All bitmaps (textures) are converted to suitable formats (.jpg, .png or .ppm) for importing. Real-time rendering features include proprietary developed algorithms for anti-aliasing, transparency, environment mapping, shading, reflections, shadows, level of detail support etc.

The cost for EON Studio is US $3,795. It has Support for advanced display and user interaction with systems like Concave Reality System^{TM} (seamless cylindrical projection wall; starting at US $295,000), Immersive Reality System^{TM} (consisting of EON Immersive software, data gloves, stereo capable HMD's, motion tacking system; starting at US $39,995) and Desktop Reality System^{TM} (consisting of EON Studio, shutter glasses and PC workstation; starting at US $9,950).

EON Raptor^{TM} Web Studio, is a plug-in for Discreet® 3ds max^{TM} and Autodesk VIZ®, for real-time viewing of large models and creation of interactive content for the internet. Using EON Raptor, it is possible to create interactive real-time 3ds max^{TM} content with intuitive controls. Interactive behaviors can also be authored without additional programs outside 3D Studio.

Once animations have been created in 3D Studio, interactivity can easily be added and the animation can be published to the Internet using Raptor's wizard driven web publisher. The completed application can be published to a web page for viewing with a browser, exported as a standalone file for viewing with the free EON Viewer. It can also be exported to EON Studio^{TM} for further editing.

The workflow for EON Raptor^{TM} consists of creating the 3D model and animation in 3D Studio. EON Raptor also allows real-time viewing of the 3D data at anytime within 3D Studio. Interactivity can be added into the model using the interaction wizard. The model can then be published to the internet using the web wizard or published to a CD-ROM.

EON Raptor^{Tm} creates a highly compressed file format with support for user-customizable geometry and texture compressions. All 3ds max^{TM} content can be exported, including animations, lights, cameras, parent-child relationships, pivot point, multi-materials, etc.

Sense 8 World Toolkit®, is a cross platform software development system for building high-performance, real-time, integrated 3D applications for scientific and commercial use. WorldToolKit® R9 has the function library and end-user productivity tools needed to create, manage, and commercialize applications. With the high-level application programmer's interface (API), one can quickly prototype, develop, and reconfigure applications as required. WorldToolKit R9 also supports network-based distributed simulations and supports Immersive Display Options for configuring and supporting complex projection display systems such as the CAVE^{TM} and ImmersaDesk^{TM}, as well as a large array of interface devices, such as headmounted displays, trackers, and navigation controllers; further it has added support for Reflection and Environment Mapping.

The architecture of WorldToolKit® R9 has been designed to incorporate the power of scene hierarchies. This efficient visual database representation provides increased performance, control, and flexibility through features such as hierarchical object culling and efficient use of transform information and level-of-detail switching.

Platforms supported are Windows® NT/95/98, OpenGL®, Win95 D3D and OpenGL, SGI® - SUN^{TM} - HP Unix®. Developers can bring their CAD files into WorldToolKit® or World Up® and efficiently produce 3D interactive and virtual reality applications. Engineering Animation Inc.'s (EAI's) CAD solution for large-model rendering involves a two-step process. First, EAI's translators convert CAD data-such as Pro/Engineer®, CATIA®. Unigraphics, and IGES files-into DirectModel/JT format. These Direct Model/JT formatted files are then brought into WorldToolKit® or World Up® using the Sense8's new CADLoader DirectModel technology. Once in WorldToolKit® or World Up®, the CAD data can respond to complex behaviors and interact with a variety of simulation input/output devices, adding interactivity to the previously static CAD data.

There are several **Multigen Paradigm Runtime Products** specifically developed for real-time simulation. Multigen Creator^{TM} is a comprehensive software toolset for real-time 3D modeling. Multigen Creator^{TM} and its options deliver a multi-purpose polygon based authoring system for the rapid generation of optimized object models, high-fidelty terrain and synthetic environments.

MultiGen Creator^{TM} simulates almost all real-time rendering techniques for true WYSIWYG (what you see is what you get) interactive modeling, enabling you to build within your world and preview interactively within your developing scene. OpenFlight files can be visually inspected and exercised in a vega-based viewer for early runtime validation.

Creator^{TM} is available for SGI® IRIX and Windows® NT 4.0, 98 or 2000 platforms. Its features include low polygon count modeling, efficient organization, Binary Space Partitions (BSP) modeling abilities, compatible with leading real-time rendering systems. It has Level of Detail (LOD) and bounding volume tools and allows setting up of different nodes like the LOD, Degree of Freedom (DOF), BSP, clip, which are especially important for real-time visualizations.

Vega^{TM} is Multigen-Paradigm's software environment for the creation and deployment of real-time visual and audio simulation, virtual reality, sensor and general visualization applications. Its Baseline is C language API. Vega's modular environment allows users to mix and match necessary feature sets or create their own custom functionality within the Vega environment.

Game Engines

'Virtual Reality is still considered by many to be in its infancy, therefore, it is given that the proper tools and skills necessary for its ideal construction are limited only by what previous research has forged to this day' (S.Fischer, 1990).

There have been several attempts now, exploring solutions using not the traditional CAD technologies at all. Instead looking at an industry that has been involved in the creation of real-time virtual worlds for nearly a decade now: the game industry. Game engines can provide a viable alternative method for creating interactive, real-time VE's while still maintaining a low-cost development and execution platform.

Game engines usually consist of three main components that are typically used in sequence: a set of plugins that easily move content from a modeling package into the engine, a set of tools that make it easy to optimize and use that content, and the C++ run-time API that constitutes the core real-time graphics engine. Most game engines have an editor where models can be imported in from other modeling packages and functionalities added. This is a simple user friendly icon based interface. Most of the geometry, textures, texture effects, animations, particles and other effects are supported.

Game engines offer powerful optimization and tools for real-time rendering and visual effects. Run-time performance optimization features like culling and sorting, and portal based visibility enable applications to render and manage large datasets. Current game engines support dynamic RGB lighting. These dynamic lights can be any color and may be point, infinite, or spotlights. Specular highlights and pre-lit vertices are also supported.
There is support for infinite detail texture mapping and a wide range of texture effects, including Multi-textures, such as lightmaps, dark maps, gloss maps, decals, etc; projected textures, including projected lights and shadows (for interesting lighting

effects); animated textures, for effects such as fog, fire, smoke, and explosions; and environment maps.

As a model is developed, the complexity increases as the level of realism (textures, lighting, abstract elements, etc.) increases. This increases rendering time, and drops the framerate of a real-time walk-through quite noticeably. Game engines with their powerful optimization and rendering capabilities do not suffer from these problems, being able to handle over 60,000 polygons in a single level, fully textured and lit. Features like model- and texture-based continuous level- of-detail and mipmapping allow for larger textures to be used in levels and on more detailed meshes, without affecting framerate.

Given a base configuration of a Pentium®-class workstation running at 200 MHz with 64 MB RAM gives tolerable performance. Faster processors, 3D accelerators, etc. all enhance the experience.

Current shortcomings of game engines are their high licensing fees, no official manuals, lack of awareness and acceptance and technology limitations.

There is increasing support for the use of this technology for development of VE's and architectural design and visualization. Gifu University's Virtual Systems Laboratory is working with Epic Games to help solidify the UnrealEd development environment, and provide custom integrated tools, from rapid proyotyping systems to support for advanced VR devices such as HMD's, Polhemus® trackers, and more (Miliano, 1999).

OPEN SOURCE SOFTWARES AND TOOLKITS

All the softwares and toolkits discussed above are proprietary and licensed. Besides these, there are some open source softwares and toolkits for developing and implementing 3D virtual environments. The 'Open Source Model' allows for evolution of the software by providing complete access of the source code to the users. The users can add features for their use and submit back to the community, participating in the overall development process. With access to the source code, people using a wide variety of platforms, operating systems, and compiler combinations can compile, link, and run the code on their system to test for portability. There are no licensing fees, neither development nor runtime. Upgrading is mostly a technical decision, not just a financial one. The choice to upgrade depends upon the user. The source code is available for support at any release level. Some of these open source softwares are discussed below:

DIVERSE

Device Independent Virtual Environments- Reconfigurable, Scalable, Extensible

"DIVERSE is a highly modular collection of complimentary software packages designed to facilitate the creation of device independent virtual environments. It is free/open source software, containing both end-user programs and C++ APIs" (Kelso et al, 2001). It runs on GNU/Linux® and IRIX® systems. DIVERSE is currently being written at Virginia Tech by a team of faculty, research associates and students who are (or were) affiliated with Virginia Tech's University Visualization and Animation group.

DIVERSE is comprised of three components:

- DIVERSE graphics interface for Performer (DPF)- This provides a framework to implement 3D virtual environment and desktop graphics applications by augmenting OpenGL Performer^{TM}.
- DIVERSE GL (DGL)- DGL is an extension to DIVERSE, which supports rendering with the OpenGL programming interface. It is like DPF without the scene graph. It is currently only available as a beta pre-release.
- DIVERSE ToolKit (DTK)- DTK is a separate standalone package. It is used by DPF and DGL to provide access to local and networked (real and virtual) interaction devices.

DIVERSE is designed in such a way, that a user can use parts of the code that he needs, without being forced to follow a particular design for his code, or having to add features that are not required.

DIVERSE provides a common interface to interactive graphics and/or VE programs. Using DIVERSE the same program can be run on the CAVE^{TM}, ImmersaDesk^{TM}, desktop and laptop without modification. In addition, DIVERSE also provides a common API to VE oriented hardware such as trackers, wands, joysticks, and motion bases. A "remote shared memory" facility allows data from hardware or computation to be asynchronously shared between both local and remote processes.

VR JUGGLER

VR Juggler is an active research project headed by Dr. Carolina Cruz-Niera and a team of students at Iowa State University's Virtual Reality Applications Center. It is an open source virtual reality application development framework.

VR Juggler provides virtual reality (VR) software developers with a suite of application programming interfaces (APIs) that abstract, and hence simplify, all interface aspects of their program including the display surfaces, object tracking, selection and navigation, graphics rendering engines, and graphical user interfaces. An application written with VR Juggler is essentially independent of device, computer platform, and VR system. VR Juggler may be run with any combination of immersive technologies and computational hardware.

VR Juggler is the core API to build applications and is supported by: JCCL, the VR Juggler Configuration and Control Library; Gadgeteer, the I/O device management system; VPR, the VR Juggler Portable Runtime layer; Sonix, the library for portable sound objects; Tweek, the portable GUI for applications.

CHAPTER 5: METHODOLOGY

5.1 Introduction

This thesis studies the issues involved in 3D photo-realistic construction and real-time simulation of architectural models. The aim is to develop a simple, complete and open methodology for the creation of real-time photo-realistic virtual environments for the new generation of low-end and low-cost VR systems.

Although the work of Zhukov et al (1998) and Carmack, Abrash (1997), pioneered the use of real-time photo-realism, their solutions are nowadays available mostly through proprietary, costly real-time 3D game Engines.

This study describes a methodology based on widely available standard tools to build a compelling, interactive and highly realistic virtual environment where the static lighting is based on pre-calculated lighting data. This data is obtained by the use of photo-realistic lighting models, such as the radiosity global illumination technique. The resulting lighting information is stored in 2D textures called lightmaps, which according to the multi-pass multitexture technique are blended in real-time with the model's ordinary texture maps. The sample 3D application that is used for real-time rendering and visualization is based on standard open programming languages and API's. These include C++ and OpenGL®. Using this process one can create and navigate real-time virtual environments on any operating system that supports the above (Windows®, Mac® operating system, Linux®, IRIX® SGI® system). The VE's are modeled using professional commercial 3D modeling packages and 2D architectural data for maximum precision and accuracy.

5.2 Preparatory Phase

The first phase of the methodology was based upon two-dimensional architectural and photographic data to create a three dimensional authoring, modeling and representation of the architectural model. Also during this phase texture lightmaps were created. These contained the pre-calculated realistic lighting information and were used to modulate the diffuse model textures in later stages, during the real-time visualization. The output of this stage was a 3DS file, which had the model geometry, diffuse texture maps and lightmaps. This 3DS file was then loaded by the real-time engine, which supports multi-pass multitexture rendering, and was visualized in real-time using the 3D application.

5.2.1 Data Acquisition

The architectural model that was constructed using the proposed methodology was 'Church on the Water', designed by Tadao Ando. Located on a plain in the province of Hokkaido, Japan, this church has a plan of two overlapping squares of different sizes. The building faces towards a shallow artificial lake, created by the diversion of a nearby stream. A free-standing, L-shaped wall extends along one side of the lake, ascends alongside the wall, leading to the top of the smaller volume where, within a glass enclosed space open to the sky, four large crosses are arranged in a square formation. From this point the visitor descends a darkened stairway to emerge into the rear of the chapel. The wall behind the altar is fully glazed, providing a panorama of the lake, in which the large crucifix is seen rising from the surface of the water. This wall can be slid entirely to one side, directly opening the interior of the church up to the natural surroundings.

Location	Hokkaido, Japan
Construction Period	April 1988-September 1988
Structure	Reinforced concrete
Site Area	6730.0 m^2
Building Area	344.9 m^2
Total Floor Area	520.0 m^2

Table 5-1 Architectural Data for Church on the Water

The primary data used for the 3D construction was collected from two main sources:
Church on the Water, Church of the Light, Tadao Ando (Philip Drew, Architecture in Detail, 1996 Phaidon Press Limited).
The Colours of Light, Tadao Ando Architecture (Richard Pare, 1996 Phaidon Press Limited).

These books provided architectural sections and plans that were used as a basis for the 3D modeling of the Church. Images and photographs were studied and used to model architectural details not represented on the plans, and at a later stage they were used as a base for extraction and creation of the material textures. An attempt was made to construct a highly accurate and detailed representation.

5.2.2 3D Modeling

The 3D modeling of the Church was done keeping in mind that the final model would be visualized in a VR real-time environment. Polygonal modeling techniques were mainly used in order to achieve the best trade-off ratio between the precision of the 3D representation of the model and the maximum amount of displayable polygons for a real-time platform at an acceptable frame rate.

The initial detailed model had 99,700 polygons, consisting of the building, interior furniture and detailing, site, landscape and trees. This was optimized for real-time using several optimization techniques, bringing the polygon count down to 65,000.

The first step was to carefully examine the model and remove surfaces that were not visible. However thoughtful the initial modeling, using software packages like AutoCAD and 3D Studio usually creates a lot of surfaces that no one will ever see. A second point to consider was to analyze where detail was really needed. Say, incase of a light switch that has been modeled, geometry can easily be deleted and replaced by a texture of a light switch. Rather than intricate modeled details, it is the lighting, simulation and smoother frame rates that are more noticeable and should be kept in mind when modeling for real-time simulation. Current and some of the earlier computer games offer a lot to learn in this respect. They have some very nice environments that have very few surfaces. Most detail coming from the textures.

5.2.2.1 Guidelines for Modeling

a. Models

A.1 Common Primitives

The surface models which represent geometry in virtual environments are typically made up of polygons as their most basic element. These planar polygons are commonly three or four-sided, and can be combined to create an infinite number of shapes and volumes. Most CAD packages generate dozens of different kinds of 3D primitives, and many of these are valid for describing the surface models we wish to render in real-time. Almost any CAD package can be used to generate the initial model data, which we will later embellish to create our virtual environments, including AutoCAD®, form•Z, 3D Studio VIZ®, 3ds max^{TM}, MicroStation, and Alias|Wavefront™.

Whichever CAD package you choose to work with, however, will generate data, which will need to be translated into a format, which can be simulated in real-time. Unfortunately, every software package has its own file format and each one interprets other file formats in its own way. Thus, there is little predicting which primitives are supported by other software packages, and specifically how certain primitives will be translated. You really just need to experiment to find what elements from a specific package will translate well into others, and with what unique characteristics.

Of all the primitives available, a combination of only a few can be used to create most of the architectural virtual environments: extruded lines and polylines, and 3D faces. With practice one can represent almost anything with these few primitives, and easily maintain a low polygon count in the virtual environments.

The following table lists AutoCAD objects and properties and the Autodesk VIZ objects they convert to when imported from AutoCAD to Autodesk VIZ.

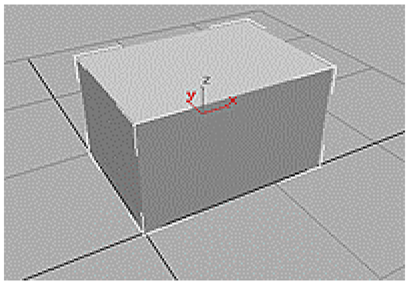
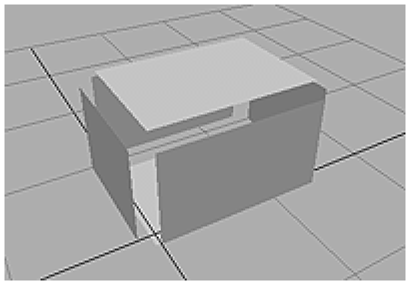
AutoCAD Object or Property	Imported VIZ Object
Line	Spline Shape
Arc	Arc Shape
Circle	Circle Shape
Ellipse	Ellipse shape
Solid	Closed Spline Shape
2D Polyline	Spline Shape
3D Polyline	Spline Shape
Spline	Spline Shape
3D face	Mesh Object
Polyline Mesh	Mesh object
Polyface Mesh	Mesh Object
Region	Spline Shape
Blocks	Objects or group by option
Thickness property	Extrude Modifier
Polyline Width	Spline Outline

Table 5-2 AutoCAD Object or Property conversion to 3D Studio Viz

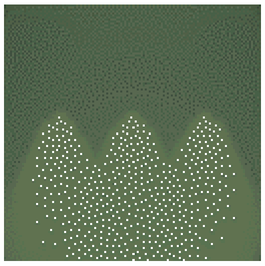
A.2 Layers

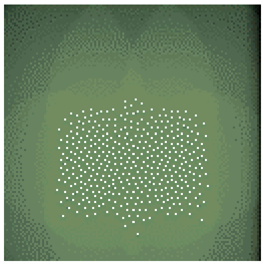
It is a good approach to structure CAD data into layers while constructing virtual environments. **Typically, one needs to have a different layer in the CAD file for each different material in the virtual environment**. The reason for creating a layer for each material is that the layers (in AutoCAD) are usually converted into distinct objects (in 3D Studio). One *may* even need to have a distinct layer for each different object of the same material.

One implication of these conditions that is not immediately obvious is that buildings with different texture maps on their facades, and rooms with different texture maps on their walls, must also have each of their facades/walls on a different layer. Often this is overlooked in the initial construction of the geometry, which represents these elements, and it may be necessary to "break up" these buildings and rooms after they have been constructed on the same layer in AutoCAD and translated to 3D Studio.

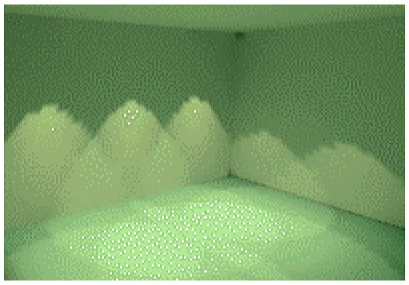



A room modeled as a single box, and then its surfaces are "broken up" onto different layers so that they become separate objects.





Different textures can be applied to each distinct surface.



Each plane on its own layer, with a unique texture applied to it.

Fig 5-1 Advantage of structuring CAD Data into layers
[*Images adapted from Dace Campbell's VR Guide*]

A.3 Tessellation

3D CAD data is broken down into its most basic elements, polygons, at the time of rendering. These polygons can be four-sided, but most often they are broken down into three-sided polygons in a process called tessellation. In fact, 3D Studio and many other rendering packages tessellate all faces into triangles as data is imported into the program, even if the data includes four-sided faces.

Although tessellation process cannot be controlled, but one can manage and create data, in a way that it is tessellated in a desirable manner.

- Triangular faces (like 3D faces with just three vertices) are not tessellated any further since they are already triangles.
- Quadrilateral faces (flat) are tessellated across one of their diagonals. It is rarely predictable which one, but this isn't often an issue if the polygon is flat.
- Quadrilateral clusters of vertices, which define non-planar surfaces are broken down into two triangular faces along one of the surface's diagonals. Again, it is rarely predictable which diagonal will be used to split the surface, and if this matters to your design, then you are better off building the surfaces as two triangles, which share an edge.
- Avoid concave quadrilaterals (or other concave surfaces), as their tessellation is unpredictable. Instead, break them up into multiple triangles before the computer tessellates the data.
- For surfaces with more than four vertices, the data will almost certainly be tessellated in an unpredictable manner. Often, this yields many oddly proportioned surfaces and long, skinny triangles. These often yield undesirable rendering artifacts, in both real-time phong rendering and especially in the processing of radiosity solutions. Plan ahead for this and build the surface out of triangular and quadrilateral primitives such that predictable shapes are generated and tessellated.

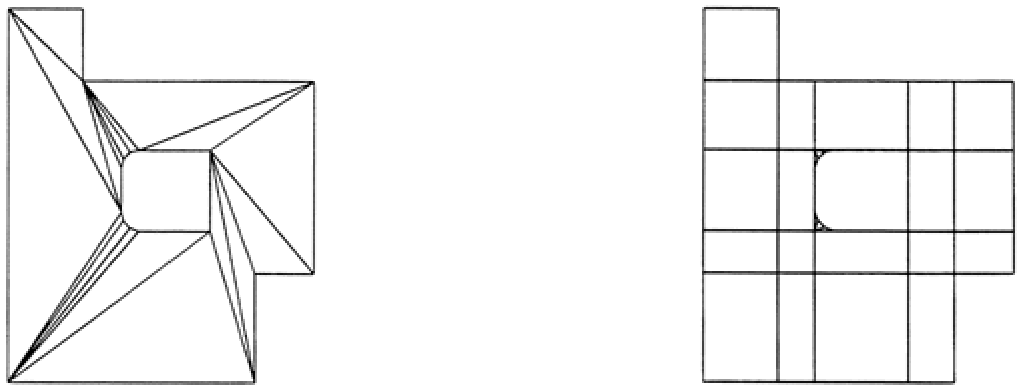


Fig 5-2 Correct Modeling for desirable Tessellation

Long skinny triangles can be avoided by modeling using quadrilateral and triangular primitives.

A.4 Polygon Management

A fundamental characteristic of a virtual environment is its ability to be simulated in real-time, which means that a computer can render a CAD model many times per second as a participant navigates through it. Whether this virtual environment is being simulated on a state-of-the art SGI® workstation, or simply a desktop PC, it has certain characteristics, which determine how well it can be rendered in real-time. Of these characteristics, the number of polygons in the model being simulated is perhaps the most significant. Basically, the more polygons you have in your virtual environment, the more slowly it will render in "real-time." Conversely, if your virtual environment has been modeled with fewer polygons, it will render much more quickly. Although each kind of computer will render an environment at a different rate, and although "real-time" is a slippery term to define, we are basically aiming for a frame rate (number of renderings per second) of 15-30 Hz.

Often, the designer/builder of the virtual environment must make trade-offs between how complex a virtual environment can be versus the capability of the environment to be rendered in real-time. There are several strategies available for building efficient virtual environments, like billboarding and level of detail techniques, but many times **one can reduce the polygon count simply by modeling efficiently and removing unnecessary polygons in the virtual environment.** Below are some examples:

Planes vs. Boxes

In architectural representation, one can often get away with one plane (broken into two triangular polygons at rendering time) for many planar elements, such as walls, windows, floors, etc. The use of the box primitive for this is often inefficient from a polygon-budget point-of-view, because each six-sided box is typically broken down into twelve polygons at rendering time. Try constructing an architectural element out of faces (like 3D faces in AutoCAD®), and limit the number of faces you construct to those that will be seen/experienced in the virtual environment. For example, a window constructed out of a box-like element in a solid modeler can be constructed and rendered with only 1/6th as much information if it is built as a rectangular plane (12 polygons vs. 2 polygons).

Coplanar Surfaces

Often in architectural representation, 3D primitives share a face. That is, objects rest upon or butt up against other objects, and coplanar surfaces are generated in the virtual environment. This is undesirable for two reasons. One is that in computer graphics, the rendering of coplanar surfaces often results in undesirable artifacts like streaks or "flashing" surfaces. The other reason coplanar surfaces should be avoided is simply that they are often not necessary in our representation of the virtual environment

Intersecting Surfaces

Just as there are many places in a virtual environment in which one can and should remove coplanar surfaces to reduce the polygon count, there are often several opportunities to remove and reduce the number of polygons using intersecting surfaces. For example, one may want to place a building on a sloped terrain, and this can be accomplished by placing a box into plane with a hole cut out of it. Don't forget to remove the bottom face of the building, which will never be seen.

Curved Surfaces

With the above examples, one can eliminate unseen polygons. Curved surfaces present a different opportunity for reducing polygon count. Curved surfaces like cylinders, spheres, and extruded arcs and circles, can generate hundreds of polygon faces at rendering time when using default file translation settings. Often, the surfaces appear perfectly smooth because there are so many tiny polygons as shown below.

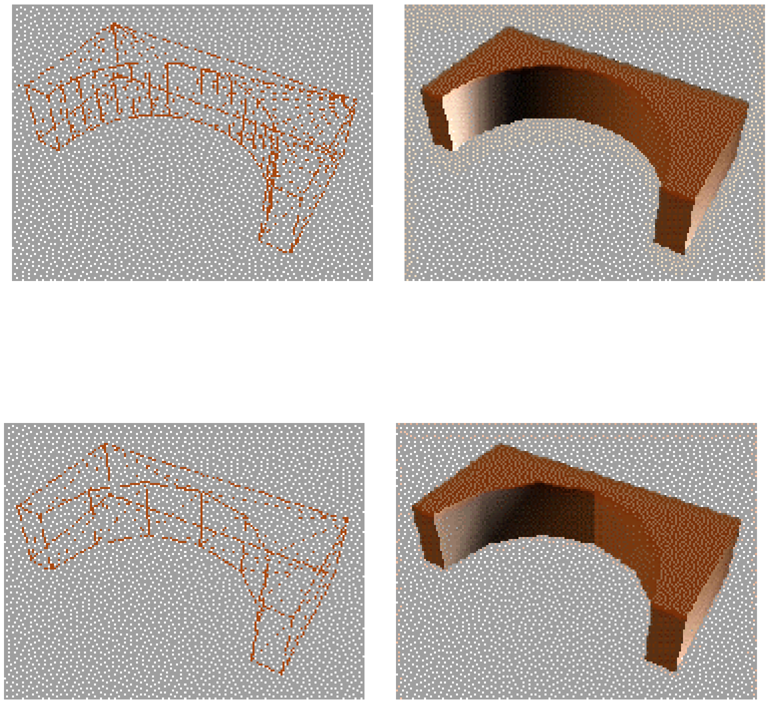


Fig 5-3 Polygonal Management for Curved surfaces
[*Images adapted from Dace Campbell's VR Guide*]

While this may sometimes be necessary for a particular rendering or effect, often the high number of polygons is inappropriate because such a finely rendered curve is not necessary. Rather, curves can be modeled with faces such that a circle may only have 12 to 24 faces. In most cases, this abstraction of a curve is acceptable for real-time simulation, and can save on thousands of polygons in the model used in the virtual environment. Additionally, these faceted surfaces can be smoothed using 'Smoothing' function available in most 3D software packages. 3D studio also provides the 'Optimize' and the 'Multires' modifiers, which are very useful for controlled reduction of the polygon count in such surfaces. The model in the diagram below has less than half of the number of polygons as the model above, and may effectively represent the designer's intent for the participant in the virtual environment.

b. Images

B.1 Types and Bit-Depths

There are many types of digital images, which can be created and used, in architectural renderings, including Photoshop®, BMP, GIF, EPS, JPEG, PICT, RGB, Targa, and TIFF. Each of these files formats has unique characteristics, which distinguish them from the others, and only a few of them are appropriate to the construction of virtual environments.

GIF and JPEG formats work well for standard textures. TGA, PNG, TIFF file formats work for textures where an alpha channel is required. RGB, Silicon Graphics Red-Green-Blue image, a standard on SGI® machines, is a 24 bit true-color, uncompressed image with three channels (one for each color of red, green and blue). Photoshop® for SGI®, as well as many standard SGI® utilities can write and read the RGB format. Also note that there are RGBA files, which is an RGB file with a fourth "alpha" channel.

Bit depth refers to the number of bits of information per pixel in an image, which is how much color information is stored to describe the image. Higher bit depths, like 24, means that 16 million colors are available per pixel, yielding more accurate color representation in the image, and larger file size. Lower bit depths, like 8, limit the color palette to 256 possible colors per pixel, but generate much smaller files. Often, for real-time rendering, GIF files or compressed files such like JPEG are used because of their small size, enabling quicker rendering.

B.2 Sizes

The size of the image, usually specified in pixels, width by height, effects the overall file size and how efficiently the image can be loaded into the computer's memory for rendering. Although there are many common file sizes (320x200, 640x480, 800x600, etc), only a few specific sizes are appropriate for real-time rendering. Appropriate file sizes for real-time rendering are those whose dimensions are powers of two. That is: 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, and so on. Use square files for nearly square images, and use rectangular (1:2, 1:4, etc) sizes for elongated images.

It's really up to you to decide what sizes to make the images. Doubling an image's size will make its file size (and load time, and space used in memory) four times as big, while cutting its size in half will cut the file size to a quarter. Often the size you specify depends on the kind of information you are trying to show, how closely it is meant to be viewed, how large the image will be in the environment, and whether it is a tileable pattern, like grass or concrete, versus a specific image like a building elevation or a horizon.

SGI® hardware has a texture memory limit of 64 MB, and most PC graphics cards do as well. NVIDIA®'s GeForce^{TM} 4 and ATI's Radeon^{TM} 8500 now carry 128 MB of onboard memory, which is double the standard 64 MB. The growing memory certainly does boost performance, but the rule is that it is never enough. Texture sizes should be kept as small as possible and another option is to use the same texture for multiple surfaces that have similar appearance.

Lightmaps are generally low resolution images, since lighting changes very slowly over a surface. It is not required to have a very high resolution texture map to capture the general lighting on a wall, if there aren't any hard shadows on the wall. In this project, I started out with wall textures at 1024x512 pixels and realized that for a lot of walls 256x128 pixels was adequate.

B.3 Alpha Maps

To reduce the amount of geometry needed to represent complex objects, a technique commonly used is "alpha mapping." Basically, this is a way to add information to an image to make portions of it transparent. With this technique, we can represent an object, like a tree, as a flat plane with a masked-out texture map of a tree applied to it. One needs to create an *opacity map* and an RGBA image to be used for alpha mapping in real-time rendering.

An opacity map is typically a greyscale image (often just black and white), and the value of its pixels determines how opaque (or how transparent) a texture map is. White yields a fully opaque pixel in the texture map, while black yields a completely transparent pixel and therefore it won't be rendered. Grey values generate semi-transparent colors in the texture map.

Care should be taken to create the mask so that it lines up pixel-for-pixel with the original texture map. Once this opacity map has been created, it should be resized so that it will load efficiently into memory at rendering time. It should be resized to match the texture map size.

Once the opacity map and the original image have been resized, they can be used in a couple of different ways. To render the image correctly in 3D Studio while defining materials for surfaces, the images can be used "as is”, in the opacity channel. To render the image in real-time on an SGI®, convert each image to an RGB format using standard SGI® utilities (like "fromgif") or Photoshop® for SGI®.

For this project, the tree images were created as RGBA images, where the alpha map was placed in the alpha channel using Photoshop®. The images were saved as TIFF format, which supports the alpha channel.

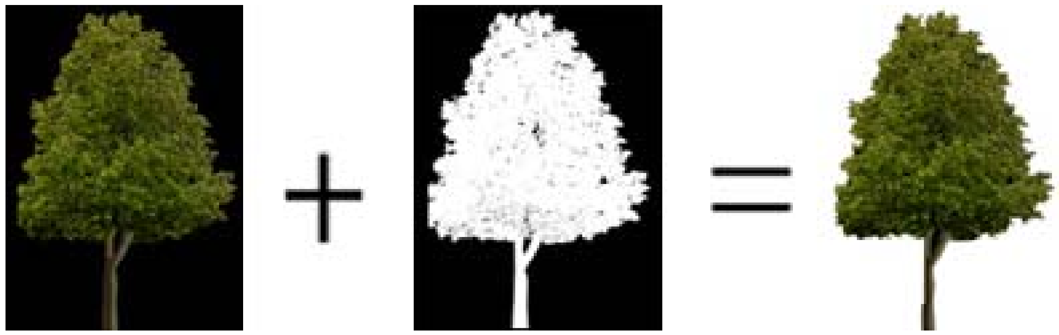


Fig 5-4 Alpha mapping to achieve transparency

C. Surfaces

C.1 Normals

In computer graphics rendering, each polygon has, by definition, a "front" and a "back," defined by a *normal* to the polygon. That is, each polygon face has one imaginary line (ray, actually) projecting at 90-degrees from its surface. This normal is commonly, but not always, defined by the *right-hand rule*, which means that if the order of the vertices of a polygon is counterclockwise, the normal points up (if you take your right hand and curl your fingers with the thumb up, the ordering of the vertices is indicated by the direction your fingers point, while the direction of the normal is represented by your thumb).

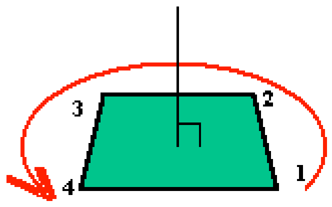


Fig 5-5 Right hand rule for defining Surface Normals

The normal of a polygon has several uses in rendering, but of greatest importance to us is that by default it can increase display and rendering performance by hiding faces whose normals point away from the participant in the virtual environment. This default setting has advantages and disadvantages, and one needs to be aware of them as well as be able to modify this characteristic to best suit one's needs. In most cases, it is an advantage that, by default, only one side of all polygons is rendered. This will increase rendering performance by almost twice as much as if both sides were rendered. However, the direction of the normals in the model should be controlled to avoid having "holes" in the geometry. A complete surface, a mesh for instance, will appear to have holes in it where normals of individual polygons face away from the participant, when viewed from one side. In fact, when the other side of the mesh is viewed, only these polygons will show, with the other polygons appearing as holes.

Sometimes, it is not good enough to have one-sided faces in the virtual environment. Thin, planar elements occur in architecture all the time. While in the real world, these objects actually consist of a thin volume with two sides, often in a virtual environment these can and should be represented as planes rather than volumes. In this case, it is necessary to force two-sided rendering of the surface. This tells the computer at rendering time to ignore the normal of the polygon and render it despite the fact that it may face away from the viewer.

Keep in mind that **you should limit the number of two-sided polygons in your virtual environment**, because with the number of two-sided faces, the rendering time of the environment is increased by almost twice as much. Like many other things in the construction of virtual environments, the number of two-sided polygons that can be used is a judgment call, and one needs to experiment.

C.2 Smoothing

Objects with polygon faces that share edges can appear faceted or smooth, depending on the characteristics specified when representing curved surfaces in virtual environments. With smooth surfaces, a curved object can typically be modeled with much fewer polygons to achieve a similar effect as that of a faceted object with a high polygon count. Controlling the smoothness of an object is typically done in two ways: settings in the conversion of files (like between DWG, DXF, 3DS) and by editing smoothing groups in packages like 3ds max^{TM}.

D. Environment

D.1 Lights

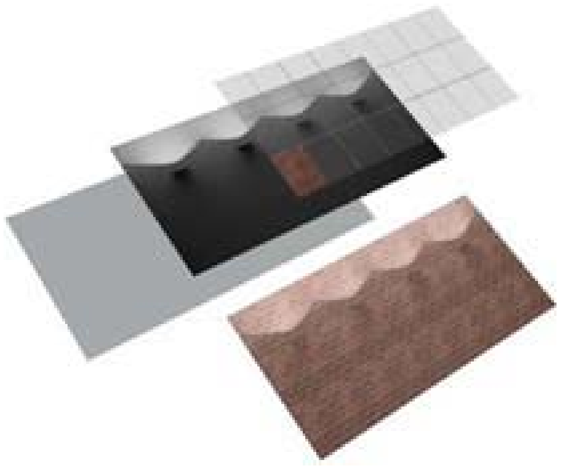
One key element, which determines how objects in the environment are perceived, is the addition of lights to the model. Without explicitly adding lights to the environment, many real-time rendering packages will add a simple "headlight" to the environment. This is a light, which attaches to the participant’s position and shines directly forward. Sometimes this is a desirable effect, but more often it does not accurately represent the desired lighting for an architectural environment. Other rendering packages will render the world without a light source, which means that the objects may show up dimly lit by ambient light settings, if at all. An exception to this is the real-time rendering of radiosity solutions.

There are many kinds of lights, which can be added to the model to achieve almost limitless effects. In general, limit the number of lights used for real-time rendering. The more lights added to an environment, the slower that world will render because the effects of each light source must be added to the next.

5.2.3 Global Illumination simulation and Lightmap generation

Radiosity lighting simulation method is used in order to create a realistic 3D model and extract from it the light distribution in the form of 2D lightmaps. For static lighting in a virtual environment, the diffuse component on any surface remains the same from any angle. Because of this view independence, the contribution of light to a surface can be captured in a 2D texture map (Moller, 1999).

For the actual radiosity simulation and for generation of lightmaps, the Discreet® Lightscape 3.2 commercial software package is used. In real-time applications, it is most efficient to separate the lightmaps from the texture maps because texture maps generally contain full color information that is often repeated or tiled over a surface while the lighting information is generally monochromatic and non-repetitive. Taking an example: A non-illuminated brick wall uses one 16-bit, 320x240 pixel texture map that is tiled on a surface 24 times. The lighting of the surface is stored in a single 8-bit, 320x240 illumination map that is stretched and blended with the tiled texture to produce the illusion of an illuminated brick wall. The total texture memory used is only 20.4 KB versus the 417 KB that would be required if a single non-tiled 16-bit texture map were used to produce the same effect. (Using 'mesh to texture' to create illumination maps for games or virtual sets, FAQ, Discreet® website)



Wall Geometry + illumination map + Brick tile = Final rendered brick wall

Fig 5-6 Using Lightmaps to simulate illumination

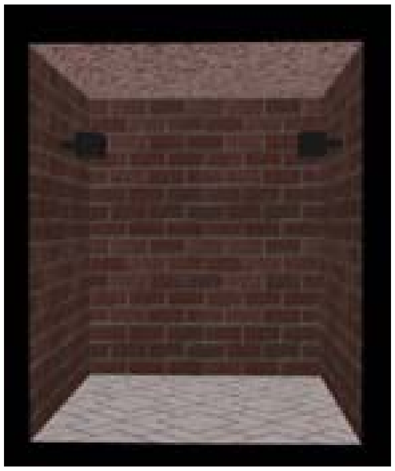
The pipeline for creating the photo-realistic model consists of:

- Creating the geometry in a 3D modeling package (Autodesk VIZ® 4.0) and exporting it to Lightscape, with its already specified materials and lights.
- Setting up the file in Lightscape.
  -Re-specifying more precisely material values according to the lightscape templates. For processing a radiosity solution, correct reflectance values of the materials is a must and this is where the templates help.
  -Setting the lights for a physical simulation- setting their intensity and luminaire processing settings.
- Processing the radiosity solution depending on the required level of detail (high). Analyzing photo metrics and adding daylight support.
- Extracting the 2D lightmaps from the 3D model using "Mesh to Texture" methods, which convert the color per vertex mesh information, to texture lightmaps.

The advantage of this technique is that there is no restriction on the complexity of the technique used for the pre-calculation of the light stored in the lightmap. In this project for example, the 'radiosity' view-independent global illumination technique is used to generate lightmaps, as described above. These maps can either be stored separately from the texture maps, or used to pre-modulate the object's texture map with their lighting information. If the lightmap is kept as a separate entity, it is calculated at a far lower resolution than the ordinary texture map to save the required memory storage. Any aliasing effects at shadow edges can be eliminated using linear texture filtering techniques.

There are two options for generating lightmaps using ‘mesh to texture’ tools in lightscape. ‘**Convert each texture to a single texture per surface’** creates one illumination map for each surface on which the mesh to texture tool is run. The second option ‘**Project all selected geometry into one texture’** creates a single texture from a group of selected surfaces. Depending on how the model is constructed, how lightscape has subdivided the surfaces in the solution model, both these options will need to be used. For example, a wall with an opening would be subdivided by lightscape into several different surfaces. These surfaces should be coplanar and reconstituted into one surface when mesh to texture is done. Sometimes it is also advisable to use just one texture for a curved surface rather than one for each of the polygons- this can drastically reduce the number of texture maps. Sometimes it is also useful to convert geometry into textures like artwork hanging on a wall. All these cases call for ‘project all selected geometry into a single texture’ else the ‘single texture per surface’ option should be used.

The important point to remember while generating lightmaps is the texture memory limit of graphics cards. The geometry itself should be highly optimized before running the radiosity solution, keeping in mind the number of lightmaps that would be generated later on. It is usually better to start with higher resolutions and then scale them down as much as possible once in the virtual set so that their appearance can be confirmed. Because of the support for mip mapping, most textures can be fairly small. Only the largest and closest surfaces, or ones that have distinct shadows require higher resolution maps, but no greater than 512x512 pixels.


Model with diffuse textures in 3D Studio

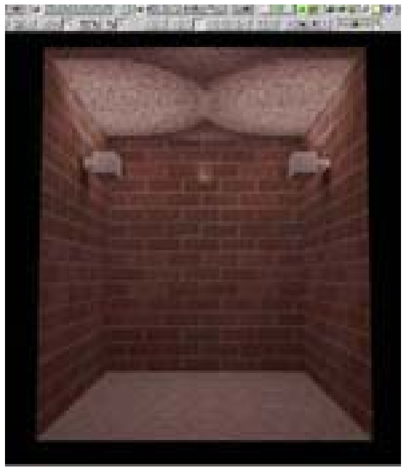
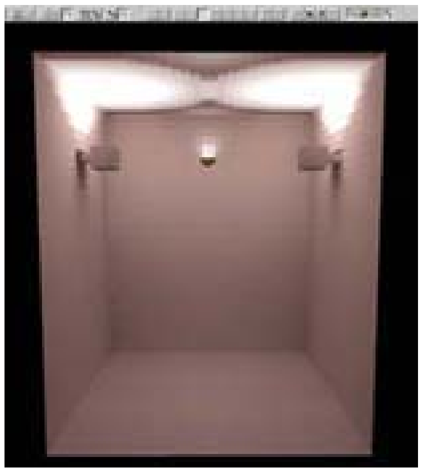
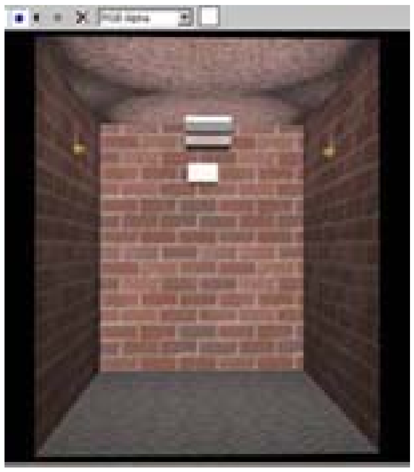

Radiosity solution in Lightscape

Fig 5-7 3D Studio rendering with diffuse textures and radiosity processed image in Lightscape.

‘Convert each texture to a single texture per surface’ option is the most automatic. It creates a lightmap for each surface that is selected. The disadvantage is that there is little control over the number of lightmaps that would be generated eventually overloading the texture memory. The second option, ‘Project all selected geometry into one texture, is more time consuming but gives more control over generation of lightmaps and in terms of keeping their number limited. One must find a balance between the geometric complexity and lighting quality, and selecting which option works best is a judgment call depending upon the model.

- **Alternative method for generating lighmaps**
  As seen, the above process can be quite cumbersome and time consuming. For this project we also tried another approach by establishing an open methodology for automatic generation of lightmaps. This significantly improves the preparatory scene production stages too.
  At present only omni lights are supported by the application. For generation of lightmaps, raytracing takes place as a preprocessor, which raytraces lightmaps for all surfaces and saves them to disc. The engine can load these during the multitexturing stage at run-time.

Lightmaps generated In Lightscape Lightmaps imported in 3D Studio

Fig 5-8 Merging lightmaps with model geometry and diffuse textures in Autodesk VIZ® 4

5.2.4 Use of Multitexture Rendering in 3D Construction

Multipass multitexture rendering is a technique that builds up a final rendered image from a combination of the results of a set of separate rendering passes (Watt et al, 2001). These rendering passes involve pixel-by-pixel operations, and in this case, blending two images together.

The preparatory production pipeline before multi-pass multitexture is applied, involves the following steps:

- Utilize a global illumination simulator to create, extract and store the 2D lightmaps.

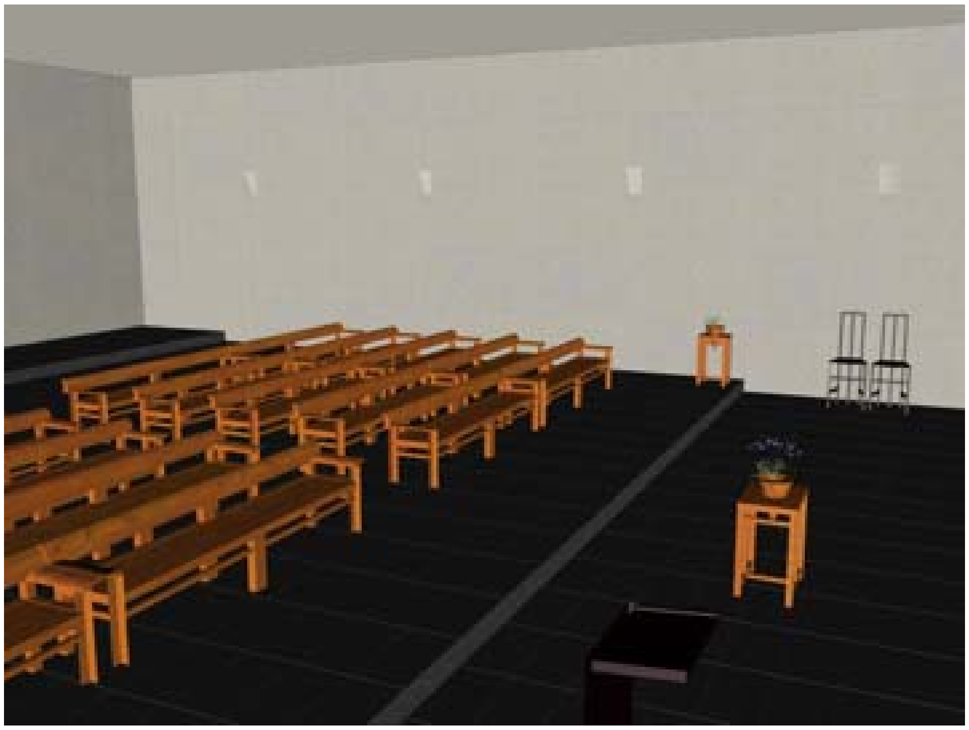
- Once the lightmaps have been generated they need to be imported back into the model in Autodesk VIZ® 4. This is done because the real-time engine requires the texture coordinates for the lightmaps in order to do multipass rendering.
  This can be done in two ways:
  -Using the lightscape plugin, import the geometry that was created in lightscape just before the radiosity solution. Although one may create an effective model that contains less than the allowable number of polygons, it should be kept in mind that the lightscape radiosity process discretizes the surfaces of the model into mesh elements (essentially creating additional polygons). Before the lightmaps can be imported the geometry has to be imported and this must correspond to the one that was used to generate the maps. So, the goal is to import the geometry with the diffuse materials as a first step.

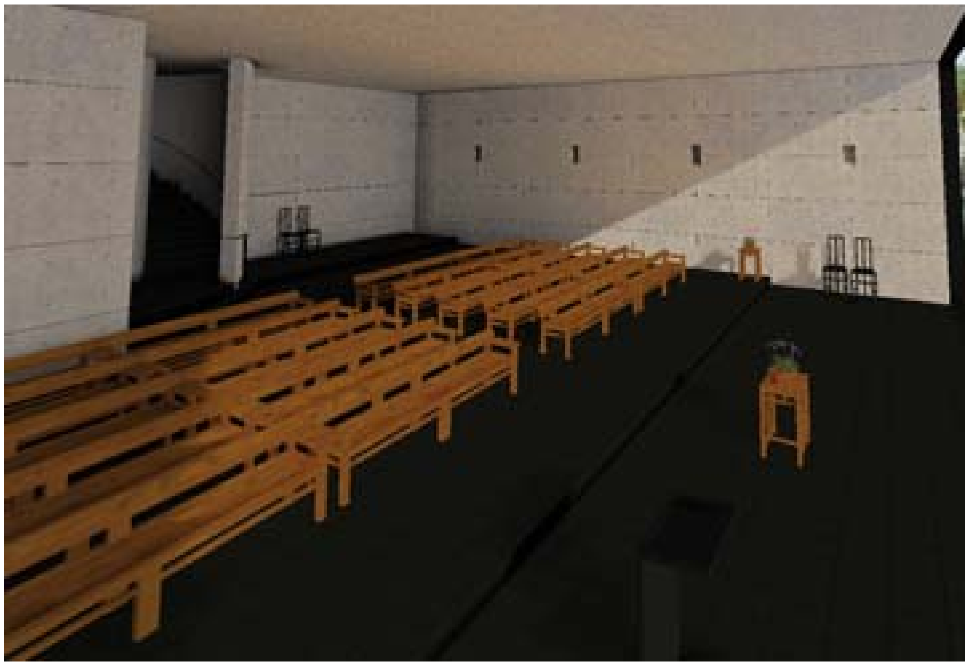
  Once the geometry has been imported, the lightmaps must be brought on top of the geometry in the self illumination channel. This is done by importing the lightscape solution file containing the lightmaps that were created using the 'mesh to texture ' methods. But in this case only the illumination maps are imported.

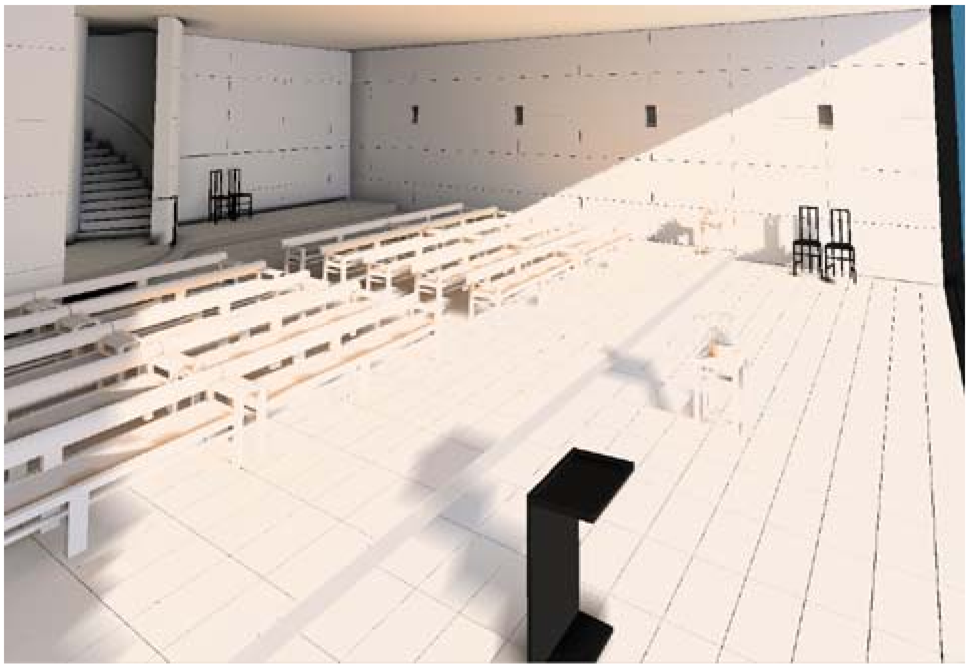
  -A second method for merging the lightmaps with the original 3D model is to texture map the 3D model manually by placing the lightmaps in the self illumination channel of the materials. This process is more time consuming, but the greatest advantage being that it allows the use of the original optimized 3D geometry (as against the one subdivided by lightscape). Hence the multitextured model at run-time would be based on the original model geometry.

- Export this 3D Studio file, which has the geometry, diffuse textures, and the lightmaps as .3DS file. Our Real-time engine at present can import only a 3D Studio (.3DS) file format.

- Visualizing the virtual environment in the real-time application, where hardware multitexturing is performed at run time to achieve the final photo-realistic model.


Model with diffuse textures in 3D Studio


Radiosity solution in Lightscape



Textures replaced by lightmaps in Lightscape

Fig 5-9 Lightmapping demonstrated in Church on the Water model

5.3 Real-Time Visualization

The 3D walk-through application for viewing the model in real-time was developed by Matthew Campbell, presently an undergraduate student in the Computer Science department at Virginia Tech. The application is based on international open standard programming languages and API's. It allows the user to interactively navigate through the model, enabling a real-time interactive photo-realistic simulation of the virtual environment.

For purposes of the real-time visualization, the following steps are involved in the creation of the 3D rendering application:

- The first phase involved in the real-time rendering process is that of mesh optimization. Currently, only simple methods are utilized as the model itself contains around 70,000 faces. Basically, the model is converted from a 3ds max^{TM} (.3DS) file to a basic format, which contains lists of vertices and triangles sorted by material. The model is then saved to a file. The materials are written to a text file, namely a shader file. The shader file contains material (shader) definitions which specify rendering parameters such as how many texture layers, whether alpha blending is on or off, culling (front, back, disabled), etc.

- The next phase is that of the actual rendering. The shader file discussed above is loaded and parsed. Materials are created based on the information in the shader script. The model is rendered utilizing basic hardware multitexturing to achieve some simple yet nice looking effects. Environment mapping is performed on the water surfaces and also for the glass planes. This affect is achieved through the use of a cubic environment map. A cubic enviroment map is constructed using six textures, which are mapped onto the six sides of a cube. The cubic environment map, otherwise known as a skybox was generated using Terragen. A scene consisting of a lake, mountains, and sky was generated and six renderings were taken using a 90-degree field of view with the camera rotated in alignment with each cube face. The final step needed to complete this effect is to setup the hardware texture units so the diffuse texture, is on the first unit, which is then blended with the environment map located on the second texture unit. The glass is environment mapped as well to achieve reflectance, but is alpha blended with the rest of the scene.

- Due to the high complexity of polygonal tree models, we chose to construct trees with intersecting planes. The effect is achieved by using TGA files containing an alpha channel, which is then mapped onto the tree planes. Alpha testing is enabled allowing areas with a zero in the alpha channel to be ‘see through’.

5.3.1 Features of the Real-Time engine:

- It is a simple object oriented C++ interface to simplify 3D rendering.
- Extensible framework (via C++ subclassing, for example), which eases application programming and makes adding hard-to-generalize capabilities a straightforward process.
- OpenGl® support with window management via Simple DirectMedia Layer (SDL). This makes porting to Windows® or Mac®s very easy.
- Hardware transform and lighting

- Common virtual file system, which allows one to load multiple resource directories or archives and reference files without worrying about long path names and the like.
- Model management- import 3D Studio (.3DS) files.
- World management- spatial culling via octrees.
- Hierarchical scene graph, minimized render state changes, derived positions orientations and hierarchical culling. The scene graph is optimized for real-time applications. It is a rooted, directed, acyclic graph consisting of objects that represent the drawable parts of an application's 3D world. The leaves of the tree represent geometric objects, while the internal nodes serve to group and manage parts of the scene graph. Multiple types of geometric primitives are supported, including lines, particles, indexed triangle sets, and indexed triangle strips. Geometric data sharing (a form of instancing) is also supported to reduce memory requirements. For example, geometry for each of the four wheels of a car can be shared to avoid three extra copies of the vertex information.
- Culling: automatically culls any geometry that will not appear on the screen using standard view frustum culling, leaving more processor cycles for drawing visible objects.
- Sorting: A sorting framework is also included. This framework is designed to attain the best possible performance and create special rendering effects.

  Multiple sorting methods may be used simultaneously in different parts of the same scene graph. Such an approach enables applications to sort parts of a scene graph using one method (say, back-to-front), while another is used on another part of the scene graph (such as by "priority" or by texture). Right now our engine sorts faces ‘by material’.
- Textures and Transparency a wide range of texture effects are supported including

  -Multi-textures, such as lightmaps, dark maps, gloss maps, decals, etc.

  -Animated textures, for effects such as fog, fire, smoke, and explosions.

  -Environment maps

  -Textures may be blended with vertex colors in a variety of modes: replacing the base color of the polygon, blending with the color of the polygon, or modulating the vertex colors and/or the alpha channel.

-Supports transparency, including alpha-blended translucency, which can be used to create cutout billboards, stained-glass windows and other effects.

- Texture management, with loading via SDL image. Support for BMP, PNM, XPM, LBM, PCX, GIF, JPEG, PNG, TGA file formats.
- Procedural texture coordinate generation for effects such as environment mapping, or texture matrix ops. (rotation, scaling, scrolls)
- Scripted shader file format. Uses text files to represent complex shaders allowing one to specify abstract blending effects without worrying about the underlying hardware.
- Hardware multitexturing
- Lightmapping (currently, only omni lights supported)
- Easy to use camera
- Optimised render state manangement
- Skyboxes

CHAPTER 6: RESULTS

The study has laid out a methodology for 3D photo-realistic construction and real-time simulation of virtual environments, using widely available standard tools. The outcome of the overall methodology is from a non-interactive 3D model of the virtual environment to a final real-time interactive photo-realistic simulation in our custom made 3D walk-through application.

The 3D walk-through application, which is based on the standard tools and API's described before, exhibits the whole VR construction with great visual impact under the Linux® operating system running on a PC. The base library used for all windowing events, mouse, keyboard input is SDL (Simple DirectMedia Layer). This is multiplatform and makes porting easy to other platforms.

The computer used for modeling and rendering was a PC running Windows 2000^{TM}, with system specifications as follows: Intel® Xeon^{TM}, 1.5 GHz CPU and 512 MB RAM. All modeling and rendering was done in AutoCAD Architectural Desktop^{TM} 3.0, Autodesk VIZ® 4.0 and Lightscape 3.2. Image touch up, texture modulation and other image manipulations were done in Adobe Photoshop® version 6.0.

The initial detailed model had 99,700 polygons, consisting of the building, interior furniture and detailing, site, landscape and trees. This was optimized for real-time using several optimization techniques while modeling, bringing the polygon count down to 65,000.

A total of 73 texture maps have been used, including bump maps, opacity maps, and environment maps, utilizing 8.5 MB of texture memory.

The 3D model performed adequately in real-time. Various tests were performed with different graphics cards and PC configurations. At present the demo is very simple with just geometry and diffuse textures and in some cases using alpha mapping. In addition effects like cubic environment mapping and reflections have also been attempted.
This kind of real-time rendering is dependent on the graphics card and the speed of the AGP bus, which determines the speed of the transfer of model vertices (stored in the

system memory) to the graphics card memory. Effects such as collision detection, physics, dynamic lightmapping are processor dependent.

Also, at present no optimization algorithms are performed. All polygons get rendered for each frame. Spatial culling is yet to be incorporated. The only optimization that takes place is sorting of faces 'by material' which can be further optimized by using some hierarchy. Backface culling (OpenGL® initiated) is taking palce. It is anticipated that the frame rate would increase by 80 percent once optimization takes place.

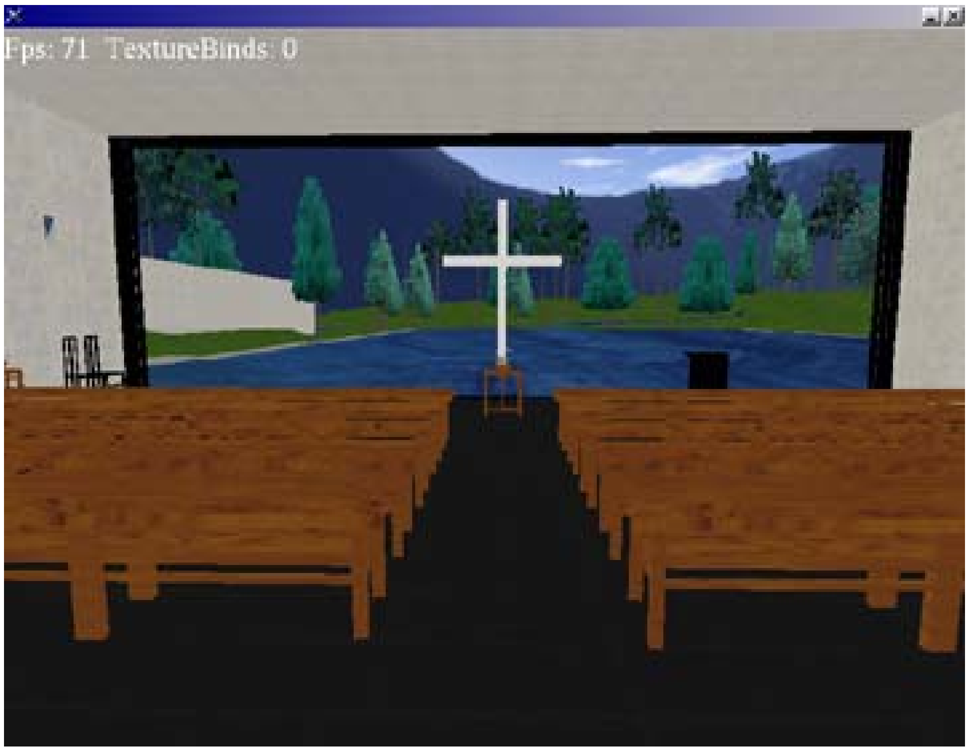
Frame Rate achieved	**System Configuration**
50-75 frames per second	Intel® Pentium® 4 Processor, 1.5 GHz, 256 MB hard RAM, NVIDIA® GeForce^{TM} 2 ultra Graphics card, AGP-4x
30-45 frames per second	Intel® Celeron® II, 550 MHz, 320 MB SDRAM, NVIDIA® GeForce^{TM} 2 Ultra Graphics card, AGP-2x

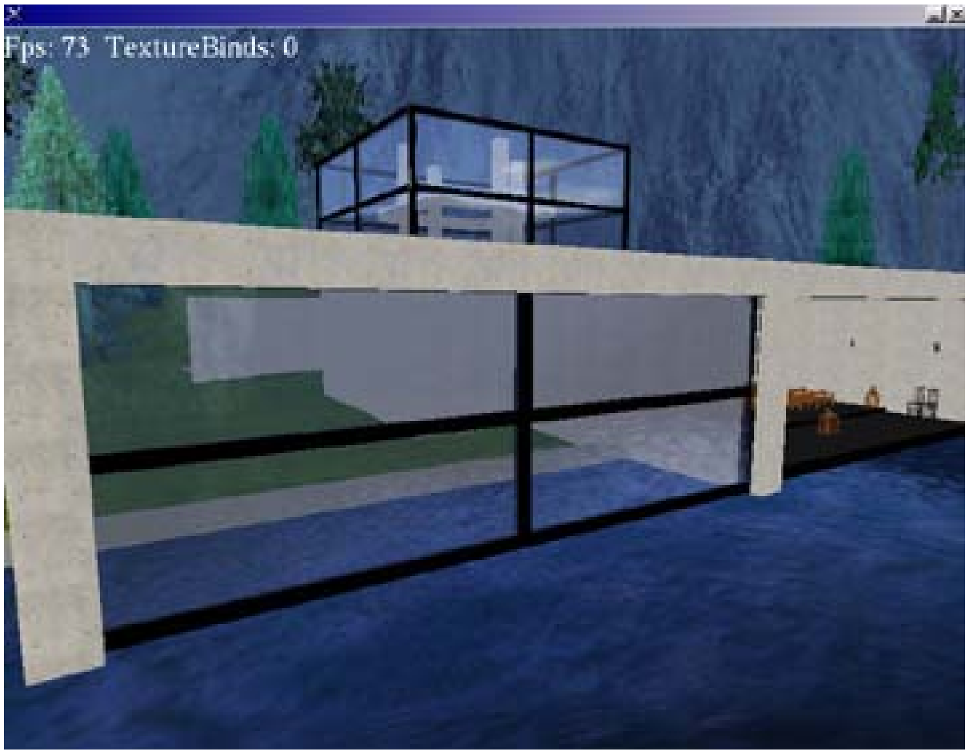
Table 6-1 Frame rates achieved during real-time simulation of the final model

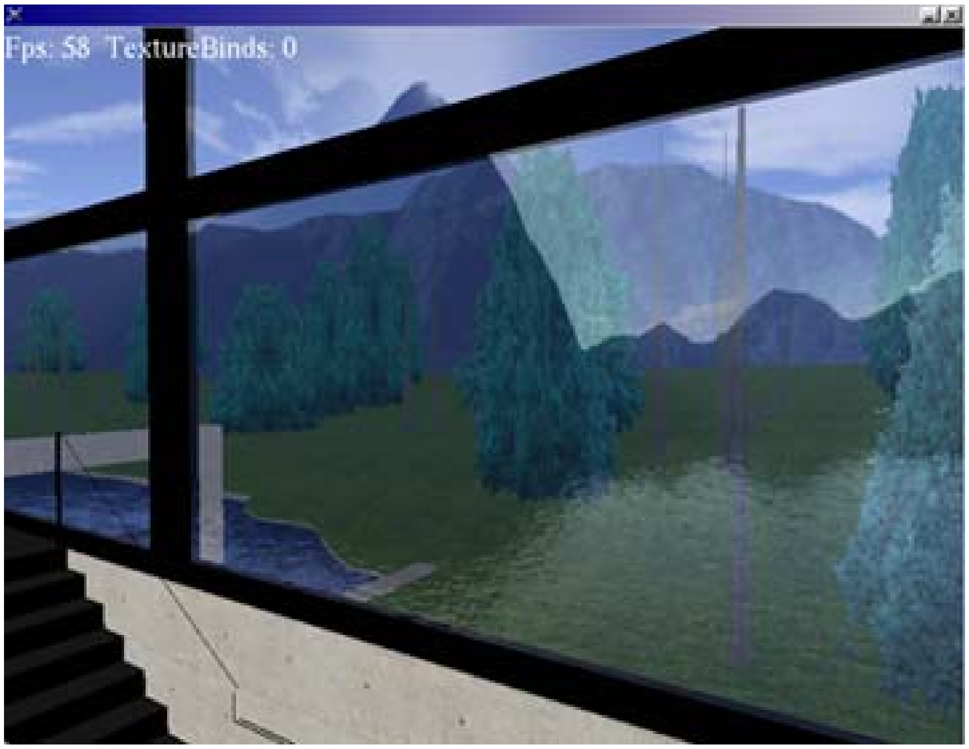
Automatic generation of lightmaps was attempted, but still needs to be highly optimized. At present, 16x16 pixel size lightmaps are generated for each face. This process is very inefficient, because during rendering the engine has to bind a lightmap texture for every single face, which can be very time consuming. Lightmaps for colored light take three bytes per pixel, one byte each for RGB component as compared to lightmaps for white light, which take up one byte per pixel. This saves on texture memory usage too. Going by the present performance, each lightmap would take 16x16x3 bytes, that is 768 bytes and an overall 53 MB texture memory usage for just lightmaps for all the faces in the scene.

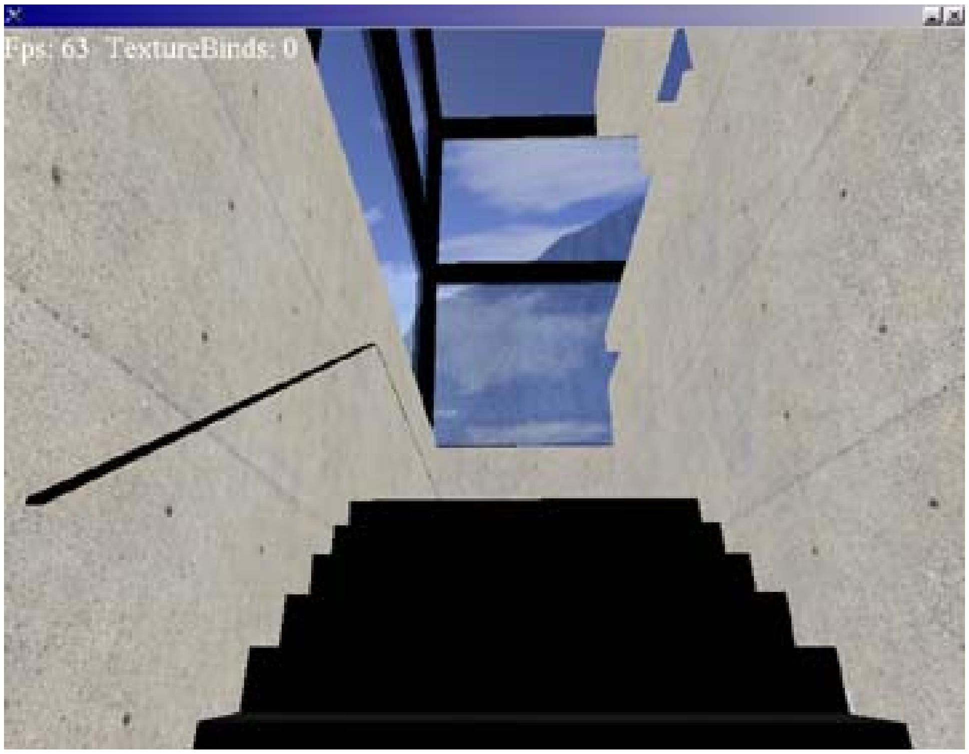
This has to be optimized, whereby, single lightmaps can be packed into one big texture. What would be done basically is to group faces together based on some specific criteria, and pack all the separate lightmaps into a bigger texture for these group of faces. The lightmaps should also be of multiple sizes, based on the area and dimensions of the actual face. Some blurring operation or filtering techniques are also required to remove the jagged edges and antialias the lightmaps.

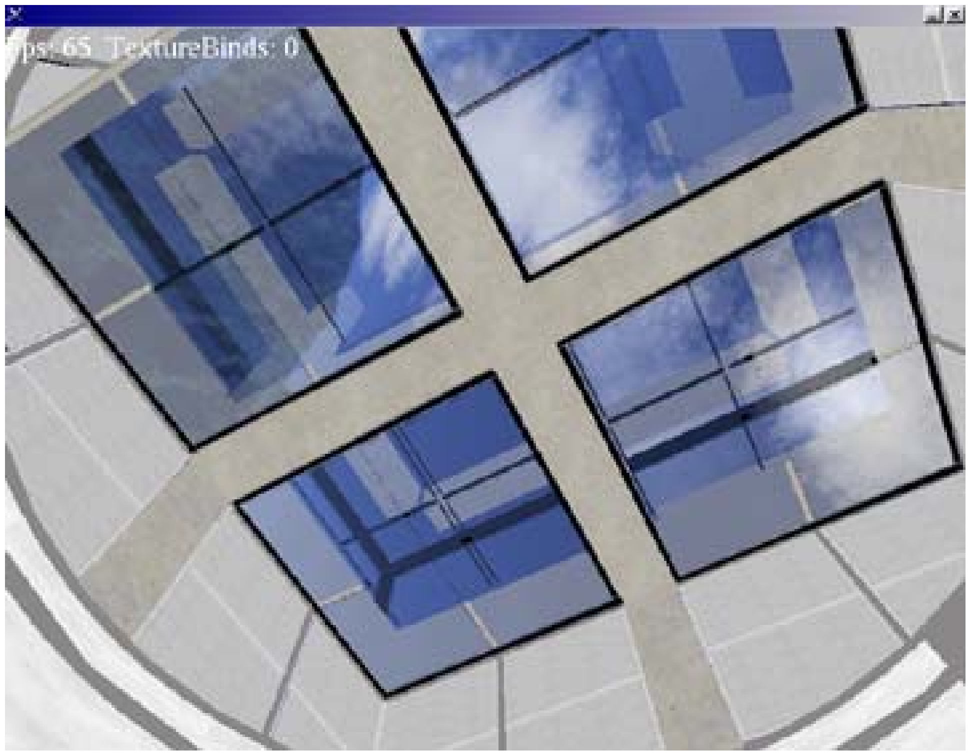
Screen captures in our Real-Time Application






Fps: 58 TextureBinds: 0


Fps: 63 TextureBinds: 0



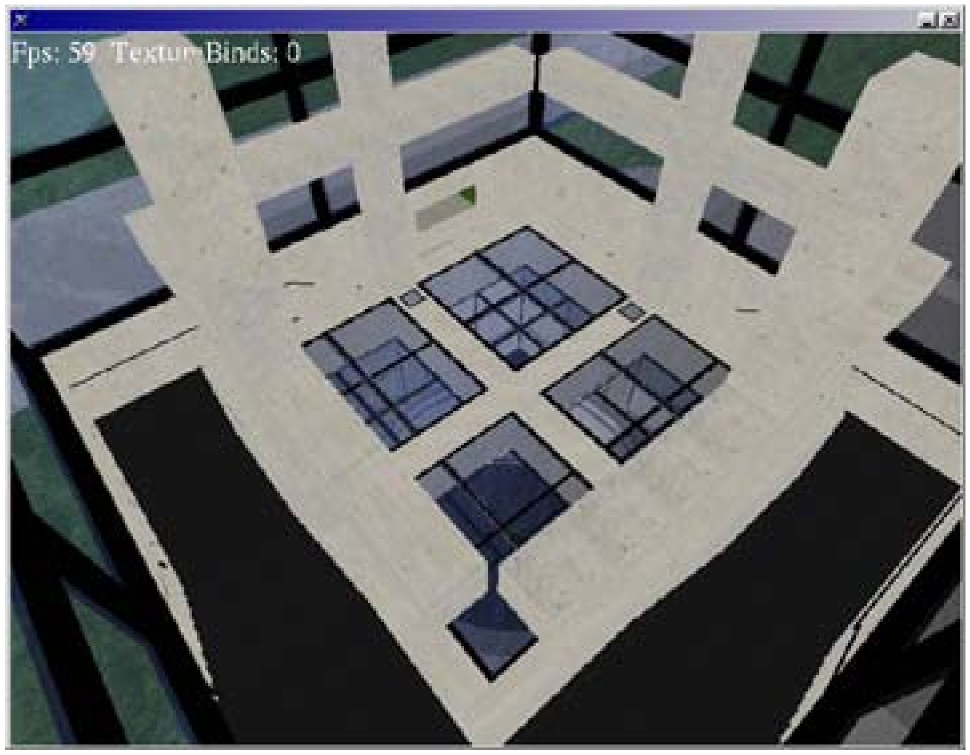


Fig 6-1 Screen captures using the Real-time Application

CHAPTER 7: CONCLUSIONS AND FUTURE WORK

7.1 Conclusions

This study delves into the issues involved in real-time rendering of virtual environments developing a simple, clean and open methodology for the creation of interactive, photo-realistic virtual environments. This methodology is based on standard widely available tools and API's giving more people an opportunity to create photo-realistic VR simulations without the need to rely on licensed commercial software such as proprietary multi-thousand dollar 3D game engines. The technical choices for this project were made keeping in mind that the system remains open, and can be easily adapted or extended to fit particular needs.

When building a model for real-time rendering, controlling polygon count and how surfaces are formed and intersect is important for achieving efficient results. When creating geometry in a CAD or modeling package there are several guidelines that one should follow. A little planning can go a long way to avoid unnecessary problems in the later stages. The guidelines for efficient modeling as discussed earlier in section 5.2.2.1 are summarized below.

Models

- Common Primitives
  A few common primitives and their combinations can be used to create most architectural virtual environments: extruded lines and polylines and 3D faces. The data generated by any chosen modeling package needs to be converted into a format, which can be simulated in real-time. Every software package has its own file format and each one interprets other file formats in its own way. Till a standardization of file formats is achieved, there is little predicting which primitives are supported by other software packages, and specifically how certain primitives will be translated. For example, the inherent smallest face unit in 3D Studio is a triangle. For some other softwares, like Lightwave, it is a quadrilateral. Any data exported from 3D Studio gets converted into triangulated polygons.

This is a critical factor, which affects the number of faces that will be generated while converting between different softwares, and needs to be kept in mind.

- Layers
  It is a good approach to structure CAD data into layers while constructing virtual environments. Typically, one needs to have a different layer in the CAD file for each different material in the virtual environment. The reason for creating a layer for each material is that the layers (in AutoCAD®) are usually converted into distinct objects (in 3D Studio). One *may* even need to have a distinct layer for each different object of the same material.

- Tessellation
  3D CAD data is broken down into its most basic elements, polygons, at the time of rendering. These polygons can be four-sided, but most often they are broken down into three-sided polygons (depending upon the modeling software used) in a process called tessellation. Although tessellation process cannot be controlled, one can manage and create data in a way that it is tessellated in a desirable manner.

- Polygon Management
  One of the fundamental factors for a virtual environment to be simulated in real-time is number of polygons it contains. Basically, the more polygons you have in your virtual environment, the more slowly it will render in "real-time." There are several strategies available for building efficient virtual environments, like billboarding and level of detail techniques. Many times one can reduce the polygon count simply by modeling efficiently and removing unnecessary polygons in the virtual environment. Some of the points to be kept in mind while modeling are, using planes vs. boxes, removing coplanar and intersecting faces and modeling curves with fewer faces. The thumb rule to remember is-‘ Do not draw faces that will not be seen’.

Images

- Types and Bit depths

  There are many types of digital images, which can be created and used, in architectural renderings and only a few of them are appropriate to the construction of virtual environments. GIF and JPEG formats work well for standard textures. TGA, PNG, TIFF file formats work for textures wherever an alpha channel is required. RGB, Silicon Graphics Red-Green-Blue image is a standard on SGI® machines. Also, note that there are RGBA files, which is an RGB file format with a fourth "alpha" channel. Often, for real-time rendering, GIF files or compressed file formats such as JPEG are used because of their small size, enabling quicker rendering.

- Sizes

  Appropriate file sizes for real-time rendering are those whose dimensions are powers of two. That is: 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024 pixels and so on. Use square files for nearly square images, and use rectangular (1:2, 1:4, etc) sizes for elongated images. Often the size you specify depends on the kind of information you are trying to show, how closely it is meant to be viewed, how large the image will be in the environment, and whether it is a tileable pattern, like grass or concrete, versus a specific image like a building elevation or a horizon. Keeping in mind the texture memory limit of available graphics cards, texture sizes should be kept as small as possible. Another option is to use the same texture for multiple surfaces that have similar appearance.

- Alpha Maps

  To reduce the amount of geometry needed to represent complex objects "alpha mapping" should be used. Basically, this is a way to add information to an image to make portions of it transparent. With this technique, we can represent a complex object, like a tree, as a flat plane with a masked-out texture map of a tree applied to it. One needs to create an *opacity map* and an RGBA image to be used for alpha mapping in real-time rendering.

Surfaces

- Normals
  The normal of a polygon has several uses in rendering, but of greatest importance to us is that by default it can increase display and rendering performance by hiding faces whose normals point away from the participant in the virtual environment. The direction of the normals in the model should be controlled to avoid having "holes" in the geometry. Converting 3D data between different modeling softwares can result in elements, which have their face normals flipped the wrong way. This can be detected in 3D Studio by turning off backface cull in the display panel or by rendering the objects. The Normal modifier can be used to correct this.

- Smoothing
  Objects with polygon faces that share edges can appear faceted or smooth, depending on the characteristics specified when representing curved surfaces in virtual environments. With smooth surfaces, a curved object can typically be modeled with much fewer polygons to achieve a similar effect as that of a faceted object with a high polygon count.

- Optimization
  Modeling softwares usually have options for simplifying geometry in order to speed up rendering while maintaining an acceptable image. 3D Studio has modifiers like Optimize and Multires, which reduce the memory overhead needed to render models by decreasing the number of vertices and polygons. Careful modeling can further improve the results of these algorithms.

Environment

- Lights
  The number of lights used for real-time rendering should be limited. The more lights added to an environment, the slower that world will render because the effects of each light source must be added to the next.

Besides the efficient modeling techniques discussed above, maintaining real-time frame rates for providing an interactive experience when viewing rendered scenes also depends upon efficient implementation of the graphics pipeline.

Organizing the scene data into structured scene graphs can help in executing different filtering and optimization algorithms systematically. This reduces the number of polygons being drawn by discarding those polygons that are not visible. A node in a scene graph is not visible if it isn't within the view frustum, or if it is obscured by other elements of the scene being rendered. Culling algorithms test an input node to determine if it should be rendered or not. There are two kinds of culling algorithms: *frustum culling* algorithms, which test a node to determine if its within the viewing frustum or not, and *occlusion culling* algorithms, which test nodes to determine if they are visible (not obscured by other elements of the scene being rendered).

The spatial data structure approach enables combining occlusion culling with frustum culling, so that another traversal of the scene database is not necessary. If the scene is highly dynamic, the cost of updating the data structures swamps the savings provided through their use, so the bounding-volume approach is better. Most scene graph based applications implement some form of frustum culling: Direct3D® (Microsoft, 2000), IRIS Performer^{TM} (Rohlf, 1994), and OpenInventor^{TM} (Wernecke, 1994) have frustum culling enabled by default. Frustum culling can be used in conjunction with other culling algorithms; for example frustum culling algorithm is performed before portal culling is done, and occlusion culling is usually performed after frustum culling has been done. This process is more efficient, since most of the polygons lying outside the view frustum have already been discarded during the first test. The subsequent stages have lesser polygons to test saving computation power and rendering time. Using this method, many polygons can be culled away with only a few overlap tests.

Besides keeping the polygon count low and optimizing the model for real-time simulation, an interesting situation that arises from rendering hundreds of textures is that the main area to optimize now is texture memory usage rather than polygon count. The standard 64 MB just isn't enough for high resolution textures (especially in a large environment). Rendering lighting as textures is also pushing the movement from SGI®s to Linux® PCs, since PC graphics cards will do multi-texturing (and SGI® hardware will not). This allows us to combine a tiled texture map with a non-tiled lightmap on a surface (similar to current games such as Quake^{TM} 3).

Rendering lighting as textures is an efficient technique for simulating illumination in virtual environments. Rendering the scene lighting (including radiosity) as textures enables more realistic lighting at run-time and is more memory efficient. Radiosity solution discretizes the surfaces generating a 3D mesh that requires more memory than the original surfaces. Rather than determining the color for each pixel on a screen, radiosity calculates the intensity for all surfaces in the environment. This is accomplished by first dividing the original surfaces into a mesh of smaller surfaces known as elements. Extracting the radiosity view independent lighting as lightmaps has significant advantages, since the mesh is converted into texture maps that preserve the lighting information, and the mesh itself can be deleted. Using lightmaps virtual real-time walk-throughs and animations can be created easily, without having to recalculate radiosity for each frame.

Zhukov et al (1998) had proposed an automatic methodology for sampling, creating and storing lightmaps, targeted for environments of small to average geometric complexity. The generation and mapping of lightmaps as described using Lightscape is certainly more time consuming, as compared to the automatic generation of lightmaps. But, the inclusion of standard 3D modeling packages in the methodology pipeline enables designers, architects and modelers to have utmost control over the lightmap generation, sampling and application as well as the scientific accuracy required for constructing their virtual environments.

Through our real-time 3D application, a method for automatic generation and storing of lightmaps is proposed. Manual generation of lightmaps using the ‘mesh to texture’ tools in Lightscape is a tedious process. The option of ‘Convert each texture to a single texture per surface’ is the most automatic, but provides little control over the number of lightmaps that would be generated, eventually overloading the texture memory. The second option, ‘Project all selected geometry into one texture, is more time consuming but gives more control over generation of lightmaps and keeping their number limited. One must find a balance between the geometric complexity and lighting quality, and selecting which option works best is a judgment call depending upon the model.

The proposed ’automation’ process for generation of lightmaps greatly speeds up the production phase. However, it still needs to be optimized, by packing smaller patches into single larger textures to increase real-time performance.

"Static lighting," using lightmaps, places all of the lighting information for the world into textures, storing lighting on a ‘per texel level’. These textures are then used to modulate the diffuse color of surfaces. The advantage of this type of lighting is that very complex lighting effects can be achieved that require days of processing before the scene can be viewed in the engine. Unfortunately, it is too expensive to modify the lightmaps at runtime, so if dynamic lighting is needed, it must be added to the system using per-vertex processing.

Following the proposed methodology, extremely realistic lighting of unlimited complexity can be created for virtual environments. In practice, however, there are two factors that ultimately limit what one can do: the real-time polygonal display rates and texture memory of the system. For a successful real-time set, the ideal display rate for the model is 60 frames per second. This means that one must manage the polygonal complexity and keep the texturing within the limits of available texture memory.

Today, with technological advances in graphics hardware, 3D accelerators can transform and light triangles, perform complicated operations of blending several textures and execute shaders - short programs operating with vertices and pixels. With the advent of vertex and pixel shaders, which allow the graphics accelerator itself to be programmed, more accurate modeling of the interaction between light and materials is now possible.

“The final output of any 3D graphics hardware consists of pixels. Depending on the resolution, in excess of 2 million pixels may need to be rendered, lit, shaded and colored. Prior to DirectX® 8.0, Direct3D® used a fixed-function multitexture cascade for pixel processing. The effects possible with this approach were very limited on the implementation of the graphics card device driver and the specific underlying hardware. A programmer was restricted on the graphic algorithms implemented by these.

With the introduction of shaders in DirectX 8.0 and the improvements of pixel shaders in DirectX 8.1, a whole new programming universe can be explored by game/demo code developers. “Pixel shaders are small programs that are executed on individual pixels. This gives an unprecedented level of hardware control for their users.” (Engel, 2002)

The shaders used in DirectX®/OpenGL® (vertex and pixel shaders) are purely aimed at real-time rendering. The new (as of March, 2002) high-end graphics cards like GeForce™ 3/4TI and RADEON™ 8500-based cards support pixel shaders. Vertex and pixel shaders allow visual effects that are directed towards enhanced photo-realism. Many types of effects are possible, like single pass, per-pixel lighting, true phong shading, anisotropic lighting, volumetric effects, advanced bump mapping, procedural textures and fur/hair rendering to name a few.

Shaders in 3D-accelerators, is a new and relatively complicated technology. The major problem today is to achieve compatibility. “If the DirectX® 8.0 interface could be called unified, implementations of OpenGL® extensions by ATI (ATI_vertex_object) and NVIDIA® (NV_vertex_program) are, generally speaking, incompatible” (Martynenko, 2001). The manufacturers of 3D-accelerators should work out a unified standard within the framework of the future versions of OpenGL®.

The future holds endless opportunities and great scope in the field of real-time photo-realistic rendering of virtual environments. The quest for a sense of presence in virtual environments demands increased visual realism from real-time computer graphics. The accelerating progression of ever-higher computational and rendering speeds and ever-lower costs will, in the not too distant future, bring real-time rendering effects within reach of an average PC user.

7.2 Future Work

Autodesk VIZ® 4.0 was initially selected for the project since it has a completely reformulated radiosity engine. The new engine does not use the 'progressive refinement' approach (as does Lightscape), but is based on new statistical sampling techniques that produce fully converged radiosity solutions in a fraction of time Lightscape requires. It also has pixel regathering functionality that eliminates most of the artifacts that one gets with Lightscape (floating objects, light leaks, and shadow leaks). It also supports a wide array of rendering effects that are not possible in Lightscape, such as, bump mapping, opacity mapping and volumetric lights.

With all these plus points, Autodesk VIZ® 4.0 was but an obvious choice over Lightscape for processing the radiosity solution for the model. But there is no way, yet, in VIZ to generate lightmaps once radiosity has been processed. So Lightscape had to be used for that purpose. It would be a great advantage if a tool could be incorporated in VIZ for generating lightmaps. This would save a lot of production time, and simplify the workflow, avoiding the path from Autodesk VIZ® to Lightscape and back to Autodesk VIZ®.

Our custom 3D application creates and stores lightmaps, but currently only omni lights are supported. This would have to be worked upon to support other lights such as target and directional lights, spot lights, point lights and photometric lights.

The lightmap generation process needs to be optimized. Currently a single lightmap is generated for every surface in the scene. This process has to be optimized to pack all the lightmaps along one plane into a single larger texture to increase real-time performance.

Spatial structures need to be built in order to optimize the model for real-time performance, even more so once the size of the model increases. Basic scene graph support needs to be supported as well, to maxmize efficiency not only through view frustum culling but also to organize the render state changes.

REFERENCES

[Brooks86] F.Brooks. Walk-through: A Dynamic graphics system for simulating virtual buildings. In ACM Symposium on Interactive 3D graphics, Chapel Hill, NC, 1986.

[Aliaga99] D.Aliaga, J.Cohen, A.Wilson, W.Stuerzlinger. A Framework for the Real-Time Walk-through of Massive Models, UNC, Chapel Hill, 1999.

[Bastos99] Rui Bastos, Kenneth Hoff, Anselmo Lastra. Increased Photo-realism for Interactive Architectural Walk-throughs, UNC, chapel Hill, 1999.

[Lischinski98] D. Lischinski and A. Rappoport. Image-based rendering for non-diffuse synthetic scenes. *Rendering Techniques '99*, 1998, pp.301-314.

[N.Greene86] N.Greene. Environment mapping and other applications of world projections, In *IEEE CG&A*, 1986, pp.21-29.

[Abrash97] Michael Abrash, Graphics programming Black Book Special edition, ISBN 1-557610-174-6, Coriolis group, 1997, pp. 1245-1256.

[Zhukov98] S.Zhukov, A.Iones, G.Kronin, Using Lightmaps to create realistic lighting in real-time applications, Proc. Of WSCG'98- Central European Conference on Computer Graphics and Visualization, 1998, pp. 464-471.

[DeLeon00] V.DeLeon, R.Berry Jr, Bringing VR to the Desktop: Are you game?, IEEE Multimedia, April-June 2000, pp.68-72.

[Maciel95] Paulo W.C Maciel and peter Shirley. Visual navigation of Large environments using Textured Clusters. In 1995 Symposium on Interactive 3D Graphics, ACM SIGGRAPH, 1995, pp. 95-102.

[Hoff97] Kenneth E. Hoff III. Faster 3D game Graphics by not drawing what is not see, ACM Crossroads, 1997.

[Funkhou96] T. Funkhouser, S. Teller, C. Sequin and D. Khorrambadi. The UC Berkeley System for Interactive Visualization of Large Architectural Models. In *Presence,* volume 5 number 1.

[Rohlf 94] J. Rohlf and J. Helman. Iris Performer: A high performance multiprocessor toolkit for real-time 3D graphics. In *Proc. of ACM Siggraph, 1994*, pp. 381-394.

[Jepson95] W. Jepson, R. Ligget and S. Friedman. An Environment for Real-Time Urban Simulation. In *Proceedings of 1995 Symposium on Interactive 3D Graphics*, 1995, pp. 165-166.

[Aliaga98] D. Aliaga, Anselmo Lastra. Smooth Transitions in Texture-based Simplification. In *Computer and Graphics, Elsevier Science, Vol 22:1*, pp. 71-81, 1998.

[Teller91] S. Teller and C.H. sequin. Visibility Preprocessing for Interactive walk-throughs. In *Proc. of ACM Siggraph*, 1991, pp. 61-69.

[Luebke95] D. Luebke and C. Georges. Portals and Mirrors: Simple, Fast Evaluation of Potentially visible sets. In *ACM Interactive 3D Graphics Conference*, Monterey, CA, 1995.

[Aliaga97] D. Aliaga and A. Lastra. Architectural walk-throughs using portal textures. In *IEEE Visualization '97*, 1997, pp. 29-38.

[Rafferty98] M. Rafferty, D. Aliaga, A. Lastra. 3D image warping in Architectural Walk-throughs, UNC TR#97-019, submitted for publication, 1997.

[McMillan94] L. McMillan, G.Bishop, H.Fuchs, E.J. Scher Zagier. Frameless Rendering: Double buffering considered harmful. In *Computer Graphics (SIGGRAPH '94 Proceedings*), 1994, pp. 175-176

[Cohen93] M. Cohen, J. Wallace. Radiosity and Realistic Image Synthesis. Academic press, Boston, 1993.

[Diefenbach96] P. Diefenbach. Pipeline rendering: Interaction and Realism through hardware- based Multi-Pass rendering. University of Pennsylvania, department of Computer Science, Ph.D. dissertation, 1996.

[Pabst00] Thomas Pabst. Tom's Take on NVIDIA®'s new GeForce^{TM} 2 GTS. In *Tom's Hardware Guide, Category: Graphics,* 2000.

[Watt92] Alan Watt and Fabio Policarpo, 3D Games Real-time rendering and software Technology. ISBN: 0201-61921-0. Addison Wessley, New York, 2001, pp. 315, 346-348.

[Moller71] T. Moller and E. Haines. Real-Time Rendering. ISBN 1-56881-101-2, A K Peters, 1999.

[S.Fischer, 1990] S. Fisher. Virtual Interface Environments, The Art of Human-Computer Interface Design. B. Laurel (ed.), Addison-Wesley, California, 1990.

[Miliano99] Vito Miliano and Perilith Industrielle. Unreality: Application of a 3D game Engine to Enhance the Design, Visualization and Presentation of Commercial Real Estate, 1999.

[Watt et al, 2001] Alan Watt, Fabio Policarpo. 3D Games Real-time Rendering and Software Technology, Addison-Wesley, 2001.

[Engel, 2002] Wolfgang Engel, Shader Programming, Part III: Fundamentals of Pixel Shaders. In *GameDev.net,* 2002.

[Martynenko, 2001] Konstantin Martynenko, Introduction to Shaders. In *Reactor Critical, review-shadersintro,* October 2001.

[Micr, 2000] Microsoft Corp., *Microsoft DirectX 8.0 Directx Graphics*, October 2000.

[Wernecke, 1994] J. Wernecke, The Inventor Mentor, Addison-Wesley, Reading, Massachusetts, 1994.

[Kelso et al, 2001] John Kelso, Lance E. Arsenault, Steven G. Satterfield, Ronald D. Kriz. DIVERSE: A Framework for Building Extensible and Reconfigurable Device Independent Virtual Environments, December 2001.

[N. Greene et al, 1993] Ned Greene, M. Kass, G. Miller. Hierarchical Z-Buffer Visibility, *Computer Graphics, SIGGRAPH 1993 Proceedings*, pp. 231-238, August 1993.

[Eftekhar, 2001] Judy Lin-Eftekhar. Virtual Temple Mount, Digital tools bring archeological site back to life, *UCLA Today,* 2001.

OTHER REFERENCES

Urban Simulation Team, UCLA
http://www.ust.ucla.edu/ustweb/ust.html

VRND: Notre Dame Cathedral, A Real-Time Virtual Reconstruction, Digitalo Studios.
http://www.vrndproject.com

[Deleon98] V. DeLeon and R. Berry. Virtual Florida Everglades, UNESCO Virtual World Heritage Project, In *Virtual Systems and Multimedia VSMM98*, 1998.
http://www.digitalo.com/deleon/vrglades

EON Studio, Virtual Reality Toolkit.
http://www.eonreality.com

Sense8 World Toolkit
http://www.sense8.com

Multigen Paradigm Runtime Products
http://www.multigen.com

Church on the Water, Church of the Light, Tadao Ando. Philip Drew, Architecture in Detail, 1996 Phaidon Press Limited.

The Colours of Light, Tadao Ando Architecture. Richard Pare, 1996 Phaidon Press Limited.

Lightscape 3.2 User's Guide, Overview Technology, ISBN 60903-010000-5022.

GLOSSARY OF TERMS

AGP Bus (Accelerated Graphics Port)

The Accelerated Graphics Port (AGP) interface is a new platform bus specification that enables high performance graphics capabilities, especially 3 dimensional, on PCs. It is a dedicated bus from the graphics subsystem to the core-logic chipset. This new bus shifts the memory requirements for the 3D portions of a graphics subsytem from the local frame buffer memory to main system memory.

Alpha Blending

In computer graphics, each pixel has three channels of color information--red, green, and blue--and sometimes a fourth called the alpha channel. This channel controls the way in which other graphics information is displayed, such as levels of transparency or opacity. Alpha blending is the name for this type of control, and it's used to simulate effects such as placing a piece of glass in front of an object so that the object is completely visible behind the glass, unviewable, or something in between.

Alpha Channel

Digital images typically store their information in four channels- red (R), green (G), blue (B), and the alpha, or matte channel. The alpha channel is a grayscale image that describes the opacity of the corresponding color channels with black being completely transparent and white being completely opaque. When compositing the foreground image over another image, it is the gray values of the alpha channel that determine what percentage of the color channels appear over the background image.

Anti-aliasing

A rendering technique used to make jagged edges, referred to as jaggies or stairstepping, appear smoother by inserting pixels of an intermediate color between adjacent pixels with abrupt edges.

API (Application Program Interface)

An API is a series of functions that programs can use to make the operating system do their dirty work. Using Windows APIs, for example, a program can open windows, files, and message boxes--as well as perform more complicated tasks--by passing a single instruction. Windows has several classes of APIs that deal with telephony, messaging, and other issues.

Bandwidth

Bandwidth refers to the total amount of data that can be transmitted through a network over a given amount of time. All transmitted signals, either analog or digital, have a defined bandwidth. For digital, bandwidth data speed is most commonly measured in bits per second (BPS), whereas analog systems tend to measure bandwidth as the difference between the highest- and lowest- frequency signal components. The greater the bandwidth, the greater the capacity.

Bit

Abbreviation for binary digit, a bit is the smallest unit of information a computer can recognize. One mathematical bit is defined by only two levels, or states, of information that can be obtained by asking a yes or no question. For example, the answer to the question can be only one of two values, such as true or false, 0 or 1, black or white. Two bits can define four levels of information, three bits can define eight, and so on. Eight bits are equal to one byte.

Bit depth

Each pixel component in a digital image is represented by a number of bits, and the number of bits per channel is known as the bit depth of that image. A commonly used bit depth is 8 bits per channel, which is also referred to as a 24-bit image (8 bits x 3 channels=24 bits). If you have 8 bits per channel (also called 8 bits per component), it means that each channel can have 2^8 or 256 different possible color values from 0 to 255. Also referred to as color depth.

Bitmap

A 3D array of bits used to represent a 2D array of pixels that make up a digital image. A bitmap is defined by the number of colors or shades of gray it can represent, and its size is determined by the width and height of the pixels in the image. A bitmap representing a color image is called a pixel map. Also called a raster image, bitmapped image, pixel map.

Binary Space Partition Trees (BSP trees)

A method of subdividing a volume in 3D space. The volume is originally represented as a single rectangular bounding box, and if more detail is needed, it is split into two volumes. This process continues until each sub-box contains the desired level of detail (LOD).

Bump mapping

A texture mapping technique in which the intensity of the gray values contained in an image, called a bump map, is used to simulate a bumpy surface on an otherwise flat 3D object. Because a flat surface contains surface normals that point straight up and a bumpy surface contains normals pointing out in different directions, bump mapping is achieved by tricking the renderer into believing the surface is bumpy by artificially tiling or perturbing its normals in many different directions. If a bump map pixel is bright, the surface normal is perturbed in one direction, and if the bump map pixel is dark, the surface normal is perturbed in the opposite direction.

Bus

An internal pathway in the computer that sends digital signals from one part of the system to another. The size of a bus is determined by the number of bits of data it can carry. For example, many microprocessors have 32-bit buses both internally and externally. Also called the main bus.

Byte

A group of 8 bits is equal to one byte of information, which is the amount of computer memory required to represent one character of alphanumeric data. For example, the number ‘0’ is represented by an array of 8 bits, 00000000, and the number 255 is represented as 11111111.

Diffuse

The color of an object when it is hit by direct light. A surface with a small diffuse component will be darker because it reflects less light, whereas a surface with a high diffuse component will be brighter because it reflects more light.

Direct 3D

Microsoft's Direct3D is part of the larger DirectX standard that takes away one of the great burdens of 3D software development: addressing display hardware efficiently. Direct3D is a type of API called a hardware abstraction layer that stands between the application and the hardware that will display it. The developers of the software (usually a game) write instructions to Direct3D, which then translates them to the graphics card. To take advantage of the API, therefore, both the application and the graphics card must support Direct3D.

Direct X

DirectX is a type of API called a hardware abstraction layer that acts for Windows 95 and various types of hardware. The DirectX standard includes Direct3D (which speeds up texture mapping and other 3D graphics processes), DirectSound (for audio cards), DirectDraw (for vector graphics), DirectVideo (for AVI files and other moving pictures), and the DirectPlay and DirectInput team (which simultaneously supports sound, drawing, video, networked gameplay, and joystick standards).

Frame rate

Frame rate refers to the speed at which a series of images are captured or displayed. Frame rate is measured in frames per second (FPS), such as 24 FPS for film and 30 FPS for video.

Global Illumination

A general term used to describe the reflected and transmitted light that originates from every surface in a scene.

Graphics accelerator

A circuit board that is inserted into a computer to allow it to display images on screen. The refresh rate, resolution, and the bit depth of the images that can be displayed on a monitor are determined by the type of graphics board used and is also tied to the limits of the monitor. Also called graphics adaptor, graphics card, graphics board and video card.

Graphical User Interface (GUI)

A computer interface based on graphical elements, such as icons, menus, and windows, that allow the user to interact with the computer.

Local Illumination

A term used to describe the color or light that a surface appears to emit, transmit, and reflect from itself.

MIP mapping

From Latin ‘multi in partem’ meaning “many parts”, this is a texture mapping technique that calculates the resolution of the texture maps applied to a surface based on each object’s proximity to the camera view. The texture maps are successively averaged down to produce antialiased results by using several texture pixels to contribute to the single pixel being rendered.

Object-oriented programming language

A programming language in which the programmer can define the data type of each data structure as well as the types of operations and functions that can be applied to those data structures. With this method, each data structure is treated as an object, and each object is grouped into a hierarchy of classes, with each class inheriting the characteristics of the class above it. This type of programming allows for the creation of new object types that can inherit many of their features from other existing object types.

Octrees

A method of subdividing a volume in 3D space. The volume is originally represented as a single rectangular bounding box, and if more detail is needed, the box is split in half along its X, Y, and Z axis, which results in eight smaller boxes. This process continues until each sub-box contains the desired level of detail (LOD). Octrees are often used to boost efficiency. For example, if a particular algorithm needs to compare a set of particles with its neighboring particles, it can compare each particle to every other particle in the scene. However, if the particles are instead sorted into octree partitions, then each particle need only be compared to every other particle within that particular octree partition. This technique can even save tremendous amount of calculation time. Octrees are the 3D equivalent of quadtrees.

Open Inventor™

Open Inventor™ is an object-oriented 3D toolkit offering a comprehensive solution to interactive graphics programming problems. It presents a programming model based on a 3D scene database that dramatically simplifies graphics programming. It includes a rich set of objects such as cubes, polygons, text, materials, cameras, lights, trackballs, handle boxes, 3D viewers, and editors that speed up your programming time and extend your 3D programming capabilities. Open Inventor is built on top of OpenGL®.

OpenGL Performer™

OpenGL Performer™ is a powerful and comprehensive programming interface for developers creating real-time visual simulation and other performance-oriented 3D graphics applications. It simplifies development of complex applications used for visual simulation, simulation-based design, virtual reality, interactive entertainment, broadcast video, architectural walk-through, and computer-aided design. The latest major release, OpenGL Performer 2.5, is built atop the industry standard OpenGL® graphics library, includes both ANSI C and C++ bindings, and is available for both the IRIX™ operating system and GNU/Linux®. It forms the foundation of a powerful suite of tools and features for creating real-time visual simulation applications on IRIX & Linux systems.

Open Source

A method of software licensing and distribution that encourages users to use and improve the software by allowing anyone to copy and modify the source code.

Pixel

A pixel, from the words “picture element”, is the smallest individual unit, defined as an array of dots, used to describe a digital image. The larger the number of pixels, the greater the resolution of the image.

Primitive

Geometric primitives are a group of basic geometric shapes that are used so frequently in computer graphics (CG) that most 3D software packages include them as pre-defined topology. They include the familiar shapes of a sphere, torus, cylinder, cube, cone, grid, circle, and truncated cone. Geometric primitives are quadratic surfaces that can be defined by a set of quadratic equations.

Reflectance

The ratio of light reflected from a surface to the incident light striking it.

RGB

Abbreviation for the three additive primaries of red (R), green (G), blue (B).

Simple DirectMedia Layer (SDL)

Simple DirectMedia Layer is a cross-platform multimedia library designed to provide fast access to the graphics frame buffer and audio device. It is used by MPEG playback software, emulators, and many popular games, including the award winning Linux port of "Civilization: Call To Power." Simple DirectMedia Layer supports Linux, Win32, BeOS, MacOS, Solaris, IRIX, and FreeBSD. SDL is written in C, but works with C++ natively, and has bindings to several other languages, including Ada, Eiffel, ML, Perl, PHP, Python, and Ruby.

Specular

The highlight caused by a light hitting an object that causes the perception of ashiny surface. The shinier the surface, the more pronounced the specular highlights across that surface. The amount of specularity seen in a surface depends on the viewing angle and the angle of incidence between the surface and the light that is hitting it. The color of the specular reflection is typically the color of the light source in the scene.

Stereoscopic image

An image that contains two slightly different offset images that are placed side by side and are designed for the viewer’s right and left eye as a means of simulating a sense of depth. A stereo image can be viewed with a stereo viewing system or by using a free viewing method.

Texel

Texture coordinates (UV coordinates)

The 2D directional coordinates assigned to each point on a surface that define the placement of texture maps.

Texture map

2D images that are applied to a 3D object to define the texture of a surface. Texture maps can be used for color mapping, bump mapping, displacement mapping, transparency mapping, environment mapping, and projection mapping.

Viewing frustum

The truncated pyramid of vision that is created based on the values for the near and far clipping planes in a 3D scene. Any objects that lie within the viewing frustum will be rendered, whereas those that lie outside will not. Also called the frustum of vision.

Viewpoint

Viewpoint is the location of the viewer or the camera relative to the scene. Also called the viewing location.

What You See Is What You Get (WYSIWYG)

A catchphrase from the old TV show Rowan and Martin's Laugh-In that became a desktop publishing byword, WYSIWYG (pronounced "whizzy-wig") refers to any technology that enables you to see images onscreen exactly as they will appear when printed out. As scalable screen and printer fonts have become more sophisticated, and as graphical user interfaces have improved their display, people have come to expect everything to be WYSIWYG. But it isn't always the case--and certainly wasn't in the 1980s, when this term was first applied.

Z-buffer

The Z-buffer controls which 3D objects are displayed in the view-port in front of other objects based on their distance from the camera view. The Z-buffer gathers and stores the depth information for all objects in a scene and then calculates and writes the proper “foreground” pixels to the display. Also called the *depth buffer*.

VITA

Priya Malhotra

1202, Snyder Lane, Apt #1400F, Blacksburg, VA 24060

http://filebox.vt.edu/users/pmalhotr/resume/res_arch.htm

Email: pmalhotr@vt.edu

Phone: (540) 449 8372

Education

Master of Science, Architecture with focus in Advanced computer and virtual reality applications in Architecture, Virginia Polytechnic Institute and State University.
(Aug '00-July '02)
Bachelor of Architecture, Sushant School of Art and Architecture, Gurgaon India
(Aug '93- Aug '98)

Work Experience

Graduate Teaching Assistant, (Scholarship), Virginia Tech.
(Aug '00– May '02)
Teaching '3D Modeling and Animation', a graduate level course focusing on creating walkthroughs and 3D visualizations of architectural and interior spaces.

Graduate Assistant, (Scholarship), Virginia Tech.
(Aug '00– May '02)
Responsible for training faculty and students on modeling and simulation for virtual environments.

Architect, Jasbir Sawhney & Associates, Architects and Urban Designers, New Delhi, India. (June'99 – June'00)
Worked on large-scale projects spanning institutional, educational, transportation and hospitality fields. Experience with overall project execution- tender documents, municipal approval, cost estimation, design development, construction documents, and interior finishes.

- Projects handled include Tis Hazari Metro Station (part of the MRTS system in Delhi), Hyatt Regency Hotel, Goa; Hyatt Regency Hotel, Calcutta; Sanskriti School, New Delhi.

Architect, Sumit Ghosh and Associates, Architects and Urban Designers.
(Aug'98 – June '99)
Performed tasks related to design, construction drawings and details, municipal approval, site supervision and interaction with clients and consultants for Institutional and Educational projects.

- Designed award winning competition entry for a Housing project in New Delhi.
- Other projects include Hospital and Community Welfare Center, New Delhi; Embassy of Bengali Art and culture, New Delhi; Sterling Resorts, Rishikesh.

Architectural Intern, Hayes, Seay, Mattern & Mattern (HSMM), Architects, Engineers, Planners, Roanoke, Virginia.
(May '01 – Aug '01)
Performed multifaceted roles ranging from design of architectural and interior spaces, site planning, 3D visualizations and immersive simulations for studying environmental impacts of design.

Architectural Intern, Sumit Ghosh and Associates, Architects and Urban Designers, New Delhi, India
(Feb'98 – Aug '98)
Worked on construction documents, design, research and documentation for Sita Ram Bharatiya Institute of Science and Research, New Delhi.

Skills

Operating systems
Windows 95/98/NT/2000, Mac OS, SGI

Modeling and Rendering Packages

3D Studio Max, 3D Studio Viz, AutoCAD, Architectural Desktop, Lightscape, QuickTime VR Authoring Studio.

Graphics programs

Adobe Photoshop, Adobe PageMaker, Adobe Premiere, Media 100, Cleaner

Web based Applications

Macromedia Director & Shockwave studio, Macromedia Flash, Macromedia Dreamweaver.

VR Hardware

Virtual Research HMD, Fakespace Pinch gloves, Cave Automated Virtual Environment (CAVE), Responsive workbench, Immersa Desk.

Other

3D modeling and animation, photo-realism and optimizing models for real-time applications, visualization of architectural spaces.

Sample Projects

Professional Work

- Tis Hazari MRTS Railway Station, India, Jasbir Sawhney & Associates
  http://filebox.vt.edu/users/pmalhotr/resume/dmrc.htm
- Hyatt Regency Hotel, India, Jasbir Sawhney &Associates
  http://filebox.vt.edu/users/pmalhotr/resume/hyatt.htm
- Sanskriti School, India, Jasbir Sawhney &Associates
  http://filebox.vt.edu/users/pmalhotr/resume/sanskriti.htm

Summer Internship, Haes, Seay, Mattern & Mattern, Inc.

- Virtual reality simulation of Wackenhut Prison, North Carolina
  http://filebox.vt.edu/users/vbalasub/pmalhotr/virtualwackenhut.htm
- 3D visualization, Von Braun Complex, Redstone Arsenal
  http://filebox.vt.edu/users/vbalasub/pmalhotr/VonBraun.htm

Academic Projects

- Masters Thesis- Studying issues involved in real-time rendering of virtual environments with special emphasis on photo-realism and optimization of models for real-time simulation.
  http://filebox.vt.edu/users/pmalhotr/resume/thesis.htm
- Process of creating a CAVE walkthrough of an Architectural model.
  http://www.sv.vt.edu/classes/ESM4714/Student_Proj/class00/malhotra/3DSIMAG /HTML/index.html
- 3D Modeling and Rendering
  http://filebox.vt.edu/users/pmalhotr/resume/renandmod.htm
- Creating a QuickTime VR movie of an architectural model and adding interactivity using Director
  http://filebox.vt.edu/users/pmalhotr/resume/finalproject4.htm
- QuickTime VR of an Architectural model
  http://filebox.vt.edu/users/pmalhotr/resume/QTVR.htm
- Interacting with objects in architectural models in the CAVE
  http://filebox.vt.edu/users/vbalasub/pmalhotr/interactivewalkthru.htm
- Mars attacks, a short animation
  http://filebox.vt.edu/users/vbalasub/pmalhotr/marsattacks.htm

Publications

Undergraduate Thesis
'Kalaneri', An Integrated Arts and Crafts Settlement, Rajasthan, India, Published for college records, Jan 1998.

Dissertation
'Idea of Spontaneous Urban Space', Published for college records, April 1997

Student Project
Transforming a fort into a restaurant- Revitalization of Athar Mahal, Bijapur. Published in 'Architecture+Design', A journal for the Indian Architect, Volume XIII-Jan-Feb 1996.

Relevant Coursework
Design, Building Construction, Structural Mechanics, Urban Design and Planning, Arts and Graphics, 3D Modeling and Rendering, Computer Applications in Design

(Programming in C++), Advanced Animation, Virtual Environments, Time based authoring (Director 8 and Lingo), Scientific visual data analysis and Multimedia, Digital Video.

Attributes

Focused, intrinsically motivated, team player, research background, creative, imaginative, highly trained in design, 3D modeling and Animation.

Affiliations

Registered Architect, Council of Architecture, India.

Virginia Tech virtual environments research group, CAVE SLUG (Student Lead User Group), 3DUI(3D User Interaction) group.