Real-Time Rendering For Autostereoscopic Display Technology

by

Alexander Douglas Raine

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Electrical [Computer] Science and Engineering

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 22, 2000

[June 2000]

Author_____
Department of Electrical Engineering and Computer Science
May 22, 2000

Certified by_____
Stephen A. Benton, Ph.D.
Thesis Supervisor

Accepted by_____
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Real-Time Rendering For Autostereoscopic Display Technology

by
Alexander Douglas Raine

Submitted to the
Department of Electrical Engineering and Computer Science

May 22, 2000

In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Computer [Electrical] Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

The Mark II autostereoscopic display system is a fully-functional demonstration of interactive stereo image display without the need for special glasses or constraints on the user' location. The computer-graphic renderer for the Mark II is a modified version of the Genesis 3D game engine that provides a fully interactive, immersive environment that highlights the capabilities of the Mark II display. While previous versions of the autostereo display were limited to pre-rendered static frames, the new version supports dynamically rendered worlds that users can fully explore. Interaction with the display can be through separate controls, or simply through the user's head motions while watching the display. New facetracking technology also provides more robust tracking and the potential for new tracking features while reducing system requirements. The entire system can also now be run by only one PC (rather than the previous two SGI workstations).

Thesis Supervisor: Stephen Benton
Title: E. Rudge ('48) and Nancy Allen Professor of Media Arts and Sciences

Acknowledgements:

I would like to thank all of the Spatial Imaging Group at the MIT Media Lab for their help and support. In particular, I'd like to thank Professor Stephen Benton for the chance to work with the group for so many years, Wendy Plesniak for both for all of her help over the years and for getting me involved with the group in the first place, Steven Smith for several years of projects and support, and Elroy Pearson and Aaron Weber for advice, assistance, and enthusiasm. Thank you also to the various sponsors of the Spatial Imaging Group and the Digital Life Consortium for their continued support of our work over the years, and to all of the other faculty, students, staff, and hangers-on at the Media lab who make it such a wonderful place to work and learn. I would like to thank the users of the Genesis 3D forum, who helped me with many technical questions and challenging issues, and the developers of the Genesis 3D engine for creating such a fine product and making it available to users like me. Finally, thank you to my wife Teresa, for giving me motivation, support, and a reason to finish my work on time every day.

Table of Contents

Section 1: Introduction

Conventional display systems show us a two-dimensional representation of the world, whether they are displaying text or a rendered image. For many applications, this is perfectly acceptable; there is no need for a three-dimensional image when one is dealing with an ATM machine. However, applications that could benefit greatly from a fully 3-D display are becoming increasingly pervasive in our society. Product prototype model development, architectural design walkthroughs for interior decorating, teleconferencing, and even video games could all take advantage of a display that incorporates depth.

Unfortunately, 3-D displays have their own set of drawbacks. Holographic displays, such as the MIT Media Lab holographic video [1] system, require enormous amounts of computational power to achieve even tiny images with slow refresh rates. Stereoscopic systems, which comprise the majority of 3-D displays, require some sort of glasses or head-mount to separate the right and left eye views for the user. Various other autostereoscopic display systems have their own downsides, from extra rendering overhead to gigantic footprints for the display [2]. The Mark II seeks to remove most of these obstacles through a combination of facetracking technology, a real-time rendering engine, and a polarization-based image projection system. The Mark II also has the advantage of being easily mass-produced, and can be constructed mostly from commonly available parts. Perhaps best of all, the system is flexible enough to serve as both a 2-D and 3-D display, and converting software to work on it and utilize its 3-D capabilities is simple and straightforward.

The scope of this thesis was the development of a software architecture for the Mark II that would allow real-time rendering of an interactive environment, while preserving the benefits of the facetracking and stereoscopic aspects of the display. In addition, a new facetracking program was adapted from the IBM Blue-Eyes project [3], both to improve the performance and robustness of the code and to demonstrate the modularity and extensibility of the system design. In the process of establishing this new architecture, the system was also ported from SGI's Irix operating system to Windows

98/NT, dropping the system hardware cost from well over $30,000 to under $5,000 while making it compatible with a much wider variety of existing systems and programs.

Because of the time constraints of the project, and the desired level of interactivity and feature set, the decision was made to base the new renderer on an existing graphics engine. Since the most up-to-date, robust, feature-complete real-time graphics engines available are developed for 3-D video games, the logical conclusion was to modify an existing open-source or licensable video game engine so that it could work with the Mark II. Video game engines are now being used to make amateur movies, give walk-throughs of potential real estate purchases, design homes, and teach skills through simulations; it is only natural to use their capabilities in this field of research. Not only did this substantially reduce the amount of effort required to build the application, it also proved that it would be possible to modify existing applications in relatively short order to work with the display.

In keeping with the modular plan for the final architecture, the renderer is a completely separate program from the facetracker. The renderer is compatible with both the original Mark I facetracker and with the new Blue-Eyes IR facetracker. Currently, the facetracker runs on Windows NT or Windows 2000. The renderer runs on Windows 95/98. Future versions of the system are planned to share the same OS.

Section 2: Background

Previous System:

The Mark I Autostereoscopic viewing system was developed by the MIT Media Lab's Spatial Imaging Group [4]. Its purpose was to serve as a proof-of-concept for a new approach to stereoscopic rendering that did not require special glasses or head-mounted displays for a full stereoscopic effect.

The Mark I used a similar physical configuration to the Mark II's design. The only real difference in the two systems is in the underlying computer hardware and software, and the resolution, scale, and optical properties of the LCDs and lenses. The Mark I used an SGI Indigo workstation to serve pre-rendered images to the display, and an SGI Octane to track the position of the user and control the display of data from the Indigo. The images displayed by the Indigo consisted of a number of pre-rendered frames, each of which consisted of a right-eye and left-eye image of the scene in question vertically interlaced together. The succession of images together created an animation of the scene. Images early in the animation were displayed when the viewer watched the screen from left of center; images later in the animation were displayed when the viewer watched the screen from right of center. By shifting their point of view from left to right across the screen, users would see the animation played out in stereoscopic vision.

Display Setup:

The basic setup of the Mark II autostereoscopic display consists of an image LCD, a micropolarizer-equipped LCD (constructed by VRex Inc.[5]), a light source, a video camera, and the rendering and facetracking computers. Various optical components are also in place to expand the field of view and improve the display, but these can be neglected in a simple overview of the system (Figure 1).

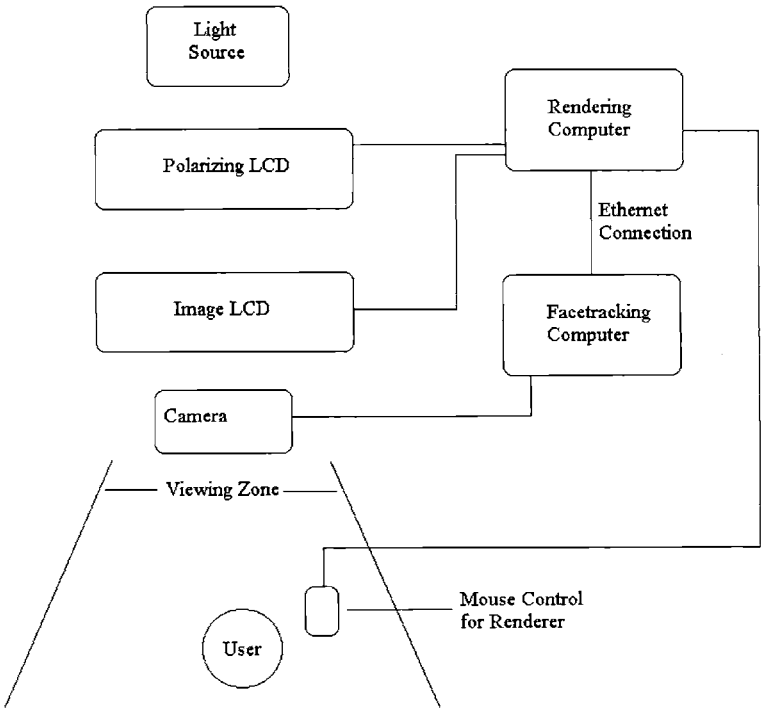

Figure 1: Basic overview of the Mark II display setup

The video camera is directly linked to the facetracking computer. The camera is set so that it views the entire expected viewing area of the display. If a user is in a position such that they are able to see the display's content, the camera can see them. The type of camera will vary according to the requirements of the facetracker. In the Mark I system, a simple color digital camera was used. In the Mark II system, an infrared-sensitive camera with special IR LED's and additional timing circuitry is used. Viewcast's Osprey 100 Video Capture board was used to bring the video signal into the computer for processing.

The facetracking computer in the Mark I system was an SGI O2. In the Mark II, it is a PC running Windows NT or Windows 2000. Once platform compatibility is addressed, the facetracking computer and the rendering computer will be the same machine; for the time being, they must be two separate but networked computers.

The rendering computer in the Mark I was an SGI Indigo2. It had the relatively simple job of serving up previously rendered frames in response to network calls from the facetracker. The rendering computer for the Mark II is a PC running Windows 98 SE. It

includes a Matrox G400 graphics card, which allows the computer to output to two monitors simultaneously. This allows the rendering computer to control both the micropolarizer-based display and the image display. In addition, the system is equipped with a Microsoft Intellimouse Explorer. This mouse has the advantage of 5 buttons (which can be programmed to correspond to keystrokes), and will work on any surface. This mouse serves as the second form of interface to the display; the facetracking is the first.

The image-bearing or micropolarizer-equipped LCD is a normal LCD screen that has been modified so that alternating lines may be illuminated only by alternating polarizations of light. In other words, the first line of the display is sensitive only to horizontally polarized light, the second sensitive to vertically polarized, the third to horizontally, fourth to vertically, etc. If horizontally polarized light shines on a line sensitive to only vertically polarized light, the result is a black screen. If light of the correct polarization shines on a given line, however, it performs as a normal LCD screen. Shining polarized light over multiple lines of the display has the effect of selecting out every other line, displaying only half of the screen's vertical resolution. Thus, if two images (L and R) are displayed interlaced on the screen (Figure 2), and polarized light is directed through the LCD, only the appropriate image will appear. Switch the polarization, and the first image disappears while the other image appears. This is the basic mechanism by which the display selects between displaying the left-eye view and right-eye view.

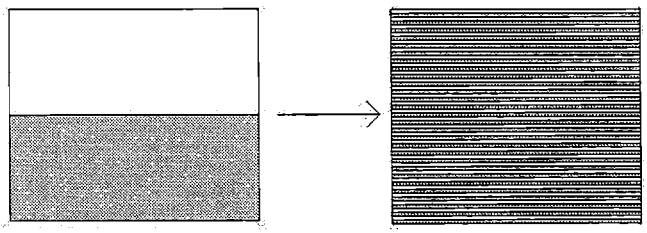

Figure 2: Interlacing image A and image B together.

The polarization-switching LCD is simply an LCD screen with the second polarizer removed. Thus it polarizes the light passing through it depending on the

"color" of the pixel the light passes through. If the pixel is black, the transmitted light is polarized vertically. If the pixel is white, the light is polarized horizontally. Other grey-scale levels produce angled polarizations, but this feature is not used in the display. The micropolarizer is the component that gives the display directional selectivity over where the left and right eye images are displayed. Light travels from the light source (which is a simple clip-on lamp) directly through the polarization-switching LCD, where it is polarized either vertically or horizontally. Because the light travels directly from the lamp to the user, passing through the LCD screens, a user looking towards the lamp through a region of the micropolarizer-equipped LCD that is polarized vertically, they will see the corresponding selected image. Since a user's eyes are offset from each other by a small distance, each eye views the lamp through a slightly different section of the image of the polarization-switching LCD. If one of these sections is black, and the other white, each eye will see a different image. By changing the polarization-switching LCD so that a user is always looking through one black section with one eye and one white section with the other eye, we can consistently show the user a pair of images (one to an eye). If these images are a stereo pair, the user will see the resulting scene in 3D.

Rendering Technology:

Rendering in the new system is accomplished through a fairly standard real-time rendering pipeline, as used in most 3-D video games currently on the market. The rendering engine used is an implementation of the Genesis 3D engine v1.0. The surfaces of objects and environments are modeled entirely with triangles, resulting in a fair approximation of the intended shape. The rendering is performed on a Windows 98 SE platform, with a Matrox G400 Dual-Head Output graphics card. The API used for the rendering driver is Microsoft's DirectX6 Direct3D.

Section 3: Architecture of Current System

The Mark II software architecture can be divided into three primary sections: the renderer, which controls the state of the virtual world and the user's non-facetracked interaction with it; the driver, which controls the majority of the low-level rendering and the interface to the graphics card; and the facetracker, which interfaces with the camera through a video capture card in order to determine the location of the user, and transmits this information to the renderer. Figure 3 gives a schematic overview of the hardware and software interaction in the system.

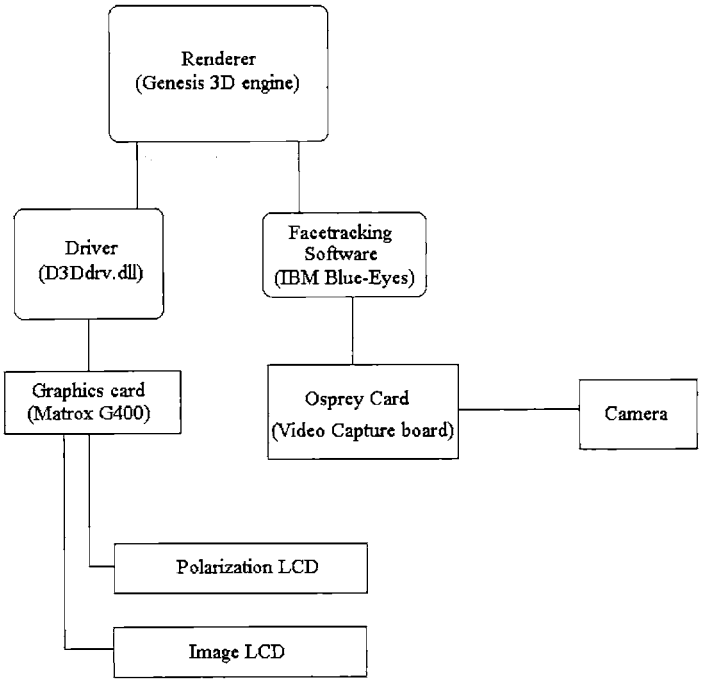

Figure 3: Hardware and Software Architecture Overview

Renderer:

The rendering application is based on a simple client-server model, where the server maintains the state of the world, and the client deals with rendering and relaying user inputs and actions to the server. The majority of the modifications to the engine for the Mark II application dealt with modifying the Client object and operations loop, as well as the rendering process.

The Client object records persistent information about the application and the state of the user's interaction with the system. It contains pointers to objects and constructs needed by the operations the Client object's procedures perform. In addition, it contains information about the current rendering drivers being used, and data about the placement of the user's character in the world.

Driver:

The rendering driver is a separately compiled .DLL file that allows the engine to interface with the graphics hardware according to a specific API. For example, using the Direct3D driver (D3Ddrv.dll) will translate commands from the renderer into Direct3D commands for the graphics hardware to execute. So far, only the Direct3D driver has been modified to work with the Mark II system; there are also software drivers and Glide drivers, and plans for an OpenGL driver exist.

Facetracker:

The facetracker is original C++ code from IBM's Blue-Eyes project. It contains a basic thread structure that continuously captures interlaced video frames from an infrared-sensitive camera. Frames from the camera are interlaced combinations of images recorded by the camera 1/60th of a second apart, giving it a total refresh rate of 30 Hz. During the first frame, the camera's field of view is illuminated by infrared LEDs positioned close to the lens aperture, so as to capture the "cat's eye" retroreflection from the user's eyes. During the second, this illumination is moved far from the lens aperture, so that the retroreflection misses the lens, but the general infrared illumination of the face stays almost the same.

When the facetracker receives an interlaced frame from the camera, it first de-interlaces it into the near-the-lens-illuminated image (call it image A), and far-from-the-lens-illuminated image (call it image B). Because of the high relative brightness of the retroreflections from the eyes, the primary difference between image A and image B should be a much higher intensity at the location of the pupils in image A. This can be extracted from the rest of the image by subtracting pixel values of image B from image A, removing everything that was constant between the two images. A thresholding function can then be used to remove extraneous noise in the image, and to equalize the

remaining image bright spots. This thresholding function in the original IBM code was based solely on the histogram of the image after subtraction; however, after some experimentation, we had better results with a threshold equal to the histogram plus an additive constant.

After the thresholding, the IBM software uses a process that detects appropriately sized and shaped areas of contiguous saturated pixels; if this process detects a blob it deems potentially an eye, it calculates the centroid of the blob and places it in a list for further examination. This list is then iterated through, and those blobs that fit specified criteria are deemed to be eyes. In the original facetracker, this simply means that they are highlighted on the screen; in the Mark II version, the eyes are then paired, and the position of the pair's center is calculated and sent to the renderer. Figure 4 gives a step-by-step overview of the process.

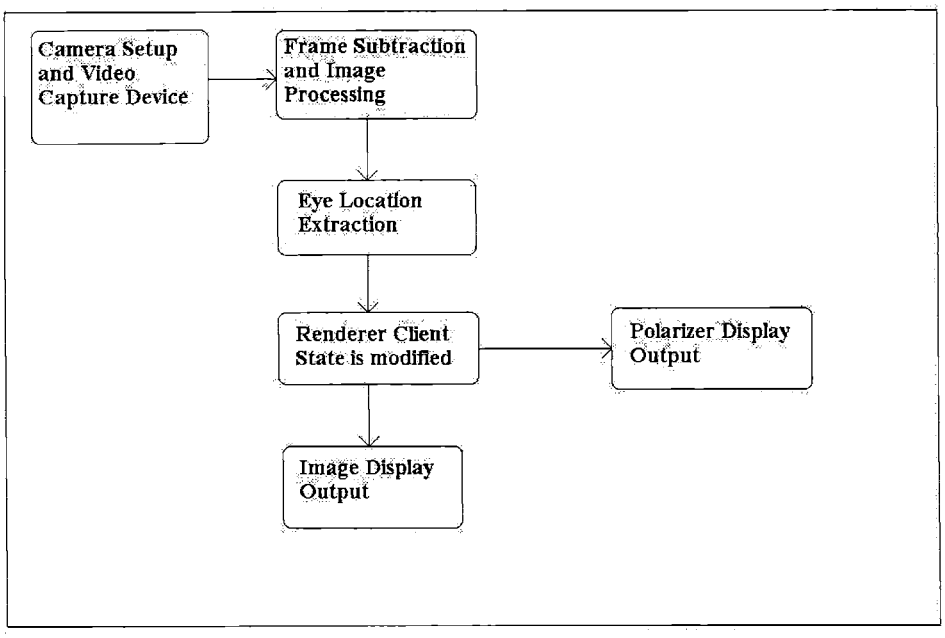

Figure 4: Facetracking and rendering process outline

Section 4: Rendering Engine Selection

Early on in the development of this project, it was decided that coding a new rendering engine from scratch would not be the most effective solution. It would it add large amounts of code development overhead, resulting in a product that would be in many respects inferior to other available Open-Source rendering engines. Using a commercial or freeware engine would also allow us to demonstrate how straightforward it would be to convert existing software to utilize the display's capabilities. In addition, by using a rendering engine developed specifically for first-person video gaming, we could gain high quality real-time hardware-accelerated rendering capabilities, physics modeling, network interaction code, artificial intelligence support for interactivity with the user, and support for the software from both the developers and from the user community associated with the engine. With these benefits in mind, a list of criteria was created by which potential rendering engines could be evaluated.

Criteria:

1. Availability

   Regardless of the capabilities of the engine in question, without the full source code of the engine being accessible in a timely manner, the engine is useless to us. Note that just because products utilizing the engine are available, or Software Development Kits (SDKs) for the engine are available, does not mean that the engine is a viable alternative. For example, the Quake 2 game code was released almost simultaneously with the game's release; however, this was code that governed play mechanics only, and would not have allowed us to make the rendering alterations necessary to implement the capabilities of the display. The rendering engine source code itself must be available. Also, many gaming engines are available for licensing from their creators; however, such licenses are often prohibitively expensive, and would be beyond the scope of a thesis project.
2. Engine Quality

While this is a somewhat subjective category, it can be considered quantitatively in terms of the feature set of the engine. What kinds of "eye candy" are included, such as fog, colored lighting, mirrored surfaces, etc.? What are the limitations of the engine in terms of size of areas rendered, speed of rendering, maximum polygon counts, etc.? How much support is present for physics modeling, AI code, network interactivity, and other extensions of the basic engine? How complete is the code? If further features are planned, when will they be completed (and how realistic is that deadline)? Finally, what rendering APIs are supported, and what operating systems? DirectX, OpenGL, Glide, and software rendering are all alternatives for running on a Windows platform, and future decisions about the modifications to the engine must be based on what APIs are supported.

3. Tools and Applications

   A rendering engine itself is of little use if there is no way to create content for it to render. Tools are programs that allow content creation in a format that can be used by the rendering engine. For example, level-design software allows the structure, design, and implementation of the virtual world that users interact with. Modeling software allows the creation of entities within that world, from desk lamps to people; in general, those things in the virtual world that are directly interacted with are built in modeling software. Textures for texture-mapping are also created in a separate program, which can be anything from a digital photograph to an image made in a painting program. Since most engines use their own format for each type of content, it is important to find an engine that either provides its own content creation tools, or is supported by other freely available tools.

4. Code Quality

   Is the software written professionally, with full documentation, commented code, and easily understood variable and procedure names? Is there a clear, consistent outline of the rendering pipeline available? Is there test code, or a demonstration package that contains all of the basics of the engine implemented in a clear fashion? While this criterion may make little

difference in the final product, it will substantially affect the amount of time it takes to complete the project, and the ease with which future researchers may continue the work.

5. Community Support

   Many gaming engines develop an on-line following of individuals devoted to modifying, improving, and expanding upon the capabilities of the engine and the games made with it. These communities are an invaluable resource for developers, as they present tutorials, example code, forums for discussion, technical support and advice, and archives of information that often dwarf that provided by the engine developers themselves. The larger a game engine's community is, and the greater the experience of its members, the easier and faster new developers can create their own modifications to the engine.

Alternatives:

After extensive web searching, four viable alternatives were chosen for evaluation. They were:

1. Quake Engine (id Software)--http://www.idsoftware.com/ [6]

   Developed by id Software for the game Quake, one of the first and most influential games in the genre of first person shooters. While it is an older engine, it has been dramatically updated by the user community. In addition, it has been proven over time to be a very stable, very robust engine with many successful licensees. In 1999, id announced plans to release the full source code of the engine as soon as the last licensee using the code had shipped their game (Anachronox, by Ion Storm).

2. Golgotha Engine--http://gameznet.com/golgotha/pages/crack.html [7]

   The Golgotha engine is managed and extended by an independent group of programmers that took over the source code from a development company that went defunct (Crack.com). When the company went under (due primarily to mismanagement and poor planning), the owners decided to release their source

code to the public so that others could profit from their mistakes and successes. Thus, the engine is Public Domain, and not under a license such as GPL.

3. Crystal Space-- http://crystal.linuxgames.com/ [8]

   Crystal Space is a gaming engine distributed under the LPGL license. It was developed initially as a personal programming project that has grown into a fairly major community development effort. It is in a constant state of development, with new features being continually worked on.

4. Genesis 3D SDK--http://genesis3d.com/index.html [9]

   The Genesis 3D engine was developed by Eclipse Entertainment as a potential commercial venture, and was then released to the public as an Open-Source program. It has been used for numerous free products, and was recently used to create the commercially available game "AI Wars".

Evaluations:

#1 : Genesis 3D SDK

Genesis 3D was the clear winner in the evaluations. Initially, it was considered a poor candidate because its source code was not freely available; however, a decision by Eclipse Entertainment to make it Open-Source made it completely available and fully accessible. The engine code itself is of high commercial quality, while the example application was of slightly less quality. However, there is a strong community behind Genesis, with tutorials and resources available at a number of sites, and a forum run by Eclipse that can provide answers to technical questions. The forum is frequented by both community members and the programmers at Epic, providing a wide range of experience and technical knowledge. While the tools available are comprehensive, and 3rd party applications exist that can substitute for the official programs, the tools are slightly below professional level.

#2 : Crystal Space

Crystal Space was freely available under the LPGL license. In addition, it had a fairly strong community, though weaker than that for Genesis 3D. However, the code was of a slightly lower level (both in terms of features and documentation), and tools were (at the time) less comprehensive and less usable than those of Genesis 3D. This combination made it a decent second choice.

#3 : Quake Engine

The Quake Engine would have been ideal in terms of community support (the Quake on-line community is both significantly older and larger than any other considered for this project) and quality of source code (it has been used for numerous successful licensed games, and its programmer is regarded as being one of the best programmers in the industry). In addition, there are numerous tools available for it, both from the developers and from 3rd party developers. However, the source code was not released to the public until 12/21/1999, which would have caused a delay of up to half a year of work on the project. In addition, the source code is somewhat less feature-complete than the other engines being considered.

#4 : Golgotha Engine

The Golgotha engine was the least likely candidate for the autostereo display. While the source code was freely available, it was not very complete, and the community was relatively small (at least at the time it was first being considered). The tools available were relatively incomplete, and the engine quality was below that of the others. While this alternative would be easier than coding an engine from scratch, it was clearly the last choice.

Section 5: Rendering Engine and Tools Overview

Engine:

The Genesis 3D engine is a robust professional-quality rendering engine with the following features:

- Exceptionally fast rendering
- Radiosity lighting
- Integrated rigid body physics simulation support for world objects
- Pre-computed lighting for animating light intensities and simulating caustics
- Environment uses BSP trees for fast visibility culling
- Dynamic RGB lights
- Dynamic shadows
- Dynamic fog
- Dynamic mirrors
- Dynamic water effects
- Dynamic texturing effects such as procedurals, animations, blending, and morphing
- Area portals allow selective rendering of world geometry
- Translucent world geometry for windows, or other effects
- Spherically mapped sky for seamless sky and horizon
- 3D sound positioning and attenuation
- User extendable special effects and particle systems
- Bitmap and Windows font support for labeling the screen, textures, or bitmaps
- Skeletal animation for characters
- Second-order interpolation between animation keyframes
- Basic network transport support
- Multiple-camera rendering
- D3D, Glide, Software, and AMD-optimized Software rendering support

The engine is highly flexible, and all source code is available for modification under the Open-Source license. A wide variety of existing content can easily modified to

work with the engine, both in terms of actor models and environments, and numerous content creation tools are available that produce content compatible with or portable to the Genesis 3D engine. There are many programming resources and tutorials for the engine available on the web, and a large user community that includes professional developers, the original programmers and designers of the engine, and amateur game designers and hobbyists.

The Mark II uses version 1.0 of the engine, and version 1.1 of the Genesis SDK. The Genesis 3D engine is currently undergoing a major revision to version 2.0; when released, it will be substantially more feature-complete, and will be competitive with the highest-quality licensable engines available. In addition, the editing software (Gedit) will undergo a major overhaul, and will be substantially more user-friendly and feature-complete. A demonstration of Gedit V2.0 has been released, and judging by it version 2.0 of Genesis 3D will be an ideal future platform for the Mark II, with a minimum of work put into transferring the altered code.

Gtest:

Gtest is a sample application released by Eclipse Entertainment with the Genesis 3D engine source code. It demonstrates the basic functionality of the engine, allowing multiplayer deathmatch in several sample levels with computer and/or human opponents. While it is regarded as being somewhat rushed and poorly coded by most users of the engine, it is fairly feature-complete, and it serves as a decent foundation for larger and more substantial projects like the Mark II. The Mark II currently runs with a version of Gtest that has been modified to include multiple-monitor support, multiple-camera rendering, an interlacing video driver, and socket server capability for facetracking clients.

D3D Driver:

A driver is a pre-compiled piece of code that allows a program to interface with a specific piece of hardware. In this case, the D3D driver allows the Genesis 3D engine to used Microsoft's Direct3D API to communicate with the graphics hardware. The driver works on a much lower level than the Genesis 3D engine does. While the engine will simply have a command to render a shape in the scene, the driver actually goes through

and draws each triangle of the shape in turn. (Note that the Direct3D API works at an even lower level, actually doing the computation to render each pixel and apply the texture mapping to the shape.)

Because the driver works at a lower level than the engine, it has more direct control over the final image. In fact, the engine cannot directly access the video memory in which the image is stored before displaying; only the driver has direct access to that section of memory. This is important to the project because the interlacing of the stereo views for the display can only be done after the image has been rendered, but before it has been copied onto the screen. The engine cannot do the interlacing, since it does not have control of the image or access to the memory at the critical stage in rendering, so the driver must be modified to do the interlacing itself.

Genesis 3D can work with Direct3D driver, as well as a Glide driver and a software driver. Both D3D and Glide utilize the hardware features of the computer's graphics card to improve graphics performance. The software driver performs all rendering calculations on the CPU of the computer, resulting in drastically reduced performance but improving portability and compatibility. The Direct3D driver was chosen for this project over the software driver for performance reasons, and over Glide because only graphics cards produced by 3dFX support Glide (it is their proprietary API). With the focus on keeping the renderer flexible and non-hardware dependent, Direct3D is the best option.

Gedit:

Gedit is a level-editing program that comes packaged with the Genesis 3D engine. It has all of the necessary basic functions of a level-editor, but is not a fully complete commercial product. However, the source code is included with the Genesis 3D source code, so modifications can be made to it as needed.

"Levels", in 3D games, are contained environments in which the user can roam. A level might be a floor of a building, the bottom of a rocky canyon, or the top of a skyscraper. Within a level, there are objects that are effectively part of the world (such as fixed, immovable desktop lamps, or chairs, or boxes), entities (pieces of the world that

the engine must interact with in a special way, such as lights or tools that the user can pick up), and actors (special entities that have some ability to interact with the world around them, such as artificially intelligent characters). Entities and actors are normally created in modeling programs, such as MilkShape 3D or 3D Studio Max. Environments are normally created in level-editing programs, such as Gedit and WorldCraft.

WorldCraft:

WorldCraft is another level-editing program, used primarily with Quake-engine games. It is a full commercial product from Valve Software [10](makers of Half-Life), and has many features unavailable in Gedit (such as undo capabilities and individual vertex manipulation). Since both Quake-engine games and Genesis 3D use geometry based on Binary Space Partition (BSP) trees, the levels output by WorldCraft can be imported in Genesis 3D with a minimum of effort. WorldCraft is most useful in this project for designing complex environments that contain detailed geometry, which are very difficult to produce in Gedit.

Milkshape 3D:

Milkshape 3D is a modeling program developed by Chumbalumsoft [11]. It is a shareware program that provides modeling support for Genesis 3D, as well as most other major video gaming engines. It also supports skeletal animation, texture mapping, and material definitions. Modeling programs are used for creating actors (see Gedit entry). They allow the creation of complex polygonal models for characters, enemies, weapons, and other such gaming elements. Most modeling programs also provide support for animating such models, giving them pre-defined motions that can be triggered by the game engine at the appropriate time.

Wally:

Wally is a texture editing suite developed by Ty Matthews and Neal White III [12]. It is a freeware program, and works with most major video game engines and rendering programs. Wally allows users to make textures, which are used in the rendering engine to cover surfaces and make them appear to have more detail than is present in their geometry. For example, a carpet contains a large amount of surface

deformation, color variance, and shading differences that cannot be captured by a flat surface. But modeling the details on the carpet in actual geometry is prohibitively difficult, and would result in large amounts of computational overhead for the renderer. It is much simpler to model the carpet as a flat surface, and color that surface with an image of a carpet. The effect is surprisingly effective, and computationally not much more expensive than rendering a plain flat surface. Textures are often repeated ("tiled") over a large surface, which means they must be created so as to show no seams when adjoined with a copy of themselves. Wally facilitates this by allowing users to draw across tiling boundaries, allowing them to easily create textures that have no discernable seams when tiled.

In the environments created for demonstrating the Mark II, Wally was used to create floor, wall, ceiling, and object textures.

Section 6: Demonstration Application and Environments

Basic Application:

The basic application is a simple walk-through of different environments that has been enhanced to use the features of the Mark II. Facetracking controls the user's viewpoint within the application; if they shift left, right, up, or down, the viewpoint shifts with them. This gives more of a sense of immersion, and allows users to look around objects and deal with the interface in a more natural way. Mouselook (a method of interface in which the movement of the mouse controls the way your character "looks" at the world) is also included for angular viewpoint changes, and the buttons on the mouse (a 5-button Microsoft Intellimouse Explorer) control forward, backwards, and side-stepping movement.

The application works with any precompiled Genesis 3D level, and supports network connections to other instances of the engine as well as AI-controlled characters and physics modeling.

Hang Gliding Simulation:

The hang-gliding simulation is a slight variation of the basic application. It was made to demonstrate alternative ways in which the user could interact with the display. While facetracking in the basic application translates the viewpoint of the user in the application, facetracking in the hang-gliding simulation actually controls the hang-glider. The user's velocity through a scene is controlled by their angle of ascent/descent. Their motion side-to-side controls the turning of the glider, and their motion up and down controls the pitch of the glider. The physics of the simulation are also modified to reflect the lower terminal velocity of the glider and the slower fall rate overall. For convenience, jumping in the application has been amplified to return the user to a decent gliding height in the case of landing.

Office Environment:

The office environment is a demonstration of realistic environment rendering. While creating alien, futuristic, or fantastic environments is relatively easy in most game engines, it is often difficult to create more mundane and true-to-life environments. The

office environment is a recreation of E15-095, a room in the basement of the Media Lab where Spatial Imaging was housed for the summer of 1999. It shows the Mark II display (non-functioning, though with some software additions to the engine it would be possible to have a rendered functioning display), as well as the overall office layout as it was at the time, with a few additions. A refrigerator not found in the original room was added to allow demonstration of a simple colored lighting effect, and the exit from the room is not modeled.

Museum Environment:

The museum environment is designed to show some of the more advanced features of the engine. Colored lighting, mirrored surfaces, coronas, electrical discharges, fog lighting, dynamic lighting, and sky texturing are combined with creative geometry to make a simulation of a modern art museum without the limitations of physics. This shows one of the most artistic applications of the Mark II: the ability to create new environments for people to enjoy and experience, with artistic creations that can come to life, interact with users, and achieve effects impossible in real life.

The layout of the museum is four rooms, each with a different theme. The first is relatively normal, featuring a simple couch and some wall paintings (one of which demonstrates transparent textures). The next room features three "sculptures", all of which help demonstrate depth and geometric effects. There is also a mirrored surface with a semi-transparent texture, a corona as part of one of the sculptures, and various colored lighting effects. The third room is a planetarium, complete with rotating star-filled sky. In addition, several coronas coupled with fog lights give the effect of close-by stars, and electrical discharges between the stars give a dynamic touch to the scene. The fourth room is a representation of a historical diorama of a castle dungeon, complete with flickering dynamic-lighting torches.

Section 7: Modifications to Rendering Engine

Several small modifications had to be made to the rendering engine in order to use it on the Mark II display. While the precise nature of the modifications that have to be made to any given 3D engine are dependent upon the architecture of that engine, the set of modifications necessary is constant for most rendering engines. Each of the necessary modifications are discussed in turn, and their particular implementations in the Genesis 3D architecture are explained.

Multiple-camera Rendering:

It is necessary to render two views within the rendering engine for each screen of the Autostereo display. The two views correspond to the left eye view and right eye view of the user. Most graphics engines do not have explicit support for rendering with multiple cameras; however, most do have the capacity to create and control multiple cameras and to assign them to render in different areas of the screen. In the Genesis 3D engine, new data structures were added to the system that allow the engine to keep track of both cameras and their positions, as well as the screen space that they render to.

A separate viewing transform can be specified for each camera, based off of a simple transformation of the absolute location of the player model. For the right eye camera, the transform is simply a shift of x units in the direction of the vector to the right of the player model's transform, where x is ½ of the offset between the two eyes. The left eye camera's transform is a shift of $-x$ units in the direction of the vector to the right of the player model's transform. Identical rendering processes are then run on both cameras, resulting in images of the scene from the two viewpoints specified by the transforms.

Offset x in the final version of the software is a variable that can be tweaked dynamically by the user (by pressing F7 to lower the offset, F8 to increase it). This allows the user to set an eye separation that is most comfortable for them without recompiling the source code.

Interlacing of Stereo Views:

In order for the two views to be separated by the polarization of the autostereo display, they must be interlaced (so that the lines of the left-eye image are on the odd lines of the interlaced image, and the lines of the right-eye image are on the even lines). Interlacing can be done at any point in the rendering pipeline that is supported by the engine architecture. However, the most convenient and efficient point at which to interlace is likely to be after all rendering has been completed, but immediately before the images are copied from video memory onto the screen.

In the Genesis 3D engine, the most convenient way to perform the interlacing is by doing a series of "blits" (copy operations that move sections of visual information to other sections of video memory) to put the alternating lines into the correct arrangements. This must be done at the video driver level, since higher levels do not have direct access to the video memory. Therefore, the Direct3D video drivers were modified so that the entire screen is divided into top and bottom sections, which are then interlaced with blits. This approach has the disadvantage that whether or not the renderer is providing multiple camera views, the screen is being interlaced. Future software versions may include a method for toggling the interlacing on or off depending on the rendering method being used.

Socket-Based Communication with FaceTracker:

The facetracking program runs entirely separately from the renderer. This is done primarily to allow different facetracking methods to be quickly and efficiently interchanged without altering the rendering code. In fact, the renderer does not even have to be re-launched to switch facetrackers; one can launch the renderer, link a given facetracker, un-link it, and link another facetracker without ever having to exit the renderer. This is accomplished through the use of Windows sockets programming. Sockets are virtual ports on the computer that can serve as message buffers for programs. Information can be written to a socket by one program, and read by another program. This can happen on a single machine, across a local network, or even across the Internet.

In order to make socket communication feasible between the renderer and facetracker, both must be modified to include new code. In the case of the facetracker, code is added to open a connection to a specified computer over a specified port. Next, code is put in place that transmits current tracking data over the socket during each iteration of the facetracking algorithm in which an eye is found. For the renderer, code is put in place to listen for socket connection attempts over a specific port on the computer. Then, code is added to read messages sent over the socket once a connection has been made, and to apply the contents of those messages to the transforms of the player model's viewing positions. In the Genesis 3D engine, this was done by adding code to the client data loop in which the transforms are generated. Each time a transform was generated, a check was made to see if there were new messages on the socket connection. If so, the parsed contents of the new message were applied to the transforms. If not, the most recently received message was still used to adjust the transform, in order to maintain consistency.

Multiple-Monitor Display for Polarization:

In the Mark I setup, the facetracker was responsible for displaying the polarization image that selected out left or right eye views. Responsibility for this was shifted to the renderer for two primary reasons. The first is that the end goal of the display is to have it run by a single computer; when most video game engines start up, they blank out any multiple-monitor displays in favor of the primary screen. This makes having a separate application rendering the polarization display very difficult, since running the renderer will make it impossible for the facetracker to update its screen. The second reason for shifting responsibility for the polarization display to the renderer is that updating the system with a new facetracker is far easier if only network communications code has to be added to the facetracker. If polarization rendering code also has to be added, it is more difficult to adapt new facetracking software to the display, and features or constraints of the polarization display will not stay constant between facetrackers.

Section 8: Facetracking software

Neural Network Version:

The original neural network facetracking program used in the Mark I was coded by Tom Slowe and modified by Adam Kropp [13], based on research by Rowley, Baluja, and Kanade [14]. It consisted of a neural network which was trained to look for human faces in the field of vision of a digital video camera, and report their position in that field of view. This position was sent over a socket connection to the image server program, running on a separate computer, which served images appropriate for the position of the viewer's face. In addition, the program placed a masking rectangle on a portion of the screen. This portion of the screen was output to the polarization screen of the display using the standard SGI "videoout" command, where it controlled the state of the polarization-switching LCD to select the field of view appropriate to the user.

Modified Neural Network Version:

Relatively few changes were made to the neural network facetracker to get it functioning with the real-time renderer. Only a few minor changes to the format of the data sent out over the socket connection were necessary, and those only because of platform-dependent programming issues. The basic overall function of the facetracker remained the same, including the control of the polarization switch.

Infrared Version:

The infrared-illuminated facetracker was originally part of the IBM Blue Eyes research program. The initial version had no capacity for external communication with other software packages, and was remarkably insensitive to the presence of eyes in our testing conditions. It also had no capacity for display output that could serve as a polarization switch control.

Modified Infrared Version:

The original plan for the project was to integrate the facetracker directly into the renderer; however, this would have been relatively difficult to do because of programming languages, program architecture, and overall complexity. Therefore, since the framework was already in place in the renderer for socket-to-socket communication, it was decided to keep the facetracker as a separate program and to simply add socket-based communications to it. Responsibility for the polarization display output was shifted to the renderer, using multiple-monitor display capabilities added to the rendering engine. This minimizes the amount of extra code needed to modify a given facetracking program to work with the renderer.

Section 9: Applications

The special capabilities of the autostereoscopic display make it well suited for a number of different applications. The following sections list various possibilities that have already been considered by the research group and its sponsors, and which could be achieved with little or no additional software or hardware for the system. This is by no means, however, a comprehensive list of the possible extensions of the basic mechanism of the current system; the Future Expansions section (Section 10) has details on more far-reaching applications.

Virtual Worlds:

By increasing the level of interactivity possible with the virtual environments being displayed, the Mark II can make the exploration of such environments more engaging, fun, and stimulating for the user. Video games are the most common way in which users interact with a virtual world, and they would benefit greatly from any increased level of immersion for the user. Fooling the user into thinking that they are actually exploring an alien or fantastic environment requires suspension of disbelief, and flat 2-D images that don't change perspective with you require more effort to believe. The Mark II does not provide total suspension of disbelief, but it makes it that much easier for a user to achieve it.

However, there are many other forms of virtual world exploration available, and they are becoming increasingly available to consumers and professionals around the world. For example, Unrealty is a program that uses a modified version of the Unreal graphics engine to show prospective property owners a virtual walkthrough of available real-estate. While a simple walkthrough may suffice to show the basic layout of the property, the added immersion and interactivity of the Mark II display would give users a much stronger sense of the actual appearance of their potential home or office. Similar software packages are available that allow users to remodel or redecorate their homes in a virtual setting, allowing them to see the results of their plans without actually committing to them. Again, while floorplans and 2-D renderings of the finished project are helpful to

the user, adding stereo vision and the ability to look around the environment in a more natural way can improve the quality of the experience, as well as its usefulness.

Tele-Interaction:

The Mark II display can work off of any interlaced video feed. Research is currently underway to allow it to display stereo data from a pair of mounted video cameras that could be connected to a remote-controlled device. While this setup would not be compatible with the perspective-tracking feature of the display (unless the cameras were mounted on a more complicated track system), it would still provide stereo viewing information of whatever scene the cameras are looking at. If the perspective-tracking information from the facetracker is used as control information for the remote-controlled vehicle, a natural and intuitive tele-operation system is formed.

The benefit of stereo information in a tele-operational system is quite strong [15] [16]. When dealing with a display showing the world from a perspective or scale that the user is unaccustomed to, visual cues like the size of familiar objects can be drastically different. Perceiving the world from the height and scale of a remote-controlled car can make an ordinary coke can seem to be closer or further away than would normally be expected. This can make remote control difficult when interacting with the depth of objects. For example, trying to manipulate a robotic arm within the tele-operational camera's field of view such that the arm can pick up a can is very difficult to do, because it is hard to judge whether the arm is behind or in front of the can. Occlusion becomes the only clue to the relative depth of objects in the field of view. However, if stereo viewing information is added, it is significantly easier to perform this task; seeing depth in the scene makes it possible to judge relative depth without relying on occlusion.

Simulations:

A simulation allows you to try an experiment or experience without physically taking part in it. This is desirable for many reasons. For example, simulated hang gliding is a safe alternative to real hang-gliding, and is feasible for unskilled users to try who would end up crashing a real hang-glider within minutes. In addition, simulation is generally cheaper, easier to repeat, and more controllable than an actual experience.

There are a wide range of types of simulation, and each can derive a particular benefit from the Mark II.

Simulations of dangerous situations are often used for training purposes. The more immersive such simulations are, the more effective the training will be, and the easier it will be to relate the training to reality. Hang gliding is a good example; interacting with the simulation by simply pushing buttons on a keyboard, or using a mouse, gives very little of the feel of actually controlling a hang-glider with your body's motion. Interacting with the Mark II display is much closer to the actual experience. While it is not the same as the actual experience, it comes closer than earlier versions of the simulation would.

Situations that are simply too expensive or difficult to create are another key area of opportunity for simulation. Bobsledding is an activity that requires practice to develop skill, but bobsled runs are only available in very select places in the world, and even those locations are subject to seasonal unavailability. An immersive bobsledding simulation would allow year-round training for professional athletes, and would allow amateur athletes to try the sport without requiring them to travel to distant locations. Again, while the simulation is not as good as the actual experience, it comes far closer with the Mark II than with earlier versions, and it is closer than most people will ever get to a real bobsled run.

Many situations involve tests or trials, of either the user or of some simulated actor that the user interacts with. For example, a simulation could model the performance of an SUV on an off-road dirt track, being driven by the user. Being able to change the track with a few clicks of a mouse instead of a few hours with a bulldozer allows trials to be run much more quickly and cheaply. Such a simulation would benefit greatly from the depth-perception of the Mark II (driving around obstacles with one eye closed is difficult at best). While real-life trials must be run eventually, repeated simulations can help identify what the most challenging tests will be, and what the final results will probably be like.

Section 10: Performance Evaluation

Renderer:

The usual performance metric for video game engines is how many frames they can render of a reasonably complex scene per second (frames per second, or FPS). Since this value is dependent on both the scene being rendered (high polygon count makes for a much more difficult render) and on the hardware platform (faster CPU and graphics card = higher FPS), benchmarks are given in terms of scene complexity and rendering platform.

On a Windows 98 Pentium III 450-MHz system, with a Matrox G400 graphics card and 128 MB of RAM, a scene with approximately 2,000 on-screen polygons would render at 20 FPS. Scenes of much lower complexity (around 500 polygons) rendered at approximately 60 FPS. The difference between the 20 and 60 FPS rendering rates, however, was barely noticeable, and the renderer ran smoothly throughout the tests. Frame rates were not affected by the facetracker's operation.

Facetracker:

The facetracker's performance is measured in two separate metrics: reliability and refresh rate. Reliability is a measure of how well the facetracker acquires new users, how well it stays with them through normal motion ranges, and how accurate it is at discerning eyes from background noise or reflective surfaces. It is a fairly qualitative measurement, since we are unable to control many of the situational variables, and since we have few other systems to compare to. The neural network facetracker works on different principles from the infrared facetracker, and thus has a different situational variance and set of control variables.

Refresh rate is simply a metric of how often the facetracker can scan its view range for eyes per second. This is limited by the refresh rate of the cameras, and can never be greater than 30 times a second. It is also hardware-dependent, so statistics are given in terms of facetracking platform.

The facetracking platform was a Windows NT-based Pentium II 350-Mhz system with 128 MB of RAM, using Viewcast's Osprey 100 video capture card. The facetracker

was able to run at maximum speed under these conditions, tracking at a rate of approximately 30 times a second. Reliability, compared to the neural-network facetracker, was a mixed prospect; acquiring of a new user worked much more smoothly and reliably, but staying with a user through quick motion was about the same if not slightly worse. False positives were also a definite concern, as motion discrepancies between the interlaced frames or infrared-reflecting surfaces had a tendency to fool the facetracker. The Blue-Eyes facetracker was also much more sensitive to the user's viewing depth, and would lose users entirely if they moved too far back or forward outside of a range of about 1-2 feet. Finally, some users were simply not acquired by the facetracker; whether this is because of ethnical differences in pupil reflectivity, size of pupils, eye color, or some other factor was not clear. Overall, however, the infrared facetracking system showed strong merit, and has a similar level of reliability to the neural network facetracker (but with different strengths and weaknesses).

Section 11: Future Expansions

New Interaction Methods:

Currently, the extent of interaction with the display is in user movement and standard keyboard-mouse interfaces. However, the display has the potential to add in a wide range of other methods for interaction with it, as well as new ways of applying the current facetracking interface.

Facetracking currently controls either a simple viewing transform of the scene, or controls the motion of the avatar in the scene. However, it would be fairly simple to add code such that leaning to the left while looking at the screen would bring up a Heads-Up-Display (HUD) that gave relevant information to the scene or user. In a tele-operations situation, this HUD might display information about the quality of the remote connection, statistics about performance of the user, position indicators (via GPS or similar technologies), and a listing of objectives to complete.

Haptic interfaces have already been added to 2-D images on one end of the scale, and holographic video on the other [17]. Adding haptics to the Mark II would enable users to physically touch and manipulate the images they see. The Mark II display has the strong advantage over the holographic video system of computational efficiency; only the needed images are rendered, as opposed to every potential view of an object. This advantage allows much more detailed and complex imagery, with reaction times much more in keeping with the speed of a haptics interface.

An aspect of the IBM Blue-Eyes project that was not included in our current facetracker was pupil-direction tracking—the ability to tell which way a user is looking, as well as where their eyes are. This adds a new layer of interaction that is very intuitive and responsive to a user. In a video game, for example, a targeting reticule could appear in the display wherever the user is looking; firing a weapon would aim for where the user was looking, instead of where the avatar was facing. Combined with a voice interface, pupil-direction tracking would allow a user to simply look at an object in the scene and say "what is that?" The computer could respond with an HUD display detailing the object's properties, a pre-recorded (or computer generated) speech about the object, or simply by zooming the camera in on the object for a better look.

Better Tracking:

The current infrared facetracking system has many advantages over the neural network approach. However, it has many of its own problems and shortcomings. Some people have significantly less infrared-reflective pupils than others, due both to genetic background personal variance. Glasses have a somewhat random effect on the facetracking; some people are tracked equally well with or without glasses, while for others glasses make a major difference. Contact lenses have a similar effect. The system as a whole is also very range-sensitive. If the user is not at the proper viewing distance, the facetracker perceives a person's infrared reflection as too small or large to be eyes, and loses them. Because the camera takes the non-infrared saturated image at a slightly different time than the saturated infrared image, rapid motion can cause disparities between the two images, which the facetracker perceives as reflections (and thus may mistakenly identify as eyes). Perhaps most difficult of all, pupils are not unique in strongly reflecting infrared light. Certain types of clothing, nametags, jewelry, and objects in the background of the user can all reflect enough infrared to give false positives, and infrared ambient lighting can render the facetracker effectively useless.

The facetracking software is integral to the proper functioning of the display. Improving upon the basic method, increasing accuracy and reliability, and making the facetracker more robust in a variety of situations can all make the display as a whole dramatically more effective. While no specific plans are currently in place to move to a new facetracking system, there is a significant amount of research being done by other research groups on the problem. The modular nature of the facetracker in relation to the renderer makes it a very straightforward task to adapt new facetracking code to the Mark II. Approximately 100 lines of C++ code are sufficient to add the necessary functionality to a given tracking program, making the transition fast and easy.

Viewer-dependent Content:

Currently, the facetracker recognizes only the presence or absence of human facial features in a scene. However, other groups in the media lab are working on

identifying more about a detected face, such as its owner's identity, gender, age, and mood. Adding this capability to the facetracker would allow it to send information to the renderer about what content to display. A potential application of this would be directed advertising; a display in a public location could show advertisements tailored to men when presented with a male viewer, advertisements aimed at women when presented with female viewers, advertisements directed towards children when a youthful face appeared, and neutral advertisements when presented with unidentifiable faces. Advertisers could even pay a prorated fee based on how many times the display was viewed by the appropriate category of viewers.

Another application that benefits from additional information about the user is privacy-protected displays. Monitors in banks and other personal service offices often have screens in place that prevent anyone not sitting directly in front of the monitor from seeing the information on the screen. However, with a user-recognition program, the display would only show information to a user authorized to see it, regardless of their position relative to the screen. Different levels of information could be shown to different users based on their relative authority and need-to-know. A manager at the bank, for example, would be shown all information, while a teller would only be shown account balances and not credit history or personal information.

Section 12: Ongoing Projects

The Joyce Project:

The Joyce project is an artistic and technical project that seeks to actualize the works of James Joyce through a CAVE virtual reality system. To build the framework of their CAVE, they are using the Mark II system, with some modifications. Their eventual goal is an entirely enclosed environment with autostereo screens covering every wall; a user would have a full 360-degree range of real-time, interactive, perspective-tracking displays surrounding them, providing a completely immersive experience.

There are many technical hurdles that the project must overcome. The greatest challenge in making an environment surrounded by the autostereoscopic display is simply one of optics. Illuminating displays at with a user at various angles of view relative to the screen will require a re-thinking of some of the optical design of the display. In addition, rendering to a large number of screens simultaneously will require a large amount of processing power, especially at higher resolutions and with each screen showing some part of a continuous image (rather than each display showing its own content). Finally, tracking the user reliably and adding in new methods of interaction with the system will require new code and new approaches to the problems. However, if successful, the Joyce project will be one of the most immersive and exciting virtual reality displays available, and may set the standard for such projects in the future.

Section 13: Conclusions

The renderer implementation for the Mark II autostereoscopic system has been a success. The rendering engine has met or exceeded all of the design goals, and it is a strong framework for future work to be performed on. In addition, the modification process has proved simple and straightforward enough to allow conversion of any reasonably well-coded program to work with the display's features in a relatively short period of time. Creating new content for the display is well-supported by existing software freely available on the Internet, with no need for modification. Creating new applications within the current framework is fast and easy, as is changing the method and effect of interaction with the display. In addition, the modular nature of the display makes it easy to switch in new facetracking technology as it becomes available, further improving the expandability and robustness of the display.

The technology of the display itself is fully compatible with existing displays, and could (in the not-so-distant future) replace 2-D monitors without the need for rewriting non-3-D software. The performance of the display is essentially unaltered from that of a normal LCD display. This, combined with the ease of adding new features to existing software, means most of the usual obstacles to new technology adoption are alleviated, if not removed entirely.

Through the use of the Mark II autostereoscopic display system, applications can take on a whole new depth, both literally and figuratively. Interaction is made intuitive and flexible, while image content and complexity is increased dramatically. Entirely new applications become possible, and the room for expansion is significant in many areas. Each new feature of the display adds a new potential place for new technology to further enrich the experience of interactive 3-D media.

Bibliography:

1. Thomas A. Nwodoh and Stephen A. Benton, "Chidi Holographic Video System" SPIE Proceedings on Practical Holography, vol. 3956, 2000.
2. Halle, Michael, "Autostereoscopic Displays and Computer Graphics" Computer Graphics (A publication of ACM SIGGRAPH) Volume 31, Number 2, May 1997.
3. Carlos Morimoto, Dave Koons, Arnon Amir, Myron Flickner, "Pupil Detection and Tracking Using Multiple Light Sources", http://www.almaden.ibm.com/cs/blueeyes/find.html, May 2000
4. Stephen Benton, T.E. Slowe, A.B. Kropp, and S.L. Smith, "Micropolarizer-based multiple-viewer autostereoscopic display" SPIE Proceedings Volume 3639: Stereoscopic Displays and Virtual Reality Systems VI, (SPIE January 1999) paper 3639-10.
5. VRex, Inc., 85 Executive Blvd., Elmsford, NY 10523, tel: (914) 345-8877, fax: (914) 345-8772, http://www.vrex.com
6. id Software, "id Software", http://www.idsoftware.com/, May 2000
7. Jonathan Clark, "Golgotha Source Code", http://www.jitit.com/golgotha/, May 2000
8. Jorrit Tyberghein, "Crystal Space: A Free 3D Game Engine", http://crystal.linuxgames.com/, May 2000
9. Eclipse Entertainment, "Genesis 3D Home Page", http://www.genesis3d.com/, May 2000
10. Valve Software, "valve", http://www.valvesoftware.com/, May 2000
11. Mete Ciragan, "chUmbaLum sOft", http://www.swissquake.ch/chumbalum-soft/index.html, May 2000
12. Ty Matthews and Neal White III, "Wally's Home", http://home.telefragged.com/wally/, May 2000
13. Slowe, Thomas E., "People Objects: 3-D Modeling of Heads in Real-Time", MAS SM thesis, August 1998
14. Henry A. Rowley, SHumeet Baluja, and Takeo Kanade, "Neural Network-Based Face Detection," Transactions on Pattern Analysis and Machine Intelligence, Vol. 20, No. 1 (January 1998).
15. Smith, D.C., R.E. Cole, J.O. Merritt, R.L. Pepper, "Remote Operator Performance Comparing Mono and Stereo TV Displays: The Effects of Visibility, Learning and Task Factors," NOSC Tech Report 380 (Naval Ocean Systems Center, February 1979).
16. Merritt, J.O., "Often-overlooked advantages of 3-D displays," SPIE Vol. 902 _Three-Dimensional Imaging and Remote Sensing Imaging_(SPIE, Bellingham, WA, 1988) pp. 46-47.
17. Wendy Plesniak and Ravikanth Pappu, "Spatial interaction with haptic holograms" Proceedings of the IEEE International Conference on Multimedia Computing and Systems, (ICMCS'99), June 1999.