View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Calhoun, Institutional Archive of the Naval Postgraduate School

Calhoun: The NPS Institutional Archive

Theses and Dissertations Thesis Collection

2004-09

Synthetic vision visual perception for computer generated forces using the programmable graphics pipeline

Pursel, Eugene Ray

Monterey, California. Naval Postgraduate School

http://hdl.handle.net/10945/1358

Calhoun is a project of the Dudley Knox Library at NPS, furthering the precepts and goals of open government and government transparency. All information contained herein has been approved for release by the NPS Public Affairs Officer.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

http://www.nps.edu/library

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

SYNTHETIC VISION: VISUAL PERCEPTION FOR COMPUTER GENERATED FORCES USING THE PROGRAMMABLE GRAPHICS PIPELINE

by

Eugene Ray Pursel

September 2004

Thesis Advisor: Christian J. Darken
Second Reader: Joseph A. Sullivan

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** September 2004	**3. REPORT TYPE AND DATES COVERED** Master's Thesis

4. TITLE AND SUBTITLE: Synthetic Vision: Visual Perception for Computer Generated Forces Using the Programmable Graphics Pipeline	**5. FUNDING NUMBERS**
6. AUTHOR(S) **Eugene Ray Pursel**	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited	**12b. DISTRIBUTION CODE** A

13. ABSTRACT (maximum 200 words)

In visual simulations, the human must make most of her decisions based on the visual cues rendered to her display. On the other hand, synthetic forces have the luxury of basing their decisions on the data contained in the simulation's model. Line of sight calculations are often examples of the synthetic player's excess of information. Current methodologies for determining a synthetic player's line of sight to a target are generally variations of a ray-casting technique. Hiding from a synthetic player "in plain sight" by using shadow, camouflage, or by simply remaining motionless is not possible. Synthetic vision is an alternative to ray-casting. We perform multiple renders from each synthetic player's point of view and temporarily maintain those images in graphics memory. We then execute vertex and fragment shader programs to make comparisons of the stored images. All the renders and calculations are performed on the Graphics Processing Unit (GPU) and the result is returned to the synthetic player in the form of an annotated list of visible targets. Performing these target visibility calculations on the GPU gives the synthetic player a more robust spectrum of visual inputs with which to make decisions, enabling more realistic behaviors

14. SUBJECT TERMS Line of Sight, Target Detection, Pixel Buffer, Render To Texture, Artificial Intelligence, Synthetic Player, Programmable GPU, Fragment Program, General Purpose Computation Using GPU	**15. NUMBER OF PAGES** 115
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

SYNTHETIC VISION: VISUAL PERCEPTION FOR COMPUTER GENERATED FORCES USING THE PROGRAMMABLE GRAPHICS PIPELINE

Eugene Ray Pursel
Captain, United States Marine Corps
B.S., Pennsylvania State University, 1995

Submitted in painful fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN MODELING, VIRTUAL ENVIRONMENTS AND SIMULATION (MOVES)

from the

NAVAL POSTGRADUATE SCHOOL
September 2004

Author: Eugene Ray Pursel

Approved by: Christian J. Darken
Thesis Advisor

Joseph A. Sullivan
Second Reader

Rudolf P. Darken
Chair, MOVES Curriculum Committee

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

In visual simulations, the human must make most of her decisions based on the visual cues rendered to her display. On the other hand, synthetic forces have the luxury of basing their decisions on the data contained in the simulation's model. Line of sight calculations are often examples of the synthetic player's excess of information. Current methodologies for determining a synthetic player's line of sight to a target are generally variations of a ray-casting technique. Hiding from a synthetic player in plain sight by using shadow, camouflage, or by simply remaining motionless is not possible. Synthetic vision is an alternative to ray-casting. We perform multiple renders from each synthetic player's point of view and temporarily maintain those images in graphics memory. We then execute vertex and fragment shader programs to make comparisons of the stored images. All the renders and calculations are performed on the Graphics Processing Unit (GPU) and the result is returned to the synthetic player in the form of an annotated list of visible targets. Performing these target visibility calculations on the GPU gives the synthetic player a more robust spectrum of visual inputs with which to make decisions, enabling more realistic behaviors.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I. INTRODUCTION 1
  A. STATEMENT OF THE PROBLEM 1
    1. Disparity of Information Access 1
    2. Line of Sight Calculations 2
    3. Purpose of the Study 4
    4. Research Questions 4
      a. Feasibility of Implementing the Architecture Without Using the GPU 4
      b. Feasibility of Implementing the Architecture Using the GPU 5
      c. Possible Algorithms for Making Comparisons 5
  B. LIMITATIONS OF THE STUDY 5
  C. THESIS ORGANIZATION 5

II. CURRENT METHODOLOGIES 7
  A. INTRODUCTION 7
  B. BACKGROUND 7
    1. Johnson Cycle Criteria 7
    2. ACQUIRE Model 8
  C. RECENT APPLICATIONS 10
    1. Combined Arms and Support Task Force Evaluation Model (CASTFOREM) and Janus (A) Training Simulation 10
    2. Effects of Vegetation on Line of Sight 11
    3. Team Tactical Engagement System (TTES) 13
  D. SUMMARY 14

III. TECHNOLOGY REVIEW 17
  A. INTRODUCTION 17
  B. REVIEW OF THE RENDERING PIPELINE 17
    1. Background 17
    2. Vertex Processing 18
    3. Fragment Processing 19
  C. PROGRAMMABLE GRAPHICS PROCESSING UNIT (GPU) 20
    1. Background 20
    2. Considerations 21
    3. General Purpose Computing on the GPU 22
    4. Summary 23
  D. PIXEL BUFFERS (PBUFFERS) 23
    1. Definition 23
    2. OpenGL Specification 24
    3. Synthetic Vision Implementation 24

E. RENDERING TO A TEXTURE 25
1. Definition 25
2. OpenGL Specification 25
3. Synthetic Vision Implementation 26
F. SHADER PROGRAMS 27
1. Definition 27
a. Vertex Shader 27
b. Fragment Shader 28
c. General-Purpose Processing Using Shaders 29
2. Open Graphics Language Shading Language (OGLSL) 29
3. Synthetic Vision Implementation 30
G. SUMMARY 31

IV. IMPLEMENTATION 33
A. INTRODUCTION 33
B. SYNVISION 33
1. Description 33
2. Visibility Algorithms 35
a. Color-Based Visibility Algorithm 35
b. False-Color Visibility Algorithm 36
3. Architecture and Components 37
a. OpenGL 37
b. Shaders 38
c. RenderTexture 38
d. Modeling Components 39
C. DTAI: PROPOSED SCENEGRAPH IMPLEMENTATION 39
1. Description 39
2. Architecture and Components 40
a. The Open Scene Graph (OSG) 40
b. Delta3D Simulation Engine 42
3. Visibility Algorithm 43
D. SUMMARY 44

V. TESTING AND RESULTS 47
A. INTRODUCTION 47
B. TEST SYSTEM 47
C. METHODOLOGY 48
D. RESULTS 49
E. SUMMARY OF FINDINGS 50
1. Considerations 50
2. Research Questions 51
a. Feasibility of Implementing the Architecture Without Using the Programmability of the GPU 51
b. Feasibility of Implementing the Algorithms Using the GPU 51
c. Possible Algorithms for Making Comparisons 52

VI. CONCLUSIONS 53
A. INTRODUCTION 53
B. CONCLUSIONS 53
1. SynVision 53
2. dtAI 54
3. Synthetic Vision 54
C. RECOMMENDATIONS FOR FUTURE RESEARCH 55
1. Optimizations 55
a. Reduction Process 55
b. OpenGL Context Switching 55
c. SynVision System Design 56
2. Visibility Algorithms 56
3. Implementation in Constructive Simulations Augmented by Three-Dimensional Model Information 57
APPENDIX A: SYNVISION APPLICATION ALGORITHM 59
APPENDIX B: DTAI CLASSES 61
APPENDIX C: SYNVISION.CPP 67
GLOSSARY 89
LIST OF REFERENCES 93
INITIAL DISTRIBUTION LIST 97

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

Figure 1. CASTFOREM Target Detection Algorithm (From Driel95) 11
Figure 2. Visual Fields of View. (From Cham96) 13
Figure 3. Programmable Pipeline (From Rost04) 21
Figure 4. SynVision Demonstration Application 34
Figure 5. Color-Based Visibility 35
Figure 6. False Color Visiblity 36

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

Table 1. SynVision Shaders 38
Table 2. Rendering Times in Milliseconds 49
Table 3. Frame Rates and Times in Milliseconds by Algorithm 50

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

"Fast, Thorough, or Visually Stunning; choose any two.

First and foremost, I must acknowledge the infinite patience of my family. Without their support, this endeavor would not have been attempted, let alone completed. To them I owe my success in this effort just as in every other facet of my life.

I am also grateful to Dr. Chris Darken not only for his support of my vision, but also for sharing his. Dr. Darken's experience and direction helped me focus on the questions at hand and avoid major pitfalls, or pratfalls, depending on your vantage point. At the same time, he allowed me the latitude to run into and learn from the obstacles of my own design. I hope to have the pleasure of working with Chris in the future.

The Delta3D development team was instrumental in getting as far as I did in creating dtAI. Of particular note are Andrzej Kapolka and Erik Johnson who suffered my inane C++ questions and steered me in the, what seems now obvious, direction.

Finally, for Commander Joe Sullivan's computer graphics and scene graph expertise, I am thankful. Possibly more importantly, I appreciate and envy his endlessly positive spirit. CDR Sullivan knows how to make tedious tasks fun, and enjoyable ones more so.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. STATEMENT OF THE PROBLEM

1. Disparity of Information Access

In three-dimensional visual simulations, the human players' visual input is generally limited to an image rendered to a display. The display can be any device from a standard CRT monitor to a head-mounted device or a multi-screen "cave." In any case, the image displayed is almost always a two-dimensional representation of the player's field of view in the simulated world. The human must make most of her decisions based on the visual cues garnered from this image on her display.

Not only is the image two-dimensional, removing the three-dimensional cues humans normally rely on, but the field of view is also greatly reduced compared to normal vision. For example, a player whose eyes are 21 inches (51.45 cm) from a 17-inch monitor is only realizing a vertical field of view of 54.6° and 71.6°, horizontally, into the simulated world. Compared to a human's normal vertical and horizontal fields of view of approximately 90° and 190°, respectively, we see that the human player's visual inputs from the simulation are a fraction of those the player normally enjoys. This deficiency is often reduced by adding information to the player's display in the form of a heads-up display.

On the other hand, computer-generated, or 'synthetic', entities, whether friend or foe, have the luxury of basing their decisions on the entire body of data contained in the simulation's model. Without appropriate limitations, the synthetic force can potentially be omniscient. For example, a synthetic player with unlimited access to information can know the location and status of any other player, human or synthetic. Any restrictions on a synthetic player in accessing information from the model are aspects of the synthetic forces intentionally engineered by the simulation's designers. This filtering or hiding information from the synthetic force is a challenging, but necessary, requirement. The synthetic player has, by default, access to 100 percent of the model's

information. Without limitations, the synthetic player will have an excess of information and will, most likely, behave unconvincingly.

2. Line of Sight Calculations

Line of sight calculations can often be examples of a synthetic player's excess of information. Current methodologies for determining a synthetic player's line of sight to a target are generally variations of a ray-casting technique. Ray-casting involves defining a line segment between a predetermined point relative to the three-dimensional model of the synthetic player (its "eye") in the simulated world and a predetermined point on another model – the target. If the ray between those two points does not intersect any other polygon in the model, whether terrain or any other object, then the ray passes and the synthetic player is considered to have line of sight to the target [Proc04]. Using a one-dimensional point to determine the visibility of a three-dimensional target is a fairly simple, if not over-simplified, model. An underdeveloped implementation of ray-casting ineffectively filters the abundance of information available to the synthetic player.

Several methods are used to make this method more reasonable. One method, for example, is to cast multiple rays per target. The target's silhouette is simplified to a two-dimensional polygon and rays are cast from the synthetic player to each vertex of the polygon and to the polygon's center. The weighted number of passing rays are compared to some threshold. If fewer rays than the threshold pass, the target is considered to not be visible. Regardless of the method, ray casting depends on determining intersections of lines with polygons. These are well known calculations, but not trivial. Additionally, as simulations use increasingly complex rendering techniques, these ray-casting calculations also become increasingly complex.

Consider the situation where a synthetic player is on one side of a pane of glass facing a target on the other side of the pane. A ray cast between the two players will intersect the polygon that is used to model the pane of glass, indicating the target is not visible. In order to mediate this disparity, additional calculations must be made to determine if, at the point of intersection with the

ray, the polygon is opaque. (A pane of glass might be assumed to be uniformly transparent, but perhaps it is modeled with a texture of dusty corners, or a pixel shader is used to create glare or reflections.) Now, if it is opaque, the ray is indeed intersected; however, if it is not, still further calculations must be performed to determine to what degree the polygon is transparent at that point and how that effects the overall visibility calculation. Using ray-tracing becomes more difficult as scene complexity increases.

The above scenario is an example of a larger, overarching disparity– ray-casting is a mathematical solution to a perceptual problem. Target visibility is not a binary, black-and-white problem. It is a problem fraught with many shades of grey. Ray-casting uses line of sight to determine visibility, but takes no environmental effects into account. There is, generally, no consideration given to effects that make a target more or less visible as compared to her background such as contrast or camouflage. Hiding from a synthetic player "in plain sight" by using shadow, camouflage, or by simply remaining motionless is not possible. Again, the synthetic player's information is not sufficiently being filtered, and unnaturally observant behavior emerges.

Also, as a mathematical and iterative solution, ray casting is very discrete. That is, at its core, ray casting is determining the visibility of an object by sampling the visibility of a few points on that object. Consider a ray tracing algorithm that uses a computationally expensive 12 rays to determine visibility of a humanoid target. In an environment of moderate-to-dense vegetation, all of those 12 discrete points on the target's model can easily be obscured, especially if the target is a moderate distance from the observer. Consider that a threshold, say 8 of the 12 rays, for example, is normally used to determine visibility. We can see that a target is even more likely to be mistakenly considered invisible even if in stark contrast to its surroundings. In general, ray casting has the ability to not only give too much information to the synthetic player, but can also provide too little. The solution to both extremes lies in using additional, available information to determine a target's visibility.

3. Purpose of the Study

Synthetic Vision is an alternative to ray-casting in determining visibility of targets in a three-dimensional simulation. By rendering a scene for the synthetic player just as is done for human players, we more closely limit the synthetic player's amount of information to that of the human player. Instead of having 100 percent of the available information and having to pare down from there, our synthetic player begins with a limited amount of information and gathers only that additional information which is necessary. This process is somewhat analogous to the human player's limited field of view augmented with a heads-up display.

Synthetic Vision is the implementation of an architecture with which synthetic players determine visibility of possible targets. It involves performing multiple renders from each synthetic player's point of view and temporarily maintaining those images in graphics memory. Vertex and fragment shader programs are executed to make comparisons of the stored images. All the renders and calculations are performed on the Graphics Processing Unit (GPU) and the result is returned to the synthetic player in the form of an annotated list of visible targets. Performing these target visibility calculations on the GPU accounts for and is mediated by visual properties (lighting, texturing, and shading) rather than dumb rays. This gives the synthetic player a more robust spectrum of visual inputs with which to make decisions, enabling more realistic behaviors.

4. Research Questions

a. Feasibility of Implementing the Architecture Without Using the GPU

Can the algorithm be implemented using the Central Processing Unit (CPU) only? Can the synthetic player's viewpoint be rendered to graphics memory and can comparisons be made of those generated images in order to determine the visibility of targets within the player's field of view?

b. Feasibility of Implementing the Architecture Using the GPU

Assuming that the synthetic player's viewpoint can efficiently be rendered to graphics memory; can comparisons be performed on the GPU? Is there a performance gain and, if so, under what circumstances is this gain maximized?

c. Possible Algorithms for Making Comparisons

What are some possible algorithms that can be used to compare the rendered images? Can they be implemented accurately and efficiently? Will these algorithms be original, taken from computer vision theory, or hybrids?

B. LIMITATIONS OF THE STUDY

The domain of this thesis is three-dimensional virtual simulations. One or more human players participate in the simulation through an input suite and some type of display, e.g. a personal computer. Synthetic forces participate in the simulation as the human players' teammates and/or opposition. The ability of a synthetic player to determine the visibility of other players in its field of view, whether synthetic or human, is the focus of this thesis. In particular, the investigation of an architecture upon which different visibility algorithms can be implemented is the goal.

There are many factors involved in the manner in which humans detect and identify objects within their field of view. This study does not approach modeling the identification process. The development of a suite of algorithms to be used in detection is also out of the scope of this thesis. We implemented two algorithms for the purpose of determining feasibility of implementation and also to gather insights into the efficiency of an implemented algorithm.

C. THESIS ORGANIZATION

The remainder of this thesis will be organized as follows:

- Chapter II. Current Methodologies. We will look at and evaluate some current methodologies for determining detection (generally assuming line of sight). Of particular interest is the US Army's ACQUIRE model for target detection.

- Chapter III. Technology Review. Chapter III will describe the central technologies used in implementing Synthetic Vision to include Pixel Buffers, Rendering to Texture, and Shading Programs.
- Chapter IV. Implementation. The Synthetic Vision algorithms and implementation will be described in detail for both the "proof of concept" application and a proposed library.
- Chapter V. Testing and Results. Two types of testing are required to evaluate the research questions listed above: feasibility and time. These are evaluated for our two types of algorithms implemented in the demonstration application described in Chapter IV.
- Chapter VI. Conclusions. The implementation and results are briefly revisited and we discuss some of the successes and shortcomings of Synthetic Vision, in our estimation. A few recommendations for further investigation round out this chapter.

II. CURRENT METHODOLOGIES

A. INTRODUCTION

Different approaches have been taken to achieve the same goal of more realistic target detection and identification. Realistic is here defined as similarity to human target detection performance. In fact, determining whether a synthetic entity's detection behavior is similar to a human's is difficult in itself. There is simply very little data representing the multitude of variables involved in this human perception task.

What has been done in earnest is to model the detection process for electronic sensors. As described by [Johnson58], the performance capability of a sensor (in his specific case, image intensifiers) can be modeled, given a number of variables. The US Army's Night Vision Electronic Sensor Division (NVESD) used this algorithm to generate models for a number of sensors versus lighting conditions. This is the US Army's ACQUIRE model for target detection and identification in high-resolution visual simulations. NVESD went further to interpolate their model to include the unaided human eye and sunlight [NVESD2].

Versions of ACQUIRE appear in the OneSAF, JANUS, and Combat^{XXI} simulations, all of which are currently being used or developed. The ACQUIRE model is also modified and used by Champion, Fatale, and Krause to model line of sight in vegetated areas [Champ96], Lind and Driels to design a prototype line of sight algorithm for JANUS (A) [Driels95], and also Reece and Wirthlin in modeling synthetic player target detection and identification for the Team Tactical Engagement System [Reece96].

B. BACKGROUND

1. Johnson Cycle Criteria

Central to ACQUIRE is an algorithm developed by John Johnson of what were the US Army Engineer Research and Development Laboratories in Fort Belvoir, Virginia. Johnson's idea was that when using electro-optics, the output of the system was a visible image that a human observer can use for

interpretation and decision making [Johnson58]. Further, he discretized the levels of discrimination to No Detection, Detection, Shape Orientation, Shape Recognition, and Detail Recognition. Johnson's Cycle Criteria is dependent upon how many *resolvable cycles* can be determined across a function a two-dimensional projection of the presented area of a target. "This concept assumes that a target is characterized by a critical target dimension, which contains the target detail essential to discrimination." [NVESD1]. Electro-optic sensors have a finite resolution. This resolution is defined by a frequency, or number of cycles, that fit within the field of view of the sensor. This is analogous to defining the resolution of a computer monitor by the number of rows of pixels the monitor can display. The resolvable cycles mentioned above refers to the number of these finite units that span the critical dimension of the target in the sensor's field of view. This is, basically, using a function of the resolution of the optics and range-to-target to determine the possible level of detail the human observer will realize based on the size of the target's critical dimension, typically the minimum dimension, in the observer's view. Again, this is a model designed to predict performance in electro-optical sensors where resolvable cycles and other variables can be quantified by field testing.

This testing was, in fact, done by NVESD to generate criteria for the discrimination of targets of interest for the various levels of discrimination [NVESD1]. The US Army's standard Contrast Model is an example of one of these criteria. The Contrast Model, is a mix of targets and backgrounds, spectral data, and sensor, filter, and source characteristics and provides values for the Inherent Contrast variable to Johnson's algorithm [NVESD2]. While the number of permutations of this table is quite large, the resulting values are still discrete. Any variables for which data has not been collected must be extrapolated from similar known values.

2. ACQUIRE Model

The Army's current standard algorithm for modeling Search and Target Acquisition (STA) is the ACQUIRE model. ACQUIRE is an empirical model based on Johnson's Cycle Criteria used to determine target acquisition

performance for imaging systems. It is able to predict performance for three different types of tasks: target spot detection, target discrimination, and time-dependent target detection. Target spot detection is based on a signal-to-noise (contrast or temperature) ratio of a target against a uniform background. Target discrimination involves Johnson's Cycle Criteria and is used for targets against a heterogeneous or cluttered background. Finally, time-dependent target detection determines the probability of detecting a target within a given duration of time. The model was originally developed by the NVESD to be compatible with a study of the long range use of forward-looking infrared sights. Equations appropriate for direct view optics (DVO), such as binoculars or the unaided eye, were extrapolated from the original model. DVO, however, is not a recommended use of ACQUIRE [NVESD1].

Inputs to ACQUIRE fall into four categories: Target Characteristics, Environmental Effects, Sensor Characteristics, and Task Description Inputs. Many of these inputs are static values representing unchanging characteristics of either the target or the sensor. Examples of these are light level and the Sensor Characteristics. Many other inputs such as apparent signature and battlefield obscuration, though, are dynamic. These values are generally found by referring to look-up tables for known values or interpolated values for those values which are unknown. The Standard Contrast Model, described above, for finding the Target Characteristic of Inherent Contrast is an example of one look-up table. How these dynamic inputs are found and implemented separates the various implementations of ACQUIRE.

> . Ranges and probabilities predicted by the model represent the expected performance of an ensemble of trained military observers with respect to an average target having a specified signature and size. [NVESD1]

What ACQUIRE returns as outputs are a list of the portion of observers described above that successfully complete each acquisition task (spot detection, discrimination, and time-dependent detection) as a function of the observer-to-target range. For each task, the results are presented in two formats: probability

of an observer to be in the portion of observers who successfully complete the task, and the maximum range for successfully accomplishing the task at probability ranges from 0.05 to 0.95.

C. RECENT APPLICATIONS

1. Combined Arms and Support Task Force Evaluation Model (CASTFOREM) and Janus (A) Training Simulation

CASTFOREM and Janus are both entity-level models, that is they model company level units and below and are capable of modeling single entities, such as individual vehicles or infantrymen. They both use the ACQUIRE model to determine probability of detection. The two models differ, though, in their use of ACQUIRE; Janus uses constants as ACQUIRE parameters which remain unchanged over the duration of the entire simulation while CASTFOREM has more dynamic parameters. Driels and Lind investigated a method of deriving dynamic inputs for ACQUIRE and implementing them in Janus. An algorithm by Driels and Lind [Driels95] centers around ray casting over a regular grid. Their particular focus was to perform their derivations in a database driven simulation using a perspective view generator (PVG).

The PVG creates a raster of pixels representing the observer's point of view. One ray is cast per pixel from the point of view at a calculated offset from boresight until it encounters the ground, a target, or an overhang. An overhang is a feature under or behind which an object can be hidden. This ray casting results in a raster of pixels on the framebuffer. The color of each pixel is determined by the outcome of a corresponding ray cast: targets are red, open terrain is a grayscale interpolation of the relative elevation, and areas obscured by an overhanging feature are colored black. This framebuffer, once rendered to the display, is an annotated perspective view of the battlespace from the observer's point of view. It is annotated by the location of targets in red and all unobservable areas in black.

The PVG enables some of the inputs to CASTFOREM's implementation of ACQUIRE as depicted in Figure 1, such as Apparent Contrast, to be calculated dynamically, increasing the realism of the model. The ability of an entity to enter

and leave unobservable areas by were a great improvement over previous implementations of ACQUIRE. Driels and Lind's ability to incorporate this algorithm on a database driven simulation allowed them the luxury of calculation unaffected by the density of objects in their scene. The same number of rays is cast for a complex scene as a simple one; it is dependent only on the dimensions of the observer's view.

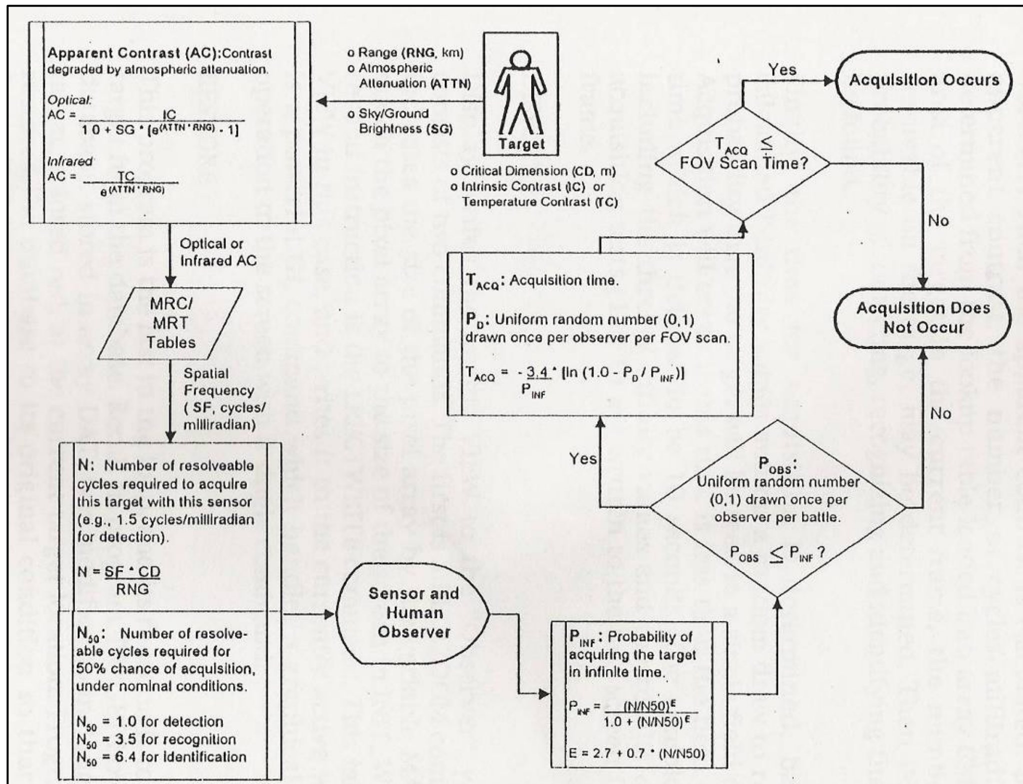


Figure 1. CASTFOREM Target Detection Algorithm (From Driel95)

2. Effects of Vegetation on Line of Sight

Again, predicting line of sight in combat simulations realistically is a key to accurate target detection. Before Champion et al, no systematic approach was taken to quantify the effects of vegetation on line of sight and implement them in combat simulations. One of the current methods of modeling these effects is to partition the battlespace into categories of surface features and have different line of sight criteria for each feature. For example, a dense forest feature may enable line of sight to extend only 3 meters into the feature while the rest (its

interior) has no line of sight from outside the feature. Champion points out the lack of quantitative data to support the current methods.

US Army Training and Doctrine Command Analysis Center – White Sands Missile Range (TRAC-WSMR) developed a study to identify a wide variety of vegetation types and to collect data within each area in order to determine percentage of target visible when LOS exists. Their goal was to develop a function that would give the percentage of a target that is visible to an observer given the surrounding vegetation type and LOS range.

This percentage coupled with the range of the LOS is used to determine the resolvable cycles as described in Johnson's Cycle Criteria. That is, the ability for the current optical sensor to determine detail [Champ96]. It is somewhat intuitive that the vegetation surrounding a target will have an effect on the amount of it an observer can see. The less intuitive, aspect of this model is that this effect is implemented by modifying the characteristics of the sensor in Johnson's algorithm.

For demonstrative purposes, Champion implemented the function as a modification to the ACQUIRE model used in CASTFOREM. His example is described in [Champ96, pp 33-36]. As usually implemented, Inherent Contrast, Sky Over Ground ratio, and Atmospheric Attenuation are all constants from a look-up table representative of the location of the scenario: in this case, Europe. His results show a significant difference between the probability of detecting soldiers kneeling and on the move and those prone and in motion at ranges from 0 to 400 meters under various light conditions [Champ96].

TRAC-WSMR's study presents an enhancement to the ACQUIRE model by introducing vegetation- and climate-specific modifiers to target detection and identification processes. Champion identifies the requirement for the inclusion of factors which are particular to a geographic region in simulation, particularly in the realm of target detection. While the factors for determining Resolvable Cycles for Johnson's Algorithm are now more easily found for dismounted infantry observed with the human eye, other possibly dynamic factors are still

described as constants. In Champion's example application, Apparent Contrast is calculated with three of its four inputs defined as representative constants.

3. Team Tactical Engagement System (TTES)

TTES is an application that was being developed for the US Marine Corps. In developing visual and aural detection and visual identification abilities for their "individual combatants (ICs)", Reece and Wirthlin did not model the eye, but modeled characteristics of human vision. In particular, they consider the difference between the highly acute 1° of foveal vision and the less acute ~95° of peripheral vision. They generalize detection in the foveal and near-foveal (30°) areas of the field of view, simulating visual search. In general, objects in the model which are mathematically determined to fall inside the fan described by this field of view are evaluated for visibility and/or identification.

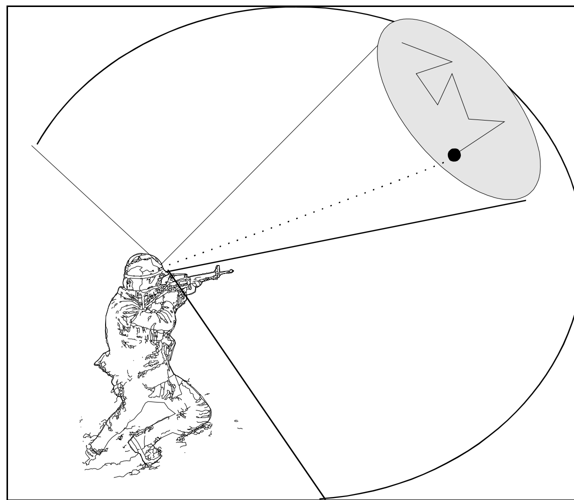


Figure 2. Visual Fields of View. (From Cham96)

> Detections in the peripheral field of view are effectively immediate; in this 30° area, however, we compute an acquisition time for detecting objects because the fovea has to search the area. This model also assumes that identification is immediate once the fovea fixates on the object. [Cham96]

The IC sighting algorithm used is based on Lind's sighting model, similar to the model used in JANUS and ModSAF. The inputs to the algorithm include the range and visibility of the target, environmental variables, attributes of the visual sensor (in this case, the modeled field of view), and a normally distributed

random number. The output of the algorithm is a sighting status: Invisible, Visible, Detected, Recognized, or Identified.

Reece acknowledges limitations in their algorithm. Several of the input variables for their sighting algorithm are constants. In particular, "default values" are used for both light level and brightness contrast with background. Other limitations of the study are outlined: effects of low light levels on acuity, effects of observer motion, and modeling specific direction of gaze during search. These factors are understandably difficult to define and quantify solely from the simulation's mathematical model. However, some of Reece's obstacles, such as "effect of contrast—brightness, color, and texture" and specific direction of gaze can be addressed by using additional information that can be gathered from the IC's rendered field of view.

D. SUMMARY

The US Army's ACQUIRE model and Johnson's Cycle Criteria, indirectly, are implemented in several current combat simulations. The effort to make this model more representative of target detection in the real world is challenging, yet ongoing. The above examples and applications of ACQUIRE have two aspects in common: they still require global or at least very discrete values for some variables and they are all in the domain of constructive simulations.

These two traits are surprisingly related. Constructive simulations are primarily mathematically based models with associated visualizations. These simulations manipulate and model single entities on the battlefield, but these single entities are simply copies of one discrete mathematical model. Accordingly, each interaction between similar opponents, with the imposed slight randomness aside, is nearly identical to every other. For example, an engagement described as one soldier engaging another soldier will be calculated similarly regardless of the particular instances of each soldier. The variables do not lie with the instances of soldiers, but in the environmental factors. These environmental factors are also very discrete with constant values given to regions of similar vegetation. A soldier in an area labeled as 'dense vegetation' engaging another soldier 'in the open' is an example of an engagement that will differ from

the one described above. However, if this engagement occurs a second time at a short distance from the first engagement, the results will, again, be nearly identical because there is no mechanism to discern between ‘this dense vegetation’ and ‘that dense vegetation.’ For the purposes of these combat simulations, this phenomenon is completely acceptable. With this in mind, it is understandable that the variables most closely coupled with the individual entity can be discrete or even constant.

This thesis experiments in the domain of virtual simulations, though. How can the evaluation of constructive simulation methods of target detection be of any interest or value? Simply, virtual simulations have no published or accepted standard for performing target detection or identification outside the creative use of ray casting as described in the Introduction. The current methodology for target detection and identification in both games and first-person simulations is some variation on ray casting. We believe that we can use information available in a first-person simulation to garner a more complete suite of inputs analogous to those required by Johnson’s algorithm. Performing calculations involving the color and depth information inherent in a scene rendered from the observer’s point of view can result in accurate and dynamic determination of target area presented and the apparent contrast. Of particular note, the literally apparent contrast takes into account combat-induced obscurants and atmospheric attenuation if these facets are present in the model.

THIS PAGE INTENTIONALLY LEFT BLANK

III. TECHNOLOGY REVIEW

A. INTRODUCTION

In order to better understand our implementations of Synthetic Vision, a basic review of the rendering pipeline is provided. Additionally, we describe some of the other technologies used in developing our algorithm. These tools either augment or circumvent certain portions of the normal rendering pipeline. This combination of methods and technologies is an experimental amalgamation and while the individual components and their implementations are fairly well known, the aggregation of these components is not.

One example is our use of the GPU. We not only render a point of view off-screen to a texture, but we also use that texture as an input to another off-screen render which performs visibility calculations. In this chapter, we will take a general look at each technology, such as the GPU. In the following chapter, we will describe our particular implementations of each.

B. REVIEW OF THE RENDERING PIPELINE

1. Background

Software applications, especially real-time simulations, perform initializations and then execute some form of a run cycle, once per frame, until the application is terminated. This run cycle is normally composed of update and draw stages. Inputs to the system are generally processed by an event-handling scheme and can be considered relatively continuous and not a discrete stage of the cycle. In describing the "rendering pipeline," we are referring to the draw stage of the run cycle. This is the process of taking data from the software application, processing that information into a geometric representation, and generating a visible image of that geometry to the user on some type of display. This process is generally implemented in today's commodity hardware by at least one of two specifications: the Open Graphics Library (OpenGL) and DirectX. OpenGL is licensed by Sililcon Graphics, Inc. and is governed by the OpenGL Architecture Review Board. "End users, independent software vendors, and others writing code based on the OpenGL API are free from licensing

requirements." [OpenGL2] DirectX is a proprietary standard that was created by Microsoft for rendering in Microsoft Windows applications. The DirectX specification includes many aspects of game development to include input devices and audio. Both specifications are complied with in nearly all the products by graphics hardware vendors, nVidia, ATI, and 3DLabs, for example.

Akenine-Möller and Haines further breaks down the rendering pipeline into three stages in [Aken02]. These are the Application, Geometry, and Rasterizer stages. [ARB99] and [Rost04] describe, in depth, the functions of the latter two stages. This brief discussion will, similarly, focus on the last two stages, Geometry and Rasterizer, from the perspective of OpenGL with emphasis on the latter.

2. Vertex Processing

The goal of the Geometry and Rasterizer stages is to take the geometric information from the application broken down into a stream of single primitives and process them for display. Primitives are defined by a set of one or more vertices. Each vertex has a color and location in object space. By specification, all primitives are closed and convex and are recommended to have coplanar vertices. Triangles are most commonly used since any group of three points are coplanar.

These geometry primitives are passed from the Application stage and the positions of their vertices are transformed from object space to eye space to clip space. If one or more vertices of a primitive are located outside the intermediate clip space, they are discarded and vertices are created at the boundary of the space to allow partial primitives to be displayed. The final transformation is to window space, described in detail in Chapter 3 of [ARB99]. The result of all these transformations is a location in window space of each 'visible' vertex of the primitive. The above transformations are appropriately referred to as "vertex processing." These locations are then passed along with each vertex's color and depth information to the Rasterizer for "fragment processing."

While discussing vertex processing, we must also introduce the idea of textures. A texture is an image which is applied to a set of geometry. A simple example is a pair of triangles used to form a rectangle over which a single texture is placed. Instead of the rectangle being a color defined by its vertices, it now has an image applied to it, like a decal stretched over the rectangle. The texture in this example is defined to span both triangles; in general, a texture can span either a single primitive or span several contiguous primitives. Prior to the primitive being defined, the texture image is stored in texture memory on the video card. The boundaries of the texture are defined by a set of texture coordinates. Each coordinate is coupled to a corresponding primitive vertex. When the vertex information is passed down the pipeline for fragment processing, the corresponding texture coordinates are also passed.

3. Fragment Processing

Since all primitives are closed and convex, an "inside" region can be interpolated from the vertices. This region is divided into discrete units. Each of these units is called a fragment which is analogous to, but not always equivalent to, a pixel. A pixel is a color sample at a particular point on an image. As described quite enthusiastically in [Smith95], "A pixel is not a small square!" Fragments, on the other hand, have a sense of area in that they are later mapped to an area on the display. For primitives without textures, the color and depth of each fragment is interpolated from the colors and depths of the vertices composing the primitive. If a primitive is textured, the color value for each fragment is found by doing a look-up on the texture image in texture memory. The location queried is based on an extrapolation of the primitive's texture coordinates. Regardless of the method of determining the color value, each one of these fragment's color and depth values are passed to the Rasterizer.

In rasterizing each fragment, several tests are performed to determine the color and visibility of each fragment. At the simplest, if the depth value of a particular fragment is less than the existing depth value at the same position (hence, closer to the eyepoint), then the fragment is considered to be visible. Its depth value is then written to the depth buffer at the appropriate position(s) and

its color value is similarly written to the frame buffer. This process is done for each fragment in a primitive and for each primitive in the scene's geometry. The result is an array of color values in the frame buffer and a corresponding array of depth values in the depth buffer. Finally, the contents of the frame buffer are mapped to the display device.

This discussion was a brief simplification of the rendering pipeline for the purpose of providing some background to the following sections. There are a great number of complexities we have glossed over. These include double-buffering, the stencil and accumulations buffers, and alpha values. Still, we now have a common, basic understanding of the rendering pipeline. From this basic model, we will now diverge. The remaining sections will discuss the augmentations and alterations to this model which are required to be understood before moving on to the discussion of implementation in the next chapter.

C. PROGRAMMABLE GRAPHICS PROCESSING UNIT (GPU)

1. Background

The GPU is not, in itself, a new technology. It is the microprocessor specifically designed to process three-dimensional graphics data. On August 31, 1999, nVidia coined the term with the unveiling of the GeForce 256 video card. While ATI later coined a term of its own, Visual Processing Unit (VPU), GPU is now used in reference to all modern graphics chips [Prank]. The processes described above are referred to, loosely, as the OpenGL fixed functionality pipeline. This behavior is prescribed by the OpenGL specification version 1.1 and is the default behavior for the OpenGL rendering pipeline. With the advent of programmable GPUs, this distinction has become significant. It is significant because the GPU's programmability allows users to alter the default behavior of the pipeline at two points, vertex processing and fragment processing, as depicted in Figure 3.

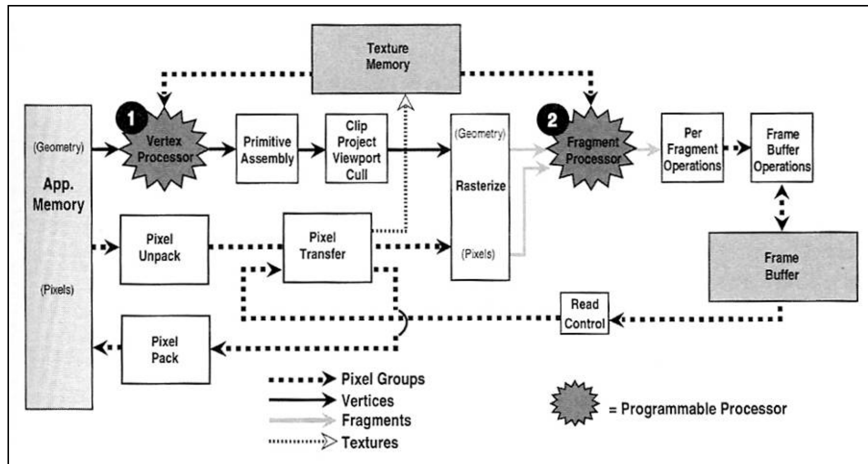


Figure 3. Programmable Pipeline (From Rost04)

The GPU is engineered for graphics processing. It is a highly parallel architecture with supporting random-access memory, texture memory, frame buffer memory, and a cache. All of this supporting architecture resides with the GPU on the graphics card. The current commodity GPU can perform, roughly, an order of magnitude more operations per second than a commodity CPU. In the CPU's defense, the GPU's operations are quite specialized and optimized. The current GPU can access its memory much quicker than CPU, also. Overall, the processing done onboard the graphics card can be done much quicker than on the CPU for many operations.

2. Considerations

A very important aspect of the GPU to consider is the idea of mandatory parallelism. As discussed previously, the input to the GPU is a stream of primitives. These primitives are broken down into vertices and then further into fragments. This is where the parallelism comes into play; the fragments are processed independently and in parallel. That is, every fragment processed has no information regarding the fragments that came before, will go after, or any of its neighboring fragments. There are no comparisons between fragments. This enables a large number of fragments to be processed in the amount of time required by the single slowest fragment.

One particular operation for which the GPU is optimized is random look-ups from texture memory. While it can not write directly to texture memory,

accesses from that memory are relatively quick. Texture data are placed in the texture memory onboard the graphics card and are used as inputs. If few enough textures are used by the application, these textures all remain 'active' and are immediately accessible. Otherwise, textures are paged into and out of texture memory from system memory.

3. General Purpose Computing on the GPU

Using the GPU to perform computations other than rendering is certainly not unique to this study. A recent workshop for discussing this very enterprise, the 2004 ACM Workshop on General-Purpose Computing on Graphics Processors (GP^2), showcased a number of current applications and studies varying from urban gas flow and dispersion to studying molecular dynamics to the work described in this thesis [GP2]. There are even application libraries upon which applications can be written to utilize the GPU with little knowledge of the rendering pipeline. Stanford University's BrookGPU is an example of one of these libraries [Brook].

Regardless of the architecture, the problem set for which the GPU is well-suited is constrained. As mentioned above, the GPU has a form of mandatory parallelism. With each piece of data being processed independently, applications requiring functions as commonplace as minimum, maximum, or even sums across a dataset are not readily programmed. One way to address these problems is to perform additional passes of the pipeline to implement any desired comparison operations in a reduction process as described by Ian Buck and Tim Purcell in [Buck04]. This is an example of the requirement of tuning an application to fit the constraints of GPU processing.

The process of general-purpose computing on the GPU begins with loading input data into texture memory. Programs are also loaded onto one or both of the programmable segments of the rendering pipeline to perform calculations on the pre-loaded data. Some geometry must be passed to the pipeline to trigger processing and the results are written to the frame buffer. (After all, the programmable portions of the graphics pipeline will not be activated if no primitives are being processed!) The final step is to retrieve the information

from the frame buffer back to the CPU and this step turns out to be surprisingly non-trivial.

The idea of passing data between system memory and the graphics card brings us to our most limiting and often most important, consideration when considering using the GPU for general-purpose computing: the data bus between the CPU and the GPU. The only distinction between the Peripheral Component Interconnect (PCI), the Accelerated Graphics Port (AGP), and the new PCI-Express (PCI-X) busses is the magnitude of the limitation. Regardless of the data bus, the exchange of data between the CPU and the graphics card will be orders of magnitude slower than the exchange between either processor and its memory. With this in mind, it is easy to imagine a scenario where using the GPU is less efficient than simply processing the data on the CPU. Generally, any time data is passed to the GPU for processing, enough calculation must be performed to sufficiently take advantage of the increased performance of the GPU in order to offset the overhead of passing inputs and outputs through the CPU-GPU data bus.

4. Summary

Neither the GPU nor performing general-purpose calculations on it are new concepts. More and more studies and applications are being developed to exploit the performance gap between GPUs and CPUs. However, despite the increased performance, the GPU is still engineered primarily for the rendering of geometric data to a display of some sort. Any attempts to use the GPU for general calculations must consider several factors. Not the least of which are the ideas of mandatory parallelism and data bus overhead.

D. PIXEL BUFFERS (PBUFFERS)

1. Definition

The frame buffer is the default destination of processed geometry to be rendered on screen. In contrast, a pixel buffer, or pbuffer, is a destination for geometry that is not destined to be displayed, at least not directly. A pbuffer is analogous to the frame buffer on many levels. Just as a frame buffer is a portion of memory on the graphics card, a pbuffer is also. In fact, pbuffers are

implemented in available frame buffer memory. Pbuffers also have multiple rendering buffers, front- and back-left, for example, a depth buffer, and often, stencil and accumulation buffers. These are, again, similar to the frame buffer. The difference is that the frame buffer can be mapped directly to a display device whereas the data in a pbuffer must first be transferred to the frame buffer in order to be displayed. Logically, since pbuffers are non-visible, they are commonly referred to as being off-screen or used for off-screen rendering.

2. OpenGL Specification

The OpenGL specification for the implementation of pbuffers is prescribed in OpenGL ARB Extension Number 11. In the specification, both syntax and semantics are described in addition to dependencies on other OpenGL extensions. Of particular note is a recommendation made multiple times in the specification: “Pbuffers should be deallocated when the program is no longer using them -- for example, if the program is iconified.” [ARB11] This follows from the fact that pbuffers take resources directly from the frame buffer. On the other hand, the pbuffer is also recommended to be treated as a relatively static entity. That is, it should not be reallocated every rendering loop.

3. Synthetic Vision Implementation

Because Synthetic Vision is envisioned to work as part of a visual simulation, any direct intrusion on the frame buffer would be visible to a user. Not only would the user’s apparent frame rate be effected, but her overall experience would be palled. Pbuffers provide us with a workspace on which we can manipulate visual information without distracting the user.

Not only do we use pbuffers as targets to render off-screen, but also as inputs to our vertex and fragment programs. Several ways exist to access information on the frame buffer and pbuffers, but almost all involve the iterative reading of groups of pixels from texture memory and copying them to system memory. These methods exacerbate the GPU-CPU data bus issue discussed earlier. The next section describes an alternative method of retrieving data from the pbuffer without the exorbitant overhead generated by copying pixels across the data bus.

E. RENDERING TO A TEXTURE

1. Definition

As discussed above, a texture is an image that can be applied to the surface of a set of primitives. In particular, the texture is stored, appropriately, in texture memory on the graphics card in order to be in proximity to the processor which will access it, probably a multitude of times. The idea of rendering to a texture is that data is passed to the graphics card, processed, and written to a pbuffer. But more importantly, that image on the pbuffer is now directly accessible to the GPU in the same manner as a texture. Without rendering to a texture, the image would be rendered normally, and then its pixels would have to be copied back to system memory through the data bus just to turn around and be passed back to the graphics card to be stored into texture memory. Rendering to a texture saves a round trip through the bus! Depending on the size of the texture, the savings of a single render to texture could be in the order of seconds, not milliseconds.

2. OpenGL Specification

The OpenGL specification for rendering to texture is prescribed in OpenGL ARB Extension Number 20. The extension has a prefix of "WGL" indicating that it is only specified for Microsoft Windows platforms. Of additional note is the dependency of this extension on the pbuffer extension described above. Several constraints must be addressed when using this extension. Of central concern are (from [ARB20]):

- Only color buffers of a pbuffer can be bound as a texture. It is not possible to use the color buffer of a window as a texture.
- When a color buffer of a pbuffer is being used as a texture, the pbuffer can not be used for rendering; this makes it easier for implementations to avoid a copy of the image since the semantics of the pointer swap are clear.

- The application must release the color buffer from the texture before it can render to the pbuffer again. When the color buffer is bound as a texture, draw and read operations on the pbuffer are undefined.

The first constraint may seem intuitive. We find, though, that rendering the depth values of a system of geometry can be useful, too. In fact, Synthetic Vision uses this idea to make depth comparisons. An additional extension enables us to render to a depth texture. It is WGL_NV_render_depth_texture [ARB263]. Again, "WGL" denotes that the extension is only specified for Microsoft Windows and the additional "NV" similarly denotes nVidia specificity. In short, ARB Extension Number 263 is build upon the ARB_render_texture extension and enables a pbuffer to be bound to a color or depth texture.

The remaining constraints lay out a rendering cycle. To perform any non-visible rendering, the pbuffer must first be enabled, any bound color texture is released, the geometry is rendered through the graphics pipeline, a color texture is bound to the newly-populated pbuffer, and finally, the pbuffer is disabled to allow the application to continue on-screen rendering.

3. Synthetic Vision Implementation

Rendering to texture is intrinsic to the Synthetic Vision algorithm. We render two images to textures and then use a shader program, described in the next section, to perform pixel-by-pixel comparisons of those two textures. Again, without the ability to render to a texture, making those two images accessible to the shader program on the GPU would be impossible to accomplish in a timely fashion. The ability to generate the images locally on the graphics card and then performing calculations on those images with the performance of the GPU is how this algorithm offsets the overhead of pushing a small amount of data across the data bus.

F. SHADER PROGRAMS

1. Definition

Traditionally, shader programs are instruction sets defined by the application which specifically effect the processing that occurs at the two programmable points in the rendering pipeline. This is done to create and apply particular effects to objects in the rendered scene. Some examples provided by [Rost04] include:

- Increasingly realistic materials
- Increasingly realistic lighting effects
- Non-photorealistic effects
- Image Processing

The two programmable points mentioned above are at the vertex processor and the fragment processor and the particular program types for each point are appropriately called vertex shaders and fragment shaders. One or more of each shader type can be compiled and linked together to form an executable. A shader program is composed of one or more of these executables and is run on the programmable vertex and fragment processing units. Syntactically, the two programs are very much alike, but they differ in their functions. In particular, if shader programs are implemented, they must perform the same functions as their OpenGL fixed functionality equivalent in addition to whatever special behavior the shader defines. The next two sections will discuss each type of shader in slightly more detail.

a. Vertex Shader

In fixed functionality, the vertex processor performs the Geometry stage functions. That is, the process performs the transformations of vertices and texture coordinates from object space to window space. Of course, much more is done, but these are the functions we discussed in some detail earlier. Again, the vertex shader program must also perform these functions. These functions are fairly straightforward and well documented, so including them in a custom shader is no great ordeal.

What the vertex shader receives as inputs are built-in variables such as color and position which are passed in from the front end of the rendering pipeline. User-defined variables and textures can also be passed into the vertex shader from the application. The shader performs the fixed-function-equivalent functions and any special functions, or one may even be incorporated in the other. Finally, the shader returns built-in variables such as position and color to the fixed functionality processes in place between the vertex and fragment processors. User-defined variables can also be output in order to pass varying data from the vertex shader to the fragment shader.

Probably most importantly, remember that the programmable vertex processor does just as the name implies – processes vertices. The vertex program's calculations are only performed once per vertex. Vertex programs, generally, lay the foundation for the per-pixel fragment program by altering the color or location of the vertex or its associated texture coordinate. The fragment processor later uses these perturbed values as a basis for whatever effect it is designed to achieve.

b. Fragment Shader

Inputs to the fragment shader come from three sources: built-in variables passed in from the fixed function pipeline, user-defined variables or textures passed in from the application, and user-defined inputs which were calculated and output from the vertex shader. Care must be taken in considering this last type of variable. These user-defined variables are not passed directly from the vertex processor to the fragment processor. Remember, the input values to the fragment processor are interpolated from the output values of all the vertices of the primitive to which the fragment belongs. This interpolation is accomplished by the section of fixed function between the two processors.

With the above described inputs, the fragment shader must, just as the vertex shader does, perform calculations to fulfill the requirement of fixed-functionality equivalence. Texture lookup is a particularly interesting function performed by the fragment shader. Normally, given a texture coordinate, the fragment shader performs a lookup on the input texture at the provided texture

coordinate and retrieves the color information. Unlike the vertex shader, the fragment shader has random read access to texture memory. That is, if a texture is passed in as in input, the shader can read the color value of any pixel in that texture.

c. General-Purpose Processing Using Shaders

Consider, again, the idea that fragment shaders can perform random look-ups of texture memory. This is the means by which sets of data can efficiently be passed to the GPU for general purpose calculations – stored on the video card in the format of a texture! If the data is correctly partitioned by texture coordinates and accurately addressed by the vertex shader, multiple independent calculations can be performed on a data set in a single rendering pass by the fragment shader. Of course, in this context, rendering pass is somewhat of a misnomer in that no rendering is really being accomplished. In fact, the fixed function equivalence that we are normally concerned with in creating visual effects with shader programs can be ignored completely. What cannot be ignored is the requirement to assign values to all special output variables for each shader. Most of these outputs will be used as designed both by the vertex and fragment shaders, so this constraint is almost trivial.

While texture memory is used to load input data to the GPU, vehicles by which to return results are limited. The possible options all revolve around the frame buffer. A common target is the stencil buffer. Another is the frame buffer itself, if it is not being used by the application for on-screen rendering. Finally, pbuffers are popular targets and our choice for returning results. The result of a shader program is then a pbuffer populated with, generally, four float values clamped from 0 to 1. In itself, a rectilinear array of floats is unusable. More manipulations, such as the reduction scheme mentioned earlier, must be performed before useful results can be gleaned.

2. Open Graphics Language Shading Language (OGLSL)

Individual shader code very much resembles assembly language. However, within the last few years, higher-level languages have been developed to make the development of shaders more resemble the workflow in developing

C or Java code. nVidia and Microsoft formed a cooperative partnership of sorts and they created “C for Graphics” (Cg) and the “High Level Shading Language” (HLSL), respectively. Meanwhile, 3DLabs developed a C-like language, and supporting OpenGL extensions encapsulating the functionality of shader programs, named OpenGL Shading Language (OGLSL).

In June 2003, 3DLab’s implementation of OGLSL and the OpenGL extensions that support it were adopted as OpenGL ARB extensions. Further, on 3 September 2004, the newest specification of OpenGL, 2.0, was officially approved. Among other features, OGLSL and its APIs were added to the core of the OpenGL specification [OpenGL]. If graphics card vendors continue to fully support the OpenGL specification as in the past, then we should soon see full OGLSL implementations on commodity graphics cards. OGLSL was chosen over Cg and HLSL for this project in anticipation of its acceptance as part of the core of OpenGL. As such, we expect that not only hardware support will continue, but the technologies described above should also continue to evolve, making Synthetic Vision a more straightforward and accessible architecture.

3. Synthetic Vision Implementation

Our implementations go to great lengths to generate and bind textures as inputs for comparison by shading programs. As it turns out, once these input textures – whole scene, target-only, and their depth textures -- are in place, many different comparison schemes are possible. Each can be implemented simply by replacing the shader which performs the comparisons. The architecture is very modular in this regard. Also possible is the layering of multiple different shaders, each will be executed once per rendering pass. We found this aspect to not be optimal for real-time simulation applications, but offer interesting investigations involving those which are not executed at real-time or near-real-time. Remember, retrieving results from the shading program is not trivial. After the comparisons are complete, Synthetic Vision implements a reduction, distilling a single result from the buffer of resultant data. This reduction is, in itself, several successive passes.

G. SUMMARY

The only way to tap into the computing power of the GPU is currently through shader programs. There are quite a few restrictions involving not only inputs and outputs, but also the access of either by the CPU. Despite these restrictions, the speed of executing these simple programs can absorb the apparent overhead required to setup the problem and retrieve the results.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. IMPLEMENTATION

A. INTRODUCTION

Many factors played roles in determining the path to implementing our algorithm. Of course, determining feasibility and the ability to reproduce our results were considerations, but we also intended to engineer the additional ability to generalize the algorithms to be able to reused in practical, real-time applications. In the end, we took two separate roads: a relatively small-scale 'proof-of-concept' application and an integrated API-like library. SynVision is the application, while dtAI is an integrated library.

B. SYNVISION

1. Description

While rendering to texture and general-purpose shaders are, separately, widely known and used, the integration of the two is much less so. Examples of each exist, but the amalgamation of these examples was the task central to creating this 'proof-of-concept' application. The particular examples and the extent to which they were used are listed below in the Architecture subsection. Something to remember is that SynVision is a visualization of an algorithm that is normally invisible. What are displayed to the user in this demo are processes that would be non-visible in a production application.

SynVision is a simple three-dimensional world. This application represents the simplest case of our visibility tests. In this world are three objects: a background object, a target object, and an obstacle object as illustrated in Figure 4. Additionally, the point of view can be moved left and right, giving the user the ability to position the point of view such that the target object can be in the open, completely obscured, or in some degree of partial obscuration. Along the bottom of the application window are four small 'mini-renders.' These are visualizations of the textures which are the inputs to and outputs from the comparison shaders. The left-most mini-render is the whole-scene texture. To its right is the target-only texture. Continuing to the right, we see the visible surface of the target against a red background and, finally, a mini-render

visualizing the false color whole scene texture. Statistics including the number of visible pixels can also be displayed in the console window. The code for SynVision is contained in Appendix C.

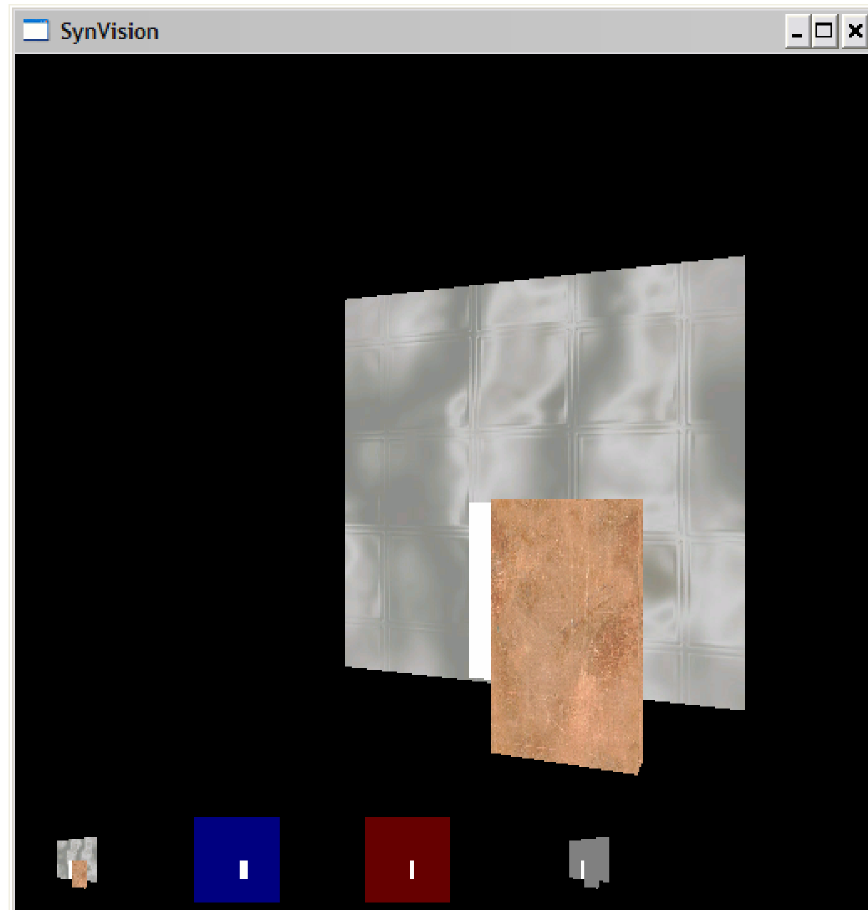


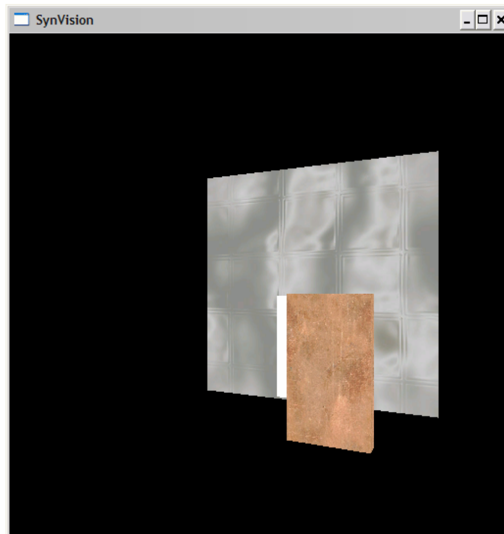
Figure 4. SynVision Demonstration Application

SynVision currently uses two separate, but related, algorithms simultaneously: color-based visibility and false-color visibility. Color-based visibility relies on comparing two textures, each in their 'natural' color, while false-color visibility makes comparisons of the colors on one prepared texture. These differences are embodied in the textures used as inputs to the comparison shaders. The algorithms for implementing the comparisons in the main application are provided in Appendix A, while the specific visibility algorithms are described in the next subsection.

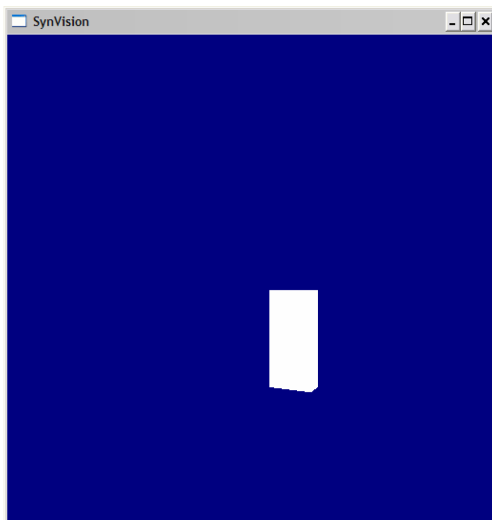
2. Visibility Algorithms

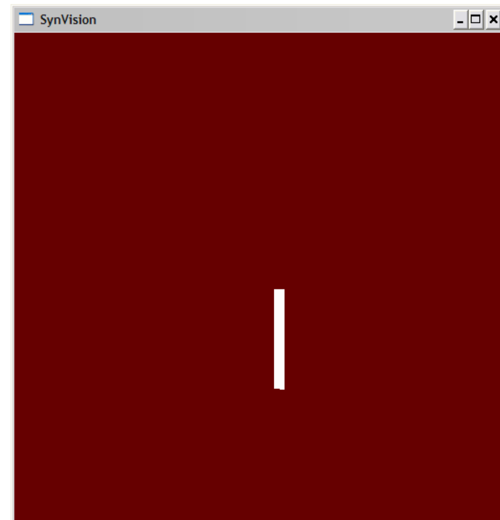
a. Color-Based Visibility Algorithm

The primary input to this algorithm is the texture representing the target alone against a clear background from the observer's viewpoint. Another input is a texture of the whole scene just as it would be seen from the same viewpoint. These are represented by panels a and b, respectively, of Figure 5.



a. Whole Scene





b. Target-Only

c. Visible Surface

Figure 5. Color-Based Visibility

Each pixel in the primary texture (b) that is not the background color is compared to the pixel at the same location in the whole scene texture (a). If the colors are the same, then that pixel is considered to be visible and it is written to the visible surface texture (c). Otherwise, it is not. Panel c of Figure 5 illustrates the result of this comparison.

The color-based visibility scheme is simply a binary algorithm and is not sensitive to color mutations such as the effects of smoke or partially transparent surfaces, for instance, those used to model screen doors. Visible pixel counts can also be erroneously high if the target is obscured by similarly-colored object.

b. False-Color Visibility Algorithm

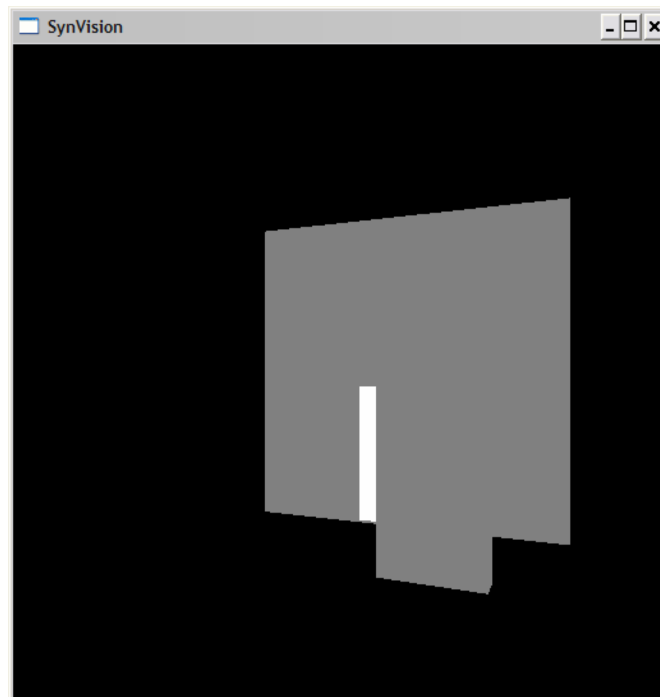


Figure 6. False Color Visiblity

The False-Color algorithm is slightly more complex to set up than its Color-Based equivalent, but simpler to make actual comparisons. It is an adaptation of an occlusion culling algorithm presented in [Klim03]. When rendering the whole scene, a shader is used to create a flat scene composed of three colors: background color, target color, and non-target color. What is

meant by “flat scene” is one which has no apparent depth; everything in the scene is drawn as a monochromatic silhouette with no details or shading. This results in a two-dimensional image in which all pixels composing the target are a single color, and everything else is another. Figure 6 shows the results of a color scheme where the target is white, the background is black, and all other objects are gray.

The comparison shader has a relatively simple job, then – given the color assigned to all target pixels (white, in the above Figure) as an input, compare every pixel’s color to that input color. If the pixel is ‘target-colored’, then it is considered to be visible. Since all models’ color information is discarded in encoding with the false colors, this algorithm is also not sensitive to semi-opaque surfaces.

3. Architecture and Components

a. OpenGL

SynVision is, at its heart, an OpenGL application. The run cycle is implemented using a standard library, GL Utility Toolkit (GLUT). This cycle is essentially the same loop as discussed above in the Chapter III. GLUT manages the execution of this loop by requiring the application to register functions with each stage of the cycle. For example, `void display(void)` is the signature of the routine that is called at the ‘draw’ stage of each frame. GLUT also encapsulates the creation and modification of windows in which to render.

In using OpenGL, extensions are often used to access functionalities of a graphics card that are not available widely enough to be accepted as core OpenGL components. While these extensions are not a part of the core of OpenGL, they are still standards with specified behaviors and syntax. For example, SynVision must use the Microsoft Windows-only extensions WGL_ARB_PBUFFER and WGL_ARB_RENDER_TEXTURE, to name a few. To manage these extensions and to be able to test if they are available at run-time, the GL Extension Wrangler (GLEW) library is used. While OpenGL 2.0 has recently been released, this application was developed on OpenGL Version 1.5.

b. Shaders

An overview of the OpenGL Shader Language is given above in Chapter III.F.2. It is used in SynVision to implement the comparison and reduction functions as described briefly in Table 1. Our methods of reading shader source, creating instances of shader programs, and linking the fragment shaders to those programs are based loosely on an example from [Rost04], named ogl2brick. The source code for this example can be found at [RostWeb], the companion website to [Rost04].

Name	Scheme	Function
boolvis.frag	Color-Based	Crop all pixels not belonging to Target
colorbasedvis.frag	Color-Based	Encode pixels as visible or not visible
boolvis2.frag	False Color	Color pixels as visible or not visible
sumreduce.frag	Reduction	Perform Sum over entire pbuffer

Table 1. SynVision Shaders

Both, the visibility comparison and the reduction shaders are fairly simple programs. Through the methods described above, the programs are read, created, linked, and compiled. While SynVision currently has two sets of shaders, this configuration is not immutable. Any interesting visibility comparison scheme can be implemented by a new shader if the required inputs can be bound to that shader. This new shader can complement or replace the existing shaders. In short, the shaders are very modular. Normally, with small infrastructure modifications to bind the correct inputs, comparison schemes in the form of shader programs can be easily swapped.

c. RenderTexture

RenderTexture.cpp and its associated header file, RenderTexture.h, are integral to SynVision. RenderTexture is a Component that encapsulates the render to texture process to include the instantiation of a pbuffer and the associated textures to be bound to the pbuffer, to include depth

textures. RenderTexture allows us way to implement the multiple renders to textures required by our algorithm a relatively intuitive way. It provides us a means to, first, instantiate a pbuffer and its associated textures, and then to enable and disable the render-to-texture process. Additionally, methods are provided to customize the pbuffer and textures and to query the states of those objects at run-time.

RenderTexture was originally created and is copyrighted by Mark Harris; but, it is openly licensed with a few caveats. The details of the licensing can be found in the header of either rendertexture.cpp or rendertexture.h. RenderTexture and Harris' other real-time graphics research can be found at [HarrisWeb].

d. Modeling Components

The models used in SynVision were created using MilkShape 3D. Several files are required to load and draw these models. These files are:

- MilkshapeModel.h
- MilkshapeModel.cpp
- Model.h
- Model.cpp
- ModelUtil.cpp

All but the last file, ModeUtil.cpp, are from NeHe Tutorial Number 31 [NeHe31]. They were created and copyrighted by Brett Porter. The last file is a utility file that loads a texture from a file into texture memory. Together, these files make the conversion from a MilkShape 3D file to a set of primitives and their associated textures.

C. DTAI: PROPOSED SCENEGRAPH IMPLEMENTATION

1. Description

Our desired path to reusability and generality involves implementing the Synthetic Vision algorithms in a simulation architecture. Currently at the Naval Postgraduate School's MOVES Institute, a simulation engine is being developed. We will discuss Delta3D in more detail below in subsection 2.b. dtAI (pronounced "delta AI") is a proposed component of the Delta3D engine. (Components of Delta3D have 'dt' suffixes, hence the AI component is dtAI)

While the implementation of Synthetic Vision was the impetus of dtAI, it turned out to be only one facet. dtAI can be considered to be the fledgling Artificial Intelligence component of Delta3D. This component is desired to encapsulate the required functionality of computer generated forces, whether teammates or opposing forces and to our knowledge, it will be one of very few open sourced artificial intelligence APIs.

In order to implement Synthetic Vision as part of dtAI, a number of architectural components were first developed. These components include a representation of the synthetic player, AIObject, and a manager of these players, AIObjectManager. Additionally, implementations of a pbuffer, and mechanisms for rendering to texture and implementing shader programs were also put into place. In dtAI, these components are PBuffer, RenderToTextureStage, and ViewInterpreter, respectively.

Currently, all of these components above are implemented in Delta3D and stand as a foundation for further development of both a usable artificial intelligence API and a test bed for research, such as Synthetic Vision itself. In the next section, we describe dtAI's underlying architectures. Appendix B describes, in detail, the components of dtAI.

2. Architecture and Components

a. The Open Scene Graph (OSG)

i. Description. The Open Scene Graph is a cross-platform, open source application toolkit written in C++ and OpenGL [OSG1]. It is widely supported by a public community and is actively developed. It started in 1998 as a project by Don Burns. In 1999, Robert Osfield joined Burns in developing the OSG. That same year, the source code was open sourced and Robert took on the role of project lead. Burns and Osfield continue to drive the development of the OSG and also support users of the OSG through their private companies. [OSG2] The OSG is open source, licensed with the OpenSceneGraph Public License which is based on the GNU Library Public License.

ii. Scene Graph. To understand the OSG, we must first discuss *scene graphs*. Simply, a scene graph is a tree of nodes. Each node represents an object in the scene. Each node also has attributes and these attributes are specific to each type of node. Some examples of nodes are drawable objects, transformable (movable) objects, and lights. The notion of a tree in the sense of data structures also implies that a node can be the parent of child nodes. In the case of transformables, when moving the parent node to a new position, the child nodes are, normally, also moved based on some offset relative to the parent. This is an example of one of the simpler relationships present in a scene graph. The idea of the scene graph begins simply, but as complexity and robustness are added and further compounded, the structure can quickly become unwieldy. This is where the OSG comes in – the OSG defines the types of nodes, manages the relationships, and most importantly, maintains structural and syntactic standards throughout the scene graph.

One reason that the scene graph is organized as a tree is that an operation central to the scene graph is a *traversal*. The OSG's run cycle is significantly more complex than that of GLUT. Understanding traversals is a key to understanding the OSG's run cycle. Instead of a simple update-and-draw cycle, the OSG performs update, cull, and draw traversals. On each traversal, if a node is flagged as having work to be done during that traversal, the work is executed. For example, some nodes on the scene graph are not drawable and so nothing is done by that node during the draw traversal.

The way a node flags that it has work to be done during a traversal is through *callbacks*. If a node desires to execute some process during each cull traversal, for example, it creates a cull traversal callback which will execute that process once per cull traversal. There are usually also pre- and post- traversal callbacks which are executed, appropriately, before and after the simple callback.

iii. Dependencies. The OSG is principally built upon two other libraries: OpenThreads and Open Producer. OpenThreads is an open source library designed to manage multi-threaded applications and Open

Producer is a programming interface which encapsulates the actual rendering of the information represented by the scene graph. Open Producer operates on the analogy of a camera complete with lens (point-of-view frustum) and film (render surface). Open Producer was also developed and is currently maintained by Don Burns' consulting business.

iv. Summary. The OSG is a robust, open source graphics library composed of a scene graph coupled with the rendering capabilities of Open Producer. It is portable, free, and still actively developed. Traversals are central to performing operations over the entire scene graph and callbacks are the methods by which work is done by a node during each traversal.

b. Delta3D Simulation Engine

i. Description. The Modeling, Virtual Environments, and Simulation (MOVES) Institute is located at the Naval Postgraduate School in Monterey, California. One working group at the Institute is developing a portable, high-performance simulation engine [Delta]. The project, managed by Erik Johnson, is on a course to engineer a library with which high-fidelity, highly-immersive simulation applications can be designed and implemented relatively quickly. This library brings together numerous input devices from mouse-and-keyboard to inertial trackers and output devices such as head-mounted displays and CAVE projection systems. The final product is an applications library tailored to networked simulation.

ii. Architecture. The OSG is at the heart of Delta3D. It is the model of any application written using Delta3D. OSG, at its core, is a data structure and a means to display that data. It is not directly designed to referee physical collisions or perform character animations, for example. So Delta3D integrates the OSG with a few other libraries such as Open Dynamics Engine (ODE), Character Animation Library (CAL3D), and others. With these functional components integrated into a single application programming interface, Delta3D has the potential to be a robust applications development library freely available via the GNU Library Public License [Delta].

iii. Summary. Delta3D is a portable, open source simulation engine. dtAI is proposed as a component to it. As a component, dtAI is more a complement to Delta3D than a user of that API. Except at the topmost levels, dtAI is primarily a user of the OSG. In other words, Synthetic Vision is a functionality developed using the OSG and is encapsulated by dtAI for use with Delta3D.

3. Visibility Algorithm

The algorithm for determining visibility in dtAI is an extension of the Color-Based visibility algorithm discussed in section B.2.a. It adds the ability to perform depth comparisons on each pixel as well as color. Armed with this additional information, we can partially address the weaknesses mentioned in both tested visibility algorithms. That is, we can now make assumptions concerning situations which affect a color change in the target object, like shadowing, by comparing both the depth and color of the pixel in the whole scene compared to the pixel at the same position in the target-only scene. In particular, if the depth values are the same but the color values are different, then the surface color of the target has changed. The specific algorithm is proposed as follows:

For each pixel comprising the target in the Target-Only image:

- If the depth of this pixel in the Target-Only image is the same as the pixel at this location in the Whole-Scene image, then:
    - If the color of this pixel is the same in both images then:
        - Label this pixel as ‘Visible’
    - Else, the color is different from one image to the other. Label this pixel as ‘Hazy’ and encode the differences in color.
- Else, a pixel of another object is obscuring the target pixel. If the alpha value is not 1.0, signifying semi-transparence, then:
    - Label this pixel as ‘Hazy’ and encode the differences in color.
- Else, label this pixel as ‘Not Visible’

This additional functionality comes at a price. The price is relatively small in computational terms, but much larger in an architectural sense. This nearly doubles the complexity of the pbuffer to add a second texture and the methods to initialize, bind, release, and destroy them. There is also an increase in the number of OpenGL extensions required, further limiting the domain of computers able to execute Synthetic Vision. Currently, with the use of pbuffers and render-to-texture, Microsoft Windows is the required platform. With the addition of rendering to depth texture, these platforms are additionally reduced to those which support the ARB_NV_RENDER_DEPTH_TEXTURE extension. As an _NV_ extension, only nVidia graphics cards can be assumed to support this extension. Once implemented, an investigation should be made to determine if the ability to detect color changes allay the additional overhead and platform limitations.

D. SUMMARY

SynVision is a demonstration application. It is the visualization of what is designed to be computed off-screen, invisible to the user. More importantly, it is the "proof of concept" that pbuffers can be used as inputs to comparison functions in the form of OGLSL shader programs, enabling execution of Synthetic Vision to be performed by the GPU and its associated memory. Two different visibility algorithms are implemented in SynVision: Color-Based and False Color comparison schemes. By using the two different algorithms, a convenient modularity was exposed. It is simple to use either one or both of the algorithms by simply disabling small sections of code. SynVision could easily be modified to allow the enabling and disabling of one or both algorithms at run-time.

dtAI is a proposed component of the MOVES Institute's Delta3D Simulation Engine. Much work was done towards reaching this goal, but further work must be completed before dtAI, and thus a completely integrated implementation of Synthetic Vision, can be realized. dtAI can currently be compiled against Delta3D, dated 14 September as downloaded from Delta3D's CVS server. As dtAI is continued to be developed and eventually integrated into Delta3D, the latest version will be available online at

http://www.cupertinist.net/dtAI/index.html. Current issues, bugs, and recommendations for investigation will also be available.

THIS PAGE INTENTIONALLY LEFT BLANK

V. TESTING AND RESULTS

A. INTRODUCTION

As discussed in the Introduction in Chapter 1, determining the visibility of a target within a synthetic player's field of view is currently achieved by determining if a line-of-sight exists between the player and the target. More specifically, line of sight is normally determined by some implementation of ray casting. The principle reason this method is used is that the algorithms are well known, and more importantly, they are well known to execute quickly. Any proposed alternative to casting a ray in determining a target's visibility must also execute quickly enough to not decrease the performance of the overall application.

The demonstration application, SynVision, was used to determine the timing efficiency of the visibility algorithms. The successful implementation of SynVision, itself, answers the basic question of whether the algorithms can be implemented using the GPU. With that in mind, we concentrated on testing the more quantifiable aspect – algorithm execution times. In building the demonstration, we also delved briefly into the realm of visibility algorithms. As described in the last chapter in detail, we successfully implemented two separate visibility schemes. In developing those schemes, we gained some insight into some basic strengths and limitations of our architecture.

The next section briefly describes our test platform. The subsequent sections will outline the testing process that was used to determine the timing performance of each algorithm and the overall application. Finally, we will summarize the findings of these tests by pairing our results with the research questions posed in Chapter 1.

B. TEST SYSTEM

SynVision was tested on a laptop computer equipped with a 3.06GHz Intel Pentium 4 processor, 512MB of DDR RAM, and an nVidia FX Go5600 video chipset with 128MB of dedicated RAM. Iwas written, debugged, and tested in the integrated development environment, Microsoft Visual Studio 2003.

C. METHODOLOGY

The Windows.h API was used to calculate the execution times. QueryPerformanceCounter(), coupled with QueryPerformanceFrequency() gave a much finer time resolution than the clock() or getTime() methods included in time.h. This resolution was 1/3059250000, or 3.2688×10^{-10}, seconds as compared to clock()'s 1/1000 second resolution.

To time a section of code, the system time was first queried immediately before the code was executed. This pre-time was assumed to have no significant overhead associated. Another system time was queried at the completion of the section of code. The time associated with making this query is considered to be overhead and is calculated once per frame. The overhead was, on average, 3.25×10^{-7} seconds. The resulting time is the difference of the pre- and post-times minus the overhead. Determining the frame time, or the amount of time required to perform one complete run cycle, was found by querying the time immediately prior to executing the run cycle and then once per frame, at the completion of the draw cycle. This resulting time was, again, the difference between the "current" time and "previous" times minus the overhead value. For all tests, the calculated times per frame were accumulated for seven minutes and averaged over the number of frames which were executed in those seven minutes.

The results of a single timing calculation came in the form of '"ticks." A tick is defined differently for each computer. For our testbed, a tick was consistently 1/3059250000 seconds. We say consistently because,gg for some laptops and handheld devices which scale processing in order to manage power consumption, the frequency of the ticks can be dynamic. Our test platform did not vary the tick frequency.

To generate average times, we had to accumulate these single timing calculations. We did this by converting the single timing to seconds and adding it to the current count. We also accumulated the number of frames which had

occurred. To find the average at the test's end, the accumulated time was divided by the number of frames. This result was now in seconds.

D. RESULTS

In executing the time testing, we found that the times to perform the renders to texture were far overshadowed by the time required to perform the summing reductions of each visibility algorithm. For the sake of comparison, the rendering times are listed in Table 2. Each render is listed with the algorithm in which it is employed and its associated execution time. Of note, the large disparity between the Color-Based and False Color Whole Scene renders can be attributed to the fact that the False Color render is done by executing "flat drawing." That is, no lighting effects or textures are applied to the models, every pixel in the model is simply drawn a single color. Similarly, the Figure-Only render is a single model against a clear background, but took relatively long to render due to the application of lighting and textures.

Render	Algorithm	Time in Milliseconds
Whole Scene	Color-Based	1.14740
Figure-Only	Color-Based	0.69530
Visible Surface Only	Color-Based	0.61708
Encoding	Color-Based	0.55607
Whole Scene	False Color	0.45748
Encoding	False Color	0.67410

Table 2. Rendering Times in Milliseconds

For each algorithm, two times were taken: the frame time and the time to distill the results, whether by counting or reducing and summing. Table 3 provides these results. Frame Time is the time required for the main application to complete a single run cycle, or frame. Reduction Time is that amount of time required to perform either the summing reduction, or counting, of the comparison

shader's resultant buffer. While Buffer Count is not an algorithm, it is included in the results for comparison. The Frame Time associated with Buffer Count is actually the time required to execute the False Color algorithm and instead of performing a summing reduction on the results from the encoding shader, it simply iterates through all the pixels in those same results and counts the number of visible pixels. The time to perform just this iteration is Counting Time.

Algorithm	Component	Time in mSec	Frames/Second
Color-Based	Frame Time	23.4692	42
	Reduction Time	21.3793	
False Color	Frame Time	18.5348	53
	Reduction Time	17.1425	
Buffer Count (False Color)	Frame Time	21.4497	46
	Counting Time	20.4674	

Table 3. Frame Rates and Times in Milliseconds by Algorithm

E. SUMMARY OF FINDINGS

1. Considerations

First and foremost, while SynVision models a very simple three-dimensional world, it is not an optimized application. It is, for the study's benefit, very straightforward and iterative. Secondly, testing of the algorithms was performed using SynVision in the debugging mode of the IDE. Hence, the application was not stripped and optimized by the compiler. Both of these facts point to results that should be considered fairly conservative.

The times that were found in our tests reflect results in applying the algorithms to a single window resolution of 512x512 pixels. This resolution maps directly to the size of the pbuffers used. Changing resolutions directly affects the amount of the finite frame buffer resources available. Perhaps more importantly, the size of the pbuffers has an immediate impact on the reductions used to sum

the shader results. Comparing resolutions of 512x512 and 1024x1024, only one more pass – eight versus seven – is required to complete a reduction of the larger pbuffer, while the number of iterations in a counting scheme would increase by a factor of four – 1048576 versus 262144. We did not test this scenario, but it is easy to conjecture that the closeness of times between counting and executing a reduction at 512x512 would not exist at the larger resolution. As we implied earlier, the results of the time testing should be considered somewhat conservative.

2. Research Questions

a. Feasibility of Implementing the Architecture Without Using the Programmability of the GPU

If we assume that pbuffers and rendering to texture are supported by the video card, then a visibility test centered on the false color algorithm can be implemented. The Buffer Count entries in Table 2 are results of that very implementation. The color-based algorithm can, in theory, also be implemented. Textures are, after all, simply multidimensional arrays. The comparisons could be painstakingly done between two textures, and the encoding would involve writing to a third, very large, array to create the resultant texture. Neither of these requirements seems able to be done in a computationally-efficient manner, so no attempt to implement this without using shaders was attempted. Remember, an alternative to ray casting must offer an increase of performance or accuracy or, as our intention, both.

b. Feasibility of Implementing the Algorithms Using the GPU

The demonstration application, SynVision, attests to the feasibility of implementing these algorithms using the programmability of the GPU. While not optimal, it still provides the number of visible pixels for a target at real-time speeds. The successful implementation of both the color-based and the false color algorithms in SynVision is encouraging. Many avenues are now open to further research from developing more visibility algorithms to optimizing the architecture. Our recommendations for future work are in the final chapter.

c. Possible Algorithms for Making Comparisons

Color-Based and False Color are the two visibility algorithms implemented by SynVision. A third algorithm proposed for use in dtAI adds the depth information to mitigate the effects of obscurations. The development of additional algorithms is an on-going challenge. This study did not broach the subject of computer vision and the numerous, well-known and implemented algorithms such as color or texture histograms. Initial consideration points to investigating those computer vision algorithms that use, as inputs, values that can be taken directly from the graphics pipeline such as depth, color, and opacity.

At first look, the linchpin in determining the efficiency of these visibility algorithms seems to be the number of reductions that are required. Optimally, this reduction operation is only done once to gather the final results. Perhaps refinement of the reduction process's implementation will lessen the strength of that statement, but investigations in that direction are left for future endeavors.

VI. CONCLUSIONS

A. INTRODUCTION

Measuring the ability to detect an object in a three-dimensional simulation by determining whether an observer has line of sight to a target is reasonable, even intuitive. The problem lies in implementing a way to determining if line of sight exists. The current methodologies to evaluate line of sight are, generally, ray casting schemes. In three-dimensional worlds, ray casting is a geometric solution to a perceptual problem. In implementing ray casting, either too much or too little information can be given to the synthetic player.

In this study, we proposed an alternative to ray casting, Synthetic Vision. By using the information inherent to a rendered scene, a target's visibility can be determined. Further, the access and computing power of the programmable graphics processing unit can be leveraged to execute this scheme.

Two implementations of Synthetic Vision were intended: a demonstrative visualization, SynVision, and a library to be added to the Detla3D simulation engine, dtAI. SynVision was created and was used to test two separate detection algorithms. dtAI has not been fully developed; but, has been designed as outlined in Appendix B. Following are the conclusions drawn from these components and some recommendations for future work.

B. CONCLUSIONS

1. SynVision

SynVision demonstrates that employing the GPU in determining the visibility of targets in a synthetic player's field of view is certainly possible. We suggested that different visibility detection algorithms should be able to be used. Two different algorithms were implemented: Color-Based Comparison and False Color Comparison. SynVision was also used to employ the False Color algorithm without using shaders. This implementation was about 15 percent slower than the same algorithm using shaders.

Not only does SynVision confirm the ability to implement Synthetic Vision, but it also helped to realize the variability in the different possible visibility comparison algorithms. While the frame rates and rendering times of both implementations of SynVision can be considered real time (>30 frames per second), the False Color algorithm was consistently quicker than its Color-Based equivalent. On the topic of frame rates, SynVision's performance is encouraging, especially considering it is an application whose strengths lie in visualization, not speed.

2. dtAI

A guiding goal of this study was to have an implementation of Synthetic Vision that could be used in a real-time, three-dimensional simulation. A library to be used as part of the Delta3D Simulation Engine, dtAI, is our bid towards that end. This vision was not fully realized. This is due primarily to an underestimation of the integration effort required to implement Synthetic Vision in Delta3D's scene graph, the Open Scene Graph. The OSG has facilities for using shaders and also for off-screen rendering, but both are engineered to eventually be displayed to the user. To derail that process required much more infrastructure than we originally anticipated.

Although the implementation was not completed, its development is on-going. With the encouraging success of SynVision, we still expect the integration of Synthetic Vision into the OSG to be a useful addition to Delta3D. With this successful integration will come a whole new level of opportunity to develop and test real-time visibility algorithms. As mentioned above, progress towards this goal will be documented at http://www.cupertinist.net/dtAI.

3. Synthetic Vision

Garnering visibility information from images created by rendering a scene from a synthetic player's view point and using that information to determine whether that player should 'see' an object within its field of view is Synthetic Vision. It has been realized using commodity graphics hardware and has been shown to execute at real or, at least, near-real time. While the performance of

SynVision is promising, overall performance of Synthetic Vision will depend on both the implementation and the visibility algorithm utilized.

C. RECOMMENDATIONS FOR FUTURE RESEARCH

1. Optimizations

a. Reduction Process

A single value must be distilled from the buffer of values resulting from the execution of a shader. In SynVision, we implemented a reduction strategy which took a 512x512 buffer and quartered that area each pass until the area was 2x2. We then summed the elements of that 2x2 array. We also investigated a brute-force counting scheme which would iterate through the 262144 elements of the resultant buffer. At our test resolution, the counting scheme was only about 15 percent slower than the reduction strategy.

We hypothesize that a combination of reduction and counting would outperform either scheme executed separately. For example, instead of reducing the 512x512 to 2x2 and summing those elements, perhaps we reduce to 64x64 and count those 4096 elements iteratively. We expect that for each initial resolution, there is an optimal mix of reduction and counting to perform this distillation.

b. OpenGL Context Switching

One of the most computationally expensive operations in the OpenGL API is a context switch. OpenGL is a state-based system. A context is a complete set of OpenGL state attributes. Switching these contexts requires the unloading of one complete state and loading another. Context switching is a factor in our algorithm because each pbuffer has an individual context. We use multiple pbuffers and each time we either enable or disable a pbuffer, a context switch occurs. This equates to multiple context switches per frame. Further research into the minimizing of this context switching would directly affect the performance of Synthetic Vision. In particular, investigation of context sharing, as in wglShareLists(), or a reuse scheme are reasonable initial directions.

c. SynVision System Design

As mentioned earlier, SynVision is not currently optimized for speed. It was simply a workspace for assembling the architectures required to visualize Synthetic Vision. While originally a "proof of concept" application, SynVision was found to be useful for visualizing the various visibility algorithms. Optimizing SynVision towards this task of visualization will result in the creation of an aid in the development of visibility algorithms.

One proposed method to realize a performance gain in SynVision involves OpenGL optimization. In addition to reducing the number of context switches as described above, minimizing the number of state changes is a standard means of optimizing OpenGL performance.

Another performance gain could be realized by SynVision by refining the architecture of the reduction method. In particular, SynVision uses the frame buffer and a copy to texture to execute the reduction. By using two pbuffers which share a graphics context, the computationally expensive copy to texture can be avoided.

2. Visibility Algorithms

Two visibility algorithms were investigated using SynVision: Color-Based and False Color. We used these two in order to validate the demonstration by calculating the number of visible pixels by two different methods and comparing the results. Despite the difference in the algorithms, the results matched consistently. Many algorithms already exist in the field of Computer Vision to make comparisons of regions of an image. We mentioned that Synthetic Vision addresses problems somewhat reversed from those in Computer Vision, but there seems to be a sizeable overlap in the algorithms that can be used to solve both. An optimized SynVision could be a good platform for developing and testing these algorithms and a functional dtAI would certainly be a fair test bed. In either case, we expect many algorithms for determining visibility can be taken from Computer Vision and other fields and implemented in Synthetic Vision.

3. Implementation in Constructive Simulations Augmented by Three-Dimensional Model Information

This concept is the furthest from fruition and would require a great shift in the way constructive simulation are built and marketed. In general, constructive simulations are composed of mathematical models, to include the terrain. The interface is often a top-down, bird's eye view of the battlespace overlaid onto a map or some other two-dimensional representation of the terrain. Line of sight is just as important in these simulations as they are in virtual, three-dimensional simulations. Targeting, engagements, and communications are examples of processes heavily reliant on line of sight.

If, in addition to the mathematical models of each object on the battlefield, there existed graphical models, Synthetic Vision could be employed to determine line of sight reliably, accurately, and off the already busy CPU. The attributes of these graphical models such as orientation and position are already implemented in the constructive simulation. A quick off-screen render could be done from the observer to the target to determine the visibility of the target. Sun and moon position would be directly modeled, as would the target's posture. Instead of using constants interpolated from tables to find the single value representing the visibility of all soldiers kneeling in the open with overhead sunlight, a run through Synthetic Vision would give a dynamic result. That result would be appropriately different from the result of evaluating the same target from a nearby observer's vantage point.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A: SYNVISION APPLICATION ALGORITHM

```
// Initialization
Initialize Application Window
Ensure OGLSL Extensions are Supported by Hardware
Initialize and Position Camera Representing the Point of View
Initialize the Models: Background, Obstacle, and Target
Initialize Pbuffers: Whole Scene, Target, Visualization, and Comparison
    (Includes Initializing Textures to be Bound to Pbuffers)
Initialize Shaders:  Visualization, Comparison, and Reduction

Loop Until Termination:

        // Render entire scene to the Whole Scene Texture
        Enable Whole Scene Pbuffer
        Draw All Models
        Bind Whole Scene Texture to Pbuffer
        Disable Whole Scene Pbuffer

        // Render Target against a cleared background to the Target Texture
        Enable Target Pbuffer
        Draw Target Model
        Bind Target Texture to Pbuffer
        Disable Target Pbuffer

        // Render an image of the visible portion of the Target against
        //        a clear background to the Visualization Texture
        Enable Visualization Pbuffer
        Set Whole Scene Texture as an Input to Visualization Shader
        Set Target Texture as an Input to Visualization Shader
        Set Additional Variables as Inputs as Required
        Enable Visualization Shader (Algorithm Described Below)
                Draw a Single Quad Spanning the Entire Viewport
        Disable Shader
        Bind Visualization Texture to Pbuffer
        Disable Visualization Pbuffer
```

```
        // Generate a Boolean map of visible and non-visible Target pixels
        Enable Comparison Pbuffer
        Set Whole Scene Texture as an Input to Comparison Shader
        Set Target Texture as an Input to Comparison Shader
        Set Additional Variables as Inputs as Required
        Enable Comparison Shader
                Draw a Single Quad Spanning the Entire Viewport
        Disable Shader
        Perform Summing Reduction on Comparison Pbuffer
        Display Results to Console Window
        Disable Comparison Pbuffer

        // Render the Whole Scene, but encode the pixels of the target one color
        //      and everything else another color
        Enable Whole Scene False Color Pbuffer
                Draw Background and Obstacle Models All One Color
                Draw Target Model A Different Color
        Bind False Color Texture to Pbuffer
        Disable Whole Scene False Color Pbuffer

        // Generate Boolean map of visible and non-visible pixels
        Enable False Color Comparison Pbuffer
        Set Whole Scene False Color Texture as an Input to Comparison Shader
        Set Target Color as an Input to Comparison Shader
        Enable Comparison Shader
                Draw a Single Quad Spanning the Entire Viewport
        Disable Shader
        Perform Summing Reduction on Comparison Pbuffer
        Display Results to Console Window
        Disable False Color Comparison Pbuffer

        // Render the User's View (the "real" scene plus each of the rendered
        //      textures)
        Draw All Models in the "Main" Window
        Draw Quads with the Whole Scene, Target, and Visualization Textures

        // Update
        Read any Inputs from User
        Update Camera Position as Required, Given Inputs

End Loop
```

APPENDIX B: DTAI CLASSES

AIObject

AIObject is simply a computer generated entity and as such, an object in the scene. In particular, it is an dtCore::Object that can be drawn, transformed (moved), and acted upon physically. AIObject can be a friend or foe of the human player, or both. The methods, Decide() and Act() are called in an update traversal callback. As an Object, methods for moving AIObject are inherited from dtCore::Transformable, and physical traits from dtCore::Physical.

Members:

Focus. A Point in world coordinates at which the AIObject is looking
Eye Position. A point in world coordinates from which AIObject observes
Eye Offset. A vector in object space from the AIObject's model's origin to the Eye Position
Target. An object on whose origin AIObject is focused
List of Targets A list of objects in AIObject's field of view which have been categorized at least as Detected
Field of View. A geometric frustum representing AIObject's field of view.

Methods:

Decide. Process inputs and choose an action.
Act. Update AIObject to implement Decide's choice.
Getters and Setters for each Attribute listed above.

AIObjectManager

AIObjectManager provides target detection, intra-team adjudication and communication between teams of AIObjects. Currently, AIObjectManager only implements the Synthetic Vision algorithm for target detection. AIObjectManager is a non-drawable node on the scene graph. It is iterated through on each traversal, but is not drawn on the draw traversal. Target detection is performed for all AIObjects in the AIObjectManager's team each frame in a cull traversal callback. We chose the cull traversal because the output of performing target detection should be made available at the update traversal. Traversals are multi-threaded, so if target detection was executed during the update traversal, not all targets would be processed where they actually appear after the traversal.

Members:

List of team members (AIObjects)

Node. Pointer to AIObjectManager's osg::Node on the scene graph

Eye Point. A single 'camera' to be used when traversing through team members to perform target detection. This camera is moved to each AIObject's eye position and Synthetic Vision is performed from that point of view.

Methods:

Add Team Member

Remove Team Member

Get List of Team Members

Process Targets. Execution of Synthetic Vision Algorithm on each team member.

PBuffer

This is the implementation of the pixel buffer as described in Chapter 3. It can not be created until the application's windowing system creates graphics and device contexts. From these contexts, OpenGL derives the pbuffer's contexts.

Members:

Last Graphics and Device Contexts. Before making the pbuffer current, store these values so we can restore these contexts later

PBuffer. Actual OpenGL pixel buffer

Pbuffer Contexts. Pbuffer's Graphics and Device Contexts

Size

Draw Buffer. Current portion of the pbuffer being rendered to (WGL_FRONT_LEFT_ARB, for example)

Initialization Booleans. Values required to create the pbuffer. Examples are doubleBuffered, RGB, shareLists.

Pixel Format Variables. Also necessary to create the pbuffer. These are generally minimum number of bits required for each type of buffer (depth, alpha, stencil, etc)

State Booleans. Run-time queries concerning the state of the pbuffer

Methods:

Handle Mode Switch. Ensures the pbuffer is never lost without being recreated

Create pbuffer. Initialization routine

Enable pbuffer. Makes the pbuffer the current rendering context

Disable pbuffer.. Makes the Last Contexts current again.

Bind Textures. Bind textures to the pbuffer's frame and depth buffers

Release Textures. Releases the textures. This must be done before rendering to the pbuffer. Otherwise, results are undefined.

Setters and Getters for some Members

RenderToTextureStage

In the OSG, a RenderStage is an encapsulation of a complete stage in rendering. “Stage” is used here in the Hollywood sense – lights, camera, and only those objects that are supposed to be seen from the given camera angle. RenderToTextureStage is a derived class of osgUtil::RenderStage that renders to a pbuffer instead of the frame buffer. RenderToTextureStage encapsulates pbuffer and owns the textures which are to be bound to that pbuffer. dtAI::RenderToTextureStage should not be confused with osgUtil::RenderToTextureStage which copies the texture from the frame buffer to the CPU back to texture memory.

Members:
- PBuffer
- Size
- Color and Depth Textures. These textures will be bound to the pbuffer.
- Render Mode. Choice between Depth Texture or Color Texture or both.

Methods:
- Draw. Overriden method of osgUtil::RenderStage. It enables the pbuffer before doing the standard draw, binds the textures to the pbuffer and then diables the pbuffer.
- Set Viewport. Overriden method of osgUtil::RenderStage. If the size of the Viewport has changed, resize the pbuffer as well.
- Initialize Textures
- Getters and Setters for the Members

OffscreenSceneView

An osgUtil::SceneView, to a large extent, performs the OSG’s run cycle. Generally, there is one SceneView per camera in the scene. This run cycle has three basic steps: Update, Cull, and Draw. OffscreenSceneView is derived from SceneView for the purpose of rendering off-screen (to the pbuffer). In order to render off-screen and not update the scene graph, the three stages are overridden.

Methods:
- Update. Does nothing since this cycle is executed for each AIObject, each frame. We’ll let the main application perform Updates.
- Cull. Given a node from the scene graph, decide which of the node and its children should be on stage.
- Draw. Draw those objects which are on stage.

ViewInterpreter

ViewInterpreter is the shader utility portion of the library. It specifically creates, intitializes, and updates the shaders used to do visibility comparisons and also to perform the summing reductions. It is taken in large part from an OSG example, osgshaders. In addition to the shaders, it also creates the geometry over which the shaders will be executed. In theory, this is a quad that is the same size as the viewport. This is most easily accomplished by setting the OffscreenSceneView to be an orthographic projection aligned with the quad.

Members:

- Root Node. The osg::Group which is the root of the quad to be rendered
- Whole Scene Color Texture. These textures are used as
- Whole Scene Depth Texture. uniform variable inputs
- Target Color Texture. to the visibility
- Target Depth Texture. comparison shaders
- View Program Object. An OGLSL program consisting of a single fragment shader to be used for visibility comparisons
- View Fragment Program. Visibility Comparison shader
- Reduce Program Object. An OGLSL program to perform reductions
- Reduce Fragment Program. Summing Reduction shader

Methods:

- Load Shader Source. Retrieve the shaders' source code from a file
- Enable Shader
- Set Textures. Set the textures to be used as inputs to shaders
- Initialize Shaders. Creates, links, and compiles shaders.
- Build Scene. Initialize the quad on which the shaders will be run

OffscreenSceneHandler

A Producer::Camera::SceneHandler is an abstract class which prepares the scene to be rendered for the Producer::Camera. OffscreenSceneHandler is a class derived from SceneHandler and is taken largely from dtCore::Scene::_SceneHandler. After all, the off-screen rendering must exactly mimic the on-screen rendering, especially with regard to lighting. In fact, a consideration that has not been implemented in dtAI is the synchronization of the application's SceneHandler with OffscreenSceneHandler. They both have the same defaults, but if SceneHandler is changed, then the off-screen render no longer mimics the on-screen one.

Members:

OffscreenSceneView. This is the link to the camera.

List of Possible Targets. After culling an AIObject's point of view, the custom CullVisitor produces this list of possible targets.

Current RenderToTextureStage. Three RenderToTextureStages are used in this algorithm. This is the one being used currently.

Custom CullVisitor. This culls the AIObject's point of view and collects pointers to the osg::Nodes of all the objects within the field of view. This is used by the whole scene RenderToTextureStage

Default CullVisitor. This culls the point of view, but does not produce a list of possible targets.

Methods:

Clear. Resets the Members

Cull. Collect all objects in the AIObject's field of view for drawing

Draw. Draw the objects collected by Cull

Frame. Called by Camera to initiate the run cycle

Set Scene Data. Set an osg::Node as the head of a subsection of the scene graph. All culling and drawing will be done on this node and its children and below on the scene graph.

Set Draw Buffer. Set the buffer of the pbuffer to which the scene will be rendered

Set RenderToTextureStage. Set the current RenderToTextureStage Member

Set CullVisitor. Set the active CullVisitor. Custom for the whole scene, and default for all others.

AIEyepoint

This is the Synthetic Vision workhorse. This combines a Producer::Camera, a OffscreenSceneHandler and OffscreenSceneView, a ViewInterpreter, and three RenderToTextureStages to implement the visibility algorithm. This brings it all together.

Members:

ViewInterpreter. Encapsulates the shaders

Root Node. The current head of the subsection of the scene graph to do work on. To process the whole scene, the Root Node is the head of the application's entire scene graph. To process a target, the Root Node is simply the target's osg::Node.

Camera. Producer::Camera which is moved to each AIObject's Eye Position. From this position, that AIObject's point of view is rendered and processed.

OffscreenSceneHandler. Processes each RenderToTextureStage at each step of the algorithm.

Whole Scene RenderToTextureStage. Produces the texture containing the entire scene from the AIObject's point of view.
Target RenderToTexture Stage. Produces the texture containing a single target against a clear background from the AIObject's point of view.
Results RenderToTextureStage. Produces a texture encoded with the visilibility of the target at each pixel. This is also used in the Reduction process.

Methods:
Process Field of View. Execute the Synthetic Vision algorithm.
Initialize. Initialize the RenderToTextureStages and shaders.

APPENDIX C: SYNVISION.CPP

```
#include "RenderTexture.h"

#include <GL/glut.h>
#include <Windows.h>
#include <assert.h>
#include <stdio.h>

#include "Model.h"
#include "MilkshapeModel.h"

unsigned char *buf;
GLdouble eyeX;

Model *pModelGround = NULL;   // Holds The Model Data
Model *pModelFigure = NULL;  // Holds The Model Data


void Reshape(int w, int h);

GLuint      iTextureProgram     = 0;
GLuint      iPassThroughProgram = 0;

//RenderTexture *rt = NULL;
RenderTexture *rtFig = NULL;
RenderTexture *rtAll = NULL;
RenderTexture *rtVis = NULL;
RenderTexture *rtRed = NULL;
RenderTexture *rtAll2 = NULL;
RenderTexture *rtRed2 = NULL;


GLhandleARB visTexProg;
GLhandleARB visBoolProg;
GLhandleARB reduceProg;
GLhandleARB visBool2Prog;

bool        bShowDepthTexture = false;

LARGE_INTEGER lastFrame;
int ticks = 0;
double accumFrameTime      = 0.;
double accumReduction1Time = 0.;
double accumReduction2Time = 0.;
double accumCountTime      = 0.;

int printOglError(char *file, int line);
#define printOpenGLError() printOglError(__FILE__, __LINE__)
```

```
//---------------------------------------------------------------------------
// Function      : printInfoLog
// Description      :
//    From ogl2brick
//---------------------------------------------------------------------------
void printInfoLog(GLhandleARB obj)
{
   int infologLength = 0;
   int charsWritten  = 0;
   GLcharARB *infoLog;

   printOpenGLError();  // Check for OpenGL errors

   glGetObjectParameterivARB(obj, GL_OBJECT_INFO_LOG_LENGTH_ARB,
                                       &infologLength);
   printOpenGLError();  // Check for OpenGL errors

   if (infologLength > 0)
   {
      infoLog = (GLcharARB*)malloc(infologLength);
      if (infoLog == NULL)
      {
         printf("ERROR: Could not allocate InfoLog buffer\n");
         exit(1);
      }
      glGetInfoLogARB(obj, infologLength, &charsWritten, infoLog);
      printf("InfoLog:\n%s\n\n", infoLog);
      free(infoLog);
   }
   printOpenGLError();
}


//---------------------------------------------------------------------------
// Function      : isExtensionsSupported
// Description      :
//    From ogl2brick
//    The recommended technique for querying OpenGL extensions;
//    from http://opengl.org/resources/features/OGLextensions/
//---------------------------------------------------------------------------
int isExtensionSupported( const char *extension )
{
    const GLubyte *extensions = NULL;
    const GLubyte *start;
    GLubyte *where, *terminator;

    /* Extension names should not have spaces. */
    where = (GLubyte *) strchr(extension, ' ');
    if (where || *extension == '\0')
        return 0;

    extensions = glGetString(GL_EXTENSIONS);

    /* It takes a bit of care to be fool-proof about parsing the
    OpenGL extensions string. Don't be fooled by sub-strings, etc. */
    start = extensions;
    for (;;) {
```

```
        where = (GLubyte *) strstr((const char *) start, extension);
        if (!where)
            break;
        terminator = where + strlen(extension);
        if (where == start || *(where - 1) == ' ')
            if (*terminator == ' ' || *terminator == '\0')
                return 1;
        start = terminator;
    }
    return 0;
}


//---------------------------------------------------------------------------
// Function      : shaderSize
// Description       :
//     From ogl2brick (modified)
//     Returns the size in bytes of the shader fileName.
//     If an error occurred, it returns -1.
//---------------------------------------------------------------------------
int shaderSize(char *fileName)
{
    int shader;
    int count;

    // Open the file
    shader = _lopen(fileName, OF_READ);
    if (shader == -1)
        return -1;

    // Seek to the end and find its position
    count = _llseek(shader, 0, SEEK_END);

    _lclose(shader);
    return count;
}


//---------------------------------------------------------------------------
// Function      : readShader
// Description:
//     Reads a shader from the supplied file and returns the shader in the
//     arrays passed in. Returns 1 if successful, 0 if an error occurred.
//     The parameter size is an upper limit of the amount of bytes to read.
//     It is ok for it to be too big.
//
//     From ogl2brick (modified)
//---------------------------------------------------------------------------
int readShader(char *fileName, char *shaderText, int size)
{
    FILE *shader;
    int count;

    // Open the file
    shader = fopen(fileName, "r");
    if (!shader)
        return -1;
```

```
    // Get the shader from a file.
    fseek(shader, 0, SEEK_SET);
    count = fread(shaderText, 1, size, shader);
    shaderText[count] = '\0';

    if (ferror(shader))
        count = 0;
    else count = 1;

    fclose(shader);
    return count;
}


//-----------------------------------------------------------------------------
// Function     : readShaderSource
// Description      :
//    From ogl2brick
//-----------------------------------------------------------------------------
int readShaderSource(char *fileName, GLcharARB **fragmentShader)
{
    int fSize;

    // Allocate memory to hold the source of our shaders.
    fSize = shaderSize(fileName);

    if (fSize == -1)
    {
        printf("Cannot determine size of the shader %s\n", fileName);
        return 0;
    }

    *fragmentShader = (GLcharARB *) malloc(fSize);

    // Read the source code
    if (!readShader(fileName, *fragmentShader, fSize)) {
        printf("Cannot read the file %s.frag\n", fileName);
        return 0;
    }

    return 1;
}
```

```
//-----------------------------------------------------------------------------
// Function      : installShader
// Description       :
//    From ogl2brick
//-----------------------------------------------------------------------------
GLhandleARB installShader(GLcharARB *fragment)
{
    GLhandleARB shader, program;   // handles to objects
    GLint       fragCompiled;    // status values
    GLint       linked;

    // Create a fragment shader object
    shader = glCreateShaderObjectARB(GL_FRAGMENT_SHADER_ARB);

    // Load source code strings into shaders
    glShaderSourceARB(shader, 1, (const char**)&fragment, NULL);

    // Compile the brick vertex shader, and print out
    // the compiler log file.
    glCompileShaderARB(shader);
    printOpenGLError();
    glGetObjectParameterivARB(shader,
                GL_OBJECT_COMPILE_STATUS_ARB, &fragCompiled);
    printInfoLog(shader);

    if (!fragCompiled)
        return 0;

    // Create a program object and attach the two compiled shaders
    program = glCreateProgramObjectARB();
    glAttachObjectARB(program, shader);

    // Link the program object and print out the info log
    glLinkProgramARB(program);
    printOpenGLError();
    glGetObjectParameterivARB(program,
                GL_OBJECT_LINK_STATUS_ARB, &linked);
    printInfoLog(program);

    if (!linked)
        return 0;

    return program;
}
```

```
//---------------------------------------------------------------------------
// Function      : printOglError
// Description:
//    Returns 1 if an OpenGL error occurred, 0 otherwise.
//---------------------------------------------------------------------------
int printOglError(char *file, int line)
{
    //
    // Returns 1 if an OpenGL error occurred, 0 otherwise.
    //
    GLenum glErr;
    int    retCode = 0;

    glErr = glGetError();
    while (glErr != GL_NO_ERROR)
    {
        printf("glError in file %s @ line %d: %s\n", file, line,
                 gluErrorString(glErr));
        retCode = 1;
        glErr = glGetError();
    }
    return retCode;
}



//---------------------------------------------------------------------------
// Function      : CreateRenderTexture
// Description:
//---------------------------------------------------------------------------
RenderTexture* CreateRenderTexture(const char *initstr)
{
   printf("\nCreating with init string: \"%s\"\n", initstr);

   int texWidth = 512, texHeight = 512;

   RenderTexture *rt = new RenderTexture();
   rt->Reset(initstr);
   if (!rt->Initialize(texWidth, texHeight))
   {
      fprintf(stderr, "RenderTexture Initialization failed!\n");
   }

   // for shadow mapping we still have to bind it and set the correct
   // texture parameters using the SGI_shadow or ARB_shadow extension
   // setup the rendering context for the RenderTexture
   if (rt->BeginCapture())
   {
      pModelFigure->reloadTextures();
      pModelGround->reloadTextures();
      Reshape(texWidth, texHeight);
      glClearColor(0.5, 0.2, 0.2, 1);
      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
      rt->EndCapture();
   }
```

```
   bShowDepthTexture = false;

   printOpenGLError();
   return rt;
}


//---------------------------------------------------------------------------
// Function      : DestroyRenderTexture
// Description       :
//---------------------------------------------------------------------------
void DestroyRenderTexture(RenderTexture *rt)
{
    delete rt;
}

//---------------------------------------------------------------------------
// Function      : Keyboard
// Description       :
//---------------------------------------------------------------------------
void Keyboard(unsigned char key, int x, int y)
{
    switch(key)
    {
     case 'a':
            eyeX += 2;
            break;
     case 'd':
            eyeX -= 2;
            break;
    case 'q':
        exit(0);
        break;
    default:
        return;
    }
}

//---------------------------------------------------------------------------
// Function      : Idle
// Description       :
//---------------------------------------------------------------------------
void Idle()
{
    glutPostRedisplay();
}
```

```
//----------------------------------------------------------------------------
// Function      : Reshape
// Description       :
//----------------------------------------------------------------------------
void Reshape(int w, int h)
{
    if (h == 0) h = 1;

    glViewport(0, 0, w, h);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    gluPerspective(60.0, (GLfloat)w/(GLfloat)h, 0.1, 5000.0);
}


//----------------------------------------------------------------------------
// Function      : sumReduce
// Description       :
//----------------------------------------------------------------------------
int sumReduce( RenderTexture* rt )
{
   // Get Application Window Width and Height
   int ww = glutGet(GLUT_WINDOW_WIDTH);
   int wh = glutGet(GLUT_WINDOW_HEIGHT);

   // Enable Reduction Shader
   glUseProgramObjectARB(reduceProg);

   // Set Inputs to Shader
   glActiveTexture(GL_TEXTURE0);
   rt->Bind();
   glUniform1iARB(glGetUniformLocationARB(reduceProg, "InputTexture"), 0);
   glUniform1fARB(glGetUniformLocationARB(reduceProg, "Offset"), 1.0/ww);

   // Set OpenGL State
   glClearColor(0.0, 0.0, 0.0, 1);
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glMatrixMode(GL_MODELVIEW);
   glPushMatrix();
   glLoadIdentity();
   glMatrixMode(GL_PROJECTION);
   glPushMatrix();
   glLoadIdentity();
   glOrtho(0.0,(float)ww,0.0,(float)wh,-1.0,1.0);

   int Nt = ww;

   // Successively half the problem size until 2x2 pixels
   for(int scale=1;ww/scale>2;scale*=2) {

      // Calculate Vertex Indices
      int nv = ww/(2.0*scale); // Pix used in viewport
      int nt = ww/scale; // Pix used in texture

      // Determine Texture Coordinate Indices
```

```
      float ta = ((float)nv-(float)nt+1.0)/((float)nv-1.0)/(2.0*(float)Nt);
      float tb = (1.0 + (2.0*(float)nv-1.0)*((float)nt-2.0)/
                       ((float)nv-1.0) )/(2.0*(float)Nt);

      // Reset OpenGL State
      glClearColor(0., 0., 0., 1);
      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

      // Enable Render To Texture
      rt->EnableTextureTarget();

      // Draw Quad and Execute Reduction Shader
      glEnable(GL_TEXTURE_2D);
      glBegin( GL_QUADS );
         glTexCoord2f(ta, ta);
         glVertex2f(0, 0);
         glTexCoord2f(ta, tb);
         glVertex2f(0, nv);
         glTexCoord2f(tb, tb);
         glVertex2f(nv, nv);
         glTexCoord2f(tb, ta);
         glVertex2f(nv, 0);
      glEnd();
      glDisable(GL_TEXTURE_2D);

      // Copy the results from the frame buffer to rt
      glCopyTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, 0, 0, nv, nv);
   }

   // Get Last 2x2 Pixels
   //    Assumes GL_PACK_ALIGNMENT is 4
   for(int k=0; k<16; k++) buf[k]=0;
   glReadPixels(0,0,2,2,GL_RGB,GL_UNSIGNED_BYTE,buf);

   // Repack
   for(k=6; k<12; k++) buf[k]=buf[k+2];

   // Uncomment this to output the results of the reduction
/*   printf("After reduction: %d %d %d %d %d %d %d %d %d %d %d %d\n",
            buf[0],buf[1],buf[2],buf[3],buf[4],buf[5],
            buf[6],buf[7],buf[8],buf[9],buf[10],buf[11]);
*/
   // Sum Visible Pixels from the Final Four pixels
   int nVisible = (buf[0]+buf[3]+buf[6]+buf[9])*65536
      + (buf[1]+buf[4]+buf[7]+buf[10])*256
      + (buf[2]+buf[5]+buf[8]+buf[11]);

   // Reset OpenGL State
   glPopMatrix();
   glMatrixMode(GL_MODELVIEW);
   glPopMatrix();

   // Check for OpenGL Errors
   printOpenGLError();
```

```
    // Disable Shader
    glUseProgramObjectARB(0);

    return nVisible;
}


//------------------------------------------------------------------------------
// Function        : Display
// Description        :
//------------------------------------------------------------------------------
void Display()
{
    // Set Application Window Height and Width
    int ww = glutGet(GLUT_WINDOW_WIDTH);
    int wh = glutGet(GLUT_WINDOW_HEIGHT);

    // Initialize Timer Baseline
    LARGE_INTEGER start;
    QueryPerformanceCounter(&start);

    // Enable Whole Scene Pbuffer
    if (rtAll->IsInitialized() && rtAll->BeginCapture())
    {
        // Set OpenGL State
        if (rtAll->IsDoubleBuffered()) glDrawBuffer(GL_BACK);
        glEnable(GL_DEPTH_TEST);
        glClearColor(0.0, 0.0, 0.0, 1);
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glMatrixMode(GL_MODELVIEW);
        glPushMatrix();
        glLoadIdentity();
        gluLookAt(eyeX, 0, -100, 0, 0, 0, 0, 1, 0);

        // Render figure + ground
        pModelGround->draw();
        pModelFigure->draw();

        // Reset OpenGL State and Disable Pbuffer
        glPopMatrix();
        printOpenGLError();
        rtAll->EndCapture();
    }
    // Calculate Whole Scene Render Time
    LARGE_INTEGER allTime;
    QueryPerformanceCounter(&allTime);
    allTime.QuadPart -= start.QuadPart;


    // Reset Timer Baseline
    QueryPerformanceCounter(&start);
```

```
  // Enable Figure-Only Pbuffer
if (rtFig->IsInitialized() && rtFig->BeginCapture())
{
   // Set OpenGL State
   if (rtFig->IsDoubleBuffered()) glDrawBuffer(GL_BACK);
   glMatrixMode(GL_MODELVIEW);
   glPushMatrix();
   glLoadIdentity();
   gluLookAt(eyeX, 0, -100, 0, 0, 0, 0, 1, 0);
   glClearColor(0.0, 0.0, 0.5, 1);
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

   // Render figure only
   pModelFigure->draw();

   // Reset OpenGL State and Disable Pbuffer
   glPopMatrix();
   printOpenGLError();
   rtFig->EndCapture();
}
// Calculate Figure-Only Render Time
LARGE_INTEGER figTime;
QueryPerformanceCounter(&figTime);
figTime.QuadPart -= start.QuadPart;


// Reset Timer Baseline
QueryPerformanceCounter(&start);

// Enable Visualization Pbuffer
if (rtVis->IsInitialized() && rtVis->BeginCapture())
{
   // Activate Visualization Shader
   glUseProgramObjectARB(visTexProg);

   // Set Inputs to Shader
   glActiveTexture(GL_TEXTURE2);
   rtAll->Bind();
   glActiveTexture(GL_TEXTURE3);
   rtFig->Bind();
   glUniform1iARB(glGetUniformLocationARB(visTexProg, "AllTexture"), 2);
   glUniform1iARB(glGetUniformLocationARB(visTexProg, "FigureTexture"), 3);
   glUniform3fARB(glGetUniformLocationARB(visTexProg, "backgroundColor"),
                                           0.4, 0.0, 0.0);
   // Set OpenGL State
   if (rtVis->IsDoubleBuffered()) glDrawBuffer(GL_BACK);
   glDisable(GL_DEPTH_TEST);
   glMatrixMode(GL_MODELVIEW);
   glPushMatrix();
   glLoadIdentity();
   glMatrixMode(GL_PROJECTION);
   glPushMatrix();
   glLoadIdentity();
   glOrtho(0.0,(float)ww,0.0,(float)wh,-1.0,1.0);
```

```
    // Create Quad Spanning Viewport and Run Shader
   glEnable(GL_TEXTURE_2D);
   glBegin( GL_QUADS );
      glTexCoord2f(0, 0);
      glVertex2f(0, 0);
      glTexCoord2f(0, 1);
      glVertex2f(0, wh);
      glTexCoord2f(1, 1);
      glVertex2f(ww, wh);
      glTexCoord2f(1, 0);
      glVertex2f(ww, 0);
   glEnd();

   // Reset OpenGL State
   glDisable(GL_TEXTURE_2D);
   glPopMatrix();
   glMatrixMode(GL_MODELVIEW);
   glPopMatrix();
   glEnable(GL_DEPTH_TEST);
   printOpenGLError();

   // Disable Pbuffer and Shader
   rtVis->EndCapture();
   glUseProgramObjectARB(0);
}

// Calculate Visualization Render Time
LARGE_INTEGER visTime;
QueryPerformanceCounter(&visTime);
visTime.QuadPart -= start.QuadPart;


// Reset Timer Baseline
QueryPerformanceCounter(&start);

// Enable Comparison Pbuffer
if (rtRed->IsInitialized() && rtRed->BeginCapture())
{
   // Enable Comparison Shader
   glUseProgramObjectARB(visBoolProg);

   // Set up Shader Inputs
   glActiveTexture(GL_TEXTURE2);
   rtAll->Bind();
   glActiveTexture(GL_TEXTURE3);
   rtFig->Bind();
   glUniform1iARB(glGetUniformLocationARB(visBoolProg, "AllTexture"), 2);
   glUniform1iARB(glGetUniformLocationARB(visBoolProg, "FigureTexture"), 3);
   glUniform3fARB(glGetUniformLocationARB(visBoolProg, "backgroundColor"),
                                              0.0, 0.0, 0.5);
```

```
    // Set OpenGL State
   if (rtRed->IsDoubleBuffered()) glDrawBuffer(GL_BACK);
   glMatrixMode(GL_MODELVIEW);
   glPushMatrix();
   glLoadIdentity();
   glMatrixMode(GL_PROJECTION);
   glPushMatrix();
   glLoadIdentity();
   glOrtho(0.0,(float)ww,0.0,(float)wh,-1.0,1.0);

   // Create Quad and Run Shader
   glEnable(GL_TEXTURE_2D);
   glBegin( GL_QUADS );
      glTexCoord2f(0, 0);
      glVertex2f(0, 0);
      glTexCoord2f(0, 1);
      glVertex2f(0, wh);
      glTexCoord2f(1, 1);
      glVertex2f(ww, wh);
      glTexCoord2f(1, 0);
      glVertex2f(ww, 0);
   glEnd();
   glDisable(GL_TEXTURE_2D);

   // Reset OpenGL State
   glPopMatrix();
   glMatrixMode(GL_MODELVIEW);
   glPopMatrix();
   printOpenGLError();

   // Disable Comparison Pbuffer and Shader
   rtRed->EndCapture();
   glUseProgramObjectARB(0);
}

// Calculate Comparison Time
LARGE_INTEGER redTime;
QueryPerformanceCounter(&redTime);
redTime.QuadPart -= start.QuadPart;


// Do Timed Reduction on rtRed
QueryPerformanceCounter(&start);

int nVisible = sumReduce( rtRed );
printf("Number of visible pixels 1:\t%d\n", nVisible);

LARGE_INTEGER reduction1Time;
QueryPerformanceCounter(&reduction1Time);
reduction1Time.QuadPart -= start.QuadPart;


// Reset Timer Baseline
QueryPerformanceCounter(&start);
```

```
// Enable False Color Whole Scene Pbuffer
if (rtAll2->IsInitialized() && rtAll2->BeginCapture())
{
   // Set OpenGL State
   if (rtAll2->IsDoubleBuffered()) glDrawBuffer(GL_BACK);
   glEnable(GL_DEPTH_TEST);
   glClearColor(0.0, 0.0, 0.0, 1);
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glMatrixMode(GL_MODELVIEW);
   glPushMatrix();
   glLoadIdentity();
   gluLookAt(eyeX, 0, -100, 0, 0, 0, 0, 1, 0);

   // Render figure + ground
   pModelGround->drawFlat(0.5,0.5,0.5);
   pModelFigure->drawFlat(1.0,1.0,1.0);

   // Reset OpenGL State and Disable False Color Pbuffer
   glPopMatrix();
   printOpenGLError();
   rtAll2->EndCapture();
}

// Calculate False Color Render Time
LARGE_INTEGER all2Time;
QueryPerformanceCounter(&all2Time);
all2Time.QuadPart -= start.QuadPart;


//Reset Timer Baseline
QueryPerformanceCounter(&start);

// Enable False Color Comparison Pbuffer
if (rtRed2->IsInitialized() && rtRed2->BeginCapture())
{
   // Enable Comparison Shader
   glUseProgramObjectARB(visBool2Prog);

   // Set Shader Inputs
   glActiveTexture(GL_TEXTURE0);
   rtAll2->Bind();
   glUniform1iARB(glGetUniformLocationARB(visBool2Prog, "AllTexture"), 0);
   glUniform3fARB(glGetUniformLocationARB(visBool2Prog, "figureColor"),
                  1.0, 1.0, 1.0);

   // Set OpenGL State
   if (rtRed2->IsDoubleBuffered()) glDrawBuffer(GL_BACK);
   glClearColor(0.0, 0.0, 0.0, 1);
   glClear(GL_COLOR_BUFFER_BIT );
   glViewport(0, 0, ww, wh);
   glMatrixMode(GL_MODELVIEW);
   glPushMatrix();
   glLoadIdentity();
   glMatrixMode(GL_PROJECTION);
   glPushMatrix();
   glLoadIdentity();
   glOrtho(0.0,(float)ww,0.0,(float)wh,-1.0,1.0);
```

```
    // Draw Quad and Run Shader
    glEnable(GL_TEXTURE_2D);
    glBegin( GL_QUADS );
       glTexCoord2f(0, 0);
       glVertex2f(0, 0);
       glTexCoord2f(0, 1);
       glVertex2f(0, wh);
       glTexCoord2f(1, 1);
       glVertex2f(ww, wh);
       glTexCoord2f(1, 0);
       glVertex2f(ww, 0);
    glEnd();
    glDisable(GL_TEXTURE_2D);

    // Reset OpenGL State
    glPopMatrix();
    glMatrixMode(GL_MODELVIEW);
    glPopMatrix();
    printOpenGLError();

    // Disable Comparison Pbuffer and Shader
    rtRed2->EndCapture();
    glUseProgramObjectARB(0);
 }

 // Calculate Red2 RenderTime
 LARGE_INTEGER red2Time;
 QueryPerformanceCounter(&red2Time);
 red2Time.QuadPart -= start.QuadPart;


 // Do Timed Redcution on rtRed2
 QueryPerformanceCounter(&start);

 int nVisible2 = sumReduce( rtRed2 );
 printf("Number of visible pixels 2:\t%d\n", nVisible2);

 LARGE_INTEGER reduction2Time;
 QueryPerformanceCounter(&reduction2Time);
 reduction2Time.QuadPart -= start.QuadPart;


 // Begin render visible scene
 // Set OpenGL State to Render Main Window
 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
 glMatrixMode(GL_MODELVIEW);
 glPushMatrix();
 glLoadIdentity();
 gluLookAt(eyeX, 0, -100, 0, 0, 0, 0, 1, 0);

 // Draw Figure + Ground
 pModelGround->draw();
 pModelFigure->draw();
```

```
// Set OpenGL State to Render Smaller Views
glMatrixMode(GL_MODELVIEW);
glPushMatrix();
glLoadIdentity();
glMatrixMode(GL_PROJECTION);
glPushMatrix();
glLoadIdentity();
glOrtho(0.0,(float)ww,0.0,(float)wh,-1.0,1.0);

float of = 0.01;  // offset fraction
float sf = 0.1;  // size fraction
float space=ww/5;

// Bind Textures to Rectangles
for(int i=0;i<4;i++) {
   if(i==0) { // Bind Whole Scene Pbuffer
      rtAll->Bind();
      if (rtAll->IsDoubleBuffered()) rtAll->BindBuffer(WGL_BACK_LEFT_ARB);
      rtAll->EnableTextureTarget();
   }
   if(i==1) { // Bind Figure-Only Pbuffer
      rtFig->Bind();
      if (rtFig->IsDoubleBuffered()) rtFig->BindBuffer(WGL_BACK_LEFT_ARB);
      rtFig->EnableTextureTarget();
   }
   if(i==2) { // Bind Occluded Figure-Only Pbuffer
      rtVis->Bind();
      if (rtVis->IsDoubleBuffered()) rtVis->BindBuffer(WGL_BACK_LEFT_ARB);
      rtVis->EnableTextureTarget();
   }
   if(i==3) {// Bind False Color Whole Scene Pbuffer
      rtAll2->Bind();
      rtAll2->EnableTextureTarget();
   }

   // Draw Rectangle with Bound Texture (Pbuffer)
   glEnable(GL_TEXTURE_2D);
   glBegin( GL_QUADS );
      glTexCoord2f(0, 0);
      glVertex2f(space*i+of*ww, of*wh);
      glTexCoord2f(0, 1);
      glVertex2f(space*i+of*ww,  (of+sf)*wh);
      glTexCoord2f(1, 1);
      glVertex2f(space*i + (of+sf)*ww,  (of+sf)*wh);
      glTexCoord2f(1, 0);
      glVertex2f(space*i + (of+sf)*ww, of*wh);
   glEnd();
   glDisable(GL_TEXTURE_2D);
```

```
    // Disable Pbuffer
    if(i==0) {
      rtAll->DisableTextureTarget();
    }
    if(i==1) {
       rtFig->DisableTextureTarget();
    }
    if(i==2) {
       rtVis->DisableTextureTarget();
    }
    if(i==3) {
       rtAll2->DisableTextureTarget();
    }
 }

 // Reset OpenGL State
 glPopMatrix();
 glMatrixMode(GL_MODELVIEW);
 glPopMatrix();
 glPopMatrix();


 // Reset Timer Baseline
 QueryPerformanceCounter(&start);


 // UNCOMMENT to Render False Color to screen and
 //    use ReadPixels to count pixels
/*   glClearColor(0.0,0.0,0.0,1.0);
 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
 glMatrixMode(GL_MODELVIEW);
 glPushMatrix();
 glLoadIdentity();
 glMatrixMode(GL_PROJECTION);
 glPushMatrix();
 glLoadIdentity();
 glOrtho(0.0,(float)ww,0.0,(float)wh,-1.0,1.0);

 // Bind False Color Whole Scene
 rtAll2->Bind();
 rtAll2->EnableTextureTarget();

 // Draw Main Window
 glEnable(GL_TEXTURE_2D);
 glBegin( GL_QUADS );
    glTexCoord2f(0, 0);
    glVertex2f(0, 0);
    glTexCoord2f(0, 1);
    glVertex2f(0,  wh);
    glTexCoord2f(1, 1);
    glVertex2f(ww,  wh);
    glTexCoord2f(1, 0);
    glVertex2f(ww, 0);
 glEnd();
 glDisable(GL_TEXTURE_2D);
```

```
    // Reset OpenGL State
    glPopMatrix();
    glMatrixMode(GL_MODELVIEW);
    glPopMatrix();

    int nv = ww;
    int sum_gray = 0;
    int sum_white = 0;
    int sum_black = 0;

    // Reset buffer
    for(int j=0;j<nv*nv*3;j++) buf[j]=0;

    // Fill buffer with pixels from framebuffer
    glReadPixels(0,0,nv,nv,GL_RGB,GL_UNSIGNED_BYTE,buf);

    // Count White, Grey, and Black Pixels
    for(int i=0;i<nv;i++) {
       for(j=0;j<nv;j++) {
          int blue = buf[i*nv*3+j*3+2];
          int green = buf[i*nv*3+j*3+1];
          int red = buf[i*nv*3+j*3+0];
          if(red==255 && green==255 && blue==255)
             ++sum_white;
          else if(red==128 && green==128 && blue==128)
             ++sum_gray;
          else if(red==0 && green==0 && blue==0)
             ++sum_black;
          else { // Some Other Color (error)
             printf("(%d %d %d) ",red,green,blue);
          }
       }
    }
    //printf("\nNumber of white pixels in rtAll2: %d  Gray pixels: %d  " +
    //        "Black: %d   Total: %d \n", sum_white, sum_gray,
    //        sum_black, sum_white+sum_gray+sum_black);
    //printf("Number of visible pixels 3:\t%d\n",sum_white);
 */
    // Calculate Count Time
    LARGE_INTEGER countTime;
    QueryPerformanceCounter(&countTime);
    countTime.QuadPart -= start.QuadPart;


    // Swap Frame Buffers to display
    printOpenGLError();
    glutSwapBuffers();

    // Get This Frame Time
    LARGE_INTEGER frameTime;
    QueryPerformanceCounter(&frameTime);

    // Find Unit Frequency ( 1/freq units per second )
    LARGE_INTEGER freq;
    QueryPerformanceFrequency(&freq);
```

```
   // Determine Overhead Involved in Querying Time
   LARGE_INTEGER ctr1;
   LARGE_INTEGER ctr2;
   LARGE_INTEGER overhead;
   QueryPerformanceCounter(&ctr1);
   QueryPerformanceCounter(&ctr2);
   overhead.QuadPart = ctr2.QuadPart - ctr1.QuadPart;

   // Accumulate Times and Frames (ticks)
   ticks++;
   accumReduction1Time += (double)(reduction1Time.QuadPart - overhead.QuadPart)
                          / (double)(freq.QuadPart);
   accumReduction2Time += (double)(reduction2Time.QuadPart - overhead.QuadPart)
                          / (double)(freq.QuadPart);
   accumCountTime += (double)(countTime.QuadPart - overhead.QuadPart)
                          / (double)(freq.QuadPart);
   accumFrameTime += (double)(frameTime.QuadPart - lastFrame.QuadPart
                          - overhead.QuadPart) / (double)(freq.QuadPart);
   lastFrame = frameTime;

   // Calculate Average Times
   double avgRedTime = accumReduction1Time / (double)ticks;
   double avgRed2Time = accumReduction2Time / (double)ticks;
   double avgCountTime = accumCountTime / (double)ticks;
   double avgFrameTime = accumFrameTime / (double)ticks;

}


//----------------------------------------------------------------------------
// Function      : main
// Description       :
//----------------------------------------------------------------------------
void main()
{
   // Create Container for Shader Source (to load shaders)
   GLcharARB *FragmentShaderSource;

   // Initialize Window
   glutInitDisplayMode(GLUT_RGB | GLUT_DEPTH | GLUT_DOUBLE);
   glutInitWindowPosition(50, 50);
   glutInitWindowSize(512, 512);
   glutCreateWindow("TestRenderTexture");

   // Check for OpenGL Errors
   int err = glewInit();
   if (GLEW_OK != err)
   {
      // problem: glewInit failed, something is seriously wrong
      fprintf(stderr, "GLEW Error: %s\n", glewGetErrorString(err));
      exit(-1);
   }
```

```
// Ensure Required OpenGL Extensions are Supported
if ( !isExtensionSupported( "GL_ARB_shader_objects" )    ||
     !isExtensionSupported( "GL_ARB_fragment_shader" )  ||
     !isExtensionSupported( "GL_ARB_vertex_shader" )    ||
     !isExtensionSupported( "GL_ARB_shading_language_100" ) )
{
   printf("OpenGL Shading Language extensions not available\n" );
   exit(-1);
}

// Register Functions to be executed in glutMainLoop
glutDisplayFunc(Display);
glutIdleFunc(Idle);
glutReshapeFunc(Reshape);
glutKeyboardFunc(Keyboard);

// Ensure Window is 512x512
Reshape(512, 512);

// Initialize Container for Retrieved Pixels
buf = new unsigned char[512*512*3];

// Setup Main App's OpenGL State
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(eyeX, 0, -100, 0, 0, 0, 0, 1, 0);
glDisable(GL_LIGHTING);
glEnable(GL_COLOR_MATERIAL);
glEnable(GL_DEPTH_TEST);
glClearColor(0.4, 0.6, 0.8, 1);

// Output Projection Matrix
float m[16];
glGetFloatv(GL_PROJECTION_MATRIX,m);
for(int i=0;i<16;i++) printf("%f ",m[i]);
printf("\n");

// Load Model of Background and Obstacle
pModelGround = new MilkshapeModel();

if ( pModelGround->loadModelData( "data/ground.ms3d" ) == false )
{
fprintf( stderr, "Couldn't load the model data\\model.ms3d");
exit( -1 );
}

// Load Model of Figure (target)
pModelFigure = new MilkshapeModel();
if ( pModelFigure->loadModelData( "data/figure.ms3d" ) == false )
{
fprintf( stderr, "Couldn't load the model data\\model.ms3d");
exit( -1 );
}
```

```
    // Create and Initialize RenderTextures
    rtAll = CreateRenderTexture("rgb tex2D depthTex2D");
    rtFig = CreateRenderTexture("rgb tex2D depthTex2D");
    rtVis = CreateRenderTexture("rgb tex2D");
    rtRed = CreateRenderTexture("rgb tex2D");
    rtAll2 = CreateRenderTexture("rgb tex2D depthTex2D");
    rtRed2 = CreateRenderTexture("rgb tex2D");

    // Load OpenGL Shading Language Shaders
    readShaderSource("colorbasedvis.frag", &FragmentShaderSource);
    visTexProg = installShader(FragmentShaderSource);

    readShaderSource("boolvis.frag", &FragmentShaderSource);
    visBoolProg = installShader(FragmentShaderSource);

    readShaderSource("sumreduce.frag", &FragmentShaderSource);
    reduceProg = installShader(FragmentShaderSource);

    readShaderSource("boolvis2.frag", &FragmentShaderSource);
    visBool2Prog = installShader(FragmentShaderSource);

    printf("Press a or d to sidestep left or right.  Press q to quit.\n");

    // Register first Time for Timing Frames
    QueryPerformanceCounter(&lastFrame);

    // And awaaaay we go...
    glutMainLoop();
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

GLOSSARY

Callback

A method which is registered with a scene graph node and is called by that node when it is traversed by the corresponding traversal. For example, a cull callback can be registered with a node to execute when that node is traversed by a cull traversal. Some common callbacks are cull, draw, and update. Some scene graphs also implement pre- and post- callbacks for execution before and after traversing the node.

Classification

Discrimination between general classes of vehicles (tracked, wheeled, etc.). [NVESD1]

Clip Space

"A coordinate space that is suitable for clipping." [Rost04] In the rendering pipeline, objects are taken from the viewing volume to clip space by an application of the Projection Transformation.

Culling

The removal of an object or objects from the list of objects to be rendered because they lie outside the view frustum.

Depth Buffer

Also referred to as the Z Buffer. A component of the frame buffer which, just as the color buffer holds the color information of each pixel, holds the depth of each pixel. The depth buffer, through depth testing, is used to determine if the current pixel is in front or behind any previous pixel written at the same window position.

Detection

Determination that a target is present within a field-of-view. [NVESD1]

DirectX

Microsoft DirectX is a suite of multimedia API's built into the Microsoft Windows operating system.

Eye Space

A coordinate space relative to the observer or camera's viewpoint. An object is transformed from object space to eye space by application the modelview matrix.

Fragment

A discrete unit of area on the interior of a primitive. Its color and depth values are interpolated from the values of the vertices of that primitive. It is put through several tests by the fragment processor and if it survives, the fragment is passed to the rasterizer to be mapped to pixels.

Fragment Program

On a programmable GPU, a program that is executed by the fragment processor, replacing the OpenGL "fixed functionality."

Frame Buffer

A section of video memory dedicated for use to store rendering information in the form of arrays of pixels. The frame buffer comprises several subbuffers: color, depth, stencil, accumulation, and pixel buffers.

Frame Buffer Memory

The portion of video memory dedicated for frame buffer use.

Graphics Library Utility Toolkit (GLUT)

A platform-independent toolkit for creating windowed applications. Documentation can be found at http://www.opengl.org/documentation/specs/glut/spec3/spec3.html

Graphics Library Extension Wrangler (GLEW)

An API to initialize and use OpenGL extensions. GLEW also provides a means to determine if particular extensions are available at run time. GLEW was originally written by Milan Ikits and Marcelo Magallon. [http://glew.sourceforge.net/]

Identification

Discrimination between specific targets (T72, M1, Chieftain, etc.). [NVESD1]

Line of Sight

A geometric determination of whether the view of the target from an observer is unobstructed.

Load Balancing

The effort to equalize the loads of the CPU and GPU to realize optimal performance. This is a dynamic effort as at any moment, an application can be either CPU- or GPU-bound. Load balancing attempts to detect and relieve the computational load of the bound processor.

Object Space

A coordinate space relative to an object. The origin of that object space is coincident with the object's origin.

Open Graphics Library (OpenGL).

A C++ API for rendering objects to a display device. It is widely supported by hardware manufacturers. OpenGL is a trademark of Silicon Graphics, Inc.

Pixel

The color value at a specific point in a scene. Technically, as a point, a pixel has no area. So, when talking about processing pixels and pixel shaders, we are actually referring to fragments.

Pixel Buffer (PBuffer)

A segment of graphics memory analagous to a frame buffer. Unlike the frame buffer, the pixel buffer is not designed to be displayed directly to the user. It is often used for off-screen rendering and rendering to texture, where the pbuffer can be bound in a fashion similar to a texture.

Primitive

A basic unit of graphics geometry. All geometric models are composed of at least one primitive. There are few primitives in OpenGL: triangle, quad, and point. There are also optimized groups of these primitives: triangle strip and quad strip. One basic rule in OpenGL is that the vertices of a primitive must be coplanar. Otherwise, results are undefined.

Recognition

Discrimination between categories within a class of similar objects (tank, APC, self propelled howitzer, etc.). [NVESD1]

Render To Texture

A process in which a scene is rendered to a render target and that render target is accessible as a texture. Currently render to texture is only supported in Microsoft Window by OpenGL. Other platforms must use a “copy to texture” scheme where the scene is rendered to a render target and then the color information is copied from that render target to a texture.

Resolvable Cycles

The number of finite units of resolution that span the critical dimension of a target in the electro-optic sensor’s field of view. [Johnson58]

Scene Graph

A tree composed of nodes representing objects and operations on those objects in a three-dimensional, graphical world. Operations on a scene graph are usually performed by traversals, where each node in the tree is acted upon only once per traversal.

Shader Object

A component of a Shader Program, whether a vertex or fragment shader object. The shader objects are compiled and linked to form Shader Programs. [Rost04]

Spot Detection

"The target spot detection (also referred to as "star" detection) methodology used in ACQUIRE is designed for cases in which the target is viewed against a uniform background." [NVESD1]

Target Discrimination

"The target discrimination methodology is useful for target detection situations in which a separation of the target characteristics from the background is required (e.g., when a target is embedded in a non-uniform or cluttered background). The target discrimination methodology can be used for the prediction of greater levels of target discrimination beyond detection such as classification, recognition, and identification." [NVESD1]

Texture

An image which can be applied to the surface of an object. Textures can be images read from files or can be generated procedurally. Because textures can be created procedurally and can be used as inputs to shader programs, they are used to provide array of inputs to shader objects.

Texture Memory

Video card memory dedicated to the storage of texture information. This memory is finite, so if more textures are required in a scene than can be stored in texture memory, the textures are paged in and out of texture memory.

Traversal

In a scene graph, a visiting task in which each node in the scene graph is acted upon only once. Cull, Draw, and Update are three examples of common scene graph traversals.

Vertex

Similar to the geometric definition, a point that describes a corner of a primitive. Triangles have three vertices, for example.

Vertex Program

On a programmable GPU, a program that is executed by the vertex processor, replacing the OpenGL "fixed functionality."

LIST OF REFERENCES

Aken02 Akenine-Möller, Thomas, and Haines, Eric. *Real Time Rendering, Second Edition*. A.K. Peters. 2002.

ARB11 OpenGL Architecture Review Board. OpenGL Extension Number 11. *WGL_ARB_pbuffer*. 12 March 2002

ARB20 OpenGL Architecture Review Board. OpenGL Extension Number 20. *WGL_ARB_render_texture*. 16 July 2001

ARB99 OpenGL Architectural Review Board, Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*. Addison Wesley, 1999

ARB263 OpenGL Architecture Review Board. OpenGL Extension Number 263. *ARB_NV_render_depth_texture*. 8 January 2003

Brook "BrookGPU". [http://graphics.stanford.edu/projects/brookgpu/index.html]. September 2004

Buck04 Buck, Ian and Purcell, Tim. "A Toolkit for Computation on GPUs." In R. Fernando (Ed), *GPU Gems* (pp 621-636). Addison-Wesseley. 2004

Burns Burns, D. "Open Producer". [http://www.andesengineering.com/Producer/]. September 2004

Champ96 TRAC White Sands Missile Range. TRAC-WSMR-TR-99-001(R). *Effects of Vegetation on Line-Of-Sight (LOS) for Dismounted Infantry Operations*. Champion, D., Fatale, L., and Krause, P. June 1999.

Delta Johnson, E. "Delta3D Open Source Engine". [http://www.nps.navy.mil/cs/research/vissim/Engine/enginemain.html]. September 2004

Driels95 Driels, Morris R. and Judith H. Lind. *Prototoype Line of Sight and Target Acquisition Software for JANUS (A) High Resolution Databases*. Naval Postgraduate School, 1995.

GP2 Lastra, A., Lin, M., and Manocha, D. (Editors). *2004 ACM Workshop on General-Purpose Computing on Graphics Processors*. Organizing Committee of the 2004 ACM Workshop on General-Purpose Computing on Graphics Processors. 2004

HarrisWeb Harris, M. "Mark Harris' Real-Time Graphics Research. [http://www.markmark.net/misc/rendertexture.html]. September 2004

Johnson58 Johnson, John. "Analysis of Image Forming Systems". *Image Intensifier Symposium 6-7 October 1958*, pp 249-274. US Army Engineer Research and Development Laboratories; Corps of Engineers.

Klim03 Klimenko, Stanislav, Nikitina, Lialia, and Nikitin, Igor. "Parallel Visibility Test and Occlusion Culling in Avango Virtual Environment Framework", *Proceedings of Eurographics 2003***.**

Milk Ciragan, M. "chUmbaLum sOft ". [http://www.swissquake.ch/chumbalum-soft/]. September 2004

NeHe31 Nehe Productions: OpenGL Lesson # 31. [http://nehe.gamedev.net/data/lessons/lesson.asp?lesson=31]. September 2004.

NVESD1 U.S. Army Night Vision & Electronic Sensors Directorate. Standard Category Acquire Approved Standard.

NVESD2 U.S. Army Night Vision & Electronic Sensors Directorate. Standard Category Acquire Contrast Model.

OpenGL1 "OpenGL – The Industry Standard for High Performance Graphics." [http://www.opengl.org/]. September 2004

OpenGL2 "OpenGL Overview". [http://www.opengl.org/about/overview.html]. September 2004

OSG1 Osfield, R. "OpenSceneGraph: the high-performance open source graphics toolkit." [http://openscenegraph.sourceforge.net/]. September 2004

OSG2 Burns, D. "Open Scene Graph: Features & Goals." [http://www.openscenegraph.org/featuresngoals/]. September 2004

Peli95 Peli, Eli. *Vision Models for Target Detection and Recognition*. World Scientific, 1995.

Prank "GPU definition – define GPU." [http://www.isprank.com/glossary/GPU.html]. September 2004

Proc04 Proctor, Michael D., and William J. Gerber. "Line-of-sight Attributes for a Generalized Application Program Interface". *JDMS: The Journal of Defense Modeling and Simulation: Applications, Methodology, Technology*, Volume 1, Issue 1, pp 43-57. The Society for Modeling and Simulation International: April 2004

RostWeb "Orange Book Shader Source Code." [http://www.3dshaders.com/shaderSource.html]. September 2004

Rost04 Rost, Randi. *OpenGL Shading Language*. Addison-Wesley, 2004.

Reece96 Reece, Douglas A., and Ralph Wirthlin "Detection Models for Computer Generated Individual Combatants". *6th Conference on Computer Generated Forces and Behavioral Representation*, University of Central Florida, 1996.

SLang Kessenich, John, Dave Baldwin, and Randi Rost. THE OPENGL® SHADING LANGUAGE. Version 1.10, Document Revision 59. 30 April 2004.

Smith95 Smith, Alvy Ray. "A Pixel Is *Not* A Little Square, A Pixel Is *Not* A Little Square, A Pixel Is *Not* A Little Square! (And a Voxel Is *Not* A Cube)". Microsoft Technical Memo 6. Microsoft. 1995.

Thomp00 Thompson, Chris J., Sahngvun Hahn, and Mark Oskin. "Using Modern Graphics Architectures for General-Purpose Computing: A Framework and Analysis". IEEE, 2000.

UOFACT1 AMSO. "AMSO MOUT FACT" [https://www.moutfact.army.mil/research.asp#search]. September 2004

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Chris Darken
   Naval Postgraduate School
   Monterey, California

4. Joe Sullivan
   Naval Postgraduate School
   Monterey, California

5. Rudy Darken
   Naval Postgraduate School
   Monterey, California

6. Marine Corps Representative
   Naval Postgraduate School
   Monterey, California

7. Director, Training and Education, MCCDC, Code C46
   Quantico, Virginia

8. Director, Marine Corps Research Center, MCCDC, Code C40RC
   Quantico, Virginia

9. Marine Corps Tactical Systems Support Activity (Attn: Operations Officer)
   Camp Pendleton, California