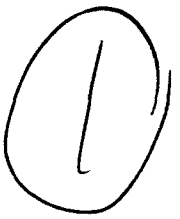
AFIT/GCS/ENG/94M-01



A FORMAT FOR STORING AND MANAGING MULTIPLE LEVEL OF DETAIL TERRAIN FOR SIMULATED ENVIRONMENTS

THESIS

Keith L. Meissner, Captain, USAF

AFIT/GCS/ENG/94M-01

AFIT/GCS/ENG/94M-01

A FORMAT FOR STORING AND MANAGING MULTIPLE LEVEL OF DETAIL TERRAIN FOR SIMULATED ENVIRONMENTS

THESIS

Presented to the Faculty of the Graduate School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Computer Systems

Keith L. Meissner, B.A.
Captain, USAF

March 1994

Approved for public release; distribution unlimited

Acknowledgments

So many people helped me with my work on this thesis, I could not remember enough to name them all.

I wish to thank my advisor, Lt Col Patricia Lawlis, along with the other members of my thesis committee, Lt Col Philip Amburn and Lt Col Martin Stytz. They were patient with me, giving me the time to make this thesis work. I'm thankful for their advice and hard to answer questions that made me think. I also wish to thank Lt Col David Neyland of the Advanced Research Project Agency, whose sponsorship made this project possible.

I'm grateful to Steve Sheasby and Dave Doak for their technical support and answers to questions that would have taken me days to figure out. And many thanks for inspiration from my classmates and friends Mike Gardner, Mark Snyder, Alain Jones, Andrea Kunz, Brian Soltz, Kirk Wilson, Matt Erichsen and Bill Gerhard.

Of course, I thank my wife, Kristina, for constantly reminding me to spend some time with my family. And I thank my children, Erika and Stephen, for occasionally taking my mind off all the difficult things.

Most of all, I thank Jesus Christ, my Lord, without whom I could not have kept the peace of mind to complete this project.

Table of Contents

List of Figures

List of Tables

Abstract

This study investigated a method of storing, managing and rendering terrain data, while addressing conflicting goals of: rendering speed, display detail and memory usage. A data structure is presented to store terrain data, with an object oriented system to manage the data stored in the structure. The structure stores terrain data in a compact form which is converted into rendering structures in real time. The structure uses levels of detail to maintain display detail. The structure is compared against an existing format for storing terrain data, MultiGen Flight. The system managing the structure is shown to decrease memory usage and increase frame rate, while maintaining display detail. The structure offers features not attainable from Flight format, including: interpolating vertices to prevent cracking, defining levels of detail by altitude or speed, and betweening. The structure allows storing and rendering larger databases than previously manageable with Flight format.

A FORMAT FOR STORING AND MANAGING MULTIPLE LEVEL OF DETAIL TERRAIN FOR SIMULATED ENVIRONMENTS

I. Introduction

Terrain is an important part of simulated environments. However, large areas of highly detailed terrain require massive databases, and rendering terrain substantially affects a simulation's frame rate. Level of detail strategies can improve frame rate, but increase database storage needs. Large databases also have the problem of paging needs, having data in main memory when it is needed.

This thesis presents a format for storing and managing terrain data. The format stores terrain at an abstract level of cells, with each cell representing a small area of terrain. The cell format compacts the data to reduce redundancy, while making the data available for multiple levels of detail. A system to manage terrain rendering may build rendering structures for many cells at various levels of detail.

The strategy behind the cell format: as we get larger databases, the first response is, "Throw hardware at it." This hardware might be in the form of main memory or disk packs. Terrain data can amount to Gigabytes of information. Storing all data in main memory becomes expensive. Without enough main memory, paging terrain data into memory may slow down frame updates. What if the data may be compacted, reducing paging needs, but requiring more CPU time to convert from the compact format into a rendering format? Memory and paging needs are reduced, and the terrain may be built in more flexible methods than in a fixed rendering format.

1.1. Background

Training is essential to an effective military force. Training prepares military troops to fight, giving them the experience and confidence to fight effectively. During training, troops can make mistakes and learn from those mistakes. They can experiment with new tactics.

In a nation of rapidly declining defense spending, we must search for cost effective methods of training. Specifically, we are turning to simulated training environments.

To be effective, simulated training environments must be realistic enough to be credible to those being trained. Hence, we want to make our simulations as real as possible. In simulated training, realism results from two characteristics: action and appearance. Action is the heart of a simulation, determining such things as enemy reactions and weapon system response. Appearance is the interface to the simulation, getting information to and from the participants.

One might assume the action is the most important portion of a simulation. However, the action and appearance are extremely interdependent. The appearance is the ambassador for the simulation. It gives the first impression, which can make or break the simulation's credibility. A good simulation cannot overcome a bad interface. A participant must be able to get the essential information from a simulation to make effective use of the simulated training.

One part of a simulation's interface is the graphical interface. A graphical interface provides participants with information through pictures, as opposed to a text based interface that provides information through words. In simulated training, the graphical interface supplies participants with a view of the field of battle.

To provide realism, a graphical interface needs certain external characteristics (the appearance of the graphics): detail and rapid update. Detail includes the representation of objects, how a participant sees the objects on the field of battle. The detail of an object may vary, depending on how "close" the object is to a participant's location on the field of

battle. The update rate determines how often the picture can change. A rapid update rate provides a smooth flowing, moving picture.

For flexibility, a graphical interface needs certain internal characteristics (the structure of the software): understandability and maintainability. Understandability determines how easily a new programmer can determine how the software operates. Maintainability determines how easily the software can be modified.

Unfortunately, these characteristics (external and internal) are often mutually elusive (but not necessarily mutually exclusive); to gain in the area of one characteristic might mean giving up gains in another characteristic. The most elusive characteristic is rapid update. Gains in each of the other characteristics adversely affect the rate of update.

This thesis effort will address the balancing of these characteristics for rendering a simulation's view of terrain. Simulations are often performed in open areas, usually covering great expanses of terrain. For realistic training, we need terrain that reflects the real world[15].

Real world terrain models require massive databases, with little opportunity to reuse (instance) pieces of the terrain. Terrain covers large areas of a simulated view; some of the terrain is close, while most of the terrain is at a distance. Levels of detail in terrain requires breaking the terrain into sections, creating levels of detail, then combining the sections into a solid piece of terrain.

1.2. Problem

We need a method of rendering terrain that manages the conflicting goals of: rendering speed, display detail and memory usage. This method must be adaptable, allowing for future modifications.

1.3. Summary of Current Knowledge

Students at the Air Force Institute of Technology (AFIT) have been researching the area of simulated training for several years. Recent simulations include prototypes of a

Virtual Cockpit[34][27][19][10][12] and Synthetic BattleBridge[14][32][36]. These simulations provide participants with a real time battlefield view through graphical displays.

Terrain for these simulations is provided in Software Systems' MultiGen Flight format.[31] The Flight format files are either constructed by hand (without levels of detail) or built using automated tools. The simulations use Silicon Graphics' Performer system to render the terrain (and other objects in the simulations' view). The terrain data is stored in Flight format and converted to Performer format for rendering.

1.4. Scope

This thesis is limited to rendering terrain in levels of detail, aimed at the view from a low flying aircraft. The characteristics of the terrain are limited to flat shading, Gouraud shading and texturing.

1.5. Approach

This thesis deals with some of the conflicting goals of rendering terrain: rendering speed, display detail and memory usage. It employs a data structure to store terrain data and a system to manipulate the data. The terrain data is stored in sections with a common structure.

First, I approached this thesis by creating a data structure to store the terrain information in small sections, called cells. The data structure views terrain as a single layer of skin. To reduce the redundancy of the data, a cell stores several levels of detail in one data set. The vertices that make up a cell have a standard connectivity, allowing connectivity information to be inferred from the structure. Additionally, the standard connectivity guarantees the terrain will form triangle meshes.

Second, I created a system to manipulate the terrain data. The terrain is divided into lattices of cells. Each cell can be represented at an appropriate level of detail or

ignored, depending on its distance from the viewpoint. Rings of declining level of detail cells are built around a viewpoint.

The system views the data as being stored in an intermediate structure from which to fill the rendering data structures. The data in the rendering structure can be manipulated to interpolate between levels of detail ("betweening"), without affecting the base data.

1.6. Standards

The software created for this thesis was measured against several criteria. Since the software will be modified, the software was measured against selected maintainability metrics. The execution characteristics of the software were measured for memory usage, paging needs, conversion processing, and rendering speed.

1.7. Materials and Equipment

The Virtual Cockpit and Synthetic Battle Bridge produce their graphical displays on Silicon Graphics IRIS 4D/440VGXT and ONYX Reality Engine2 workstations, using C++ and the Performer system provided by Silicon Graphics.

The IRIS 4D workstations are general purpose machines of relatively low cost (in the realm of graphical workstations). The ONYX workstations are more powerful (and more expensive) machines. The system should run on both machines, because cost is a factor; a simulation may require many workstations for many participants.

The C++ programming language was chosen because it was the only object oriented language available at the outset of this thesis.

1.8. Other Support

This thesis was built on top of Capt Mark Snyder's ObjectSim software.[30] ObjectSim uses Silicon Graphics' Performer to provide an object oriented management

system for entities in a synthetic environment. Object orientation is a software engineering technique whereby software is divided into units suggested by real world boundaries.

1.9. Thesis Overview

The following chapter presents an overview of graphics issues and software engineering issues applied to the system developed for this thesis. Chapter 3 offers a detailed look at the cell data structure, while Chapter 4 outlines the system structure. Chapter 5 presents an analysis of the system, and the final chapter submits results and discusses recommendations for further study.

II. Background

2.1. Overview

This thesis approaches terrain processing and rendering from two interrelated perspectives: graphics and software engineering. Graphics is concerned with levels of detail (LODs), database storage and memory usage. Software engineering looks at these issues from the perspective of understandability and maintainability, as well as efficiency of the software.

This thesis builds from past theses and fits into current projects at AFIT. Past thesis efforts include: design and application of an object oriented Graphical Database Management System[1]; statistical estimation techniques applied to terrain modeling[9]; and synthetic environments such as the Virtual Cockpit[34][27][19] and the Synthetic BattleBridge[14]. Current projects at AFIT involve building on the existing synthetic environments[10][12][32][36]. These projects are using new tools such as Performer and MultiGen. Most recently developed is a graphics interface system, ObjectSim[30], using Performer.

2.2. Graphics Issues

Terrain is generally modeled using adjacent polygons, creating a terrain "skin." Usually, the vertices of each triangle represent an elevation on the terrain, and the triangles interpolate the surface between elevations.

2.2.1 Levels of Detail

As mentioned in Chapter 1, rendering terrain has conflicting goals. LODs address the conflict between rendering speed and display detail. However, it also may add to the problem of memory usage.

Because terrain covers a large area, part of the terrain may be viewed up close, while other parts are viewed at a distance. Using the same number of polygons per unit area for the close and distant parts may overload the rendering system.

> If we are to provide a visually acceptable representation of the terrain within a computationally acceptable polygon load, we must employ a multiple level of detail (LOD) terrain strategy, where the distant portions of the terrain scene are built from progressively larger polygons.... The terrain will thus appear uniformly complex from near to far, and changes in the terrain appearance due to LOD transitions will be uniform in effect from near to far.[6]

Generally, terrain LODs are organized in circles or squares, creating rings around the viewpoint. Each ring is filled with a single LOD of the terrain.

2.2.1.1. Model Switching. In his paper, "Level-of-Detail Control Considerations for CIG Systems," Robert Rife presents an early LOD strategy. The strategy works with three LODs per object; for each object, the strategy uses three models of varying detail, switching between the models based on the object's distance from the view point. The strategy deals with 2-D and 3-D models. To implement the strategy, Rife defines formulas for calculating the distances for switching between each object's LOD models.

For 3-D models, the formula takes into consideration: the model's size, the tangent of the angle subtended by one raster element (pixel viewed by the user), and the minimum number of raster elements a model must subtend (or extend over) to be displayed at an LOD. For 2-D models, the formula additionally considers the user's altitude above the model.

The strategy also defines two squares (100 and 36 miles) around the viewpoint, to implement a basic culling scheme. The strategy only considers drawing large models within the large square and small models in the small square. (The small square is contained within the large square, so large models will be drawn in the small square.)[24]

2.2.1.2. Many Levels of Detail. Although Michael Cosman's paper, "A System Approach for Marrying Features to Terrain," discusses placement of features on terrain surface, it contains a more basic concept I wish to discuss: many LODs.

The paper presents formulas for determining the number of LODs necessary to display terrain to a specified distance from a viewpoint. The formulas use factors such as the size of polygons at the highest LOD (closest to the viewpoint) and the ratio of polygon edges between LODs.

The formulas demonstrate that the lower the ratio of polygon edges between LODs, the fewer total polygons needed for the display. As the ratio of polygon edges decreases, each new LOD can begin closer to the viewpoint. With larger ratios, each size LOD must be displayed to a further distance before switching to the new polygon size.

The paper also addresses the issues of memory and paging. A higher number of LODs means more information must be stored and quickly available. As a viewer moves through an area, the viewed terrain will change more quickly than with fewer terrain LODs, so paging is a bigger problem.[6]

2.2.1.3. Betweening Levels of Detail. Charles Clark and Michael Cosman's paper, "Terrain Independent Feature Modeling," also discusses placement of features on terrain surface, but again I wish to discuss its more basic concept: "betweening" LODs.

A switch between LODs may be noticeable by a viewer. One way to minimize LOD switch noticeability is to reduce LOD transition ranges by adding more LODs (discussed in [6] and [2]). The next step is "betweening," interpolating between LODs.

With betweening, "each vertex [or elevation point] can be defined as having an initial and final position." An LOD switch can take place over a period of time, where vertices are interpolated between (hence the name) the two positions. "This process is in general, powerful, and results in greatly reduced noticeability for LOD changes."[2]

2.2.1.4. Discussion of Levels Of Detail. Levels of detail address the conflict between rendering speed and display detail. By using LODs, a system can lower the polygon load while retaining detail in close areas.

At low altitudes (close to the surface), terrain is viewed as a 3-D entity, and Rife's 3-D formula can be used to determine the visual discrepancy between LODs. At higher altitudes, terrain is viewed as a 2-D entity, and Rife's 2-D formula applies. The visual discrepancy can be interpreted as the number of pixels a vertex might move during a LOD transition.

For rendering speed, the lowest number of polygons seems best. Using many LODs allows the lowest number of polygons to be rendered at any time. However, many LODs require additional overhead for processing and paging the LODs. A rendering system must address these issues to ensure managing the LODs does not slow down the system. Also, memory needs may increase to store the many representations of the terrain. A strategy to minimize memory use would also reduce paging.

2.2.2. Triangle Minimizations

In its simplest form, triangle minimization reduces the number of triangles in an object by removing vertex points. Triangle minimization strategies attempt to reduce the number of polygons in an object with minimal effects on the detail of the rendered objects; it is an effort to address the conflict between display detail and memory usage.

2.2.2.1. Re-Tiling Polygonal Surfaces. In his paper "Re-Tiling Polygonal Surfaces," Greg Turk "[p]resents an automatic method of creating surface models of several levels of detail from an original polygonal description."

Turk's method places new (candidate) vertices on the surface of an object and distributes them using a Point Repulsion algorithm. The vertices are moved, repelling each other, until they are uniformly distributed across the surface. Subsequently, the method connects the candidate vertices by creating a "mutual tessellation" of the original and candidate vertices, then removing the original vertices.

Uniformly distributed vertices may not capture detail in highly curved areas of a surface. Turk addressed curvature approximation by varying the Point Repulsion for vertices based on the curvature of the surface around the vertices. This strategy allows a denser concentration of vertices in higher curvature areas.

Turk's method may be used to create LODs. First, the method creates a low LOD model. Then, a new LOD model can be created by fixing the points in the low LOD model and adding new points. This process may be repeated for each new LOD model.

To interpolate between LOD models, Turk presents a method for fragmenting triangles between the models. Then, the fragments can be interpolated into the new LOD model.[35]

2.2.2.2. Decimation of Triangle Meshes. In their paper "Decimation of Triangle Meshes," William Schroeder, and others, present "an algorithm that significantly reduces the number of triangles required to model a physical or abstract object. The algorithm makes multiple passes over an existing triangle mesh, using local geometry and topology to remove vertices that pass a distance or angle criterion. The holes left by the vertex removal are patched using a local triangulation process."

The paper compares its algorithm to filter-based techniques such as subsampling and averaging, two "naive" approaches. Subsampling uses even spacing ("every n^{th} point"). Averaging resamples the data, using neighboring points. Filter-based techniques apply uniformly to the entire data set, so they fail to capture detail in high curvature areas.

Terrain modeled with this algorithm compares quite favorably over terrain modeled with filtering methods. This algorithm automatically keeps points in high curvature areas and discards points in flat areas. However, the algorithm may propagate some errors, because each iteration is compared to the previous iteration's model, not the original model.[26]

2.2.2.3. Mesh Optimization. In their paper "Mesh Optimization," Hugues Hoppe, and others, present another method of reducing vertices by minimizing "an energy

function that captures the competing desires of tight geometric fit and compact representation." During optimization, the function varies "the number of vertices, their positions, and their connectivity."

The method finds a new triangulation of lower complexity that is as close as possible to the original triangulation. This method "automatically retains more vertices in areas of high curvature, and leads to faces that are elongated along directions of low curvature."[16]

2.2.2.4. SGI Documentation Suggestions. SGI documentation suggests placing triangles into a triangle mesh. This mesh differs from what is described above as meshes. The triangle mesh contains a defined connectivity, such that the triangles can be placed in strips; adjacent triangles in a strip have two vertices in common. As a strip is rendered, the first triangle's vertices are processed. Then, the last two vertices processed are the first two vertices for the next triangle; each subsequent triangle needs only one more vertex to be processed. (Note: in a triangle strip, each vertex, except the first two and last two vertices, is shared by exactly three triangles.)

If a strip of triangles shares vertices, but their vertices cannot be ordered as above, the triangles still might be put into a triangle mesh with vertex swapping. For vertex swapping, the last two vertices processed are swapped, so the last vertex is only used by two triangles. The other swapped vertex may be used by four or more triangles.

SGI documentation discusses the hardware pipeline, which processes polygons and pixels in different steps. If the polygon count is low enough, the pixel processing is the bottle neck process. Therefore, polygon minimization is not always the highest priority. We might actually get more detail and complexity with no loss in rendering speed.[29][28]

2.2.2.5. Discussion of Triangle Minimization. With virtual flight environments, these absolute polygon minimizations are not the best option, because they may produce triangles of unconstrained size. Levels of detail require the polygons be

organized in such a way as to vary polygon size based on the distance from the view point. Unconstrained polygons do not fit well into LOD strategies.

Virtual flight environments need a guaranteed update rate. High frequency areas may need to be cut back to keep the frame rate; however, low frequency areas may be left with more information than needed, because they will not harm the guaranteed frame rate (established for the high frequency areas).

These triangle minimization techniques are good for minimizing triangles, but not necessarily minimizing the amount of memory needed to store the objects. Because connectivity is not consistent, information on connectivity must be maintained with the vertices. If connectivity of vertices is inherent in the organization of the data, the connectivity information does not need to be stored in memory.

Turk's method provides LODs. However, the intermediate model requires breaking the triangles into "fragments" to interpolate between levels, creating two problems: the fragments require additional overhead for a system to manage them, and the fragments create additional polygons to render.

Triangle meshing (with strips) allows more detail without sacrificing rendering speed. By creating a method that minimizes triangles while maintaining a defined connectivity, we would be able to increase rendering speeds.

The Turk or Hoppe methods could be modified to minimize polygons while maintaining a consistent connectivity of the vertices. This strategy will produce more polygons (than without consistent connectivity) but would also reduce memory needs and increase rendering speed.

2.3. Software Engineering Issues

2.3.1. Characteristics

A goal of applying software engineering to this effort is to create software which has favorable internal and external characteristics. The internal characteristics, as seen

through the eyes of the programmer, are understandability and maintainability. Internal characteristics "are considered to be the key to improving software quality."[11] The external characteristics, as seen through the eyes of the user (pilot or battlefield commander) are detail and update rate. "Quality is perceived in terms of those external [characteristics] which are relevant for particular types of users."[11]

Two general programming concepts that lead to favorable internal characteristics are abstraction and encapsulation (also known as information hiding).

Abstraction is:

> The principle of ignoring those aspects of a subject that are not relevant to the current purpose in order to concentrate more fully on those that are.[4]

There are two aspects of abstraction. First, abstraction aids handling complexity by breaking a system into smaller pieces. Second, it aids handling complexity by ignoring details where it is not necessary to consider them.

Breaking a system into smaller pieces enhances understandability and maintainability, when the pieces (represented as modules) reflect real-world objects or concepts. The system is broken along intuitive boundaries; each module has an intuitive, real-world function.

Once the system is abstracted into real-world pieces, those pieces fit in at different detail levels. Such a structure makes it possible to change the detail level of a piece as applicable.

Encapsulation might be better known as "tight cohesion" and "loose coupling." Encapsulation is:

> A principle, used when developing an overall program structure, that each component of a program should encapsulate or hide a single design decision [i.e., tight cohesion].... The interface to each module is defined in such a way as to reveal as little as possible about its inner workings[i.e., loose coupling].[4]

Encapsulation is really an extension of abstraction. Abstraction breaks the system into pieces, and encapsulation isolates each piece. "Encapsulation prevents a program from becoming so interdependent that a small change has massive ripple effects."[25]

Encapsulation is useful to isolate parts of a system which may change. The system is more maintainable, because when part of the system changes, the software affected by the change is isolated. Encapsulation is also useful to isolate system dependent parts of a system. The system is more reusable, because the non-system dependent software can be moved easily without concern for the system dependencies; the system dependent software affected by the move is isolated.

The external characteristics reflect the desired performance of the system. Although we attempt to maximize performance, we begin by addressing the internal characteristics. The internal characteristics can be more readily addressed during system design than the external characteristics. After an initial system implementation, we can measure its performance and attempt to improve its performance.

Certain aspects of external characteristics should be addressed during design. For example, the choice of data structures has a substantial impact on performance.

2.3.2. Metrics

Given the desirable characteristics of a system, we need a method to determine how well we have attained them. Metrics give us a means to measure the desirable characteristics.

2.3.2.1. Internal Characteristics. The internal characteristics considered for this project are complexity, coupling and cohesion. The metrics used to measure these characteristics are discussed in more detail in Section 5.1, Software Engineering Metrics.

Complexity. "Complexity is commonly used as a term to capture the totality of all internal [characteristics]."[11] Thomas McCabe proposed a complexity measure on a control flow representation of a program.[17] McCabe's cyclomatic complexity provides a quantitative measure of a program's logical complexity.

Coupling. "Coupling is a measure of interconnection among modules in a program structure."[23] Measuring coupling indicates the complexity of the interfaces between modules. Coupling should be considered at two levels: between classes and between the functions within a class.

Cohesion. "Cohesion is the measure of the strength of functional relatedness of elements within a module."[22] Measuring cohesion indicates how well a module hides a single concept. Cohesion should be considered at two levels: by class and by function within classes.

2.3.2.2. External Characteristics. The external characteristics considered for this project are frame rate, memory use, paging needs, and conversion processing time. Frame rate is the main issue in graphics rendering. However, a consistent frame rate depends on other issues; the frame rate may be slowed by paging and conversion processing. The amount of paging needed depends on the amount of main memory available and disk storage space needed to store the terrain.

The measurements considered below compare converting and rendering terrain from the new system against rendering terrain already converted into Performer. Conversion of another format such as MultiGen Flight into Performer is not considered, because this conversion generally occurs before the rendering process begins. The following measurements are considered:

Memory Usage. Compare the memory used to store terrain information for the new system against a similar terrain set built in MultiGen (discussed below). Compare the memory used to manage the terrain information for the new system against the equivalent terrain set already converted into Performer.

Paging Needs. Compare the paging and pre-paging needs for managing the terrain information for the new system against the equivalent terrain set already converted into Performer.

Conversion Processing Time. Measure the processing time required to convert the terrain information for the new system into the equivalent terrain in Performer. Compare the paging and conversion processing needs for terrain information for the new system against the paging needs for the equivalent terrain set already converted into Performer.

Frame Rate. Compare the frame rate for terrain converted from the new system against similar terrain converted from MultiGen. (Each storage format converts into a Performer database for rendering.)

2.4. Past Thesis Efforts

In 1991, Capt John Brunderman designed and implemented and object oriented Graphical Database Management System (GDMS) to support research sponsored by Rome Laboratories. The GDMS provided "data structures, file formats and algorithms to manage and render hierarchical, three-dimensional, polygonal models." To demonstrate the functionality of GDMS, he also developed the DataBase Generation System (DBGen) which allowed users "to orient, scale, move, delete and add multi-resolution objects to synthetic environments interactively."[1]

At the same time, Capt Donald Duckett investigated "methods of generating accurate and realistic polygonal terrain models by reducing sampled terrain elevation data such as DMA DTED." He compared kriging, a geostatistical estimation technique, to filtering methods of data reduction. His goal was to create an estimation method that could "build polygonal terrain models at any resolution."[9]

In 1991, the Defense Advanced Research Project Agency (DARPA) sponsored AFIT to investigate the possibility of a low cost virtual cockpit. The Virtual Cockpit was developed in three functional components: flight dynamics, tracking vehicle status of other simulators, and display of the out-the-window imagery. The components were implemented by Capts John Switzer, Steven Sheasby, and Dean McCarty, respectively.

The out-the-window display used the rendering software developed by Capt Brunderman for DBGen.[34][27][19]

As production progressed on the virtual cockpit, Capt Rex Haddix developed the Synthetic BattleBridge (SBB), "designed to provide the user with a synthetic three-dimensional view of moving and stationary vehicles dispersed over a hundred thousand cubic mile volume." The SBB also used the rendering software developed by Capt Brunderman for DBGen.[14]

2.5. Current Work

Current work on the Virtual Cockpit and Synthetic BattleBridge involve Performer, ObjectSim, and MultiGen Flight format. ObjectSim, with Performer, replaces the previously used rendering software. Tools developed for Flight format provide database generation and maintenance to replace GDMS and DBGen.

Performer is an application development environment developed by Silicon Graphics. Performer provides a software tool kit for creating visual simulation applications. Silicon Graphics developed Performer to aid graphics programmers in using their IRIS workstations, to effectively use the workstations' rendering capabilities. Performer creates its own run-time data base which it manages for high-performance rendering.[29]

Capt Mark Snyder developed ObjectSim, an object oriented architecture that uses Performer to render graphics. ObjectSim provides an interface between players and terrain.[30]

Currently, we use software tools such as makeTerrain12[18] to create Flight format files. We use Performer to convert the entire Flight format database and manage the data real time. That is, Performer handles database culling steps, paging and displaying in its own format.

2.6. Summary

Many graphics and software engineering issues are involved with rendering terrain. The graphics issues consider level of detail and memory minimization techniques. Software engineering looks at these issues from the perspective of understandability and maintainability, as well as efficiency of the software.

In this thesis, I attempt to address these issues by creating a terrain management system which maximizes rendering speed while minimizing memory usage.

III. The Data Structure

Researchers at AFIT have experienced problems in creating, managing and rendering terrain data stored in Flight format. Flight format and our tools to create terrain in Flight format do not handle large data sets well. Creation of large Flight format terrain files is processing and memory intensive. Conversion of large Flight files to Performer is processing and memory intensive.

Flight format is an organizational format aimed at easy manipulation of the data, but not at fast rendering. It is used as a storage format to store data on disk. Performer format is aimed at fast rendering. Performer maintains its own format of the data for rendering from main memory. Conversion from Flight to Performer is slow. Both formats waste memory storage space by replicating data for easier management. For small data sets, it is reasonable to convert the entire data set into Performer before the run-time rendering occurs. For large data sets, this conversion takes too much up front time and memory, especially if only a small portion of the data set may be needed. When the environment does not know what section(s) of the data set will be used, up front conversion might not be a reasonable option.

What if we can store terrain data in a more compact form, converting to Performer only what is necessary for the current view? An ideal format would be one that minimizes main memory and disk storage needs, minimizes paging, supports levels of detail, supports triangle meshing, and can be quickly converted into Performer format. Effectively, the new format would trade high memory usage and paging for higher computational needs.

This chapter presents a data structure to store terrain information. Many factors play into a choice of data structure. The factors affect the amount of memory needed to store the terrain information, the time needed to access the data, the time needed to convert the data into a rendering format, and the speed with which the data may be

rendered. First, these factors are discussed. Then, a data structure is presented. Finally, issues concerning the data structure are discussed.

3.1. Set Up

The goal of this chapter is to define a data structure for storing terrain information which allows the fastest update rate with the most amount of detail. The data structure considers only the polygon representation of the terrain. It does not consider the storage of cultural features and texture maps for the terrain. These items should be minimized by replication. However, the terrain polygon description varies so much across the globe, that it cannot be simply represented by replications. The goal is to store and access the terrain data in such a way that it does not hinder (although it might help...) the rendering process.

"[T]he most important property of a real-time system should be predictability; that is, its function and timing behavior should be as deterministic as necessary to satisfy system specifications."[33] In a real-time graphics system, the desired predictable behavior is a consistent update rate. Basically, an application has a stated minimum acceptable update rate. Then, detail is added until the load jeopardizes the update rate.[24] Hence, this problem was approached from the point of view of increasing detail while maintaining the minimum update rate; we always want to do more than we can, so where do we compromise?

In creating the terrain data structure, several factors were considered (from Section 2.2, Graphics Issues):

Update rate:

- Level of Detail (LOD) strategies
- Triangle Meshing

Access to Data:

- Prepaging

- Caching

The basis of the terrain data structure is storing terrain data in a format which allows for quick access (paging and caching) while allowing levels of detail and triangle meshing. The strategy should also allow for betweening. However, betweening is an aesthetic feature which may harm the update rate. Yet, if betweening can be accomplished without adversely affecting the update rate, the data structure should support betweening.

The terrain data used for this thesis were taken from storage formats (as opposed to rendering formats) which store elevation points across a terrain area. The elevation points are spread across the terrain in a relatively evenly spaced grid. The earth is viewed as a smooth sphere or flat plane whose surface can be perturbed by adding altitude information. That is, imagine the sphere or plane as having an elastic surface. By picking specific points and assigning altitudes to them, the points may be moved away from the original surface to approximate the variations in altitude on the earth. The elastic interpolates the surface between the elevation points.

In graphics rendering, the interpolation of the surface between elevation points must be approximated by breaking the surface into distinct polygons, with the vertices of these polygons defined by the elevation points. If the elevation points are evenly spaced, then each point may be connected to its closest neighboring points, forming a grid of squares. However, the four points in each square may not be coplanar, so they need to be further broken into triangles by connecting two opposing vertices. The choice of which vertices to connect may be arbitrary and may not give the most accurate representation of the earth.

For example, consider nine points, as in Figure 3.1. The points might be connected as shown in Figure 3.2.

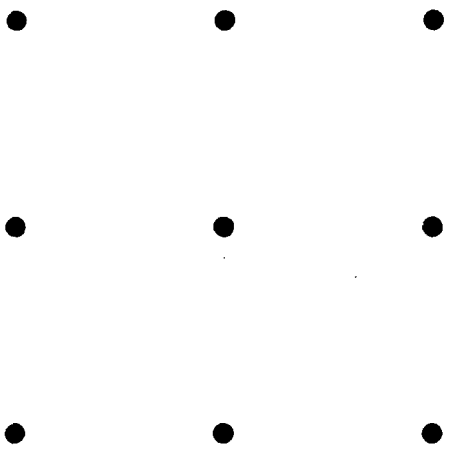
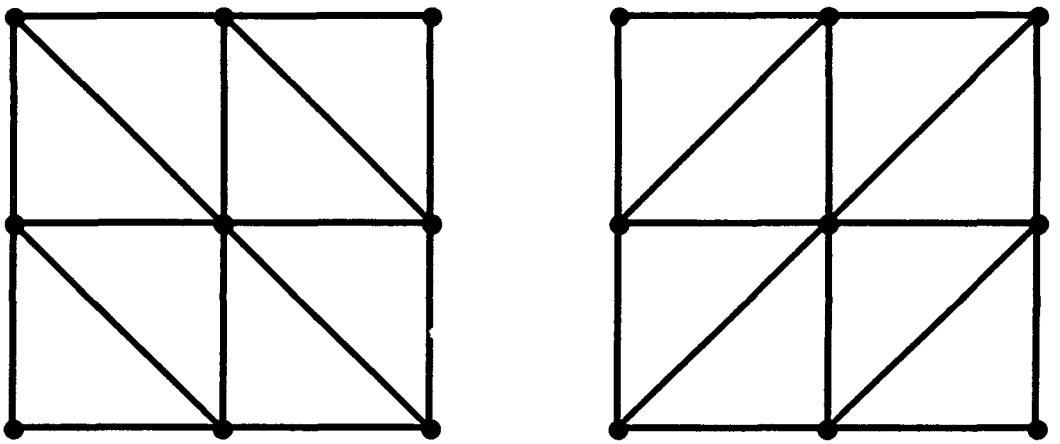


Figure 3.1. Evenly Spaced Grid of Nine Points



(a) (b)

Figure 3.2. Connectivity Examples of Evenly Spaced Grid

Now, imagine the points defining posts on a surface, where the post goes from the surface to an elevation above (or below) the surface, as in Figure 3.3. Suppose the actual terrain represented by these elevations is a valley, running from the near right corner to the far left corner. Following the point connections as in Figure 3.2(a), Figure 3.4(a) accurately represents the valley. However, following the point connections as in Figure 3.2(b), Figure 3.4(b) does not accurately represent the valley. Specifically, the diagonal edges of the cross shaded triangles cut across the valley, and the terrain represents a crater more than a valley.

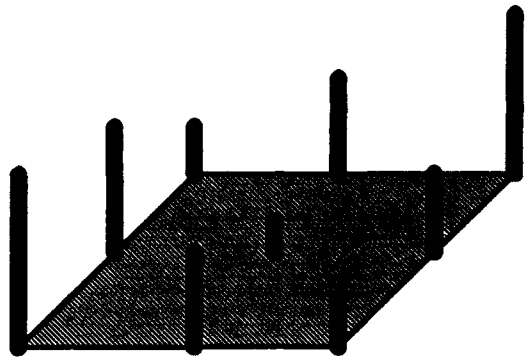
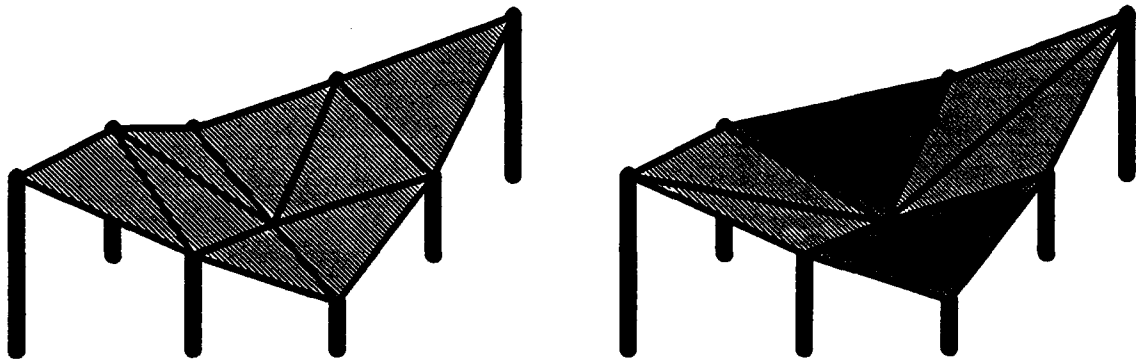


Figure 3.3. Evenly Spaced Grid Points Shown as Elevations



(a) (b)

Figure 3.4. Connectivity Examples of Evenly Spaced Elevations

Because the points can be connected, based on their locations relative to each other, the points can be easily stored without explicit edge (i.e., vertex connection) information. That is, the connections can be inferred from the storage structure.

As mentioned earlier, most terrain storage formats store elevation points across a terrain area, with these elevation points spread across the terrain in a relatively evenly spaced grid. (Spacing between points might be measured evenly in meters or relatively in degrees and minutes.) The data can be stored in a compact manner as just a list of elevations, along with a few values to indicate where a corner of the data lies in real world coordinates and what the spacing is between elevations.

The ordering of the points is standardized, so their placement on the earth's surface can be computed. This storage format resembles a two dimensional array stored in

linearly addressed memory. Each column or row is stored contiguously in the linear memory. In this manner, the points can be numbered with array indices, as in Figure 3.5. The points can be connected as in Figure 3.2(a) by defining two triangles for each (i, j) as: (i, j), (i, j+1), (i+1, j+1); and (i, j), (i+1, j), (i+1, j+1).

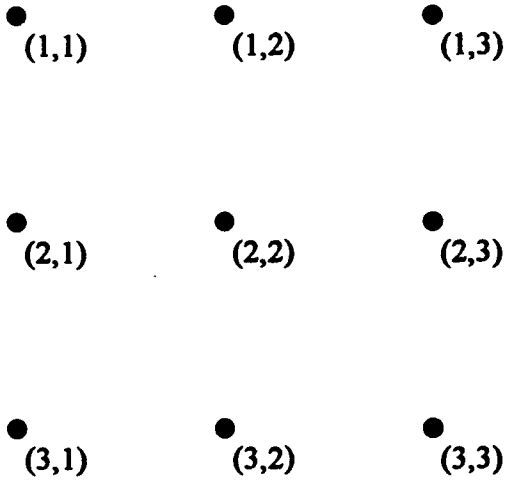


Figure 3.5. Array Indices for Points in a Grid

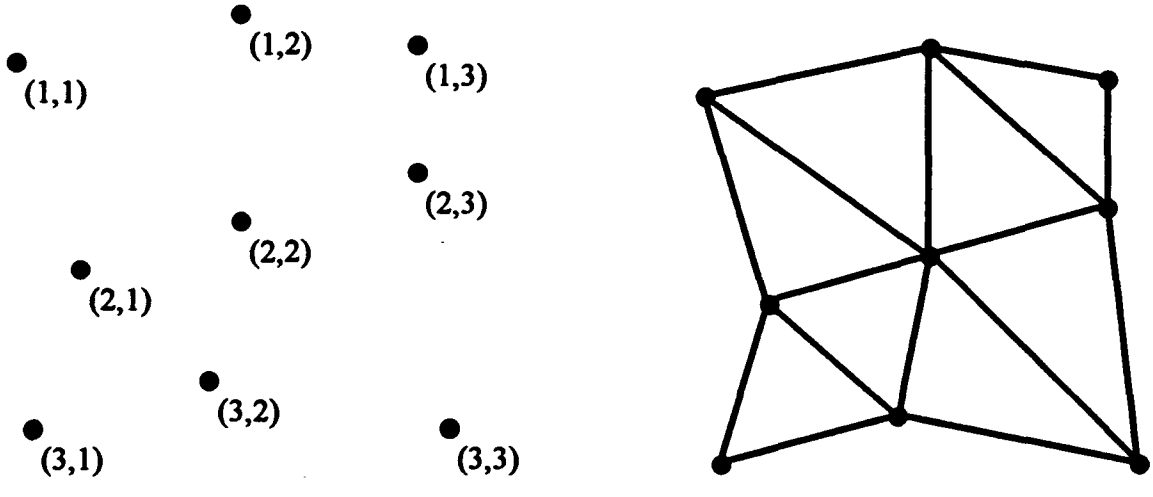


Figure 3.6. Array Indices and Connectivity Example of Unevenly Spaced Grid

If the elevation points are not evenly spaced, connecting them falls into two categories. (1) The points might be organizable into an array equivalent structure, such as Figure 3.6(a). In this case, each point's placement on the earth must also be stored, but

connection of the points can still be inferred from the data structure, as in Figure 3.6(b). (2) Or, the points might be sporadically placed, such that their connection cannot be inferred from the data structure. In this case, the connection of the points must be stored in the data structure or computed. For example, the data might be derived from an evenly spaced data set, with some points removed, as in Figure 3.7. Two possible connections of points are shown in Figure 3.8.

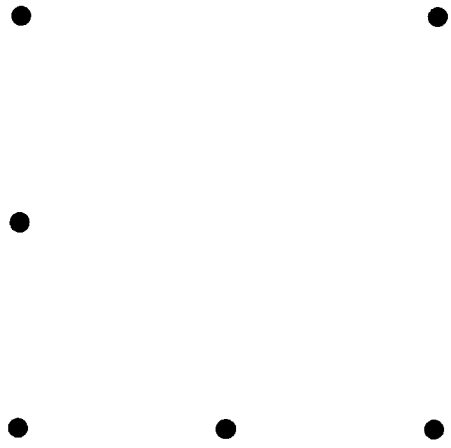
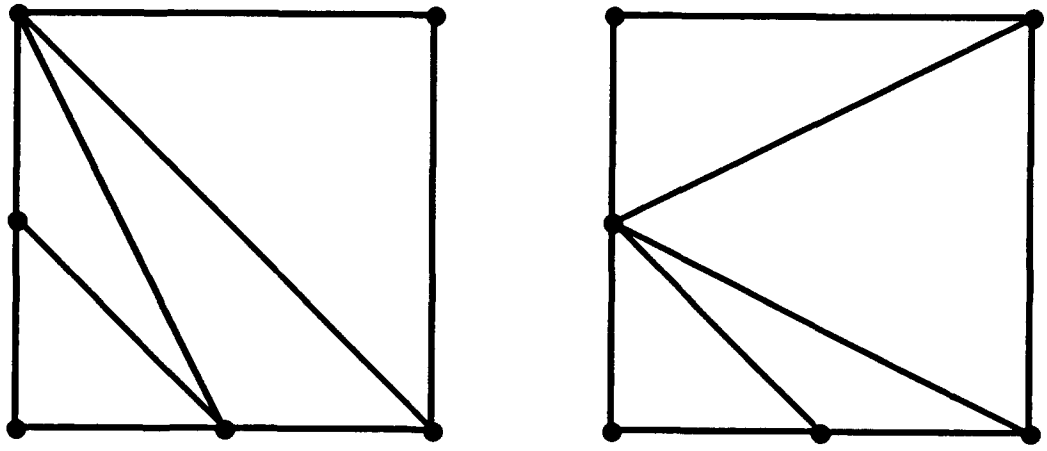


Figure 3.7. Sporadically Placed Points



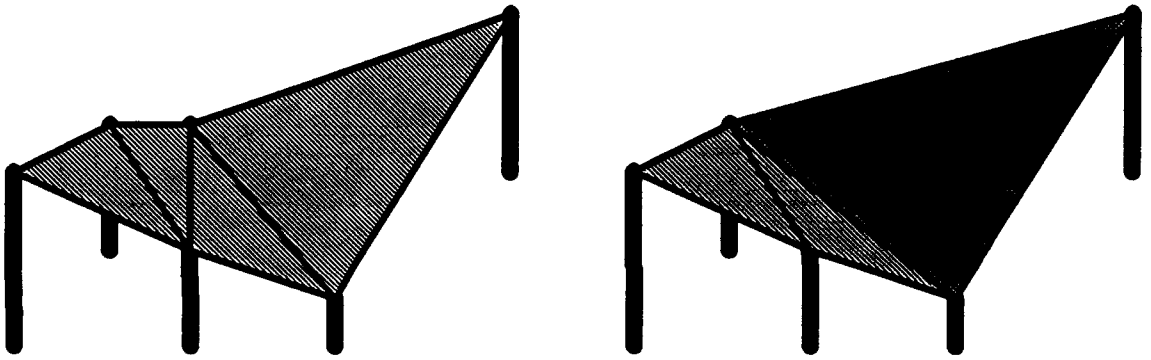
(a) (b)

Figure 3.8. Connectivity Examples of Sporadically Placed Points

Sporadically placed points usually are obtained through Triangle Minimization strategies, as discussed in chapter 2. Although methods for traingulating sporadically

placed points exist, such as the Delaunay method[8], such methods are not practical for real-time rendering. Additionally, such triangulations may not conform to the original terrain surface. For example, suppose the points in Figure 3.7 were taken from the data set represented by Figure 3.3. Recall that Figure 3.3 represents a valley running from the near right corner to the far left corner.

The connection of points from Figure 3.8(a) is applied to Figure 3.9(a), which accurately represents the valley. The connection of points from Figure 3.8(b) is applied to Figure 3.9(b), which does not accurately represent the valley. Specifically, the far edge of the cross shaded triangle cuts across the valley, creating a ridge rather than a valley.



(a) (b)

Figure 3.9. Connectivity Examples of Sporadically Placed Elevations

With all of this in mind, what is the best way to store and retrieve terrain descriptions for real time rendering of terrain? The storage strategy should consider access to the data as well as the update rate obtainable with the data. The data should be as compact as possible, allowing for fast paging. The data should be organized to minimize caching. The storage strategy should consider the advantages of levels of detail and triangle meshing.

Consider the current terrain rendering of the Virtual Cockpit and Synthetic Battle-Bridge applications, discussed in Chapter 2. The terrain data is stored in the MultiGen Flight format. When either application runs, it uses Performer to convert the Flight format

data into Performer's internal (real time) format. Once converted, the data is managed (paged and culled) by Performer. This strategy works well for small terrain databases. However, Performer's conversion and management may not be the best solution for large databases.

A large database must be converted to the Performer format either at the beginning of the application's execution (up front) or during the application's execution (real time), as needed. Up front conversion might take an unreasonable amount of time, especially if only a small portion of the terrain is actually used. Also, up front conversion requires duplicating the Flight format data into Performer format, taking more memory. Real time conversion requires additional real time computing, which may affect the update rate.

For example, consider an area defined by 129 by 129 posts in Flight format without levels of detail. To allocate memory for the Performer structures and convert the Flight format data into Performer format requires at least 15 seconds on an ONYX Reality Engine2, and at least 30 seconds on an IRIS 4D/440VGXT. If a one degree cell of DMA DTED data were stored in sections of this size, the entire area would require over 20 minutes to load on an ONYX, and over 40 minutes to load on an IRIS 4D. (These statistics account for allocating memory for the Performer structures and converting Flight data into Performer. They do not consider time required for paging if data needs exceed main memory.)

Additionally, both the Flight and Performer formats contain redundant information in their storage of LODs. In both formats, each LOD is stored separately, repeating information from lower LODs. Each time a new LOD is paged in, the entire LOD must be paged in. Given the expense of paging versus computing, there might be a better scheme. Also, within LODs, each format contains redundant information, if it stores each vertex's information more than once. The format might store each vertex's information only once, using an integer to index into a "pool" of vertices, thus decreasing the size the data

storage. However, storing each vertex's information only once may slow rendering due to increased caching, because each vertex may need to be cached individually.[28]

A reasonable strategy might be to store the terrain data in a format which is quickly paged, minimizes caching, and is quickly convertible into the Performer format. If the area of the database currently converted into Performer fits in main memory, then paging during rendering is not a concern. Moreover, the Performer format should not use indexing. The waste of space for storing vertices multiple times is traded for rendering speed.

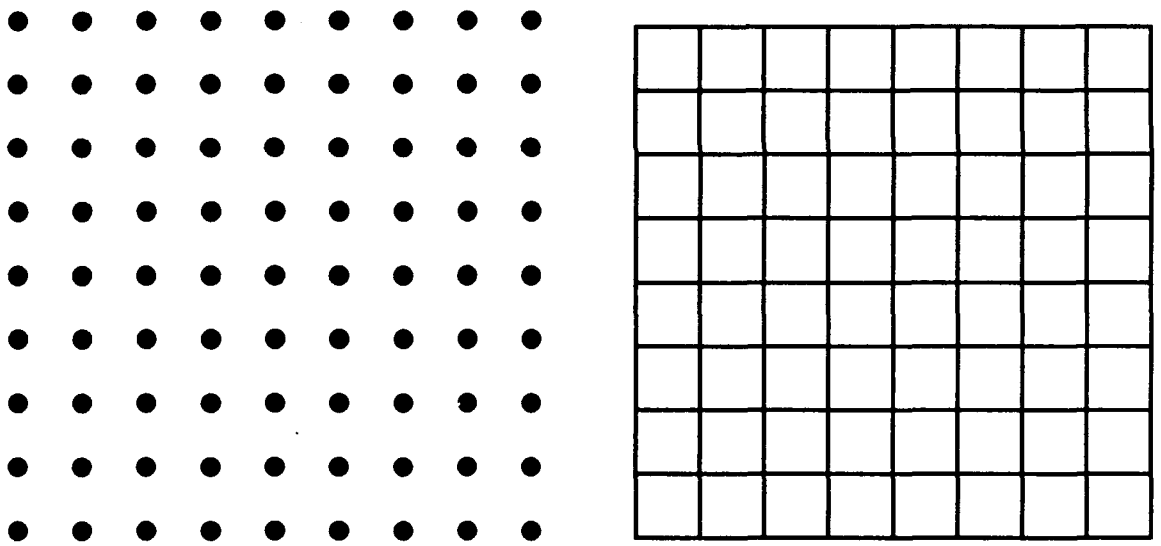
The main goal is to have the necessary information in memory when it is needed, in such a format as to give the fastest rendering. If paging is slow, the first solution is to pre-page what might be needed in the near future. That is, if the viewer is approaching an area, page in the data for that area before it is needed. When a viewer moves through the terrain in non-deterministic manner, the next areas needed cannot always be accurately predicted. A pre-paging algorithm would need to page in much more data than is actually needed.

I considered storing terrain data that had been processed by triangle minimization techniques. However, storing this data has three problems associated with it. First, triangle minimized data does not support levels of detail. The unbounded triangle size of triangle minimized data means a triangle could span more than one level of detail area. Second, triangle minimization does not support triangle meshing. For triangle meshing, the triangles must be organized in strips of triangles with shared vertices. Third, triangle minimized data does not support texture mapping of the terrain surface. Points within a texture map must be mapped to vertices of a polygon. The unbounded triangle size of triangle minimized data means a texture may be spread over a large area. Thus, detail in texture mapping may require more polygons than would otherwise be needed to accurately represent the earth's surface.

3.2. *The Strategy of the Structure*

So, what format might decrease paging and caching in exchange for increased computational needs? This section defines a format, starting with points which are either evenly spaced, or sporadically spaced but organizable in an array structure. The strategy is to store the terrain data in a format which is quickly paged, minimizes caching, and is quickly convertible into the Performer format. The data are organized into units of bounded size. Units are pieced together to form a representation of the terrain being modeled. Each unit may be represented in its own level of detail, depending on its distance from the viewer.

Consider a unit of 81 points, as in Figure 3.10(a), organized such that they represent 64 quadrilaterals, as in Figure 3.10(b).

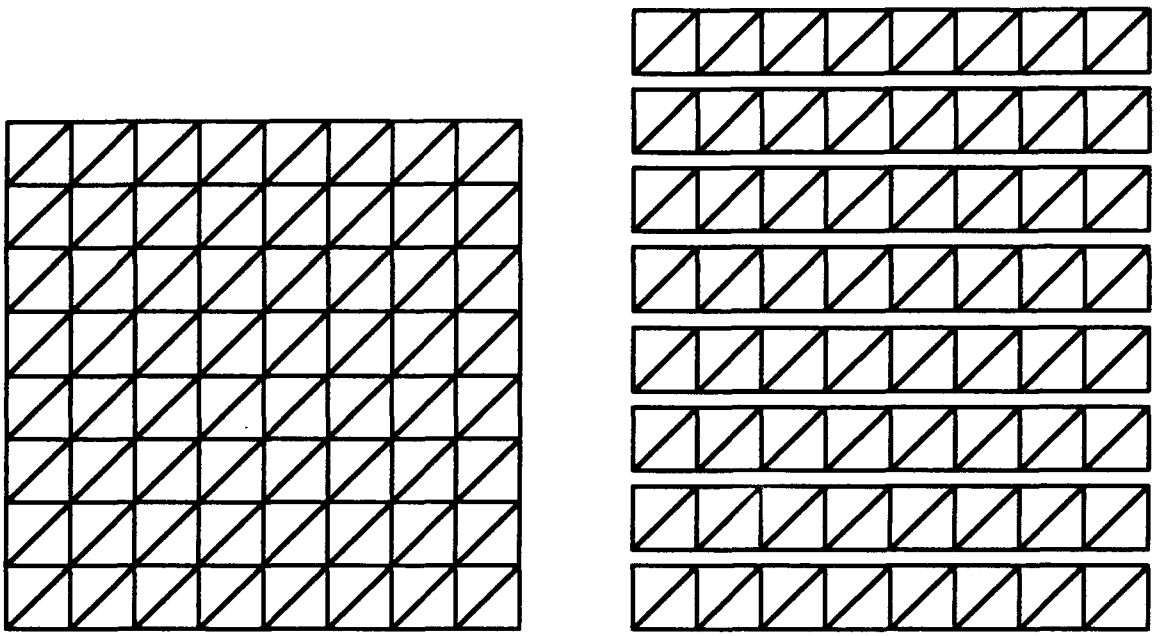


(a) (b)

Figure 3.10. Grid of 81 Points Representing 64 Quadrilaterals

The point information may be stored in one array, with each row of points stored together. Thus, nine rows of points will define eight rows of quadrilaterals. The eight rows of quadrilaterals can be described in Performer format as eight triangle meshes. The triangle meshes can easily be created by stepping through two rows of points and storing

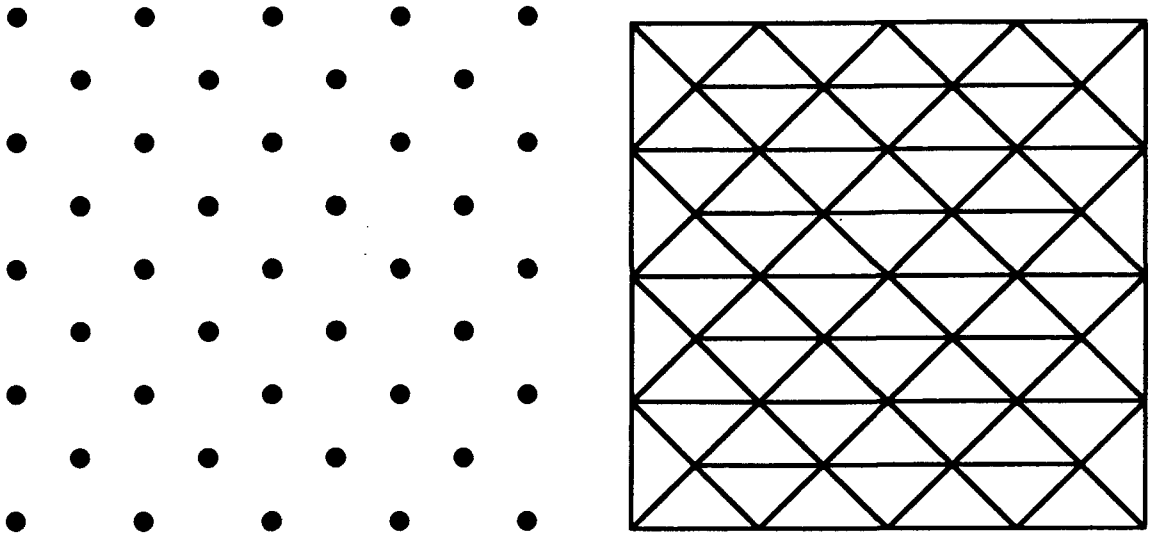
them as a triangle mesh in Performer. The triangulation of the unit shown in Figure 3.11(a) can be made into triangle strips as shown in Figure 3.11(b).



(a) (b)

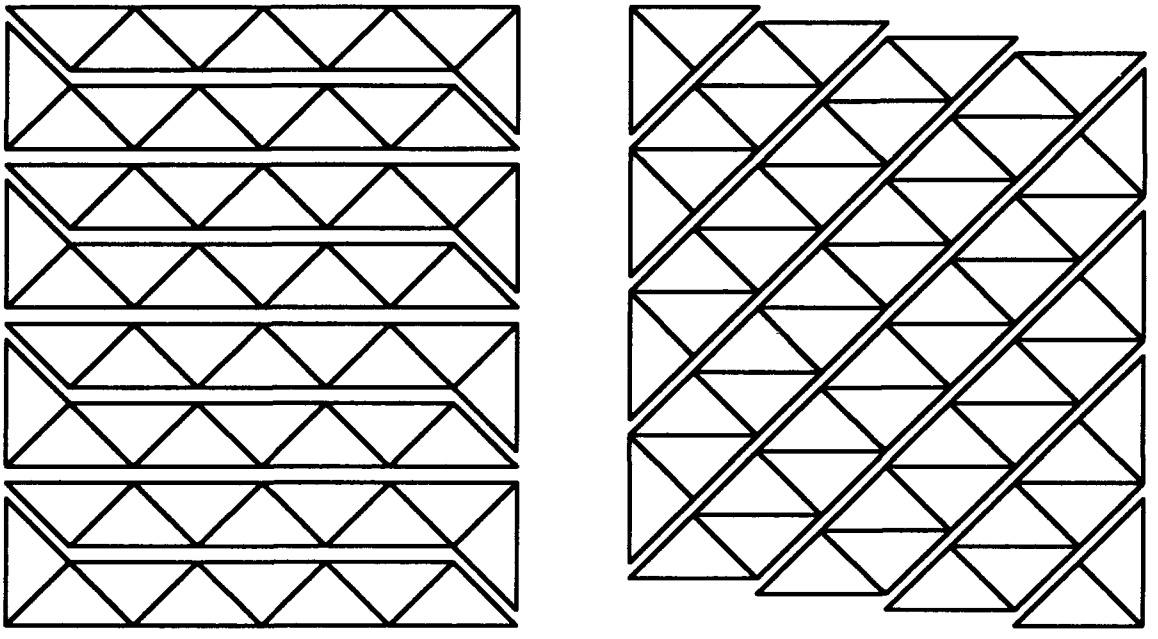
Figure 3.11. Example Connectivity of 81 Grid Points and Triangle Strips

As discussed in Chapter 2, [6] advocates a low ratio of the polygon edge lengths between LODs. A ratio of $\sqrt{2}$ between polygon edges in the first and second LODs can be obtained by removing every other point, as shown in Figure 3.12(a). These points might be connected as shown in Figure 3.12(b). The triangulation of the unit shown in Figure 3.12 can be made into triangle strips as shown in Figure 3.13.



(a) (b)

Figure 3.12. Grid of 41 Points, with Example Connectivity

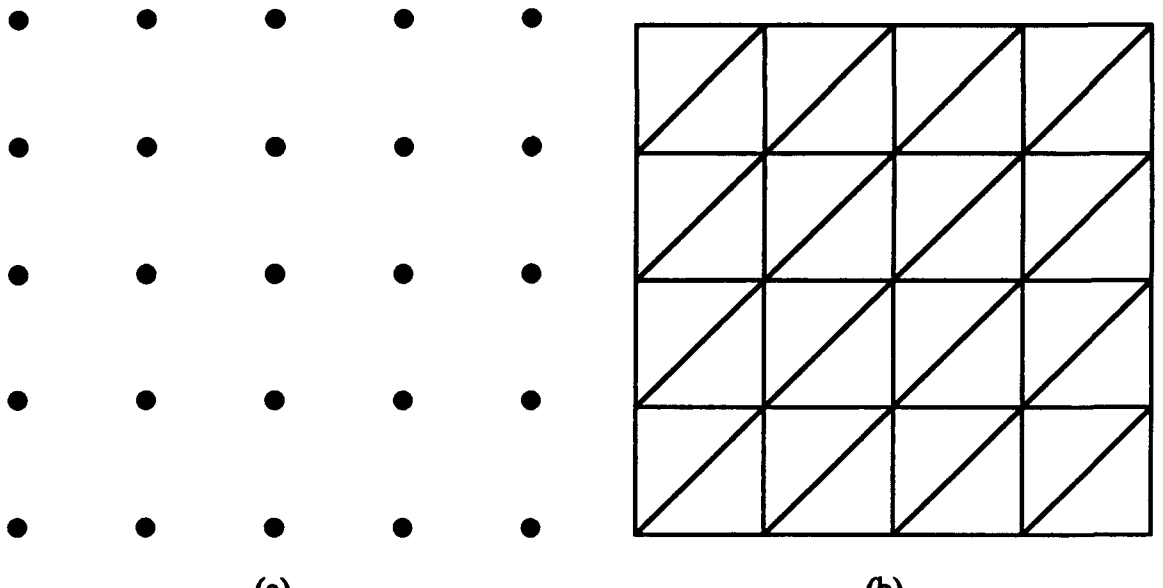


(a) (b)

Figure 3.13. Example Triangle Strips of 41 Grid Points

Again, a third LOD with an edge length ratio of $\sqrt{2}$ (between the second and third LODs) can be obtained by removing every other point, as shown in Figure 3.14(a). (In this case, which points represent the "every other point" is not as obvious as in the

previous example. "Every other point" refers to the order of the points as they appear in a triangle strip of the higher LOD.) These points might be connected as shown in Figure 3.14(b). The triangulation of the unit shown in Figure 3.14 can be made into triangle strips as shown in Figure 3.15.



(a) (b)

Figure 3.14. Grid of 25 Points, with Example Connectivity

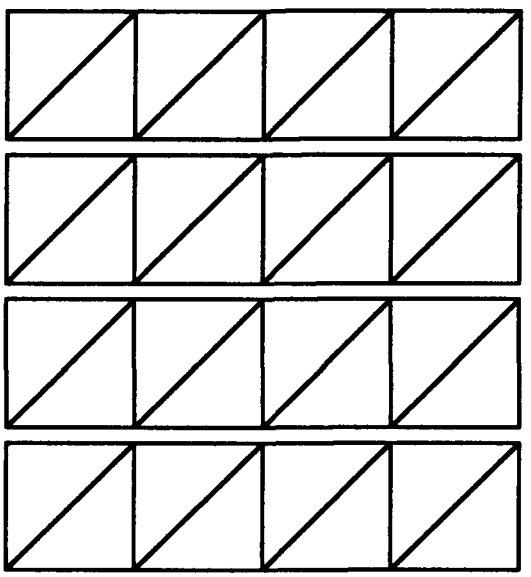


Figure 3.15. Example Triangle Strips of 25 Grid Points

As a final example, a fourth LOD with an edge length ratio of $\sqrt{2}$ (between the third and fourth LODs) can be obtained by removing every other point, as shown in Figure 3.16(a). These points might be connected as shown in Figure 3.16(b). The triangulation of the unit shown in Figure 3.16 can be made into triangle strips as shown in Figure 3.17.

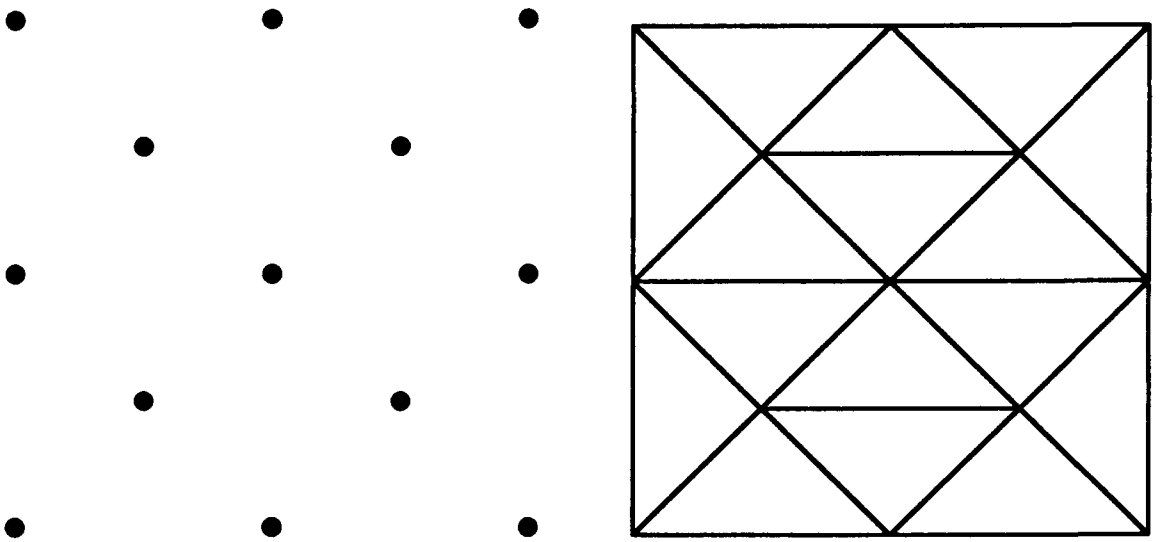
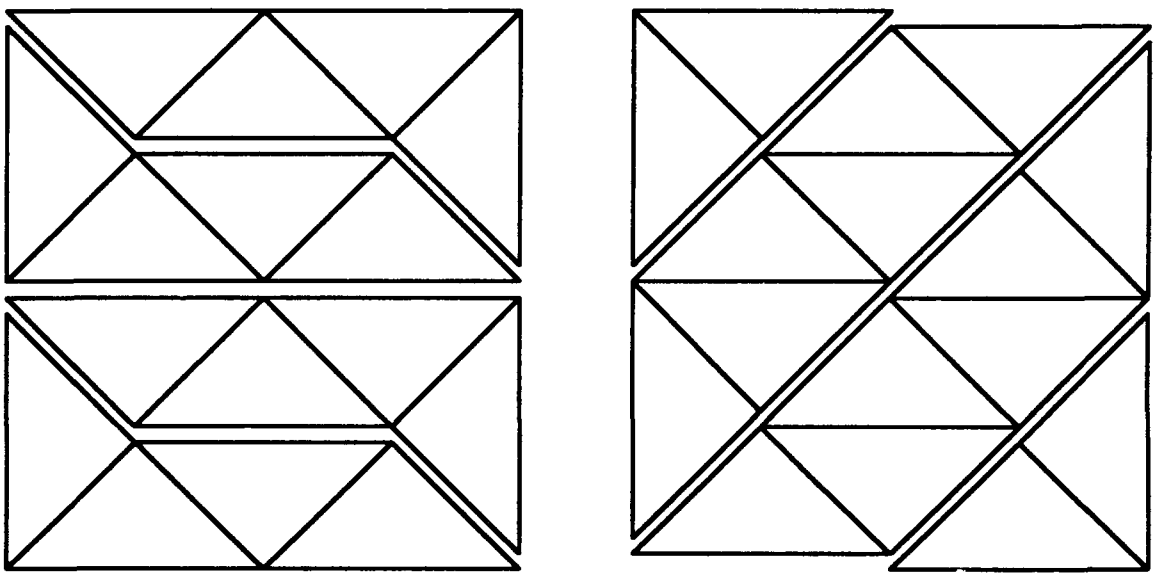


Figure 3.16. Grid of 13 Points, with Example Connectivity



(a) (b)

Figure 3.17. Example Triangle Strips of 13 Grid Points

Of course, if all 81 points were stored as one unit, then each LOD of a unit would require the entire unit be in memory. Having the entire unit in memory requires more paging than is necessary. Also, since the each LOD other than the first would only need every other point or every fourth point in a row (it might skip some rows entirely), an additional caching penalty would be incurred. The algorithm should have available only the data that is needed for the current LOD.

3.3. Defining the Data Structure

Consider the unit LOD shown in Figures 3.14 & 3.15. If its 25 points were stored in one structure (such as an array), then the next LOD as in Figures 3.12 & 3.13 could be created by adding an "every other points" array of 16 points. To build this LOD's Performer structure, an algorithm would need to step through each array's points concurrently. Then, the LOD in Figures 3.10 & 3.11 could be created by adding an "every other points" array of 40 points (or two arrays of 20 points each). To build this LOD's Performer structure, an algorithm would again need to step through each array's points concurrently.

Stepping through multiple arrays concurrently should not increase caching needs, if the cache has multiple caching locations. However, there is a substantial decrease in paging needs. For each of the three LODs discussed, the algorithm only needs to page in the data it will use, nothing more.

The data structure is defined as an array of points in sections. In the example above, three sections are used. The first section defines the points in the third LOD representation of the unit. The first and second sections together define the second LOD. All three sections together represent the first LOD. This is a single data structure, with many possible interpretations to connect the points into triangles.

This data structure is extensible in three ways. First, the structure may be extended for additional LODs; the first array might be broken into smaller sections. Again

with the example above, the array of 25 points could be broken into two arrays of 13 and 12 points each. The 13 point array represents a fourth LOD, as in Figures 3.16 & 3.17.

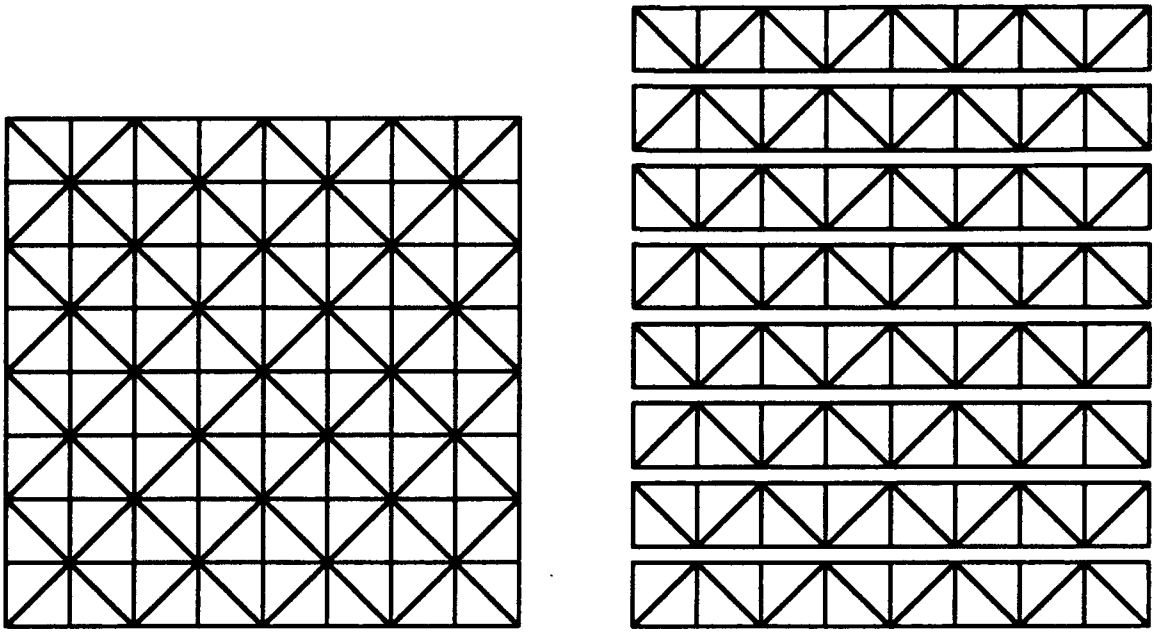
Second, the structure may be extended by using more points. For example, a 16 by 16 grid of quadrilaterals is represented by 289 points for the first LOD. The second LOD uses 145 points, and the third LOD uses 81 points.

Third, the structure may be applied in levels. If more LODs are needed than can be obtained from a unit, larger units may be used. For example, a fifth LOD may be represented by the first LOD of a unit which uses every fourth point (of every fourth row) from the original grid.

3.4. Another Interpretation of the Structure

The triangulations shown in Figures 3.11 through 3.17 show many interpretations of the data structure defined in the previous section. Each of those triangulations was aimed at triangle meshing without vertex swapping.

Here is another way of interpreting the structure. The 64 quadrilaterals of Figure 3.10(b) might be divided into triangles as shown in Figure 3.18(a). The triangulation of the unit shown in Figure 3.18(a) can be made into triangle strips as shown in Figure 3.18(b).

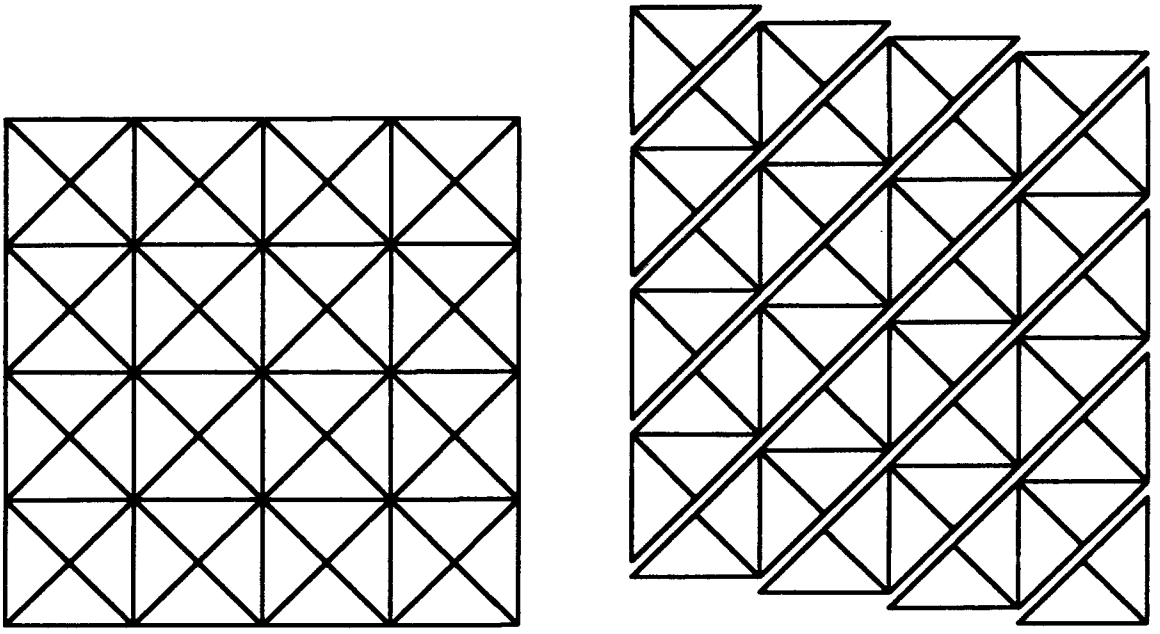


(a) (b)

Figure 3.18. Connectivity of 81 Grid Points and Triangle Strips with Vertex Swapping

The main difference with this interpretation is the direction of the diagonals in the triangulation of the unit. The diagonals are no longer parallel, but they cross. In terms of rendering, this difference means the rendering process must "swap" vertices between triangles in the mesh. The swapping process may have an impact on rendering speed.[28] However, this triangulation has two distinct advantages: it maintains edge lines between levels of detail, and it maintains edges in four directions (vertical, horizontal, and two diagonal).

Consider the next level of detail in Figure 3.19(a), and its triangle strips shown in Figure 3.19(b). The difference between Figure 3.18 and Figure 3.19 is half the vertical and horizontal edges have been removed.



(a) (b)

Figure 3.19. Connectivity of 41 Grid Points and Triangle Strips with Vertex Swapping

The advantages mentioned above (maintaining edges between LODs and maintaining edges in four directions) allow for better rendering in two areas: lower discrepancy between LODs and lines to follow the terrain's contour in more directions.

Organizing the triangles in this manner allows less discrepancy between LODs. In the LODs presented in the previous section, some edges disappear and new edges appear between each LOD. In the LODs presented in this section, some edges disappear, but no new edges appear going from a higher LOD to a lower LOD. It is as if a single point changes elevation, rather than a set of points and edges. This difference allows for effective betweening of the LODs, as mentioned in Chapter 2. If changing LODs changes the edges, then betweening cannot smoothly interpolate between LODs without introducing more points or more polygons.

Organizing the triangles in this manner allows lines in four directions to follow the terrain's contours. The points in each unit do not need to be evenly spaced. The points may be moved to conform to the terrain, aligning edges along ridges and valleys.

3.5. Discussion

This data structure introduces many advantages over storage formats such as the Flight format. It minimizes memory needs. It offers a standard format for conversion into a rendering format. It can directly address cracking between adjacent units of differing level of detail. Switches between LODs can be based on a viewer's speed and altitude. Culling can be based on the viewer's altitude. The data structure also introduces some disadvantages, such as additional real time computational requirements and unit level of abstraction for terrain areas.

3.5.1. Memory Minimization

Memory minimization leads to quick paging. Quick paging not only means less time spent paging each unit, but also paging fewer units, overall. The slower paging is, the more data that needs to be prepaged to make sure it is available when it is needed. (Of course, there must be some upper limit, where pre-paging cannot keep up with a view.)

The data structure minimizes memory use in two areas. First, memory use is minimized by only storing a vertex's information once, and inferring the connections of points from the data's structure. Second, memory use is minimized by adding information to successive LODs, rather than repeating information in each LOD.

3.5.2. Standard Format

The data structure presented offers a standard data format with standard connectivity between the points in each unit. Standard connectivity means quick conversion into a rendering format, because the conversion algorithm does not need to make decisions on how to handle exceptions in the data. The standard connectivity also allows consistent triangle meshing for fast rendering.

3.5.3. Cracking

Storage structures such as the Flight format maintain static LODs. When adjacent units of terrain are rendered at differing LODs, cracks may exist between the units where the LODs do not match up. Dean McCarty used the solution of adding "skirts" of extra

polygons descending down from each unit of terrain.[18] This solution has two problems. First, it introduces extra polygons that must be stored, paged, and rendered. Second, it assumes that the points in each edge of a unit are coplanar to a vertical plane. If the points in an edge are not coplanar, then there will be cracking between the skirts.

The data structure presented can directly address cracking between adjacent units of differing level of detail. When two adjacent units of differing LODs exist in a view, the lower LOD unit has every other point removed from the edge matching the higher LOD unit. The extra points in the higher LOD unit's edge can easily be interpolated to match the lower LOD unit's edge.

(Also note: with the data structure presented, some LOD changes only remove points from within a unit, so cracking is only a problem between every other LOD change.)

3.5.4. Switching Levels of Detail

In Performer LOD, switching is based on the viewer's absolute distance from each unit. With the data structure presented, switches between LODs can be based on a viewer's speed and altitude.

If a viewer is traveling at a high rate of speed, detail may not be as important as at low speeds. The LOD strategy may be changed to lower detail at high speeds.

At high altitudes, Performer may still render high detail terrain under the viewer, if the viewer is still within the switching distance of the high LOD. With the data structure presented, the LOD strategy may be changed to lower detail at high altitudes.

3.5.5. Culling

In Performer, culling is based on an absolute distance plane, the far clipping plane. (The far clipping plane can be set, but not dynamically changed.) With the data structure presented, culling can be based on the viewer's altitude. That is, the closer the viewer is to the earth's surface, the less distance the viewer needs to see. The data conversion

algorithm might perform culling by only converting data to a specific distance, independent of Performer's far clipping plane.

3.5.6. Paging versus Computation

Although the proposed data structure reduces paging needs, it requires real time computation to transfer its data into Performer's rendering format. The decrease in paging needs must be weighed against the additional computational requirements.

3.5.7. Level of Abstraction

The data structure manages each unit of terrain area as a whole. It minimizes memory use by storing each polygon as a part of the unit, but does not deal with individual polygons. That is, it abstracts terrain down to the unit level, not the polygon level. Thus, all polygons within a unit share the same attributes such as material and texture.

Within Performer, storing polygons in the same group with similar attributes will speed rendering. Any storage method should take advantage of similarities in terrain areas by storing and rendering like polygons in the same group. However, the data structure presented represents a fixed number of polygons. If the boundaries of a unit are shifted, it still must contain the standard number of points with the standard connectivity.

3.6. Summary

This chapter presented a data structure to store units of terrain information. Discussed were the factors around which the data structure was designed, along with the advantages and disadvantages of the structure. For the data structure to be useful, a system must be built to manage the units of terrain. Such a system is presented in the next chapter.

IV. System Design

This chapter outlines the system supporting the data structure of Chapter 3. The system design is Object Oriented. The main goal of the Object Oriented structure is abstraction of the system into classes which encapsulate the functional areas of the system. The system's object model is shown in Figure 4.2. (Some of the attributes and methods are left out of the diagram, for simplicity and brevity.) Toward abstraction, each class is designed to provide a complete function, minimizing communication among classes. Toward encapsulation, each class is designed to hide its structure from the other classes. Access to a class's structure is solely through its methods.

The Object Model is drawn using Rumbaugh's[25] notation. A brief key to the notation is shown in Figure 4.1. Each class has a name, a set of attributes, and a set of operations. The attributes are the variables owned by the class. The operations are the functions implemented in the class. In Rumbaugh's terminology, an implementation of an operation is a method. In the inheritance structures shown in Figure 4.2, operations shown in a subclass override its superclass's operations. If an operation is not overridden, it is inherited.

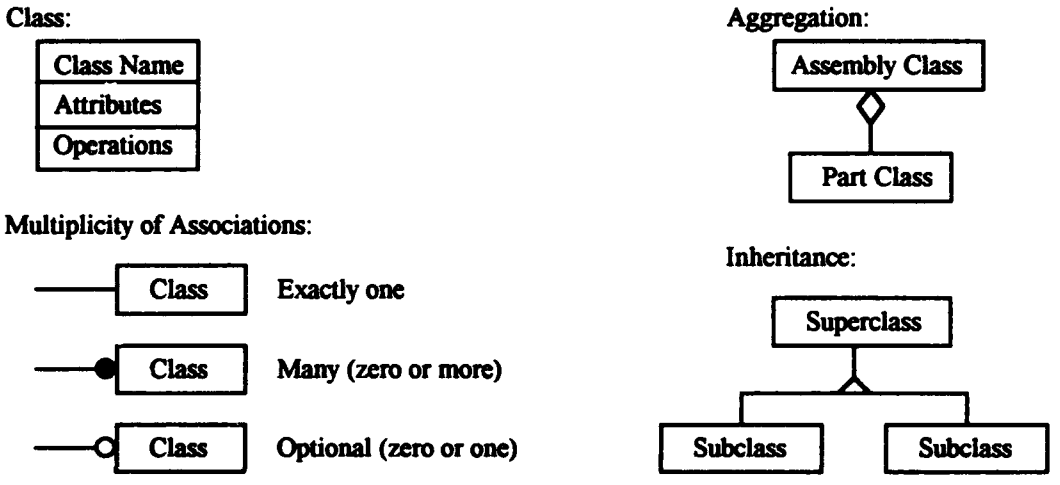


Figure 4.1. Key to Object Model Notation

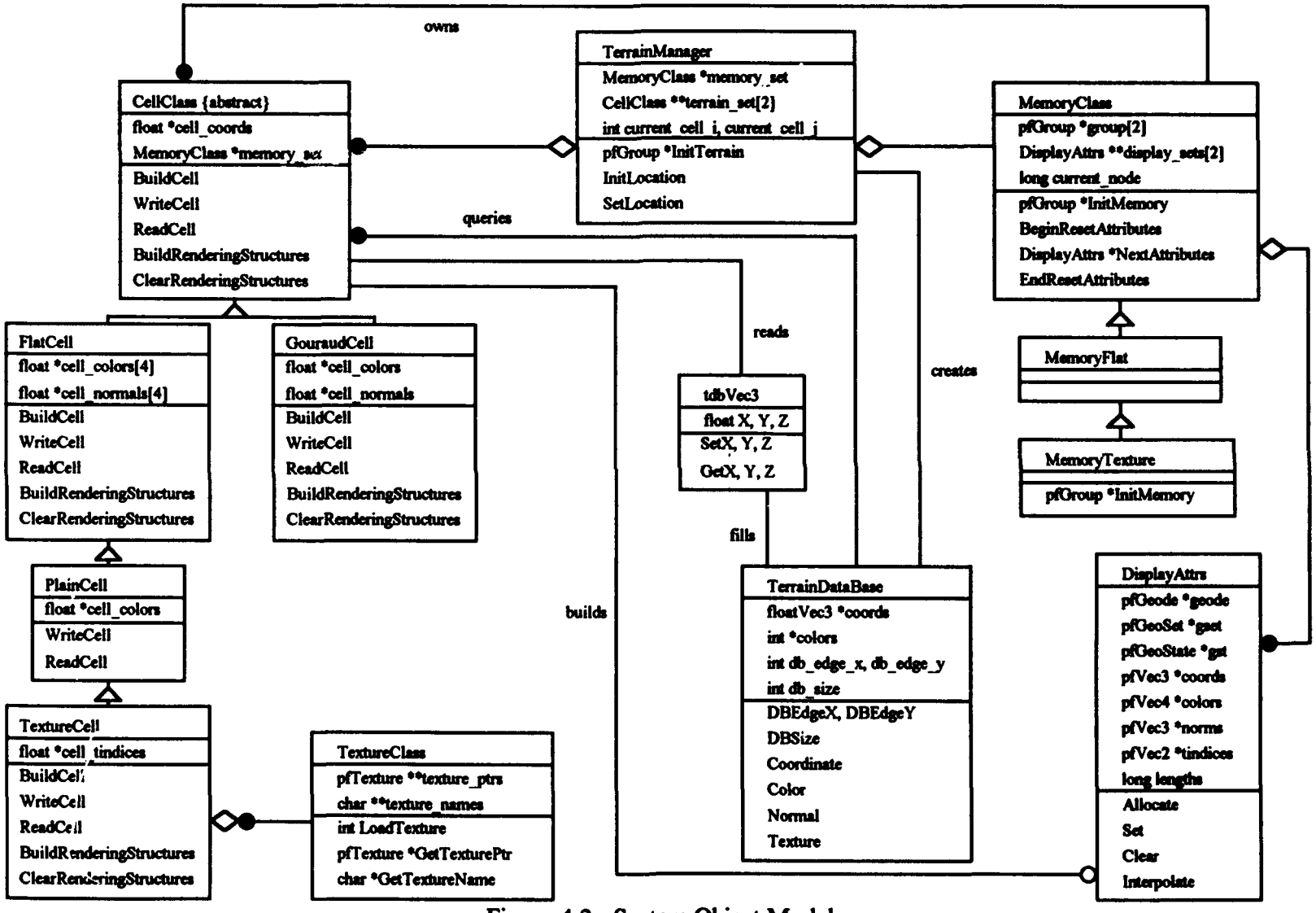


Figure 4.2. System Object Model

The system is divided into two areas: management of the terrain data and management of the rendering structures. The classes in each area are encapsulated. The classes which manage the terrain data, such as CellClass and TerrainDataBase, hide the data's organization from the classes which manage the rendering structures. (The CellClass and TerrainDataBase hide their data organization from each other.) The classes which manage the rendering structures, such as MemoryClass and DisplayAttrs, hide the machine dependent structures from the classes which manage the terrain data. Where the terrain data needs to be stored in a machine dependent structure, a separate class is used, such as the TextureClass contained in the TextureCell (a subclass of CellClass).

To pull it all together, the TerrainManager class controls the classes in both areas. The TerrainManager provides the interface for an application using this system.

4.1. Design Overview

The TerrainManager comprises the CellClass and MemoryClass. Each CellClass object stores sections of the terrain in the data structure discussed in Chapter 3. The MemoryClass manages the rendering structures. Actions of the TerrainManager are broken into two areas: building of the CellClass objects and MemoryClass object (initialization), and building of the MemoryClass's rendering structures from the CellClasses' data (execution).

During initialization, the TerrainManager creates a TerrainDataBase. It populates the CellClass by passing the TerrainDataBase and location information to the CellClass. Then, it allocates the rendering structures by passing level of detail strategy information to the MemoryClass.

During execution, the TerrainManager (based on information received from the application program) decides how to build the terrain view from the CellClass objects. It calls each needed CellClass object, directing the object to build itself into the rendering structures at a specified level of detail.

The CellClass contains the data necessary to build an area of the total terrain. Each CellClass object owns a MemoryClass object. (A MemoryClass object may be owned by more than one CellClass object.) During initialization, the TerrainManager creates CellClass objects to cover the entire terrain area. Each CellClass object queries the TerrainDataBase. To pass vector values, the TerrainDataBase fills a tdbVec3 object, which the CellClass reads. During execution, the CellClass gets a DisplayAttrs object from the MemoryClass. The CellClass builds the rendering structures contained in the DisplayAttrs.

The CellClass is an abstract class, providing only the attributes and methods to build the cell's coordinates into the rendering structures. It provides the basis for subclasses rendered with Gouraud shading, flat shading, single color flat shading and texture mapping; represented by GouraudCell, FlatCell, PlainCell and TextureCell, respectively.

A TextureCell object contains a TextureClass object that encapsulates Performer's structure for loaded textures. One TextureClass object may be associated with many TextureCell objects, so a given texture does not need to be loaded more than once.

The MemoryClass manages the rendering structures. During initialization, it creates the rendering structures in DisplayAttrs objects. Each DisplayAttrs object contains enough memory for rendering a cell at a specific level of detail. The MemoryClass keeps the DisplayAttrs in order by level of detail. During rendering, MemoryClass gives the DisplayAttrs objects to the CellClass objects in order.

The MemoryClass is a base class, managing the rendering structures for Gouraud shading. The MemoryFlat subclass manages the rendering structures for flat shading. The MemoryTexture subclass manages rendering structures (flat shading) for texture mapping.

The MemoryClass is a base class, as opposed to an abstract class like CellClass. It was implemented as a base class, because Performer uses the same amount of memory for Gouraud and flat shaded triangle strips. A flat shaded triangle strip needs two fewer

colors and normals than a Gouraud shaded triangle strip. In flat shading, Performer ignores the first two colors and normals in each triangle strip. The only difference between MemoryClass and MemoryFlat is the draw mode.

4.2. Classes

The subsections below discuss the classes as implemented for this thesis. (The Cell Definitions section is not a class, but holds global constants.) Each subsection title shows the class name, along with the file names containing the class. Each file with a '.h' suffix defines a header file, containing information which is visible to other classes. Each file with a '.cc' suffix defines a program body, containing operation implementations which are hidden from other classes.

4.2.1. Cell Definitions (cell_def.h)

This section defines constants used in defining cells. The values defining cell size are set here, so the cell size may be changed without changing the CellClass. Currently, the cell size is set to 17 by 17 vertices.

4.2.2. CellClass (cellclass.h, cellclass.cc)

This section defines a basic cell, which is derived from the data structure discussed in Chapter 3. The CellClass is an abstract class, containing several pure virtual functions. The intention is to have a base class, which the main program thinks it is manipulating. That is, the main program declares pointers to the base class type (but it cannot create objects of the base class, because it is an abstract class), yet it allocates objects derived from the base class. Then, when the main program operates on the objects, it uses the functions provided by the derived class(es).

The CellClass contains an array of coordinates and the methods (functions) to build the coordinates (representing polygon vertices) into the rendering structure. However, it will use the methods provided by derived classes for building normals, colors and texture indices into the rendering structures. The building of the normals, colors and

texture indices in the derived classes must be tightly coupled to the building of the coordinates; the order of the coordinates in the triangle strips dictates the required order of the normals, colors and texture indices.

The coordinate building routines really should be virtual functions, as well, so the coordinates can be connected using various strategies. Specifically, a connectivity strategy could use triangle meshing with vertex swapping, aimed at the betweening strategy discussed in Chapter 3. Unfortunately, Performer does not support vertex swapping, so only one connectivity strategy is implemented.

The implemented system has two types of build methods for cells. First, it builds the storage structure (coordinate array) from the terrain data base. The BuildCellCoords method builds the storage structure for the cell's coordinates. Then, as needed, it builds the rendering structure from the storage structure. Each cell may be represented in the rendering structures at one of several LODs. Hence, there are several BuildRenderingCoords methods, each of which builds the rendering structure for the cell at a single level of detail.

The implemented system uses four levels of detail for each cell. Each cell has three sets of coordinates, where each set builds on the previous set(s). The three sets are placed in a single array of coordinates in four sections. The sections are labeled A, B, C and D. The A section represents the third LOD, as discussed in Chapter 3, Section 3. Adding the B section represents the second LOD. Adding the C and D sections represent the first LOD. The fourth LOD can be extracted from the A section, using only a subset of the coordinates from the A section.

Extracting the fourth LOD from the A section is obviously inconsistent with the strategy outlined in Chapter 3. I had originally intended to have three LODs per cell. When I came to defining lattices (levels of cells) for the TerrainManager (Section 4.2.7.), I found that with an odd number of LODs, the LODs between lattices overlap. With only three LODs per cell, only two LODs are usable for each lattice above the first. Rather

than rewrite the CellClass methods for the first three LODs, I added a fourth LOD to extract its information from the A section. The CellClass methods could be rewritten giving the fourth LOD its own section without affecting the other classes.

Each point is shown in Figure 4.3 labeled with its section and the index in that section. The coordinates are stored in row-major order, with the lower left hand corner of the cell as the first element in the array. (Note: The actual implementation uses cells of 17 x 17 coordinates. However, for simpler illustration, 9 x 9 cells are used in the figures. Of course, either cell size is valid, and the current implementation was written with great care to accommodate cell size changes.)

Figure 4.3. Grid of 81 Points with Array Section and Index Labels

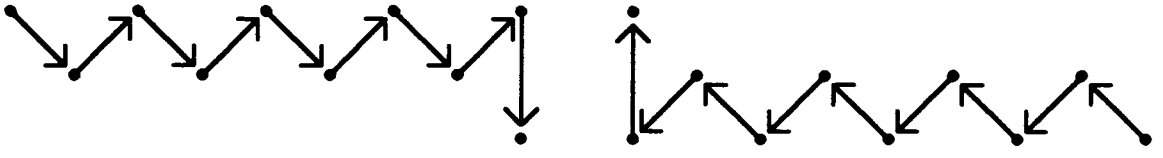
The BuildRenderingCoords methods build the cell's storage structure into triangle strips in the rendering structure. The most efficient way to step through arrays is contiguously. Thus, the BuildRenderingCoords methods build the triangle strips in the

order which best supports stepping through the array of coordinates in a contiguous manner.

The method BuildRenderingCoords1 builds the first LOD using all sections in the array of coordinates, building triangle strips as shown in Figure 3.11(b). BuildRenderingCoords1 creates two sets of triangle strips, so it can step through the four sections of the array in a contiguous manner. First, it creates the odd triangle strips (1st, 3rd, etc.). Then, it creates the even triangle strips (2nd, 4th, etc.). (Most of the coordinates in a cell are used in two triangle strips. So, building the rendering structure requires two passes through the cell's array of coordinates. After building the first triangle strip, the indices into the array sections are already in position for building the third triangle strip.) The order of coordinates in the triangle strips is shown in Figure 4.4.

Figure 4.4. Triangle Strip Order of Vertices for First Level of Detail

The method BuildRenderingCoords2 builds the second LOD using the A and B sections in the array of coordinates, building triangle strips as shown in Figure 3.13(a). BuildRenderingCoords2 creates two sets of triangle strips, so it can step through the two sections of the array in a contiguous manner. First, it creates the odd triangle strips (1st, 3rd, etc.). Then, it creates the even triangle strips in reverse order (16th, 14th, etc.). The order of coordinates in the odd triangle strips is shown in Figure 4.5(a); the even triangle strips, in Figure 4.5(b).



(a) (b)

Figure 4.5. Triangle Strip Order of Vertices for Second Level of Detail

The method BuildRenderingCoords3 builds the third LOD using the A section in the array of coordinates, building triangle strips as shown in Figure 3.15. BuildRenderingCoords3 creates one set of triangle strips in order (1st, 2nd, etc.), using two indices into the A section. (After building the first triangle strip, the indices into the array sections are already in position for building the second triangle strip.) The order of coordinates in the triangle strips is shown in Figure 4.6.

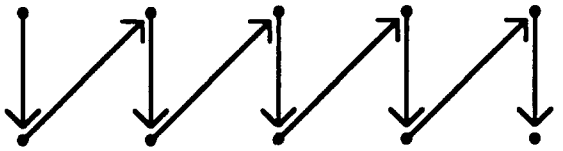
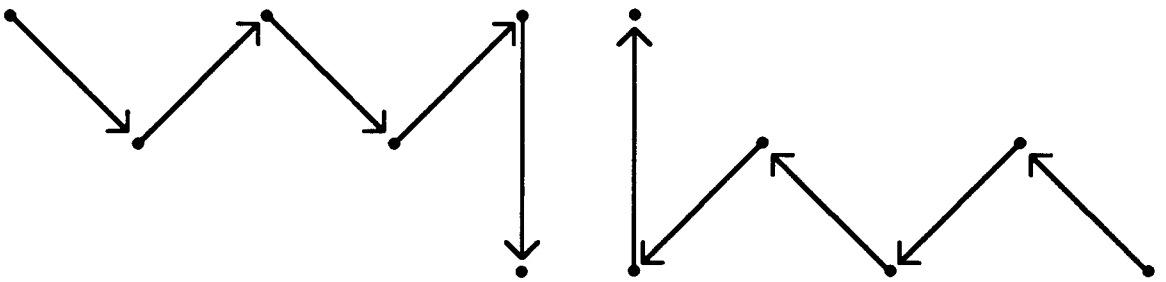


Figure 4.6. Triangle Strip Order of Vertices for Third Level of Detail

The method BuildRenderingCoords4 builds the second LOD using the A section in the array of coordinates, building triangle strips as shown in Figure 3.17(a). BuildRenderingCoords4 creates two sets of triangle strips, so it can step through the two sections of the array in a contiguous manner. First, it creates the odd triangle strips (1st, 3rd, etc.). Then, it creates the even triangle strips in reverse order (8th, 6th, etc.). The order of coordinates in the odd triangle strips is shown in Figure 4.7(a); the even triangle strips, in Figure 4.7(b).



(a) (b)

Figure 4.7. Triangle Strip Order of Vertices for Fourth Level of Detail

The BuildRenderingCoords1 and BuildRenderingCoords3 methods accept a parameter telling if the cell being built will be adjacent to any less detailed cells. That is, if the cell shares edges with cells of less detail, then certain coordinates within the shared

edges must be interpolated to avoid cracking between the cells. The even numbered BuildRenderingCoords do not have such a parameter, because no cracking can occur around an even numbered LOD. The difference between an even numbered LOD and the next less detailed odd numbered LOD are interior points, only. The shared edges between these two LODs have the same coordinates.

4.2.3. GouraudCell (gouraud.h, gouraud.cc)

The GouraudCell is a derived class, inheriting the structure of the CellClass. The GouraudCell adds an array of colors and an array of normals, along with the methods to build the colors and normals into the rendering structure.

This section defines a cell colored with Gouraud shading. In Gouraud shading, colors and normals are associated with the vertices of each polygon. The colors and normals are interpolated across the polygon to determine the color and shading at each pixel, creating a smooth coloring and shading effect. Because the colors and normals are associated with vertices, they can be stored and processed (to build geoset representations of the cells) in relatively the same manner as the coordinates in the CellClass.

4.2.4. FlatCell (flatcell.h, flatcell.cc)

The FlatCell is a derived class, inheriting the structure of the CellClass. The FlatCell adds an array of colors and an array of normals, along with the methods to build the colors and normals into the rendering structure.

This section defines a cell colored with flat shading. In flat shading, colors and normals are associated with the face of each polygon. The colors and normals are applied once to each polygon, so all pixels in the polygon have the same color and shading.

Because the colors and normals are associated with faces, the storing and processing scheme used for the coordinates does not apply. From the coordinates' point of view, LOD switching involves adding or deleting polygon vertices. From the colors' and normals' point of view for flat shading, LOD switching involves changing polygon faces. It might be possible to share face colors and normals in a scheme similar to sharing

coordinates. However, for this implementation, separate colors and normals arrays are maintained for each LOD.

4.2.5. PlainCell (plaincell.h, plaincell.cc)

The PlainCell is a derived class, inheriting the structure of the FlatCell. The PlainCell replaces the array of colors with a single color to assign to all polygons in the cell, along with the methods to build the color into the rendering structure.

The PlainCell is intended for use with texture mapping. Specifically, if the color will not show through the texture (applied as a decal), or if the texture needs a single base color, then the cell should not waste memory storing the same color several times.

Ideally, if the texture is applied as a decal, then the color should not be built into the rendering structure every time the terrain is rebuilt. Either the color should be stored in the rendering structure during initialization, or the rendering structure could remain uninitialized.

4.2.6. TextureCell (texcell.h, texcell.cc)

The TextureCell is a derived class, inheriting the structure of the PlainCell. The TextureCell adds an array of texture indices, along with the methods to build the texture indices into the rendering structure.

This section defines a texture mapped cell. In texture mapping, indices into a texture are associated with the vertices of each polygon. The texture is interpolated across the polygon to determine the coloring at each pixel. Because the texture indices are associated with vertices, they can be stored and processed (to build geoset representations of the cells) in relatively the same manner as the coordinates in the CellClass.

4.2.7. TerrainManager (terrainmgr.h, terrainmgr.cc)

The TerrainManager manages the cells defining an area of terrain, along with their associated levels of detail. A cell by itself merely describes a small area of terrain. It is the TerrainManager which pieces cells together, at varying levels of detail, into a specific view

of the terrain. With a hierarchical view of the system, the TerrainManager is the top component which interfaces with an application system.

The TerrainManager contains a MemoryClass object and many CellClass objects. The MemoryClass object maintains and manages the rendering structures. The CellClass objects are organized in a hierarchy of sets, referred to as "lattices." Each lattice contains a set of cells at a specific size. In this implementation, the first lattice contains the cells representing the terrain with the shortest post spacing; the second lattice, with posts spaced at four times the shortest post spacing.

Figure 4.8 shows a terrain area, bounded by a thick line, covered by cells in two lattices. The cells of the highest detail lattice are bounded by dotted lines. The cells of the lowest detail lattice are bounded by narrow lines. The edges of each lattice may extend beyond the edge of the terrain, to maintain the standard number of points in each cell. In this system, the "out of bounds" points drop to a zero altitude below the edge of the terrain, creating a cliff.

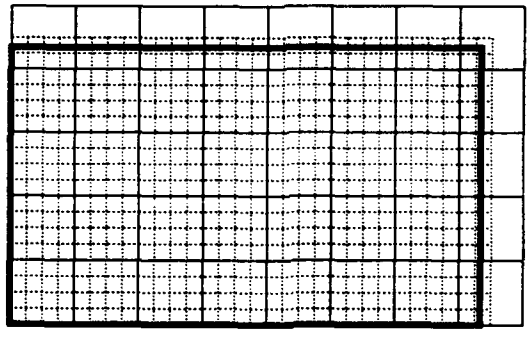


Figure 4.8. Lattice Structure Applied to a Terrain Area

Lattices of cells add more levels of detail than are available from a single set of cells. The basis for lattices is three-fold. First, it is difficult to manage the elements of a cell for all the levels of detail it may represent. For example, the implemented system represents a cell with three sets of coordinates, for three explicit levels of detail. To represent more levels of detail, the entire lowest set of coordinates must be in memory. Much main memory and paging would be wasted to manage lower levels of detail.

Second, if a cell were to represent all possible levels of detail, the advantages of triangle meshing would be lost at the lowest detail (due to the very short strips built), and many small rendering sets would be built. Third, the terrain cannot be managed above the level of abstraction of a cell size. (If a view required polygons larger than the cell size, the polygons might be extracted from several cells.)

A lattice is similar to a general level of detail, as discussed in Chapter 2. A lattice may be viewed as a level of levels of detail, where each lattice represents a subset of the total levels of detail. In the implemented system, the first four levels of detail are built from the first lattice; the next four levels of detail, from the second lattice.

An application may control the TerrainManager through three methods: InitTerrain, InitLocation, and SetLocation. InitTerrain initializes and returns a pointer to the rendering structures. InitLocation sets the initial location of the viewer and builds the rendering structures for the initial view. (As discussed below, SetLocation only rebuilds the rendering structure if the viewer *moves far enough* to induce a change in LOD. So, InitLocation must build the initial view, or there may be nothing to view until the viewer moves far enough.)

SetLocation accepts locations from the application as the viewer moves through the terrain area. As the viewer moves, the TerrainManager rebuilds the terrain's rendering structures to maintain level of detail rings around the viewer. The TerrainManager must decide how many rings of each level of detail to build around the viewer. The decision may be based on the viewer's location (including altitude), speed, and any other factor deemed useful.

The implemented system reads a file named "Command" to determine the type of cells (Gouraud, flat, plain or textured), where to get the terrain data, and what level of detail strategy to use. The system allows two level of detail strategies, broken by altitude. The Command file format is shown in Appendix A.

To create a view combining cells from more than one lattice, the TerrainManager must ensure that the edges of cells in one lattice match the edges of the cells in the next lattice. Because the cells of differing lattices are of different sizes, the area drawn in one lattice must be bounded in an area that matches the edges of the cells in the next lattice (if more than one lattice is used in a view).

For example, the implemented system uses four LODs from each lattice. The next (fifth) LOD is taken from the next lattice. An example of these rings is shown in Figure 4.9. The rings from the first lattice are outlined in dotted lines, with the cells of the fourth LOD shown explicitly.. (Note that there are four rings of cells in the fourth LOD.) The cells of the fifth LOD (represented by the first LOD from the second lattice) are outlined by solid lines.

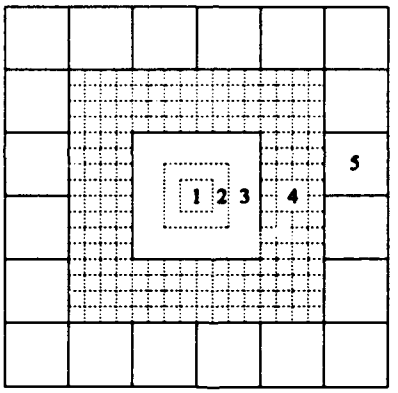


Figure 4.9. Level of Detail Rings Applied to a Terrain Area

If the viewer moves to a location where the rings from the first lattice are not aligned with the second lattice, some areas of terrain may overlap while other areas are not rendered. This unaligned situation is shown in Figure 4.10.

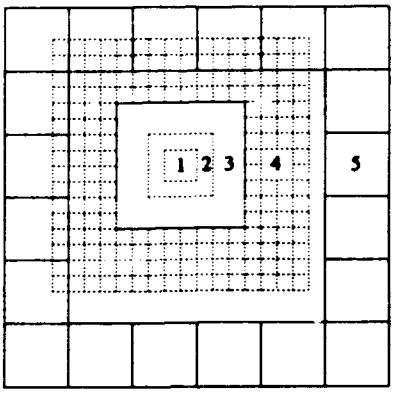


Figure 4.10. Unaligned Level of Detail Rings Applied to a Terrain Area

As a solution to this problem, the BuildOffsetLOD method adjusts the rings in the fourth LOD up to two cells, so the cells used in the fourth LOD align to the larger cells used in the fifth LOD. (If the viewer moves more than two cells in any direction, then the second lattice rings would be shifted, also.) This aligned situation is shown in Figure 4.11.

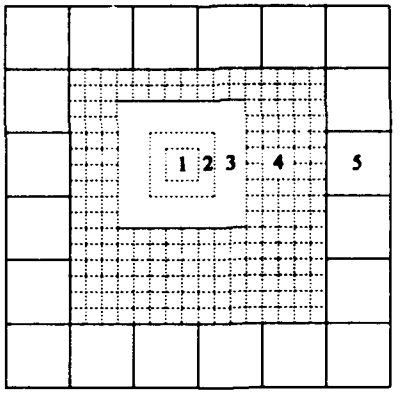


Figure 4.11. Offset Level of Detail Rings Applied to a Terrain Area

On the top edge of Figure 4.11, there are two cells in the fourth LOD between the third and fifth LODs; on the bottom edge, six cells. Four rings are used in the fourth LOD so that, when the fourth LOD is offset, there will be at least two cells between the third and fifth LODs in any given direction. If only two rings are used, then in some views an edge (or two) of the third LOD will meet the fifth LOD. Thus, the implemented system requires at least four rings in the fourth LOD to maintain a minimum of two cells in each direction.

The solution presented above for matching cells built from different lattices is only one of many possible solutions. For example, the offset could be done in the fifth LOD, if the system allowed building partial cells. This situation is shown in Figure 4.12.

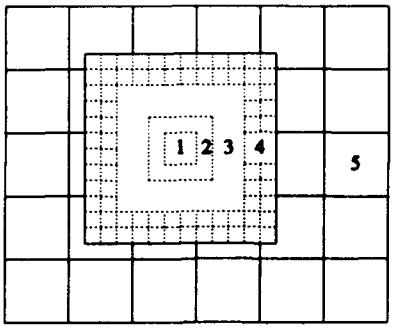


Figure 4.12. Level of Detail Rings with Partial Cells

4.2.8. TerrainDataBase (tdb.h, tdb.cc)

The TerrainDataBase hides the terrain format, providing a standard interface to the CellClass (and its derived classes). It provides methods to set the coordinates, colors, normals and texture file names in a CellClass. The TerrainDataBase returns these values for vertices, only. The CellClass contains information about the connectivity of the vertices (to connect vertices into polygons), which should not be known by the TerrainDataBase. Therefore, the CellClass must derive the colors and normals for polygon faces from the vertex values.

The implemented TerrainDataBase does not contain methods to set texture indices. It would need to contain such methods if the indices are applied unevenly to a cell, based on the texture and the world location of the cell.

The TerrainDataBase is not an integral part of this thesis effort, other than providing data to the CellClass objects. As such, many of the color values are hard coded to specific geographic areas. Currently, the terrain elevation data is taken from '.pts' files.[1] The TerrainDataBase class may be written to use data from any terrain format without affecting the other classes.

4.2.9. tdbVec3 (tdb.h)

The TerrainDataBase object may return vector values for coordinates or normals. Of course, the TerrainDataBase and CellClass may have differing views on how to store a vector value, and neither should care how the other deals with vectors. The tdbVec3 class provides an interface structure between TerrainDataBase and CellClass, with methods to load and extract the vector values.

4.2.10. MemoryClass (memclass.h, memclass.cc)

The MemoryClass manages the rendering structures used by the application program to render the terrain. It hides the Performer rendering structures from the TerrainManager and the CellClass. The MemoryClass contains portions of the rendering structure (specifically the pfGeodes) in DisplayAttrs objects (discussed below). The organization of Performer rendering structures is shown in Figure 4.13.

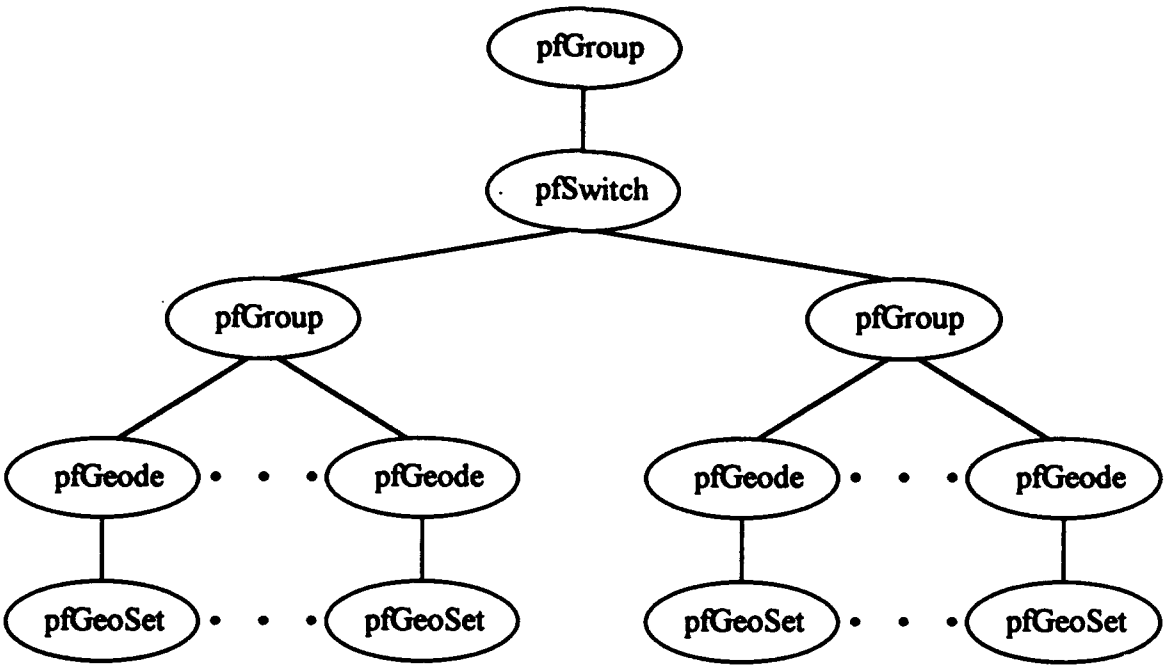


Figure 4.13. Organization of Performer Rendering Structures used by MemoryClass

The pointer to the top pfGroup Performer object is returned to the application program. The pfGroup objects below the pfSwitch are identical in structure; one

represents the current terrain view for rendering, while the other is for rebuilding the next terrain view. The pfSwitch is controlled by the TerrainManager through a MemoryClass method (EndResetAttributes). Each pfGeode and pfGeoSet combination represent the rendering structures built from CellClass.

Aside from encapsulating the rendering structures, the MemoryClass speeds terrain building by pre-allocating the rendering structures. Memory allocation is a slow process. Pre-allocation ensures the memory is available when the system needs it, without holding up the terrain building process.

The MemoryClass contains methods for the TerrainManager to set the level of detail strategy (number of LOD rings in each LOD set), to initialize and return a pointer to the rendering structures, and to manage the rendering structures. (The portion of the rendering structures built by the CellClass are kept in the DisplayAttrs class. Hence, DisplayAttrs contains the methods for CellClass to build the rendering structures.)

The MemoryClass allocates enough memory (for the rendering structures) to accommodate the level of detail strategy employed by the TerrainManager. It can accept more than one strategy, allocating the maximum amount of memory needed. The TerrainManager could create a MemoryClass object for each level of detail strategy; then, the TerrainManager would need to pass the proper MemoryClass object to each CellClass object during terrain rebuilding. (Currently, each CellClass contains a pointer to a single MemoryClass.)

4.2.11. MemoryFlat (memflat.h, memflat.cc)

The MemoryFlat class is a derived class, inheriting the structure of the MemoryClass. This is the simplest class of all. The only difference between Gouraud and flat shading, from Performer's point of view, is the draw mode. MemoryFlat contains one method with one line, to change the draw mode to flat shading.

Performer leaves memory unused with flat shading on triangle strips. In a Gouraud shaded triangle strip, a color and normal are associated with each vertex. In a

flat shaded triangle strip, a color and normal are associated with each face, which is two less than the vertices in the strip. However, Performer maintains the memory for a color and normal for each vertex but ignores the first two colors and normals in the strip.

4.2.12. MemoryTexture (memtex.h, memtex.cc)

The MemoryTexture class is a derived class, inheriting the structure of the MemoryFlat. MemoryTexture overrides the InitMemory, GetLengths and SetGSetAttrs methods to include the allocation of texture mapping rendering structures.

The entire InitMemory method should not need to be overridden, since much of it remains unchanged. Overriding InitMemory could be avoided by providing more methods which InitMemory may call for special processing, such as GetLengths and SetGSetAttrs. For example, GetLengths calls MemoryClass::GetLengths to get the coordinates, colors and normals array lengths, then adds the extra code needed to get the texture indices array length.

MemoryTexture adds a pfGeoState to the rendering structures built by CellClass, as shown in Figure 4.14. The pfGeoState describes the texturing attributes and contains the texture file reference (set by CellClass).

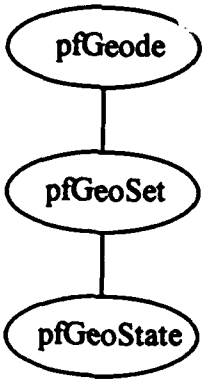


Figure 4.14. pfGeoState added to Performer Rendering Structures used by MemoryClass

4.2.13. DisplayAttrs (dispattr.h, dispattr.cc)

The DisplayAttrs class encapsulates the rendering structures built by the CellClass. DisplayAttrs' methods allow allocation of the rendering structures by MemoryClass, then

filling of the rendering structures by CellClass. The allocated structures include the pfGeode, pfGeoSet and pfGeoState, along with the arrays belonging to the pfGeoSet: coords, colors, norms, tindices and lengths. The detail of the rendering structures is shown in Figure 4.15. (The structure shown details what is used by this system. For more detail, see the *IRIS Performer Programming Guide*.)

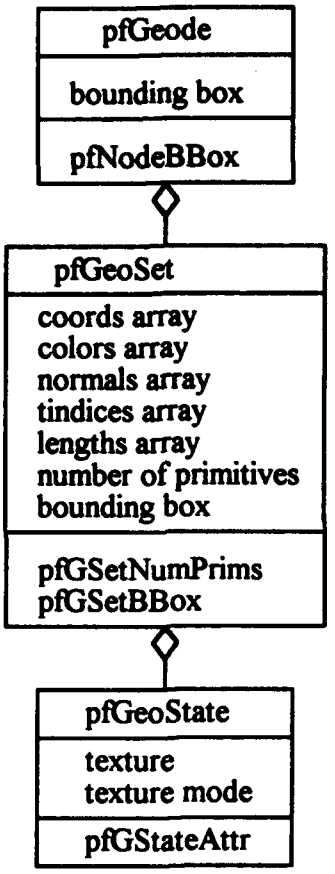


Figure 4.15. Object Model of Performer Rendering Structures used by DisplayAttrs

The MemoryClass actually allocates the rendering structure, then passes each structure's pointer to the DisplayAttrs object. A CellClass object can fill the rendering structure's arrays from the cell's data. For each array, DisplayAttrs accepts the array's values in order. That is, the array's method (such as SetColor) accepts the value and increments the array's index.

After filling the rendering structures, the CellClass needs the ability to interpolate certain values to prevent cracking. This interpolation takes two vertices, interpolates their midpoint, and places the midpoint values into one or two other vertices, as shown in Figure 4.16. Of course, DisplayAttrs should know nothing of the organization of the cell's vertices, and CellClass should know nothing of how to interpolate values stored in Performer structures. So, DisplayAttrs' Interpolate methods accept indices into the rendering structures which describe the vertices to be used in the interpolation. (The CellClass should know the order it filled the rendering structures.) Currently, this interpolation is only provided for coordinates.

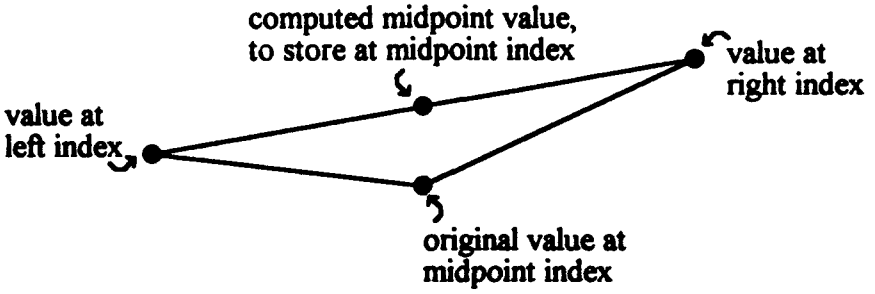


Figure 4.16. Interpolation of Edge Coordinates to Prevent Cracking

Additional (unused) methods provide for interpolation used in "betweening." This interpolation takes three vertices: one from a higher level of detail and two from a lower level of detail. First, it interpolates the midpoint between the two lower LOD vertices. Then, it interpolates a "betweening" value, between the midpoint and the higher level of detail vertex, based on an offset factor. This interpolation, shown in Figure 4.17, is provided for coordinates, colors and normals.

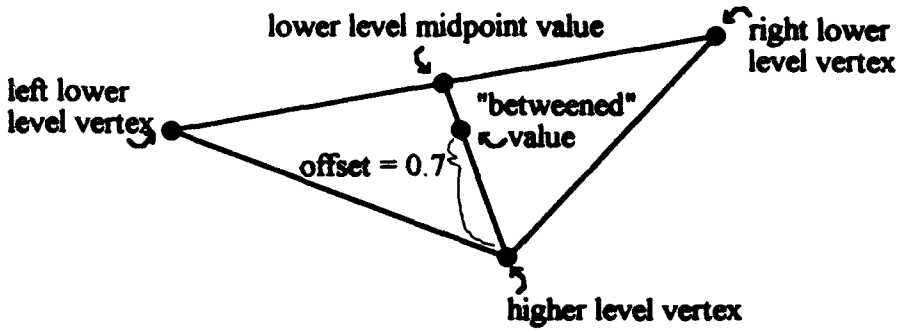


Figure 4.17. Interpolation of Coordinates for Betweening

4.2.14. TextureClass (texclass.h)

The TextureClass hides the Performer texture structure from the CellClass. A textured cell needs access to its loaded texture, so the cell can pass its texture to the rendering structure. However, a texture file is loaded into a Performer pfTexture object. So, TextureClass holds the loaded textures.

The TextureClass holds the additional advantage of saving memory, by only loading each texture once. A single TextureClass object is used for all CellClass objects. TextureClass keeps a list of all loaded textures. When a cell needs to load a texture, it calls LoadTexture with the texture's file name. If the texture has been loaded, then LoadTexture returns the index of the already loaded texture. Otherwise, it loads the texture and returns the index of the newly loaded texture.

4.3. Summary

This chapter presented a system to manage cells of terrain. The system is Object Oriented, encapsulating each major area of the system in a class. The next chapter presents an analysis of the system, comparing it to terrain data stored in MultiGen Flight format which is managed and rendered by Performer.

V. Analysis

This chapter presents an analysis of the system discussed in Chapter 4. The analysis is in two parts, measuring the internal and external characteristics of the system. First, the system was measured against selected maintainability metrics: cyclomatic complexity, coupling, and cohesion. Second, the functionality of the system was compared against terrain data stored in MultiGen Flight format which is managed and rendered by Performer.

5.1. Software Engineering Metrics

The metrics presented in this section indicate the maintainability of the system. McCabe's cyclomatic complexity assigns an integral measure to each function, indicating the understandability of the function. Coupling indicates the complexity of the interfaces between classes and the functions within a class. Cohesion indicates how well a class or function implements a single service.

5.1.1. McCabe's Cyclomatic Complexity

McCabe's cyclomatic complexity metric assigns an integral measure to each function. Beginning at one, a higher value indicates higher complexity. The value is based on a control flow representation of a program.[17] The value is not an absolute measurement of complexity, but it does indicate the areas of a program which are likely to be overly complex. The metric was applied to the system twice. After the first application, I broke some functions into smaller functions when the metric highlighted them as potential problem areas. In a few cases (discussed below), I reviewed the function but did not break it into smaller functions.

The results of McCabe's cyclomatic complexity metric applied the second time to each function in each class are shown in Table 5.1. The first two functions in each class are the class's constructor and destructor.

Class & Functions	Complexity
CellClass	
CellClass	1
~CellClass	1
BuildCellCoords	4
BuildRenderingCoords1	6
FillCoords1	9
BuildRenderingCoords2	5
BuildRenderingCoords3	4
FillCoords3	9
BuildRenderingCoords4	5
ClearRenderingStructure	1
GouraudCell	
GouraudCell	1
~GouraudCell	1
BuildCell	1
WriteCell	1
ReadCell	1
BuildCellColors	4
BuildCellNormals	4
BuildRenderingStructure1	1
BuildRenderingStructure2	1
BuildRenderingStructure3	1
BuildRenderingStructure4	1
BuildRenderingColors1	5
BuildRenderingColors2	5
BuildRenderingColors3	3
BuildRenderingColors4	5
BuildRenderingNormals1	5
BuildRenderingNormals2	5
BuildRenderingNormals3	3
BuildRenderingNormals4	5

Table 5.1. Class Functions' Cyclomatic Complexity

Class & Functions	Complexity
FlatCell	
FlatCell	1
~FlatCell	1
BuildCell	1
WriteCell	1
ReadCell	1
BuildCellColors1	5
BuildCellColors2	5
BuildCellColors3	3
BuildCellColors4	5
BuildCellNormals1	5
BuildCellNormals2	5
BuildCellNormals3	3
BuildCellNormals4	5
BuildRenderingStructure1	1
BuildRenderingStructure2	1
BuildRenderingStructure3	1
BuildRenderingStructure4	1
BuildRenderingColors1	3
BuildRenderingColors2	3
BuildRenderingColors3	3
BuildRenderingColors4	3
BuildRenderingNormals1	3
BuildRenderingNormals2	3
BuildRenderingNormals3	3
BuildRenderingNormals4	3
PlainCell	
PlainCell	1
~PlainCell	1
WriteCell	1
ReadCell	1
BuildCellColors	1
BuildRenderingColors1	3
BuildRenderingColors2	3
BuildRenderingColors3	3
BuildRenderingColors4	3

Table 5.1. Class Functions' Cyclomatic Complexity (continued)

Class & Functions	Complexity
TextureCell	
TextureCell	2
~TextureCell	1
BuildCell	1
WriteCell	1
ReadCell	1
BuildCellTextureIndices	4
BuildRenderingStructure1	1
BuildRenderingStructure2	1
BuildRenderingStructure3	1
BuildRenderingStructure4	1
BuildRenderingTextureIndices1	5
BuildRenderingTextureIndices2	5
BuildRenderingTextureIndices3	3
BuildRenderingTextureIndices4	5
TerrainManager	
TerrainManager	18
~TerrainManager	5
InitTerrain	1
InitLocation	2
SetLocation	10
BuildTerrain	1
BuildTerrainSet	6
WriteTerrain	5
ReadTerrain	2
ReadTerrainSet	6
BuildLOD	14
BuildOffsetLOD	7
CalculateFillEdge	8
tdbVec3	
tdbVec3	1
~tdbVec3	1
SetX	1
SetY	1
SetZ	1
GetX	1
GetY	1
GetZ	1

Table 5.1. Class Functions' Cyclomatic Complexity (continued)

Class & Functions	Complexity
TerrainDataBase	
TerrainDataBase	9
~TerrainDataBase	3
DBEdgeX	1
DBEdgeY	1
DBSize	1
Coordinate	2,1,3*
Normal	6,2,1,1,3*
Color	2,1
Texture	2
OutOfBoundsCoordinate	11
NonNeg	2
CrossProduct	1
NormalizeVector	2
ComputeNormal	1
MemoryClass	
MemoryClass	4
~MemoryClass	1
InitMemory	5
BeginResetAttributes	1
NextAttribute	1
EndResetAttributes	1
GetLengths	5
SetGSetAttrs	1
SetGSetDrawMode	1
AllocateAttrs	2
MinLOD	4
RingsInLOD	1
MaxCells	1
CellsInLODSet	1
SetRingsInLOD	1
SetSizeOfLOD	1
SetLODinLattice	1
MemoryFlat	
MemoryFlat	1
~MemoryFlat	1
SetGSetDrawMode	1

Table 5.1. Class Functions' Cyclomatic Complexity (continued)

Class & Functions	Complexity
MemoryTexture	
MemoryTexture	1
~MemoryTexture	1
InitMemory	5
GetLengths	1
SetGSetAttrs	1
DisplayAttrs	
DisplayAttrs	2
~DisplayAttrs	1
AllocateColors	1
AllocateNorms	1
AllocateCoords	1
AllocateTextureIndices	1
AllocateLengths	1
AllocateGeode	1
AllocateGSet	1
AllocateGState	1
Initialize	1
SetCoord	1,1*
SetNormal	1,1 *
SetColor	1,1*
SetTextureIndex	1
SetLength	1
SetNumberOfStrips	1
SetBoundingBox	1
Clear	1
InterpolateCoord	1,1*
SetTexture	1
TextureClass	
TextureClass	2
~TextureClass	1
LoadTexture	4
GetTexturePtr	1
GetTextureName	1

Table 5.1. Class Functions' Cyclomatic Complexity (continued)

*Where a function has more than one implementation, based on parameters, the cyclomatic complexity for each implementation is shown.

Analysis of McCabe's metric has shown 10 to be a practical upper limit for a function's complexity.[23] Few functions in this system exceed the limit: TerrainManager's constructor has a cyclomatic complexity of 18; TerrainManager's BuildLOD, 14; and TerrainDataBase's OutOfBoundsCoordinate, 11.

The TerrainManager constructor contains initialization statements. Most of these statements are enclosed in if-else-if or switch constructs, which rapidly increase a function's complexity. The segments of this function with high cyclomatic complexity could be broken into more functions. However, the constructor has a straight flow with few loops. Only one loop contains an if-else-if construct for reading the input file. Breaking the constructor into smaller functions would likely break its flow and diminish understandability.

The TerrainManager's BuildLOD function contains many nested loops with switch and if-else statements at the center of the loops. It is an overly complex function. The main switch statement could be put into another function. Unfortunately, the statements within the switch use many of the loop counters; passing so many parameters to a function would be unwieldy. A better solution would be the use of function pointers. The main switch statement calls a BuildRenderingStructure function for the proper level of detail. The BuildRenderingStructure function pointers may be stored in an array. A call made to one of the functions, based on a calculation of an index into the array, could replace the switch statement.

The TerrainDataBase's OutOfBoundsCoordinate contains if-else-if statements nested two deep. If a cell needs a coordinate for a vertex out of the terrain area, this function determines the region of the vertex and calculates a coordinate neighboring the terrain area. The entire complexity measure results from determining the region of the vertex. Breaking the function into smaller functions would likely break its flow and diminish understandability.

5.1.2. Coupling

Coupling indicates interconnection between classes and functions. Coupling may be represented as a spectrum from low to high coupling. From low to high, the spectrum includes no direct coupling, data coupling, stamp coupling, control coupling, external coupling, common coupling, and content coupling.

> In software design, we strive for the lowest possible coupling. Simple connectivity among modules results in software that is easier to understand and less prone to a "ripple effect" caused when errors occur at one location and propagate through the system.[23]

5.1.2.1. Between Classes. Coupling between most classes (through each class's methods) is limited to no direct coupling or data coupling. Data coupling is the lowest form of coupling between two modules which communicate. Stamp coupling exists where a class object is passed between classes. No complex data structures are passed between classes unless the structure is encapsulated in a class.

In one case, control coupling exists between two classes. The TerrainManager may call CellClass's BuildRenderingStructure functions with a parameter directing whether or not to interpolate any of the edges to control cracking between cells. On the surface, this case of control coupling does not appear to be a problem; the TerrainManager built the cells, so it knows what edges of a cell should be interpolated in a particular terrain rendering. However, this results in "tramp data,"[22] where a variable is passed through BuildRenderingStructure and other functions until it reaches FillCoords. A better solution would be to have the TerrainManager call FillCoords directly (or something more appropriately named, such as InterpolateEdge).

5.1.2.2. Within Classes. Coupling between functions within classes ranges high, to external and common coupling. There is also a more subtle form of coupling between functions which do not even communicate, which I term logical coupling.

External coupling within certain classes (MemoryClass, DisplayAttrs and TextureClass) refers to Performer's rendering structures. These classes are designed to hide the Performer structures from other classes, so their external coupling is appropriate.

Common coupling within classes is inherent in Object Oriented Programming. A class defines a data structure (attributes) and the functions (methods) to manipulate the data structure. Basically, a class's data structure is visible to the functions in the class; each object accesses its attributes as global data. Common coupling within classes is appropriate.

The most dangerous form of coupling in this system is between some functions which do not communicate. I term it logical coupling. For example, The CellClass defines methods to build the cell's terrain data into polygons in Performer's rendering structures. The CellClass's subclasses define various ways to apply colors to the polygons. The logic used to build the polygons must be equivalent to the logic used to apply the colors. If the methods to build the polygons change, then the methods to apply the colors must also change.

5.1.3. Cohesion

Cohesion indicates how well a class or function hides a single concept. Cohesion may be represented as a spectrum from low to high coupling. From low to high, the spectrum includes coincidental cohesion, logical cohesion, temporal cohesion, procedural cohesion, communicational cohesion, sequential cohesion, and functional cohesion.

> We always strive for high cohesion, although the mid-range of the spectrum is often acceptable. The scale for cohesion is non-linear. That is, the low-end cohesiveness is much "worse" than middle-range, which is nearly as "good" as high-end cohesion.[23]

5.1.3.1. By Class. The cohesion of each class is functional. Object Oriented Analysis and Design creates classes which encapsulate a single function. The function of each class can be simply stated:

CellClass (and its subclasses): Manage terrain data at the cell level of abstraction.

TerrainManager: Manage the terrain rendered from cells.

MemoryClass (and its subclasses): Manage Performer's rendering structures.

DisplayAttrs: Encapsulate Performer's rendering structures.

TextureClass: Encapsulate Performer's loaded texture file structure.

TerrainDataBase: Encapsulate external terrain data formats.

5.1.3.2. By Function Within Classes. "In practice a designer need not be concerned with categorizing cohesion in a specific module. Rather, the overall concept should be understood and low levels of cohesion should be avoided when modules are designed."[23]

The functions within each class generally perform a single service. The methods (functions callable from other classes) of each class were designed to perform one service from the perspective of the calling class. Within a class, the cohesion of its methods may appear lower than seen by the calling class. For example, the TerrainManager calls the CellClass's BuildRenderingStructure methods to perform the single service of building the cells rendering structures for a specific level of detail. Within the CellClass, the BuildRenderingStructures methods may be viewed as temporally cohesive, because they call (at least) three other functions to build the cells coordinates, normals and colors.

5.2. Measurements & Comparisons

Comparisons are difficult between this system and the current method of building terrain, using MultiGen to build Flight format. This thesis came about because of the limitations of building terrain within Flight format. Building terrain within Flight format is extremely time and memory intensive. When I tried to build areas as small as 64

kilometers square with 125 meter post spacing and five levels of detail, our machines ran out of swap space. Our machine with the largest main memory, 160 Megabytes, has a swap space of 40624 Kilobytes. I was able to build 16 kilometer square areas with 125 meter post spacing, which is equivalent to 64 kilometer square areas with 500 meter post spacing. With the system developed for this thesis, I was able to load and fly through a data base of 115 by 274 kilometers, at 250 meter post spacing.

The comparisons given here between Flight format terrain and the new Cell format terrain are based on 64 kilometer square areas with 500 meter post spacing. For terrain rendered from Flight files, four levels of detail are used, with the post spacing increasing by a factor of two between LODs. Each LOD area (equivalent to a cell) is eight kilometers square. The first LOD switch-in distance is 16 kilometers; the second, 48 kilometers; and the third, 112 kilometers. The fourth LOD encompasses the entire 64 kilometer square area, switching in at 240 kilometers. (The switching distances are from the viewer to the center of a terrain area.)

For terrain represented with the Cell format, eight levels of detail are used, with the post spacing increasing by a factor of $\sqrt{2}$ between LODs. Because of the different ratios in post spacing, one LOD in the Flight format covers the same area as two LODs in the Cell format. The first LOD is carried to a distance of eight kilometers from the viewer; the second, 16 kilometers; the third, 32 kilometers; the fourth, a minimum of 48 kilometers; the fifth, 80 kilometers; the sixth, 112 kilometers; the seventh, 176 kilometers; the eighth, 240 kilometers.

5.2.1. Memory Usage

The memory usage comparison is shown in Table 5.2, shown graphically in Figure 5.1. The comparison is based on necessary polygon information. Where texture mapping is used, only the texture indices are considered, without the memory needed for the actual texture. The comparison assumes Flight rendering (in Performer) uses triangle and quadrilateral meshing. (Without meshing, the Performer format uses more memory.)

	Flight Storage		Flight Rendering		Cells	
Shading Type	Total Bytes	Bytes Per Polygon	Total Bytes	Bytes Per Polygon	Total Bytes	Bytes Per Polygon
Flat	4661352	88.0697	2620800	49.5163	1802544	27.6125
Plain	4661352	88.0697	2620800	49.5163	1020000	15.6250
Textured	5180680	97.8817	3144960	59.4196	1177216	18.0333

Table 5.2. Memory Usage by Format in Bytes

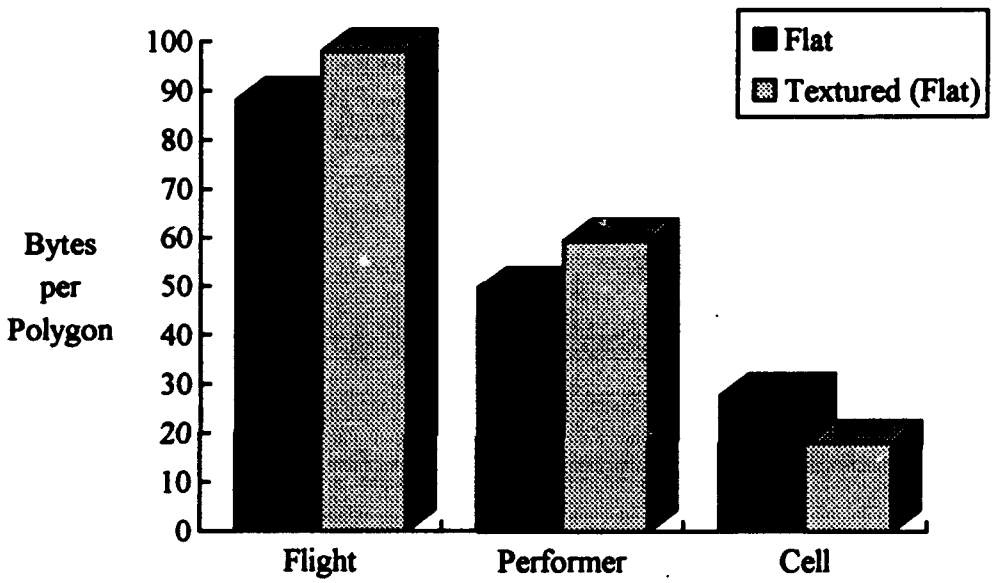


Figure 5.1. Memory Usage by Format in Bytes

The memory usages for Flight storage are taken from the Flight format files. The memory usages for Flight rendering (in Performer format) are calculated, based on the arrays needed to store the coordinates, colors, normals and texture indices. The memory usages are the same for Flat and Plain polygons, because Performer stores all polygon colors separately.

(Note: In Performer, the colors for Plain polygons could be indexed, saving almost 75% of the memory needed for storing colors [5.1] However, if one item is indexed, Performer requires all items to be indexed. For terrain, all coordinates, normals

and texture indices are likely to be unique. Indexing the coordinates, normals and texture indices would increase their respective memory needs by 33%, 33% and 50%. Moreover, indexing would likely slow the frame rate.)

The memory usages for the Cell format are calculated, based on the memory needed to store the coordinates, colors, normals and texture indices. The memory usage figures shown here for Flight rendering are low, because they do not include the overhead for storing management information, such as switching distances. The Cell format does not have this additional overhead, because the TerrainManager decides how to piece the cells together.

Table 5.3 shows the memory comparison in the percent of memory the Cell format uses compared to the other formats.

Shading Type	Flight Storage	Flight Rendering
Flat	31.55%	55.76%
Plain	17.74%	31.56%
Textured	18.42%	30.35%

Table 5.3. Cell Format Percent Memory Usage by Bytes per Polygon

It is obvious the Cell format requires less memory overall, as well as less memory per polygon. It is not so obvious that the Cell format represents more polygons than the Flight format. The Cell format has twice the LODs as the Flight format files. If we had the tools to build a Flight format file with as many LODs as the Cell format, the Flight format memory needs would increase by approximately 50% (and the per polygon need would increase slightly).

5.2.2. Paging & Conversion Processing

5.2.2.1. Paging for the Entire View. The memory usage comparison shows only part of the advantage of the Cell format. Within a lattice, the Cell format can increase the level of detail of a cell by paging in the additional needed information; half the

information is already in memory for the current LOD rendering of the cell. Also within a cell, the Cell format can decrease the level of detail of a cell without any paging necessary. In contrast, a Performer managed LOD strategy would require paging in the entire new area.

In the worst case paging situation, a viewer is traveling in such a manner as to always encounter new terrain. Table 5.4 shows the worst case paging needed when a viewer travels one cell's width, requiring an update to the view's LODs. (A cell's width is based on the post spacing and the number of posts in the cell. In this case, a cell has 17 posts on an edge, with a post spacing of 500 meters, for a cell width of eight kilometers. If post spacing were 125 meters, a cell's width would be two kilometers.) The data in Table 5.4 is limited to three levels of detail for Flight format and six levels of detail for Cell format.

	Flight Rendering		Cell Rendering	
Shading Type	Demand Paging	Pre-Paging	Demand Paging	Pre-Paging
Flat	520960	670080	545760	760824
Plain	520960	670080	325656	455184
Textured	625152	804096	394584	552800

Table 5.4. Pre-Paging Needs by Format in Bytes

The differences between paging Flight format and Cell format shown in Table 5.4 are not very significant. In fact, Flat Cell rendering appears to require more paging than Flight format rendering. There are three reasons for the high worst case paging for Cell format. First, the fourth LOD is overly large, to maintain a minimum distance for the LOD. Second, the fourth LOD pages twice what it needs, because it is extracted from the cell's section representing the third LOD. These two reasons account for much of the Cell format paging. Third, and most importantly, the paging for the Cell format's fourth, fifth and sixth LODs is only needed when the viewer travels four cell widths. If a pre-paging

strategy is used, the paging for these last three LODs can be averaged over four cells. Table 5.5 compares the pre-paging needs, with the averaged values for Cell format.

	Flight Rendering	Cell Rendering
Shading Type	Pre-Paging	Pre-Paging
Flat	670080	263358
Plain	670080	154782
Textured	804096	184946

Table 5.5. Pre-Paging Needs by Format in Bytes, Averaged for Cell Rendering

Table 5.5 shows Cell format pre-paging needs are, at worst, less than half the Flight format pre-paging needs. Of course, the Cell format's pre-paging needs are only the beginning. The Cell format must be converted into Performer data structures for rendering. The conversion process must be fast enough to maintain the view at the desired level of detail.

5.2.2.2. Conversion Processing for the Entire View. The conversion process builds each cell in the view to the proper level of detail. The first four levels of detail are taken from the first lattice. The remaining levels of detail are taken from the second lattice, using the same processing. The data in the following tables show the processing times required to build the viewable terrain area at appropriate LODs into Performer format. Table 5.6 shows the processing times required to build the LODs on an IRIS 4D/440VGXT. Table 5.7 shows the processing times required to build the LODs on an ONYX Reality Engine2.

Cell Rendering		
Flat	Plain	Textured
0.22	0.21	0.25

Table 5.6. Cell Conversion Time for Terrain View in Seconds, IRIS 4D/440VGXT

Cell Rendering		
Flat	Plain	Textured
0.11	0.11	0.14

Table 5.7. Cell Conversion Time for Terrain View in Seconds, ONYX Reality Engine2

These processing times should be added to the pre-paging times for Cell format and compared to the pre-paging times for Flight format. The maximum disk access rate is ten megabytes per second (which is a pretty generous figure that does not consider seek times).[21] The comparison of pre-paging and conversion times are shown in Tables 5.8 and 5.9, and graphically in Figure 5.2.

	Flight Rendering	Cell Rendering
Shading Type	Pre-Paging	Pre-Paging & Conversion
Flat	0.06701	0.24633
Plain	0.06701	0.22548
Textured	0.08041	0.26849

Table 5.8. Pre-Paging & Conversion Times in Seconds, IRIS 4D/440VGXT

	Flight Rendering	Cell Rendering
Shading Type	Pre-Paging	Pre-Paging & Conversion
Flat	0.06701	0.13633
Plain	0.06701	0.12548
Textured	0.08041	0.15849

Tables 5.9. Pre-Paging & Conversion Times in Seconds, ONYX Reality Engine2

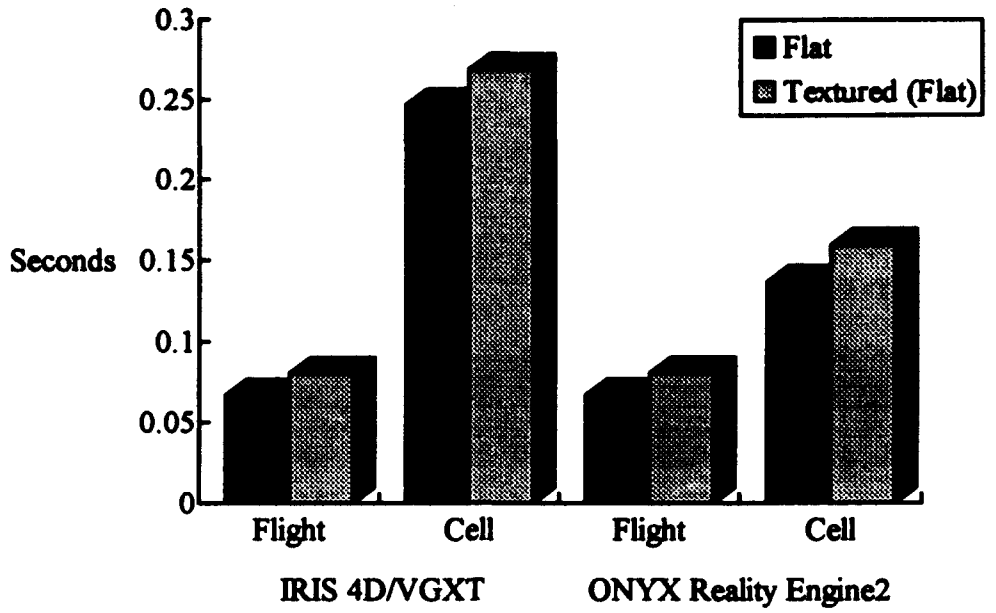


Figure 5.2. Pre-Paging & Conversion Times in Seconds

Tables 5.8 and 5.9, and Figure 5.2, show the Cell format processing far exceeds the paging needs of the Flight format. Table 5.10 shows pre-paging and conversion times for the Cell format as a percentage of the pre-paging time for rendering Flight format.

Shading Type	IRIS 4D/440VGXT	ONYX Reality Engine2
Flat	367%	203%
Plain	336%	187%
Textured	334%	197%

Table 5.10. Cell Format Processing Time as Percentage of Flight Pre-Paging Time

After studying these statistics, I realized the conversion time comparisons are stacked against the Cell format. While only part of the terrain area is paged for each view, the system presented by this thesis rebuilds the rendering structures for the every cell in the viewed area each time the view changes. Many of the cells should not be rebuilt, saving some time. However, many cells must be rebuilt even though their LOD representation does not change; many cells must be rebuilt, because their adjacencies change, requiring new edge interpolation to prevent cracking.

A fairer comparison would require the Flight format to store several representations of each area in an LOD, one for each possible adjacency. Unfortunately, Performer can not manage such a data set, since it can only switch terrain areas based on distance from the viewer, not LOD adjacencies. So, I present a comparison based on a by area basis.

5.2.2.3. Paging for a Terrain Area. For the comparison of terrain areas, equivalent areas are Cell format LOD 1 and 3 compared against Flight format LOD 1 and 2 without skirts to prevent cracking. The comparisons in Tables 5.11 and 5.12 show two values for the Cell format: for paging the entire cell, and for paging part of a cell if the lower LOD is already in memory.

	Flight Rendering LOD 1	Cell Rendering LOD 1	
Shading Type		Whole Cell	Partial Cell
Flat	2.0752	1.5026	1.3367
Plain	2.0752	0.9178	0.7507
Textured	2.4902	1.1383	0.8606

Table 5.11. Paging by Terrain Area in Milliseconds, First

	Flight Rendering LOD 2	Cell Rendering LOD 3	
Shading Type		Whole Cell	Partial Cell
Flat	0.5493	0.3857	0.3387
Plain	0.5493	0.2403	0.1923
Textured	0.6592	0.3021	0.2228

Table 5.12. Paging by Terrain Area in Milliseconds, Second

Tables 5.11 and 5.12 show the Cell format paging needs per terrain area are much less than for Flight format. A comparison which adds the processing time for Cell format follows.

5.2.2.4. Conversion for a Terrain Area. Tables 5.13 and 5.14 show the conversion time for each type of cell.

	Cell Rendering	
Shading Type	LOD 1	LOD 3
Flat	2.22	0.59
Plain	2.20	0.59
Textured	2.61	0.70

Table 5.13. Conversion by Terrain Area in Milliseconds, IRIS 4D/440VGXT

	Cell Rendering	
Shading Type	LOD 1	LOD 3
Flat	1.26	0.30
Plain	1.23	0.29
Textured	1.53	0.39

Table 5.14. Conversion by Terrain Area in Milliseconds, ONYX Reality Engine2

Combining the times for paging and conversion in Tables 5.15 through 5.18 show that the Cell and Flight formats are more comparable on a by area basis, especially on the ONYX Reality Engine2. This comparison is show graphically in Figures 5.3 and 5.4. Tables 5.19 and 5.20 show the pre-paging and conversion times by area for the Cell format as a percentage of the pre-paging time for rendering Flight format.

	Flight Rendering LOD 1	Cell Rendering LOD 1	
Shading Type		Whole Cell	Partial Cell
Flat	2.0752	3.7226	3.5567
Plain	2.0752	3.1178	2.9507
Textured	2.4902	3.7483	3.4707

Table 5.15. Processing by Terrain Area in Milliseconds, IRIS 4D/440VGXT, First

	Flight Rendering LOD 2	Cell Rendering LOD 3	
Shading Type		Whole Cell	Partial Cell
Flat	0.5493	0.9757	0.9087
Plain	0.5493	0.8303	0.7823
Textured	0.6592	1.0021	0.9228

Table 5.16. Processing by Terrain Area in Milliseconds, IRIS 4D/440VGXT, Second

	Flight Rendering LOD 1	Cell Rendering LOD 1	
Shading Type		Whole Cell	Partial Cell
Flat	2.0752	2.7626	2.5967
Plain	2.0752	2.1478	1.9807
Textured	2.4902	2.6683	2.3906

Table 5.17. Processing by Terrain Area in Milliseconds, ONYX Reality Engine2, First

	Flight Rendering LOD 2	Cell Rendering LOD 3	
Shading Type		Whole Cell	Partial Cell
Flat	0.5493	0.6857	0.6387
Plain	0.5493	0.5303	0.4823
Textured	0.6592	0.6921	0.6128

Table 5.18. Processing by Terrain Area in Milliseconds, ONYX Reality Engine2, Second

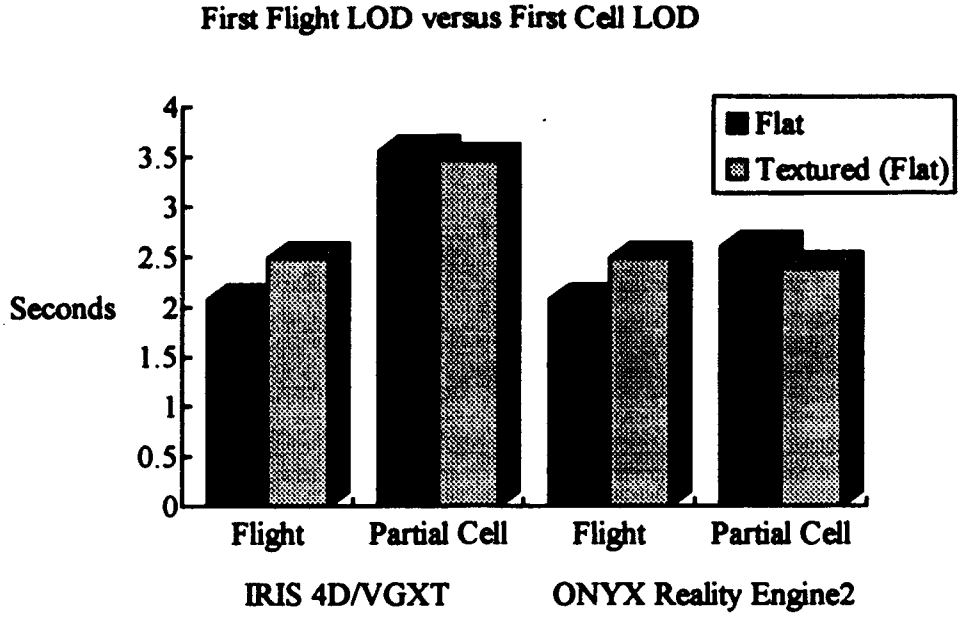


Figure 5.3. Paging & Conversion by Terrain Area in Milliseconds, First

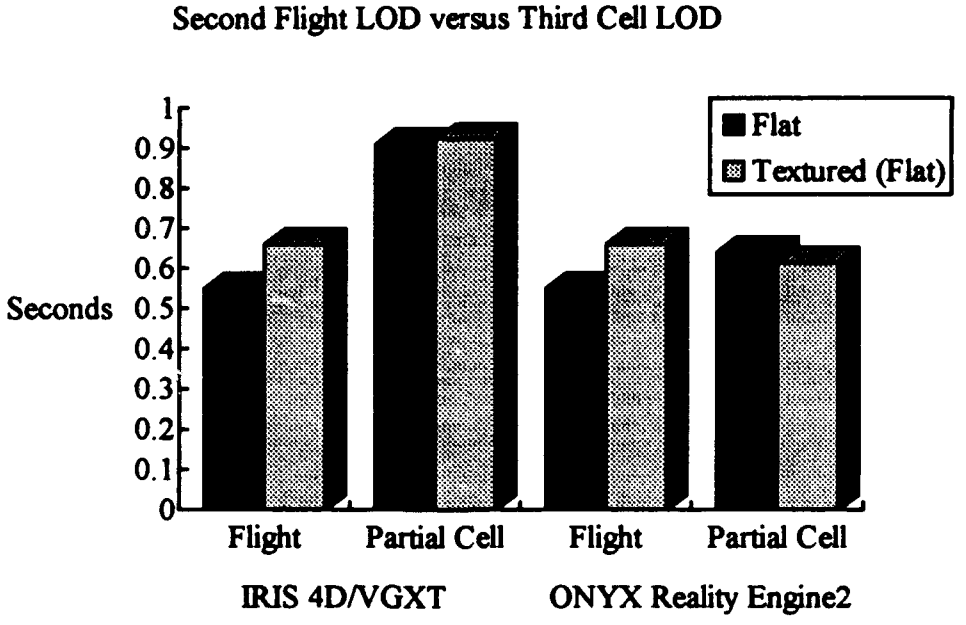


Figure 5.4. Paging & Conversion by Terrain Area in Milliseconds, Second

	LOD 1		LOD 3	
Shading Type	Whole Cell	Partial Cell	Whole Cell	Partial Cell
Flat	179%	171%	178%	165%
Plain	150%	142%	151%	142%
Textured	150%	139%	152%	140%

Table 5.19. Cell Format Processing Time Percentage by Area, IRIS 4D/440VGXT

	LOD 1		LOD 3	
Shading Type	Whole Cell	Partial Cell	Whole Cell	Partial Cell
Flat	133%	125%	124%	116%
Plain	103%	95%	96%	88%
Textured	107%	96%	105%	93%

Table 5.20. Cell Format Processing Time Percentage by Area, ONYX Reality Engine2

5.2.3. Frame Rate

Finally, the frame rate is the most important factor of a rendering system. The frame rate comparison was made using a 64 kilometer square area of terrain data from North West Iraq. The view was taken from one corner, moving diagonally to the far corner, to put the highest number of polygons in the view. The frame rate comparisons are shown in Tables 5.21 and 5.22, and graphically in Figure 5.5.

Shading Type	Flight Rendering	Cell Rendering
Flat	7.5 Hz	12 Hz
Plain	8.6 Hz	12 Hz
Textured	4.3 Hz	5 Hz

Table 5.21. Frame Rates, IRIS 4D/440VGXT

Shading Type	Flight Rendering	Cell Rendering
Flat	20 Hz	30 Hz
Plain	20 Hz	30 Hz
Textured	20 Hz	30 Hz

Table 5.22. Frame Rates, ONYX Reality Engine2

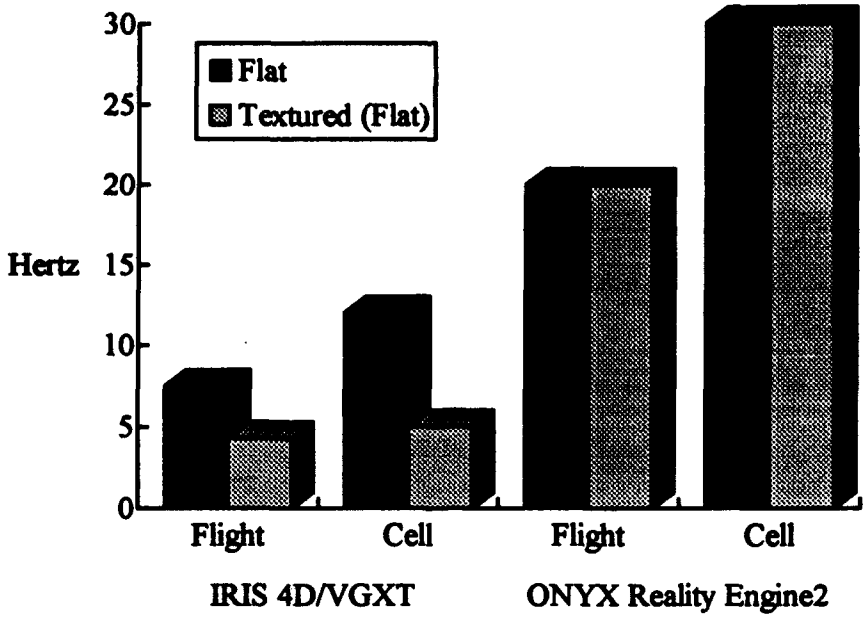


Figure 5.5. Frame Rates

Rendering in Cell format has an obvious advantage over rendering terrain built with our current tools in Flight format. The main reason for faster rendering from the Cell format is more levels of detail which allow fewer polygons. The Cell format also has no added polygons to prevent cracking. Finally, the Cell format does not require Performer to check level of detail switching distances; the TerrainManager makes those decisions when it builds the rendering structures.

5.3. Summary

The analysis in this chapter has shown the system developed for this thesis to be a well designed, maintainable system. The system provides advantages over Flight format in memory usage, paging, and rendering speed. Still, the real time conversion from the Cell format into performer is slow. The next chapter discusses the analysis of this chapter, along with the additional capabilities available with the Cell format.

VI. Conclusion and Recommendations

6.1. Conclusion

The Cell format data structure and its "on the fly" management have been shown to be a potentially viable system. It is able to reduce memory needs and increase frame rate, two key terrain rendering desires. It sets up easier pre-paging, although the conversion from storage to rendering format is slow.

The Cell format also allows many features not attainable from Flight format. These features include: interpolating vertices to prevent cracking, levels of detail by altitude or speed, and betweening.

Interpolating vertices to prevent cracking has several advantages over a fixed format such as the Flight format. First, the interpolation strategy uses fewer polygons. Our current tools to build Flight format terrain use extra polygons to fill in between terrain sections. Second, it does not require the vertices on an edge of a cell to be coplanar. Again, our tools to build Flight format terrain assume the vertices between LODs are all in the same plane, to ensure the added (cracking protection) polygons meet the neighboring terrain section. Third, the interpolation strategy allows less disruption of the terrain surface. The vertical surfaces inserted to prevent cracking not only disrupt the terrain's surface, but they also disrupt the continuity of texture mapping; with the added polygons, the edges of a texture map cannot meet at LOD boundaries, creating an artificial line between LODs.

The comparisons in Chapter 5 were made at a low level of flight. The Cell format has an additional advantage at higher altitudes which cannot be demonstrated by Flight format: it allows the LOD strategy to change based on the viewer's altitude. At higher altitudes, the viewer is likely looking forward, not down. Even if the viewer is looking down, a high level of detail is likely not necessary. A system managing the Cell format can

build the terrain below the viewer based on altitude. The Flight format translated into Performer has limited capabilities of lowering the detail directly below the viewer, because the level of detail is based on the viewer's distance from the area of terrain. If Flight stores the highest detail terrain in sections even as small as one kilometer squares, then the switching distance must be at least 1.707 kilometers, to ensure the area comes to its highest level of detail when the viewer is less than one kilometer from an edge of the area. (The greatest distance from the center of a square area to an edge, i.e., a corner, is half its width times the square root of two.) A viewer traveling at 1500 meters (almost 5000 feet) will sweep a path 1630 meters wide within range of the highest LOD, as shown in Figure 6.1. Flying at this altitude, the viewer will still have the highest level of detail area rendered below. With larger terrain sections, the problem is worse.

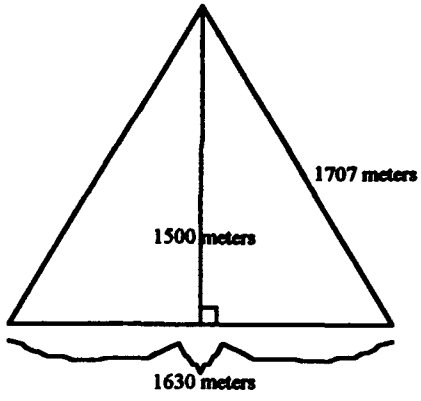


Figure 6.1. Area Below a Viewer at 1500 Meters

The sections of the system presented in this thesis have been encapsulated, making changes and extensions easier. That is, the data structures in each section can only be accessed by another section through encapsulating functions. The terrain data source is independent from the rest of the system. The rendering structures are hidden from the TerrainManager and CellClass. The TerrainManager can change LOD strategies without affecting the Cell structure. New CellClass subclasses can be added by only adding their initialization into the TerrainManager.

6.2. *Recommendations*

To make this a usable system, several areas need continuing research.

6.2.1. Optimization

The system needs a thorough review to optimize the Cell format to Performer conversion process. The speed of this conversion process is the key to the Cell format's usefulness.

6.2.2. Pre-Paging

A pre-paging system is necessary to speed LOD changes. A pre-paging system would need to determine what terrain areas the viewer may need soon, making those terrain areas available in memory.

6.2.3. Compressing Data

The implemented system was aimed at reducing memory needs by eliminating redundancy, only. It did not look at compressing the data such as: storing colors as a single integer; storing colors by indexing; computing normals from polygon information; computing x and y coordinates of evenly spaced vertices; or storing low values for coordinate information with an overall (cell level) offset. These methods could further reduce data by increasing processing time.

Although this system is aimed at reducing redundancy, it does not eliminate redundancy. Cells share vertices on their edges. In 16 by 16 cells, redundancy is almost 11.5%. Combining cells completely would break each cell into pieces in memory. Combining cells at their top and bottom edges (making the cells into strips) would reduce redundancy to under 6%. However, this would separate the sections of a cell, making paging more difficult.

6.2.4. Non-Uniform Grids

This system was originally aimed at managing terrain constructed from unevenly spaced vertices. Hence, values such as the x and y coordinates are stored with every

vertex, rather than being computed. Non-uniform grid management would require a more sophisticated method of determining the viewer's current cell location.

6.2.5. Aligning Cells to Terrain Features

The intent of using a non-uniform grid would be to align the vertices (and polygon edges) to major features in the terrain, such as ridges and valleys. A system to align vertices in such a manner, while retaining the connectivity required by a cell, would greatly increase the value of the Cell format data structure.

For example, consider the break necessary between land and sea. Strategies might include: aligning cells to the coast, so different cells are used for the land and sea; using textures which include land and sea, adjusting the texture indices so the texture matches the coast; or using special cells which build more than one geoset, so the land and sea polygons have different attributes (however, this strategy would pose additional problems for the MemoryClass).

6.2.6. Data Minimization

Data minimization could be accomplished by analyzing terrain and storing it only down to the level of detail necessary. If a terrain area is not represented in the highest level of detail lattice, then a lower detail cell must be used. This strategy would require methods to represent part of a cell, to fill around the cells from a higher detail lattice.

6.2.7. Round Earth

The only areas requiring change for rendering round earth terrain coordinates should be the TerrainDataBase (independent of the TerrainManager and CellClass) and the method for determining the viewer's current cell location.

6.2.8. Adjusting the Origin

The Virtual Cockpit employs a terrain strategy that adjusts the rendering system's origin to be under (or very near) the viewer, preventing jitter due to the large values of terrain coordinates. Such an adjustment should become parameters to the CellClass's BuildRenderingStructure methods. The items to be adjusted are the coordinates and

normals. Perhaps, instead of storing and adjusting normals, the normals could be calculated directly from the adjusted coordinate values.

6.2.9. Storing the Whole Earth

The sections of terrain make up quadrilateral areas of the earth's surface. The earth's surface can be divided into six quadrilaterals (like the surface of a volleyball), as the mapping of a cube to the surface of a sphere.[20] Each quadrilateral would be represented by a set of lattices.

6.2.10. Betweening

The Cell data structure is set up for betweening; it separates the vertices which remain unchanged from the vertices which will be changed. Betweening strategies generally break up the polygons from two levels of detail, involving more polygons than either level of detail. The Cell data structure establishes lines of change for betweening, so no extra polygons need to be introduced for betweening.

The Cell's betweening strategy requires vertex swapping within triangle meshes, but Performer does not allow vertex swapping. A simple change to Performer's handling of vertex pointers during rendering should make vertex swapping possible.

6.2.11. Texture Mapping

The cell is the lowest level of abstraction for terrain in this system. A cell can only have one texture applied (because a whole cell is stored in a geoset, and a geoset can only have one texture), although differing LODs could have different textures. However, using more than one texture for a cell within an LOD would have additional memory implications: (1) it increases the number of texture maps used, and (2) it means the same texture indices could not be used for all LODs in the cell. If only one texture map is used for the entire lattice of a cell, with the triangulation process used in the current implementation, texture mapping is best done with one texture map covering the entire cell. This means each lattice needs new texture maps, so the detail remains consistent.

(The texture needs to cover the entire cell, because the 2nd and 4th LODs do not form squares with the cell.)

The strategy outlined for betweening could be used to apply the texture multiple times across a cell, because the cell is always broken into squares. A single application of the texture must be large enough to cover the fourth LOD (the largest squares formed). With betweening and uneven vertex spacing, the texture can be distorted for visual effects, such as approximating a coastline.

6.2.12. Cultural Features

Cultural features and associated processing need to be added to this system, including the data required for placement and orientation. The orientation information may be stored for each LOD, or calculated based on the terrain orientation.

6.2.13. Integrate with an Application Program

This system needs to be analyzed with a real application program. This system was developed on top of ObjectSim, the same object management system used by the Virtual Cockpit and Synthetic Battle Bridge applications. Integration with these applications should be easy to set up. Yet, the most recent developments on these applications did not involve terrain with LOD capabilities. It is possible that the portions of these applications which interact directly with the terrain will not operate properly with LOD terrain.

Because this system would run as part of an application program, the possibility of competition for processing time between this system and an application program is high. It would be possible to limit terrain updates during high processing activity by the application.

6.3. Final Word

Terrain managed from the Cell format presented in this thesis has distinct advantages over terrain stored in Flight format which is managed by Performer. The

terrain data in Cell format saves memory, decreases paging, and speeds rendering. However, the Cell format requires real time conversion into Performer structures, which slows switching of LODs.

The Cell format offers many additional advantages not available from Flight format. These advantages include interpolating vertices to prevent cracking, levels of detail by altitude or speed, and betweening.

Finally, the Cell format allows management of terrain data bases larger than manageable with Flight format. The Cell format's lower memory usage and real time conversion allow an application to use large terrain data bases without the up front conversion from Flight format into Performer.

Appendix A. *Command File Format*

The Command file (named "Command" in the current directory) tells the system what type of cells to create, where to get the terrain data, and what level of detail strategy to use. The format is as follows:

cell type
terrain build, read, and write command
first level of detail strategy
switching altitude
second level of detail strategy

The first word in the Command file must be the cell type. The cell type determines whether the system creates Gouraud shaded, flat shaded, plain flat shaded, or textured cells. The cell type commands are *gouraud, flat, plain,* and *texture.*

Next, the Command file contains directions on where to get terrain data, and whether to store the terrain data in Cell format. The commands are *build, read,* and *write.* The *build* command tells the system to obtain its terrain data from the TerrainDataBase. (The TerrainDataBase does its own file handling. Currently, it reads from a file named "terrain.pts" in the current directory.) The *read* command directs the system to read a Cell format file. The input Cell format file name must follow the *read* command. The *write* command directs the system to write a built terrain set to a Cell format file. The output Cell format file name must follow the *write* command.

The first level of detail strategy tells the system how to build LOD rings when the viewer is below the switching altitude. Each level of detail command begins with *lod,* followed by an LOD number and the number of rings for the LOD. The LODs are numbered 1 through 4 for the first lattice and 5 through 8 for the second lattice.

The switching altitude determines the altitude at which the system will switch between the level of detail strategies. The switching altitude begins with *alt,* followed by a switching altitude in meters.

The second level of detail strategy tells the system how to build LOD rings when the viewer is above the switching altitude. The format for the second level of detail strategy is the same as the first.

Note: all commands are lower case. A command line may be easily commented out by capitalizing its first character.

An example Command file:

```
texture
build
write cell.fmt
lod 1 1
lod 2 1
lod 3 2
lod 4 4
lod 5 1
lod 6 1
alt 1000
lod 4 4
lod 5 1
lod 6 1
lod 7 2
lod 8 2
```

Appendix B. *List of Acronyms*

AFIT	Air Force Institute of Technology
ARPA	Advanced Research Project Agency
DARPA	Defense Advanced Research Project Agency
DBGen	DataBase Generation System
DFAD	Digital Feature Analysis Data
DMA	Defense Mapping Agency
DTED	Digital Terrain Elevation Data
GDMS	Graphical Database Management System
LOD	Level of Detail
SBB	Synthetic Battle Bridge
VC	Virtual Cockpit

Appendix C. *Glossary of Terms*

cell	An subset of a terrain area, managed as a single unit by the system developed for this thesis.
color	The color associated with a polygon vertex or polygon face.
coordinate	The location of a polygon vertex in 3-D space.
culling	The act performed by a rendering system to determine what portions of a database might be in view. Any portion determined not to be in view is not processed for rendering.
lattice	A set of cells covering an entire terrain area, representing certain levels of detail.
model	A polygon representation of a real world object for rendering.
normal	A vector representing the direction of a polygon vertex or polygon face is oriented in 3-D space. When associated with a vertex, the normal refers to the faces around it.
paging	The act of moving data from disk to main memory.
texture index	The point in a texture map (usually a unit square) associated with the vertex of a polygon.
triangle mesh	The term *triangle mesh* is used two ways. It is referred to by triangle minimization articles as the resulting triangulation of a minimized polygon set. It is referred to by Silicon Graphics as a polygon set stored in triangle strips. Its usage should be discernible by context.
triangle strip	A set of triangles with shared vertices, stored as a list of vertices such that the last two vertices in a triangle are the first two vertices of the next triangle.
vertex	Usually *vertex* refers to the placement of polygon vertices (corners). With regard to the system developed for this thesis, the term *vertex* refers to all information which must be associated with a polygon corner: coordinates (location), normal, color and texture index.

Bibliography

[1] Brunderman, John A. Design and Application of an Object Oriented Graphical Database Management System for Synthetic Environments. MS thesis, AFIT/GA/ENG/91D-01. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991.

[2] Clark, Charles L. and Michael A. Cosman. "Terrain Independent Feature Modeling," Interservice/Industry Training Systems Conference Proceedings, Nov 6-8, 1990: 7-17.

[3] Clark, Charles L. and Michael R. Pafford. "Geographical Subdivision and Top Level Data Structures: Columbus, Magellan, and Expanding CIG Horizons," 1984 Image III Conference Proceedings: 129-149. Williams AFB AZ: USAF, 1984.

[4] Coad, Peter and Edward Yourdon. Object Oriented Analysis (Second Edition). Englewood Cliffs NJ: Prentice-Hall, 1991.

[5] ----. Object Oriented Design. Englewood Cliffs NJ: Prentice-Hall, 1991.

[6] Cosman, Michael A. "A System Approach for Marrying Features to Terrain," Interservice/Industry Training Systems Conference Proceedings, Nov 13-16, 1989: 224-231.

[7] Costenbader, J. L. "CIG Data Bases in an Instance: Bits and Pieces," 1984 Image III Conference Proceedings: 151-163. Williams AFB AZ: USAF, 1984.

[8] De Floriani, L., B. Falcidieno, and C. Pienovi. "Delaunay-based Representation of Surfaces Defined over Arbitrarily Shaped Domains." Computer Vision, Graphics and Image Processing, 32(1): 127-140 (October 1985).

[9] Duckett, Donald P., Jr. The Application of Statistical Estimation Techniques to Terrain Modeling. MS thesis, AFIT/GCE/ENG/91D-02. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991.

[10] Erichsen, Matthew N. Weapon System Sensor Integration for a DIS-Compatible Virtual Cockpit. MS Thesis, AFIT/GCS/ENG/93D-07. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1993.

[11] Fenton, N. E. Software Metrics: A Rigorous Approach, Chapter 10 "Measuring Internal Product Attributes," Chapter 11 "Measuring External Product Attributes." Van Nostrand Reinhold: Chapman & Hall, 1991.

[12] Gerhard, William E. Weapon System Integration for the AFIT Virtual Cockpit. MS Thesis, AFIT/GCS/ENG/93D-10. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1993.

[13] Gomaa, H. "A Software Design Method For Real-Time Systems." Communications of the ACM, 27(9): 938-949 (September 1984).

[14] Haddix, Rex G., II. An Immersive Synthetic Environment For Observation and Interaction with a Large Volume of Interest. MS thesis, AFIT/GCS/ENG/93M-02. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 1993.

[15] Hoog, Thomas W. and John D. Stengel. "Computer Image Generation Using The Defense Mapping Agency Digital Data Base," Proceedings of the 1977 Image Conference: 202-218. Williams AFB AZ: USAF, 1977.

[16] Hoppe, Hugues and others. "Mesh Optimization." Computer Graphics (SIGGRAPH '93 Proceedings), Annual Conference Series: 19-26 (August 1993).

[17] McCabe, Thomas. "A Software Complexity Measure," IEEE Trans. Software Engineering, 2(4): 308-320 (December 1976).

[18] McCarty, Dean W. makeTerrain12 software tool. Developed for AFIT, 1993.

[19] McCarty, Dean W. and others. "A Virtual Cockpit For A Distributed Interactive Simulation Environment." Unpublished Report. Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 1993.

[20] Maillot, Jerome and others. "Interactive Texture Mapping." Computer Graphics (SIGGRAPH '93 Proceedings), Annual Conference Series: 19-26 (August 1993).

[21] Norcross, Troy W. Electronic mail message, 3 February 1994.

[22] Page-Jones, Meilir. The Practical Guide to Structured Design (Second Edition). Englewood Cliffs NJ: Prentice-Hall, 1988.

[23] Pressman, Roger S. Software Engineering: A Practitioner's Approach (Second Edition). New York: McGraw-Hill Publishing Company, 1987.

[24] Rife, Robert W. "Level-of-Detail Control Considerations," Proceedings of the 1977 Image Conference: 142-159. Williams AFB AZ: USAF, 1977.

[25] Rumbaugh, James and others. Object-Oriented Modeling and Design. Englewood Cliffs NJ: Prentice-Hall, 1991.

[26] Schroeder, William J., Jonathan A. Zarge and William E. Lorensen. "Decimation of Triangle Meshes." Computer Graphics (SIGGRAPH '92 Proceedings), 26(2): 65-70 (July 1992).

[27] Sheasby, Steven M. Management of SIMNET and DIS Entities in Synthetic Environments. MS thesis, AFIT/GCS/ENG/92D-16. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1992.

[28] Silicon Graphics. Graphics Library Programming Tools and Techniques.

[29] ----. Iris Performer Programming Guide.

[30] Snyder, Mark I. ObjectSim - A Reusable Object Oriented DIS Visual Simulation. MS Thesis, AFIT/GCS/ENG/93D-20. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1993.

[31] Software Systems. MultiGen Flight Modeler's Guide.

[32] Soltz, Brian B. Graphical Tools for Situational Awareness Assistance for Large Synthetic Battle Spaces. MS Thesis, AFIT/GCS/ENG/93D-21. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1993.

[33] Stankovic, John A. "Misconceptions About Real-Time Computing." Computer, 21(10): 10-19 (October 1988).

[34] Switzer, John C. A Synthetic Environment Flight Simulator: The AFIT Virtual Cockpit. MS thesis, AFIT/GCS/ENG/92D-17. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1992.

[35] Turk, Greg. "Re-Tiling Polygonal Surfaces." Computer Graphics (SIGGRAPH '92 Proceedings), 26(2): 55-64 (July 1992).

[36] Wilson, Kirk G. Synthetic BattleBridge: Information Visualization and User Interface Design Applications in a Large Virtual Reality Environment. MS Thesis, AFIT/GCS/ENG/93D-26. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1993.

Vita

Captain Keith L. Meissner was born on 26 May 1964 in Berkeley, California. He graduated from Sonora Union High School in Sonora, California in 1982. He attended the University of California (Cal) at Berkeley, California on an Air Force ROTC scholarship. In December 1986, he graduated from Cal with honors, receiving Bachelor of Arts degrees in Computer Science and Applied Mathematics. That same month, Captain Meissner received a commission as an officer in the United States Air Force. He entered active duty in October 1987, assigned to the Air Force Wargaming Center at Maxwell AFB, Alabama. Captain Meissner was the Senior Simulation Analyst for the Wargame Engine of the Air Force Command Exercise System (ACES). In May 1992, he entered the Air Force Institute of Technology.

Captain Meissner is married to Kristina Marie Meissner. They have two children, Erika and Stephen, with another child expected shortly after this writing.

Permanent Address: P.O. Box 569
Columbia, CA 95310

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1994	Master's Thesis

4. TITLE AND SUBTITLE

A FORMAT FOR STORING AND MANAGING MULTIPLE LEVEL OF DETAIL TERRAIN FOR SIMULATED ENVIRONMENTS

5. FUNDING NUMBERS

6. AUTHOR(S)

Keith L. Meissner, Captain, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GCS/ENG/94M-01

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

ARPA/ASTO
4301 N. Fairfax Ave, Suite 200
Arlington, VA 22203

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This study investigated a method of storing, managing and rendering terrain data, while addressing the conflicting goals of: rendering speed, display detail and memory usage. A data structure is presented to store terrain data, with an object oriented system to manage the data stored in the structure. The structure stores terrain data in a compact form, which is converted into rendering structures in real time. The structure uses levels of detail to maintain display detail. The structure is compared against an existing format for storing terrain data, MultiGen Flight. The system managing the structure is shown to decrease memory usage and increase frame rate, while maintaining display detail. The structure offers features not attainable from Flight format, including: interpolating vertices to prevent cracking, defining levels of detail by altitude and speed, and betweening. The structure allows storing and rendering larger databases than previously manageable with Flight format.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Terrain Modeling, Computer Graphics, Level of Detail, Object Oriented, Simulation, Virtual Reality	112
	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