

REAL-TIME VISUALIZATION OF  
MASSIVE IMAGERY AND VOLUMETRIC DATASETS

---

A Thesis  
presented to  
the Faculty of the Graduate School  
University of Missouri, Columbia

---

In Partial Fulfillment  
of the Requirements for the Degree  
Master of Science

---

by  
IAN JOSEPH ROTH  
Dr. K. Palaniappan, Thesis Advisor

MAY 2006

© Copyright by Ian Joseph Roth 2006

All Rights Reserved

The undersigned, appointed by the Dean of the Graduate School, have examined the thesis entitled

REAL-TIME VISUALIZATION OF  
MASSIVE IMAGERY AND VOLUMETRIC DATASETS

Presented by Ian Joseph Roth

A candidate for the degree of Master of Science

And hereby certify that in their opinion it is worth acceptance.

Dr. K. Palaniappan

---

Dr. Ye Duan

---

Dr. Curt Davis

---

Dr. Jeffrey Uhlmann

---

# Acknowledgements

First and foremost, I would like to thank Dr. K. Palaniappan for introducing me to this project three years ago and discussing many ideas along the way. I also thank the other committee members, Dr. Ye Duan, Dr. Curt Davis and Dr. Jeffrey Uhlmann, for attending my defense in July 2004 and for their advice and recommendations.

I also extend my gratitude to the others who have contributed to this project. Most important is Joshua Fraser, whose work I have not only built upon and extended, but also learned from. I also thank Jared Hoberock, Dave Metts, Vidyasagar Chada and Ian Scott for their past, present and future contributions to the source code.

Lastly, I thank my wife, my family, and my friends and fellow students for all of their input and support.



# CONTENTS

<b>Acknowledgements</b>	<b>ii</b>
<b>List of Figures</b>	<b>ix</b>
<b>Abstract</b>	<b>xi</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Background . . . . .	3
1.3 Credits . . . . .	4
<b>2 Out-of-core Architecture for Real-time Visualization</b>	<b>6</b>
2.1 Overview of Kolam Modules . . . . .	6
2.1.1 Definitions of Images and Tiles . . . . .	7
2.2 Hierarchical Caching — Multi-level Memory Management within the Data Transfer Pipeline . . . . .	8
2.2.1 Disk Cache for Offline Storage . . . . .	8

Pyramid File Format . . . . .	11
2.2.2 Memory Cache for Fast Data Access in Software . . . . .	12
2.2.3 Texture Cache for Fast Data Rendering on GPU . . . . .	15
2.3 The Kolam Engine . . . . .	16
2.3.1 Worker Threads for Background Data Transfer . . . . .	16
2.3.2 Tiles and Requests — A Tile’s Path from Disk to Memory . .	17
<b>3 Visualization Techniques for Large Geospatial Datasets</b>	<b>22</b>
3.1 Overview . . . . .	22
Running Time . . . . .	23
3.2 Orthogonal Viewer . . . . .	24
3.2.1 Visibility Culling . . . . .	25
3.3 Oblique Viewer . . . . .	26
3.3.1 Analytic Solution . . . . .	27
Visibility Culling . . . . .	27
Multiple Resolution Tiles . . . . .	28
3.3.2 Iterative Solution . . . . .	30
Visibility Culling . . . . .	30
Multiple Resolution Tiles . . . . .	35
3.4 Sphere Viewer . . . . .	37
3.4.1 Analytic Solution . . . . .	38
3.4.2 Iterative Solution . . . . .	40
Visibility Culling . . . . .	40
Multiple Resolution Tiles . . . . .	45
3.5 Arbitrary Geometry Viewer . . . . .	46
3.5.1 Visibility Culling . . . . .	47

3.5.2	Hidden Surface Removal . . . . .	48
3.6	Volume Viewer . . . . .	49
3.6.1	Visibility Culling . . . . .	50
3.6.2	Multiple Resolution Tiles . . . . .	51
3.7	Rendering Tiled Image Data . . . . .	52
3.7.1	Raster Renderer — The Bare Essentials . . . . .	52
3.7.2	Texture Renderer — Storing Data on the Graphics Hardware .	52
	Texture Borders — Interpolating Boundary Samples . . . . .	53
3.7.3	Terrain . . . . .	54
<b>4</b>	<b>Kolam User Interface</b>	<b>57</b>
4.1	Windows . . . . .	57
4.2	Scene Components . . . . .	60
4.2.1	Layers . . . . .	60
4.2.2	Colormaps . . . . .	62
4.3	Navigators for Interpreting User Input . . . . .	62
4.3.1	Roam, Zoom and Tilt Navigators . . . . .	63
	Vector vs. Velocity Navigators . . . . .	64
4.3.2	Image Processing Navigators . . . . .	64
4.3.3	Modeling Navigators . . . . .	65
<b>5</b>	<b>Performance Optimizations and Characterization</b>	<b>66</b>
5.1	Performance Optimizations . . . . .	66
5.1.1	Hiding Disk Latency . . . . .	66
5.1.2	Maintaining Constant FPS . . . . .	68
5.1.3	Prioritizing Requests . . . . .	69
5.1.4	Prefetching . . . . .	69

5.2	Performance Characterization . . . . .	70
<b>6</b>	<b>Conclusions and Future Work</b>	<b>73</b>
6.1	Future Work . . . . .	73
6.2	Conclusion . . . . .	75
<b>A</b>	<b>User's Manual</b>	<b>76</b>
A.1	Introduction . . . . .	76
A.2	Kolam Windows . . . . .	76
A.2.1	Display Window . . . . .	78
A.2.2	Overview Window . . . . .	78
A.2.3	Cache View Window . . . . .	79
A.3	Editor Dialogs . . . . .	80
A.3.1	Layer Dialog . . . . .	80
	Layer Tab . . . . .	80
	Layer Information Tab . . . . .	81
	Navigation Tab . . . . .	83
	Rendering Tab . . . . .	83
	Histogram Tab . . . . .	85
A.3.2	Colormap Dialog . . . . .	87
	Colormap Tab . . . . .	87
	Colormap Information Tab . . . . .	87
	Colormap Creation Tab . . . . .	87
A.4	Menus and Toolbars . . . . .	93
A.4.1	Menus . . . . .	93
A.4.2	Toolbars . . . . .	94
A.5	Keyboard Shortcuts . . . . .	95

A.6	Command Line Options . . . . .	95
A.6.1	Global Options . . . . .	95
A.6.2	Layer Options . . . . .	96
A.7	Pyramid File Processing Tools . . . . .	97
A.7.1	16sto16u . . . . .	97
A.7.2	16uto8u . . . . .	97
A.7.3	chopFooter . . . . .	97
A.7.4	chopHeader . . . . .	98
A.7.5	extractChannel . . . . .	98
A.7.6	extractLevel . . . . .	99
A.7.7	interleaveRaw . . . . .	99
A.7.8	jpgToHybrid . . . . .	100
A.7.9	pyramidCompare . . . . .	100
A.7.10	pyramidInfo . . . . .	101
A.7.11	removeChannel . . . . .	101
A.7.12	writeTiledPyramid . . . . .	102
<b>B</b>	<b>Developer’s Manual</b>	<b>104</b>
B.1	Building the Kolam Application from Source . . . . .	104
B.1.1	Platforms and History . . . . .	104
B.1.2	Third-party Dependencies . . . . .	104
	Required Dependencies . . . . .	104
	Optional Dependencies . . . . .	105
B.1.3	Compilers . . . . .	106
B.2	Writing Custom Plug-in Application Extensions . . . . .	107
B.2.1	Plugin Loading and Initialization . . . . .	107

B.2.2	Application Data Access and Processing Hooks . . . . .	108
B.2.3	Attaching Custom Data to Layers and Colormaps . . . . .	109
B.3	File Formats . . . . .	111
B.3.1	Pyramid File Format . . . . .	111
B.3.2	Colormap File Format . . . . .	112
B.3.3	Kolam Raw Header File Format . . . . .	113
<b>Bibliography</b>		<b>115</b>

# LIST OF FIGURES

2.1	Hierarchy of Caches . . . . .	9
2.2	Temporal vs. Spatial Paging . . . . .	13
2.3	Worker Thread Data Flow Diagram . . . . .	18
2.4	Single-Priority LRU Queue Example . . . . .	20
3.1	Intersection of View Frustum and Image Plane . . . . .	29
3.2	Distance from the Eye to a Tile at Actual Resolution . . . . .	31
3.3	Image Vs. Tile Coordinates . . . . .	31
3.4	Area of a Quadrilateral . . . . .	37
3.5	Intersection of a Sphere with a Frustum Plane . . . . .	39
3.6	Sphere clipping plane . . . . .	43
3.7	2D Slice of a Sphere . . . . .	44
3.8	Volume Viewer . . . . .	50
4.1	Main Window, Overview, and Cache Glyph . . . . .	59
4.2	Main Window and GUI Windows . . . . .	61

5.1	Multithreading performance test results . . . . .	71
5.2	Cache size performance test results . . . . .	71
A.1	Kolam Windows . . . . .	77
A.2	Layer Tab . . . . .	82
A.3	Layer Information Tab . . . . .	82
A.4	Navigation Tab . . . . .	84
A.5	Rendering Tab . . . . .	84
A.6	Histogram Tab . . . . .	86
A.7	An Unenhanced Image . . . . .	88
A.8	An Enhanced Image . . . . .	88
A.9	Colormap Tab . . . . .	89
A.10	Colormap Information Tab . . . . .	90
A.11	Colormap Creation Tab . . . . .	90
A.12	World Lights Colormap . . . . .	91
A.13	Cloud Colormap . . . . .	91
A.14	Cloud Colormap with Elevation Data . . . . .	92
A.15	Cloud and World Lights Colormaps with Grid and Elevation Data . .	92
A.16	Main Menu and Toolbar . . . . .	93
B.1	HistogramPlugin Header File . . . . .	110
B.2	Example Kolam Raw Header File . . . . .	114



# Abstract

The visualization of extremely large multi-dimensional datasets requires highly scalable geometric algorithms. We consider an algorithm to be scalable if its complexity remains constant independent of the size of the complete dataset. The complexity of the algorithm should only depend upon the visible volume and display resolution.

This thesis develops several algorithms for the display of 2-D and 3-D datasets that achieve scalable performance. We present approaches for visibility culling and level of detail calculation for large datasets in orthogonal and oblique projections. These techniques are extended to support visualizing geophysical data on a sphere with terrain elevation data, as well as volumetric data.

The algorithms presented herein are implemented on top of an existing out-of-core image tile caching and paging system known as Kolam, developed at the University of Missouri, Columbia. Discussions of Kolam's architecture are provided, which include image representations, tile request methods, cache structures, and thread interactions. A detailed user interface description is included as well, covering GUI components, navigation modes, API functions, and a third-party extension framework.

# CHAPTER 1

## Introduction

### 1.1 Motivation

The amount of data being produced by scientists and technicians in a wide variety of fields has been increasing steadily over the past decade, and this trend is likely to continue. As sensor technology improves and the cost of storage decreases, the task of producing and archiving data on a large scale is becoming feasible for more and more organizations.

Imaging companies using satellites such as IKONOS and MODIS produce panchromatic and multispectral 2D geospatial imagery that can be hundreds of gigabytes or more in size. Images of this magnitude are typically anything from 30 meter per pixel resolution of the continental US to 1 meter resolution of an entire city. Multiple such images are often georegistered and mosaicked to produce composite images of even greater size [1, 2, 3].

Medical practitioners regularly produce large amounts of data in the form of 3D volumetric datasets during routine scanning procedures using Magnetic Resonance Imaging (MRI), Computed Axial Tomography (CAT or CT), Positron Emission Tomography (PET), and Ultrasound [4]. The Visible Human Project is a particularly notable example where two human bodies, both male and female, were digitized using MRI, CT and anatomical images. The datasets provide resolutions of 0.33 mm per pixel (1 mm per pixel over the height of the male dataset), and require 40 gigabytes of storage (15 gigabytes for the male) [5].

The size of these datasets makes visualizing them with a traditional approach nearly impossible. Since a dataset of this size is larger than the amount of physical memory on a typical computer system, a specialized visualization algorithm is necessary to extract the relevant portions of the dataset to load into memory. This is accomplished by discarding data outside of a given region of interest (ROI) in a process known as visibility culling. To prevent the entire dataset from being loaded when the entire dataset is within the ROI, the algorithm must also calculate the correct level of detail (LOD) for each data element. The LOD is based on each data element's contribution to the final rendered image, and should indicate the resolution of the data to be loaded.

The focus of this paper will be to provide solutions to the problems of visibility culling and level of detail calculation for arbitrarily large 2D and 3D datasets. For 2D datasets within a 3D world, several geometric image projections are explored as well. In many cases, these solutions will be discussed in the context of their implementation during the development of the Kolam software package (see Section 1.3).

## 1.2 Background

The problem of large data visualization (LDV) has received much attention over the past decade. Both commercial and academic researchers have proposed numerous solutions for a wide range of applications [4, 6, 7, 8, 9, 10, 11].

The traditional and most straightforward approach for extracting a region of interest is to subdivide the image into tiles suitable for display as single textures on graphics hardware. The advantage of this approach lies in its simplicity and speed; a tile may be found on disk using a lookup table and loaded into memory with a single read operation. The disadvantage is that, in most cases, more data is read from disk than is necessary to produce the visual result.

The traditional approach for extracting various level of details is to use a pyramid structure, which caches additional coarser resolutions of the image on disk. The advantage of this approach is that retrieving tiles from disk at coarse resolutions is as fast as retrieving the original tiles since these resolutions have been precomputed. The downside is that most images originate as a scanline order array and must be converted to the pyramid format before reaping this structure's benefits.

Wavelet-based image encoding techniques used in the JPEG-2000 and MrSID file formats have been developed as an alternative to the tiled pyramid structure. These encoding schemes work by recursively downsampling and filtering an image in such a way that any intermediate resolution can be retrieved by performing the correct number of recursion steps during the decoding process. Each recursive step works by dividing the image into subbands, each one quarter the size of the original. Each of these four are filtered such that the top-left image contains the low frequency information, the top-right contains high horizontal frequencies, the bottom-left contains high vertical frequencies, and the bottom-right contains high vertical and horizontal

frequencies. In the next recursive step, the same process is performed on the top-left image.

Several researchers at Silicon Graphics Inc. (SGI) in 1998 provided another solution to the LDV problem that is not based on tiling. Their concept of a *clipmap* caches data surrounding a given point of interest at both the finest resolution and successively coarser resolutions. They use *toroidal addressing* to perform incremental updates to cached data without processing data that remains resident in the cache [12]. Their caching mechanism is inherently spatial since data is loaded based on its proximity to the point of interest.

Our implementation is based on the tiled pyramid concept. While this formulation is suboptimal for reasons described earlier, it has the advantage of ease of implementation and flexibility. With this structure, individual tiles may be compressed using any desired method rather than being limited to wavelet-based algorithms. Also, the caching mechanism is not tied to any particular navigation pattern, so erratic viewpoint manipulation such as instantaneous jumps to distant locations can potentially be supported with high efficiency

### 1.3 Credits

Kolam is a software package designed for interactive visualization of arbitrarily large datasets. Its development has taken place at the University of Missouri Columbia since the year 2000 under the direction of Dr. K. Palaniappan.

The original version of Kolam was written by Joshua Fraser in late 2000. This version was written in C and relied upon OpenGL for rendering and GLUT (GL Utility Toolkit) for window management and event handling. Its only rendering capabilities were raster drawing techniques, which limited the visualization of data

to orthogonal projections of 2D data. GLUT was also not sufficient for providing an intuitive and useful GUI.

A series of dataset processing utilities were written by Jared Hoberock in early 2002. These include pyramid file read/write utilities, image mosaicking tools, and encoding/decoding methods for ZLIB, BZ2, JPEG and JPEG-2000 compression standards.

The current version was written by Ian Roth from summer 2001 to summer 2004. The majority of the original code was rewritten to take advantage of features of C++ and the Qt GUI toolkit (see Section B.1.2). Many additional features were added including texture mapping, support for non-orthogonal projections, and terrain and volumetric data visualization. A Small Business Innovation Research (SBIR) proposal was written by Ian in June 2004 with the help of Dr. K. Palaniappan to obtain funding for continued research and development in large data visualization.

Dr. K. Palaniappan is responsible for much of the vision and inspiration behind this work. He participated in discussions on the direction of the project and techniques for efficiently accomplishing technical objectives. He authored several grant proposals and research papers regarding the software to obtain funding and to disseminate knowledge obtained during the research.

# CHAPTER 2

## Out-of-core Architecture for Real-time Visualization

### 2.1 Overview of Kolam Modules

The goal of Kolam is to provide tools for visualizing any number of arbitrarily large datasets simultaneously while providing interactive frame rates and intuitive navigation controls. Kolam achieves these objectives by delegating them to specialized components.

**Engine** The Kolam engine consists of one or more worker threads that perform data I/O and processing for requests from the client application. It provides a layer of abstraction so that applications can access the image as if it were entirely resident in memory.

**Application** Applications are clients that determine which sections of the data are

needed and request them from the engine. They are responsible for displaying images, processing data and providing user interface components.

**Application Extensions** Application extensions are implemented as plugins to add user-specific features without modifying the application source code. Typical extensions may define file format I/O procedures, data processing algorithms, viewing modes and user interface components.

### 2.1.1 Definitions of Images and Tiles

An *image pyramid* is an arbitrarily large dense multi-dimensional dataset stored at multiple resolutions. Each resolution, or *pyramid level*, is an image consisting of a single resolution. Each pyramid level is segmented into *tiles*.

We denote an image pyramid  $\mathbf{I}$  as a vector of raw images. The functions  $\mathbf{dim}(\mathbf{I})$  and  $\mathbf{res}(\mathbf{I})$  are used to extract the number of dimensions and resolutions of the image. Each vector element is denoted  $I_r$ , where  $r$  is the image resolution index such that  $0 \leq r < \mathbf{res}(\mathbf{I})$ . We represent the size of the image in pixels using the function  $\mathbf{sip}_d(I_r)$  where  $d$  is the dimension index such that  $0 \leq d < \mathbf{dim}(\mathbf{I})$ . Each resolution  $r > 0$  of an image is subject to the constraint that  $\mathbf{sip}_d(I_{r-1}) = 2 \mathbf{sip}_d(I_r)$  for each value of  $d$  such that  $0 \leq d < \mathbf{dim}(\mathbf{I})$ . This implies that each successive resolution is half the size of its predecessor in each dimension.

We denote a tile  $T$  of an image pyramid  $\mathbf{I}$  as an  $n$ -dimensional array of pixels, where  $n = \mathbf{dim}(\mathbf{I})$ . The notation  $T_r$  is used to represent a tile within a specific pyramid level  $I_r$ . The function  $\mathbf{sip}_d(T)$  determines the size of tile  $T$  in pixels. Each tile within an image pyramid is required to be the same size in pixels, and the pixel size in each dimension must be a power of 2. Under these constraints, it is possible to use the image's tile grid as a uniform coordinate system, with the origin at the top-left corner



of the image. Tiles that overlap the image’s right or bottom boundaries may contain padded data. The function  $\text{sit}_d(I_r)$  is used to determine the size of a pyramid level in tile coordinates, which is equivalent to  $\lceil \text{sip}_d(I_r) / \text{sip}_d(T) \rceil$ .

Although some techniques presented throughout the rest of this paper apply to any image dimensionality, we are often interested in viewing 2D and 3D images. Hence it is helpful to introduce a more convenient notation. We denote the width, height and depth of an image  $I_r$  as  $w(I_r)$ ,  $h(I_r)$  and  $d(I_r)$ . These values may be expressed without the resolution as  $w_I$ ,  $h_I$  and  $d_I$  with the assumption that  $r = 0$ . We represent the width, height and depth of a tile  $T$  from an image pyramid  $\mathbf{I}$  similarly as  $w_T$ ,  $h_T$  and  $d_T$ , respectively.

## 2.2 Hierarchical Caching — Multi-level Memory Management within the Data Transfer Pipeline

Kolam builds a hierarchy of caches for each dataset in order to optimize interactivity and performance. Each tile enters at the lowest level of the cache and is copied to higher levels as needed.

Each level of the cache typically holds more data than the next higher level, but the cost of performing IO is more significant on lower levels (see Figure 2.1).

### 2.2.1 Disk Cache for Offline Storage

When an arbitrary image is loaded into Kolam for visualization, it is most likely laid out on disk in scanline order. The tiles displayed by Kolam are  $2^n \times 2^m$  subimages of the original where  $n$  and  $m$  are positive integers. To read such a tile from a scanline ordered image would require  $2^m$  separate disk reads. By segmenting the image into

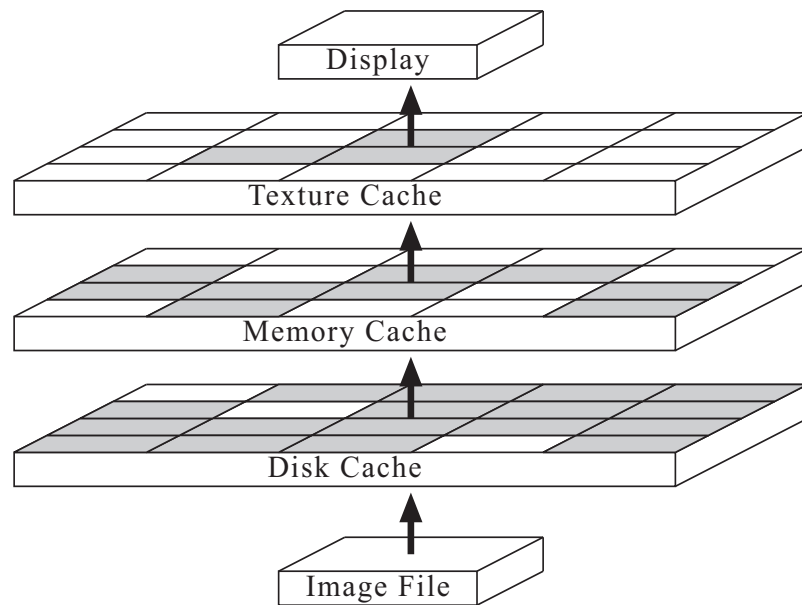


Figure 2.1: Each image tile is cached three times before display: first on disk, then in memory, and finally in texture memory on the graphics hardware. Darkened squares represent filled cache slots in this example.

tiles on disk, each tile requires only one read operation to load it into memory.

The current Kolam implementation builds the entire disk cache for a given image before loading it. This is a slow process that can take several minutes to several days, depending on the size of the dataset. For uncompressed images at the original resolution, the disk cache can be built on-the-fly by reading each requested tile from the original image and immediately writing it to the disk cache. To produce additional coarser resolutions, a more complex scheme is needed.

Coarse resolutions generated from the original can be produced in a variety of ways. These methods are sometimes called downsampling filters. Two of most common downsampling filters use the mean and median of four high resolution pixels to determine the value of one low resolution pixel. Hence a low resolution tile is generated using the four high resolution tiles that compose it.

The naive approach for creating additional resolutions on-the-fly is to use the same method as the cache precomputation algorithm; simply read in all original resolution tiles that compose a given lower resolution tile and downsample them to get the desired low resolution tile. Of course, this is unsatisfactory since it requires the entire image to be read at the original resolution to obtain a single tile at a low enough resolution.

A better approach is to read low resolution tiles as uniform samples from the original image. For a given resolution  $r$  having width and height  $\frac{w}{2^r}$  and  $\frac{h}{2^r}$  where  $w$  and  $h$  are the original image dimensions, every  $r^{\text{th}}$  scanline is read from the original image. Every  $r^{\text{th}}$  pixel from each scanline is then used in constructing resolution  $r$ . This same process is done on a per-tile basis for uncompressed images by only reading  $m$  scanlines of length  $r \cdot n$ .

Although the uniform sampling method serves as an easy way to read a low resolution tile quickly, we may wish to use more visually pleasing downsampling filters

as more high resolution tiles become available in the cache hierarchy. For example, a tile at resolution  $r > 0$  created by uniform sampling can be regenerated using a different downsampling filter once the four tiles that compose it at resolution  $r - 1$  are cached. Low resolution tiles may be signaled for refiltering once enough data is available.

For compressed images, this uniform sampling method becomes considerably more complex. For example, run-length encoding (RLE), one of the simplest encoding methods, creates scanlines of variable length. Hence we can no longer simply read every  $r^{\text{th}}$  pixel since they are not uniformly spaced in the file, nor can we read every  $r^{\text{th}}$  scanline since we must examine the intermediate scanlines to find the position of the next  $r^{\text{th}}$  scanline.

One way to avoid reading every scanline is to scan the entire image during the initial loading step and create a lookup table for the starting position and length of each scanline. If the size of each scanline is very large, the lookup table may also contain one or more breakpoint positions for each scanline, each breakpoint corresponding to a tile boundary.

Precomputing the disk cache remains a useful option since it is far more efficient than constructing the cache on-the-fly. The overhead of refiltering downsampled tiles once more high resolution tiles are available is not an issue for disk cache precomputation. Building a lookup table for scanline positions is not necessary either.

## Pyramid File Format

Kolam currently uses a pyramid file format to represent the disk cache. We define a pyramid format to include an original resolution image of size  $S_0$  along with several coarser resolutions, each additional resolution  $r > 0$  being approximately  $\frac{1}{4}$  the size of the image at resolution  $r - 1$ . The total size of the image data  $S_p$  associated with

a pyramid file (not including header information) is computed by

$$S_p = \sum_{i=0}^{r-1} \frac{S_0}{4^i} \quad (2.1)$$

where  $r$  is the total number of resolutions in the file. Since the base of the exponent  $i$  is less than 1, we conclude that this geometric series [13] converges to

$$\lim_{r \rightarrow \infty} S_p = S_0 \sum_{i=0}^{\infty} \frac{1}{4^i} = \frac{4}{3} S_0 \quad (2.2)$$

as the number of resolutions increases to infinity [1]. This number is only approximate since each image resolution may include border tiles containing padded data.

In addition to image data, Kolam’s pyramid file format includes a header specifying the size and layout of the image data portion. The original image dimensions, the pixel format, the total number of resolutions and the total number of tiles are each stored in addition to a lookup table specifying the location of each tile within the file. See section B.3 for a more detailed description.

For future implementations, we are considering switching to the HDF5 file format [14]. This format supports the same concept of tiles as our pyramid format (except called chunks), but includes several other promising features. The biggest advantage is that the IO library handles tile insertions and deletions automatically, thus making the implementation of on-the-fly disk caching much simpler.

### 2.2.2 Memory Cache for Fast Data Access in Software

The Kolam memory cache is where tiles are held temporarily in RAM. They enter this cache from either an image file or the disk cache and are removed when no longer needed. The process of swapping old tile for new ones is known as *paging*.

The two basic paging strategies are *temporal* and *spatial* strategies. Temporal strategies remove tiles based on how long they have been in the cache. These methods

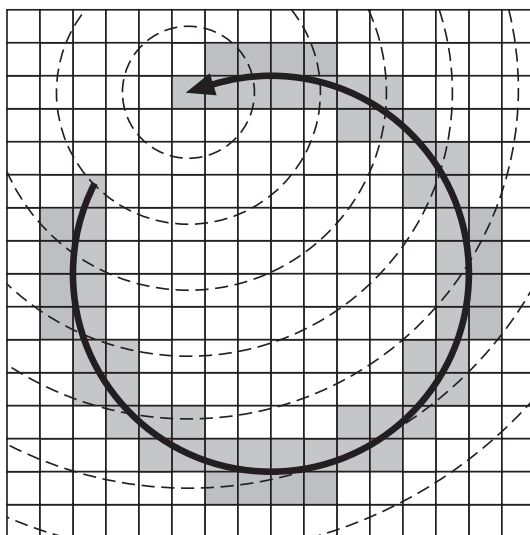


Figure 2.2: An example of a cache and a possible user navigation path. Assuming the cache is full at this point, temporal paging will remove the least recently used tiles even though they are close to the user’s current viewpoint and are likely to become visible soon. Spatial paging will remove those furthest from the user’s current position.

are simple to implement and work well regardless of the visualization technique being used. Spatial strategies remove tiles based on their distance from the visible region of interest (ROI). These methods are sometimes more difficult to implement since the distance between tiles is highly dependent on the method used to project the image onto geometry during the visualization process. However, spatial methods are still preferable in certain cases, such as the one illustrated in Figure 2.2.

Kolam’s temporal paging mechanism uses a least recently used (LRU) queue to remove those tiles that have not been used by the application for the longest period of time. Each time the application uses a tile, the tile is moved to the back of the queue. Tiles that are no longer needed are removed from the front of the queue, a constant time operation. When a tile needs to be moved to the back of the queue,

it is located in constant time by maintaining a pointer for each cache slot indicating the tile's position on the queue. We will assume a LRU caching scheme is used in the rest of this paper.

Prefetching is accomplished with this scheme by using more than one queue, each with a unique priority. Currently visible tiles receive the highest priority, while prefetched tiles have successively lower priorities based on their distance from the visible ROI. Tiles on a queue are not removed until all lower priority queues have been emptied, and tiles are only allowed to move to higher priority queues based on the priority of their request.

The memory cache has five basic methods available to application developers:

**Checkout** This method checks the cache to see if the specified tile is resident in memory and returns a handle to the tile if it is present. If a LRU queue is used, this method also moves the tile to the back of the queue.

**Checkin** After a tile has been successfully checked out, no other threads are allowed to access it until the checkin method is invoked. Every checkout call that returns a non-null value must always be followed by this method or else a deadlock may occur on subsequent checkouts.

**Query** This method allows an application to query the status of a tile without modifying its position on the LRU queue. This is useful for doing cache performance analysis on the application level.

**Add Request** This method places a request for a specific tile. If the tile is already resident in memory or a request for it has already been placed, nothing happens.

**Invalidate Tile** This method removes a tile from the cache and frees its resources. This is useful for removing outdated requests as described in Section 2.3.2.

The memory cache provides three variations of the checkout method:

**Basic Checkout** This method returns immediately with a handle to the tile if it is resident in memory. A null value is returned otherwise.

**Checkout First Available** If the specified tile is already resident in memory, this method works exactly like the basic checkout method. If the tile is not resident, the method searches for the same tile at a coarser resolution. If no coarser resolution is found, a null value is returned.

**Wait for Tile** This method does not return until either the specified tile has been loaded into the cache or a time limit has been reached. The calling thread is suspended until a new tile appears in the cache to avoid busy waiting.

### 2.2.3 Texture Cache for Fast Data Rendering on GPU

The texture cache is a tile's last stop in the cache hierarchy before being displayed. This cache is controlled on the application level since its parameters are highly dependent on the visualization techniques being used.

The texture cache is essentially a simplified version of the memory cache. No synchronization is necessary since the visualization process is confined to a single thread. No requests are needed since transferring tiles from the memory cache to the texture cache requires no system calls or unbounded waiting.

The only major difference between the memory and texture caches is in the way that space is allocated for cache entries. The memory cache allocates tile memory either by using the program's heap space or by segmenting its own internal memory block. The texture cache must store its tile data on the graphics hardware, so the allocation is often done through some external graphics API (OpenGL in our case). A



temporal paging strategy is generally preferred since the limited texture memory on graphics hardware eliminates most advantages of the more complex spatial strategies.

## 2.3 The Kolam Engine

The Kolam engine is the entity that performs low-level data IO and memory management for one or more applications. It is based on the workpile design pattern. The simplest form of this pattern takes a large task and breaks it down into numerous subtasks, dispatching a new thread to handle each subtask. A more efficient implementation will have a limit on the number of concurrently running threads and be able to reuse existing threads rather than spawning new ones for each new task.

### 2.3.1 Worker Threads for Background Data Transfer

Kolam implements the workpile pattern by first spawning a fixed number of worker threads at startup. Each thread is initially idle and remains so until it receives a request. Requests are produced by the application to indicate that a particular piece of data that is not resident in memory is needed for rendering or processing. Once a request is made, an idle thread is awoken to process it. After its work is done, a thread will check to see if any more requests have been queued and are awaiting processing. If not, the thread returns to its idle state.

Worker threads have several states that they can be in at any given time:

**Idle** A worker thread is idle when there are no requests to be processed. Threads do not periodically check the request queue, a process known as *busy waiting*. Instead they suspended themselves upon finding an empty queue and are awoken whenever a new request is added by the thread making the request.

**Caching/Paging** In this state, the worker thread attempts to find a free block of memory in the cache to store incoming tile data. If no free space is available, some cached data must be discarded to make room (see Section 2.2.2).

**Reading/Writing** A worker thread is reading or writing when tile data I/O is being performed. While currently only disk/memory I/O capabilities are implemented in Kolam, this state applies equally well to I/O to and from a network, printer, scanner, digital camera, or other I/O device.

**Notifying** After the requested I/O operation is complete, the thread notifies the application that new data is available. This will usually trigger a display refresh, although any other response can be implemented at the application level.

### 2.3.2 Tiles and Requests — A Tile’s Path from Disk to Memory

Each cache slot is a pointer that may refer to a tile, a tile request, or a null pointer. These values correspond to three tile states of resident, requested, and empty. Since both tiles and requests share the same data structure and are often treated identically, we will refer the union of both groups as *cache entries*. A cache entry is uniquely identified by its *image* and *index*. It is treated as either a tile or a request based on its state. Cache entries have several states that they can be in at any given time:

**Inactive** This state corresponds to a non-existent cache entry. This value is returned by the cache whenever a query is made to a slot containing a null pointer.

**Requested** In this state, the cache entry is a request rather than a tile and is guaranteed to be on the request queue.

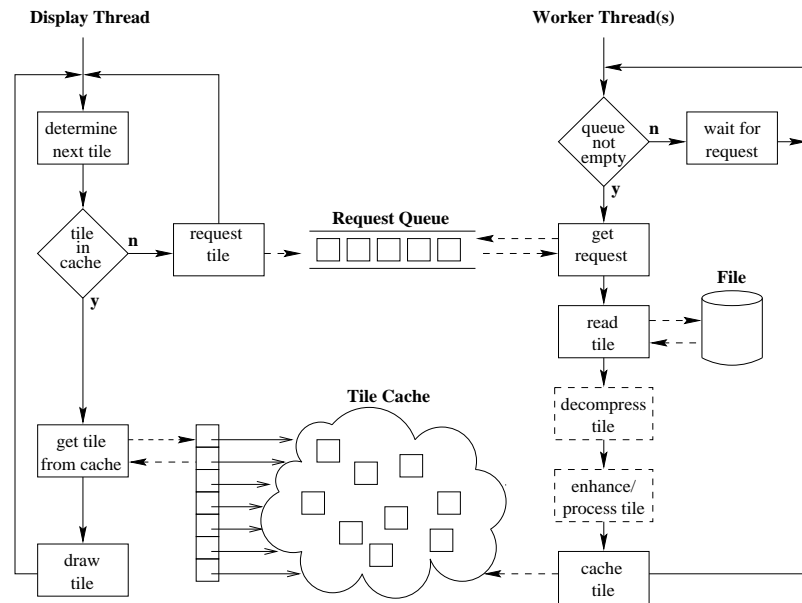


Figure 2.3: This diagram shows the interaction between the display thread and multiple worker threads. The display thread determines which tiles should be rendered and places requests onto a queue. The worker threads process these requests by loading tiles from the disk cache and inserting them into the memory cache. Image courtesy of [15].

**Processing** In this state, the cache entry is a tile rather than a request and is either preparing for or actively performing data I/O.

**Resident** After the requested I/O operation is complete, the tile is fully resident in memory and ready to be used by the application.

Kolam maintains two main memory cache representations at all times: an array of cache slots that may or may not have data associated with them and two priority queues to keep track of current cached tiles and tile requests. Requests are always stored on a priority queue whereas cached tiles may or may not be depending on whether a LRU caching scheme is being used (see Section 2.2.2). Here a LRU caching scheme will be assumed.

The array representation allows constant time lookup of tile data, while the queues provide linear time traversal of cached and requested tiles. By combining the two, constant time push, pop and remove operations can be implemented on the priority queue (see Figure 2.4). For a cache with a constant tile size, these are the only operations required and all caching and paging operations can be performed in constant time. For tiles of varying sizes, several tiles near the tail of the queue may need to be popped for a single paging operation before enough memory has been freed for incoming data. Since tile sizes often do not vary dramatically, the number of additional pops is usually small.

Kolam uses a priority queue to manage both tiles and requests. This queue is more accurately described as an array of queues, each with a unique priority value. Multiple priorities are especially useful for prefetching tiles as currently visible tiles should always have a higher priority than non-visible tiles.

When the application makes a request for some tile from the image, the request is put onto a queue to await processing by a worker thread. This becomes a problem

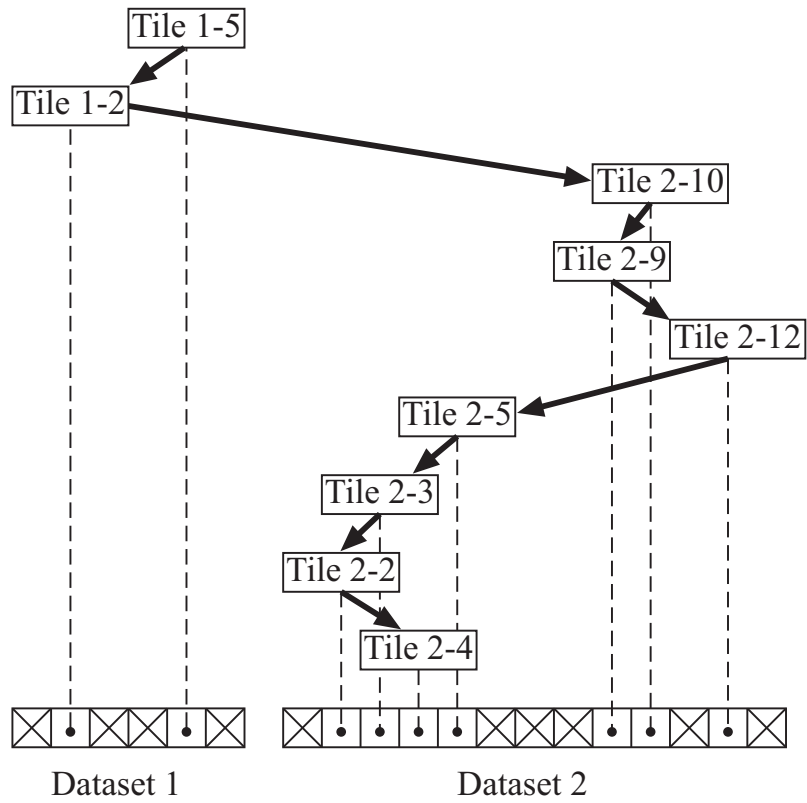


Figure 2.4: An example of a single-priority least recently used (LRU) queue with constant time push, pop and remove operations. Random access is provided using the associated cache entry lookup tables (bottom).

when requests are produced faster than the threads can consume them. The queue structure forces the threads to process the oldest requests first, in turn increasing the time needed for the user to see the visible tiles. This problem can be solved by using a stack instead of a queue, but this still requires the threads to waste time processing old requests after the new ones.

The solution is to let the application maintain two lists: one for the visible tiles in the current frame, another saved from the previous frame. At each frame, the application compares the two lists to find the tiles that have ceased to be visible between the last two frames. The requests for these tiles are then removed from the request queue.

# CHAPTER 3

## Visualization Techniques for Large Geospatial Datasets

### 3.1 Overview

The viewer components in Kolam are responsible for determining which tiles are visible and at what resolution. Once these tiles are determined, they are requested from the core server. The server will either respond with a block of image data to be drawn or indicate that the requested tile is not yet resident in memory. In the latter case, the next available tile covering the same region at a coarser resolution may be requested by the viewer. If no coarser resolution is available, the viewer will have no choice but to display no data. However, this case is rare since the overview tiles (the coarsest resolution) are never discarded. For more information on retrieving data from the cache, see section 2.2.2.

## Running Time

The goal of Kolam is to be able to handle *any* size dataset, not withstanding size limitations of the operating system or physical storage device. Keeping this in mind, the efficiency of the visualization algorithms can be measured by the number of tiles processed per frame (this may, and probably will, be different than the number of tiles *rendered* per frame). For convenience, we define the value

$$n_r = \prod_{d=0}^{\dim(\mathbf{I})-1} \text{sit}_d(I_r) \quad (3.1)$$

to be the number of tiles at resolution  $r$ . We also define

$$n = \sum_{r=0}^{\text{res}(\mathbf{I})-1} n_r \quad (3.2)$$

to be the total number of tiles in a given image (see Section 2.1.1).

We also wish to define the total number of visible tiles for any given frame since this number is the minimum number of tiles that can be process per frame. While this number does not have a well-defined minimum or maximum value for arbitrary image projections, it is always well-defined at *a single frame*. Therefore we will represent it as a time-variant function  $m(t)$ , or simply  $m$  in the context of a single frame.

A visualization algorithm that processes  $O(n)$  tiles per frame will scale linearly as the number of tiles  $n$  grows arbitrarily large. Processing only the finest resolution yields the same complexity since  $n_0 \approx \frac{3}{4}n = O(n)$  (see Section 2.2.1). While linear running time is generally considered quite good for most algorithms, it is more time than is necessary in our case since we are only interested in rendering  $m$  visible tiles.

Since some image projections presented in this paper can be represented analytically (i.e. planes, spheres and hexahedra), we can compute their intersection with the viewing volume in constant time relative to  $n$ . The intersected region can then be projected into tile coordinates, thereby yielding the set of visible tiles. If all tiles



are at the same resolution and approximately equidistant from the eye, the time taken to render one frame using this algorithm will be  $O(m)$ . To account for multiple resolutions, we can subdivide the viewing volume into different regions based on their distance from the eye. Since a 2D image has approximately  $\log_4 n$  resolutions, this new formulation can require  $O(\lg n)$  separate intersection calculations, yielding a running time of  $O(m + \lg n)$ .

In practice, it is usually far simpler to implement an algorithm that determines tile visibility using an iterative approach over each resolution of an image. Such algorithms will run in a somewhat slower  $O(m \lg n)$  time (this will be justified later at the end of Section 3.3.2). This approach has a big advantage over the previous formulation in that we do not have to derive a separate intersection scheme for every type of image geometry. This geometry can also be represented as a discrete or piecewise function, for example a triangular mesh, rather than as an implicit surface or volume.

In the next few sections, if it suffices to render all tiles at the same resolution, an  $O(m)$ -time algorithm will be presented. If multiple levels of detail may exist within the same view, an iterative  $O(m \lg n)$ -time algorithm will be presented. If an image projection can be represented implicitly, an analytical  $O(m + \lg n)$ -time algorithm will be presented also.

## 3.2 Orthogonal Viewer

The orthogonal viewer is the simplest and often the most useful for 2D imagery. It acquires its name from its restriction that, given a camera located at  $\mathbf{c}_{eye}$  with focal point  $\mathbf{c}_{center}$ , the vector

$$\mathbf{n} = \mathbf{c}_{center} - \mathbf{c}_{eye} \tag{3.3}$$

is always orthogonal to the image plane. Most image processing programs use this and only this viewing paradigm since it maintains a uniform sampling of image pixels.

A translation vector  $\boldsymbol{\tau}$  may be applied by computing  $\mathbf{c}_{center} = \mathbf{c}_{center} + \boldsymbol{\tau}$  and  $\mathbf{c}_{eye} = \mathbf{c}_{eye} + \boldsymbol{\tau}$ . Scaling the image by a diagonal matrix  $\mathbf{S}$  may be accomplished by computing  $\mathbf{c}_{eye} = \mathbf{c}_{center} + \mathbf{S}\mathbf{n}$ . Arbitrary rotations about  $\mathbf{n}$  may be applied using successive rotations about the  $x$ ,  $y$  and  $z$ -axes. As shown in [16], these rotations can be combined to form a generic system of linear equations for performing vector rotations.

In practice, it is often assumed that the image plane is orthogonal to the  $z$ -axis and that the vector  $[0, 1, 0]^T$  (the positive  $y$ -axis) indicates the user's upward orientation. In this case, the camera representation may be discarded in favor of a unit eye distance from the image. Under these assumptions, the above formulation can be simplified considerably.

### 3.2.1 Visibility Culling

If it is assumed that the viewport is rectangular, a list of visible tiles can be determined by a series of simple affine transformations. For non-rectangular viewports, the more general tile culling method presented in the next section (see Section 3.3) would be more appropriate.

Assume we are given a 2D translation vector  $\mathbf{t}$  in image pixel coordinates and a zoom factor  $z$  such that  $z \geq 1$  at resolution  $r = 0$ , and  $1/2^r \leq z < 1/2^{r-1}$  for all coarser resolutions  $0 < r < \mathbf{res}(\mathbf{I})$ . All tiles will be at the same resolution since each point on the image plane is equidistant from its corresponding point in the parallel viewing plane. Hence a global image resolution  $\rho$  can be computed by

$$\rho = \lceil \log_2(1/z) \rceil. \tag{3.4}$$

The global scale should be clamped to the range  $0 \leq \rho < \mathbf{res}(\mathbf{I})$  to ensure a valid resolution index is produced.

The translation vector  $\mathbf{t}$  can then be transformed from pixel coordinates to tile coordinates using the transformation matrix

$$\mathbf{S} = \frac{1}{2^\rho} \begin{bmatrix} 1/w_T & 0 \\ 0 & 1/h_T \end{bmatrix}. \quad (3.5)$$

We can also compute the visible region  $R$  in tile coordinates so that each integral coordinate contained in  $R$  is a visible tile at resolution  $\rho$ . Since the region  $R$  is a rectangle in an orthogonal projection, it is sufficient to determine its width  $w_R$  and height  $h_R$ . This is accomplished by computing

$$\begin{bmatrix} w_R \\ h_R \end{bmatrix} = \frac{1}{z} \mathbf{S} \begin{bmatrix} w_V \\ h_V \end{bmatrix} \quad (3.6)$$

where  $w_V$  and  $h_V$  are the viewport width and height. Combining equations (3.5) and (3.6), we can obtain the top-left tile coordinate  $(x_R, y_R)$  of the rectangular region  $R$  using

$$\begin{bmatrix} x_R \\ y_R \end{bmatrix} = \mathbf{S} \mathbf{t} - \frac{1}{2} \begin{bmatrix} w_R \\ h_R \end{bmatrix}. \quad (3.7)$$

### 3.3 Oblique Viewer

The oblique viewer is similar to the orthogonal viewer in the respect that each visualizes a dataset confined to a plane. However, the oblique viewer relaxes the constraint that the camera angle must be orthogonal to the image plane. As a consequence, most assumptions that allow us to simplify the tile culling formulation of the orthogonal viewer are no longer valid. First of all, the image plane and viewing plane are not necessarily parallel, so tiles of different resolutions may be present in the same visible

region. Second, although the viewport is still rectangular in 3D, it is likely to become deformed once projected onto image coordinates.

### 3.3.1 Analytic Solution

#### Visibility Culling

The analytic solution to image tile culling calculates the intersection of the view-frustum and the image plane using implicit equations.

We begin with the simple case where it is assumed that each tile is to be displayed at the same resolution  $r$ . Each plane of the viewing frustum is represented by the equation

$$\mathbf{n}_F \cdot (\mathbf{x} - \mathbf{p}_F) = 0 \tag{3.8}$$

where  $\mathbf{n}$  is the plane's unit normal vector (facing inward w.r.t. the frustum) and  $\mathbf{p}$  is any point on the plane. The image plane is defined similarly as

$$\mathbf{n}_P \cdot (\mathbf{x} - \mathbf{p}_P) = 0. \tag{3.9}$$

The intersection between this view frustum plane and the image plane can be calculated by equating the two planar equations (both are equal to zero) and solving for  $\mathbf{x}$ . The resulting system of equations is underdetermined, but a solution can be found by picking an initial value for a single component of  $\mathbf{x}$ . The vector  $\mathbf{m}$  parallel to the line of intersection is found by

$$\mathbf{m} = \mathbf{n}_F \times \mathbf{n}_P. \tag{3.10}$$

The parametric equation for the line is given by

$$\mathbf{x}(t) = \mathbf{p} + t\mathbf{m}. \tag{3.11}$$

where  $\mathbf{p}$  is the renamed value of  $\mathbf{x}$  from equations (3.8) and (3.9).

The lines of intersection of each of the six view frustum planes can then be intersected in the image plane to find set of points. The view frustum lines should also be intersected with the edges of the image in the plane. The intersection of two lines is given by

$$\mathbf{x} = \mathbf{p}_1 + \mathbf{m}_1 \frac{(\overrightarrow{\mathbf{p}_1 \mathbf{p}_2} \times \mathbf{m}_2) \cdot (\mathbf{m}_1 \times \mathbf{m}_2)}{|\mathbf{m}_1 \times \mathbf{m}_2|^2}. \quad (3.12)$$

The convex hull of all of these points yields a polygon containing the region of visible tiles in the image.

Once the polygon is found in the image plane, any standard polygon fill algorithm can be used to traverse the tiles within the polygon boundaries. Scanline filling methods are directly applicable. Flood fill methods may be used if the polygon edges are first discretized by marking individual tiles as boundary tiles using the DDA or Bresenham line drawing algorithms [17].

### Multiple Resolution Tiles

We can now expand upon this formulation to incorporate tiles of varying resolutions within the same framework. We do so by dividing the view frustum into regions based on their distance from the eye. Each region can be intersected with the image plane individually, and the tiles within each resulting polygon can be rendered at that region's resolution (see Figure 3.1).

We can split the view frustum into regions using either planes or spheres. Planes are simpler, although tiles near the left, right, top and bottom clipping planes may be calculated to be closer than they actually are. Spheres provide a more exact calculation at the expense of added complexity.

Regardless of whether planes or spheres are used to divide the frustum, the distance from the eye to the division must be calculated for each available image resolution. We start by calculating the distance  $p$  in pixel coordinates (1 unit =

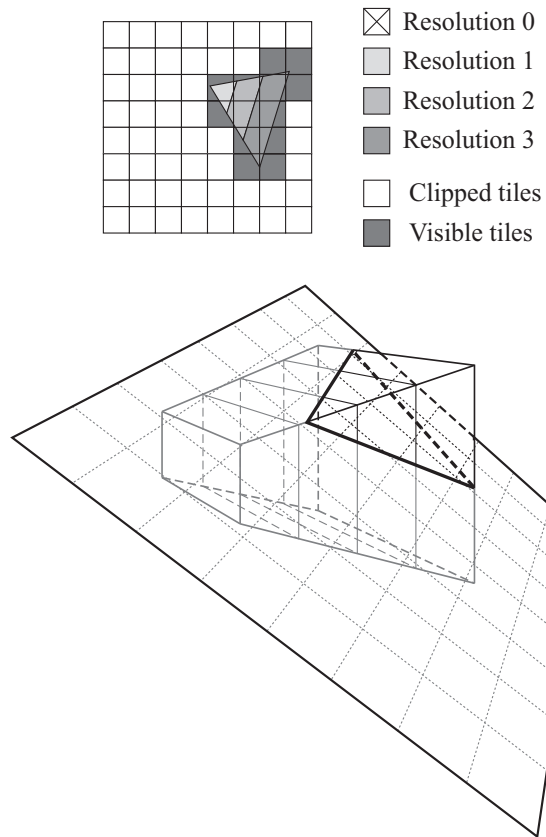


Figure 3.1: The intersection of the view frustum with the image plane show in 2D (top) and 3D (bottom).

$\frac{\text{tile width in global coords}}{\text{tile width in pixels}}$ ) from the eye to a point in the image plane displayed at its actual resolution. Imagine a tile orthogonal to the viewing vector with the same width and height as the display viewport that fills the entire screen pixel-for-pixel when rendered (see Figure 3.2). This tile is at actual resolution and its distance can be calculated by

$$p = (h_V/2) \cot \theta. \quad (3.13)$$

Given a distance  $d$  in pixel coordinates, we can now use  $p$  to scale it to resolution coordinates. The resolution of any point in space can be obtained by taking the log of this scaled distance using

$$r = \lfloor \log_2(d/p) \rfloor. \quad (3.14)$$

By solving for  $d$  using

$$d = p2^r \quad \text{for } r = 0, 1, 2, \dots, \text{res}(\mathbf{I}) - 1 \quad (3.15)$$

we get the distance from the eye of the plane or sphere used to divide the view frustum into regions.

### 3.3.2 Iterative Solution

#### Visibility Culling

One way to determine which tiles are visible is to mimic the approach used by well-known scene graph culling algorithms. The different resolutions of the visible image form a *quadtree* if each tile at resolution  $r > 0$  is considered to have at most 4 children at resolution  $r - 1$ . A tile  $t$  with tile coordinates  $\{x, y\}_t$  could have children with coordinates  $\{2x, 2y\}_t$ ,  $\{2x + 1, 2y\}_t$ ,  $\{2x, 2y + 1\}_t$ , and  $\{2x + 1, 2y + 1\}_t$ . In image coordinates  $\{x, y\}_i$ ,  $t$ 's children could have coordinates  $\{x/2, y/2\}_i$ ,  $\{(x + 1)/2, y/2\}_i$ ,  $\{x/2, (y + 1)/2\}_i$ , and  $\{(x + 1)/2, (y + 1)/2\}_i$  (see figure 3.3). Since the boundary of

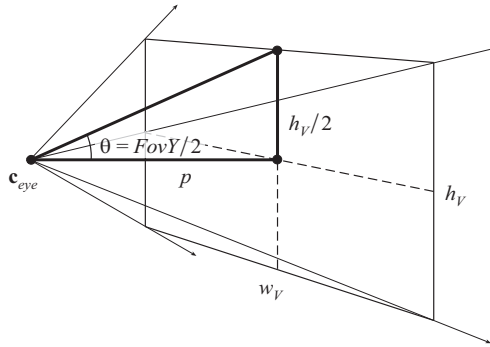


Figure 3.2: A tile with the same width and height as the viewport  $\{w_V, h_V\}$  is displayed at its actual resolution (pixels are the same size on the tile as on the viewing plane) when its four edges touch but do not cross the left, right, top and bottom planes of the viewing frustum. The distances  $w_V$ ,  $h_V$  and  $p$  are measured in pixels.

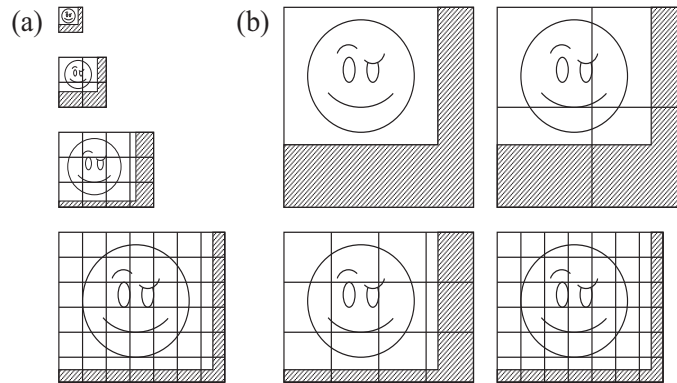


Figure 3.3: Tile coordinates shown in (a) vary depending on the resolution while image coordinates in (b) are resolution invariant.

each child is completely contained by the boundary of any of its ancestors *in image coordinates*, we can say that if no part of  $t$  is visible, then no part of any of its children is visible either.

We can exploit this property of multi-resolution images by performing a top-down



search through the image tiles, i.e. a search from coarsest to finest resolution. For each tile, we check to see whether or not it is visible and only continue searching its children if the tile itself is visible. For now, we will assume that the search will terminate when some constant resolution is reached.

Here we present scene graph rendering algorithms (3.3.2) and (3.3.2) for rendering all visible tiles at the finest resolution. This approach will be modified slightly in the next section to reflect view-dependent tile resolution calculations.

---

**Algorithm 1** GetVisibleTiles

---

```

for all tiles  $t$  at resolution  $r_{max}$  do
    ProcessTile(  $t$  )
end for

```

---



---

**Algorithm 2** ProcessTile(  $t$  )

---

```

if  $t$  is visible then
    if  $r_t = 0$  then
        render tile  $t$ 
    else
        for all children  $c$  of tile  $t$  do
            ProcessTile(  $c$  )
        end for
    end if
end if

```

---

We can determine whether or not a vertex of a given tile is visible by performing a standard view-frustum intersection test. This test can be performed in either  $\mathbb{R}^3$  or in the canonical view volume (CVV) space. Each has its advantages and disadvantages which we will discuss shortly. In either case, we begin by transforming our world

coordinates using a transformation function  $f(x)$ .

To perform the intersection test in  $\mathbb{R}^3$ , we only need to transform our world coordinates into eye coordinates. We define a transformation function  $f(\mathbf{x}) : \mathbb{R}^3 \rightarrow \mathbb{R}^3$  on a vertex  $\mathbf{x}$  by

$$f(\mathbf{x}) = \mathbf{M} \mathbf{x} \tag{3.16}$$

where  $\mathbf{M}$  is a standard modelview matrix [18]. If the transformed vertex  $\mathbf{x}' = f(\mathbf{x})$  lies within the six planes of the viewing frustum, we conclude that it is visible.

To perform the intersection test in CVV space, we need to perform an additional non-linear transformation using homogeneous coordinates to fit the entire visible region into the unit cube defined by  $[-1, 1]^3$ . We define a transformation function  $f(\mathbf{x}) : \mathbb{R}^3 \rightarrow \mathbb{R}^4$  on a vertex  $\mathbf{x}$  by

$$f(\mathbf{x}) = \mathbf{P} \mathbf{M} \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix} \tag{3.17}$$

where  $\mathbf{M}$  and  $\mathbf{P}$  are standard modelview and projection matrices [18]. If the transformed vertex  $\mathbf{x}' = f(\mathbf{x})$  lies within the canonical view volume (CVV)  $[-1, 1]^3$ , we conclude that it is visible. This test can be performed in either homogeneous coordinates or after the homogeneous division step [17].

We will first describe the view-frustum intersection test in  $\mathbb{R}^3$ . The view frustum has six planes, each of which has a corresponding equation

$$\mathbf{n} \cdot \overrightarrow{\mathbf{p}\mathbf{x}} = 0 \tag{3.18}$$

where  $\mathbf{n}$  is the plane's unit normal vector and  $\mathbf{p}$  is any point on the plane. Equation (3.18) is satisfied if  $\mathbf{x}$  lies on the plane. If  $\mathbf{x}$  does not lie on the plane, the sign of the left side of equation (3.18) determines whether  $\mathbf{x}$  is inside or outside the plane. If  $\mathbf{x}$  lies inside each of the frustum's six planes, it intersects the frustum's volume.

We can use the results of the vertex visibility test to determine if any part of a tile is visible. Obviously, any of a tile's four vertices passing the visibility test implies that part of the tile is visible. However, if no vertices of a tile are visible, at least one edge or interior point may still be visible.

We can check for edge visibility whenever two connected vertices lie on opposite sides of one of the frustum's six planes. We then compute the intersection point of the edge with the plane and check whether this point passes the vertex visibility test, thereby implying a visible edge. This can be accomplished by substituting the parameterized line segment equation formed by the two opposing vertices into the equation for each plane. The equation for the parameterized line segment  $\overline{\mathbf{q}_1\mathbf{q}_2}$  is given by

$$\mathbf{x} = \mathbf{q} + t\mathbf{v} \quad (3.19)$$

where  $\mathbf{q}$  is one of the two vertices on the line segment, i.e.  $\mathbf{q}_1$ , and  $\mathbf{v}$  is a vector parallel to the line, i.e.  $\overrightarrow{\mathbf{q}_1\mathbf{q}_2}$ . We substitute equation (3.19) into (3.18) and solve for  $t$  to obtain

$$t = \frac{\mathbf{n} \cdot (\mathbf{p} - \mathbf{q})}{\mathbf{n} \cdot \mathbf{v}}. \quad (3.20)$$

By substituting equation (3.20) into (3.19) and solving for  $\mathbf{x}$  we get the point of intersection between the line segment and the plane. We then verify that this point lies within the quadrilateral created by the intersection with the current plane and its four neighboring planes on the frustum. This can be accomplished using the vertex/plane intersection test described above. If the point of intersection lies within the quadrilateral, it lies on the frustum boundary and hence the line segment  $\overline{\mathbf{q}_1\mathbf{q}_2}$  intersects the frustum.

Even if no vertices or edges of a tile intersect the frustum, any of its interior points may still intersect. We can perform a simple test for the visibility of the interior of

a tile after both the vertex and edge tests fail. The failure of the previous two tests implies that the tile in question must either fill the entire visible region with its interior (i.e. the intersection of its interior with the left, right, top and bottom planes forms a quadrilateral in  $\mathbb{R}^3$ ) or not be visible at all. We can then perform one additional plane/line segment intersection test, this time intersecting the line segment along the line of sight between the near and far clipping planes with the plane defined by the tile.

Performing the visibility culling test in the transformed CVV space rather than  $\mathbb{R}^3$  has a big performance advantage. For an arbitrary frustum in  $\mathbb{R}^3$ , each component of each plane's normal vector  $\mathbf{n}$  is likely to be non-zero, hence requiring 18 multiplications, 18 subtractions, 12 additions and 6 comparisons. After transforming to CVV space, each frustum plane is aligned with one of the three axes of  $\mathbb{R}^3$ , thus yielding values of  $\mathbf{n}$  with one unit and two zero terms. The result is that two multiplications cancel out due to zeros and the other is unnecessary because the factor is one, so no multiplications need to be performed at all. Since the sides of the CVV are all constant distances from the origin, the additions and subtractions can be eliminated yielding only a single comparison for each plane. Hence only six comparisons is all that is necessary to determine if a vertex intersects the view frustum. A similar strategy can be used to simplify the plane/line segment intersection test in CVV space.

### **Multiple Resolution Tiles**

The approaches for tile culling listed above are limited in that they only determine the appropriate tiles to render in spatial coordinates. Additional steps must be taken in either approach to ensure that the correct resolution of each tile is chosen.

For the scene graph approach, determining the correct resolution is fairly straight-

forward. During traversal of the graph, instead of continuing to process children of a visible tile  $t$  until a leaf node is reached, we can check whether the current resolution is fine enough for rendering before continuing.

This method is illustrated in algorithm (3.3.2), a modification of algorithm (3.3.2).

---

**Algorithm 3** ProcessLODTile(  $t$  )

---

```

if  $t$  is visible then
    if  $t$  is at a fine enough resolution then
        render tile  $t$ 
    else
        for all children  $c$  of tile  $t$  do
            ProcessTile(  $c$  )
        end for
    end if
end if

```

---

We can determine if the current resolution is fine enough by simply calculating the area of the quadrilateral projected into window coordinates from each tile  $t$ . Due to the perspective transformations used for projection, we cannot make any assumptions about the projected quadrilateral, i.e. it is not guaranteed to have any parallel edges. Therefore we must use the generic quadrilateral area formula

$$A = \frac{1}{4} \sqrt{4p^2q^2 - (b^2 + d^2 - a^2 - c^2)^2} \quad (3.21)$$

where  $a$ ,  $b$ ,  $c$  and  $d$  are the lengths of each edge in counter-clockwise order and  $p$  and  $q$  are the distances between opposite vertices (see Figure 3.4) [19]. If the area  $A$  is above some threshold  $T$ , the children of tile  $t$  can be traversed to achieve a finer resolution.

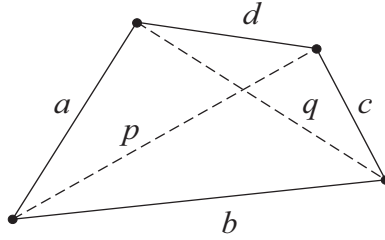


Figure 3.4: The quantities  $a$ ,  $b$ ,  $c$  and  $d$  are the lengths of each edge, while  $p$  and  $q$  are the distances between opposite vertices.

The value of  $T$  is typically chosen in the range  $S \leq T \leq 4S$  where  $S$  is the number of pixels used by a tile. Setting  $T = S$  will increase the likelihood that no tiles are displayed more coarsely than their actual resolution.

In section 3.1 we stated without proof that this algorithm will run in  $O(m \lg n)$  time. Here  $\lg n$  is the maximum height of the graph and  $m$  is the number of visible tiles. For each tile, a maximum of 4 children can be processed, so at least  $4m$  tiles are guaranteed to be processed per frame. Since each tile has at most  $\lg n$  ancestors, each of which may require up to 4 children to be processed, the graph traversal of each visible tile can require no more than  $4 \lg n$  tiles to be processed. For a total of  $m$  visible tiles, no more than  $4m \lg n$  tiles can be processed. Therefore we can state that the upper bound on the running time of this algorithm is  $O(m \lg n)$ .

### 3.4 Sphere Viewer

The sphere viewer takes a 2D image and projects it onto a spherical shape approximated by a triangular mesh. We still maintain the concept of a 2D image  $I$ , but each tile  $t$  not only has an  $\{x, y\}$  offset in the image plane, but also a latitude and longitude pair  $\{u, v\}$ . While latitudes and longitudes are usually measured in degrees in the

range  $[-90, 90]$  and  $[-180, 180]$ , we will use radian values in the range  $[-\pi/2, \pi/2]$  and  $[0, 2\pi]$  to simplify our calculations. Vertices on the surface of a sphere can be calculated using parametric equations defined in [20] by

$$\mathbf{x} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} \cos v \cos u \\ \sin v \\ \cos v \sin u \end{bmatrix}. \quad (3.22)$$

The trigonometric functions are prohibitive to calculate for every tile in every frame, so each tile's position  $\mathbf{x}$  on the sphere can be precalculated. To reduce the space requirement from  $\Theta(n)$  to  $\Theta(w_I + h_I)$ , the trigonometric functions of  $u$  and  $v$  can be precomputed and then combined using equation (3.22) for each frame.

### 3.4.1 Analytic Solution

We can directly calculate the intersection of the viewing frustum and the sphere onto which the image is projected using similar methods to those described in section 3.3. There are a few key differences, mainly in the determination of the polygon in the image plane which overlaps the region of visible tiles.

The intersection of a view frustum plane with the image sphere is a circle in 3D space. This circle is defined by the same center and radius as is used to define a 2D circle, except that it is restricted to lie in the view frustum plane. We can calculate the circle's center by first finding the distance  $d$  from the sphere's center  $\mathbf{c}_s$  to the frustum plane along the plane's unit normal vector  $\mathbf{n}$ . The value of  $d$  is given by

$$d = |(\mathbf{c}_s - \mathbf{p}) \cdot \mathbf{n}| \quad (3.23)$$

where  $\mathbf{p}$  is any point on the frustum plane. This circle's center  $\mathbf{c}_c$  can then be calculated using

$$\mathbf{c}_c = \mathbf{c}_s - d\mathbf{n}. \quad (3.24)$$

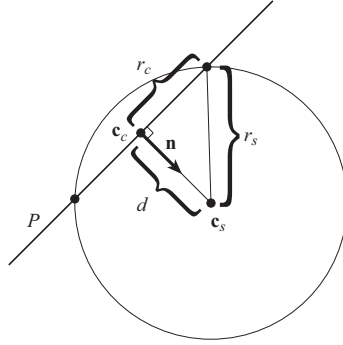


Figure 3.5: Intersection of a sphere with a frustum plane.

By observing figure 3.5 we see that the circle's radius  $r_c$  is related to  $d$  and the sphere's radius  $r_s$  by

$$r_c = \sqrt{r_s^2 - d^2}. \quad (3.25)$$

Now that we have found the circles of intersection, they still need to be projected onto the image plane to be of any use. We do so by defining the parametric equations for a circle and sphere and solving for the spherical parametric coordinates  $u(t)$  and  $v(t)$  in terms of the circle parameter  $t$ .

The parametric equation for a circle in 3D is given by

$$\mathbf{x}(t) = \mathbf{c}_c + r_c \cos t \mathbf{a} + r_c \sin t \mathbf{b} \quad (3.26)$$

where  $\mathbf{a}$  and  $\mathbf{b}$  are any two relatively orthogonal unit vectors that are also orthogonal to  $\mathbf{n}$ . If an “up vector” is defined by  $\mathbf{u}$ , we can use it to determine suitable values for  $\mathbf{a}$  and  $\mathbf{b}$  using

$$\mathbf{a} = \mathbf{n} \times \mathbf{u} \quad (3.27)$$

and

$$\mathbf{b} = \mathbf{n} \times \mathbf{a}. \quad (3.28)$$



The parametric equation for a sphere is given by

$$\mathbf{x}(t) = \mathbf{c}_s + r_s \begin{bmatrix} \cos v(t) \cos u(t) \\ \sin v(t) \\ \cos v(t) \sin u(t) \end{bmatrix} \quad (3.29)$$

where  $\mathbf{c}_s$  is the sphere's center.

Combining these two equations and solving for  $u(t)$  and  $v(t)$ , we obtain two equivalent relations for  $u(t)$

$$u(t) = \cos^{-1} \left( \frac{x_x(t) - c_{s_x}}{r_s \cos v(t)} \right) = \sin^{-1} \left( \frac{x_z(t) - c_{s_z}}{r_s \cos v(t)} \right) \quad (3.30)$$

that are both dependent on  $v(t)$  and a single relation for  $v(t)$

$$v(t) = \sin^{-1} \left( \frac{x_y(t) - c_{s_y}}{r_s} \right) \quad (3.31)$$

that can be solved directly. By picking sample values of  $t$  in the range  $[0, 2\pi]$ , we can use equations (3.30) and (3.31) to generate points on the curve represented by equation (3.27) projected onto the image plane. These points form a boundary around an unclipped region of the image. Two unclipped regions can be intersected by tessellating them into non-convex polygons and performing polygon intersection tests between each pair of polygons between two unclipped regions (here each unclipped region corresponds to a single projected circle).

### 3.4.2 Iterative Solution

#### Visibility Culling

As described in the previous section, a simple and efficient algorithm for determining visible tiles in 3D can be achieved by representing the image as a scene graph. For the sphere viewer, the same overall approach can be used; however, extra steps must

be taken at each tile to determine whether or not is it visible. The main difficulty in spherical visibility culling is that tiles are no longer confined to a plane. Instead, they are projected onto a sphere and can be extremely warped at the coarsest resolutions. Therefore we must treat the tiles as more general curved surfaces rather than planar polygons.

To generalize the previous visibility culling strategy described in section 3.3 to general non-linear 3D surfaces, we can use bounding spheres to simplify the view frustum intersection tests. Bounding spheres are particularly attractive in comparison to bounding boxes (BB), axis-aligned bounding boxes (AABB), and binary space partitioning (BSP) trees due to their simplicity and efficiency. All that is necessary is to find the center of the object and its radius and the sphere is defined. The center vertex of an object can be found by simply by finding the minimum and maximum positions of points separately for each of the three spatial dimensions. The average of these two vertices is the center. The radius can be found by computing the maximum distance from the center to each point on the object. The center and radius can be precomputed for each tile as the spherical geometry does not change throughout the visualization process.

Itersecting a bounding sphere with the viewing frustum is simply a matter of determining if the center of the sphere is close enough to the frustum volume, i.e. within a threshold distance equal to the radius of the sphere. For each plane of the frustum, the center is translated along the plane's normal vector by a distance equal to the radius of the sphere (assuming the frustum plane normals are pointing inward). The same vertex/frustum intersection test described in section 3.3 can then be performed.

When performing this test in CVV space rather than  $\mathbb{R}^3$ , the sphere undergoes some non-linear warping due to the transformed space. On the surface, this may

seem negligible since the bounding sphere intersection method is an approximation anyway. However, the errors from the intersection test in  $\mathbb{R}^3$  are guaranteed to be false positives, meaning that more tiles will be found to be visible than the number that actually are visible. This will have no effect on the visible result, although some inefficiency is inherent. The results of the warping in CVV space do not guarantee the errors to be false positives. The false negatives which can occur in this space will result in unfilled pixels in the viewing plane. It is recommended that the test be performed in  $\mathbb{R}^3$  as correctness of output is generally valued more than efficiency of computation.

One major flaw with this method presented so far is that it allows both the near and far surfaces of the sphere to pass the intersection test if the entire sphere lies between the near and far clipping planes. One simple solution for removing the occluded portion of the sphere surface is to move the far clipping plane closer to the viewpoint. This solution is valid for the simple case when the vector

$$\mathbf{v} = \mathbf{s} - \mathbf{c}_{eye} \tag{3.32}$$

is parallel to  $\mathbf{n}$  as defined in equation (3.3). However, the problem of exactly how close to move the far clipping plane requires a little more thought.

By observing Figure 3.6, we see that regardless of the orientation of the camera, the set of all rays  $\mathbf{u}_i \in U$  emanating from  $\mathbf{c}_{eye}$  tangent to the sphere form a cone. The intersection of this cone with the sphere is a circle in 3D space. The plane containing this circle separates the front-facing potentially-visible portion of the sphere from the back-facing occluded portion.

It should now be clear that the simple case where the vectors  $\mathbf{v}$  and  $\mathbf{n}$  are parallel is very similar in theory to the more general case where the two are not necessarily related at all. However, it is still useful to distinguish them for implementation pur-

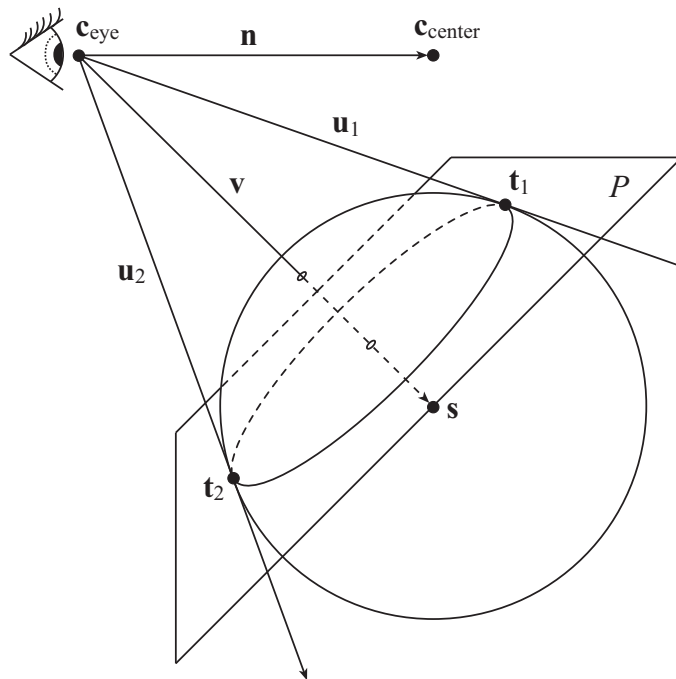


Figure 3.6: Clipping plane  $P$  divides the sphere into visible and invisible regions. Any point  $\mathbf{p}_i$  on  $P$  combined with  $\mathbf{v}$  define the equation  $\frac{\mathbf{v}}{\|\mathbf{v}\|} \cdot (\mathbf{x} - \mathbf{p}_i) = 0$  for plane  $P$ .

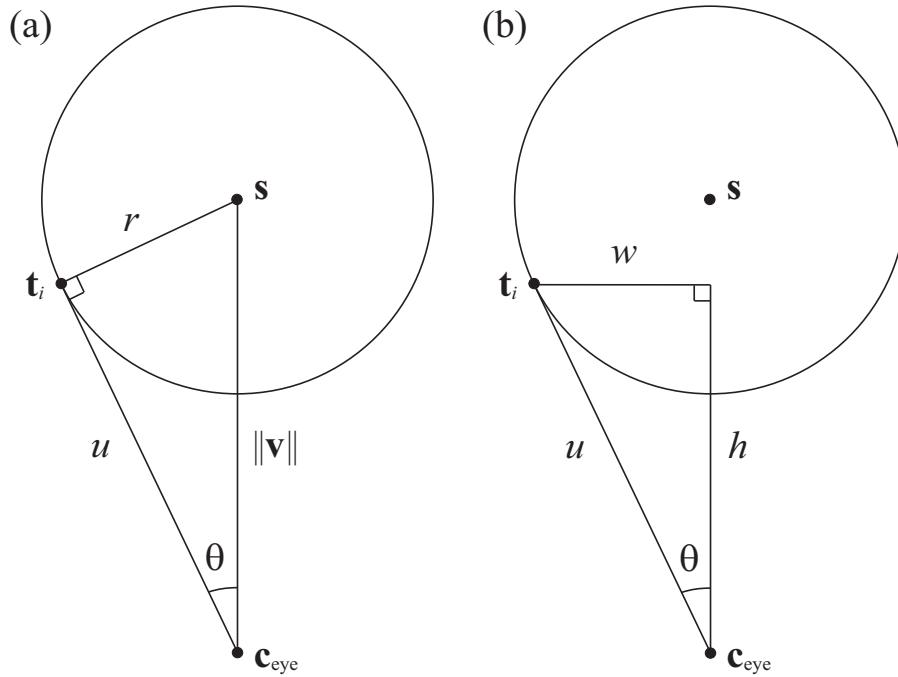


Figure 3.7: 2D slice of a sphere through a plane containing  $\mathbf{s}$  and  $\mathbf{c}_{eye}$ .

poses since clipping planes orthogonal to  $\mathbf{n}$  can be included in the projection matrix  $\mathbf{P}$  whereas arbitrarily-oriented planes cannot, thereby requiring an extra intersection test to determine the visibility of an object. Keeping this in mind, we will now focus our attention to the case of arbitrary plane and camera orientations.

We can see in Figure 3.6 that  $P$  must be orthogonal to  $\mathbf{v}$ , so our planar equation will be of the form

$$\frac{\mathbf{v}}{\|\mathbf{v}\|} \cdot (\mathbf{x} - \mathbf{p}) = 0 \quad (3.33)$$

where  $\mathbf{p}$  is any point on  $P$ . We can find a suitable value for  $\mathbf{p}$  by finding  $h$ , the shortest distance from  $\mathbf{c}_{eye}$  to  $P$ , and computing

$$\mathbf{p} = \mathbf{c}_{eye} + h \frac{\mathbf{v}}{\|\mathbf{v}\|}. \quad (3.34)$$

The value of  $h$  can be found using trigonometry, as shown in Figure 3.7. We first

compute the distance  $u$  using

$$u = \sqrt{\|\mathbf{v}\|^2 - r^2}. \quad (3.35)$$

Using Figure 3.7 (a), we can see that

$$\cos \theta = \frac{u}{\|\mathbf{v}\|}. \quad (3.36)$$

In Figure 3.7 (b), we have a similar relationship

$$\cos \theta = \frac{h}{u}. \quad (3.37)$$

By combining equations (3.36) and (3.37), we get

$$h = \frac{u^2}{\|\mathbf{v}\|}. \quad (3.38)$$

By substituting equations (3.38) and (3.34) into (3.33), we get the equation for the plane that divides the visible regions from the occluded regions of the sphere. During the top-down scene graph traversal of the quadtree, each tile is first checked for an intersection with the viewing frustum. An intersection test with the plane defined by equation (3.33) must also be performed. In the general case where the camera may be arbitrarily oriented, these two test must be performed separately since the intersection of the space in front of this plane with the interior of the viewing frustum is not guaranteed to produce a hexahedral volume.

### Multiple Resolution Tiles

Just as the visibility culling techniques had to be modified for curved tiles, the method for determining the appropriate resolution of a tile must be augmented likewise. The method presented in the iterative solution part of section 3.3 works well for planar polygonal tiles, but fails for curved surfaces. This is because curved tiles are not

guaranteed to have a one-to-one and onto mapping between window coordinates and object coordinates. Curved surfaces can overlap when projected onto the viewing plane, so some portions of the projected shape represent more pixel data than is represented by a polygon area calculation.

Instead of calculating the area of a tile to determine its resolution, we can compute the resolution at sample points and pick a resolution for the entire tile based on the sample calculations. Recall equation (3.14) from the analytic solution part of section 3.3. By calculating the value of  $r$  at various sample points, we can choose the closest, furthest or average value to determine the whole tile resolution. The tile can also be divided at the points where the resolution changes and rendered at two different levels of detail. This can be accomplished by intersecting the planar divisions of the view frustum whose distance from the eye is given by equation (3.15) with the tile plane. The line of intersection will split the tile into two halves, each containing (mostly) sample points of the correct resolution. This calculation is approximate since we have assumed that planar frustum divisions are used. If spherical divisions are used, the result will be more exact.

## 3.5 Arbitrary Geometry Viewer

While the arbitrary geometry viewer has not yet been implemented in Kolam, it is defined here as an extension of techniques used for the sphere viewer. It relies more on well-known computer graphics methods of visibility culling and hidden surface removal, as these techniques are well suited for arbitrary geometric meshes. The interesting feature here is the application of these techniques to extremely large multiple resolution datasets.

One difference between the sphere rendering algorithm and this one is that the

geometry itself is a dataset, and cannot be analytically computed at runtime as the sphere geometry is. Therefore, we must apply the same multi-resolution tiled pyramid format to the geometry as well as the imagery to avoid filling up memory with more geometry than is necessary. One of the main difficulties here is partitioning the geometry into tiles at multiple resolutions. There are several well known techniques for creating low resolution geometry tiles through model simplification. The basic idea behind these is to remove vertices and edges that connect faces whose normal vectors have the most similar orientations.

### 3.5.1 Visibility Culling

Visibility culling of tiled pyramid geometry is highly influenced by the method of tiling used. The ideal tiling method would partition the geometry into tiles of arbitrary spatial size whose convex hulls overlap as little as possible, but contain roughly the same number of geometric primitives. Using binary space partitioning (BSP) trees is a simple well known method for accomplishing this, and produces axis-aligned bounding boxes of data for easy intersection testing. Once the tiled pyramid of geometry data is assembled offline, it can be loaded and traversed at runtime using the scene graph rendering algorithms presented in earlier sections.

One important thing to note about visibility culling is that it may require two stages. Remember that in the previous viewer descriptions, each geometry tile was bound to an image tile, so culling non-visible geometry tiles was essentially culling non-visible image tiles also. In this case, we may not necessarily want to have geometry tiles match imagery tiles. While this may make it easier to implement an algorithm, it may require users to create many redundant tiled pyramid datasets for geometry: one for each image to be overlaid that uses a different translation, scale,



or tile size. This approach would also require an image's geometric tiled pyramid to be completely rebuilt each time the image location changes.

It would be more ideal for the imagery to be pasted onto arbitrary geometry tiles, much like the traditional texture mapping approach in computer graphics. To accomplish this, we may need to implement a two stage visibility culling process. We first cull the geometry tiles that are not visible, and determine the correct resolution of those that remain. Then, for each dataset we wish to paste onto the geometry, we partition each geometry tile into subsections, each mapping to a single tile of imagery. Each subsection goes through the same intersection test as the entire geometry tile to cull those image tiles that are not visible. The partitioning of geometry tiles into subsections should be done ideally once at startup, and again each time an image's location is changed during runtime. While this may be an expensive operation, it is far less expensive than rebuilding the image's entire geometric tiled pyramid.

### **3.5.2 Hidden Surface Removal**

Hidden surface removal is another challenge here. In traditional computer graphics, hidden surface removal for opaque primitives can be done entirely on the GPU using the depth test and depth buffer. Unfortunately, we cannot use this approach since we desire to remove image tiles before they are sent down the graphics pipeline. One possible approach is to use a technique called identifier image mapping. This technique requires two rendering passes. The first pass renders each image tile's geometric primitives with a unique color, which becomes that tile's identifier. If the depth test is enabled, only those tiles that are not occluded will be represented. The framebuffer is then read back into memory, and each pixel is traversed. The pixels' unique identifiers are dereferenced to create a map of non-occluded tiles. The second

rendering pass loads and renders only those tiles that are in this map. The advantage of this approach is that the graphics card still does all the work of hidden surface removal. The disadvantage is that it requires two rendering passes, a framebuffer read, and a traversal of each screen pixel. This is likely to decrease performance dramatically on PCs with anything but the top of the line CPUs and GPUs.

A far better approach for hidden surface removal may be achieved using OpenGL occlusion queries. These operations perform essentially the same task as the identifier image mapping technique, but eliminate the need to read the framebuffer back into memory and traverse each pixel. Queries can be performed on single primitives or arbitrary groups of primitives, so we can still perform queries on entire tiles. They also provide a count of how many pixels are represented by a particular tile, so tiles that are spatially near the viewer but comprise only a few pixels on the screen can be loaded at a very coarse resolution. The two pass rendering can be avoided at the cost of some momentary inaccuracies by always performing occlusion queries while the previous frame's state is still active on the GPU.

### **3.6 Volume Viewer**

The volume viewer employs similar techniques to those of the oblique viewer, but extends the imagery data to three dimensions. Instead of a quadtree to store our image data, we now use an octree. This means that each tile will now have eight children instead of four. Tiles will generally be smaller spatially, since these datasets will be more dense than their 2D counterparts.

The current volume viewer implementation is a separate codebase, and is not part of the main Kolam application. It has been tested with two of the Visible Human datasets [5], and screenshot of the application displaying one of these datasets is

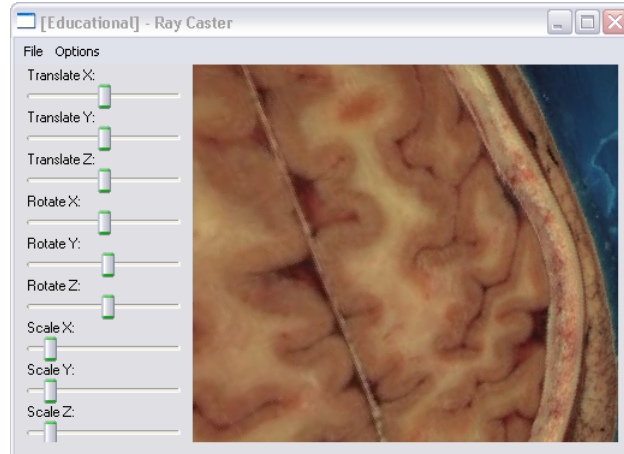


Figure 3.8: This screenshot of the volume viewer version of Kolam shows a tiled segment of one of the Visible Human datasets [5].

shown in figure 3.8. The datasets is stored in a modified image pyramid format, and each tile contains  $64^3$  voxels.

### 3.6.1 Visibility Culling

Since we defined a 3D view frustum culling test for the oblique viewer using planar tiles, this technique can be adapted directly to work for volumetric tiles also. The only difference is in the complexity of the tile intersection test. Part of the test can be done most efficiently in  $\mathbb{R}^3$ , while the other can be done most efficiently in canonical view volume (CVV) space. The test in  $\mathbb{R}^3$  will compute the intersection of the view frustum's vertices and edges with the tile's bounding box. The test in CVV space will compute the intersection of the tile bounding box's vertices and edges with the CVV. At eight test per vertex, with 12 total vertices to test, and 12 tests per edge with 12 edges to test, this comes out to a maximum of 240 tests per tile to determine if it lies withing the viewing frustum.

A less expensive test would involve using bounding spheres for each tile. Each bounding sphere is defined to have the same center point as the tile's bounding box, and a radius equal to the distance between the bounding box's center and each of its eight corners. The sphere's center is tested against each plane of the CVV, minus the sphere's radius, which requires only 6 tests per tile. While this test is less accurate, it can improve efficiency when used as the first pass of a two-stage intersection test. While the sphere test can yield false positives for up to six neighboring tiles, it can still eliminate all the others. Only if one of the seven possible tiles (the center tile plus its six neighbors) passes is the more expensive bounding box test executed.

### 3.6.2 Multiple Resolution Tiles

In oblique viewer, we determined if the current resolution of a tile was fine enough by projecting the four corners onto the viewing plane to form a quadrilateral in screen space, measuring the area of the quadrilateral in pixels, and analyzing the ratio of screen pixels to tile texels.

We can use a similar technique to determine the correct resolution of 3D tiles. The trick is in projecting the 3D tile onto the viewing plane in a way that represents how its texels will be mapped to pixels. If the tile intersects the viewing plane, we simply intersect the viewing plane with the tile's edges to get the four points of our quadrilateral. If the tile lies in front of the viewing plane, we can find which of its six faces is most directly facing the eye point by taking the dot product of each plane's normal vector and the viewing vector. The result closest to the value of  $-1$  wins, since this vector pair is most parallel, but facing opposing directions. We then project this tile's face onto the viewing plane in the same way we would project a 2D tile.

## 3.7 Rendering Tiled Image Data

Once the set of visible tiles has been determined by a viewer, the tiles must be passed to a renderer. A renderer in Kolam is a method for projecting a set of tiles from a given viewpoint onto the viewing plane. There are numerous ways to do this, and different methods will yield better performance based on the organization and format of the data, the graphics hardware present, and the complexity of the tile projection.

### 3.7.1 Raster Renderer — The Bare Essentials

The simplest renderer available in Kolam is the raster renderer. This method renders scaled and translated data from main memory directly to the framebuffer. No rotation or interpolation between color samples is performed. It is of little use except when viewing 2D data in an orthogonal projection.

The simplicity of this renderer can provide advantages in certain situations. On systems with little or no hardware accelerated 3D rendering, this method is likely to be considerably faster than any of the more complex methods. Also, since there is no cached data stored on the graphics hardware, any changes made to the data in main memory will automatically be updated on the display on the next rendering pass without having to update an additional cache.

### 3.7.2 Texture Renderer — Storing Data on the Graphics Hardware

The main difference between the texture and raster renderers is that the texture renderer stores visible data in an additional cache located on the graphics hardware. Once it is cached, the data can be rendered directly by the graphics hardware rather

than being reloaded from main memory for each frame. The graphics hardware can also perform transformations and pixel interpolation on the cached data prior to rendering. This added flexibility allows textured data to be rendered in perspective projections with blending between color samples rather than sharp color transitions at texel borders.

One subtle drawback of the texture renderer that does not exist with the raster renderer occurs when trying to composite several color channels into a single image on-the-fly. The raster renderer can use the OpenGL API to render images stored as grayscale or luminance values to any of the red, green, blue or alpha channels using a color mask. However, a texture stored as luminance values cannot be rendered to any channel other than red because the luminance values are always stored in the red channel. The only way to composite channels on-the-fly using textures is to combine the channels before loading them into texture memory, which significantly increases the display latency when adding or removing channels from the composite image.

### **Texture Borders — Interpolating Boundary Samples**

The graphics hardware's ability to perform fast hardware interpolation on enlarged areas between color samples leads to a somewhat more noticeable problem than combining color channels: how to perform interpolation between neighboring color samples from different tiles? Although the samples may appear to be adjacent on the display, they appear completely unrelated to the graphics hardware.

OpenGL provides two solutions to this problem. The first is a special flag applied to each texture in memory which tells the hardware to blend each edge pixel with itself. In other words, interpolation is performed only in the interior of a texture, but not at the edges. This is barely noticeable when viewing an image at approximately equal to or less than its original resolution. As the image is enlarged, the lack of

interpolation between two adjacent textures becomes visible as a sharp difference between color values in an otherwise smooth image.

The second solution provided by OpenGL allocated additional memory on the graphics hardware for each texture to store border sample information. This technique of using texture borders eliminates the artifacts caused by the lack of interpolation between tiles. However, it is not widely supported on all graphics hardware. If used on the wrong hardware, this technique can decrease performance dramatically by dropping frame rates to sub-interactive levels.

A third solution is to simulate texture borders without using the OpenGL solution. Increasing the texture size is not feasible since texture sizes must be powers of two. We can, however, shrink our data tiles to make room for a border in a fixed size texture. For an image with tiles of width and height  $2^n$  for some  $n > 1$ , this can be done by storing tiles of size  $2^n - 2$  in main memory. These smaller tiles are loaded into texture memory at the index  $(1, 1)$  rather than the usual  $(0, 0)$ . The one pixel radius around the actual data is filled with neighboring pixels from surrounding tiles. The borders of edge tiles should be loaded with duplicated data from the interior region. The resulting texture is rendered using texture coordinates of  $(1/2^n, (2^n - 1)/2^n)$ . The border information may be loaded on-the-fly from neighboring tiles or stored as redundant data surrounding each tile. Either way, this method eliminates the likely performance hit caused by using OpenGL texture borders.

### **3.7.3 Terrain**

The terrain renderer is an extension of the texture renderer that generates additional geometry to represent height values associated with the textured image. Height values are typically defined by an 8 or 16 bit signed integer value corresponding to each image

pixel. The height values can therefore be represented as a single component image and stored in memory using a quadtree in much the same way as any other image. The generated geometry is often simplified to maintain acceptable frame rates, as generating two additional triangles for each texel would be prohibitive to interactive display. The method of simplification can be classified as either subsampling or adaptive methods.

The current implementation of Kolam uses the subsampling method to display terrain data. This method is considerably simpler than the adaptive method, though it is less optimal in the sense that it does not select triangles based on their contribution to the projected image. This method instead chooses a uniform subsampling of the array of height samples and uses this data as the basis for geometry generation. The data between samples is effectively eliminated from the rendering pipeline, thereby lowering the resolution of the terrain.

The adaptive terrain simplification method is more complex, but can generate much more pleasing visual results. The same number of triangles as is used with the subsampling method can be used more efficiently by selecting those that contribute most to the visual accuracy of the rendered image. A fairly simple and elegant framework for accomplishing this is described by Lindstrom and Pascucci in [21].

The adaptive framework begins with the initial geometry of a tile consisting of two right triangles who share a hypotenuse. Additional triangles are added recursively by subdividing each right triangle via an additional edge from the apex to the midpoint of the hypotenuse. The result is a directed acyclic graph (DAG) with edges  $(i, j)$  and  $j \in C_i$  where  $C_i$  represents all children of  $i$ . The recursion terminates when some error threshold has been exceeded.

The most simple and common error metric used for simplified terrain is to measure the object space distance  $\epsilon_i$  between the actual height value at vertex  $i$  and its



triangular mesh approximation. However, for true terrain adaptivity, one would also expect the mesh to change with the viewpoint; as a point  $\mathbf{p}_i$  becomes closer to the camera eye  $\mathbf{e}$ , its surrounding area's triangle count should increase based on a view-dependent error metric. Lindstrom and Pascucci present a framework for combining these object space and view-dependent error metrics in a simple and elegant way.

They first generate an object space error measurement for each height value. This metric is precalculated and saved with the height values. At each frame, the view-dependent error metric is calculated and combined with the object space metric to form a generic monotonic screen space error  $\rho_i = \rho(\epsilon_i, \mathbf{p}_i, \mathbf{e})$ . To guarantee that the addition of more triangles will not increase the screen space error, the condition

$$\rho(\epsilon_i, \mathbf{p}_i, \mathbf{e}) \geq \rho(\epsilon_j, \mathbf{p}_j, \mathbf{e}) \quad \forall j \in C_i \quad (3.39)$$

is imposed. Since  $\rho$  is monotonic, defining

$$\epsilon_i = \max \{ \hat{\epsilon}_i, \max_{j \in C_i} \{ \epsilon_j \} \} \quad (3.40)$$

where  $\hat{\epsilon}_i$  is the actual error metric will satisfy  $\rho(\epsilon_i, \mathbf{p}_i, \mathbf{e}) \geq \rho(\hat{\epsilon}_i, \mathbf{p}_i, \mathbf{e})$ , but not necessarily (3.39). To satisfy (3.39),  $\rho_i$  can be rewritten as

$$\rho_i = \max_{\mathbf{x} \in B_i} \rho(\epsilon_i, \mathbf{x}, \mathbf{e}) \quad (3.41)$$

where  $B_i$  is a ball centered at  $\mathbf{p}_i$  containing all points within radius  $r_i$ . By defining the radius as

$$r_i = \begin{cases} 0 & \text{if } C_i \text{ is empty} \\ \max_{j \in C_i} \{ \|\mathbf{p}_i - \mathbf{p}_j\| + r_j \} & \text{otherwise} \end{cases}, \quad (3.42)$$

we are guaranteed that  $B_i \supseteq B_j$ . Since we are also guaranteed that  $\epsilon_i \geq \epsilon_j$  from (3.40), (3.39) is now satisfied.

# CHAPTER 4

## Kolam User Interface

### 4.1 Windows

At the application level, Kolam's interface consists of one main visualization window along with several other supplementary windows. Supplementary windows can be implemented as *plugins* so that users can develop and distribute their own visualization tools without modifying the application source code.

The main visualization window allows the most freedom for interactively viewing the image data (see Figure 4.1). Any number of *viewers* (see Section 3.1) may be attached to this windows to allow different views of the data. Additionally, a viewer may use one or more *renderers* (see Section 3.7) to display each tile using a specific rendering technique. A viewer may also employ one or more *navigators* (see Section 4.3) to define how roam, zoom and tilt operations are performed.

The overview window displays only the coarsest resolution of the current active

layer. Its viewpoint is shared by the main display, so any navigation actions taken on this view will affect the main display and vice-versa. This is useful for quickly jumping from one region to another. This window also displays all tiles currently being rendered in the main window as sub-rectangles of the coarsest resolution. Although this was originally designed as a debugging tool, it has also proved useful for quickly identifying one's viewpoint position in the main window (see Figure 4.1).

The cache glyph was also intended as a debugging tool, but it has found other uses as a tool for measuring the performance of disk I/O and cache management techniques. A user running Kolam on a specific platform may view cache updates in real time and change the application settings to maximize performance on that platform (see Figure 4.1). Options that may affect performance include caching/paging algorithms (LRU/priority queues, distance calculations, modular arithmetic regions), the number of reader threads, tile compression method, and per-tile image processing algorithms.

In addition to these visualization windows, Kolam also provides several GUI windows for manipulating data parameters. The GUI windows organize several control panes into a tabbed interface based on the class of data they can manipulate, such as layers or colormaps (see Section 4.2). Users may implement their own control panel tabs using Kolam's plugin architecture (see Section B.2).

The layer editor displayed in Figure 4.2 provides controls for manipulating individual image layers (see Section 4.2.1). This editor currently provides tools for naming layers, viewing file information, changing rendering parameters and applying histogram enhancement, colormaps and height maps. Although each layer is currently bound to a single dataset, future implementations will provide container layers to encapsulate one or more child layers. Using this mechanism, all child layers can undergo the same manipulations applied to the parent layer without having to perform the

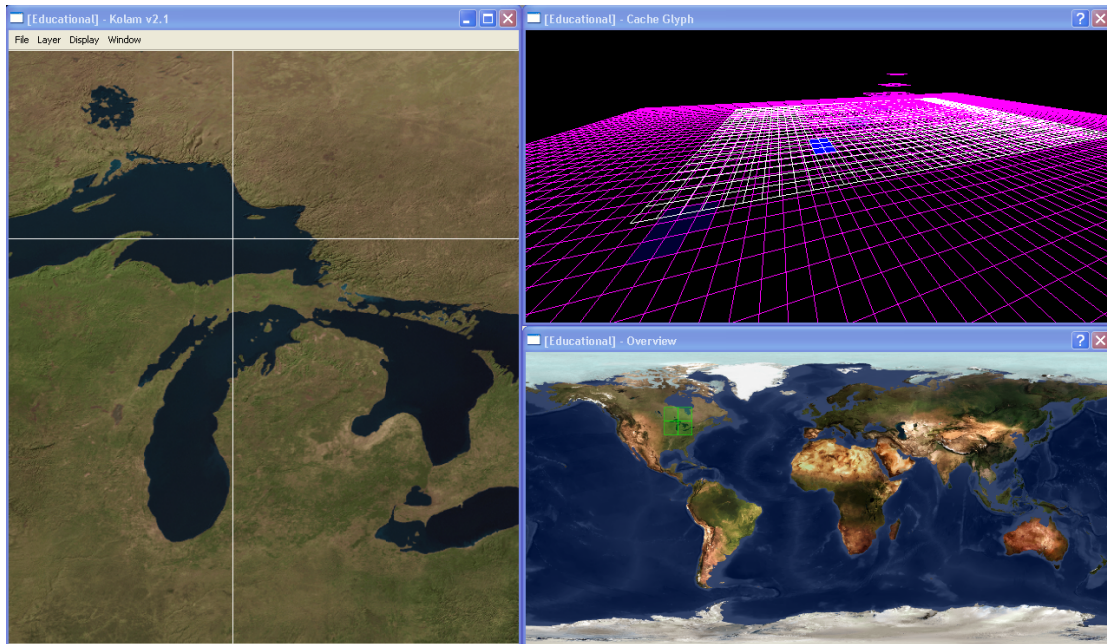


Figure 4.1: The main window (left) provides a detailed view of specific regions of the image. The overview window (bottom right) allows the user to quickly identify his position in the main window and cache glyph (top right) shows which tiles are resident in memory.

same operation multiple times.

The colormap editor displayed in Figure 4.2 provides tools for not only manipulating colormaps (see Section 4.2.2), but creating them also. Currently only simple linear gradient maps can be created, but arbitrary maps can be read from disk. The colormaps on disk are specified by a simple text file containing an RGB or RGBA color lookup value on each line.

## 4.2 Scene Components

### 4.2.1 Layers

A layer is defined as a visual representation of a dataset in Kolam. An arbitrary number of layers can be combined and displayed simultaneously at runtime to produce a composite dataset from multiple sources. The ability to combine layers can add many dimensions of information to even a small area. For example, by combining layers representing geospatial imagery, elevation data, cloud coverage and city lights, one can obtain a much more complete picture of the area's environment than by viewing any of these layers individually.

Layers are higher level representations of datasets such as images or heightmaps. They encapsulate both the dataset and any relevant metadata. This metadata may include such properties as whether the dataset is visible or not visible, whether certain colors should be displayed as transparent, and the location and size of the dataset relative to some global coordinate system.

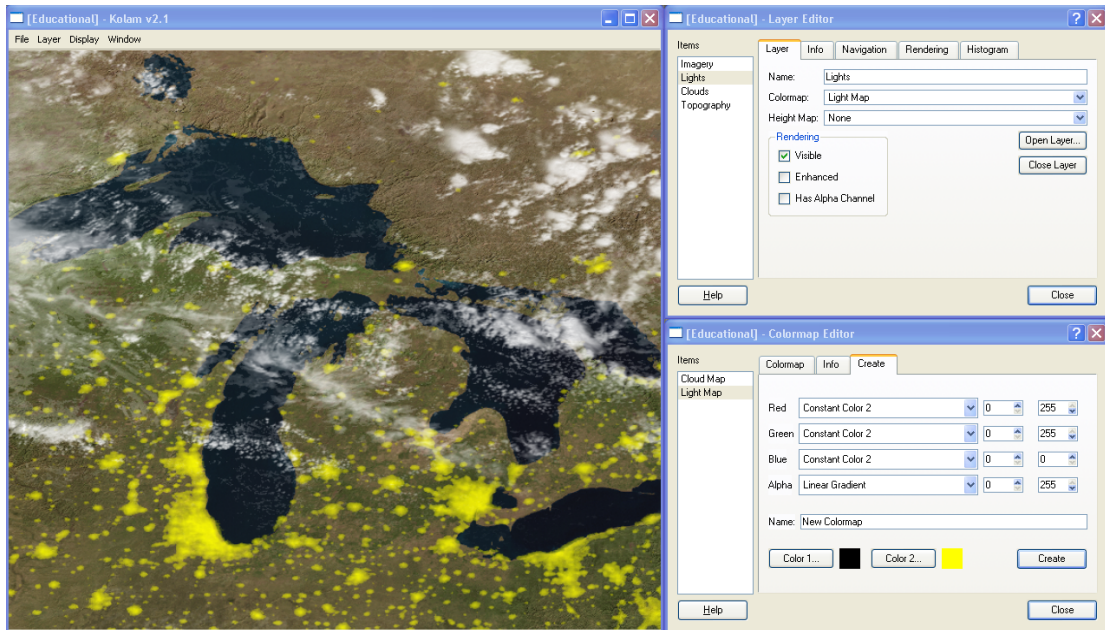


Figure 4.2: The main window (left) can show many datasets simultaneously. The layer editor (top right) allows each dataset to be configured individually and the colormap editor (bottom right) manipulates and creates colormaps that can be attached to datasets.

## 4.2.2 Colormaps

A colormap is a built-in image filter. It is used to transform the color space of an image in an arbitrary way. This is typically used for greyscale or luminance images, where each pixel is a value from a one-dimensional linear distribution. This value is used as a the key to an entry in the colormap.

One example of the usefulness of colormaps is evident in global lighting datasets. Each pixel of these datasets is a measurement of how much light is emitted by the pixel's area of the globe at a particular instant in time. If these datasets were displayed as raw images, they would show a single opaque layer of grey values, closer to white for lights areas, and closer to black for dark areas. Colormaps can be used to add both color and transparency to these datasets. By mapping each value to a constant shade of yellow, and mapping the luminance of each pixel to its opacity, one can obtain a visual result not unlike one our eyes would expect to see when perceiving an aerial photograph of the same phenomenon (see Section A.3.2).

Another example of the usefulness of colormaps can be seen in weather data. Datasets showing weather severity may use dark pixel luminance values to represent calm weather and light values to represent severe weather. Although we could use the same trick as with the global lights datasets to map the entire range to a single color, in this case it may be more useful to map ranges of values to different colors to discretize the data into something like blue for calm, green for moderately calm, yellow for moderate, orange for moderately severe, and red for severe.

## 4.3 Navigators for Interpreting User Input

Navigators in Kolam are modules that define how stimulation from various input devices affect the state of the current viewer module. The most common stimulation

and input device combination is in the form of click and drag operations from a mouse. Other potential sources of stimulation may come from joysticks, 3D trackballs, virtual reality gloves, or force feedback devices. The only input device currently supported by Kolam is the mouse, but the framework is general enough to allow for fairly painless integration of new devices and behaviours.

### **4.3.1 Roam, Zoom and Tilt Navigators**

This roam, zoom and tilt navigators define the default mode of operation in Kolam. Users perform mouse gestures to translate, rotate, and scale their current view. Each of the three standard mouse buttons maps to one of the roam, tilt, or zoom operations, and dragging while holding one of these buttons results in corresponding scene transformations.

The roam operation tries to emulate a translation in image space. For the orthogonal view, this means a translation in screen space. For the oblique view, the translation is confined to the image plane so that the eye's distance from the image plane remains constant while roaming. For the sphere view, the sphere rotates around its center to bring a new area closer to the eye. For the arbitrary geometry view and volume view, the roam operation may be implemented as a translation within the viewing plane or a rotation around some fixed point.

The zoom operation typically results in a translation orthogonal to the viewing plane or a scaling of the scene. For the orthogonal view, a simple scene scaling is sufficient. For each of the other 3D views, a translation orthogonal to the viewing plane is generally more appropriate, since the user may in some cases expect to pass through objects as they become closer rather than watch them grow infinitely larger.

The tilt operation is not present in the orthogonal view, since this would eliminate



this view's primary advantage of simplicity, and would make it unsuitable for using the raster renderer described earlier. For the oblique view, the tilt operation rotates the image plane around the current focal point, which is the intersection of the viewing vector with the image plane. For the sphere view, the tilt operation is actually a translation in screen space. This is an indirect tilt which allows the viewing vector intersect the sphere closer to the horizon, where the image will appear more oblique or tilted. For the arbitrary geometry view and volume view, this operation may have a similar effect.

### **Vector vs. Velocity Navigators**

Each gesture in the vector navigator transforms the view from one constant state to another based on the vector created by the difference between the gesture's ending and starting points in screen space. The velocity navigator instead uses each gesture to define a velocity vector which is continuously applied to the view at each frame update until the gesture is completed. The velocity vector is defined by the difference between the gesture's current point and starting point in screen coordinates. The vector navigator is the default mode of operation in Kolam. The velocity navigator is also referred to as fly mode.

### **4.3.2 Image Processing Navigators**

Navigators can also be used to perform traditional image processing operations such as image positioning, resizing, and cropping. The orthogonal view is best suited for this class of operations, since it can map rectangles in screen space directly to rectangles in image space. These operations are typically used to position and size embedded datasets when their real world position and size is not known. Mouse

gestures are usually translated into a 2D vector for positioning, a vector distance or scale factor for resizing, or a rectangle for cropping.

### **4.3.3 Modeling Navigators**

Navigators can be used to perform traditional modeling tool operations also. Terrain geometry can be modified in a variety of ways by different mouse gestures. The height of a single vertex can be changed, or an entire area can be selected for processing. The concept of spring-mass systems can be applied to neighboring vertices to pull them closer to a single modified vertex, thereby smoothing the geometry.

# CHAPTER 5

## Performance Optimizations and Characterization

### 5.1 Performance Optimizations

This section is a description of some simple tricks employed to mask some of the inherent problems with the algorithms provided in earlier sections of this document. Some of these problems are merely visually unpleasant, while others can have adverse effects on performance. Despite their simplicity, each of these techniques is significantly beneficial.

#### 5.1.1 Hiding Disk Latency

One inherent problem of an out-of-core visualization system is that the data to be rendered is not always available when it is needed. Once the view frustum culling

and hidden surface removal operations have completed, leaving a small set of visible tiles to be rendered, it is likely that some of those tiles will not be fetched from disk until several frames later. The question is what to display in place of the missing tiles until they can be rendered.

The most basic solution to this problem is to display nothing. If the tile is not available, don't render it. This was the solution used in the original version of Kolam, and the visual results made the tile loading latency painfully obvious to anyone using the software. When roaming, the effect was somewhat tolerable. The user can't see any tiles that are offscreen, so when they continue to be invisible for a moment after they appear onscreen, this quickly becomes a logical and expected phenomenon for the user. When zooming in the orthogonal view, however, the result can be a complete upset to the user. Since the resolution of all visible tiles changes at once, if none of these tiles are present in the memory cache, the user will see nothing but a blank screen when the resolution changes. Tiles will then load and become visible one by one until the screen is filled at the new resolution.

The better solution is to use a gradual level-of-detail approach to always display some form of the tile's data, no matter how coarse it may be. This is done by searching upward through the quadtree for each tile that is deemed visible, but is not present in the memory cache. The first tile found that is currently present in the cache will be displayed instead of the higher resolution tile until that tile is finally loaded. By always keeping the coarsest level of the pyramid in memory, there will always be a low resolution version of any tile available. The user will always see some form of the tile's image, even if it is excessively blurred by downsampling extremely coarse data. The texture coordinates of the low resolution tile must also be updated appropriately to select only the portion corresponding to the desired but unavailable tile. With the raster renderer, this type of on-the-fly indexing and scaling of image data can also be

done, but is more difficult and has yet to be implemented.

### 5.1.2 Maintaining Constant FPS

Another problem with early Kolam implementations was that the frame rate would increase or decrease noticeably with the number of visible tiles on the screen. This was the most problematic in the orthogonal viewer, since zooming in across a resolution boundary could cause the number visible tiles to quadruple from one frame to the next, thereby causing a sudden decrease in the frame rate.

A simple way around this issue is to artificially limit the number of visible tiles on the screen in an attempt to preserve a constant frame rate. For each frame, the resolution of tiles farthest from the edge can be artificially lowered until the total number of tiles is below the limit. When the user is zooming in, upon passing a resolution boundary, they will see the the center tiles increase in resolution first. As the outer tiles are zoomed off of the screen, the tiles surrounding the center will also increase in resolution.

Setting the limit on the number of visible tiles can be accomplished by calculating how many tiles will fit on the screen at the original resolution.

$$\lceil \frac{w_{screen} - 1}{w_{tile}} + 1 \rceil \lceil \frac{h_{screen} - 1}{h_{tile}} + 1 \rceil \tag{5.1}$$

This is the maximum number of tiles that will ever be displayed without imposing any limits. Divide this number by 4 to get the minimum number of tiles that will be displayed, except when at or above the coarsest resolution. The limit should be within this range, ideally closer to the minimum.

### 5.1.3 Prioritizing Requests

A third problem noticed early in Kolam development with the LRU cache was that tile request were never deleted until the tile's data was finally cached. This was a problem when navigating over large areas, since a trail of requests would follow the navigation path. If the user continues to navigate faster than the reader threads can process requests, the trail can become quite long and will contain request for tiles that are no longer visible. Therefore some method was needed to delete old requests before they are processed if they are no longer visible.

The deletion of expired requests is accomplished by maintaining list of visible tiles for current and previous frames. For each frame, the current list is compared with the previous frame's list. Any requests corresponding to tiles that were visible at the last frame but not at the current frame are either discarded or assigned a lower priority.

### 5.1.4 Prefetching

Prefetching is a useful technique for pre-loading tiles that are expected to be requested in the near future. While there are several basic approaches to the problem, none of them can always accurately predict where a user will navigate to next.

One prefetching method is to load tiles that are spatially close to the viewpoint but are not currently visible. This works well when the user is roaming in an arbitrary patterns but not zooming. Another approach is to load tiles that are close in resolution to the currently visible tiles. This works well when the user is zooming in arbitrary patterns. If the user's navigation pattern is more well defined, it may make more sense to characterize their motion vector at the current instant and use that to determine the most likely area that will be requested next.

## 5.2 Performance Characterization

Two tests were performed on Kolam to determine how certain variables affect performance. In each test, performance was measured in two ways. The number of frames displayed per second is a typical measure of performance in real-time graphics applications, so this was our first measurement. The number of dropped requests was chosen as the second performance measurement, since a dropped request implies that there were not enough resources available to fill the request until after it had expired (i.e. the tile was no longer visible at that resolution).

In the first test, the independent variable was the degree of multithreading, measured by the number of tile reader threads running simultaneously. It was expected that as the number of threads increases, the frame rate would decrease, since there would be fewer processor cycles available for the rendering thread. We also expected that the number of dropped requests would decrease, since there would be more threads available to process requests before they expire.

The second test used cache size as the independent variable. It was expected that this test would show that as the cache size increases, the number of dropped requests would decrease since there would not be as many requests made in the first place. It was expected that the frame rate would increase also since there would be less work for the reader threads, leaving more cycles for the rendering thread.

Each test was run with a fixed navigation script. The navigation procedure was heavily biased toward rapid zoom operations rather than roam operations, as these operations require more tile updates are likely to produce a clearer difference in performance metrics. The graphical results of these tests are displayed in figures 5.1 and 5.2.

Since there is a wide variance between results of the multithreading tests, it would

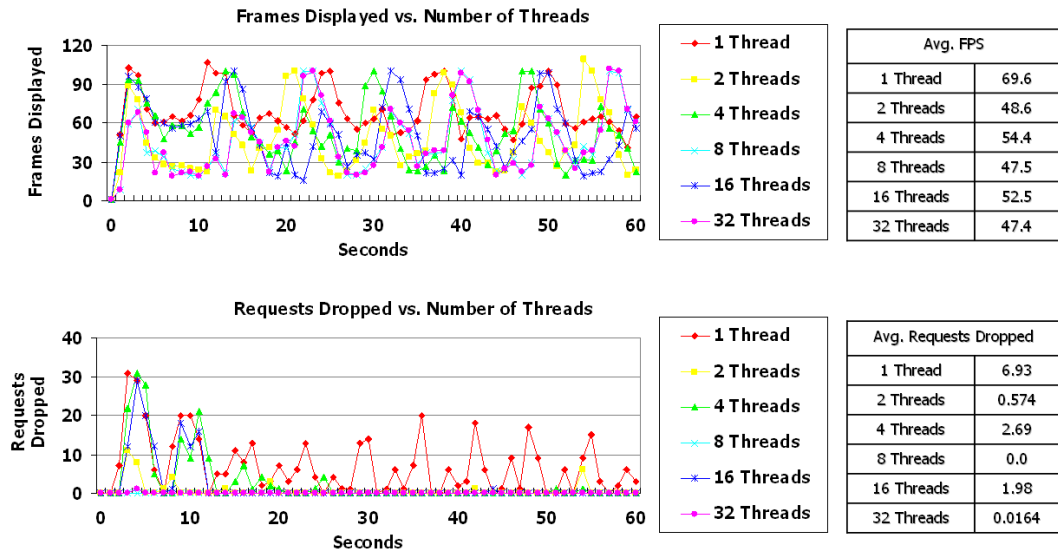


Figure 5.1: Multithreading performance test results

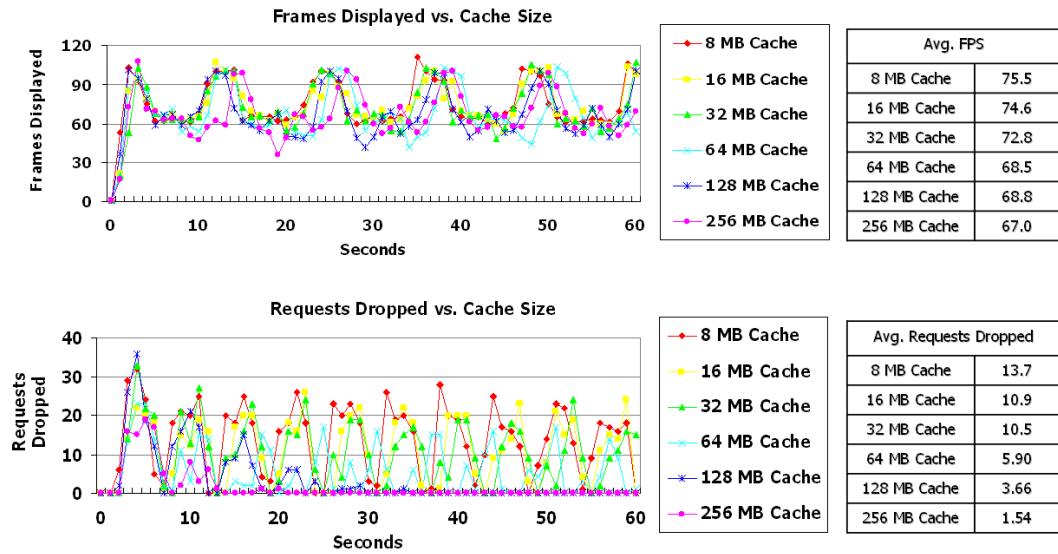


Figure 5.2: Cache size performance test results



appear that the number of reader threads does not have a well defined or easily measured effect on performance. As the number of threads increases, there is a slight trend toward a decreasing frame rate, as well as a more significant trend toward fewer dropped requests. The results support out hypothesized trend of decreasing dropped requests, but the experiment is inconclusive regarding the effect of the number of threads on frame rate.

The results of the cache size tests are much more robust than those of the multithreading tests. There is a clear indication that as the cache size increases, the number of dropped requests decreases, as does the frame rate. The trend of decreasing dropped requests matches our hypothesis. However, the fact that the frame rate decreased is contradictory to what was expected. This decrease can be explained by noting that as the cache size increases, the number of tiles that are available at the requested resolution is likely to increase. Thus it becomes more likely that the rendering thread displays more high resolution data, instead of displaying low resolution data while it waits for the high resolution data to be loaded.

# CHAPTER 6

## Conclusions and Future Work

### 6.1 Future Work

Of all the techniques outlined in Chapter 3, only a handful have been fully integrated into the Kolam software. Among these are the orthogonal and sphere viewers, the raster and texture renderers, and the roam, tilt and zoom navigators. While the combination of these abilities enables a working visualization platform, there are still many ways in which the software could potentially be improved.

In Section 3.5, several possible implementation strategies for an arbitrary geometry viewer were described. Adding this capability to Kolam would allow the visualization of not only massive images, but massive 3D models also. While a fairly large subset of practical 3D models can be displayed using heightmaps (which Kolam already supports in the sphere viewer), the heightmap representation is not general enough to visualize more complex scenes, such as anything indoors. Many potential Kolam

users, such as video game developers and real-time military simulation designers, would likely be interested in this capability.

In Section 3.6, an implementation of a volume viewer was described. Although this capability has actually been implemented using these techniques and a modified version of the Kolam core library, it was never integrated into the main Kolam application. It is also in need of considerable optimization before it can come close to matching the interactive performance of the Kolam 2D image viewing methods. Again, adding this capability to Kolam would increase the range of potential users to include medical professionals and military physical simulation designers, among others.

Even in the 2D image viewing modes already present in the Kolam software, there are still improvements to be made in data processing. Adding sparse vector overlays to layers to display metadata such as roads and political boundaries would provide new dimensions of information to users while improving overall efficiency (less data means less storage and faster I/O). Enabling transformations between widely used geospatial coordinate systems such as latitude/longitude, UTM and MGRS would simplify registration of embedded datasets. Providing the ability to edit image pixels and save tiles would open the door to the implementation of efficient tile-based image processing algorithms while using Kolam's out-of-core architecture to overcome the memory limitations of its competitors. Optimizing Kolam for loading tiles efficiently from remote sources would enable its use in a client/server architecture, with data stored in a shared repository that can be accessed by multiple Kolam visualization clients.

## 6.2 Conclusion

Kolam is a software architecture for visualizing arbitrarily large 2D and 3D dense datasets on commodity PC hardware. The techniques described in this document are used to extract visible areas of interest from these datasets and refine them to the appropriate level of detail in order to display them efficiently. The methods of extraction and refinement vary depending on the geometry used to map image data to 3D coordinates (planes, spheres, polygonal meshes, and 3D volumes) and the limitations imposed on the viewer's perception of the scene (orthogonal vs. oblique views). Layers are used to combine multiple datasets from different coordinate systems into the same view, colormaps are used to manipulate color values, and heightmaps are used to display elevation data. A tabbed dialog-based GUI provides a front end to all of these features, and is extendable through a basic plugin architecture.

The algorithms presented in this paper have been used to display a wide variety of datasets, especially those much larger than the amount of memory available on today's standard computer hardware. This fundamental capability will become increasingly important as the size of scientific datasets grows beyond anything we could imagine today.

# APPENDIX A

## User's Manual

### A.1 Introduction

The Kolam application allows users to load, view and navigate within multiple images simultaneously. The images that can be loaded are in a tiled pyramid format (see Section B.3). Once loaded, each image is encapsulated by a layer. This enables users to combine images as layers and manipulate high level properties with a simple user interface (see Section A.3.1). The application provides several views of the data (see Section A.2), each of which respond to mouse events to allow quick and easy navigation.

### A.2 Kolam Windows

Kolam currently shows three views of the currently loaded datasets (see Figure A.1).

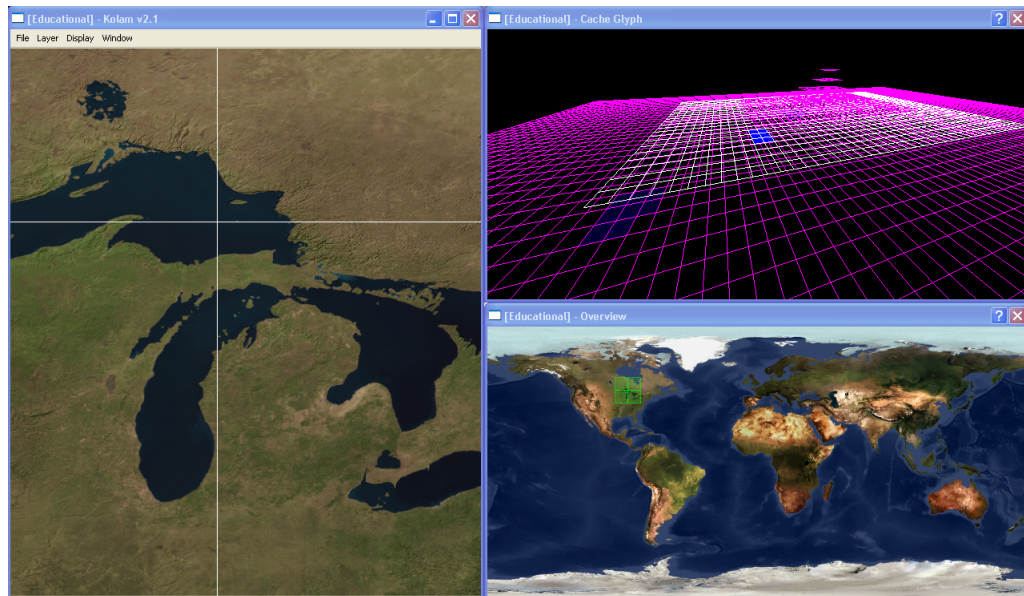


Figure A.1: The display window (left) provides a detailed view of specific regions of the image. The overview window (bottom right) allows the user to quickly identify his/her position in the display window. The cache view window (top right) shows which tiles are resident in memory.

### A.2.1 Display Window

The display window is the primary Kolam view (see Figure A.1). It can combine all layers into a single rendered image.

Navigation controls are dependent on the current viewing method (i.e. orthogonal, oblique, or spherical views). However, whenever possible, some attempt has been made to maintain consistency of operations between different viewers. For example, even though the left mouse button may translate the user in the orthogonal view and rotate the globe in spherical view, each of these amounts to a translation in image space. In both viewing modes, new portion of the image comes into view from the north, south, east or west, and the old portion of the image exits the visible region toward the opposite direction.

Users can navigate through the image using the following mouse controls:

**Left Button** Dragging with this button will change the user's position in the image (translate left/right/up/down).

**Middle Button** Dragging with this button will change the user's distance from the image (zoom in/out).

**Right Button** Dragging with this button will change the user's rotation in the image (tilt left/right/forward/backward).

### A.2.2 Overview Window

This window provides a top-level orthogonal view of the dataset as a whole (see Figure A.1). No matter where a user is in the main display, they can always use the overview window as a map to find their way around. Green squares will be displayed on the

overview around the region currently visible in the main display window. Note that only the active layer is visible in the overview window at any given time.

Users can click on the overview to change their positions in the main display:

**Left Button** Clicking with this button will move the user's position in the main display window to the location clicked on the overview.

### A.2.3 Cache View Window

This window displays the current layout of tiles in memory by showing a virtual 3D pyramid (see Figure A.1). This view is useful mainly for debugging and performance tuning. The average user will never need to use this display. Tiles are displayed in different colors to signify their current states. The colors are as follows:

**Transparent** These tiles are not present in the cache.

**Red** These tiles are not present in the cache, but have been requested to be loaded next.

**Yellow** These tiles are currently being loaded into the cache.

**Blue** These tiles are fully loaded and present in the cache.

Users can navigate through the cache display using the following mouse controls:

**Left Button** Dragging with this button will translate the user through the virtual pyramid.

**Middle Button** Dragging with this button will zoom the user in and out of the virtual pyramid.

**Right Button** Dragging with this button will rotate the user within the virtual pyramid.



## A.3 Editor Dialogs

The editor dialogs allows the user to select objects from a list and edit them individually using a tabbed interface. Items that can be edited are currently limited to layers and colormaps (see Sections A.3.1 and A.3.2), although future Kolam developers may eventually provide custom editors for heightmaps, mesh datasets, and individual image file formats.

### A.3.1 Layer Dialog

#### Layer Tab

The layer tab (see Figure A.2) displays the name of the current layer at the top. This name can be changed by clicking in the text field and typing. When many layers are open at once, naming makes it easier to find the one you want quickly.

The next two lines display the layer's current colormap and heightmap, if any. Any colormap listed in the Colormap Editor dialog can be selected as this layer's colormap, and any heightmap from the Layer Editor dialog can be selected for this layer's heightmap.

The rendering group in the bottom-left corner provides frequently used options. Toggling the visibility makes a layer visible or invisible. A layer is not processed on a per-frame basis when it is invisible. Toggling the enhancement determines whether it is subject to custom image processing settings such as histogram enhancement. Toggling the alpha channel determines whether black pixels are displayed as the color black or as completely transparent.

The two buttons on the right can open an image file for a new layer or close the current layer.

## **Layer Information Tab**

The layer information tab (see Figure A.3) shows useful data about the image file associated with the current layer.

**Filename** The filename of the image file with no directory information.

**Pathname** The full filesystem path to the image file.

**Dimensions** The width and height of the image in pixels.

**Tile Dimensions** The width and height of each image tile in pixels.

**Total Tiles** The total number of tiles in the image pyramid.

**Total Levels of Detail** The total number of levels of detail in the image pyramid.

**Color Model** The arrangement of color channels or color indices for each pixel.

**Data Type** The data type used to store each color channel or color index for each pixel.

**Bytes Per Pixel** The size of each pixel in memory measured in bytes.

**Compression Type** The compression algorithm used to compress each tile.

**Compressed Pyramid Size** The actual size of the image pyramid file on disk.

**Raw Image Size** The size of the uncompressed original resolution image excluding header information.

**Raw Pyramid Size** The size of the uncompressed image pyramid excluding header information.

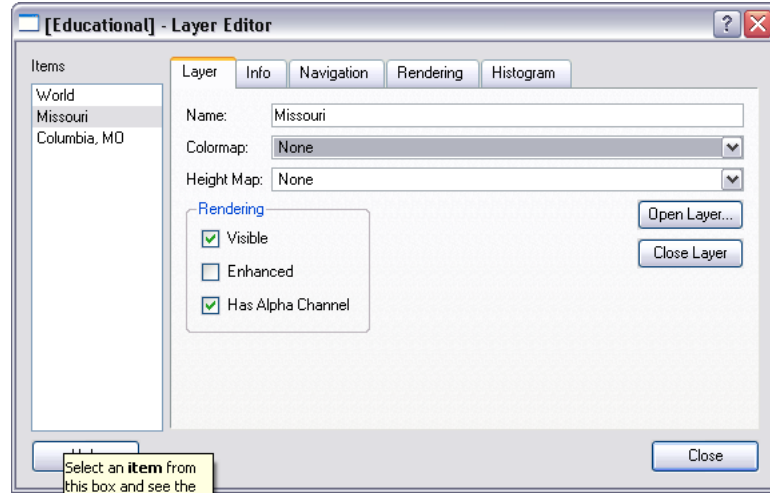


Figure A.2: This tab provides basic layer manipulation options.

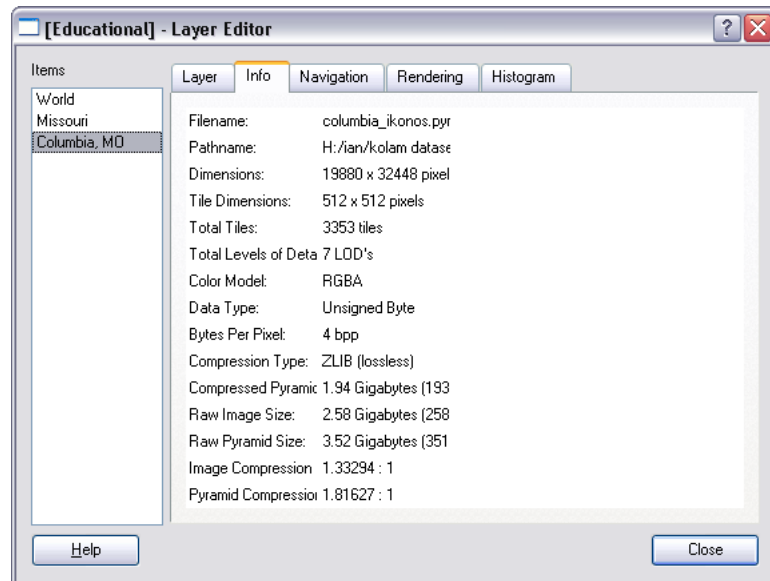


Figure A.3: This tab shows information about the image associated with this layer. Note: this screenshot was taken from an older version of Kolam where the rightmost portion of the info text was always truncated. This problem has been resolved in a more recent version.

**Image Compression Ratio** The ratio of the original resolution image's compressed size to its uncompressed size.

**Pyramid Compression Ratio** The ratio of the image pyramid's compressed size to its uncompressed size.

### **Navigation Tab**

The navigation tab (see Figure A.4) shows basic information regarding embedded datasets. The  $x$ - and  $y$ -offsets and the zoom factor of each layer are shown.

### **Rendering Tab**

The rendering tab (see Figure A.5) allows the user to manipulate how a dataset is rendered on the screen.

The pixel data group allows users to change how pixel data is interpreted by the rendering engine. This capability is useful when dealing with luminance images, which may be interpreted as a greyscale image, a colormapped image, or a single color channel of an image. By specifying the 'Data Format' parameter of three separate luminance images to be 'Red Component', 'Green Component' and 'Blue Component', a combined RGB color image can be generated (this is currently only supported by the raster renderer). All pixel data interpretations are checked to ensure the interpreted pixel size matches the number of bytes in the pixel data. For example, specifying a luminance 16 bit-per-pixel image to be interpreted as 2 bytes representing luminance and alpha is a valid combination, but interpreting that same image as 3 byte RGB pixels is not allowed.

The texture group allows users to modify certain OpenGL rendering parameters related to texture maps. The 'Wrap Mode' option controls how the borders of textures

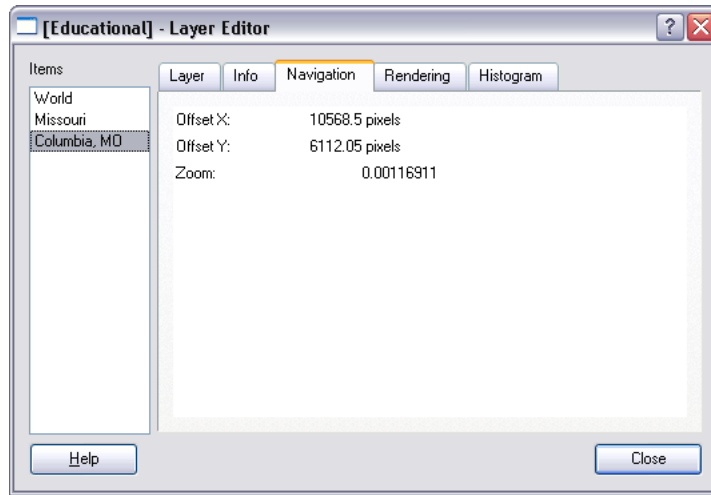


Figure A.4: The navigation tab provides information about the offset and zoom values of the current layer relative to the rest of the world.

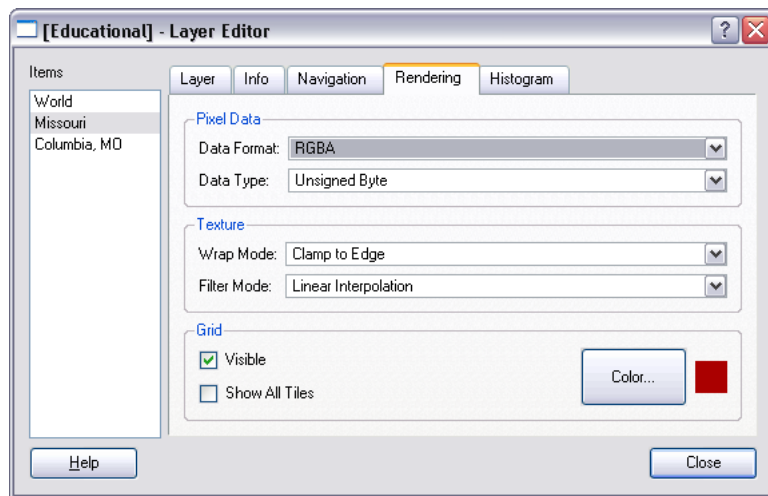


Figure A.5: The rendering tab lets users manipulate image data interpretations, OpenGL rendering parameters, and grid display properties.

are displayed (see Section 3.7.2). None of these modes are able to blend texels at a tile border with those of the opposite border on an adjacent tile, but do at least provide some control over the blending process. The default option, 'Clamp To Edge', blends texels at tile borders with themselves. This option is widely accepted as the preferred blending method and should usually not be changed. The 'Clamp' mode blends texels at tile border with the color black, thereby making tile borders more visible. The 'Repeat' mode blends texels at tile borders with the texel from the same tile's opposite edge. Moving on to the next combo box, the 'Filter Mode' option controls how interpolation between texels is performed. The default is 'Linear Interpolation', although 'Nearest Neighbor' interpolation is also available.

The grid group controls the display of a grid overlay on each layer to show current tile boundaries. This grid's visibility can be turned on or off, and its color can be changed (white is the default). If the 'Show All Tiles' option is enabled, all grid tiles will be displayed at their highest resolution. Otherwise, only the currently visible tiles will be rendered at their current resolution.

Unfortunately, some of these options were implemented early in Kolam development for testing and debugging purposes, but have since been "broken" by subsequent code modifications. The known non-functional options are currently those in the texture group, and the grid color options.

## **Histogram Tab**

The histogram tab (see Figure A.6) allows users to perform manual histogram-based enhancements on the current layer's image. The rendered histogram displays individual pixel colors as percentages of all colors in the image. Each histogram shows a single color channel. The x-axis of each channel is the range of color values, and the y-axis is the number of pixels having that color value.

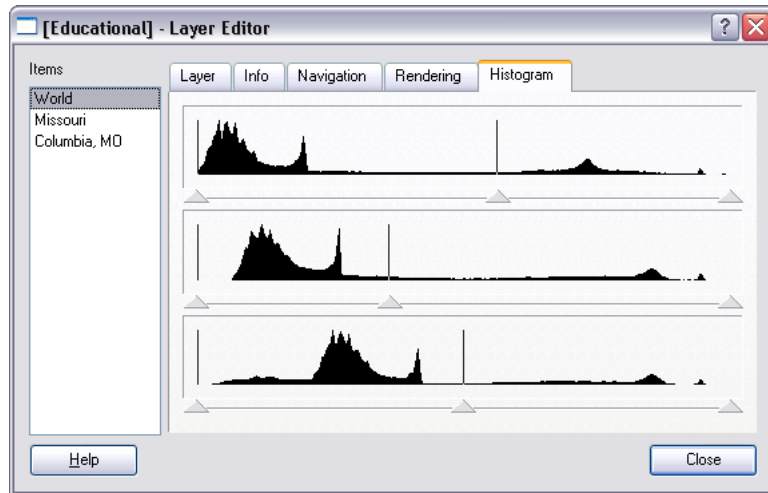


Figure A.6: By moving the three carets under each color channel display, the user can enhance the current layer's image to increase the colors' dynamic range.

The three carets under each color channel's histogram allow the user to manually adjust the mapping from input colors to output colors. The leftmost caret controls the low cutoff of the mapping; all colors to the left of the leftmost caret will display as the lowest possible color value (usually 0). The rightmost caret controls the high cutoff of the mapping; all colors to the right of the rightmost caret will display as the highest possible color value (usually  $2^8 - 1$ ). The middle caret controls the middle color value (usually  $\frac{2^8-1}{2}$ ); moving this caret modifies the color mapping between the leftmost and rightmost carets.

The typical use for modifying the color mapping is to increase the dynamic range of the colors. It is difficult for our eyes to distinguish between two similar colors, but it becomes easier the farther apart these color values are. When two carets are moved closer to one another, the colors in between are spread over a wider range, hence appearing farther apart from each other in luminance. It then becomes easier for our eyes to distinguish color values in this range. Figures A.7 and A.8 illustrate

this concept.

## A.3.2 Colormap Dialog

### Colormap Tab

The colormap tab (see Figure A.9) displays the name of the current colormap at the top. This name can be changed by clicking in the text field and typing. When many colormaps are open at once, naming makes it easier to find the one you want quickly.

The two buttons on the right can open a file for a new colormap or close the current colormap.

### Colormap Information Tab

The colormap information tab (see Figure A.10) shows useful data about the file associated with the current colormap.

**File Name** The filename of the image file with no directory information.

**Path Name** The full filesystem path to the image file.

**File Size** The size of the file containing this colormap's information on disk.

**Number of Colors** The number of color entries in this colormap.

### Colormap Creation Tab

The colormap creation tab (see Figure A.11) can be used to generate simple colormaps when an appropriate colormap file does not exist. Currently only simple linear gradient colormaps can be generated.

To generate a colormap, first choose two colors to interpolate between by clicking the 'Color 1...' and 'Color 2...' buttons. The RGB color values for color 1 will show



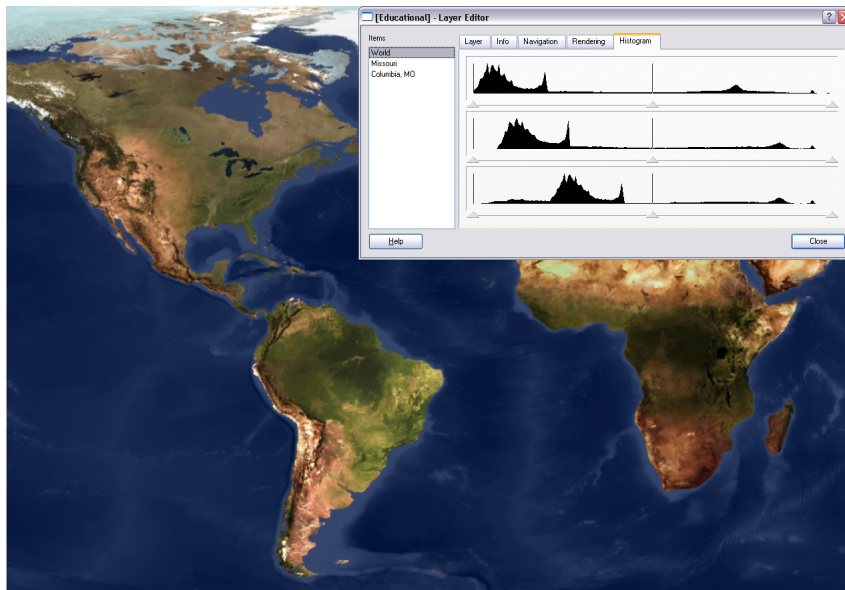


Figure A.7: The mapping from input colors to output colors is the identity (input and output colors are the same).

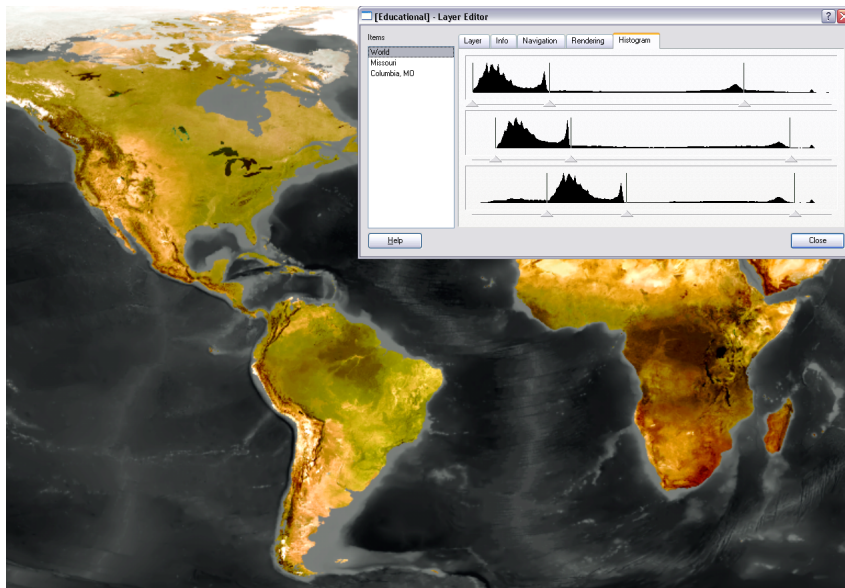


Figure A.8: The mapping from input colors to output colors has been modified to increase the dynamic range of each color channel.

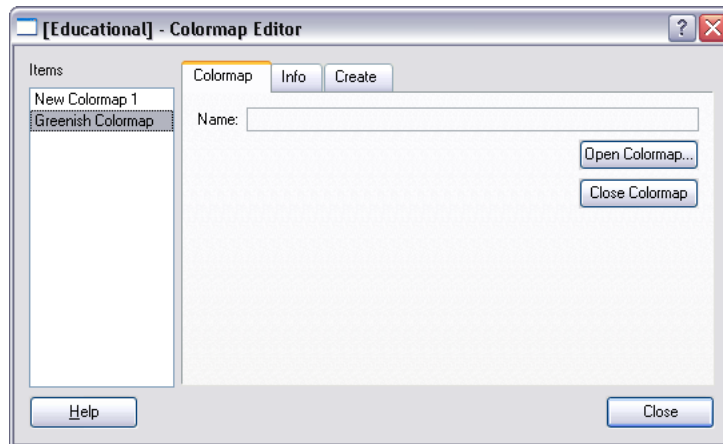


Figure A.9: This tab provides basic colormap manipulation options.

up in the top three fields of the second/middle column, and the values for color 2 will show up in the third/rightmost column. Since the color chooser dialog does not allow selection of alpha values, these values must be edited by hand in the fourth row of the last two columns. Now that two RGBA colors have been defined, the left column provides several options of how to use these colors to create a colormap. Either one of the values from color 1 or color 2 can be held constant, or values between them can be interpolated.

For a global lights dataset, it is usually desirable to keep a single color constant to represent lights and only vary the transparency. This way, bright areas occlude the underlying imagery but dark areas are transparent. Figure A.12 shows an example of this configuration. Figure A.13 shows a similar example with a cloud coverage dataset. Figures A.14 and A.15 show examples of combining colormaps and heightmaps to create some interesting images.

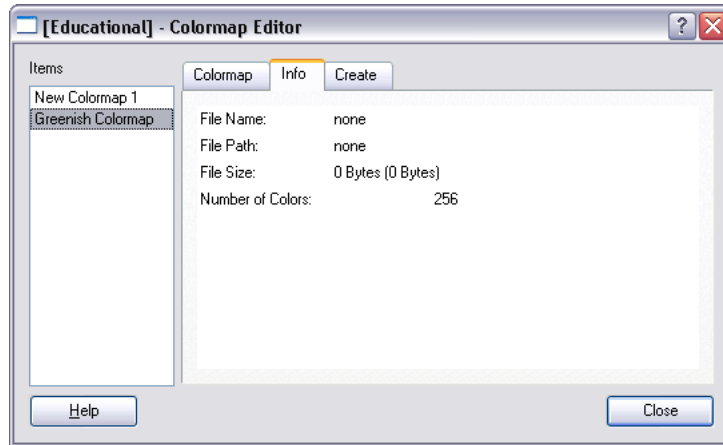


Figure A.10: This tab provides information about the current colormap. Note: there is no file information associated with this colormap since it was generated in memory by the colormap creation tab (see next section).

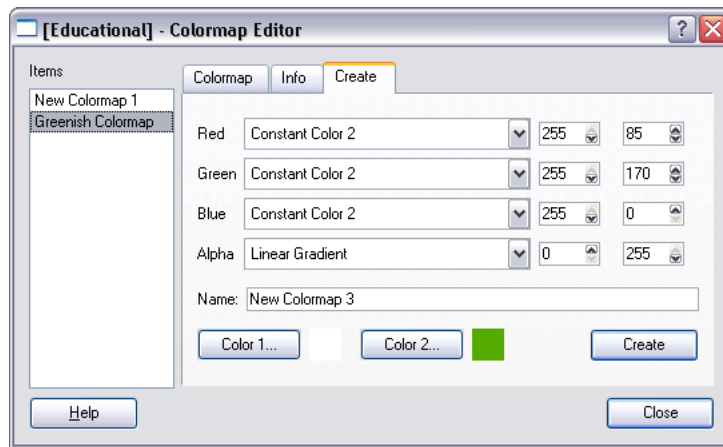


Figure A.11: The colormap creation tab allows users to create simple linear gradient colormaps on the fly.

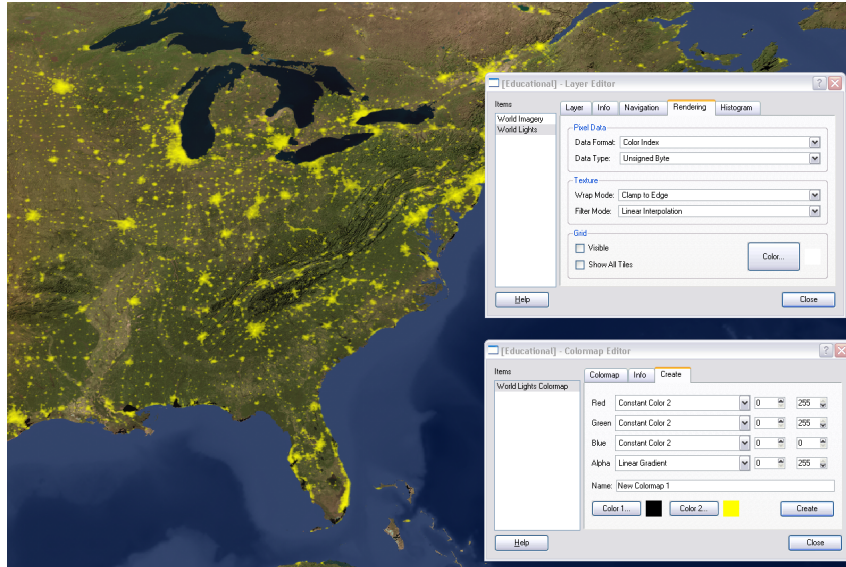


Figure A.12: This colormap was created to display world lights as a transparent overlay on top of world imagery.

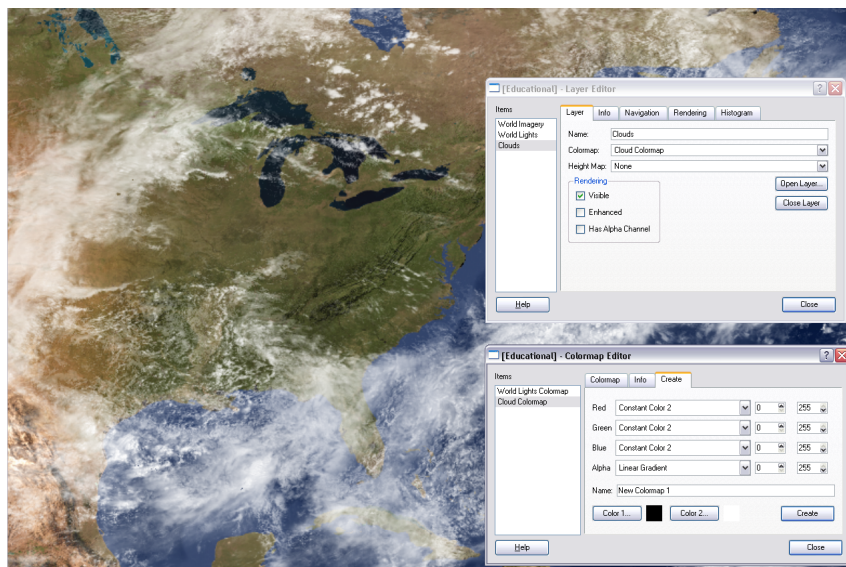


Figure A.13: This colormap was created to display cloud coverage as a transparent overlay on top of world imagery.

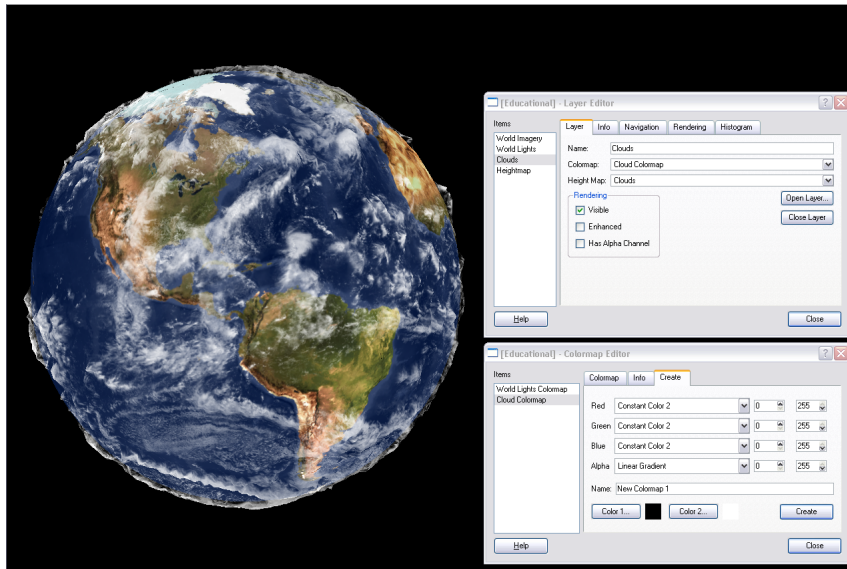


Figure A.14: This image was created using the same settings as in Figure A.13, except that the cloud layer uses its own image as a heightmap.

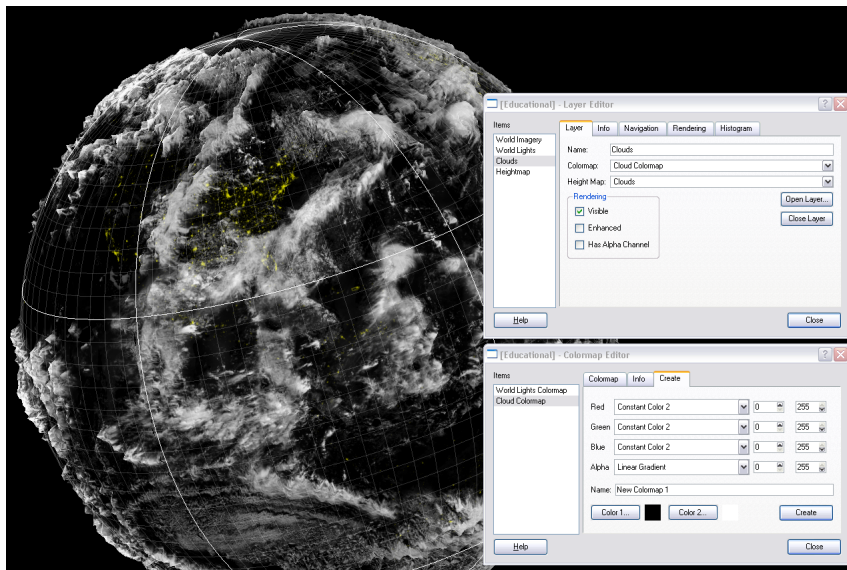


Figure A.15: This image was created using similar settings as in Figures A.12 and A.13, except that the grid display is enabled and the cloud layer uses its own image as a heightmap.





Figure A.16: Main menu and toolbar.

## A.4 Menus and Toolbars

### A.4.1 Menus

**File** This menu displays options for opening and closing datasets stored on disk.

**Open Image...** Load an image or heightmap from a file on disk. Available file formats are dependent on the application extensions loaded on startup. See Section B.3 for more information on image file formats.

**Open Colormap...** Load a colormap from a file on disk. See Section B.3 for more information on the colormap file format.

**Close** Close any files and release any memory associated with the current layer.

**Close All** Close all files and release all memory associated with each layer.

**Quit** Close the application and release all memory.

**Layer** This menu displays options pertaining to the current layer.

**Grid** Toggles grid visibility for the current layer. The grid displays boundaries between tiles in a dataset. In future implementations it may also represent

more real-world data such as latitude/longitude coordinates. See Section A.3.1 for more information on configuring the grid.

**Layer Editor...** Opens the main control panel for managing layers. See Section A.3.1 for more information on using this editor.

**Colormap Editor...** Opens the main control panel for managing colormaps. See Section A.3.2 for more information on using this editor.

**Display** This menu contains a list of the available dataset viewers. By default, this list will contain the orthogonal, oblique and sphere viewers. If a viewer has additional options, they may be accessed via a submenu of the viewer menu item. Additional viewers may be available depending on the application extensions loaded on startup. See Chapter 3 for more information on viewers.

**Window** This menu contains a list of available windows in addition to the main display window. By default, this list will contain the overview and cache view windows. See Section A.2 for more information on additional windows.

## A.4.2 Toolbars

The toolbar in Kolam provides buttons for switching between navigation modes (see Section 4.3). The two existing modes are both represented by the same icon to maximize confusion. The button on the left activates the roam/zoom/tilt navigator (see Section 4.3.1). This is the default mode and is activated when the application starts. The button on the right activates the embedded dataset positioning and scaling tool. This is an example of an image processing navigator (see Section 4.3.2). Users can use this tool to translate and scale the current active layer to "embed" it within a larger background layer.

## A.5 Keyboard Shortcuts

A few keyboard shortcuts are provided to simplify frequently used operations. All functionality described here can alternately be performed using the layer editor. Keyboard shortcuts for menu items are not listed here.

**G** Toggle grid display on/off.

**V** Toggle the current layer's visibility on/off.

**0-9** Activate one of layers 0-9 (depending on which numeric key was pressed). This layer becomes the current layer.

## A.6 Command Line Options

### A.6.1 Global Options

**-h/--help** Prints help and usage information.

**--no-plugins** Disables application extensions. Useful for debugging core application functionality.

**--num-threads** Sets the number of read/write threads used by the application. Useful for empirical performance testing and fine tuning for particular platforms.

**--cache-size** Sets the amount of memory (in megabytes) used for caching tiles in main memory. Useful for empirical performance testing and fine tuning for particular platforms.



## A.6.2 Layer Options

These options appear after an image filename in the command line sequence. Their effects are applied to the layer associated with the previous image filename.

- cm** Loads a colormap from a file and assigns it to the current layer.
- x** Sets the  $x$ -offset of the current layer. This number can be a floating point or integer value measured in pixels at a zoom factor of 1. Usually used for positioning embedded datasets.
- y** Sets the  $y$ -offset of the current layer. This number can be a floating point or integer value measured in pixels at a zoom factor of 1. Usually used for positioning embedded datasets.
- z** Sets the zoom factor of the current layer. This number is multiplied by the global zoom factor calculated by the viewing algorithm at each frame. Usually used for resizing embedded images.
- a** If this option is present, the current layer will be loaded with an artificial alpha channel. All black pixels will generate a corresponding 100% transparent alpha value while all other non-black pixels will generate a 100% opaque alpha value. The generated alpha channel uses additional space in the main memory cache but does not contribute significantly to the rendering time on typical graphics hardware. This option is useful for eliminating the black background in non-rectangular embedded images.

## A.7 Pyramid File Processing Tools

Several useful utility programs were written by Jared Hoberock to aid in the creation and modification of Kolam pyramid files.

### A.7.1 16sto16u

Biases 16 bit signed raw image data to 16 bit unsigned.

**Usage:** 16sto16u -i <infile.raw> -o <outfile.raw> -n <num\_channels>

-i: The input raw image filename

-o: The output raw image filename

-n: The number of channels per pixel

### A.7.2 16uto8u

Scales 16 bit unsigned raw image data to 8 bit unsigned.

**Usage:** 16uto8u -i <infile.raw> -o <outfile.raw> -n <num\_channels>

-i: The input raw image filename

-o: The output raw image filename

-n: The number of channels per pixel

### A.7.3 chopFooter

Removes the footer of a file.

**Usage:** chopFooter -i <infile.raw> -o <outfile.raw> -s <footer\_size>

- i: The input raw image filename
- o: The output raw image filename
- s: The number of bytes to remove from the end of the file

#### A.7.4 chopHeader

Removes the header of a file.

**Usage:** chopHeader -i <infile.raw> -o <outfile.raw> -s <header\_size>

- i: The input raw image filename
- o: The output raw image filename
- s: The number of bytes to remove from the beginning of the file

#### A.7.5 extractChannel

Extracts a channel from a raw image file.

**Usage:** extractChannel -i <infile.raw> -o <outfile.raw>  
-c <channel> -n <num\_channels> -d <bytes\_per\_channel>  
[-h <skip\_header\_bytes>] [-f <skip\_footer\_bytes>] [-w]

- i: The input raw image filename
- o: The output raw image filename
- c: The channel to extract
- n: The number of channels per pixel
- d: The number of bytes per channel

- h: The number of header bytes to skip
- f: The number of footer bytes to skip
- w: Enables swabbing of output bytes

### A.7.6 extractLevel

Extracts and writes to disk a given level from a pyramid file. Pyramid levels are numbered starting with zero at the base level, which is the level of highest resolution. Output is written as a raw image file. The output image is written as pixel interleaved, frame interleaved, or as a single color channel. A kolam header file is created for the raw output image in the same location as the output file.

**Usage:** `extractLevel -i <infile.pyr|infile.kh> -o <outfile.raw>`  
`-l <pyramid_level> -e <red|green|blue|pixel|frame> [-p]`

- i: The input Kolam header or pyramid filename
- o: The output raw image filename
- l: The pyramid level to extract
- e: Specifies which data to extract
- p: Enables writing black pad pixels

### A.7.7 interleaveRaw

Interleaves up to 10 channels into one raw image file.

**Usage:** `interleaveRaw -i <file0,file1,file2,...,file9>`  
`-o <outfile.raw> -b <pixel_depth>`

- i: A comma-separated list of input raw image filenames
- o: The output raw image filename
- b: The number of bytes per channel

### A.7.8 jpgToHybrid

Converts a JPEG-compressed pyramid file to a JPEG hybrid pyramid by changing its magic number appropriately. Kolam viewers will page this new file as a hybrid pyramid. Note that it will not reclaim disk space by removing unnecessary pyramid levels.

**Usage:** `jpgToHybrid -i <jpeg_zero_pyramid.pyr>`

- i: The input Kolam pyramid filename

### A.7.9 pyramidCompare

Compares two images and displays noise statistics for a specific pyramid level. Types of noise statistics displayed are mean squared error, root mean squared error, peak absolute error, mean absolute error, and peak signal-to-noise ratio.

**Usage:** `pyramidCompare -o <original.kh|original.pyr>  
-c <compressed.kh|compressed.pyr> -l <which_level>`

- o: The first (original) input Kolam header or pyramid filename
- c: The second (compressed) input Kolam header or pyramid filename
- l: The pyramid level for which to display statistics

### A.7.10 pyramidInfo

Displays pyramid information about an input Kolam header or pyramid file. Will output statistics for a complete pyramid or, if specified, a single pyramid level. Pyramid levels are numbered starting with zero at the lowest level, which is the level of highest resolution.

**Usage:** `pyramidInfo -i <infile.kh|infile.pyr> [-l <pyramid_level>]`  
`[-b] [-h]`

- `-i:` The input Kolam header or pyramid filename
- `-l:` The pyramid level for which to display statistics
- `-b:` Prints out the full number of bytes for each level
- `-h:` Prints a help screen and exits

### A.7.11 removeChannel

Removes a channel from a file and writes the remaining channels to the output file. You don't have to specify a channel to remove if you don't want to; do this to only chop the header and footer.

**Usage:** `removeChannel -i <infile.raw> -o <outfile.raw>`  
`[-c <channel>] -n <num_channels> -d <bytes_per_channel>`  
`[-h <skip_header_bytes>] [-f <skip_footer_bytes>] [-w]`

- `-i:` The input raw image filename
- `-o:` The output raw image filename
- `-c:` The channel to extract

- n: The number of channels per pixel
- d: The number of bytes per channel
- h: The number of header bytes to skip
- f: The number of footer bytes to skip
- w: Enables swabbing of output bytes

### A.7.12 writeTiledPyramid

Creates a Kolam pyramid file based on an input Kolam header or pyramid file. The specified compressor and filter are used to create each level of the output pyramid. When building a new pyramid from an existing pyramid, the utility creates a new pyramid level by sampling the lower level of the source pyramid. This utility can also assemble a mosaic pyramid made of an array of smaller pyramids as tiles. Following the `-m` option, the dimensions of the mosaic are listed as `rows*cols`, followed by a single string of the input files separated by commas. Use the `-h` option to show an extensive help screen describing options for downsampling filters and tile compression methods.

```
Usage: writeTiledPyramid <-i <infile.kh|infile.pyr> |
      -m <rows*cols,file1,file2,...,filerows*cols>> -o <outfile.pyr>
      -f <median|mean|subsample> -c <rle|zlib|bz2|jpg|jpgv2|jp2|none>
      [-v] [-h] [-q <quality>] [-r <quality>]
```

- i: The input Kolam header or pyramid filename
- o: The output Kolam pyramid filename
- m: *see above description*

- f: Specifies the downsampling filter type
- c: Specifies the tile compression method
- v: Print out verbose information by level
- h: Print help screen
- q: Set quality for JPEG compressor
- r: Set quality for JPEG-2000 compressor



# APPENDIX B

## Developer's Manual

### B.1 Building the Kolam Application from Source

#### B.1.1 Platforms and History

The original implementation of Kolam was developed on the SGI Irix 6.5 operating system. It has since been ported to Redhat Linux, Mac OS X, and Windows XP. The most recent development has been done predominantly on Windows XP.

#### B.1.2 Third-party Dependencies

##### Required Dependencies

These dependencies are required to build the Kolam application and its pyramid file utility programs (see Section A.7). These dependencies typically come pre-installed on many Linux and UNIX-based operating systems. Ports are available for Win32

platforms, but they must be acquired by the developer.

**Qt 3.3.3 Educational :** The Qt toolkit provides cross-platform C++ classes for GUI development. The `qmake` utility included with Qt is required to run the makefiles needed to build Kolam.

<http://www.trolltech.com>

**pthread :** The `pthread` library implements the threading component of the POSIX 1003.1 2001 standard. The Kolam thread class is based on `pthreads`.

<http://sourceware.org/pthreads-win32>

**XGetopt :** On UNIX-based systems, the standard command line parsing function is `getopt()`. This library provides a similar Win32 version.

<http://www.codeproject.com/cpp/xgetopt.asp>

## Optional Dependencies

These dependencies are optionally required to build Kolam. Each provides support for compression methods that may be used by the Kolam reader classes and the pyramid file utility programs (see Section A.7). To disable a specific reader compression type when building Kolam, open the `qmake.defs.<platform>` file corresponding to your current platform, and remove the `KOLAM_READER_<COMPRESSION>` definitions corresponding to the methods you wish to disable.

**zlib 1.1.4 :** The `zlib` data compression and decompression library.

<http://www.zlib.net>

**bzip2 1.0.2 :** The `bzip2` data compression and decompression library.

<http://www.bzip.org>

**libtiff 3.5.7** : The Tag Image File Format (TIFF) image storage library.

<http://www.libtiff.org>

**libjpeg 6b** : The Joint Photographic Experts Group (JPEG) image compression and decompression library.

<http://www.ijg.org>

**jasper 1.500.4** : An open source implementation of the JPEG-2000 Part-1 standard for image compression and decompression.

<http://www.ece.uvic.ca/~mdadams/jasper>

**libpng 1.2.4** : The Portable Network Graphics (PNG) image compression and decompression library.

<http://www.libpng.org>

**gdal 1.1.7** : The Geospatial Data Abstraction Library (GDAL) image storage library.

<http://www.gdal.org>

### **B.1.3 Compilers**

The following compilers have been used to successfully build Kolam.

**Windows** : Microsoft Visual Studio 7.0

**Linux** : gcc 3.x

**IRIX** : CC

**MacOS X** : gcc 3.x

## B.2 Writing Custom Plug-in Application Extensions

Application extensions, or plugins, have a wide range of categories in which they can add functionality. An extension may span any number of these categories to group functionality in the way that makes the most sense. The potential categories include file formats, viewers, renderers, navigators, GUI elements, and processing routines. At present, plugins providing GUI components and basic histogram processing routines have been successfully developed using the existing extension model.

### B.2.1 Plugin Loading and Initialization

Plugins are implemented as DLLs (Dynamic Link Libraries) on Windows or as shared libraries on UNIX-based operating systems. On startup, Kolam will look for a directory called 'plugins' located in the same directory as the executable file. It will then search the subdirectories of the plugins directory to find DLL or shared library files to load as plugins.

A plugin developer should first create a class that inherits from the `Kolam Plugin` class. A single object of this derived class type will be instantiated by the application at load time. The plugin must be registered with the application in order for this to happen. This is done by placing the `REGISTER_PLUGIN(PLUGIN_CLASS)` macro in one source file of the plugin library. This macro takes as its only argument the class type of the derived plugin class. It creates two C functions, `createPlugin()` and `deletePlugin()`, which are called by the application.

A plugin developer can also define the plugin's version information to ensure that dependencies between plugins are resolved. This is done using the `IDENTIFY_PLUGIN(NAME,`

VERSION) macro. The `Plugin::checkDependencies()` member function can then be used to check dependencies at runtime. A list of dependency objects is passed in as an argument, and the function ensures that the current plugin's version is greater than or equal to each of the dependencies' versions.

## **B.2.2 Application Data Access and Processing Hooks**

Plugins have access to all public data fields and functions of the application objects. A pointer or reference to any application object can typically be obtained by accessing members of the global application class. A pointer to the global application object is initialized by the base `Plugin` class when class constructor is invoked.

Certain types of plugins can be implemented using nothing more than the data access paradigm. Tabbed dialog plugins, for example, can add a new tab to the Layer Editor dialog at initialization. All further plugin tasks are initiated by GUI events. A plugin developer who wanted to change some aspect of the data visualization process could subclass one of the viewer or renderer classes, then add an instance of the new derived class to the list of viewers or renderers (remember to synchronize access to the rendering thread while modifying these lists).

Although a great deal of functionality can be added using data access alone, the concept of processing hooks can simplify the plugin development task in certain cases. A processing hook is a predefined point in the code where the current thread calls a list of callback functions. These callback functions are implemented and inserted into the appropriate lists by each plugin. There are currently very few processing hooks in the Kolam application, so this is mainly a task for future developers.

### B.2.3 Attaching Custom Data to Layers and Colormaps

When writing a plugin, it may become apparent that the layer class which encapsulates a generic dataset would benefit from additional data storage. Considering a histogram plugin as an example, it makes far more sense to cache and store histogram data with each layer rather than creating and managing a separate and parallel list for layer data. Managing this additional list becomes complicated when the user opens and closes layers during program operation, as appropriate events must be generated and sent to all plugins. Each plugins must then duplicate the functionality for adding, removing, rearranging and grouping layers. We would prefer to keep these tasks in the application and let plugin developers focus on implementing *new* functionality.

Additional data can be attached to the layer class by implementing the `LayerData` interface. The `LayerData` interface is actually a subinterface of the `AttachableData` interface. In fact, any class that implements the `Attachable` interface can have any subclass of the `AttachableData` interface attached to it. Currently, only the `Layer` and `Colormap` classes implement the `Attachable` interface. Each attachable object stores either an array or hash table of pointers to extension data objects for fast data access regardless of the number of loaded extensions. Each extension is assigned a unique ID by the application that can be used as either an array index or hash function input for accessing its `AttachableData` subclass. See Figure B.1 for a C++ code sample.

```

/* class: HistogramData                                     *
 * - Create objects of this type to be attached to each layer. */

class HistogramData : public AttachableData
{
public:

    HistogramData( Attachable *attached ) :
        AttachableData( attached )
    {
        // Plugin-specific per-layer initialization goes here...
    }

    // Plugin-specific per-layer member functions go here...

private:

    // Plugin-specific per-layer data fields go here...
};

/* class: HistogramPlugin                                  *
 * - This singleton object is created as a dynamically loaded plugin. */

class HistogramPlugin : public Plugin, public LayerDataOwner
{
public:

    HistogramPlugin( App *app ) :
        Plugin( app ),
        LayerDataOwner( app )
    {
        // Pass data owner (this) to
        // Plugin::attachLayerData() member function.

        attachLayerData( this );
    }

    /* member function: initAttached()                       *
     * - This function is called whenever a new layer is created. */

    AttachableData *initAttached( Attachable *attached )
    {
        return new HistogramData( attached );
    }

    // Plugin-specific global member functions go here...

private:

    // Plugin-specific global data fields go here...
};

```

Figure B.1: This is a skeleton of the HistogramPlugin C++ header file.

## B.3 File Formats

### B.3.1 Pyramid File Format

The file format of a Kolam image pyramid consists of a header, a tile offset table, and all tiled image data. Throughout the file, all integer values (excluding image pixel values) are stored in big-endian format. This implies that on Windows/Intel platforms, the byte order must be reversed when converting from disk integers to memory integers and vice-versa.

The header describes the pyramid's image and tile dimensions. It contains the following fields, in this order:

**Magic Number** The magic number is a variable-length string identifying the tile compression method used.

**Image Width** The width in pixels of the original resolution image as a 32-bit 2's complement signed integer.

**Image Height** The height in pixels of the original resolution image as a 32-bit 2's complement signed integer.

**Bytes Per Pixel** The number of bytes per pixel as a 32-bit 2's complement signed integer.

**Tile Width** The width in pixels of each pyramid tile as a 32-bit 2's complement signed integer.

**Tile Height** The height in pixels of each pyramid tile as a 32-bit 2's complement signed integer.



**Number of Tiles** The total number of pyramid tiles as a 32-bit 2's complement signed integer.

Notice that there is no level of detail information stored here, or in any subsequent portions of the file. The number of resolutions is calculated by finding the smallest value of  $i$  for which  $\lceil \frac{I_w}{2^{i-1}T_w} \rceil = 1$  and  $\lceil \frac{I_h}{2^{i-1}T_h} \rceil = 1$ . Since the total number of tiles can be calculated by  $\sum_{i=0}^n \lceil \frac{I_w}{2^i T_w} \rceil \lceil \frac{I_h}{2^i T_h} \rceil$ , this field is not necessary to include in the header either, but exists anyway.

Next in the file is an table of tile offsets. Since the tiled pyramid format supports any type of per-tile compression, tiles can potentially be of varying lengths. This table of offsets is necessary to perform fast seeks to the desired tile data without searching. The number of entries in this table is equal to the total number of tiles in the image. Each entry is a 64-bit 2's complement signed integer indicating the offset of the tile's data from the beginning of the file.

The final portion of the file contains the data for each tile. The tiles may appear in any order, but each tile must be contiguous. The interpretation of the tile data is dependent on the magic number read from the header. This number indicates which Kolam module should be used to decode tiles from the image. At present, tiles contain only image data, but may be either raw (uncompressed) or compressed using ZLib, BZ2 or RLE (run-length encoding).

### **B.3.2 Colormap File Format**

The colormap file is in plain-text ASCII format. The file contains a list of  $2^8$  color values for 8 bit-per-pixel color indexed images, or  $2^{16}$  color values for 16 bit-per-pixel color indexed images. Each value must be on a separate line, and should consist of either 3 or 4 integer values comprising an RGB or RGBA color value. Each integer

value must be in the range 0 to 255. Each 3 or 4 component color value represents the output (mapped) color of a color index.

Color indices are assigned to the color values in the file sequentially. For example, the color index 0 will be mapped to the first color value, color index 1 to the second color value, and so on.

Please note that the current Kolam versions do not support this format. The code from Joshua Fraser's original implementation is still present in the source files, but is commented out and requires a few syntax changes to complete the transition from C to C++. This functionality could easily be resurrected by future Kolam developers, and the format is documented here to aid their efforts.

### B.3.3 Kolam Raw Header File Format

In addition to image pyramid files, Kolam can also load raw images. Since the location on disk of any pixel in a raw image can be easily calculated, tiles can be constructed on-the-fly by reading in each scanline one at a time. Coarser level of detail tiles can be constructed by skipping an appropriate number of scanlines, and discarding the appropriate number of pixels from each scanline.

Since raw image files are a simple sequence of pixel data and contain no metadata, a separate header file format is needed to provide the image width, height and number of bytes per pixel. Additional data, such as tile width and height, may be specified also. See Figure B.2 for an example file. The following is a description of the format of the required fields:

**KOLAM RAW** This is the magic number for Kolam raw images (see Section B.3.1).

This must be the first string in the file.

**filename** The path to the raw image file on disk. This must either be a full path or

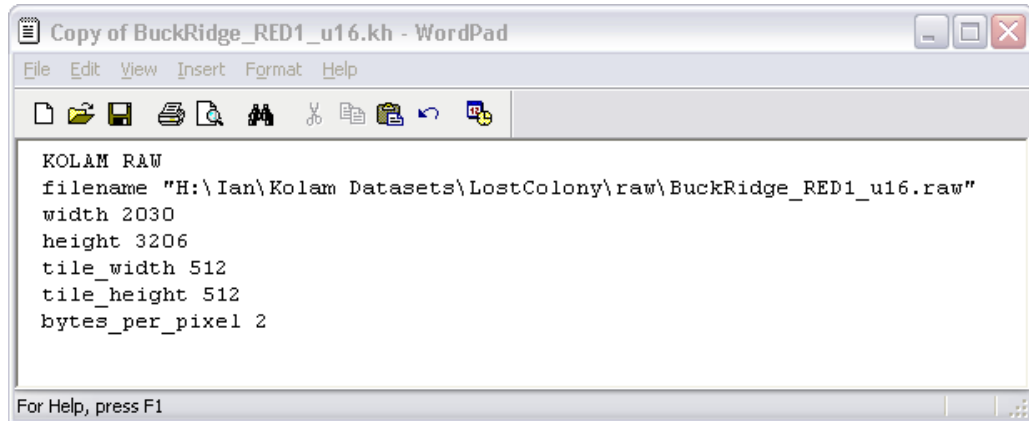


Figure B.2: An example of the Kolam raw header file format.

relative to the current directory.

**width** The width in pixels of the raw image.

**height** The height in pixels of the raw image.

**tile\_width** The desired width in pixels of each image tile.

**tile\_height** The desired height in pixels of each image tile.

**bytes\_per\_pixel** The number of bytes per image pixel.

# BIBLIOGRAPHY

- [1] K. Palaniappan and J. Fraser, “Multiresolution tiling for interactive viewing of large datasets,” in *Proceedings of the 17th International Conference on Interactive Information and Processing Systems (IIPS) for Meteorology, Oceanography and Hydrology, 81st AMS*, Jan. 2001, pp. 338–342.
- [2] Space Imaging, Inc., “IKONOS Space Imaging.” [Online]. Available: <http://www.spaceimaging.com/products/ikonos/index.htm>
- [3] National Aeronautics and Space Administration (NASA), “Moderate Resolution Imaging Spectroradiometer (MODIS).” [Online]. Available: <http://modis.gsfc.nasa.gov>
- [4] P. D. Sulatycke and K. Ghose, “A fast multithreaded out-of-core visualization technique,” in *Proceedings of the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*. San Juan, Puerto Rico: IEEE, Apr. 1999, pp. 569–575.

- [5] U.S. National Library of Medicine, “The Visible Human Project,” Sept. 2003. [Online]. Available: [http://www.nlm.nih.gov/research/visible/visible\\\_human.html](http://www.nlm.nih.gov/research/visible/visible\_human.html)
- [6] C. Silva, Y. Chiang, J. El-Sana, and P. Lindstrom, “Out-of-core algorithms for scientific visualization and computer graphics,” in *Visualization 2002*, 2002, Course Notes. [Online]. Available: <http://citeseer.ist.psu.edu/silva02outcore.html>
- [7] W. T. Correa, J. T. Klosowski, and C. T. Silva, “Visibility-based prefetching for interactive out-of-core rendering,” in *Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*. Washington, DC: IEEE, 2003.
- [8] D. Ellsworth, “Accelerating demand paging for local and remote out-of-core visualization,” in *NAS Technical Report NAS-01-004*, June 2001. [Online]. Available: <http://www.nas.nasa.gov/Research/Reports/Techreports/2001/PDF/nas-01-004.pdf>
- [9] M. Cox and D. Ellsworth, “Application-controlled demand paging for out-of-core visualization,” in *Proceedings of the 8th conference on Visualization '97*. Los Alamitos, CA: IEEE Computer Society Press, 1997, pp. 235–ff.
- [10] C. L. Bajaj, V. Pascucci, D. Thompson, and X. Y. Zhang, “Parallel accelerated isocontouring for out-of-core visualization,” in *Proceedings of the 1999 IEEE Symposium on Parallel Visualization and Graphics*. New York, NY: ACM Press, 1999, pp. 97–104.
- [11] W. Bethel, B. Tierney, J. Lee, D. Gunter, and S. Lau, “Using high-speed WANs and network data caches to enable remote and distributed visualization,” in *Pro-*

- ceedings of the 2000 ACM/IEEE Conference on Supercomputing.* Washington, DC: IEEE Computer Society, 2000.
- [12] C. C. Tanner, C. J. Migdal, and M. T. Jones, “The clipmap: A virtual mipmap,” in *Proceedings of SIGGRAPH 98*. ACM, July 1998, pp. 151–158.
- [13] E. W. Weisstein, “Geometric Series,” 2004, From MathWorld—A Wolfram Web Resource. [Online]. Available: <http://mathworld.wolfram.com/GeometricSeries.html>
- [14] National Center for Supercomputing Applications (NCSA) at the University of Illinois at Urbana-Champaign (UIUC), “A User’s Guide for HDF5,” July 2003, Release 1.4.5. [Online]. Available: <http://hdf.ncsa.uiuc.edu/HDF5/doc/H5.user.html>
- [15] J. B. Fraser and K. Palaniappan, “Kolam: A high-performance architecture for the visualization of extremely large datasets,” 2002, Dept. of Computer Science, Technical Report, University of Missouri, Columbia.
- [16] P.-G. Maillot, “Using quaternions for coding 3D transformations,” in *Graphics Gems*. London, United Kingdom: Academic Press, 1990, pp. 498–515.
- [17] F. S. Hill, *Computer Graphics*, 2nd ed. Upper Saddle River, NJ: Prentice Hall, 1996.
- [18] OpenGL Architecture Review Board, C. Frazier, and R. Kempf, *OpenGL Reference Manual*, 2nd ed. Addison-Wesley, 1999.
- [19] D. Zwillinger, *CRC Standard Mathematical Tables and Formulae*, 30th ed. CRC Press, 1995.

- [20] G. B. Thomas and R. L. Finney, *Calculus and Analytic Geometry*, 9th ed. Addison-Wesley, 1996.
- [21] P. Lindstrom and V. Pascucci, “Terrain simplification simplified: A general framework for view-dependent out-of-core visualization,” in *IEEE Transactions on Visualization and Computer Graphics*, vol. 8, no. 3, July–Sept. 2002.