

**VOLUME TERRAIN MODELING AND RENDERING FOR
VISUAL FLYTHROUGH**

An Internal Report

By

Edmond C. Prakash

Technical Report Number: cvc970415

Project Director

PROF. ARIE E. KAUFMAN

Center for Visual Computing
and Department of Computer Science

SUNY at Stony Brook

NEW YORK, NY 11794-4400, USA

April 15, 1997

Abstract

This report describes the different algorithms implemented as part of the Visual Flythrough project. The main emphasis has been to implement algorithms for realistic rendering of volumetric terrain. Functions have been implemented for handling various input data formats. A simple voxelization scheme has been implemented for generation of anti-aliased voxel terrain model. VolVis has been extended to support terrain models using a flat scene graph. Fast modeling and rendering has been achieved by using multiple CPUs on the system. Several sample terrains have been rendered to validate the utility of this scheme.

The shared memory multiprocessor system recently has attracted a lot of attention because of its low cost and wide availability to the graphics researchers. In this work, we show practical parallel solutions to the following volume graphics [7] problems: volume texture modeling, volume texture rendering, and voxelization for unstructured grid rendering. All these solutions process 2D and 3D arrays and employ multiple processors on the shared memory system. We present results for parallel execution time, parallel memory requirements and overheads for the three algorithms described in this work. We also present results of performance prediction based on both analytical and experimental results.

This report explains the following implementations which have been done as part of this work:

- Volume Modeling of Terrain
- Volume Rendering of Terrain
- Parallel Volume Modeling
- Parallel Volume Rendering

Contents

Abstract	i
1 Volume Terrain Modeling	1
1.1 Background	1
1.2 Terrain Data and Formats	2
1.2.1 Digital Terrain Elevation Data (DTED)	2
1.2.2 RGB	2
1.2.3 Aerial Photo and Satellite Images	3
1.3 Available Data Formats	4
1.3.1 Format of Hughes data set	4
1.3.2 Format of Daniel Cohen Data set	4
1.3.3 Format of Valley data set	5
1.3.4 Format of LA data set	5
1.4 Terrain Modeling	6
1.4.1 Input of Terrain data	6
1.4.2 Terrain Tiling	6
1.4.3 Terrain Texture Generation	7
1.4.4 Loading Terrain Volume in VolVis	8
1.4.5 Rendering Terrain Volume in VolVis	8
1.4.6 Utilities	8

1.5	Implementation and Results	8
1.6	Working Model and Results	9
2	Parallel Volume Graphics Algorithms	11
2.1	Introduction	11
2.2	Parallel Rendering Algorithms	12
2.2.1	Volume Texture Modeling of Terrain	12
2.2.2	Volume Texture Rendering of Terrain	13
2.2.3	Voxelization for Unstructured Grid Rendering	14
2.2.4	Related Work	15
2.2.5	Design Goals	15
2.3	Implementation and Results	16
2.4	Parallel Performance Estimation	16
2.4.1	Model of Performance - I: Green's Method	18
2.4.2	Model of Performance - II: Tom's Method	19
2.5	Source Code	21
2.6	Summary	21
3	Conclusion	23
	Bibliography	25

List of Figures

2.1	VTM _{sequential}	13
2.2	VTM _{parallel}	13
2.3	VTR _{sequential}	14
2.4	VTR _{parallel}	14
2.5	VUG _{sequential}	14
2.6	VUG _{parallel}	15

Chapter 1

Volume Terrain Modeling

1.1 Background

The main objective of this project is to design and implement a high performance system for modeling and realistic rendering of terrains using volumetric representation.

Several works have been reported in literature on Terrain Flythrough. In [18], a voxel-based forward projection algorithm for rendering surface and volumetric data has been presented which uses a special hardware system to achieve real time performance. In [1], a real time photo-realistic visual flythrough is presented running on PVS and SGI Challenge.

Several methods have been proposed for simplification of terrain models. In [12], a fast multiresolution surface meshing technique has been proposed. In [15], a technique has been proposed for automatic generation of triangular networks from dense terrain models using triangular irregular networks using greedy cuts. In [6], a technique for voxel based object simplification is presented. Multiresolution models of topographic description is presented in [3].

Other related work for GIS applications have been reported in [16] and in [8]. In [16], a technique for automated generation of visual simulation databases using remote sensing and GIS visualization is presented. In [8] a technique for real time 3D geographical information system visualization is presented.

1.2 Terrain Data and Formats

1.2.1 Digital Terrain Elevation Data (DTED)

Terrain data consists of the topographical features of land surface. Several formats are used to store the terrain information. DTED is made up of 1 degree by 1 degree cells, 1 cell per file. The terrain elevation within the cell is sampled at 3 arc second intervals in latitude and longitude (at higher latitudes the sampling rate in longitude drops off) for an effective resolution of approximately 100 meters. The absolute resolution in position is listed as 30 meters.

Higher resolution data exists, but it is very difficult to get even with an applicable contract with the US Department of Defense. DFAD is a vector definition of terrain features, both natural and cultural. It was originally developed for radar applications. As such, combining it with DTED can cause some very strange artifacts, like rivers on the sides of hills. Some of the more interesting features, such as urban areas, are just listed as an enclosed area, not very useful for a visualization.

Since both of these formats are based on sampled data, there is a fairly extensive conversion to viewable data. For most applications, a complete conversion will generate enough geometry to overload any current machine. Therefore, a runtime loader is not appropriate.

1.2.2 RGB

IRIS image files are used to store 1,2 and 3 dimensional arrays of pixel values that contain either 1 or 2 bytes per pixel. Pixel values are signed integers that cover the range 0...255 or -32768...32767 (i.e., 1 or 2 bytes). Image files are currently used to store RGB screen dumps, black and white images, color index images, as well as color maps.

The image library provides tools to manipulate these files. To include the image library place the token `-limage` on the compile line for your program. Also, be sure to include `image.h` from `/usr/include/gl` in any source files that use these routines.

Images you can download from the net are typically TIFF, GIF or JPEG format, although a number of others exist. Whilst providing a good idea of what is in an image, they are not useful for serious applications. They have the advantage of being a manageable size - typically of the order of 100KB-1MB (compared to 100MB for a full scene) and are often

available free.

1.2.3 Aerial Photo and Satellite Images

Aerial photography has two uses that are of interest: (1) Cartographers and planners take detailed measurements from aerial photos in the preparation of maps. (2) Trained interpreters utilize aerial photos to determine land-use and environmental conditions, among other things. Although both maps and aerial photos present a "bird's-eye" view of the earth, aerial photographs are NOT maps. Maps are orthogonal representations of the earth's surface, meaning that they are directionally and geometrically accurate (at least within the limitations imposed by projecting a 3-dimensional object onto 2 dimensions). Aerial photos, on the other hand, display a high degree of radial distortion. That is, the topography is distorted, and until corrections are made for the distortion, measurements made from a photograph are not accurate. Nevertheless, aerial photographs are a powerful tool for studying the earth's environment.

Because most GISs can correct for radial distortion, aerial photographs are an excellent data source for many types of projects, especially those that require spatial data from the same location at periodic intervals over a length of time. Typical applications include land-use surveys and habitat analysis, Land-Use Planning and Mapping, Geologic Mapping, Archaeology and Species Habitat Mapping

LANDSAT refers to a series of satellites put into orbit around the earth to collect environmental data about the earth's surface. The LANDSAT program was initiated by the U.S. Department of Interior and NASA under the name ERTS, an acronym which stands for Earth Resources Technology Satellites. ERTS-1 was launched on July 23, 1972, and was the first unmanned satellite designed solely to acquire earth resources data on a systematic, repetitive, multispectral basis. Just before the launch of the second ERTS satellite, NASA announced it was changing the program designation to LANDSAT, and that the data acquired through the LANDSAT program would be complemented by the planned SEASAT oceanographic observation satellite program. ERTS-1 was retroactively named LANDSAT-1, and all subsequent satellites in the program have carried the LANDSAT designation. Over time, the sensors carried by the LANDSAT satellites have varied as technologies improved and certain types of data proved more useful than others.

1.3 Available Data Formats

This section describes the code which has been developed for this project, sample data and other utilities. The current version of the software is located under FLIGHT.

```
FLIGHT = /home/fs8/flight_sim/
```

1.3.1 Format of Hughes data set

```
Path: FLIGHT/data/hughes/tape2/hier/*.rgb
```

```
Path: FLIGHT/data/hughes/tape2/hier/*.z
```

Location	Upstate New York
Terrain	8000x8000
Pixel	24 bits RGB color
File format	nX x nY pixels bytes for R nX x nY pixels bytes for G nX x nY pixels bytes for B
Elev	16 bits
Grain size	4.00000 feet/pixel 0.12500 feet/pixel

1.3.2 Format of Daniel Cohen Data set

Terrain A

```
Path: FLIGHT/data/dan/right.h
```

```
Path: FLIGHT/data/dan/imetrix.dtm
```

Location	Not Known
Terrain	1024x1024
Pixel	8 bits gray color
File format	nX x nY pixels bytes gray
Elev	16 bits
Grain size	Not available

Terrain B

```
\begin{verbatim}
```

Path: FLIGHT/data/dan2/hermon.arlrgb

Path: FLIGHT/data/dan2/hermon.dtm

Location	Not Known
Terrain	512x512
Pixel	24 bits RGB color
File format	nX x nY pixels bytes gray
Elev	16 bits
Grain size	Not available

1.3.3 Format of Valley data set

Path: FLIGHT/data/csilva/fly_dted.rgb_rrr.raw

Path: FLIGHT/data/csilva/fly_phot.rgb_rrr.raw

Path: FLIGHT/data/csilva/fly_phot.rgb_ggg.raw

Path: FLIGHT/data/csilva/fly_phot.rgb_bbb.raw

Location	Not Known
Terrain	512x512
Pixel	24 bits RGB color
Pixel File format	SGI RGB Format
Elev	8 bits
Elev File format	SGI RGB Format
Grain size	Not available

1.3.4 Format of LA data set

Path: FLIGHT/data/la/la_elev.rgb_rrr.raw

Path: FLIGHT/data/la/la_color.rgb_rrr.raw

Path: FLIGHT/data/la/la_color.rgb_ggg.raw

Path: FLIGHT/data/la/la_color.rgb_bbb.raw

Location	LA beach
Terrain	693x468
Pixel	24 bits RGB color
Pixel File format	SGI RGB Format
Elev	8 bits
Elev File format	SGI RGB Format
Grain size	Not available

1.4 Terrain Modeling

1.4.1 Input of Terrain data

The idea here is to provide the mechanisms necessary to support several terrain formats using standard Utilities. The *xv* Utility makes it possible to generate an intermediate format from components which may have been written in different formats and which may be generated on different machines. The SGI RGB format has been used as standard intermediate format in our work.

The source for component generation is:

```
FLIGHT/users/VolVis.2.1f/Utilities/terrain_format/src/terrain_format.c
```

The executable is:

```
FLIGHT/users/VolVis.2.1f/bin/SGI6_OGL/terrain_format
```

```
RGB_R_G_B()
{
  Open input image file
  Open 3 output files for making texture
  For each row of image
    getrow(red); putrow(red);
    getrow(green); putrow(green);
    getrow(blue); putrow(blue);
  Open elevation image file
  For each row of elevation
    getrow(elev); putrow(elev);
}
```

1.4.2 Terrain Tiling

The source for component generation is:

```
FLIGHT/users/VolVis.2.1f/Utilities/
    terrain_voxelize/src/terrain_voxelize.c
```

The executable is:

```
FLIGHT/users/VolVis.2.1f/bin/SGI6_OGL/terrain_voxelize
```

```

xtiles=1;
ytiles=1;
deltax = numColumns/xtiles;
deltay = numRows/ytiles;

for(i = 0; i < xtiles; i++){
    xorigin = i * deltax;
    for(j = 0; j < ytiles; j++){
        yorigin = j * deltay;
        generate_volume_texture(for tile i & j,255,raw_vol);
        array_to_slc(raw_vol, SLC_file);
        if(RGB > 0){
            generate_volume_texture(for tile i&j, imgR, volR);
            generate_volume_texture(for tile i&j, imgG, volG);
            generate_volume_texture(for tile i&j, imgB, volB);
        }
        array_to_tex(volR, volG, volB, TEX_file);
    }
}

```

1.4.3 Terrain Texture Generation

Texture mapping is a technique in computer graphics that associates more than one color (a texture) with an object. We derive the expression of the texture map for an affine relation between coordinates of these surfaces, and show efficient algorithms that implement digital filters to avoid aliasing in the resampling process.

```

generate_volume_texture( tileij,inimage,outVolume)
{
For each pixel (i,j) in this tile
    height = elev(i,j);
    for(each voxel in the height column){
        if(slc volume true) val = 255;
        else if( tex volume true)val = image(i,j);
    }
}

```

```
        if(SPLATTING is True)
            splat_object(i, y, j, val, raw_volume);
        else raw_volume(i,y,j) = val;
    }
}
```

1.4.4 Loading Terrain Volume in VolVis

The SLC files, their corresponding textures and transformations (translation, scaling and rotation) are specified in the scene graph. This avoids the interactive placement of each object in the scene.

1.4.5 Rendering Terrain Volume in VolVis

Display of a scene surrounding a view point, is done using the existing features of VolVis.

1.4.6 Utilities

Several utilities have been developed as part of this project. A splatting function using a Gaussian footprint has been implemented. The tile generation needs to be stored in SLC and TEX format for rendering using VolVis. This has been done using two functions *array_to_tex* and *array_to_slc*. The advantage of this function is that it avoids a disk write as raw volume, disk read as raw by *raw2slc*. The other utilities are tri-linear interpolation between volumes and view frustum culling of volumes.

The source for component generation is:

```
FLIGHT/users/VolVis.2.1f/lib/flight/src/*.c
```

The library is located in:

```
FLIGHT/users/VolVis.2.1f/lib/flight/lib/SGI6_libflight.a
```

1.5 Implementation and Results

The following systems have been implemented for the project:

- For an input terrain with elevation and rgb color image the output into individual components of r, g, b, elev are obtained.
- The individual r, g, b, elev are then input to the slice and texture modeling system. The output is a set of SLC and tex volume files. A SLC and tex file is generated for each tile of the volume. the options available for this module includes the number of tiles along X and Y.
- The above operation has been accelerated by using multiple CPUs.
- LOD volume generation from original Terrain as sub-terrains. An LOD factor needs to be specified to generate a low resolution SLC and TEX, if required.
- Input the SLC and TEX volume files to VolVis for rendering the terrain. Parallel code has been implemented to accelerate rendering. This implementation currently can use all the 16 CPUs available on the system. Changes to VolVis code includes:
 - Loading multiple volumes with transformations and corresponding textures.
`FLIGHT/users/VolVis.2.1f/src/C_SRC/C_io.c`
 - Rendering (ray-casting) done in parallel
`FLIGHT/users/VolVis.2.1f/lib/render/src/C_SRC/C_cast_rays.c`
- Movie of sample terrain: A sequence of images are available as a Quicktime movie file to demonstrate the results obtained.
- Movie of LA terrain: This movie has 16 frames and is also available as a Quicktime movie file.

1.6 Working Model and Results

The inputs terrain is first converted into individual RGB and elevation components. The elevation can be a byte, short or float. Currently 8 bit color components are supported.

The next step is to convert 4 volume arrays. VolVis requires a SLC file and its corresponding RGB TEX file for texture mapped volume rendering. For future enhancement however the TEX file is sufficient so we don't need the SLC file, which saves 1/4 of the CPU memory.

The first volume array is an SLC file which contains 255 for all voxels under the terrain. This volume is stored as an SLC file. The *array_to_slc()* function is used to directly convert an array of bytes into a SLC file. An alternative way is to store the raw file and then convert it into SLC using the *raw2slc* utility in VolVis.

The R, G and B volume arrays contain the individual components for all voxels in the terrain. This is then stored as a single TEX file. The *array_to_tex()* function is used to directly convert three arrays of RGB bytes into a TEX file. An alternative way is to store the RGB raw files and then convert into TEX using the *raw2tex* utility in VolVis.

A rectangular tiling scheme has been implemented. This helps to generate a NxM tiles. The advantage here is only one tile needs to be voxelized at any point of time. So large super-volumes can be used during voxelization and splatted into a low-resolution volume for storage and rendering.

Level of Detail (LOD) has been achieved by voxelization by appropriately selecting pixels along each dimension of the terrain. By selecting every other pixel we generate a volume with less detail. This new volume has a resolution $1/2$ in each dimension. However averaging or filtering the original resolution gives better results.

The next stage is to tile these volumes to generate a volume rendered image of the terrain. This has been done using VolVis.

Chapter 2

Parallel Volume Graphics Algorithms

The shared-memory multiprocessor system recently has attracted a lot of attention because of its low cost and wide availability to the graphics researchers. In this paper, we show practical parallel solutions to the following volume graphics[7] problems: volume texture modeling, volume texture rendering, and voxelization for unstructured grid rendering. All these solutions process 2D and 3D arrays and employ P processors on the shared-memory system. We present results for parallel execution time, parallel memory requirement and overheads for the three algorithms described in this work. We also present results of performance prediction based on both analytical and experimental results.

2.1 Introduction

Many efficient parallel graphics algorithms have been proposed in the literature for the shared-memory class of machines. These include algorithms for: volume splatting, volume resampling, shear-warp, etc. We refer to this group of parallel machines having up to 100 nodes as Class I systems.

Class II systems from 100 to 1000 processors have been developed commercially. Class III systems have more than 1000 processors. Several graphics algorithms have been developed and reported in literature for Class II and Class III systems. The emphasis in Class II and III has been to identify parallelism that could effectively utilize large number of processors. These algorithms, eventhough have linear and super linear speedups, are of no practical use today due to its limited availability.

The motivation of this work is to:

- demonstrate practical parallel algorithms for volume graphics on shared-memory systems.
- identify parallel algorithms that can be effectively used by a large spectrum of users who have access to two or more processors.
- exploit the execution characteristics of the powerful processor in each node that dramatically differ from the Class II and III systems.
- explore efficient partitioning schemes on Class I machines which do not have to worry about mapping of data for efficient communication.
- motivate graphics researchers to use practical parallelism in their day-to-day research.
- to provide a performance measure which could be used by graphics researchers and users to analyze existing sequential algorithms for possible parallel implementation.

Hence the emphasis of this work is to provide parallel algorithms for the relatively widespread and easy to use Class I machines. In this paper we present new results of parallel performance estimation on Shared-Memory Multiprocessor systems. The following algorithms have been implemented and the performance has been experimentally verified on a 16 CPU SGI Power Challenge:

1. Volume Texture Modeling (VTM) of Terrain
2. Volume Texture Rendering (VTR) of Terrain
3. Coherent Voxelization Algorithm (CVA) for Unstructured Grids

2.2 Parallel Rendering Algorithms

2.2.1 Volume Texture Modeling of Terrain

This is a new system for visual simulation of a volumetric terrain model. The input is the elevation data and corresponding aerial photographs or satellite images of a terrain. The output is a volumetric terrain model of the terrain. The sequential algorithm for Volume Texture Mapping is given in Fig. 2.1.

```

VTM_sequential()
{ for (each scanline in the image)
  { for (each pixel in scanline)
    { vox_column(pixel) = texture_map(elev,color); }
    write_to_vol_buffer(vox_column);
  }
  for (each voxel in intermediate volume)
    { tex_volume[voxel] = splat_kernel(); }
}

```

Figure 2.1: VTM_sequential

A voxel column is computed for each pixel in the terrain image. Each column can be computed independently. A simple yet inexpensive way of parallelization is to partition the image into scanlines and assign them to processors. Since there is no communication there is no need for mapping of adjacent scanlines to adjacent processors. The parallel texture mapping algorithm is given in Fig. 2.2.

```

VTM_parallel()
{ for (each scanline in this processor)
  { for (each pixel in scanline)
    { vox_column(pixel) = texture_map(elev,color); }
    write_to_vol_buffer(vox_column);
  }
  for (each voxel in this processor)
    { tex_volume[voxel] = splat_kernel(); }
}

```

Figure 2.2: VTM_parallel

For implementation on a shared-memory multiprocessor, the input elevation and image is loaded in shared-memory and the output volume-buffer is stored in shared-memory.

2.2.2 Volume Texture Rendering of Terrain

The VolVis system has been extended for parallel ray-casting of a Volume Texture Model on a shared-memory system. The sequential algorithm of VTR in VolVis is given in Fig. 2.3.

A ray is cast for each pixel in the final image. Each pixel can be computed independently. The image is partitioned into scanlines and assigned to processors. There is no communication among processors. The parallel raycasting algorithm is given in Fig. 2.4.

For implementation on a shared-memory multiprocessor, the input volume-buffer is loaded

```

VTR_sequential()
{ for (each scanline in the image)
  { for (each pixel in scanline)
    { image(pixel) = cast_ray(pixel); }
  }
}

```

Figure 2.3: VTR_sequential

```

VTR_parallel()
{ for (each scanline in this processor)
  { for (each pixel in scanline)
    { image(pixel) = cast_ray(pixel); }
  }
}

```

Figure 2.4: VTR_parallel

in shared-memory and the output image is stored in shared-memory.

2.2.3 Voxelization for Unstructured Grid Rendering

Voxelization has been proposed as an efficient method for rendering large unstructured grids [14][13]. The motivation of including unstructured grids in voxel form in the flight simulation work is the potential of including amorphous phenomena such as cloud, smoke, fire, fog, etc., which is represented in an unstructured way. The goal here is to achieve improved performance of parallel voxelization-based volume rendering of unstructured grids. The unstructured grid is decomposed into convex cells. The algorithm for sequential voxelization is given in Fig. 2.5.

```

VUG_sequential()
{ for (each cell in the grid)
  { for (each polygon of cell)
    { form_buffers(polygon); }
    if (polygon is front facing)
    { scan_convert(to_frontbuffer); }
    else if (polygon is back facing)
    { scan_convert(to_backbuffer); }
    voxelize_between_two_buffers();
  }
}

```

Figure 2.5: VUG_sequential

The *voxelize_between_two_buffers()* procedure interpolates the scalar values for all voxels within the cell.

In the coarse grain approach, one cell is assigned to a processor and voxelization is done on the individual cell. All other computations for the cell are done on the same processor. This helps the concurrent execution of cells in different processors because the cells do not intersect each other. The algorithm for parallel voxelization is given in Fig. 2.6.

```

VUG_sequential()
{ for (each cell assigned to this processor)
  { for (each polygon of cell)
    { form_buffers(polygon); }
    if (polygon is front facing)
      { scan_convert(to_frontbuffer); }
    else if (polygon is back facing)
      { scan_convert(to_backbuffer); }
    voxelize_between_two_buffers();
  }
}

```

Figure 2.6: VUG_parallel

In the parallel version, all the cells in the grid are not voxelized on the same processor. Each processor voxelizes only those cells which are assigned to it.

For implementation on a shared-memory multiprocessor,

- The input unstructured grid is loaded in shared-memory.
- The output volume-buffer is stored in shared-memory.
- The frame-buffers and z-buffers for each cell are local to the processor.

2.2.4 Related Work

Parallel implementations on Class I systems have been discussed in [4] [17], [11], [9] and [10]. In this paper we limit our discussion to parallel implementations on small parallel machines.

2.2.5 Design Goals

The major issues and our goals in parallel algorithms for graphics are:

- **Data Duplication:** The input data is usually a grid which consists of voxels in a regular grid or cells in an unstructured grid. Data duplication happens when more than one copy of part or all of the grid occurs. However such a problem is not normally encountered on shared-memory systems. Our design goal is not to duplicate the data.
- **Synchronization Requirement:** Most algorithms forces the synchronization among processors. Our design goal is to avoid the need for synchronization among the processors.
- **Critical Regions:** When multiple processors compute a scene in parallel, all processors cannot write concurrently to the graphics subsystem. This forces the use of spin-locks. Our objective is to avoid busy waiting of processors on a spin-lock.
- **Idle Processors:** This situation arises when there is no proper load balancing. Our aim is to avoid additional computation required for load balancing. and to provide good load balancing without much computation for load balancing.
- **Rigid Data Partitioning/Task Allocation:** Existing methods require adjoining cells and adjacent scanplanes to reside in the same processor. Our goal is to eliminate such dependencies.
- **Interprocessor Communication:** This is another major overhead in several parallel volume rendering algorithms. Our aim is to reduce and, if possible eliminate this overhead.

2.3 Implementation and Results

We have used a 16 processor MIPS R10000 (194 MHz) based Power Challenge iR graphics system with 3 GB CPU memory.

The experimental results for VTM, VTR and CVA are shown in Table 2.1-2.3.

2.4 Parallel Performance Estimation

Several methods have been proposed in literature for analyzing the performance of parallel algorithms. Two important schemes relevant for graphics algorithms are Green [5] and Tom

Table 2.1: Parallel Execution time of VTM (in seconds).

Terrain Model	No. of Processors					
	1	2	4	8	12	16
TERR1	16.73	11.12	5.90	3.13	2.19	1.82
	89.95	46.05	25.27	14.28	10.36	8.95
	389.39	196.61	100.75	54.78	39.81	36.11
TERR2	16.66	11.12	5.88	3.13	2.18	1.84
	89.71	45.91	25.13	14.32	10.34	8.83
	388.65	196.18	100.29	55.55	39.83	35.98

Table 2.2: Parallel Execution time of VTR (in seconds).

Texture Render	No. of Processors					
	1	2	4	8	12	16
256x256	4.74	2.71	1.54	0.89	0.73	0.61
512x512	18.02	9.94	5.53	3.07	2.43	2.30
1024x1024	79.30	42.89	22.96	13.03	10.53	9.77
256x256	8.41	4.95	2.64	1.44	1.17	1.01
512x512	27.49	17.18	9.44	5.10	3.88	3.15
1024x1024	92.38	55.99	30.95	15.43	12.44	11.11

Table 2.3: Parallel Execution time of Voxelization of Grids (in seconds).

Grid (Cells)	No. of Processors					
	1	2	4	8	12	16
BFIN	3.78	2.44	1.49	1.07	0.79	0.57
	14.23	10.42	7.32	4.54	3.75	2.97
	94.86	59.48	32.29	17.58	13.79	9.85
POST	5.86	4.42	2.43	2.21	2.16	2.11
	14.20	8.75	5.28	3.53	3.15	3.13
	55.45	31.88	18.39	11.77	9.37	9.10

[2]. The two schemes have been extended to analyze the performance and scalability of parallel volume graphics algorithms on shared address space systems. The Green's method helps in analyzing the peak performance and the peak scalability of the algorithm Tom's method helps in the prediction of performance for more processors.

2.4.1 Model of Performance - I: Green's Method

The objective here is to estimate the scalability of the three parallel volume graphics algorithms when more CPUs are added. Let

- p - total number of processors
- R_1 - the time for execution using 1 cpu
- α - the overhead in adding a CPU

The average rate of processing R in a system with p processors is

$$R(p) = R_1 - \alpha(p - 1)$$

Total performance of the system is

$$T(p) = pR(p)$$

To estimate the speedup s , let

- t_1 - time for single processor
- t_p - time for p processors

The speedup:

$$\begin{aligned} s(p) &= t_1/t_p \\ &= T(p)/T(1) \\ &= pR(p)/R(1) \\ &= p(R_1 - \alpha(p - 1))/R_1 \\ &= ((1 + \alpha)/R_1)p - \alpha p^2/R_1 \end{aligned}$$

For peak performance:

$$\begin{aligned} dT/dp &= 0 \\ d/dp(p(R_1 - \alpha(p - 1))) &= 0 \text{ then} \\ R_1 - 2\alpha p + \alpha &= 0 \end{aligned}$$

The peak performance p_{max} is obtained when $dT/dp = 0$ and s_{max} the corresponding s .

Then p_{max} and s_{max} are expressed in terms of R_1 and α as follows:

$$\begin{aligned} p_{max} &= (1/2)(1 + R_1/\alpha) \\ s_{max} &= (1/4)(1 + R_1/\alpha)(1 + \alpha/R_1) \end{aligned}$$

Table 2.4: Estimation of parallel performance (Green's method)

Algorithm	CPU R_1	R2	α	p_{max}	s_{max}
VTM	388.65	196.18	1.85	105	52.74
VTR	92.38	55.99	9.80	5	2.76
CVA	55.45	31.88	4.15	7	3.85

The peak performance p_{max} and scalability s_{max} are estimated for all the three volume graphics algorithms and results are shown in Table 2.4.

2.4.2 Model of Performance - II: Tom's Method

This method also analyses the scalability of a parallel algorithm by estimating the overhead involved in parallelization due to the addition of more CPUs. This method has been used to predict the performance when more CPUs are added for parallel execution.

Performance Prediction

The overheads during the different stages of execution are:

- t_{trans} - overhead in input grid transformation.
- t_{split} - overhead in splitting grid into sub-grids.
- t_{init} - overhead in volume buffer initialization.
- t_{exec} - overhead in execution of cells.
- t_{store} - the overhead in storing voxels.

The total time is then expressed as:

$$T_{parallel} = Cn/p + t_{exec} + t_{store} + t_{trans} + t_{split} + t_{init}$$

where n is the number of cells, p is the number of processors, and C is a constant which is dependent on the data.

The overheads may be either due to the extra computation required due to parallelization, due to the contention for shared bus, message communication latency, synchronization requirement or due to variations in workload.

In the shared address space scheme, the computation is done by assigning the workload to processors. The different buffers reside in shared-address space. The overheads are t_{exec} and

t_{store} only. The execution stage requires access of local buffers from shared memory which may lead to contention. The t_{exec} is also due to the shared access for the buffers. The other overhead arises during the storage of output from several processors into the volume buffer in shared-address space. This could be easily isolated. Let us take a scene with uniform workload. By adding an additional processor one half of the workload is computed on each processor. The time taken is more than the execution of just the first one-half by a single processor. This overhead incurred is due to t_{exec} and t_{store} . This is used to predict the performance of the system when multiple CPUs are added for computation.

$$t_{overhead} = t_{exec} + t_{store}$$

The overhead is assumed to be constant since we assume that the design of the shared-memory system provides the necessary bandwidth in the shared-bus and hence does not lead to contention.

Performance Prediction for VTM

For the VTM, when it is texture mapped into a volume buffer of size 570x66x695 the overhead for adding an additional CPU from the experiment is 1.855 seconds. The estimation expression assumes a constant overhead of 1.855 seconds.

In the case of VTM, when $p = 1$, for a single CPU the time taken for the Terr1 terrain data is 388.65 seconds.

$$\begin{aligned} Cn/p &= 388.65 \\ Cn &= 388.65 \end{aligned}$$

For $p = 2$, the $T_{overhead}$ from experiments is computed as follows:

$$\begin{aligned} T_2 &= Cn/p + T_{overhead} \\ T_2 &= 196.18 \\ T_{overhead} &= T_2 - Cn/2 \\ &= 196.18 - 388.65/2 \\ &= 1.855 \end{aligned}$$

Performance Prediction for VTR

For the VTR, when it is rendered into a volume buffer of size 570x66x695 the overhead for adding an additional CPU from the experiment is 9.8 seconds. The estimation expression assumes a constant overhead of 9.8 seconds.

Table 2.5: Estimation of parallel performance (Tom's method)

Grid (Cells)	No. of Processors					
	4	8	12	16	32	64
VTM	99.01	50.43	34.24	26.14	14.00	7.92
(Expt)	100.29	54.78	39.81	36.11		
VTR	32.89	21.34	17.49	15.57	12.68	11.24
(Expt)	30.95	15.43	12.44	11.11		
CVA	18.01	11.08	8.77	7.61	5.88	5.01
(Expt)	18.39	11.77	9.37	9.10		

Performance Prediction for CVA

For the CVA, when it is voxelized into a volume buffer of size 1000x1000x160 the overhead for adding an additional CPU from the experiment is 4.15 seconds. The estimation expression assumes a constant overhead of 4.15 seconds.

The above assumption holds good since there is not much variation in overhead as obtained from the experiments when more processors are added. Predicted time for p processors for the VTM, VTR and CVA are shown in Table. 2.5.

Several data sets have been tested for the three algorithms and results are shown in Fig. 4-9.

2.5 Source Code

The source code for VTM and VTR are located in:

```
FLIGHT = /home/fs8/flight_sim/
FLIGHT/users/VolVis.2.1f/Utilities/terrain_format/src/terrain_voxelize.c
FLIGHT/users/VolVis.2.1f/lib/render/src/C_SRC/C_cast_rays.c
```

2.6 Summary

The performance of these algorithms gets significantly degraded when multiple CPUs are used for execution. One possible reason is due to the number of memory references that arises frequently due to load and store operations which is not a desirable aspect for superscalar

processors.

Chapter 3

Conclusion

Several improvements are possible over the existing implementation. This section explores a number of problems that we have encountered and possible solutions. The extra disk read and write during storage of RAW and TEX files has been solved using the utility of directly storing into SLC and TEX after voxelization of terrain. The surface blending of multiple tiles and multiresolution tiles still needs improvement. By setting a control variable only a thin sheet of voxels needs to be stored. This is done by computing the minimum elevation from neighboring 3-8 voxels including the current voxel.

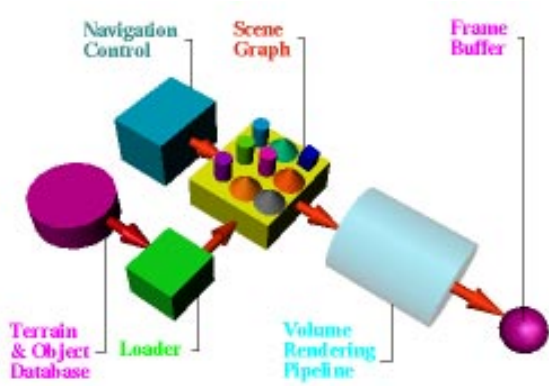


Fig 1.1: *Block diagram of Visual Flythrough*

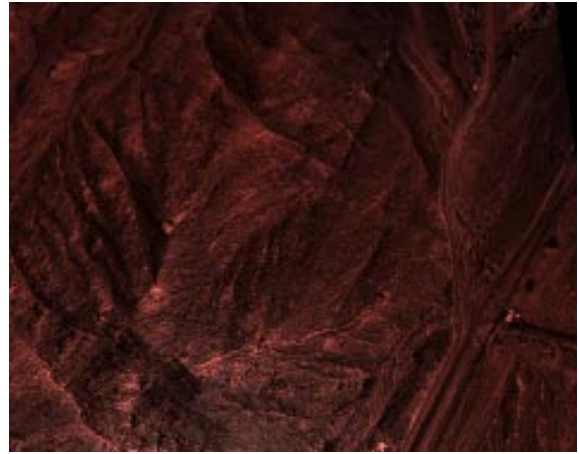


Fig 1.2: *Volume Texture with one component*

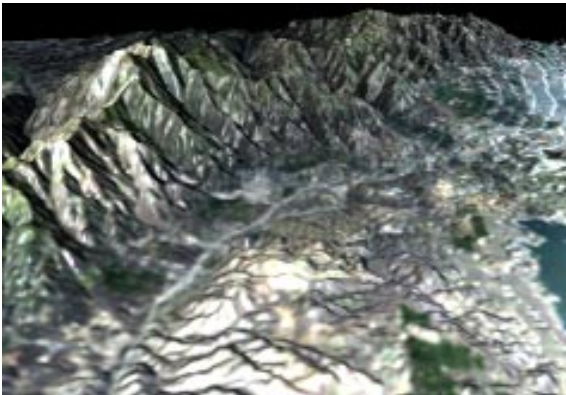


Fig 1.3: *Perspective ray-casting of terrain*



Fig 1.4: *Parallel ray-casting of terrain*

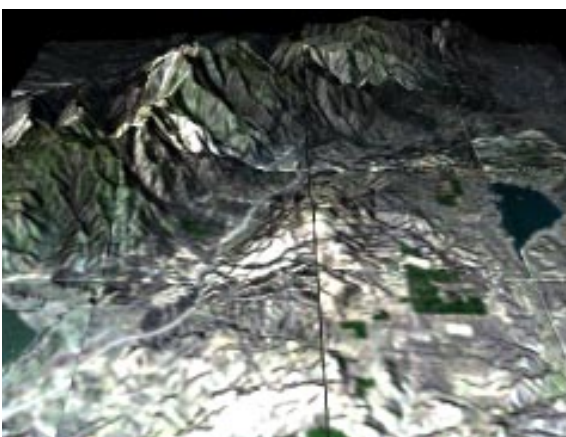


Fig 1.5: *A tiled terrain model using VolVis*



Fig 1.6: *A LA terrain with voxelized benches*

Bibliography

- [1] Daniel Cohen, Eran Rich, Uri Lerner, and Victor Shenkar. A real-time photo-realistic visual flythrough. *IEEE TVCG*, 2(3):255–265, Sep 1996.
- [2] Thomas W. Crockett and Tobias Orloff. Parallel polygon rendering for message passing architectures. *IEEE Parallel and Distributed Technology*, 2(2):17–28, Summer 1994.
- [3] Leila De Floriani, Paola Marzano, and Puppo E. Multiresolution models of topographic surface description. *The Visual Computer*, 7(7):317–345, Oct 1996.
- [4] Christopher Giertsen and Johnny Petersen. Parallel volume rendering on a network of workstations. *IEEE CG&A*, 13(6):16–23, Nov 1993.
- [5] Stuart Green. *Parallel Processing for Computer Graphics*. The MIT Press, Cambridge, MA, 1991.
- [6] Taosong He, L. Hong, Arie E. Kaufman, A. Varshney, and S. Wang. Voxel-based object simplification. *Proc. of Visualization '95*, Oct 1995.
- [7] Arie E. Kaufman, Daniel Cohen, and Roni Yagel. Volume graphics. *IEEE Computer*, 26(7):51–64, Jul 1993.
- [8] David Koller, Peter Lindstrom, William Ribarsky, Larry F. Hodges, Nick Faust, and Gregory Turner. Virtual gis: A real-time 3d geographics information system visualization. *Proc. of Visualization '95*, pages 94–100, Oct 1995.
- [9] Philippe Lacroute. Real-time volume rendering on shared memory multiprocessors using the shear-warp factorization. *Proc. of Parallel Rendering Symposium 1995, ACM Press*, pages 15–22, Oct 1995.

- [10] Philippe Lacroute. Analysis of a parallel volume rendering system based on the shear-warp factorization. *IEEE Trans. on Visualization and Graphics*, 2(3):218–231, Sep 1996.
- [11] Bruce Lucas. A scientific visualization renderer. *Proc. of Visualization '92*, pages 227–234, Oct 1992.
- [12] R. Gatti Oliver Staadt, M. Gross. Fast multiresolution surface meshing. *Proc. of Visualization '95*, Oct 1995.
- [13] C. E. Prakash and S. Manohar. Shared-memory multiprocessor implementation of voxelization for volume visualization. In *HPC for Computer Graphics and Visualization*, M. Chen, P. Townsend and J. A. Vince (Eds), Springer, London, pages 135–145, 1995.
- [14] C. E. Prakash and S. Manohar. Volume rendering of unstructured grids: A voxelization approach. *Computers and Graphics*, 19(5):711–726, Sep/Oct 1995.
- [15] Claudio T. Silva, Koseph S., B. Mitchell, and Arie E. Kaufman. Automatic generation of triangular irregular networks using greedy cuts. *Proc. of Visualization '95, IEEE CS Press*, pages 196–203, Oct 1994.
- [16] M. Suter and D. Nuesch. Automated generation of visual simulation databases using remote sensing and gis visualization. *Proc. of Visualization '95*, pages 86–93, Oct 1995.
- [17] Peter L. Williams. Interactive direct volume rendering of curvilinear and unstructured data. *PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign*, 1992.
- [18] John R. Wright and Julia C. L. Hsieh. A voxel-based, forward projection algorithm for rendering surface and volumetric data. *Proc. of Visualization '92*, pages 340–348, Oct 1992.

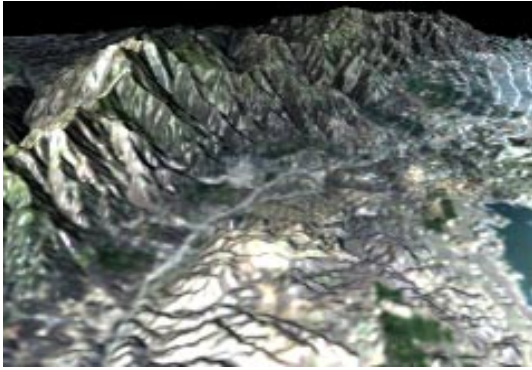


Fig 4: *Volume Texture rendered using VolVis*

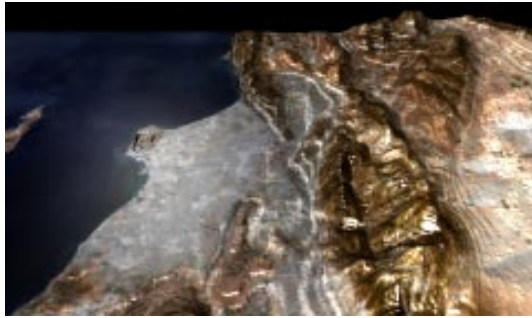


Fig 5: *Volume Texture rendered using VolVis*

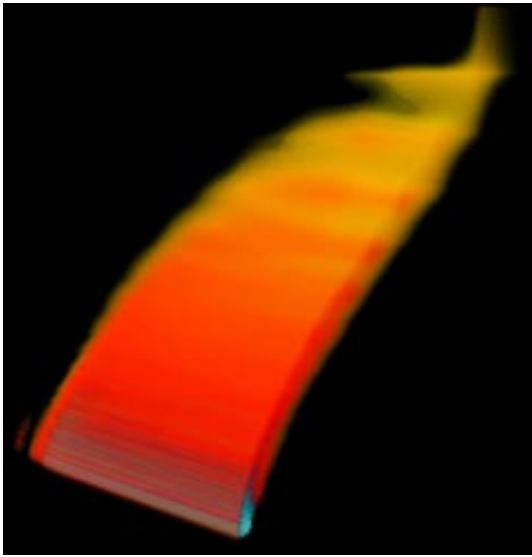


Fig 6: *Flow over bluntfin rendered using BoB*

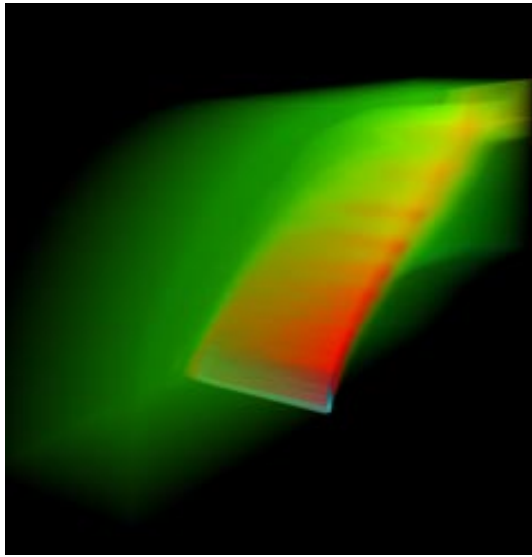


Fig 7: *A sub-volume of bluntfin flow*

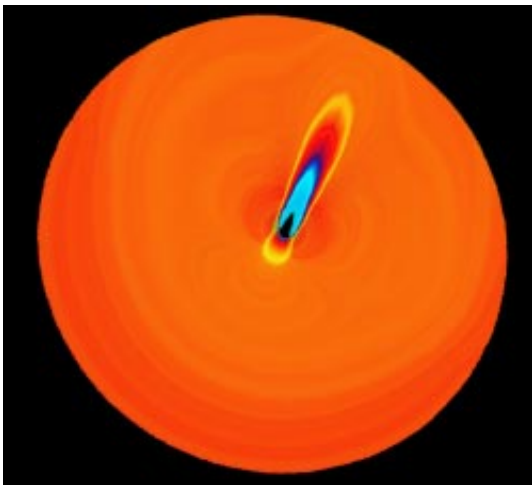


Fig 8: *Oxygen Post Data rendered using BoB*

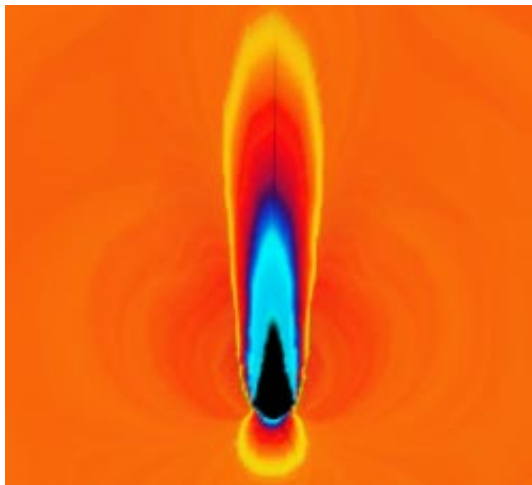


Fig 9: *A sub-volume of Oxygen Post Data*