FFI RAPPORT

VOLUVIZ 1.3 REPORT

ENGDAHL Geir Kokkvoll, HELGELAND Anders

FFI/RAPPORT-2005/02506

VOLUVIZ 1.3 REPORT

ENGDAHL Geir Kokkvoll, HELGELAND Anders

FFI/RAPPORT-2005/02506

FORSVARETS FORSKNINGSINSTITUTT
Norwegian Defence Research Establishment
P O Box 25, NO-2027 Kjeller, Norway

FORSVARETS FORSKNINGSINSTITUTT(FFI)
Norwegian Defence Research Establishment

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE
(when data entered)

P O BOX 25
NO-2027 KJELLER, NORWAY

REPORT DOCUMENTATION PAGE

1) **PUBL/REPORT NUMBER** FFI/RAPPORT-05/2506	2) **SECURITY CLASSIFICATION** UNCLASSIFIED	3) **NUMBER OF PAGES** 49
1a) **PROJECT REFERENCE** FFI-V/1016/	2a) **DECLASSIFICATION/DOWNGRADING SCHEDULE** -	

4) **TITLE**
VOLUVIZ 1.3 REPORT

5) **NAMES OF AUTHOR(S) IN FULL (surname first)**
ENGDAHL Geir Kokkvoll, HELGELAND Anders

6) **DISTRIBUTION STATEMENT**
Approved for public release. Distribution unlimited. (Offentlig tilgjengelig)

7) **INDEXING TERMS**

	IN ENGLISH		**IN NORWEGIAN**
a)	VoluViz	a)	VoluViz
b)	Visualization	b)	Visualisering
c)	Volume rendering	c)	Volumrendering
d)	GPU/Shader programming	d)	GPU/Shader programmering
e)	Multi-pipe programming	e)	Multi-pipe programmering
f)	SGI Multipipe SDK	f)	SGI Multipipe SDK

THESAURUS REFERENCE:

8) **ABSTRACT**

VoluViz is a direct volume rendering application based on SGI's Volumizer 2 and OpenGL. It is designed for interactive viewing of three-dimensional volumetric data.
The report is in three parts. The first part gives an overview of the latest features added in VoluViz. This part treats common usage patters as well as discussing some implementation issues. The second part is a short introduction to shader programming with VoluViz. While the last part documents the changes made in order to make a multi-pipe version of VoluViz.

9) **DATE**	**AUTHORIZED BY**	**POSITION**
19 August 2005	This page only Bjarne Haugstad	Director of Research

ISBN 82-464-0963-8

UNCLASSIFIED

FFI-B-22-1982

SECURITY CLASSIFICATION OF THIS PAGE
(when data entered)

PREFACE

The report is divided into three parts. The first part gives an overview of the latest features added in VoluViz. This part treats common usage patters as well as discussing some implementation issues.

The second part is a short introduction to shader programming with VoluViz. While the last part documents the changes made in order to make a multi-pipe version of VoluViz using SGI's Multipipe Software Development Kit (SDK).

The appendix includes information on how to install VoluViz as well as some details related to shader- and multi-pipe programming.

CONTENTS

page

VOLUVIZ 1.3 REPORT

1 VOLUVIZ 1.3

1.1 Introduction

VoluViz is a volume renderer based on SGI's OpenGL Volumizer 2 (1), (2) and OpenGL. It has been developed from 2001 to 2005 by Anders Helgeland, Trond Gaarder (2001-2004) and Geir Kokkvoll Engdahl (2005) at the Norwegian Defense Research Establishment. For an introduction to the basic capabilities of VoluViz, we refer to Gaarder and Helgeland (4).

Volume rendering addresses an important challenge to scientists who try to understand large 3D-dimensional datasets, often from numerical experiments. Choosing the right visualization can turn gigabytes of numbers into easily comprehendable images and animations.

Due to the enormous amounts of raw data and the goal of real-time interactivity, volume rendering is extremely resource demanding. However, recent developments in graphics hardware have made real-time rendering of large datasets feasible.

VoluViz 1.3 takes advantage of commonly available modern graphics hardware to implement a wide variety of visualization modes. It also features easy navigation through time-dependent datasets.

1.2 Multi Field Visualization

To create the best possible visualization, it is often desirable to be able to use more than one variable to compose an image. For instance, a hurricane can be visualized as shown in figure 1.1 by combining wind speed, wind direction and geographical information.

1.2.1 Fragment Programs

In order to combine several datasets into the right visualization, we have chosen to move more computation from the CPUs (traditional approach) onto the GPUs. This required us to write programs for the GPU - fragment programs.

Fragment programs are short assembly codes for execution on a graphics processor. Instead of writing assembly code directly, we chose to use the high-level shading language Cg (C for graphics), which is much more human-readable.

One fragment program has to be written for each mode of multi field visualization. For instance, three luminance textures can be considered a mode. Another mode could be one luminance texture, but with alpha values masked against another field.

In order to identify the different modes, we decided to use a system where each type of texture is

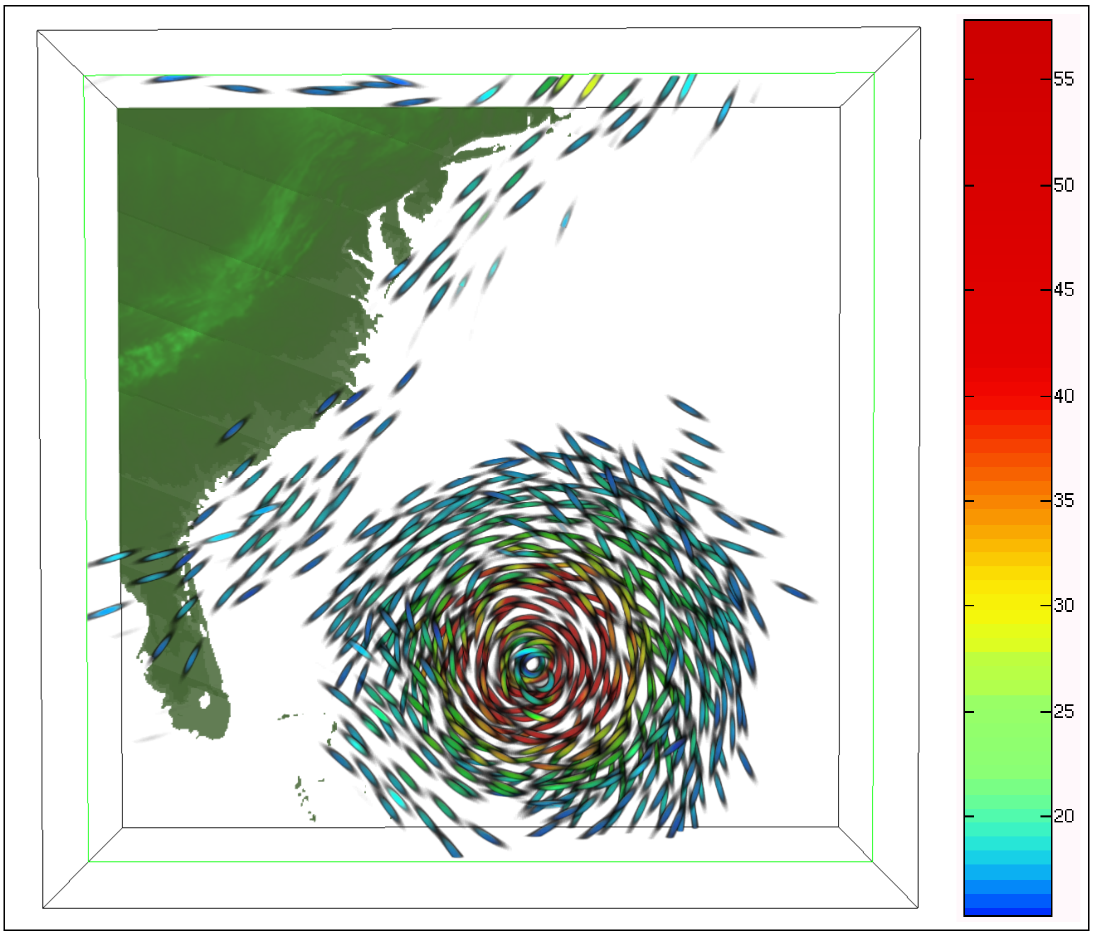


Figure 1.1 Hurricane visualization using VoluViz 1.3 and shader library.

identified by a letter. For instance, luminance got the letter L, and so a scene with three luminance textures would be identified as LLL. The scene with one luminance and one mask is represented by the string LM.

VoluViz loads the fragment program from a file with the name given by the mode string. It assumes that the fragment program is located in either

```
VOLUVIZ_HOME/shaders
```

or

```
../shaders
```

1.2.2 Choosing the right Visualization Mode

Once a dataset is loaded into memory, it appears in a list in the dataset dialog (see figure 1.2). From this dialog, the user can create different types of textures from the datasets loaded. The supported types are Alpha, Luminance and Mask.

The hardware used for testing VoluViz 1.3 did not support more than 4 volume textures and 4 color transfer functions (LUTs). This constraint may no longer be an issue with newer hardware.

1.2.2.1 Alpha

This will tell the renderer to use the values in the Alpha dataset as alpha values for the previous luminance texture. It does not make sense to use Alpha as the first texture in the scene.

1.2.2.2 Luminance

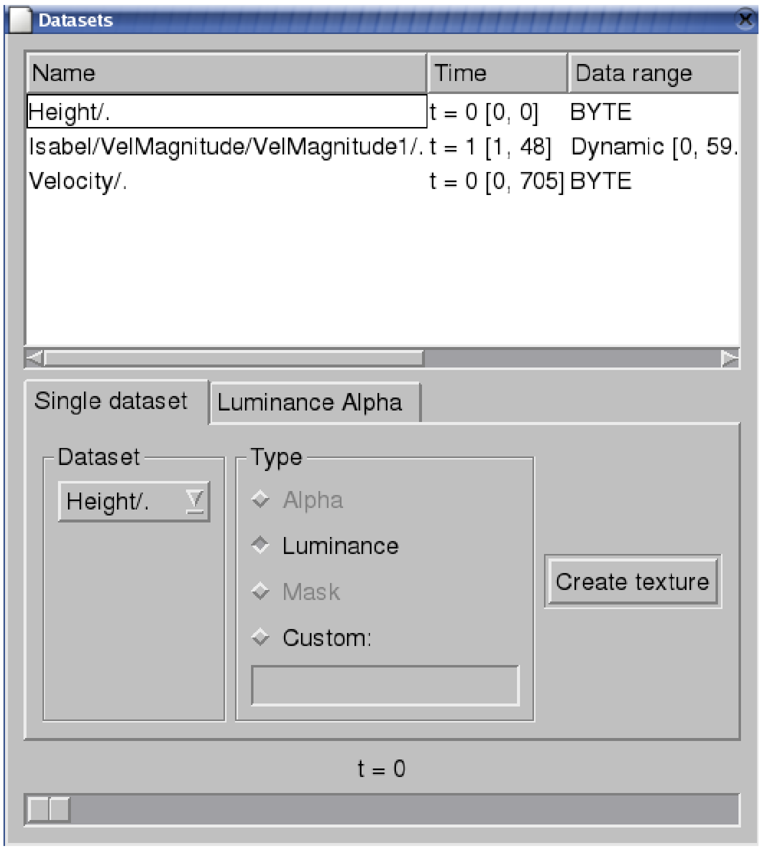
This is the standard type of texture, which reads the given dataset and uses its values to define the luminance of a new texture.

1.2.2.3 Mask

Multiplies the alpha values of all previous textures by the values in the dataset associated with the mask texture. As with alpha, it does not make sense to use a mask as the first texture in a scene.

1.2.2.4 Custom

This option is provided since there are other operations that users might find interesting. New fragment programs have to be written and placed in the shaders directory for this to work. See figure 1.3 for an example of this.



Name	Time	Data range
Height/.	t = 0 [0, 0]	BYTE
Isabel/VelMagnitude/VelMagnitude1/.	t = 1 [1, 48]	Dynamic [0, 59.
Velocity/.	t = 0 [0, 705]	BYTE

Dataset: Height/.

Type: Alpha, Luminance, Mask, Custom:

t = 0

Figure 1.2   *Dataset dialog.*

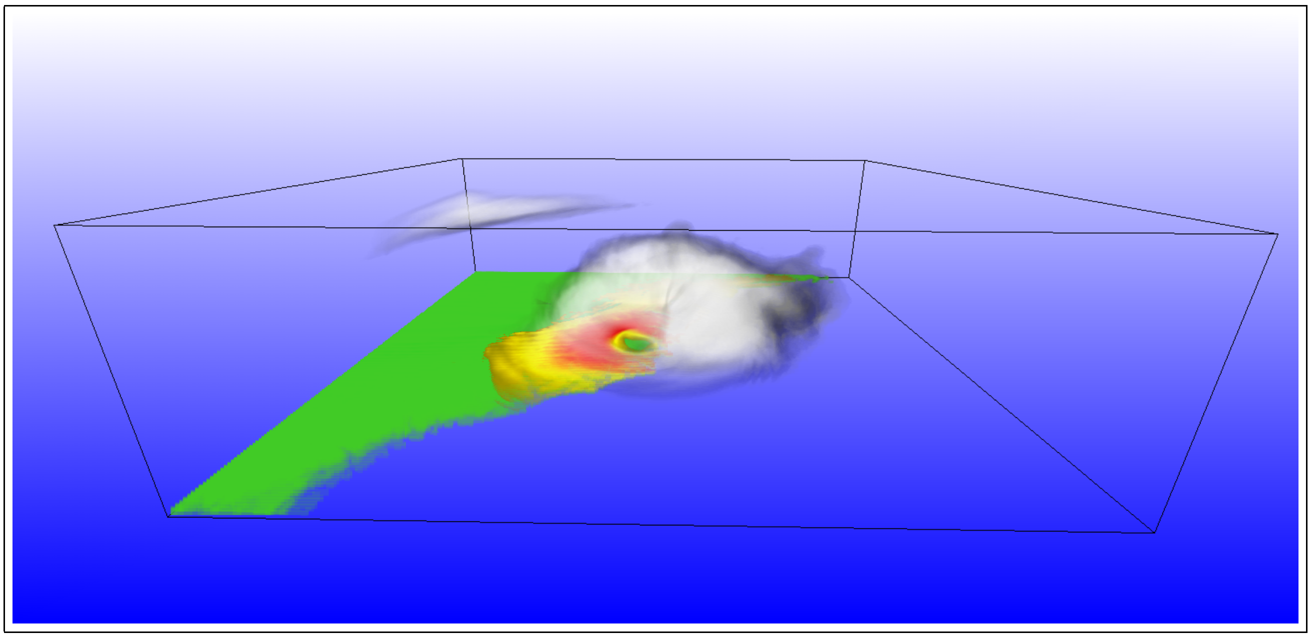


Figure 1.3   *Projection of the velocity magnitude above a certain threshold onto the ground (from yellow to red with increasing velocity magnitude) using a custom texture type and fragment program. The ground and the hurricane are rendered in green and white.*

1.2.2.5 Luminance Alpha

This is a special texture operation which takes two datasets as input and creates a single texture. It corresponds to the LA mode, but can be useful if the number of textures (for both volumes and LUTs) is an issue. This texture type is slower than LA for animations, since the two datasets have to be interleaved by the CPU at each time step.

1.3 Visualization of Time-Varying Fields

The new dataset dialog (see figure 1.2) is also equipped with a time slider at the bottom, in order to ease navigation through time-dependent datasets.

1.3.1 Automatic Detection of Dataset Time Range

When a dataset is loaded from an HDF5 file, VoluViz will probe the HDF5 file to see if there are more datasets with group and dataset names that indicate that they are other time versions of the same dataset. This probing will succeed if the following conditions hold:

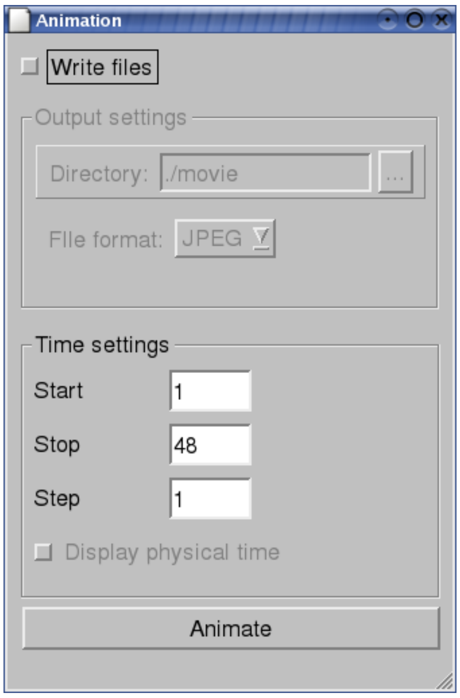
- There is a number N somewhere in either the group or dataset name.
- There exists a dataset with group or dataset name where N is replaced by 0 or 1 with 0 to 5 leading zeroes.
- There is a consecutive range of integer values N_{min} (0 or 1) to N_{max}, so that when each number is inserted for N, the corresponding group and dataset name pair is a dataset in the HDF5 file.

The probe will then have determined the start value (0 or 1) and the format (number of leading zeroes) of the dataset range. It will then use a binary search to find the last time value. If no time range can be determined, the dataset is assumed to be independent of time and is interpreted as a stand-alone dataset with $t = 0$.

1.3.2 Multiple Time Ranges

Whenever more datasets are loaded, they are assumed to have time-ranges that cover the same physical time domain, even if they have different number of time steps. A global time is calculated from the dataset with the most time steps. This global time will show on top of the time slider. If the time slider is moved to another value, VoluViz will load each dataset again at a local time step number, which is determined by the following formula:

$$local_{time} := local_{mintime} + \frac{(global_{time} - global_{mintime}) * (local_{maxtime} - local_{mintime})}{global_{maxtime} - global_{mintime}}$$



Write files

Output settings

Directory: ./movie

File format: JPEG

Time settings

Start 1

Stop 48

Step 1

Display physical time

Figure 1.4   Animation dialog.

1.3.3 Animation

Animation is accomplished by selecting **Animation** from the **Edit** menu in the main window. It will open the animation dialog (see figure 1.4). This allows the user to select the start and end global time and the time step size between each frame. The animation dialog also allows the user to specify if and where to save images from each frame.

1.3.4 Buffering

Several tasks have to be performed for each frame in an animation:

1. Read data from file.
2. Convert data to byte if necessary.
3. Create Volumizer2 scene graph. Brick volume data if necessary.
4. Render scene graph.

In order to have more interactive animations, step 1 and 2 can be done in advance and buffered. The precomputed data is then stored in an array of size equal to the number of time steps. This gives a considerable performance boost if the datasets are in floating point format and/or has to be read from disk.

Some systems will not gain any performance by using this option, since SCSI drivers are already caching the datasets. Running through an animation once will make the animation much faster the second time on such systems. However, there is a performance gain in any event if the datasets are stored as floating-point data.

To buffer all the time steps of a dataset, right-click on the dataset in the dataset dialog. Then select **Cache** from the list. This can use a considerable amount of memory depending on the number of time steps and data size. It can also take several minutes of processing time if the data has to be converted from floating-point to byte.

Always **Fix data range** or **Edit data range** (see section 1.3.5) before issuing the **Cache** command, as this cannot be done after the caching operation. To remove the cached datasets from memory, right-click on the dataset again and select **Delete cache**.

1.3.5 Data Range

Datasets are often stored as floats. Since VoluViz only works on byte data it will automatically convert float data into normalized byte data. This is done by finding the min and max value of the dataset. Since the min and max values of each time step may vary during an animation, the colors could represent different values at each frame. This can be fixed by setting a global min and max value of all the datasets in a time series. These values need not be the actual min and max values. It is done in the dataset dialog by right-clicking on a float dataset and selecting **Edit range** or **Fix range**. The latter will set the global min and max values to the min and max values of the current time step. Using a fixed data range will not only make the coloring of an animation more consistent, it will also improve performance quite a bit by removing the need to search for the min and max value of each time step.

1.4 Frames Per Second Test Utility

Selecting the **Test FPS** option from the **View** menu will run a FPS test. This will cause the renderer to render the current scene for ten seconds or more, and then display the measured frame rate in the status bar at the bottom of the main window. The test is based on the system clock and the `clock()` function in the `time.h` C library.

1.5 Known Issues / Limitations

1.5.1 Volumizer Bricking Bug

Volumizer 2.8 does not allow different texture sizes on overlapping textures if they are bricked internally. Textures larger than the texture memory (currently 256 MB) will get bricked, so VoluViz cannot currently handle large datasets of different size.

1.5.2 ATI Graphics Driver Bug

The ATI graphics driver has a bug which causes the OpenGL state required by a fragment program to be validated incorrectly. This can cause a texture to be absent from texture memory when it is needed and Volumizer freezes. A work-around is in place: Set the environment variable `VOLUMIZER_FORCE_TEXTURE_DOWNLOAD` to 1.

2 SHADER PROGRAMMING

2.1 Introduction

Recent developments in graphics hardware have turned the graphics processing units (GPUs) into small supercomputers. Over the past five years, GPU technology has advanced at an explosive pace. The rendering speed, measured in pixels per second, has been doubling approximately every six months during those five years (3). Hardware that is available to consumers at very low cost today, features parallel architecture, often as many as 16 independent pixel rendering pipelines. Each pipeline is typically capable of 4 floating-point operations in one clock cycle. This gives a GPU like NVIDIA GeForce 6800 Ultra a processing speed of about 50 Gflops compared to theoretically 12 Gflops for a modern CPU (3 GHz Pentium 4). Currently, the annual growth for GPUs is much higher than for CPUs. As this is expected to last for the next few years, the ratio in processing speed between the CPU and the GPU will only increase further.

Shader programs are executed on graphics hardware (GPUs). Generally, shader programs are invoked millions of times for each frame rendered. The GPUs can usually run 16 or more such programs in parallel, because of their architecture. One drawback is that GPUs are only efficient at computations that can be broken down into many independent sub-problems.

There are two main categories of shader programs:

2.1.0.1 Vertex Programs

work on geometric primitives, which makes them of little use to volume rendering. Vertex programs are executed on the vertex pipelines of a graphics card.

2.1.0.2 Fragment Programs

work on primitives that can be directly associated with a voxel and are able to do texture lookups. This makes fragment programs very useful for volume rendering purposes. Fragment programs are executed by the fragment pipelines of the graphics card.

2.2 Writing Fragment Programs

Fragment programs can be written in assembly or through a more high-level languages such as Cg. We chose to use Cg, since it makes the code more readable to people who are used to the C programming language. This approach works by having the NVIDIA Cg compiler handle the conversion to assembly code.

There are drawbacks to this approach, as some control is lost. The NVIDIA Cg compiler can for instance pick an ordering of texture lookups that fails on the hardware. We have not found any way to enforce a particular order on the operations from the Cg code. It was necessary for us to sometimes manually edit the assembly code generated by the compiler in order to make particular fragment programs work at all.

2.3 Compiling Fragment Programs

The Cg compiler can be downloaded from `developer.nvidia.com`. It is invoked by the command `cgc` from the command line.

2.3.0.3 Hardware Profiles

The Cg code is meant to be hardware independent, but the assembly code is not, so the Cg compiler needs to know what hardware it compiles to. This is specified with the `-profile` flag. Hardware that supports the ARB fragment program extension will usually be able to run programs compiled with the `-profile arbfp1` option.

2.3.0.4 Other Hardware Options

There are also other limits associated with graphics hardware that may not be directly addressed through the hardware profile. Another useful Cg option is `-profileopts`, which allows more specific hardware limits to be addressed.

We used the `-profileopts MaxTexIndirections=4` option to make Cg use as few dependent texture lookups as possible. This option does not always work, so even if Cg (version 1.3) tells you your code uses too many texture indirections, it might still be possible to modify the assembly code to have it work. Use MaxTexIndirections=5 or more to have Cg compile your code and try to reorder the texture lookups.

2.3.0.5 An Example

Cg output which doesn't work (too many dependent texture lookups counted):

```
  TEX R1.w, fragment.texcoord[3], texture[3], 3D;
  TEX R3, R1.w, texture[7], 1D;
  TEX R0.w, fragment.texcoord[2], texture[2], 3D;
  TEX R2, R0.w, texture[6], 1D;
  TEX R1.w, fragment.texcoord[1], texture[1], 3D;
  TEX R1, R1.w, texture[5], 1D;
  TEX R0.w, fragment.texcoord[0], texture[0], 3D;
  TEX R0, R0.w, texture[4], 1D;
```

Modified texture lookup order which works:

```
  TEX R1.w, fragment.texcoord[3], texture[3], 3D;
  TEX R0.w, fragment.texcoord[2], texture[2], 3D;
  TEX R3, R1.w, texture[7], 1D;
  TEX R2, R0.w, texture[6], 1D;
  TEX R1.w, fragment.texcoord[1], texture[1], 3D;
  TEX R0.w, fragment.texcoord[0], texture[0], 3D;
  TEX R0, R0.w, texture[4], 1D;
  TEX R1, R1.w, texture[5], 1D;
```

2.4 Running Fragment Programs

OpenGL offers support for fragment programs through the ARB fragment program extension. Volumizer offers a simple way to load fragment programs into memory:

```
vzTMFragmentProgram *fp = vzTMFragmentProgram::load(filename);
```

Textures are uploaded through callbacks. A volume texture upload callback will look something like this:

```
static void tex1(vzTMShaderData *shaderData) {

 // Bind the "volume" texture
if(!shaderData->bindVolumeTextureCB("volume", shaderData, VZ_TM_BIND))
   vzError::error(VZ_OPERATION_FAILED, "CustomShader::tex1():
                  Error binding volume texture 'volume'");
}
```

and a color lookup table texture callback is only slightly different:

```
static void lut1(vzTMShaderData *shaderData) {

// Bind lookup_table
if(!shaderData->bindLookupTableCB("lookup_table", shaderData, VZ_TM_BIND))
   vzError::error(VZ_OPERATION_FAILED, "CustomShader::lut1():
                  Error binding lookup table 'lookup_table'");
}
```

2.5 Known Issues / Limitations

2.5.1 Texture Indirections

The SGI Prism comes with ATI graphics cards, ours has a FireGL card which currently does not support more than 4 dependent texture lookups. As each color table lookup is dependent on the previous volume texture lookup, we are thus limited to 4 volumes and 4 color tables. See A.3 for more information on texture indirections and how they are counted.

We also tried the NVIDIA GeForce FX 6800, which had the same limit. The GeForce also required the 1D color table textures to be uploaded to graphics memory before the 3D volume textures. The ATI card handles both orderings.

2.5.2 Cg 1.4 Incompatibility

Cg 1.4 was released as a beta this summer (June 2005), and apparently, the order of the texture coordinates has changed from 1.3. This means that our Cg files will not produce correct fragment programs if compiled with Cg 1.4.

3 SGI MULTIPIPE SDK

3.1 Introduction

The SGI Prism computer can come with multiple graphics cards. Thus software has to utilize multiple rendering pipelines to realize the full potential of the hardware.

SGI Multipipe SDK (MPK) is an API designed to help developers write multi-pipe applications. MPK handles the management of rendering threads, events and windows.

3.2 Special Considerations

3.2.1 Shared Memory

The introduction of more than a single thread represents a challenge to the developer. Care must be made to protect shared memory from being corrupted by other threads, which can produce unpredictable results.

3.2.1.1 Read-Only Data

is the simplest data to deal with in a multi-threaded setting. It is safe for several threads to access the same data as long as it isn't modified. For instance, simple data set or texture access is thread-safe as long as the datasets and textures don't change.

3.2.1.2 Data Library

One strategy is to create a data library class, in which all data is stored. All data access can go through get/set methods in this class. These methods can then easily be mutex-protected, so that no two threads would ever have access to the same piece of data. This approach can lead to performance loss if threads spend a significant amount of time waiting for mutex locks to be released. It is also important not to create a situation where two threads wait for each other to release locks held by the other (deadlock).

3.2.1.3 Copy Data

A different strategy is to provide each thread with its own copy of the data it needs access to. This allows it to freely access the data with no need to worry about data corruption. The drawback is that it still doesn't solve the problem of being able to set global variables. However, since a rendering thread usually doesn't update any scene variables, this approach is well suited for multi-pipe rendering. Another drawback is that if the copied data is large, more memory will be consumed, and a large amount of time could be spent each frame copying data.

3.2.2 Event-Driven vs. Main Loop

MPK is designed with event processing at the end of each frame drawn as the default. One might want the renderer to only draw new frames if the scene has been updated, which makes this kind of event processing useless, as it cannot look for events continuously if it doesn't redraw continuously. A work- around is to have a main loop which calls `mpkConfigHandleEvents` once every iteration, so it can look for events continuously.

If the application does not have a main loop, because it is already event- driven based on some other framework, it might be necessary to use a separate thread with a loop that calls `mpkConfigHandleEvents`.

3.2.3 Static vs. Class-Oriented

MPK is designed to run from a static context, that is, it is not possible to have MPK threads call non-static class member functions as callbacks. The interface for setting MPK callback functions only takes a function pointer, whereas an object pointer would be necessary as well, in order to call a non-static member function.

3.2.4 Qt

Qt GUI objects are not thread-safe, and updating the GUI from another thread than the main GUI thread will likely result in a crash or freeze.

3.3 VoluViz-MPK

VoluViz-MPK is a version of VoluViz 1.3 which uses SGI Multipipe SDK to scale rendering. Taking the above considerations into account, a lot of changes have been made to the VoluViz class structure (see figure 3.1 on page 26). The program GUI has also been changed slightly, since MPK windows are used instead of the old QGLWidget main window.

3.3.1 Configuring VoluViz-MPK

VoluViz-MPK uses a standard MPK configuration file to set up the rendering pipes and output windows. VoluViz-MPK will use the file given by `VOLUVIZ_MPK_CONFIG` or `../configs/1-pipe-1-window-right` if no configuration is given. A sample configuration file with an explanation of the file format is given in appendix A.6.

3.3.2 Using VoluViz-MPK

VoluViz-MPK can be run just as VoluViz. The main window is replaced by one or more MPK windows. VoluViz-MPK is multi-threaded, so it is for instance possible to manipulate the scene while an animation is running.

3.3.3 Program Structure

VoluViz-MPK introduces five new classes and a radically changed Viewer class. The other classes are left as they were. See figure 3.1 on the next page.

3.3.3.1 MpkRenderer

This class provides as static context and holds all the MPK callback functions used. This includes initialization, render, event and exit callbacks.

3.3.3.2 MpkSharedData

is where shared data is stored. Several threads may access data stored here, which is protected by mutexes. The standard way of accessing member data is described below:

```
MyData* myData = m_mpk_shared->getMyData(); // provides a pointer and locks
// do stuff with myData
m_mpk_shared->unlockData(); // release lock when done
```

or

```
m_mpk_shared->setMyData(myNewData); // locks data, updates it, then unlocks
```

The Volumizer scene graph (vzShape) is also stored in this class, but is not mutex protected during rendering, so all render threads can access it in a read-only fashion simultaneously. It is locked whenever it is updated.

3.3.3.3 MpkCopiedData

is data that is small enough to be copied to each render pipe. This includes most of the variables that were previously members of the Viewer class.

3.3.3.4 MpkEventThread

is a QThread that continuously polls for MPK window events. See figure 3.2 on page 27.

3.3.3.5 MpkRenderThread

is a QThread that checks if the scene has changed and renders the new scene if so. See figure 3.2 on page 27.

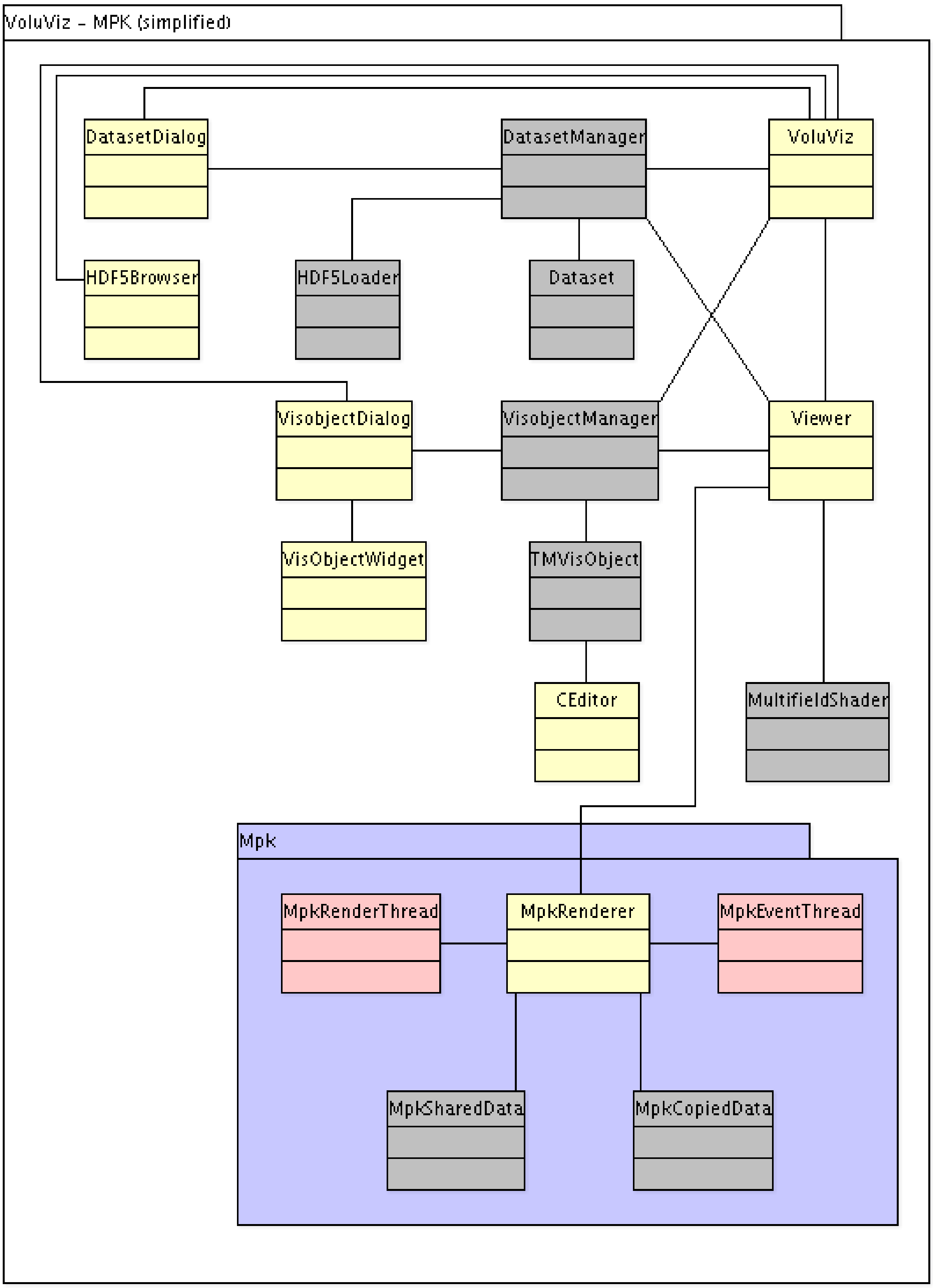


Figure 3.1 VoluViz MPK class diagram.

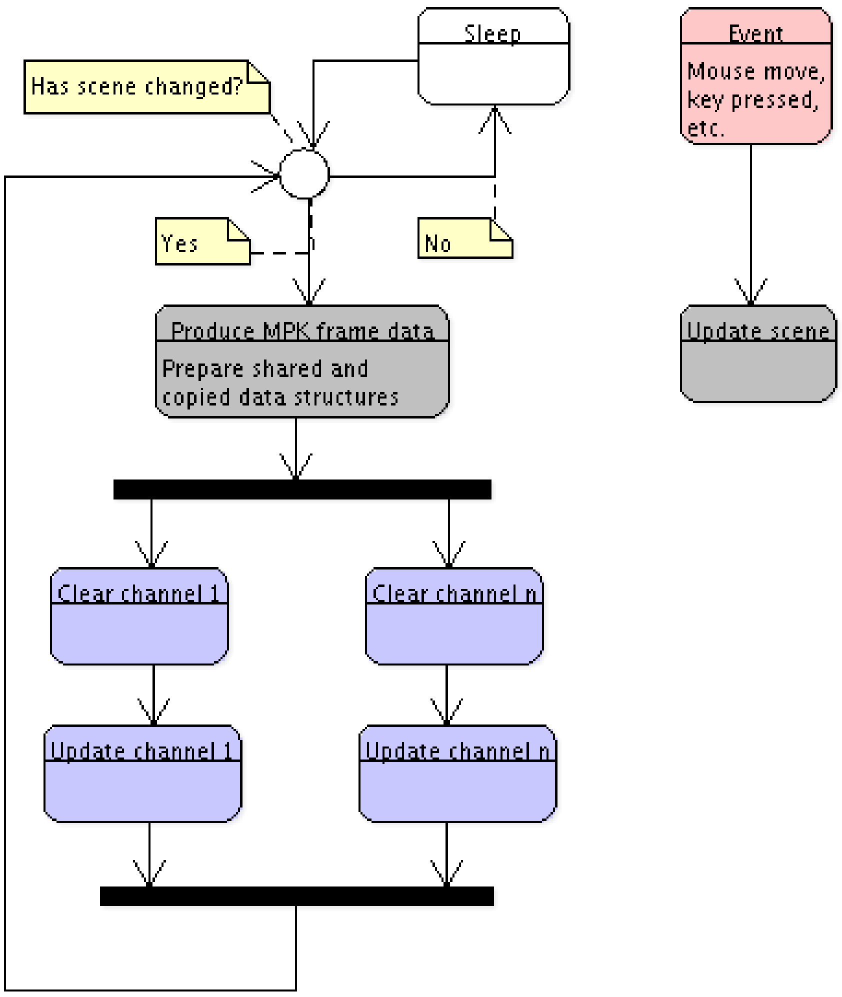


Figure 3.2 VoluViz redraw state diagram.

Dataset modes	**1 pipe**	**2 pipes**	**Rate of improvement**
L	17.7	31.2	1.76
LM	10.5	19.0	1.81
LML	4.7	9.0	1.91

Table 3.1 FPS rates for 1 and 2 pipe configurations.

3.3.3.6 Viewer

has been changed to use MpkRenderer to render instead of paintGL. It now only displays a logo in the Viewer area.

3.3.4 Scalability

There are several modes of decomposition available through MPK. Some are only interesting in very special environments, like EYE-decomposition for VR environments. The decompositions most interesting to volume rendering applications are 2D, DB and DPLEX.

3.3.4.1 2D

is the easiest way to decompose a scene for rendering on separate pipes. Each pipe renders a rectangular subregion of the total viewport. This decomposition mode requires very little change to the application code. Only the OpenGL PROJECTION and MODELVIEW matrices have to be transformed to get the right view for the different pipes. This is done by the `mpkChannelApplyFrustum` and `mpkChannelApplyTransformation` functions. The 2D-decomposition scales pixel fill rate quite well (see table 3.1), but does not scale texture memory.

3.3.4.2 DB

scales both pixel fill rate and texture memory. It is accomplished by dividing the volume into regions that are rendered on the different pipes. The rendered volumes are then blended together by a special blending routine, which is user-defined. Both partitioning and blending of the volume have to be implemented in the application for this to work.

3.3.4.3 DPLEX

decomposition is a temporal decomposition mode. Every Nth frame is rendered on the Nth pipe. This should be ideal for animations, but introduces an N-1 frame latency.

3.3.5 Known Issues / Limitations

3.3.5.1 Hardware Compositor

The SGI hardware compositor can take output from several pipes and output it onto a single output device. The hardware compositor is an expensive piece of hardware, so we have not had the opportunity to test it. Software composition involves copying the frame buffer from the source pipes onto the destination pipe. This operation is so slow that there is no performance gain at all. Thus, we are currently limited to using 2D decomposition with output on different displays.

3.3.5.2 Snapshots

VoluViz has traditionally used Qt snapshot functionality in order to grab and store the frame buffer. An MPK window does not support the same functionality, so this is still not possible. In order to get a snapshot of one screen, the `Print Screen` button can be used. To store animations is still impossible.

3.3.5.3 Rotation

The mouse rotation of a dataset is not quite right when there are more windows involved. The rotation routine is only aware of the window that is currently focused, so the dimensions of the other windows are not taken into account. This makes the rotation feel wrong in 2-window mode.

3.3.5.4 Random Crashes

The old VoluViz classes except for Viewer are not using data through thread-safe containers as of yet. If the Qt GUI thread and one of the MPK threads are using the same data at the same time, a crash will occur. This is extremely rare.

References

(1) *OpenGL Volumizer^{TM} 2 Programmer's Guide*. http://www.sgi.com/software/volumizer/.

(2) Bhaniramka P and Demange Y (2002): OpenGL Volumizer: A Toolkit for High Quality Volume Rendering of Large Data Sets. In *VVS '02: Proceedings of the 2002 IEEE symposium on Volume visualization and graphics*, pages 45–54. IEEE Press.

(3) Fernando R, editor (2004): *GPU Gems, Programming Techniques, Tips, and Tricks for Real-Time Graphics*. Addison-Wesley.

(4) Gaarder T and Helgeland A (2002): VoluViz 1.0 Report. Technical Report FFI/RAPPORT-2002/03449, FFI (Norwegian Defence Research Establishment).

APPENDIX

A APPENDIX

A.1 Installing VoluViz 1.3

VoluViz will most likely need to be compiled for your system. You need a C++ compiler. We have compiled VoluViz with gcc 3.2.3.

A.1.1 Dependencies

- Qt (tested with 3.3 and will most likely work with higher versions as well).
- SGI Volumizer 2 (tested with version 2.8).
- OpenGL.
- HDF5 (download from http://hdf.ncsa.uiuc.edu/HDF5/).
- vmq (Vector Matrix Quaternion Library).

A.1.2 Instructions

`VOLUVIZ` refers to the VoluViz base directory.

1. Go to `VOLUVIZ/src` and edit the file `voluviz_linux.pro`, `voluviz_linux_intel.pro` or `voluviz_irix64.pro` to match your system settings.
2. For gcc, type `qmake voluviz_linux.pro`.
3. Type `make`.
4. If the compile is successful, you can type `./voluviz` to start VoluViz.

A.1.2.1 Important Instructions for Qt

Be sure to compile Qt with OpenGL support! Use `./configure -thread` to configure Qt with OpenGL support.

A.2 Simple Cg Blending Program

```
// The values returned must be defined as a struct:
struct m_out {
  // Return a vector of 4 floating point values
  // to the color buffer (COLOR), RGBA-format
  float4 color : COLOR;
};

// Declaration of the fragment program
// Fetches texture indices from the TEXCOORD
// register to the texCoord variables
// Fetches textures and color tables
// loaded onto the hardware from
// the application.
m_out main(float3 texCoord1 : TEXCOORD0,// Fetch voxel index.
   float3 texCoord2 : TEXCOORD1,// etc.
   float3 texCoord3 : TEXCOORD2,// etc.
   uniform sampler3D voxture1,// Fetch first texture.
   uniform sampler3D voxture2,  // etc.
   uniform sampler3D voxture3,  // etc.
      uniform sampler1D lookup1,   // Color lookup table.
   uniform sampler1D lookup2,   // etc.
   uniform sampler1D lookup3)   // etc.
                                        // The order is IMPORTANT!
                            // Must be same as upload
// from application
{
  m_out OUT;
  fixed3x4 C;

  // Texture lookups at given coordinates:
  float4 tmpColor1 = tex3D(voxture1,texCoord1);
  float4 tmpColor2 = tex3D(voxture2,texCoord2);
  float4 tmpColor3 = tex3D(voxture3,texCoord3);

  // Lookups in color tables:
  C[0] = tex1D(lookup1,tmpColor1[3]);
  C[1] = tex1D(lookup2,tmpColor2[3]);
  C[2] = tex1D(lookup3,tmpColor3[3]);

  // Prepare alpha values:
  fixed3 a = { C[0][3], C[1][3], C[2][3] };
  fixed tmp = a[0] + a[1] + a[2];

  // Calculate C = a*C/sum(a), a = sum(a)
  OUT.color = mul(a,C);
  OUT.color /= tmp;
  OUT.color[3] = tmp;

  return OUT;
}
```

A.3 Excerpt from the ARB fragment program extension

The full specification can be read at:

`http://oss.sgi.com/projects/ogl-sample/registry/ARB/fragment_program.txt`

```
(24) What is a texture indirection, and how is it counted?

      RESOLVED: On some implementations, fragment programs that have
      complex texture dependency chains may not be supported, even if
      the instruction counts fit within the exported limits.  A texture
      dependency occurs when a texture instruction depends on the
      result of a previous instruction (ALU or texture) for use as its
      texture coordinate.

      A texture indirection can be considered a node in the texture
      dependency chain.  Each node contains a set of texture
      instructions which execute in parallel, followed by a sequence of
      ALU instructions.  A dependent texture instruction is one that
      uses a temporary as an input coordinate rather than an attribute
      or a parameter.  A program with no dependent texture instructions
      (or no texture instructions at all) will have a single node in
      its texture dependency chain, and thus a single indirection.

      API-level texture indirections are counted by keeping track of
      which temporaries are read and written within the current node in
      the texture dependency chain.  When a texture instruction is
      encountered, an indirection may be added and a new node started
      if either of the following two conditions is true:

        1. the source coordinate of the texture instruction is a
           temporary that has already been written in the current node,
           either by a previous texture instruction or ALU instruction;

        2. the result of the texture instruction is a temporary that
           has already been read or written in the current node by an
           ALU instruction.

      The texture instruction provoking a new indirection and all
      subsequent instructions are added to the new node.  This process
      is repeated until the end of the program is encountered.  Below
      is some pseudo-code to describe this:

        indirections = 1;
        tempsOutput = 0;
        aluTemps = 0;
        while (i = getInst())
        {
          if (i.type == TEX)
          {
            if (((i.input.type == TEMP) &&
                   (tempsOutput & (1 << i.input.index))) ||
                 ((i.op != KILL) && (i.output.type == TEMP) &&
                   (aluTemps & (1 << i.output.index))))
            {
              indirections++;
              tempsOutput = 0;
              aluTemps = 0;
            }
          } else {
            if (i.input1.type == TEMP)
              aluTemps |= (1 << i.input1.index);
            if (i.input2 && i.input2.type == TEMP)
              aluTemps |= (1 << i.input2.index);
            if (i.input3 && i.input3.type == TEMP)
              aluTemps |= (1 << i.input3.index);
            if (i.output.type == TEMP)
              aluTemps |= (1 << i.output.index);
          }
          if ((i.op != KILL) && (i.output.type == TEMP))
            tempsOutput |= (1 << i.output.index);
        }
```

```
For example, the following programs would have 1, 2, and 3
texture indirections, respectively:

  !!ARBfp1.0
  # No texture instructions, but always 1 indirection
  MOV result.color, fragment.color;
  END

  !!ARBfp1.0
  # A simple dependent texture instruction, 2 indirections
  TEMP myColor;
  MUL myColor, fragment.texcoord[0], fragment.texcoord[1];
  TEX result.color, myColor, texture[0], 2D;
  END

  !!ARBfp1.0
  # A more complex example with 3 indirections
  TEMP myColor1, myColor2;
  TEX myColor1, fragment.texcoord[0], texture[0], 2D;
  MUL myColor1, myColor1, myColor1;
  TEX myColor2, fragment.texcoord[1], texture[1], 2D;
  # so far we still only have 1 indirection
  TEX myColor2, myColor1, texture[2], 2D;      # This is #2
  TEX result.color, myColor2, texture[3], 2D;  # And #3
  END

Note that writemasks for the temporaries written and swizzles
for the temporaries read are not taken into consideration when
counting indirections.  This makes hand-counting of indirections
by a developer an easier task.

Native texture indirections may be counted differently by an
implementation to reflect its exact restrictions, to reflect the
true dependencies taking into account writemasks and swizzles,
and to reflect optimizations such as instruction reordering.

For implementations with no restrictions on the number of
indirections, the maximum indirection count will equal the
maximum texture instruction count.
```

A.4 Porting an OpenGL application to multipipe

```
Changes required to compile and link with MPK:
```

```
Link to: -lpthread and -lmpk

See appendix 2 for code for each step.
--- Step 1 ---
#include <mpk/mpk.h>
--------------

--- Step 2 ---
Make structure for shared
data. Data accessible from
all pipes. Only read-only
data should be accessed
by more than one pipe at
a time. If data is modified,
crashes can occur. Use mutexes
or copy data if modification
is required.
--------------

--- Step 3 ---
Make a structure for data
that is copied and sent
individually to each pipe.
This is safe data, but if
it is large, performance
overhead will be big.
--------------

--- Step 4 ---
Load MPK configuration:
mpkGlobalSetExecutionMode( MPK_EXECUTION_PTHREAD );
mpkInit();
MPKConfig *config = mpkConfigLoad(filename);
mpkConfigOutput(config,0);

Set callbacks (more thorough explanations
of the most common callback functions
follows below):
mpkConfigSetWindowInitCB(config,windowInit);
mpkConfigSetWindowExitCB(config,windowExit);
mpkConfigSetChannelInitCB(config,channelInit);
mpkConfigSetDataFreeCB(config,CBfunc4);

Activate configuration:
mpkConfigInit(config,0);
--------------

--- Step 5 ---
Program is organized around
a main loop.
- First update scene database.
- Then update frame data (copy).
  (must be allocated with mpkMalloc!)
- Finally run mpkConfigFrame to
  tell MPK to start rendering threads.
--------------
```

```
--- Step 6 ---
For each frame and each pipe,
an update callback function
is run.
Inside it, access to copied
data is provided. Rendering
is called from this function.
See updateChannel in the
sample program in appendix B
to get a more detailed idea
of the operations that this
function should do.
--------------

--- Step 7 ---
As the program terminates, end
MPK-mode and cancel the callbacks.
mpkConfigSetWindowInitCB(config,NULL);
mpkConfigSetChannelInitCB(config,NULL);
mpkConfigChangeMode( config, MPK_MONO );
mpkConfigExit(config);
mpkConfigDelete(config);
mpkFree(shared);
mpkExit();
--------------

--- CALLBACKS ---

--- mpkConfigSetWindowInitCB ---
void windowInit( MPKWindow *w)
Called once for each window
created. Use it to set window
callbacks and attach a render
action to the window.

mpkWindowSetDrawCB( w, MPK_WINDOW_DRAWCB_INIT_X, initWindowX );
mpkWindowSetDrawCB( w, MPK_WINDOW_DRAWCB_INIT_GL, initWindowGL );
mpkWindowSetDrawCB( w, MPK_WINDOW_DRAWCB_EXIT_X, exitWindowX );
mpkWindowSetDrawCB( w, MPK_WINDOW_DRAWCB_EXIT_GL, exitWindowGL );
mpkWindowSetEventCB(w,MPK_WINDOW_EVENTCB_MOUSE,windowMouse);
mpkWindowSetEventCB(w,MPK_WINDOW_EVENTCB_BUTTON,windowMouse);
mpkWindowSetEventCB(w,MPK_WINDOW_EVENTCB_EXIT,windowExit);
mpkWindowSetEventCB(w,MPK_WINDOW_EVENTCB_KEYBOARD,windowKeyboard);

vzTMRenderAction* action = new vzTMRenderAction(1);
void* data = action;
mpkWindowSetUserData(w, data);
--------------------------------

--- mpkConfigSetWindowExitCB ---
Probably not too useful - SGI's sample
application uses a NULL pointer for
this one.
--------------------------------

--- mpkConfigSetChannelInitCB ---
void channelInit( MPKChannel *c )
Set channel draw callbacks:
```

```
mpkChannelSetDrawCB( c, MPK_CHANNEL_DRAWCB_CLEAR, clearChannel );
mpkChannelSetDrawCB( c, MPK_CHANNEL_DRAWCB_UPDATE, updateChannel );
---------------------------------

--- mpkConfigSetDataFreeCB ---
void dataFree( MPKConfig *cfg, void *data )
Delete frame data. There are various
faster ways to do this than the default.

Default:
FrameData *frameData = (FrameData*) data;
mpkFree(frameData);
------------------------------

--- mpkWindowSetDrawCB’s ---
These are called on different window
events. All return void and take a
MPKWindow* as parameter.

MPK_WINDOW_DRAWCB_INIT_X called when
X-window is created. Must contain
mpkWindowCreate( w );

MPK_WINDOW_DRAWCB_INIT_GL called once
for every window to initialize OpenGL
context.
mpkWindowCreateContext( w );
mpkWindowMakeCurrent( w );
mpkWindowApplyViewport( w );

MPK_WINDOW_DRAWCB_EXIT_X called when
window is destroyed. Should probably
unmanage shape and remove RenderAction.
Must also call mpkWindowDestroy( w );

MPK_WINDOW_DRAWCB_EXIT_GL called when
window is destroyed
mpkWindowMakeCurrentNone( w );
mpkWindowDestroyContext( w );

MPK_WINDOW_EVENTCB_MOUSE called when
mouse move events occur.
void mouseCB( MPKWindow *w, MPKEvent *event )

MPK_WINDOW_EVENTCB_BUTTON called when
mouse press events occur.
void buttonCB( MPKWindow *w, MPKEvent *event )

MPK_WINDOW_EVENTCB_KEYBOARD called
when a key is pressed or released.
void keyboardCB( MPKWindow *w, MPKEvent *event )

MPK_WINDOW_EVENTCB_EXIT called when a
window is closed.
void exitCB( MPKWindow *w, MPKEvent *event )

--- mpkChannelSetCB’s ---
MPK_CHANNEL_DRAWCB_CLEAR is called once for
```

```
each frame for each channel to clear it.
void clearCB( MPKChannel *c )
mpkChannelApplyBuffer( c );
mpkChannelApplyViewport( c );
glClear( GL_COLOR_BUFFER_BIT );

MPK_CHANNEL_DRAWCB_UPDATE is called once for
each frame for each channel in order to
draw it. This is where the 'meat' of your
MPK application rendering should go!
void updateCB( MPKChannel *c, void* data )
Unpack render action:
MPKWindow* w = mpkChannelGetWindow( c );
vzTMRenderAction* action = (vzTMRenderAction*) mpkWindowGetUserData( w );
Update projection and modelview matrices to
match you MPK setup:
mpkChannelApplyFrustrum( c );
mpkChannelApplyTransformation( c );
Do your drawing!
```

A.5 Sample OpenGL application ported to multipipe

This example is based on the SGI pguide example that is shipped along with Volumizer 2.8, and the example in the SGI MPK Programmer's Guide.

```
// SGI Multipipe SDK
//-----------
// Step 1
//-----------
#include <mpk/mpk.h>

// Misc
#include <X11/keysym.h>
#include <math.h>
#include <stdio.h>
#include <string.h>
#include <iostream.h>

// IFL
#include <loaders/IFLLoader.h>

// Volumizer2
#include <Volumizer2/Version.h>
#include <Volumizer2/Error.h>
#include <Volumizer2/Shape.h>
#include <Volumizer2/Block.h>
#include <Volumizer2/Appearance.h>
#include <Volumizer2/ParameterVolumeTexture.h>
#include <Volumizer2/TMRenderAction.h>
#include <Volumizer2/TMSimpleShader.h>


#ifndef M_PI
#define M_PI 3.1415926535
#endif
```

```
#define DEG2RAD(a)  ((a)*M_PI/180.)

//-------------
// Step 2
//-------------
typedef struct
{
  int    stereo;
  int    exit;

  float   xangle, yangle,
    dx, dy, dz,
    translation[3],
    rotation[16];
  //  vzShape *shape;

} Shared;

//-------------
// Step 3
//-------------
typedef struct _FrameData
{
  float   translation[3],
    rotation[16];

  struct _FrameData *next;

} FrameData;

Shared    *shared;
FrameData *frameDataBuffer = NULL;
vzShape   *shape;
FrameData *newFrameData( Shared *shared );
void       freeFrameData( MPKConfig *, void * );

void    initSharedData( Shared *shared );
void    exitSharedData( Shared *shared );

void    initWindow( MPKWindow * );
void    initWindowX( MPKWindow * );
void    initWindowGL( MPKWindow * );
void    exitWindowX( MPKWindow * );
void    exitWindowGL( MPKWindow * );

void    initChannel( MPKChannel *c );
void    clearChannel( MPKChannel *, void * );
void    updateChannel( MPKChannel *, void * );

void    windowMouse( MPKWindow *, MPKEvent * );
void    windowExit( MPKWindow *, MPKEvent * );
void    windowKeyboard( MPKWindow *, MPKEvent * );

void    incrementRotation( float *, float, float );
static void printError(vzErrorSeverity severity, vzErrorType type,
       const char* format, va_list args, void* data);
```

```
// Global variables:
static float lightpos[] = { 0., 0., 1., 0. };
//vzTMRenderAction *renderAction = NULL;
GLint viewport[4];

//-----------------------------------------------------------
// Volume Visualization code
//-----------------------------------------------------------
// loadVolumeData - Load the volume data and initialize the shape node.
void loadVolumeData(char *fileName)
{
    // Create a data loader
    IFLLoader *loader = IFLLoader::open(fileName);
    if (loader == NULL) {
        cerr<<"Error: couldn't open file "<<fileName<<endl;
exit(0);
    }

    // Load the volume data
    vzParameterVolumeTexture *volume = loader->loadVolume();
    if (volume == NULL) {
delete loader;
        cerr<<"Error: couldn't read volume data"<<endl;
  exit(0);
    }

    // Initialize appearance
    vzShader *shader = new vzTMSimpleShader();
    vzAppearance *appearance = new vzAppearance(shader);
    shader->unref();
    appearance->setParameter("volume", volume);
    volume->unref();

    // Initialize geometry
    vzGeometry *geometry = new vzBlock();

    // Initialize shape node
    shape = new vzShape(geometry, appearance);
    geometry->unref();
    appearance->unref();

    //    if(!renderAction) {
      // Initialize the render action
    //      renderAction = new vzTMRenderAction(1);
    //    }
    // renderAction->manage(shared->shape);
}

// Cleanup up the shape node and the render action
void cleanup()
{
    shape->unref();
}


//-----------------------------------------------------------------------
//  main
//-----------------------------------------------------------------------
```

```
main( int argc, char *argv[] )
{
  //-----------
  // Step 4
  //-----------
    MPKConfig *config;
    FrameData *framedata;

    mpkGlobalSetExecutionMode( MPK_EXECUTION_PTHREAD );

    mpkInit();

    // shared must be allocated after mpkInit().
    shared = (Shared*)mpkMalloc( sizeof( Shared ));
    initSharedData( shared );

    if (argc < 2)
        config = mpkConfigLoad("../configs/1-window");
    else
        config = mpkConfigLoad(argv[1]);

    if ( config == NULL )
    {
        fprintf(stderr,"Can t load config file.\n");
        exit (0);
    }

    loadVolumeData(argv[2]);

    mpkConfigOutput( config, 0 );

    shared->stereo = ( mpkConfigGetMode(config)==MPK_STEREO ? 1 : 0 );

    mpkConfigSetWindowInitCB(config,initWindow);
    mpkConfigSetWindowExitCB(config,NULL);
    mpkConfigSetChannelInitCB(config,initChannel);
    mpkConfigSetDataFreeCB( config, freeFrameData );

    mpkConfigInit( config, 0 );


    vzError::setHandler (printError, NULL);

    while (!shared->exit )
    {
      //------------
      // Step 5
      //------------

        // update DB
        incrementRotation(  shared->rotation,
            shared->xangle,
            shared->yangle );

        shared->translation[0] += shared->dx;
        shared->translation[1] += shared->dy;
        shared->translation[2] += shared->dz;
        shared->dx = 0.;
        shared->dy = 0.;
        shared->dz = 0.;

        // new frame
//        mpkConfigChangeMode( config, shared->stereo );
        framedata = newFrameData( shared );
        mpkConfigFrame( config, framedata );
    }

    //---------- restore MONO

    //-----------
    // Step 7
    //-----------
    mpkConfigSetWindowInitCB(config,NULL);
    mpkConfigSetChannelInitCB(config,NULL);
    mpkConfigChangeMode( config, MPK_MONO );

    //---------- exit & delete config

    mpkConfigExit( config );
    mpkConfigDelete( config );

    exitSharedData( shared );
    mpkFree( shared );

    mpkExit();
}

//----------------------------------------------------------------------
//  printError
//----------------------------------------------------------------------
static void printError( vzErrorSeverity severity, vzErrorType type,
 const char* format, va_list args, void* data)
{
  if (severity == VZ_ERROR)
    cerr<<"myHandler::Error!!!";
  else if(severity == VZ_WARNING)
    cerr<<"myHandler::Warning!!!";

  // Print the error message
  vfprintf(stderr, format, args);

}
//----------------------------------------------------------------------
//  initSharedData
//----------------------------------------------------------------------
void initSharedData( Shared *shared )
{
    int i, j;

    shared->exit = 0;
    shared->stereo = MPK_MONO;
    shared->yangle = 0.;
    shared->xangle = 0.;

    for (i=0; i<4; i++)
```

```
        for (j=0; j<4; j++)
            shared->rotation[4*i+j] = (i==j) ? 1. : 0.;

    shared->dx = 0.;
    shared->dy = 0.;
    shared->dz = 0.;
    shared->translation[0] = 0.;
    shared->translation[1] = 0.;
    shared->translation[2] = -2.;
}

//-----------------------------------------------------------------------
//  exitSharedData
//-----------------------------------------------------------------------
void exitSharedData( Shared *shared )
{
    while ( frameDataBuffer != NULL )
    {
        FrameData *framedata = frameDataBuffer;

        frameDataBuffer = framedata->next;

        mpkFree( framedata );
    }
}

//-----------------------------------------------------------------------
//  initWindow
//-----------------------------------------------------------------------
void initWindow( MPKWindow *w )
{
    mpkWindowSetDrawCB( w, MPK_WINDOW_DRAWCB_INIT_X, initWindowX );
    mpkWindowSetDrawCB( w, MPK_WINDOW_DRAWCB_INIT_GL, initWindowGL );
    mpkWindowSetDrawCB( w, MPK_WINDOW_DRAWCB_EXIT_X, exitWindowX );
    mpkWindowSetDrawCB( w, MPK_WINDOW_DRAWCB_EXIT_GL, exitWindowGL );

    mpkWindowSetEventCB(w,MPK_WINDOW_EVENTCB_MOUSE,windowMouse);
    mpkWindowSetEventCB(w,MPK_WINDOW_EVENTCB_BUTTON,windowMouse);
    mpkWindowSetEventCB(w,MPK_WINDOW_EVENTCB_EXIT,windowExit);
    mpkWindowSetEventCB(w,MPK_WINDOW_EVENTCB_KEYBOARD,windowKeyboard);

    // initialize render action
    vzTMRenderAction* action = new vzTMRenderAction(1);
    action->manage(shape);

    // attach render action to window
    void* data = action;
    mpkWindowSetUserData(w, data);

}

//-----------------------------------------------------------------------
// initWindowX
//-----------------------------------------------------------------------
void initWindowX( MPKWindow *w )
{
    mpkWindowCreate( w );
}
```

```
//----------------------------------------------------------------------
// initWindowGL
//----------------------------------------------------------------------
void initWindowGL( MPKWindow *w )
{
    // create ctx
    mpkWindowCreateContext( w );
    mpkWindowMakeCurrent( w );

    // GL initialization
    mpkWindowApplyViewport( w );

    glEnable( GL_DEPTH_TEST );
    glDepthFunc (GL_LESS);

    glEnable( GL_LIGHTING );
    glEnable( GL_LIGHT0 );
    glLightfv( GL_LIGHT0, GL_POSITION, lightpos );

    glColorMaterial( GL_FRONT_AND_BACK, GL_DIFFUSE );
    glEnable( GL_COLOR_MATERIAL );

    glClearDepth( 1. );
    glClearColor( 0., 0., 0., 1. );

    // clear both buffers
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    mpkWindowSwapBuffers(w);
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
}

//----------------------------------------------------------------------
// exitWindowX
//----------------------------------------------------------------------
void exitWindowX( MPKWindow *w )
{
    // get render action
    void* windowData = mpkWindowGetUserData(w);
    vzTMRenderAction* action = (vzTMRenderAction*) windowData;

    // unmanage the shape
    action->unmanage(shape);

    delete action;

    mpkWindowDestroy( w );
}

//----------------------------------------------------------------------
// exitWindowGL
//----------------------------------------------------------------------
void exitWindowGL( MPKWindow *w )
{
    // destroy ctx
    mpkWindowMakeCurrentNone( w );
    mpkWindowDestroyContext( w );
}
```

```
//-----------------------------------------------------------------------
//  initChannel
//-----------------------------------------------------------------------
void initChannel( MPKChannel *c )
{
    mpkChannelSetDrawCB( c, MPK_CHANNEL_DRAWCB_CLEAR, clearChannel );
    mpkChannelSetDrawCB( c, MPK_CHANNEL_DRAWCB_UPDATE, updateChannel );
}

//-----------------------------------------------------------------------
//  newFrameData
//-----------------------------------------------------------------------
FrameData *newFrameData( Shared *shared )
{
    FrameData *framedata;
    if ( frameDataBuffer == NULL )
    {
        framedata = (FrameData *) mpkMalloc( sizeof(FrameData) );
    }
    else
    {
        framedata = frameDataBuffer;
        frameDataBuffer = framedata->next;
    }

    framedata->next = NULL;

    memcpy( framedata->translation,
            shared->translation, 3*sizeof(float) );
    memcpy( framedata->rotation, shared->rotation, 16*sizeof(float) );

    return framedata;
}

//-----------------------------------------------------------------------
//  freeFrameData
//-----------------------------------------------------------------------
void freeFrameData( MPKConfig *cfg, void *data )
{
    FrameData *framedata = (FrameData *)data;

    framedata->next = frameDataBuffer;
    frameDataBuffer = framedata;
}

//-----------------------------------------------------------------------
//  clearChannel
//-----------------------------------------------------------------------
void clearChannel( MPKChannel *c, void *data )
{
    mpkChannelApplyBuffer( c );
    mpkChannelApplyViewport( c );

    //    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    glClear( GL_COLOR_BUFFER_BIT );
}
```

```
//---------------------------------------------------------------------
//  updateChannel
//---------------------------------------------------------------------
//-----------
// Step 6
//-----------
void updateChannel( MPKChannel *c, void *data )
{
    FrameData *framedata = (FrameData *)data;

    // get render action for window
    MPKWindow* window = mpkChannelGetWindow(c);
    vzTMRenderAction* action = (vzTMRenderAction*) mpkWindowGetUserData(window);

    // projection matrix
    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
    mpkChannelApplyFrustum( c );

    // modelview matrix
    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();
    mpkChannelApplyTransformation( c );
    glTranslatef(  framedata->translation[0],
        framedata->translation[1],
        framedata->translation[2] );

    glMultMatrixf( framedata->rotation );

    // enable back-to-front alpha blending
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

    action->beginDraw(VZ_RESTORE_GL_STATE_BIT);
    action->draw(shape);
    action->endDraw();
}

//---------------------------------------------------------------------
//  windowMouse
//---------------------------------------------------------------------
void windowMouse( MPKWindow *w, MPKEvent *event )
{
    MPKEventXData *data = (MPKEventXData *) mpkEventGetData( event );

    if ( data->button.left )
    {
      shared->dx += (float) data->mouse.dx/500.;
      shared->dy -= (float) data->mouse.dy/500.;
    }
    else if ( data->button.middle )
    {
      shared->xangle = (float) data->mouse.dy*.5;
      shared->yangle  = (float) data->mouse.dx*.5;
    }
    else if ( data->button.right )
    {
      shared->dz += (float) data->mouse.dy/200.;
    }

}

//--------------------------------------------------------------------
//  windowExit
//--------------------------------------------------------------------
void windowExit( MPKWindow *w, MPKEvent *event )
{
    shared->exit = GL_TRUE;
}

//--------------------------------------------------------------------
//  windowKeyboard
//--------------------------------------------------------------------
void windowKeyboard( MPKWindow *w, MPKEvent *event )
{
    MPKEventXData *data = (MPKEventXData *)mpkEventGetData( event );

    if ( data->keyboard.state != MPK_PRESS )
        return;

    switch( data->keyboard.key )
    {
        case XK_S:
        case XK_s:
            shared->stereo = !shared->stereo;
            break;
        case XK_Q:
        case XK_q:
            cleanup();
    exit(0);
            break;
    }
}

//--------------------------------------------------------------------
//  incrementRotation
//--------------------------------------------------------------------
static void xformColumn(    float *m, int i, int j, int k,
                            float cosX, float sinX,
                            float cosY, float sinY )
{
    float aux = sinX*m[j] + cosX*m[k];
    float x = sinY*aux + cosY*m[i];
    float y = cosX*m[j] - sinX*m[k];
    float z = cosY*aux - sinY*m[i];
    m[i] = x;
    m[j] = y;
    m[k] = z;
}

void incrementRotation( float *matrix, float xangle, float yangle )
{
    float cosX, sinX, cosY, sinY;

    cosX = cos( DEG2RAD(xangle) );
    sinX = sin( DEG2RAD(xangle) );
    cosY = cos( DEG2RAD(yangle) );
    sinY = sin( DEG2RAD(yangle) );
    xformColumn( matrix, 0, 1, 2, cosX, sinX, cosY, sinY );
    xformColumn( matrix, 4, 5, 6, cosX, sinX, cosY, sinY );
    xformColumn( matrix, 8, 9, 10, cosX, sinX, cosY, sinY );
}
```

A.6 Simple MPK configuration file

This configuration file will make an MPK application render to two displays, using two graphics pipes.

```
global {
  # Show window decoration?
  MPK_WATTR_HINTS_DECORATION 0
}
config
{
  name    "2-pipes.decomp"
  mode    mono

  pipe
  {
    # This pipe delivers data to screen :0.0
    # (usually means the left screen)
    display ":0.0"
    window
    {
      # Title (displayed on window decoration)
      name  "MPK demo, left screen"

      # Size of window compared to size of screen:
      # 1.0 is a full screen width / height
      # Origin is upper left corner
      # viewport [ startX, startY, width, height ]
      viewport    [ 0.0, 0.0, 1.0, 1.0 ]
      channel
      {
name "destination"
# Defines the region of the window:
# that is drawn.
# viewport [ startX, startY, width, height ]
# Origin is lower left corner of window
        viewport    [ 0.0, 0.0, 1.0, 1.0 ]
        wall
        {
  # Defines which part of the global scene to draw
  # bottom_left [ x, y, z(oom) ]
  # bottom_left [ x, y, z(oom) ]
  # lower z-value zooms in, higher zooms out
          bottom_left     [ -1.0, -0.5, -1 ]
          bottom_right    [  0.0, -0.5, -1 ]
          top_left        [ -1.0,  0.5, -1 ]
        }
      }
    }
  }
  pipe
  {
    # Next pipe definition, this time in Norwegian
    # This pipe delivers data to screen :0.1
    display ":0.1"
    window
    {
      # Title (displayed on window decoration)
      name        "MPK demo, right screen"

      # Size of window compared to size of screen:
      # 1.0 is a full screen width / height
      # Origin is upper left corner
      # viewport [ startX, startY, width, height ]
      viewport    [ 0.0, 0.0, 1.0, 1.0 ]
      channel
      {
        name        "source0"

# Defines the region of the window:
# that is drawn.
# viewport [ startX, startY, width, height ]
# Origin is lower left corner of window
        viewport    [ 0.0, 0.0, 1.0, 1.0 ]
        wall
        {
  # Defines which part of the global scene to draw
  # bottom_left [ x, y, z(oom) ]
  # bottom_left [ x, y, z(oom) ]
  # lower z-value zooms in, higher zooms out
          bottom_left     [ 0.0, -0.5, -1 ]
          bottom_right    [ 1.0, -0.5, -1 ]
          top_left        [ 0.0,  0.5, -1 ]
        }
      }
    }
  }
}
```