Chapter 1: Introduction

Breast cancer is a common form of cancer among women. Mammography is used for early detection, but it cannot detect all instances of cancer. New magnetic resonance imaging (MRI) protocols developed by Stanford University and used at Presbyterian Hospital have detected breast cancer where mammography has failed. Applying post-processing computer algorithms to these scans can semi-automate detection and classification.

Radiologists have traditionally relied on 2D grayscale images to analyze MR scans as series of slices, but 3D visualizations can provide a much better spatial context. However, 3D visualizations can involve large amounts of computatation, requiring hardware support to render at interactive rates.

In the last few years, graphics hardware has made significant advances in speed and flexibility. Expensive workstations have been replaced by inexpensive consumer graphics hardware. Previously fixed-pipeline graphics processing units (GPUs) have gained low-level programmability at various stages of the rendering pipeline. *Shader programs* are used to override certain stages of the rendering pipeline in GPUs to allow a much greater variety of rendering effects. While shaders have been implemented in software in the past, shader implementations in hardware have the capability to perform much faster. As more advanced GPUs are developed, they will likely continue to gain flexibility and programmability.

Using inexpensive consumer hardware, it is now possible to render MR data in 3D at interactive rates. This paper presents the design of breast cancer detection and visualization software that uses such hardware. The software is highly interactive, allowing detection and

visualization settings to be modified in real time. This allows users to experiment with settings and immediately see the results. The software provides the familiar 2D slice visualization with the addition of a 3D view of the MR data. The 3D view provides a far more natural and intuitive visualization that allows users to see and interact with the entire MR dataset.

A design goal of this project was to produce reusable and easily maintainable code and to minimize application-specific code. The software includes a scene rendering library and a set of multi-dimensional array and dataset classes that may be used in future applications.

This paper discusses the technology, methods, and design of the software. Chapter 2 provides a brief background of volume rendering and discusses current technologies. Chapter 3 describes the detection and rendering methods employed by the software. Chapter 4 contains a guide to the user interface of the software. Chapter 5 describes experimentation with rendering and detection settings and the results. Chapter 6 concludes by discussing future work on this project. The appendices contain implementation specifics of the software and its supporting libraries.

Chapter 2: Background

2.1 Gadolinium-Enhanced Tumor Detection

The breast cancer detection methods employed by the software rely on data from a series of breast MR scans, which are produced through a new MRI protocol. In this protocol, an initial scan of the breast is taken. Gadolinium is injected, and four additional scans are taken over the next few minutes. When present in blood vessels, gadolinium yields higher intensities in scans. Intensities at a particular point within the breast can be represented graphically as a time step intensity curve. This curve represents the set of intensities at a given position within the five volumes. In cancerous cells, these curves often exhibit a high initial intensity rate and a slow, steady washout rate.

2.2 Volume Rendering

A volumetric dataset is a 3D array of scalar or vector values, often produced by sampling a continuous volume. Values in a volumetric dataset are often interpreted as cubic *voxels*, which are 3D versions of 2D pixels. The two primary approaches to volume rendering are isosurface rendering and direct volume rendering.

2.2.1 Isosurface Rendering

Isosurface rendering clearly shows distinct features in a volume by extracting the surface geometry of the desired features. An *isosurface* is a 2D surface, or a contour in three-space. It represents a connected set of values that are equal to an *isovalue*. Isosurfaces are produced by determining an appropriate isovalue and generating geometry. The *Marching Cubes* algorithm is often used to generate isosurface geometry. While isosurface rendering is well suited for

rendering easily segmented, distinct features, it works poorly with datasets containing high amounts of noise or lacking contrast.

2.2.2 Direct Volume Rendering

Direct volume rendering does not attempt to extract isosurfaces from a dataset. All data contributes to the final image, preventing less distinct features from being lost in the rendering process. Direct volume rendering is a computationally expensive but precise way to visualize volumetric data. In the past, hardware limitations have prevented it from being a feasible way of rendering at interactive rates, but it is now possible with current hardware.

All methods of direct volume rendering attempt to evaluate the color and opacity integral along a ray into the volume for each pixel. The process of rendering, or attempting to evaluate this integral at each pixel of the image, can be performed by either image-based or object-based approaches. Ray casting is an image-based approach that attempts to directly evaluate the integral through the volume for each pixel. Object-based methods operate by applying textures to *proxy geometry* such as a set of polygonal slices.

Ray casting is a highly computationally expensive image-based approach to volume rendering that requires a ray to be evaluated for each pixel in the output image. Evaluating a ray involves sampling the volume at intervals along the ray. While current hardware is not designed for ray casting methods, simple ray casting can be implemented in fragment shaders. However, it is currently an unsuitable approach for more advanced volume rendering at interactive rates.

Common object-based rendering methods are axis-aligned and view-aligned slice rendering. Both methods blend a series of polygonal, semi-transparent volume slices in front-to-back or

back-to-front order. Axis-aligned slices are rendered along the major axis closest to the view vector. This method is suitable for video cards without 3D texturing support because it only requires a series of 2D textures and bilinear interpolation. However, it requires three sets of slices to be present in video memory for the three major axes. View-aligned slices are always rendered orthogonal to the view direction, but require 3D texturing support. Sampling at an arbitrary location in a 3D texture requires trilinear interpolation. Because of the computation involved, hardware support is required for rendering at interactive rates.

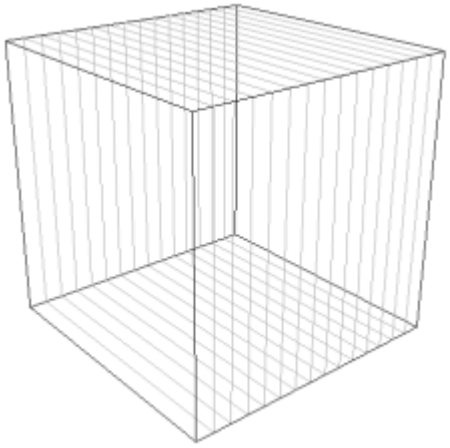


Figure 2.1a. Axis-Aligned Slices

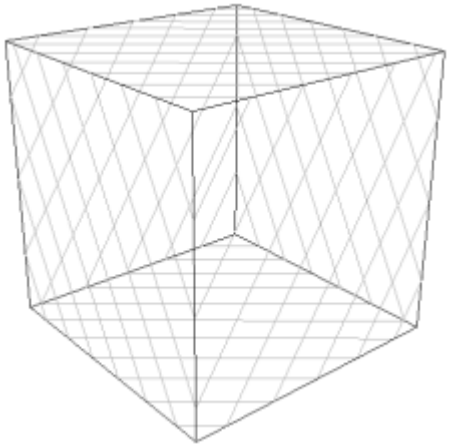


Figure 2.1b. View-Aligned Slices

2.2.3 Transfer Functions

A transfer function maps a volumetric dataset value to an actual color. Volumetric data is typically stored in its original form in a texture rather than storing color and alpha values (RGBA values) in the texture. One advantage of storing original values is that the original values will often take less space than the corresponding RGBA values. Another advantage is that changing color parameters will not require RGBA values in the entire texture to be updated. When sampling the texture, the hardware will use the texture value to obtain an

RGBA value by either sampling an additional texture used as an RGBA lookup table or through computations implemented as a shader program.

A color lookup table can be implemented as a texture with RGBA values, with original data values used as indices (or coordinates) in the table. When color parameters change, each value in the lookup table texture must be recomputed. However, this texture is usually much smaller than the original volumetric data stored as a texture.

Transfer functions can also be implemented as shader programs. The shader program receives texture coordinates for the volumetric data texture, performs a lookup to obtain the data value, and computes an RGBA value as output.

2.3 Shader Programs

A shader program is a set of GPU instructions that are executed repeatedly at a particular stage of the rendering pipeline. In the past few years, specifying shader instructions for GPUs has required loading primitive assembly instructions into the video card. More recently, high level shading languages such as Microsoft's HLSL (High Level Shading Language) and NVIDIA's Cg (C for Graphics) have been developed to provide an easier way to write shaders. HLSL is intended for use with DirectX, but Cg supports both OpenGL and DirectX.

Currently, the two types of shader programs are vertex and fragment shaders. Supplying a shader to a GPU overrides the default operations of a particular pipeline stage, requiring the shader itself to perform the appropriate operations if necessary.

2.3.1 Vertex Shaders

Vertex shaders are executed for each vertex during vertex processing. In this pipeline stage, a GPU would normally perform transform and lighting operations on each vertex. Per-vertex input consists of a position, a normal, a color, and one or more texture coordinates. Vertex shaders may also access constant values such as the current transform matrix, its inverse, or other values loaded into GPU registers. Per-vertex output consists of a homogenous position in clipping coordinates, a normal, a color, and one or more texture coordinates. Typical vertex shaders are used to apply lighting models or to displace vertices. A vertex shader can apply diffuse or specular Gouraud lighting or generate procedural terrain with a grid of vertices.

2.3.2 Fragment Shaders

Fragment shaders (or pixel shaders in DirectX terminology) are executed for each pixel during fragment processing. In this pipeline stage, a GPU would normally pass through a color or perform a texture lookup to obtain a color. Inputs to a fragment shader may include an interpolated color, an interpolated normal, and interpolated texture coordinates. Output consists of a single color and depth value. Typical fragment shaders are used to apply Phong lighting models, bumpmapping, or procedural textures.

2.3.3 Cg Shading Language

Cg is NVIDIA's attempt to create a portable standard for GPU shading languages. The Cg syntax is nearly identical to C with the addition of several built-in types and operations. New types include vectors and matrices. New operations include dot product, cross product, and vector "swizzling" operations. Cg programs may be precompiled or compiled at runtime using NVIDIA's Cg runtime library. *Shader profiles*, which specify GPU capabilities, are used to

place limitations on shaders during compilation. A GPU must support a given profile to run programs compiled for the profile. The Cg runtime library also provides support for binding variables to Cg shader inputs and loading shaders onto the GPU.

Shader inputs consist of varying parameters and uniform parameters. Varying parameters may have different values in each execution of the shader program and typically include vertices, normals, colors, or texture coordinates. Fragment shaders will receive interpolated values for varying parameters. Uniform parameters remain constant over a batch of vertices or pixels and typically include tranform matrices or references to textures.

The Cg runtime library provides support for assigning values to shader parameters. However, it is often unnecessary to explicitly assign these values if the shader provides *input semantics* to indicate other sources of the parameters. Other input sources may include OpenGL vertex, color, or texture coordinate calls.

Chapter 3: Methods

3.1 Tumor Detection

3.1.1 Dataset Representation

Datasets are produced by taking a series of MRI scans. A dataset can be interpreted as a set of 3D arrays (or volumes) of intensity values, a 4D array of intensity values, or a single volume containing vector values. Datasets may contain intensity values of either 8- or 12-bit precision, stored as one or two bytes. For 12-bit precision, the four most significant bits are unused.

3.1.2 Time Step Intensity Curves

A time step intensity curve is a graph of the intensity values across the time steps at a given position in the volume. For cancerous cells, this curve is assumed to rise sharply and fall smoothly.

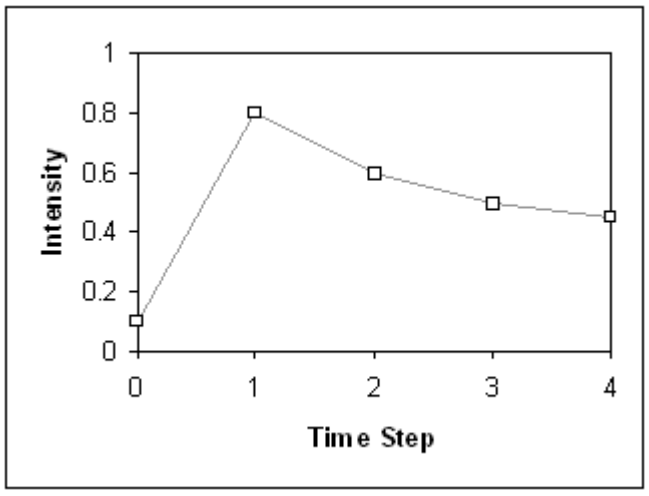


Figure 3.1. Intensity Curve of a Cancerous Cell

3.1.3 Confidence of Cancer Computation

To identify cancerous regions, confidence of cancer values in the range [0, 1] are computed for each value in the breast volume. Confidence computation requires four parameters: rise threshold, fall threshold, rise weight, and confidence threshold. The rise and fall thresholds are in the range [-1, 1] and rise weight and confidence threshold are in the range [0, 1].

For a given position in the volume, the time step intensity curve values are used as input to compute confidence. For these input values, rise is defined as the difference between the first time step value and the maximum value. Fall is defined as the difference between the maximum value and the last value. The following steps are used to compute confidence:

1. **Precomputations**. First, minimum and maximum intensity values are found within the entire dataset. Actual threshold values for the rise and fall thresholds are computed by mapping the original threshold parameters values from the range [-1, 1] to the range [min. density - density range, min. density + density range]. Next, rise and fall values are computed and stored for each position within the volume. Minimum and maximum rise and fall values are found for the next stage.
2. **Normalizing rise and fall**. Next, for each position within the volume, rise and fall values are mapped from the range [actual threshold, maximum] to the range [0, 1]. Values below the threshold are set to zero.
3. **Computing confidence**. Next, confidence values are computed for each location in the volume as a weighted sum of the normalized rise and fall values. Weights are rise weight and 1 - rise weight.

4. **Normalizing confidence**. Finally, confidence values are mapped from the range [confidence threshold, 1] to the range [0, 1]. Values below the threshold are set to zero.

3.2 Dataset Visualization

3.2.1 Scene Object Representation

Objects within the scene may include view-aligned slices, labeled axes, selection axes, and other objects. These objects are stored in a certain order and hierarchy using a simple scene graph library.

3.2.2 View-Aligned Slice Generation

Given a number of slices and a camera orientation and position, slices are generated by intersecting a unit cube with a series of planes orthogonal to the view direction. Planes are evenly spaced between the nearest and farthest cube vertices from the camera position. An intersection between a plane and the cube produces a set of unsorted vertices. The x, y, and z components of each vertex are in the range [-0.5, 0.5]. These vertices are sorted to produce a convex polygon. 3D texture coordinates are computed by offsetting the components of each vertex by 0.5.

3.2.3 Texture Generation

3D textures with RGBA components are generated by centering volume data within a 3D array of power-of-two dimensions. The red and green components of the texture are used to store intensity values (red represents the MSB and green represents the LSB). The blue component stores confidence values and the alpha value is unused. The user may specify

which volume within the dataset provides intensity values. Before rendering, the entire texture is loaded into video memory.

3.2.4 Display Controls

The software contains several controls to modify the way the volume is rendered. These controls are applied in the transfer function stage of the rendering process. *Overall opacity* controls the opacity of slices. *Confidence opacity* controls the weight placed on confidence values when computing the output color. *Number of slices* controls rendering precision. Higher numbers of slices will also cause the volume to appear more opaque and render slower. *Time step* controls which volume is displayed.

3.2.5 Transfer Functions

Transfer functions are implemented as Cg fragment shader programs. For each pixel, the shader program receives an interpolated 3D texture coordinate and a reference to a 3D texture. The following steps are implemented in a shader program to compute an output color:

1. A lookup is performed on the 3D texture using the interpolated 3D texture coordinate.
2. The intensity is computed from the red and green components. These components are interpreted as a two-byte, big endian value. Because of current fragment shader limitations, values above 255 are truncated.
3. Window and level parameters are applied to adjust the intensity.
4. A confidence factor is computed using the blue component and the confidence opacity parameter.

5. The output red value is computed as (intensity + confidence) factor. Blue and green values are computed as (intensity - confidence) factor. This produces a bright red color in areas of high confidence and shades of gray in areas of low confidence.
6. The output alpha value is set to the weighted sum of intensity and the confidence factor, multiplied by the opacity parameter. Different weights may be used for different transparency effects.

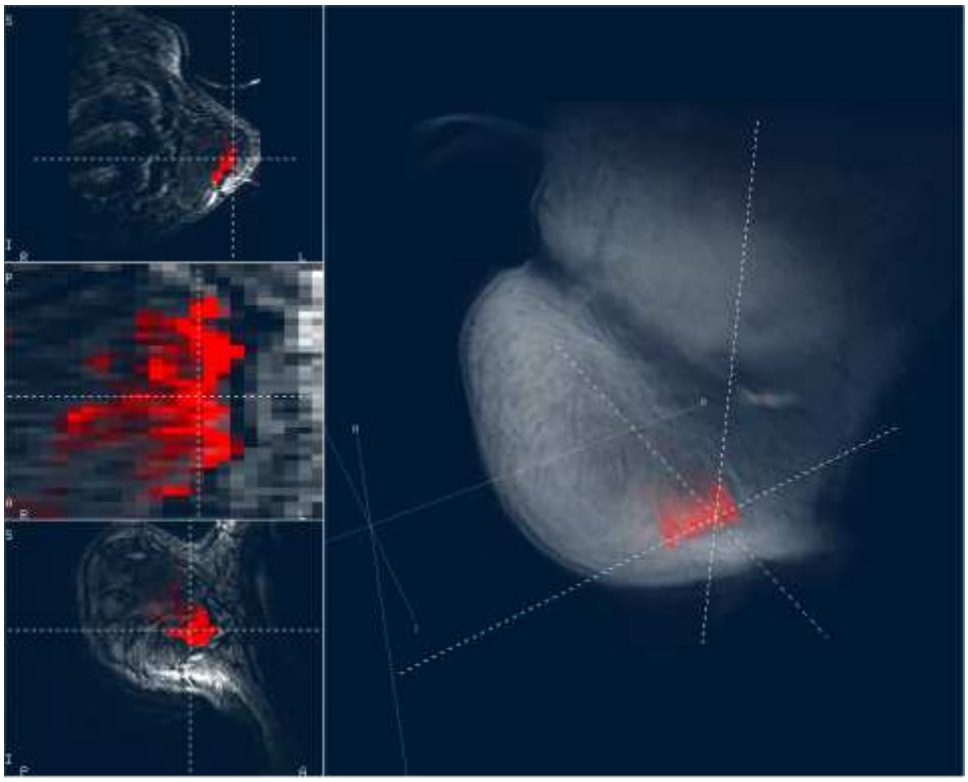


Figure 3.2. Transfer Function Coloring

Chapter 4: User Interface

4.1 Overview

This chapter contains a guide to the user interface. The interface is designed to be familiar to users by following genre conventions. The interface consists of three 2D, axis-aligned slice views, a single 3D view, and a control panel containing detection and visualization settings. The interface requires a minimum of 800x600 screen resolution. No additional dialog boxes or windows are used. Processing messages are logged to the console window for debugging purposes and to show progress to the user. Progress bars are not yet implemented. Controls are hierarchically organized, placed into tabs at the highest level and further organized into framed and labeled groups according to their function. This format is consistently applied to all controls.

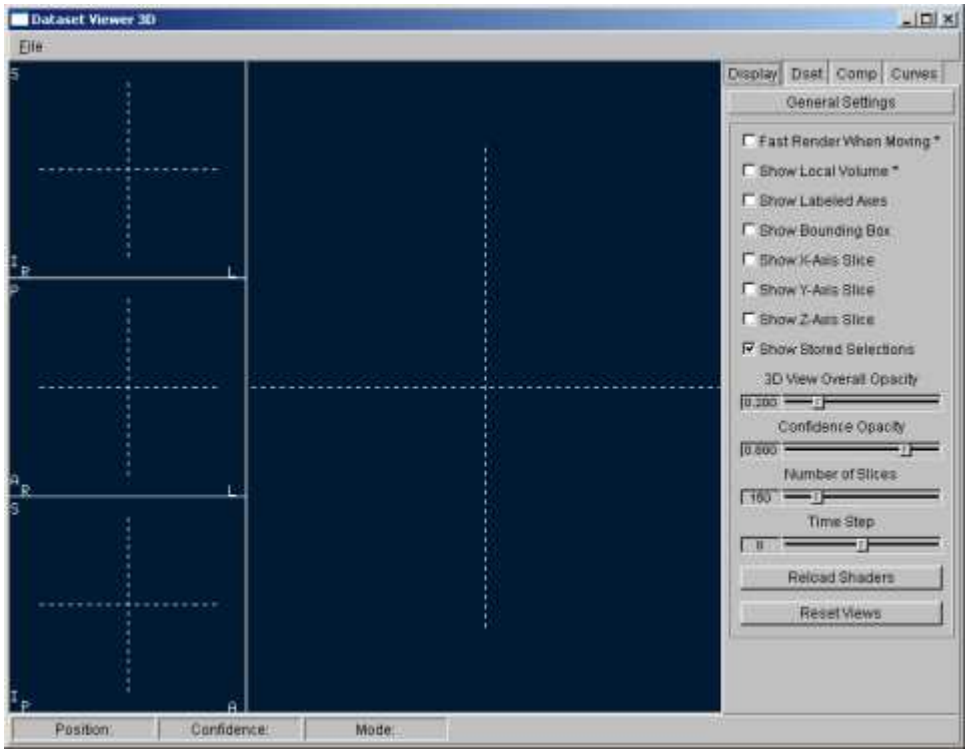


Figure 4.1. User Interface

To open a dataset, select **File->Open** and select the dataset file. Datasets must contain raw, big endian-order data with the appropriate file header (see Appendix D).

To view header information once a dataset is open, select **File->File Information**. The contents of the header will be displayed in the console window.

5.2 Two-Dimensional Views

The three 2D views show a single slice along the three principal axes. For a view along a given axis, dotted lines indicate the positions of the slices along the other two axes. The intersection of the dotted lines is the currently selected position. The following actions are available in these views:

- **Translation**. To translate the slice, click and drag the slice with the middle mouse button.
- **Scaling**. To scale the slice, rotate the mouse wheel.
- **Picking**. To pick a point, click on the point with the left mouse button.

5.3 Three-Dimensional View

The 3D view shows the entire dataset, optionally with labeled axes, a bounding box, highlighted axis-aligned slices, and stored selections. Highlighted axis-aligned slices correspond to the axis-aligned slices in the 2D views, appearing as brighter slices with increased opacity. Stored selections appear as wire cubes. The following actions are available in this view:

- **Translation**. To translate the volume, click and drag the volume with the middle mouse button.

- **Rotation**. To rotate the volume about the vertical or horizontal axes, click and drag the volume with the right mouse button.
- **Scaling**. To scale the volume, rotate the mouse wheel.
- **Picking**. To pick a stored selection, click the selection with the left mouse button.

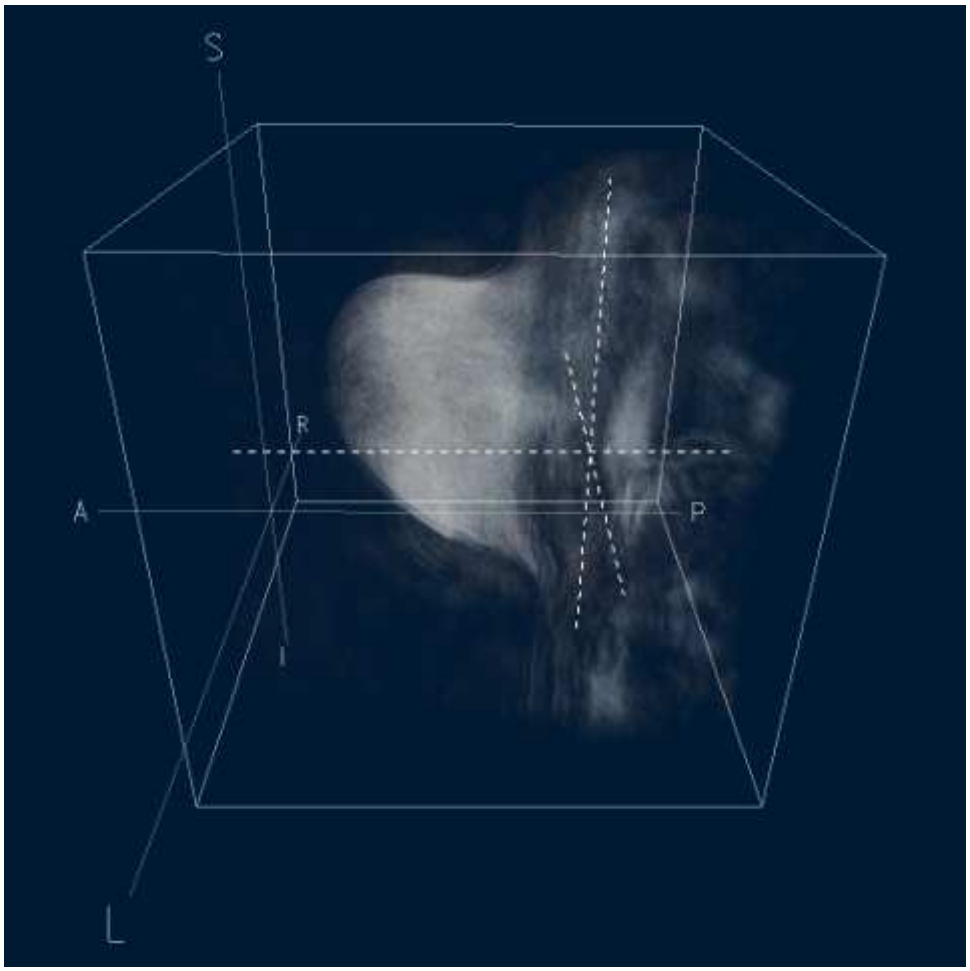


Figure 4.2. 3D View

5.4 Control Panel

5.4.1 Display Tab

This tab contains settings that determine which objects are visible in the 3D view and how slices are displayed.

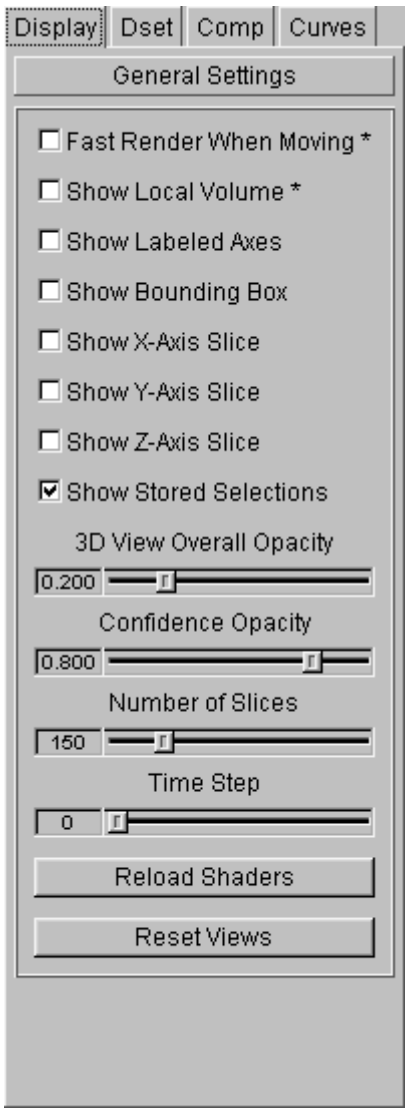


Figure 5.1. Display Tab

Fast Render When Moving. This option determines whether the number of slices is reduced when the volume is being translated, scaled, or rotated. This will render faster at the cost of image precision.

Show Labeled Axes. This option toggles the visibility of a set of labeled axes.

Show Bounding Box. This option toggles the visibility of the bounding box around the volume.

Show X-Axis Slice, **Show Y-Axis Slice**, **Show Z-Axis Slice**. These options toggle the visibility of slices within the 3D view that correspond to the axis-aligned slices in the 2D views.

Show Stored Selections. This option toggles the visibility of stored curve positions. These positions will appear as wire cubes in the 3D view.

3D View Overall Opacity. This slider determines the opacity of 3D view.

Confidence Opacity. This slider determines the opacity of confidence.

Number of Slices. Increasing this value will increase the precision of the 3D view at the cost of rendering speed.

Time Step. This slider determines which time step volume is used for rendering. Changing this will require the application to update the volume texture.

Reload Shaders. This button reloads and recompiles shader source files.

Reset Views. This button resets all views to their original position, orientation, and scaling.

5.4.2 Dataset Tab

This tab contains settings that determine the way a dataset is interpreted. Depending on the way a scan is taken, these settings may need to be modified to display the dataset properly.

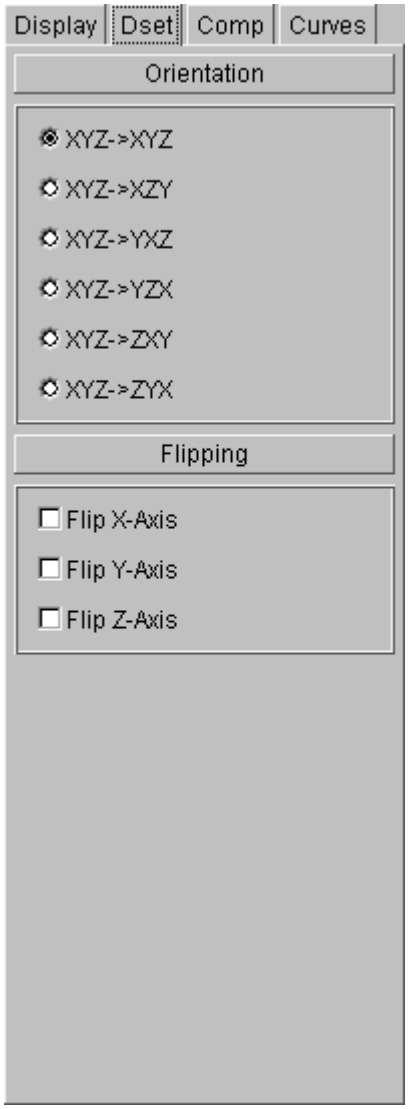


Figure 5.2. Dataset Tab

Orientation. These options may be used to orient a dataset to the appropriate axes.

Flipping. These checkboxes determine whether the dataset is flipped along a given axis.

5.4.3 Computation Tab

This tab contains confidence and tumor volume computation settings.

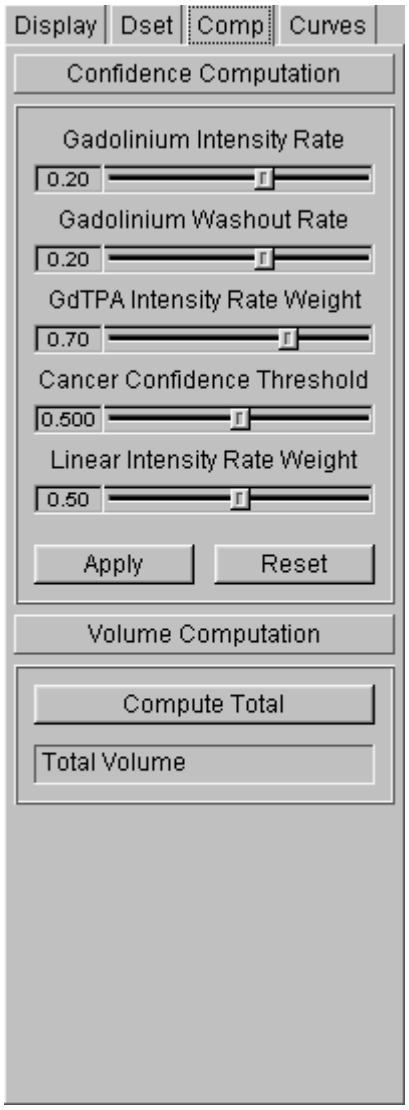


Figure 5.3. Computation Tab

Gadolinium Intensity Rate. This slider determines the *minimum rise* parameter.

Gadolinium Washout Rate. This slider determines the *minimum fall* parameter.

GdTPA Intensity Rate Weight. This slider determines the *rise weight* parameter.

Cancer Confidence Threshold. This slider determines the *confidence threshold* parameter.

Apply. This button will recompute confidence using the current settings. Only the minimum computations necessary will occur, so changing intensity or washout rates will cause computations to take longer than only changing the confidence threshold.

Reset. This button will reset sliders to their original values.

5.4.4 Curves Tab

This tab shows the curve at the selected position and a listing of stored curves.

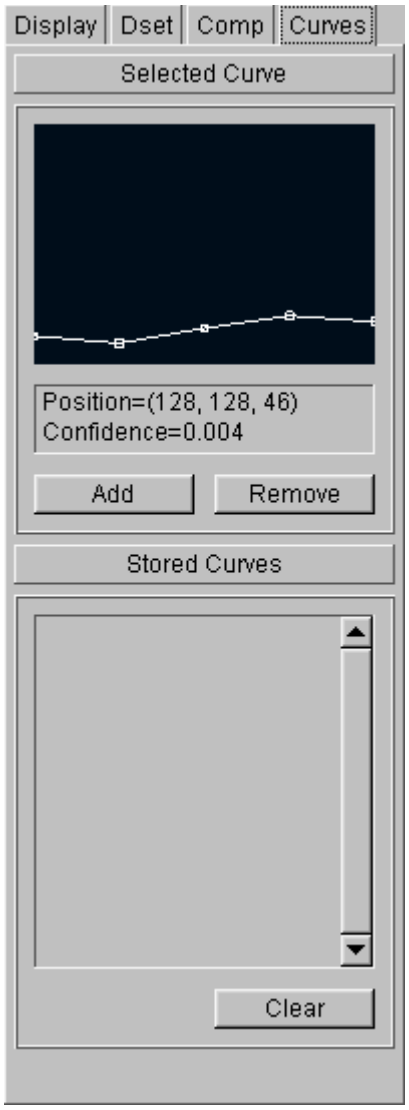


Figure 5.4. Curves Tab

Add. This button stores the currently selected point. Stored curves will appear as small wire cubes in the 3D view. They may be selected by clicking on them in the 3D view or clicking on the entry in the curves list.

Remove. This button removes the selected point from the list of stored curves.

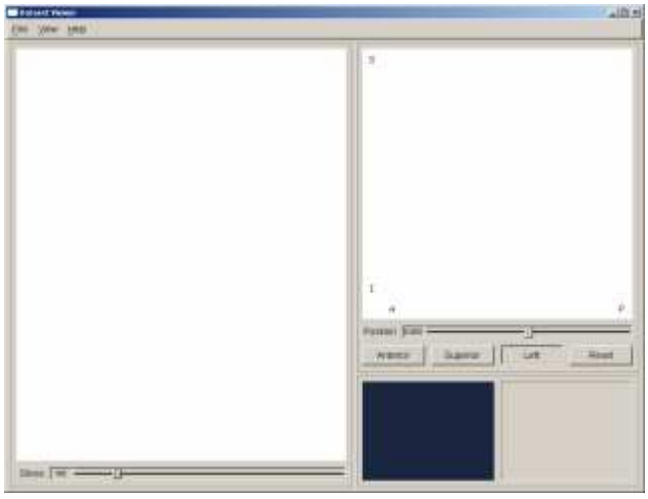
Clear. This button removes all selections from the list of stored curves.

Chapter 5: Experimentation

During the development process, the software methods, interface, and architecture changed many times. This chapter describes some of the changes.

5.1 User Interface

The original interface featured a 3D view, an axis-aligned 2D view, a time step curve plot, and a floating control panel. Most visualization options were located in menus. The visualization color scheme had a white background and a dark blue foreground, contrary to existing MRI software. These issues were resolved in the current version of the software. Settings were consolidated into a single control panel in the main window, the coloring scheme was revised to resemble existing software, and three axis-aligned 2D views replaced the single view.



Old Interface

Before reading a dataset, it must be converted from a series of 2D MR data files (of 12-bit precision) or JPEG images (of 8-bit precision) to a single file containing a header and raw

data (see Appendix C for header specifics). To test conversions and to view datasets on systems without advanced hardware, a lightweight 2D viewer was developed. This application featured window, level, zoom, and translation controls.

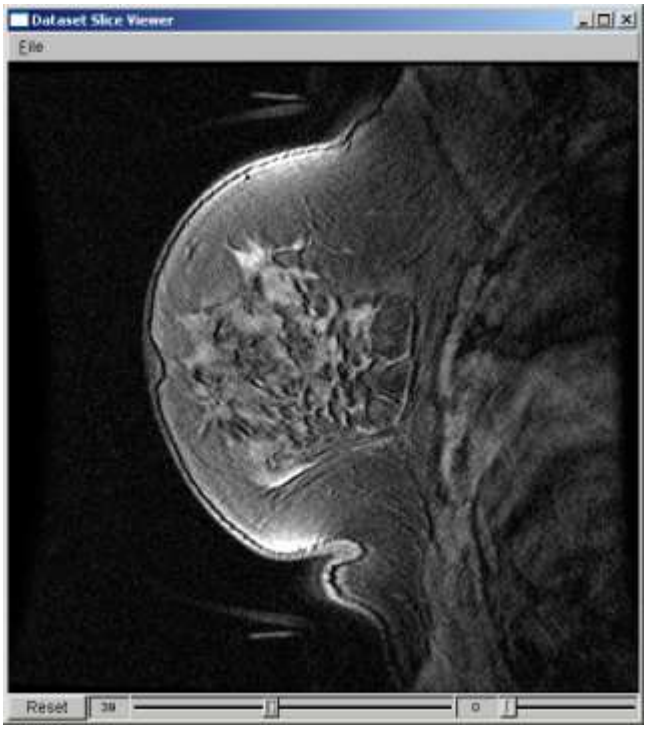


Figure 5.1. Two-Dimensional Viewer

5.2 Tumor Detection

Initially, curves were assumed to rise very quickly for cancerous cells. Rise was defined to be the difference between the first and second intensity values in the curve. However, this was not true for all datasets. Some datasets exhibited a delayed rise in intensity curves, and rise was redefined to be the difference between the first value and the maximum value.

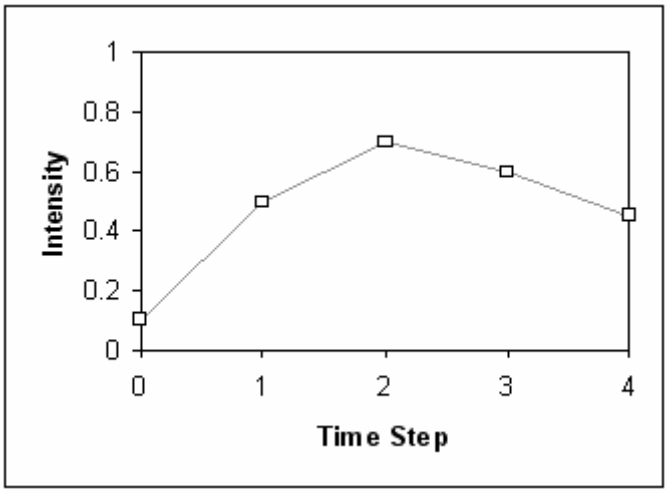


Figure 5.2. Delayed Rise

Blood vessels often produce high confidence values because of their high initial rise. As a result, the software assigned blood vessels higher confidence values than cancerous areas. This will be resolved in future versions of the software by adding a *maximum intensity rate* control.

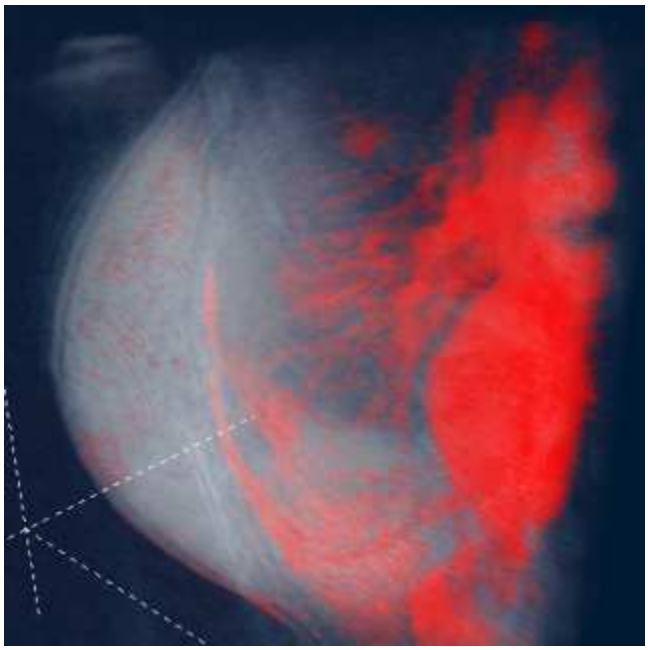


Figure 5.3. Blood Vessels and Heart

5.3 Visualization

Differences in breast tissue and scanning conditions can cause the resulting datasets to vary greatly. Some datasets contained low intensity values or lacked contrast. To some extent, the opacity and slice controls enhanced the appearance of these datasets. Window and level controls worked well in the 2D viewer, but they could not be implemented properly in the 3D viewer due to shader constraints.

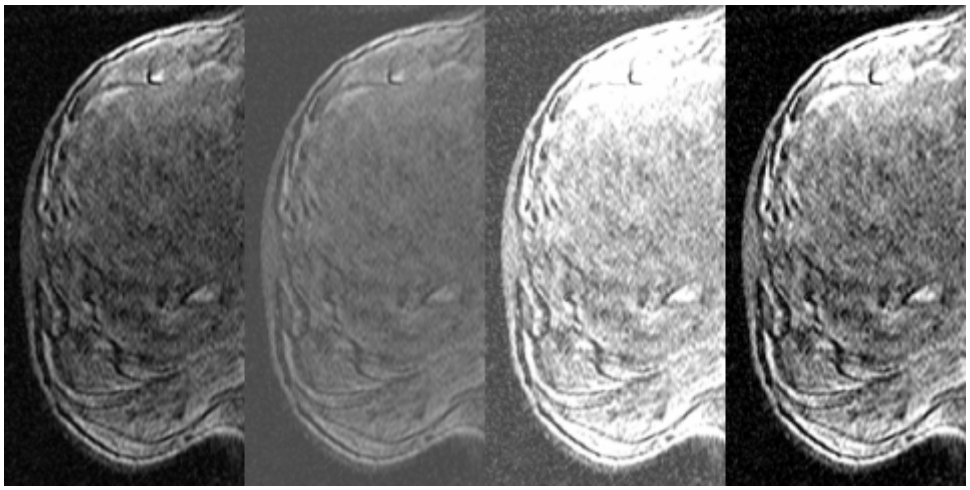


Figure 5.4. Window and Level Settings

Because the software requires entire datasets to be loaded into video memory, datasets either had to be subsampled or textures had to be compressed to fit into video memory. Subsampling did not cause critical data to be lost, but it required an additional step in the process of converting datasets to a form the software could read. Texture compression often caused noticeable artifacts to appear when rendering.

Chapter 6: Conclusion

The software provides an intuitive interface for navigating breast MR scans and locating cancerous regions. The interface, color scheme, and controls were designed to be as similar as possible to existing software. The interactivity of the software makes it appealing as a flexible detection and visualization tool. However, there are still many features that can be implemented to improve and expand the capabilities of the software.

Additional controls are necessary to isolate certain features. A maximum rise control would eliminate many of the blood vessels with high initial rises. A method of computing and thresholding a metric of rise or fall linearity could eliminate curves that do not exhibit a steady rise or fall.

Features with distinct characteristic curves could be segmented by identifying and cataloguing confidence parameter settings. Confidence values could be computed for certain features using these settings. These preset parameters could be loaded or saved.

Volume rendering should employ a "brick-rendering" method. Rather than loading an entire dataset into video memory, a series of smaller divisions of the dataset would be loaded and rendered. This removes the limitation of video memory from the rendering process.

Shaders should include gradient values in color computations. Because of instruction limits in the fragment shader profile that is currently used by the software, computations involving gradient values could not significantly enhance the visualization. As more advanced hardware becomes available, the use of gradient values in shaders should be revisited.

References

[1] Akenine-Moller, Tomas and Eric Haines. *Real-Time Rendering*. Natick: A K Peters, Ltd., 2002.

[2] Fast Light Toolkit Documentation. http://www.fltk.org.

[3] Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. New York: Addison-Wesley, 1995.

[4] Hadwiger et al. High-Quality Graphics on Consumer PC Hardware. *SIGGRAPH 2002 Course Notes*, 2002.

[5] Kniss, Joe, Gordon Kindlmann, and Charles Hansen. Multi-Dimensional Transfer Functions for Interactive Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 2002.

[6] NVIDIA Corporation. Cg Documentation. http://developer.nvidia.com.

Appendix A: Implementation Overview

Dependencies

The following external libraries are used in this application:

- **OpenGL**: Cross-platform graphics library.
- **ExtGL**: Extension loading library for Windows.
- **GLUT** (GL Utility Toolkit): Provides a platform-independent wrapper for OpenGL applications. Also provides functions to render simples primitives such as wireframe or solid cubes or spheres.
- **FLTK** (Fast Light Toolkit): Cross-platform GUI toolkit with OpenGL support.

Conventions

All code adheres to the following conventions:

File Conventions

- Header files (.h extension) contain only declarations.
- Inline files (.inl extension) contain only inline function definitions. They are included at the end of header files.
- Source files (.cpp extension) contain source code that should not or cannot be defined inline.

Stylistic Conventions

- Lines do not exceed 80 characters.
- Indentations are four spaces.
- Comments are used only to clarify less readable code or blocks of code. Excessive comments can otherwise simple code.

Naming Conventions

- Class names, enumeration types, typedefs, and functions start with a capital letter.
- Macros and constants are declared in all caps and may contain underscores.

- Variables start with a lowercase letter.
- Hungarian notation is only used to differentiate variables of different types that could otherwise have the same name.
- All distinct words within names start with a capital letter. To avoid ambiguity, adjacent capital letters are avoided.
- Accessors are named after the variable they return or modify. Getting and setting accessors are not differentiated by name.

Appendix B: Scene Library Design

Math Classes

Math classes are used by the framework primarily for transformations, intersection testing, and collision testing.

All math classes share the following traits:

- The components of each class are of type **float**.
- Each class includes operations for reading and writing to Standard Library streams.
- Angles are specified in degrees.

Math classes include vectors (**Vector2**, **Vector3**, **Vector4**), matrices (**Matrix3**, **Matrix4**), **Quaternion**, **Ray3**, and **Plane**.

Vectors

Vector classes include vectors **Vector2**, **Vector3**, and **Vector4**, which represent two-, three-, and four-vectors, respectively. They implement basic arithmetic operations and dot product and cross product operations.

```
class Vector3
{
public:
    real x, y, z;

public:
    Vector3();
    Vector3(real x, real y, real z);
    Vector3(const real v[3]);
    Vector3(const Vector3 &v);

    // accessors
    const real& operator[](int i) const;
    real& operator[](int i);

    // assignment
    Vector3& operator=(const Vector3& v);

    // boolean operations
    bool operator==(const Vector3& v) const;
    bool operator!=(const Vector3& v) const;

    // arithmetic operations
    Vector3 operator*(real k) const;
    friend Vector3 operator*(real k, const Vector3& v);
    Vector3 operator/(real k) const;
    Vector3 operator+(const Vector3& v) const;
    Vector3 operator-(const Vector3& v) const;
    Vector3 operator-() const;

    Vector3& operator*=(real k);
    Vector3& operator/=(real k);
    Vector3& operator+=(const Vector3& v);
    Vector3& operator-=(const Vector3& v);
```

```
    // vector operations
    real Dot(const Vector3& v) const;
    Vector3 Cross(const Vector3& v) const;
    real Length() const;
    real LengthSquared() const;
    void Normalize();

    // random vector generation
    void Random(const Vector3& vMin, const Vector3& vMax);

    // stream operations
    friend std::ostream& operator<<(
        std::ostream& stream, const Vector3& v);
    friend std::istream& operator>>(
        std::istream& stream, Vector3& v);

    // static functions
    static const Vector3& Zero();
    static const Vector3& Identity();
};
```

Matrices

Matrix classes include **Matrix3** and **Matrix4**, which represent 3x3 and 4x4 matrices, respectively. They implement basic arithmetic operations and vector and matrix multiplications.

```
class Matrix3
{
protected:
    real m[9];

public:
    Matrix3();
    Matrix3(real m00, real m01, real m02,
        real m10, real m11, real m12,
        real m20, real m21, real m22);
    Matrix3(real m[9]);
    Matrix3(const Vector3& r0,
        const Vector3& r1,
        const Vector3& r2);
    Matrix3(const Matrix3& m);

    // accessors
    const real& operator[](int i) const;
    const Vector3& Row(int i) const;
    real& operator[](int i);
    Vector3& Row(int i);

    // assignment
    Matrix3& operator=(const Matrix3& m);

    // arithmetic operations
    Vector3 operator*(const Vector3& v) const;
    Matrix3 operator*(const Matrix3& m) const;

    // matrix operations
    void Transpose();

    // stream operations
    friend std::ostream& operator<<(
        std::ostream& stream, const Vector3& v);
    friend std::istream& operator>>(
        std::istream& stream, Vector3& v);

    // static functions
    static const Matrix3& Identity();
};
```

Quaternion

A quaternion is an extension of a complex number, containing one real component and three imaginary components. In 3D rendering, quaternions are used for efficient rotations and spherical linear interpolation (a method of interpolating between orientations, often used for camera motion and animation).

The **Quaternion** class implements basic arithmetic operations and conversions. It includes conversions to and from Euler angle orientations, axis and angle orientations, and 3x3 matrix orientations.

```
class Quaternion
{
public:
    real w, x, y, z;

public:
    Quaternion();
    Quaternion(real w, real x, real y, real z);
    Quaternion(const Quaternion &q);

    // arithmetic operations
    Quaternion operator*(const Quaternion& q) const;
    Quaternion operator*(const Vector3& v) const;
    Quaternion operator+(const Quaternion& q) const;
    Quaternion operator-() const;

    Quaternion& operator+=(const Quaternion& q);

    // conversions
    void ToEuler(Vector3& vAxes) const;
    void ToMatrix(Matrix3& m) const;
    void ToAxisAngle(Vector3& vAxis, real& angle) const;

    void FromEuler(const Vector3& vAxes);
    void FromMatrix(const Matrix3& m);
    void FromAxisAngle(const Vector3& vAxis, const real& angle);

    // quaternion operations
    real Length() const;
    real LengthSquared() const;
    void Normalize();

    // static functions
    static const Quaternion& Identity();
};
```

Rays

Ray classes include **Ray3**, which represents a ray in three-space. It is stored as an origin vector and a direction vector.

```
class Ray3
{
protected:
    Vector3 vOrigin, vDirection;

public:
    Ray3();
    Ray3(Vector3 vOrigin, Vector3 vDirection);
    Ray3(const Ray3& ray);
```

```
    // accessors
    const Vector3& Origin() const;
    const Vector3& Direction() const;

    Vector3& Origin();
    Vector3& Direction();

    void Origin(const Vector3& v);
    void Direction(const Vector3& v);

    // assignment
    Ray3& operator=(const Ray3& ray);

    // ray operations
    void ApplyTransformMatrix(const Matrix4& m);
};
```

Plane

The **Plane** class represents a plane in three-space, stored as a plane normal and a plane constant. The plane constant represents the distance from the origin to the plane.

```
class Plane
{
protected:
    Vector3 vNormal;
    real constant;

public:
    Plane();
    Plane(Vector3 vNormal, real constant);
    Plane(const Plane& plane);

    // accessors
    const Vector3& Normal() const;
    const real& Constant() const;

    Vector3& Normal();
    real& Constant();

    void Normal(const Vector3& v);
    void Constant(real constant);

    // assignment
    Plane& operator=(const Plane& plane);

    // plane operations
    real Distance(const Vector3& vPoint) const;
};
```

Supplemental Classes

Supplemental classes include base classes and classes that are used internally by visitor and node classes. They include **Object**, **Reference**, **MatrixStack**, **Frustum**, and **RenderTree**.

Object

The **Object** base class supports reference counting. Derived classes include all bounding volumes, nodes, cameras, and visitors.

```
class Object
{
protected:
    int refCount;

public:
    Object();
    virtual ~Object();

    void IncRef();
    void DecRef();
};
```

Reference

The **Reference** class acts as a smart pointer that manages a reference-counted **Object**.

```
template <class T>
class Reference
{
private:
    T* ptr;

public:
    Reference();
    Reference(const T* ptr);
    Reference(T* ptr);
    Reference(const T& ref);
    Reference(T& ref);
    Reference(const Reference& ref);
    ~Reference();

    void operator=(T* ptr);
    void operator=(const Reference& ref);

    T* operator->();
    T& operator*();

    const T* operator->() const;
    const T& operator*() const;

    operator T*();
    operator const T*();

    T* Pointer();

    bool operator!=(T* ptr) const;
    bool operator==(const Reference& ref) const;
};
```

MatrixStack

The **MatrixStack** class stores a stack of 4x4 matrices and their inverses. For each stack entry, a matrix, inverse matrix, concatenated matrix, and concatenated inverse may be accessed without requiring computation. Inverse concatenations are typically used during traversals to convert world coordinates to object coordinates.

```
class MatrixStack
{
public:
    struct Entry
    {
```

```
        Matrix4 mEntry,
                mEntryInverse,
                mCurrent,
                mCurrentInverse;
    };

protected:
    std::vector<Entry> entries;
    int entryCount;

public:
    MatrixStack();

    // accessors
    const Matrix4& CurrentMatrix() const;
    const Matrix4& CurrentInverseMatrix() const;
    bool Empty() const;

    // operations
    void Push(const Matrix4 &m, const Matrix4 &mInverse);
    void Pop();
};
```

Frustum

The **Frustum** class consists of a set of planes that define a frustum or volume. Typically, a camera projection defines a view frustum consisting of six planes: left, right, top, bottom, near, and far. This view frustum represents the space in the scene that is visible to the camera. Frustums are also used for advanced hidden surface removal (HSR) algorithms such as occlusion culling and portal engines.

```
class Frustum
{
protected:
    std::vector<Plane> planeArray;

public:
    // accessors
    const Plane& operator[](int index) const;
    Plane& operator[](int index);

    int PlaneCount() const;
    void Resize(int planeCount);
    void Clear();

    // frustum operations
    void ApplyTransformMatrix(const Matrix4& m);
};
```

RenderTree

The **RenderTree** class is a simplified, temporary tree structure designed to hold the culled contents of scene. Each non-leaf represents a **Transform** node. Sortable leaf nodes represent sortable **State** nodes. Unsortable leaf nodes represent renderable nodes. The **RenderVisitor** class traverses a **RenderTree** object.

```
class RenderTree
{
protected:
    Node* sceneNode;
```

```
    bool sortable;
    std::vector<RenderTree*> trees;

public:
    RenderTree();
    RenderTree(Node& node, bool sortable = false);
    virtual ~RenderTree();

    // accessors
    const Node& SceneNode() const;
    Node& SceneNode();
    void SceneNode(Node& node);
    bool Sortable() const;
    void Sortable(bool state);

    // traversal operations
    void Accept(Visitor& visitor);

    // tree operations
    int Size() const;
    RenderTree& Child(int index) const;
    void Add(RenderTree& tree);
    void Clear();
    int TotalSize() const;
};
```

Bounding Volumes

A bounding volume is used as an approximation of the volume of space a node represents. Bounding volumes are usually less computationally expensive and complicated to deal with than the actual geometry contained within a node.

Bounding volumes are typically used to efficiently traverse a scene. Large parts of a scene may be completely skipped by performing bounding volume tests. The following are common traversals that use bounding volumes:

- **View Culling**. This is a method of reducing the amount of geometry that is rendered. Bounding volumes are tested against the view frustum to determine whether they are visible. If a bounding volume is visible, its children are tested. More complex algorithms, such as occlusion culling or portal engines, may use multiple frustums.

- **Picking**. Picking occurs when a user attempts to select a visible object. Because the user sees a 3D scene projected to a 2D surface, a point on the 2D surface represents a ray directed into the 3D scene. Bounding volumes are used for simple intersection tests with this ray. If the ray intersects a bounding volume, child nodes are tested. A leaf node that intersects the ray represents an object the user has picked.

- **Collision Detection**. The purpose of this process is to locate collisions between geometry that occur over a given time interval. Detecting collisions with actual geometry can be computationally expensive, so bounding volume hierarchies are used to quickly test for potential collisions. If an early collision test fails, no additional tests are required.

The **Volume** base class, derived from **Object**, contains redefineable functions to test for intersections with **Ray3**, **Plane**, **Sphere**, **AlignedBox**, and **OrientedBox** objects. Derived classes include **Sphere**, **AlignedBox**, and **OrientedBox**.

```
class Volume : public Object
{
protected:
    Volume *relativeVolume;
    int failedPlaneIndex;

public:
    Volume();
    virtual ~Volume();

    // accessors
    const Volume& RelativeVolume() const;
    void RelativeVolume(const Volume& volume);

    int FailedPlane() const;
    void FailedPlane(int i);

    // cloning
    virtual Volume& Clone() const = 0;

    // bound operations
    virtual void Update(const Matrix4& mCurrentTransform);
    virtual bool TestVolume(const Volume& volume) const = 0;
    virtual bool TestPoint(const Vector3& vPoint) const;
    virtual bool TestRay(const Ray3& ray) const;
    virtual bool TestHalfSpace(const Plane& plane) const;
    virtual bool TestSphere(const Sphere& sphere) const;
    virtual bool TestAlignedBox(const AlignedBox& box) const;
    virtual bool TestOrientedBox(const OrientedBox& box) const;
};
```

Sphere

The **Sphere** class represents a sphere, defined by a three-vector center position and a radius.

```
class Sphere : public Volume
{
protected:
    Vector3 vCenter;
    real radius;

public:
    Sphere();
    Sphere(real radius, const Vector3& vCenter = Vector3::Zero());
    Sphere(const Sphere& sphere);

    // accessors
    const Vector3& Center() const;
    real Radius() const;

    Vector3& Center();
    real& Radius();

    void Center(Vector3& v);
    void Radius(real radius);

    // cloning
    virtual Volume& Clone() const;

    // bound operations
    virtual void Update(const Matrix4& mCurrentTransform);
    virtual bool TestVolume(const Volume& volume) const;
    virtual bool TestPoint(const Vector3& vPoint) const;
```

```
    virtual bool TestRay(const Ray3& ray) const;
    virtual bool TestHalfSpace(const Plane& plane) const;
    virtual bool TestSphere(const Sphere& sphere) const;
    virtual bool TestAlignedBox(const AlignedBox& box) const;
    virtual bool TestOrientedBox(const OrientedBox& box) const;
};
```

AlignedBox

The **AlignedBox** class represents a box in three-space aligned to the world axes. This assumption makes intersection and collision testing simpler and faster. It is stored as a three-vector center position and extent.

```
class AlignedBox : public Volume
{
protected:
    Vector3 vCenter, vExtent;

public:
    AlignedBox();
    AlignedBox(const Vector3& vExtent,
        const Vector3& vCenter = Vector3::Zero());
    AlignedBox(const AlignedBox& box);

    // accessors
    const Vector3& Center() const;
    const Vector3& Extent() const;

    Vector3& Center();
    Vector3& Extent();

    void Center(const Vector3& v);
    void Extent(const Vector3& v);

    // cloning
    virtual Volume& Clone() const;

    // bound operations
    virtual void Update(const Matrix4& mCurrentTransform);
    virtual bool TestVolume(const Volume& volume) const;
    virtual bool TestPoint(const Vector3& vPoint) const;
    virtual bool TestRay(const Ray3& ray) const;
    virtual bool TestHalfSpace(const Plane& plane) const;
    virtual bool TestSphere(const Sphere& sphere) const;
    virtual bool TestAlignedBox(const AlignedBox& box) const;
    virtual bool TestOrientedBox(const OrientedBox& box) const;
};
```

OrientedBox

The **OrientedBox** class represents a box in three-space aligned to anarbitrary axes. It is a more generalized version of an **AlignedBox**, but it makes sacrifices in complexity, speed, and memory. It is stored as a three-vector center position and extent and a quaternion orientation.

```
class OrientedBox : public AlignedBox
{
protected:
    Vector3 vCenter, vExtent;
    Matrix3 mOrientation;

public:
    OrientedBox();
```

```
    OrientedBox(const Vector3& vExtent,
        const Matrix3& mOrientation = Matrix3::Identity(),
        const Vector3& vCenter = Vector3::Zero());
    OrientedBox(const OrientedBox& box);

    // accessors
    const Matrix3& Orientation() const;
    Matrix3& Orientation();
    void Orientation(const Matrix3& m);

    // cloning
    virtual Volume& Clone() const;

    // bound operations
    virtual void Update(const Matrix4& mCurrentTransform);
    virtual bool TestVolume(const Volume& volume) const;
    virtual bool TestPoint(const Vector3& vPoint) const;
    virtual bool TestRay(const Ray3& ray) const;
    virtual bool TestHalfSpace(const Plane& plane) const;
    virtual bool TestSphere(const Sphere& sphere) const;
    virtual bool TestAlignedBox(const AlignedBox& box) const;
    virtual bool TestOrientedBox(const OrientedBox& box) const;
};
```

Nodes

The **Node** base class represents the most basic type of node in the scene graph.

```
class Node : public Object
{
public:
    enum NodeFlags
    {
        NF_ENABLED  = 0x01,
        NF_CHANGED  = 0x02,
        NF_STATIC   = 0x04,
        NF_PICKABLE = 0x08,
    };

protected:
    unsigned int flags;
    Volume *bounds;
    void *userData;
    char *name;

public:
    Node();
    virtual ~Node();

    // accessors
    bool HasBounds() const;
    const Volume& Bounds() const;
    unsigned int Flags() const;
    const void* UserData() const;
    const char* Name() const;

    Volume& Bounds();
    unsigned int& Flags();
    void* UserData();
    char* Name();

    void Bounds(const Volume& relativeVolume);
    void Flags(unsigned int flags);
    void UserData(void *userData);
    void Name(char *name);

    // flag access shortcuts
    bool Enabled() const;
    bool Static() const;
```

```
    bool Pickable() const;

    void Enabled(bool state);
    void Static(bool state);
    void Pickable(bool state);

    // intersection calculations
    virtual bool TestRay(const Ray3& ray) const;
    virtual bool ComputeRayIntersection(real& tRay,
        Vector2& vCoord, const Ray3& ray) const;

    // node operations
    virtual void Accept(Visitor& visitor);
    virtual void Apply(Visitor& visitor);
    virtual void Update(Visitor& visitor, real interval);
};
```

Node contains several flags:

- **Enabled**. This flag determines whether a node is traversed.
- **Changed**. This flag indicates that a node has changed in an update.
- **Static**. This flag indicates that a node will never change. This will prevent it from being traversed in updates.
- **Pickable**. This flag indicates that a node can be picked if it passes intersection tests.

Node contains several virtual functions:

- **Apply(Visitor)**. This function performs a rendering operation. The **Node** class defines this function to do nothing, but subclasses will redefine it depending on their purpose. For example, the **Geometry** class will define it to supply geometry primitives to render while a **Transform** class will push a transformation matrix to the visitor's matrix stack.
- **Update(Visitor, Interval)**. This function updates a node over a given time interval. The **Node** class defines this function to do nothing, but dynamic subclasses may redefine it. Some nodes, such as level-of-detail (LOD) nodes, will not use the given time interval.
- **Accept(Visitor)**. This function identifies the node type to the **Visitor** by calling an appropriate **Visitor** function for the node. The **Visitor** function performs an appropriate operation given the type. For example, a **State** node would call *VisitState* and a **Transform** node would call *VisitTransform*. Depending on the subclass of **Visitor**, the *VisitTransform* function may call *Apply*, *Update*, or neither for the node. A visitor may continue traversal by calling *Accept* for each child node. This method of identifying two unknown types is called "double dispatch." The scene graph uses this mechanism to implement the "Visitor" design pattern.

Derived classes include **Geometry**, **Light**, **State**, and **Compound**.

Geometry

The **Geometry** base class represents a set of renderable primitives. Geometry is stored as an array of vertices and an array of indexes to the vertex array. The **Geometry** class does not define the type of primitives it represents.

```
class Geometry : public Node
{
public:
    enum NormalModeEnum
    {
        GEO_NM_NONE,
        GEO_NM_SINGLE,
        GEO_NM_FACE,
        GEO_NM_VERTEX,
        GEO_NM_FACEVERTEX
    };

    enum ColorModeEnum
    {
        GEO_CM_NONE,
        GEO_CM_SINGLE,
        GEO_CM_FACE,
        GEO_CM_VERTEX,
        GEO_CM_FACEVERTEX
    };

    enum TextureModeEnum
    {
        GEO_TM_NONE,
        GEO_TM_2DCOORDS,
        GEO_TM_3DCOORDS
    };

protected:
    // OpenGL display list
    unsigned int displayList;

    // rendering settings
    bool renderWireframe;
    NormalModeEnum normalMode;
    ColorModeEnum colorMode;
    TextureModeEnum textureMode;

    // vertex properties
    Reference<VertexArray>      vertexArray;
    Reference<NormalArray>      normalArray;
    Reference<TexCoord2Array>   texCoord2Array;
    Reference<TexCoord3Array>   texCoord3Array;
    Reference<ColorArray>       colorArray;
    Reference<IndexArray>       indexArray;

public:
    Geometry();

    // array accessors
    const Vector3& Vertex(int i) const;
    const Vector3& Normal(int i) const;
    const Vector2& TexCoord2(int i) const;
    const Vector3& TexCoord3(int i) const;
    const Vector3& Color(int i) const;
    const unsigned int Index(int i) const;

    Vector3& Vertex(int i);
    Vector3& Normal(int i);
    Vector2& TexCoord2(int i);
    Vector3& TexCoord3(int i);
    Vector3& Color(int i);
    unsigned int Index(int i);
```

```
    void Vertices(Reference<VertexArray> ref);
    void Normals(Reference<NormalArray> ref);
    void TexCoords2(Reference<TexCoord2Array> ref);
    void TexCoords3(Reference<TexCoord3Array> ref);
    void Colors(Reference<ColorArray> ref);
    void Indexes(Reference<IndexArray> ref);

    // display list accessors
    unsigned int DisplayList() const;
    void DisplayList(unsigned int displayList);

    // render mode accessors
    bool WireFrame() const;
    void WireFrame(bool state = true);

    NormalModeEnum NormalMode() const;
    ColorModeEnum ColorMode() const;
    TextureModeEnum TextureMode() const;

    void NormalMode(NormalModeEnum mode);
    void ColorMode(ColorModeEnum mode);
    void TextureMode(TextureModeEnum mode);

    // rendering operations
    void CreateDisplayList();
    virtual void Render() const;

    // node operations
    virtual void Accept(Visitor &visitor);
    virtual void Apply(Visitor& visitor);

protected:
    // rendering operations
    virtual void RenderVertex(int i) const;
    virtual void RenderPolygon(int iPoly) const;
};
```

Optional arrays include color, normal, 2D texture coordinate, or 3D texture coordinate arrays. 2D and 3D texture coordinate arrays may not be used simultaneously. Array specification has five modes:

- **None**: No values are supplied and the array is ignored.
- **Single**: A single value is supplied for the entire geometry.
- **Face**: Values are supplied for each face.
- **Vertex**: Values are supplied for each vertex.
- **FaceVertex**: Values are supplied for each vertex of each face. This allows faces sharing a vertex to define different values for the vertex.

Color and normal specification includes all of these modes. Texture coordinate specification includes only *None*, *Vertex*, and *FaceVertex* modes.

Derived classes include **TriangleGeometry**. Triangle strips and triangle fans are more efficient but less flexible representations of triangles. They could be implemented as additional classes derived from **Geometry**.

TriangleGeometry

The **TriangleGeometry** class represents a set of triangles. It supports ray intersection testing by testing each triangle.

```
class TriangleGeometry : public Geometry
{
public:
    // rendering
    virtual void Render() const;

    // intersection
    virtual bool ComputeRayIntersection(real& tRay,
        Vector2& vCoord, const Ray3& ray) const;

protected:
    virtual void RenderPolygon(int iPoly) const;
};
```

Light

The **Light** class represents an OpenGL light with ambient, diffuse, and specular properties. Any lights that are traversed by a **RenderVisitor** will be activated. If the maximum number of lights is reached during a traversal, no additional lights will be activated.

```
class Light : public Node
{
protected:
    Vector4 vAmbient, vDiffuse, vSpecular;
    real specularExp;

public:
    Light();
    Light(const Light& light);

    // accessors
    const Vector4& Ambient() const;
    const Vector4& Diffuse() const;
    const Vector4& Specular() const;

    void Ambient(const Vector4& v);
    void Diffuse(const Vector4& v);
    void Specular(const Vector4& v);

    // light operations
    virtual void Configure(GLenum lightNum,
        const Vector3& vPosition) const;

    // node operations
    virtual void Accept(Visitor &visitor);
};
```

State

The **State** base class, derived from **Node**, represents a rendering state that applies to all nodes visited after the **State** node. A specific state cannot be deactivated unless another state overrides it. Derived classes include **Material**, **Texture**, and **Shaders**.

```
class State : public Node
{
public:
    // node operations
    virtual void Accept(Visitor &visitor);
};
```

Material

The **Material** class represents the ambient, diffuse, and specular properties of a renderable surface.

```
class Material : public State
{
protected:
    Vector3 vAmbient, vDiffuse, vSpecular;

public:
    // constructors
    Material();
    Material(const Material &mat);

    // accessors
    const Vector3& Ambient() const;
    const Vector3& Diffuse() const;
    const Vector3& Specular() const;

    Vector3& Ambient();
    Vector3& Diffuse();
    Vector3& Specular();

    void Ambient(const Vector3& v);
    void Diffuse(const Vector3& v);
    void Specular(const Vector3& v);

    // node operations
    virtual void Apply(Visitor &visitor);
};
```

Texture

The **Texture** class contains an OpenGL texture number and a slot number for multitexturing. This class does not contain actual texture data or specify whether it is a 1D, 2D, or 3D texture.

```
class Texture : public State
{
protected:
    unsigned int texType, texNum, texSlot;

public:
    Texture();
    virtual ~Texture();

    // accessors
    unsigned int TexType() const;
    unsigned int TexNum() const;
    unsigned int TexSlot() const;

    void TexType(unsigned int type);
    void TexNum(unsigned int num);
    void TexSlot(unsigned int slot);

    // texture operations
```

```
    void Generate();
    void Delete();

    // node operations
    virtual void Apply(Visitor &visitor);
};
```

Shader

The **Shader** class can represent either a vertex or fragment shader. Shaders are based on the Cg shader language and NVIDIA's Cg runtime library.

Before using any Cg shaders, a Cg context must be created by calling *CreateContext*.

```
class Shader : public State
{
protected:
    static CGcontext context;

protected:
    CGprogram program;
    CGprofile profile;

public:
    Shader();

    // accessors
    void Profile(CGprofile profile);
    CGprofile Profile() const;

    // Cg context creation
    static void CreateContext();

    // program loading
    bool Read(const char *file);

    // node operations
    virtual void Apply(Visitor& visitor);

protected:
    // shader operations
    virtual void BindParameters();
    virtual void LoadUniformParameters();
};
```

Compound

The **Compound** base class represents any node with children. It does not define the type of storage used for children.

```
class Compound : public Node
{
public:
    // node operations
    virtual void Accept(Visitor& visitor);
    virtual void Revert(Visitor& visitor);
    virtual void Traverse(Visitor& visitor) = 0;
};
```

Derived classes include **Group**.

Group

The **Group** base class stores a set of child nodes.

```
class Group : public Compound
{
protected:
    std::vector<Node*> nodes;

public:
    // group operations
    int Size() const;
    Node& Child(int index) const;

    void Add(Node& node);
    void Insert(int index, Node& node);
    bool Remove(Node& node);
    void Remove(int index);
    void Clear();

    bool Find(int& index, const Node& node,
        int start = 0) const;

    // node operations
    virtual void Traverse(Visitor& visitor);
};
```

Child nodes are stored internally as an array. The choice of an array instead of a linked list increases access speed but decreases the speed of insertions and deletions. Applications that make many insertions and deletions in real-time should define a more suitable node.

Derived classes include **Transform**.

Transform

The **Transform** base class, derived from **Group**, represents any transformation that can be represented by a matrix. Classes derived from **Transform** must define functions to convert the transformation and its inverse to matrices. This requirement ensures that any sequence of **Transform** nodes may be concatenated and that an inverse concatenation can be computed as efficiently as possible. Derived classes include **RigidTransform**, **ScaleTransform**, **MatrixTransform**, and **TransformPath**.

```
class Transform : public Group
{
public:
    // transform operations
    void Apply() const;
    void ApplyInverse() const;
    static void Revert();

    // matrix operations
    virtual void ToMatrix(Matrix4& m) const = 0;
    virtual void InverseToMatrix(Matrix4& m) const = 0;

    // node operations
```

```
    virtual void Accept(Visitor &visitor);
    virtual void Apply(Visitor &visitor);
    virtual void Revert(Visitor &visitor);
};
```

RigidTransform

The **RigidTransform** class represents a rigid-body transformation, which consists of a translation and rotation. The translation is represented as a three-vector and the rotation is represented as a quaternion. The use of a quaternion reduces the number of arithmetic operations involved in a rotation, avoids the problem of Gimbal lock, and allows spherical linear interpolation.

```
class RigidTransform : public Transform
{
protected:
    Vector3 vTranslation;
    Quaternion qOrientation;

public:
    RigidTransform();
    RigidTransform(const RigidTransform &trans);

    // accessors
    const Vector3& Translation() const;
    const Quaternion& Orientation() const;

    Vector3& Translation();
    Quaternion& Orientation();

    void Translation(const Vector3& v);
    void Orientation(const Quaternion& q);

    // transform operations
    void Translate(const Vector3& v);
    void Rotate(const Quaternion& q);

    // matrix operations
    virtual void ToMatrix(Matrix4& m) const;
    virtual void InverseToMatrix(Matrix4& m) const;
};
```

ScaleTransform

The **ScaleTransform** class represents a scale transformation. The actual scaling is stored as a three-vector, allowing non-uniform scaling.

```
class ScaleTransform : public Transform
{
protected:
    Vector3 vScale;

public:
    ScaleTransform();
    ScaleTransform(const ScaleTransform &scale);

    // accessors
    const Vector3& Scale() const;
    Vector3& Scale();
    void Scale(const Vector3& v);
```

```
    // matrix operations
    virtual void ToMatrix(Matrix4& m) const;
    virtual void InverseToMatrix(Matrix4& m) const;
};
```

MatrixTransform

The **MatrixTransform** class represents an arbitrary matrix transformation. Both the matrix and its inverse must be explicitly defined. This class may be used for shear, projection, or other transformations that do not have their own class.

```
class MatrixTransform : public Transform
{
protected:
    Matrix4 m, mInverse;

public:
    // accessors
    const Matrix4& Matrix() const;
    const Matrix4& MatrixInverse() const;

    Matrix4& Matrix();
    Matrix4& MatrixInverse();

    void Matrix(const Matrix4& m);
    void MatrixInverse(const Matrix4& mInverse);

    // matrix operations
    virtual void ToMatrix(Matrix4& m) const;
    virtual void InverseToMatrix(Matrix4& m) const;
};
```

TransformPath

The **TransformPath** class contains a list of pointers to **Transform** nodes, but it does not store actual transformations. This implementation allows a **TransformPath** node to use **Transform** nodes that may be changing without requiring the **TransformPath** node to update itself. This class is useful when defining a camera transformation for a camera that has several transformations applied to it.

```
class TransformPath : public Transform
{
protected:
    std::list<Transform*> transformList;

public:
    // accessors
    const std::list<Transform*>& TransformList() const;
    std::list<Transform*>& TransformList();

    void PushTransform(Transform& transform);
    void PopTransform();
    void ClearTransforms();

    // matrix operations
    virtual void ToMatrix(Matrix4& m) const;
    virtual void InverseToMatrix(Matrix4& m) const;
};
```

Cameras

The **Camera** base class, derived from **Object**, represents a projection transformation. It requires a view aspect ratio (view height divided by view width). It declares a virtual function to apply the projection, but it does not define it. Derived classes include **OrthographicCamera** and **PerspectiveCamera**.

```
class Camera : public Object
{
protected:
    real aspect, zNear, zFar;

public:
    Camera();
    Camera(real zNear, real zFar, real aspect = 1.0f);

    // accessors
    const real& Near() const;
    const real& Far() const;
    const real& Aspect() const;

    real& Near();
    real& Far();
    real& Aspect();

    void Clipping(real zNear, real zFar);
    void Near(real zNear);
    void Far(real zFar);
    void Aspect(real aspect);

    // camera operations
    virtual void ComputeFrustum(Frustum& frustum) const;
    virtual void ComputeRay(Ray3& ray, real x, real y) const;
    virtual void ComputeProjection(Matrix4& m) const;
    virtual void ApplyProjection() const;
};
```

OrthographicCamera

The **OrthographicCamera** class represents an orthographic (or parallel) projection. In this type of projection, parallel lines remain parallel rather than converging as the distance from the camera approaches infinity. This class contains the width of the projection in local coordinates.

```
class OrthographicCamera : public Camera
{
protected:
    real width;

public:
    // constructors
    OrthographicCamera();
    OrthographicCamera(const OrthographicCamera &cam);
    OrthographicCamera(real zNear, real zFar, real width,
        real aspect = 1.0f);

    // accessors
    real Width() const;
    real& Width();
    void Width(real width);

    // camera operations
    real HalfWidth() const;
```

```
    real HalfHeight() const;

    // camera operations
    virtual void ComputeFrustum(Frustum& frustum) const;
    virtual void ComputeRay(Ray3& ray, real x, real y) const;
    virtual void ComputeProjection(Matrix4& m) const;
    virtual void ApplyProjection() const;
};
```

PerspectiveCamera

The **PerspectiveCamera** class represents a perspective projection, defined by a horizontal field-of-view (FOV) angle.

```
class PerspectiveCamera : public Camera
{
protected:
    real yFov;

public:
    PerspectiveCamera();
    PerspectiveCamera(real zNear, real zFar, real yFov,
        real aspect = 1.0f);
    PerspectiveCamera(const PerspectiveCamera &cam);

    // accessors
    const real& FieldOfView() const;
    real& FieldOfView();
    void FieldOfView(real yFov);

    // camera operations
    virtual void ComputeFrustum(Frustum& frustum) const;
    virtual void ComputeRay(Ray3& ray, real x, real y) const;
    virtual void ComputeProjection(Matrix4& m) const;
    virtual void ApplyProjection() const;
};
```

Visitors

The **Visitor** virtual base class represents a scene graph traversal based on the "Visitor" design pattern. This design pattern allows custom traversals to be defined without modifying existing node classes.

The **Visitor** class contains redefinable functions to traverse **Node**, **Geometry**, **Light**, **State**, **Group**, and **Transform** classes. Derived classes include **CullVisitor**, **RenderVisitor**, **PickVisitor**, **UpdateVisitor**, and **CollisionVisitor**.

```
class Visitor : public Object
{
protected:
    std::vector<Compound*> parentStack;
    int parentCount;
    MatrixStack viewMatrixStack;
    MatrixStack objectMatrixStack;

public:
    Visitor();

    // accessors
    bool ValidParent() const;
```

```
    Compound& CurrentParent() const;

    const MatrixStack& ViewMatrixStack() const;
    const MatrixStack& ObjectMatrixStack() const;
    MatrixStack& ViewMatrixStack();
    MatrixStack& ObjectMatrixStack();

    // matrix operations
    void ComputeMatrix(Matrix4& m) const;
    void ComputeInverseMatrix(Matrix4& m) const;

    // overrides
    virtual bool TestBounds(Volume& bounds);
    virtual void VisitChild(Node& node);
    virtual void VisitNode(Node& node);
    virtual void VisitGeometry(Geometry& geometry);
    virtual void VisitLight(Light& light);
    virtual void VisitState(State& state);
    virtual void VisitCompound(Compound& compound);
    virtual void VisitTransform(Transform& transform);
};
```

CullVisitor

The **CullVisitor** class traverses a scene, testing node bounding volumes against a frustum. Nodes without bounding volumes are assumed to be in the frustum. This traversal produces a simpler scene tree consisting of **Transform**, **State**, and **Geometry** nodes.

```
class CullVisitor : public Visitor
{
protected:
    std::stack<Frustum*> frustumStack;
    RenderTree *currentTree;

public:
    // accessors
    std::stack<Frustum*>& FrustumStack();
    void CurrentTree(RenderTree& tree);

    // cull operations
    virtual void Cull(RenderTree& tree, const Node& node);
    virtual void Cull(RenderTree& tree, const Node& node,
        const Camera& camera, const Transform& viewTransform);

    // visitor operations
    virtual bool TestBounds(Volume& bounds);
    virtual void VisitNode(Node& node);
    virtual void VisitGeometry(Geometry& geometry);
    virtual void VisitLight(Light& light);
    virtual void VisitState(State& state);
    virtual void VisitCompound(Compound& compound);
    virtual void VisitTransform(Transform& transform);
};
```

RenderVisitor

The **RenderVisitor** class traverses a **RenderTree** object. Any lights that are encountered are activated.

```
class RenderVisitor : public Visitor
{
protected:
```

```
    RenderTree* currentTree;
    int lightCount;

public:
    RenderVisitor();

    // accessors
    int LightCount() const;
    void LightCount(int lightCount);

    // operations
    virtual void Render(const RenderTree& tree);
    virtual void Render(const RenderTree& tree,
        const Transform& viewTransform);

    // overrides
    virtual void VisitNode(Node& node);
    virtual void VisitGeometry(Geometry& geometry);
    virtual void VisitLight(Light& light);
    virtual void VisitState(State& state);
    virtual void VisitCompound(Compound& compound);
    virtual void VisitTransform(Transform& transform);
};
```

PickVisitor

The **PickVisitor** class traverses a scene, testing for intersections with a given ray and placing intersection data in a given list.

```
class PickVisitor : public Visitor
{
public:
    struct Intersection
    {
        bool operator<(const Intersection& intersect) const;
        bool operator>(const Intersection& intersect) const;

        TransformPath   transformPath;
        Node            *node;
        Compound        *parent;
        real            tRay;
        Vector3         vPosition;
        Vector2         vCoord;
    };

    typedef std::list<Intersection> PickList;

protected:
    PickList *pickList;
    TransformPath transformPath;
    Ray3 pickRay;

public:
    // operations
    virtual void Pick(PickList& pickList,
        Node& node, const Ray3& ray);

    // overrides
    virtual bool TestBounds(Volume& bounds);
    virtual void VisitNode(Node& node);
    virtual void VisitGeometry(Geometry& geometry);
    virtual void VisitLight(Light& light);
    virtual void VisitState(State& state);
    virtual void VisitCompound(Compound& compound);
    virtual void VisitTransform(Transform& transform);

protected:
    virtual bool PickNode(Node& node);
};
```

UpdateVisitor

The **UpdateVisitor** class traverses a scene, calling node update functions with a given time interval.

```
class UpdateVisitor : public Visitor
{
protected:
    real interval;

public:
    // accessors
    real Interval() const;
    void Interval(real interval);

    // update operations
    void Update(Node& node, real interval);
    void Update(Node& node, real interval,
        Transform& viewTransform);

    // visitor operations
    virtual void VisitNode(Node& node);
    virtual void VisitGeometry(Geometry& geometry);
    virtual void VisitLight(Light& light);
    virtual void VisitState(State& state);
    virtual void VisitCompound(Compound& compound);
    virtual void VisitTransform(Transform& transform);
};
```

Appendix C: Dataset Design

Multi-dimensional Array Classes

Multi-dimensional array classes include **Vector**, **MultiIndex**, **MultiArray**, and **MultiIterator**. These classes are templated to ensure that low-level, per-element operations are performed without function calls.

Vector

The **Vector** class represents a fixed-size vector. It requires *number of elements* and *element type* template parameters.

```
template <class T, int S>
class Vector
{
protected:
    T v[S];

public:
    Vector();
    Vector(const T v[S]);
    Vector(const Vector& v);
    Vector(const T& value);

    // accessors
    const T& operator[](int i) const;
    T& operator[](int i);

    // assignment
    Vector& operator=(const Vector& v);

    // boolean operations
    bool operator==(const Vector& v) const;
    bool operator!=(const Vector& v) const;

    // arithmetic operations
    Vector operator*(const T& k) const;

    template <class T2, int S2>
    friend Vector<T2, S2> operator*(T2 k,
        const Vector<T2, S2>& v);

    Vector operator/(const T& k) const;
    Vector operator+(const Vector& v) const;
    Vector operator-(const Vector& v) const;
    Vector operator-() const;

    Vector& operator*=(const T& k);
    Vector& operator/=(const T& k);
    Vector& operator+=(const Vector& v);
    Vector& operator-=(const Vector& v);

    // miscellaneous operations
    void Fill(const T& k);

    // stream operations
    template <class T2, int S2>
    friend std::ostream& operator<<(
        std::ostream& stream,
        const Vector<T2, S2>& v);

    template <class T2, int S2>
```

```
    friend std::istream& operator>>(
        std::istream& stream,
        Vector<T2, S2>& v);
};
```

MultiIndex

The **MultiIndex** class contains a single function to compute an index within a multi-dimensional array, given a **Vector** location and a **Vector** containing array bounds. It requires a *number of dimensions* template parameter. This class is templated to allow index computation within any number of dimensions without using loops or function calls.

```
template <int D>
struct MultiIndex
{
    static unsigned int Compute(
        const Vector<int, D>& location,
        const Vector<int, D>& dims);
};
```

MultiArray

The **MultiArray** class represents a multi-dimensional array. It requires *number of dimensions* and *element type* template parameters. To resize a **MultiArray**, a **Vector** containing the dimensions of the array is required.

```
template <class T, int D>
class MultiArray
{
public:
    typedef MultiIterator<D> Iterator;

protected:
    Vector<int, D> dims;
    std::vector<T> array;

public:
    MultiArray();
    MultiArray(const Vector<int, D>& dims);

    // accessors
    const Vector<int, D>& Dims() const;
    const T& operator[](int i) const;
    T& operator[](int i);

    // array operations
    unsigned int Size() const;
    unsigned int ByteSize() const;
    bool Empty() const;
    virtual void Clear();
    void Resize(const Vector<int, D>& dims);
    void Fill(const T& value);

    // stream operations
    template <class T2, int D2>
    friend std::ostream& operator<<(
        std::ostream& stream,
        const MultiArray<T2, D2>& array);

    template <class T2, int D2>
    friend std::istream& operator>>(
```

```
        std::istream& stream,
        MultiArray<T2, D2>& array);
};
```

MultiIterator

The **MultiIterator** class represents an iteration through a multi-dimensional array. It requires a *number of bytes* template parameter. The purpose of this class is to automate efficient array iteration.

The iteration must be initialized by supplying a **Vector** containing the dimensions of an array. A range may be set by supplying minimum and maximum **Vector** positions within the array. A **Vector** step value may also be supplied. Before iteration occurs, the *Start* function must be called. This function performs precomputations to ensure maximum efficiency during iteration. The ++ operation is overloaded for iteration.

```
template <int D>
class MultiIterator
{
protected:
    Vector<int, D> dims, first, last, step;
    Vector<int, D> skip, current;
    unsigned int index;

public:
    MultiIterator();
    MultiIterator(const MultiIterator& i);
    MultiIterator(const Vector<int, D>& dims);
    MultiIterator(const Vector<int, D>& dims,
        const Vector<int, D>& first,
        const Vector<int, D>& last);

    // accessors
    const Vector<int, D>& Dims() const;
    const Vector<int, D>& First() const;
    const Vector<int, D>& Last() const;
    const Vector<int, D>& Step() const;
    const Vector<int, D>& Skip() const;
    const Vector<int, D>& Current() const;
    unsigned int Index() const;

    Vector<int, D> Extent() const;

    void Limits(const Vector<int, D>& dims);
    void Limits(const Vector<int, D>& dims,
        const Vector<int, D>& first,
        const Vector<int, D>& last);
    void Step(const Vector<int, D>& step);

    // operations
    void Clear();

    // iteration
    void Start();
    bool Valid() const;
    void operator++(int);

    static void Iterate(
        const Vector<int, D>& first,
        const Vector<int, D>& last,
        const Vector<int, D>& step,
        const Vector<int, D>& skip,
        Vector<int, D>& current,
        unsigned int& index);
```

```
};
```

Multi-dimensional Array Operations

Several common operations are provided for use with the multi-dimensional array classes. These operations are implemented as templated functions, requiring four template parameters: *input type*, *input dimensions*, *output type*, and *output dimensions*.

These operations include the following:

- **CopyArray**. This operation simply copies input values to output values. To perform a subsampling operation, the input iterator step values may be used.
- **FindMinMax**. This operation determines the minimum and maximum input values.
- **ApplyThreshold**. This operation compares input values against a threshold value. Values less than the threshold will produce a minimum value. Values greater than or equal to the threshold will produce a maximum value.
- **ApplyLinearMap**. Given an input range and an output range, this operation linearly maps input values in the input range to the output range. Values below the input minimum will map to the output minimum. Values above the input maximum will map to the output maximum.
- **ComputeWeightedSum**. Given two input arrays and two weightings, this operation computes weighted sums and stores the results in the output array.
- **ComputeGradient**. This operations performs simple gradient computations. For a given location, delta values are defined as

Dataset Classes

Dataset

The **Dataset** class represents a five-dimensional array of bytes, organized into sets of three-dimensional arrays of cells. The dimensions are *step count*, *depth*, *height*, *width*, and *cell size*. This class is not templated to allow the cell size to be determined at run-time.

Dataset declares the virtual functions *Read*, *Write*, *ReadHeader*, and *WriteHeader*. By default, *Read* and *Write* immediately call *ReadHeader* or *WriteHeader* and then read or write the dataset.

```
class Dataset : public MultiArray<unsigned char, 5>
{
public:
    Dataset();
    Dataset(int cellSize, int width, int height,
```

```
        int depth, int stepCount);

    // accessors
    int CellSize() const;
    int Width() const;
    int Height() const;
    int Depth() const;
    int StepCount() const;

    // array operations
    void Resize(const Vector<int, 5>& dims);
    void Resize(int cellSize, int width,
        int height, int depth, int stepCount);

    // dataset computations
    int VolumeCellCount() const;
    int VolumeSize() const;
    int VolumeOffset(int i) const;
    int PositionOffset(int x, int y, int z) const;

    // file operations
    virtual bool ReadHeader(std::istream& is);
    virtual bool WriteHeader(std::ostream& os) const;
    bool Read(const char file[]);
    bool Write(const char file[]) const;
    static bool SeekKey(std::istream& is, const char key[]);
};
```

MrDataset

The **MrDataset** class represents a special type of dataset. It contains X and Y field of view values, a slice thickness value, and a skip value. It overrides the *ReadHeader* and *WriteHeader* functions declared in **Dataset**.

```
class MrDataset : public Dataset
{
protected:
    float xFov, yFov, thickness, skip;

public:
    MrDataset();
    MrDataset(int cellSize, int width,
        int height, int depth, int stepCount);

    // accessors
    float FovX() const;
    float FovY() const;
    float Thickness() const;
    float Skip() const;

    void FovX(float xFov);
    void FovY(float yFov);
    void Thickness(float thickness);
    void Skip(float skip);

    // general operations
    virtual void Clear();

    // dataset computations
    float CellWidth() const;
    float CellHeight() const;
    float CellDepth() const;

    // file operations
    virtual bool ReadHeader(std::istream& is);
    virtual bool WriteHeader(std::ostream& os) const;
};
```

MrDataset file headers take the following form:

```
3.0.0 BINARY DISCRETE_SET
NUM_DIMS 4
SIZE [width] height] [depth] [steps]
FOV [x fov] [y fov]
THICKNESS [z thickness]
SKIP [z skip]
BYTES_PER_CELL [bytes per cell]
TYPE RAW
```

BigEndian

The **BigEndian** class represents a set of bytes in big endian order. It requires a *number of bytes* template parameter. Cast and assignment operations are overloaded to allow this class to behave the same as an **unsigned integer**.

```
template <int S>
class BigEndian
{
protected:
    unsigned char bytes[S];

public:
    // accessors
    unsigned char operator[](int i) const;
    unsigned char& operator[](int i);

    // conversions
    operator unsigned int() const;
    BigEndian<S>& operator=(unsigned int value);
};
```

Datasets are assumed to contain values in big endian order. To use these values, an array of **unsigned char** values within a dataset should be interpreted as an array of **BigEndian** values.

Appendix D: Application Design

GUI Classes

Director

The "Director" pattern is used to consolidate interactions among GUI objects.

The **Director** abstract base class implements the "Director" pattern. It contains the single virtual function *Changed(Fl_Widget)*.

```
class Director
{
public:
    // interface creation
    virtual void CreateInterface();

    // director operations
    virtual void Changed(Fl_Widget& widget);
};
```

A subclass of Director would implement *Changed* to determine which member object called it, what changed, and what actions should be taken. It identifies the object by comparing memory addresses of member objects and the object that called Change. The Director may then query the object to determine what changed.

CallbackWindow

The **CallbackWindow** class, derived from **Fl_Gl_Window**, provides a wrapper for an OpenGL window that supports rendering, mouse, and initialization callbacks. Defining these callbacks can avoid having to derive new window types to extend functionality. This class also tracks mouse motion and may be managed with a **Director**.

```
class CallbackWindow : public Fl_Gl_Window
{
public:
    typedef void Callback(CallbackWindow&, void*);

protected:
    Director        *director;
    float           clearColor[4];
    int             xMouse,
                    yMouse;
    unsigned int    eventState;
    Callback        *onSetDefaults,
                    *onPreRender,
                    *onPostRender,
                    *onMouseDown[3],
                    *onMouseUp[3],
                    *onMouseMove,
                    *onMouseDrag[3],
                    *onMouseWheel;

public:
    CallbackWindow(int x, int y, int w, int h,
```

```
        Director& director, const char *label = 0);

    // accessors
    void ClearColor(float r, float g, float b);

    int MouseX() const;
    int MouseY() const;
    int MouseDX() const;
    int MouseDY() const;

    void Changed();

    // callback initialization
    void OnSetDefaults(Callback *cb);
    void OnPreRender(Callback *cb);
    void OnPostRender(Callback *cb);
    void OnMouse1Down(Callback *cb);
    void OnMouse2Down(Callback *cb);
    void OnMouse3Down(Callback *cb);
    void OnMouse1Up(Callback *cb);
    void OnMouse2Up(Callback *cb);
    void OnMouse3Up(Callback *cb);
    void OnMouse1Drag(Callback *cb);
    void OnMouse2Drag(Callback *cb);
    void OnMouse3Drag(Callback *cb);
    void OnMouseMove(Callback *cb);
    void OnMouseWheel(Callback *cb);

    // window operations
    virtual int handle(int event);

protected:
    // window operations
    virtual void SetDefaults();
    virtual void Render();

private:
    void draw();
};
```

The following callbacks may be defined:

- **OnSetDefaults**. This function is called when the window must be initialized. This function may be used to load OpenGL extensions.
- **OnPreRender**. This function is called before the window renders.
- **OnPostRender**. This function is called after the window renders.
- **OnMouse1Down, OnMouse2Down, OnMouse3Down**. These functions are called when the appropriate mouse button has been pressed. They will be called unless the mouse button was previously up.
- **OnMouse1Up, OnMouse2Up, OnMouse3Up**. These functions are called when the appropriate mouse button has been released. They will be called unless the mouse button was previously down.
- **OnMouse1Drag, OnMouse2Drag, OnMouse3Drag**. These functions are called when the mouse is moved while the appropriate mouse button is down.

- **OnMouseMove**. This function is called when the mouse moves.
- **OnMouseWheel**. This function is called when the mouse wheel rotates.

SceneWindow

The **SceneWindow** class, derived from CallbackWindow, represents a 3D scene that may be translated, rotated, or scaled. It provides default input callbacks that may be used for these operations.

It contains a root node within a **RigidTransform** within a ScaleTransform. It requires a subclass of **Camera** to be defined.

SceneWindow contains a **PickVisitor::PickList** object.

```
class SceneWindow : public CallbackWindow
{
protected:
    Camera                  *camera;
    RigidTransform          cameraTransform;
    ScaleTransform          scaleTransform;
    RigidTransform          rigidTransform;
    PickVisitor             picker;
    PickVisitor::PickList   pickList;

public:
    SceneWindow(int x, int y, int w, int h,
        Director& director, const char *label = 0);

    // accessors
    PickVisitor::PickList& PickList();

    void SceneCamera(Camera& camera);
    void AddNode(Node& node);
    void Position(const Vector3& v);
    void Orientation(const Quaternion& q);
    void Scale(const Vector3& v);
    void CameraPosition(const Vector3& v);
    void CameraOrientation(const Quaternion& q);

    void Update(real interval = 1);

    // default control callbacks
    static void MouseDownPick(CallbackWindow& win, void*);
    static void MouseDragTranslate(CallbackWindow& win, void*);
    static void MouseDragRotate(CallbackWindow& win, void*);
    static void MouseDragScale(CallbackWindow& win, void*);
    static void MouseWheelScale(CallbackWindow& win, void*);

protected:
    // window operations
    virtual void SetDefaults();
    virtual void Render();
};
```

Confidence Operations

ComputeRiseFall

This operation computes a rise and fall value at each location in a dataset volume. Given a location with a time step intensity curve, rise is defined as the difference between the first intensity value and the maximum intensity value in the curve. Fall is defined as the difference between the maximum intensity value in the curve and the last intensity value.

Confidence Classes

ConfidenceData

The **ConfidenceData** contains a volume of confidence values, precomputed values, and intermediate values. It contains an *Update* function that recomputes confidence values, requiring *rise threshold*, *fall threshold*, *rise weight*, and *confidence threshold* parameters. It contains a dirty state that determines what computations are performed when *Update* is called, ensuring that the fewest possible operations are performed. Dirty states roughly represent the stages of confidence computation.

Dirty states include the following:

- **Current**. Nothing has changed since the last update.
- **ConThreshold**. Confidence threshold has changed.
- **RiseWeight**. Rise weight has changed.
- **RiseFallThreshold**. Rise and fall thresholds have changed.
- **Precomputations**. The original dataset has changed. When a new dataset is loaded, this state should be set.

```
class ConfidenceData
{
public:
    // these states are in the order of the computations
    enum DirtyState
    {
        DDS_CURRENT,
        DDS_CONTHRESHOLD,
        DDS_RISEWEIGHT,
        DDS_RISEFALLTHRESHOLD,
        DDS_MINSTEPMAX,
        DDS_LINEARRISEWEIGHT,
        DDS_PRECOMPUTATIONS
    };

    typedef MultiArray<short, 3> ShortArray3;
    typedef MultiArray<unsigned char, 3> ByteArray3;

public:
    ShortArray3 rise, fall,
```

```
                gradient;
    ByteArray3  maxSteps,
                riseNorm,
                fallNorm,
                linearDiff,
                stepStates,
                confidence;
    int         minIntensity,
                maxIntensity,
                minRise,
                maxRise,
                minFall,
                maxFall,
                minLinearDiff,
                maxLinearDiff,
                minConfidence,
                maxConfidence;

protected:
    DirtyState  dirtyState;

public:
    ConfidenceData();

    // accessors
    void Dirty(DirtyState state);
    DirtyState Dirty() const;

    // operations
    void Update(const Dataset& intensity,
        ByteArray3& confidenceNorm,
        float riseThreshold,
        float fallThreshold,
        float riseWeight,
        float linearDiffWeight,
        float conThreshold);
};
```

Rendering Classes

CubeSlices

The **CubeSlices** class, derived from **TriangleGeometry**, represents a series of plane slices through a unit cube.

```
class CubeSlices : public TriangleGeometry
{
protected:
    int         sliceCount;
    bool        facingCamera;
    Vector3     vNormal;
    Matrix3     mTexOrientation;
    Vector3     vTexOffset;
    real        zNear, zFar;

public:
    CubeSlices();

    // accessors
    void SliceCount(int sliceCount);
    void FaceCamera(bool state);
    void Clipping(real zNear, real zFar);
    void Normal(const Vector3& v);
    void TexOrientation(const Matrix3& m);
    void TexOffset(const Vector3& v);

    // node operations
```

```
    virtual void Update(Visitor& visitor, real interval);

protected:
    // slice generation
    static void FindNearestVertex(Vector3& vNearest,
        const Vector3& vNormal);
    static real GetDistance(const Vector3& v1, const Vector3& v2,
        const Vector3& vCenter);

    static void SortSlice(Vector3 verts[], int poly[],
        int vertexCount);
    static void GetSlice(Vector3 vNormal, Vector3 vPoint,
        Vector3 verts[6], int &vertexCount);
};
```

Primitives

The following classes are derived from **Node**:

- **SelectionAxes**. This node renders three dotted lines within a unit cube, intersecting at a selected position.
- **LabeledAxes**. This node renders three axes with letters to indicate the orientation of the axes.
- **WireCube**. This node renders a unit wire cube.

```
class SelectionAxes : public Node
{
public:
    Vector3 vPosition;
    virtual void Apply(Visitor &visitor);
};
class LabelledAxes : public Node
{
public:
    virtual void Apply(Visitor &visitor);
    static void RenderString(std::string str);
};
class WireCube : public Node
{
public:
    // intersection calculations
    virtual bool TestRay(const Ray3& ray) const;
    virtual bool ComputeRayIntersection(real& tRay,
        Vector2& vCoord, const Ray3& ray) const;

    // node operations
    virtual void Apply(Visitor &visitor);
};
```

ConfidenceShader

The **Confidence** class, derived from **Shader**, represents a Cg fragment shader node with *window*, *level*, *opacity*, and *confidence opacity* parameters. This node binds these parameters to Cg shader parameters when rendering.

```
class ConfidenceShader : public Shader
```

```
{
public:
    float   window, level,
            opacity, conOpacity;

protected:
    CGparameter paramWindow, paramLevel,
                paramOpacity, paramConOpacity;

public:
    ConfidenceShader();

protected:
    // shader operations
    virtual void BindParameters();
    virtual void LoadUniformParameters();
};
```

Cg Confidence Shader Programs

Three shader programs are used in the application: a 3D view shader, a 2D view shader, and a highlight shader. The 3D view shader is used to render semi-transparent slices in the 3D view. The 2D view shader is used to render the single, opaque slices in the 2D views. The highlight shader is used to render the highlighted slices that appear in the 3D view that represent the 2D view slices. These shaders generally perform the same operations, but differ in the way they compute transparency.

The following is the entire 3D view shader program:

```
float4 main(in float3 texCoord : TEXCOORD0,
            uniform float window,
            uniform float level,
            uniform float opacity,
            uniform float conOpacity,
            uniform sampler3D tex0,
            uniform sampler2D tex1) : COLOR0
{
    float4 cell = tex3D(tex0, texCoord.xyz);

    // get truncated intensity
    float intensity = cell.g;
    intensity = (cell.r > 0.001) ? 1 : intensity;
    intensity *= 2;

    // apply window and level
    intensity -= level;
    intensity *= window;

    // compute confidence factor
    float confidence = cell.b;
    confidence *= confidence;
    confidence *= 4;
    confidence *= 4;
    confidence *= conOpacity;

    float alpha = (intensity * 0.1 + confidence) * opacity;

    // return color
    return float4(
        intensity + confidence,
        intensity - confidence,
        intensity - confidence,
        alpha);
}
```

Application Classes

MainDirector

The **MainDirector** class, derived from **Director**, contains all GUI components and data associated with the main interface of the application.

The following classes are defined within **MainDirector**:

- **WheelScroll**. This class, derived from **Fl_Scroll**, overrides event handling so mouse wheel events will only be handled when the cursor is within the window. The default event handling behavior will only allow the window in focus to handle events.

- **ShaderDisabler**. This class, derived from **State**, disables any shaders when traversed. This allows the labeled axis, bounding box, and selections to render properly when they are traversed after the **ShaderDisabler**.

```
class MainDirector : public Director
{
public:
    // WheelGroup: Overrides handle to only take mouse wheel events
    // if the mouse is within the group bounds. This is necessary for
    // the viewports to work with scroll views.
    class WheelScroll : public Fl_Scroll
    {
    public:
        WheelScroll(int x, int y, int w, int h,
            const char *label = 0);
        virtual int handle(int event);
    };

    // ShaderDisabler: Disables 3D texturing and shaders so normal
    // lines may be drawn.
    class ShaderDisabler : public State
    {
    public:
        virtual void Apply(Visitor &visitor);
    };

    typedef MultiArray<unsigned char, 3> ByteArray3;

protected:
    // datasets
    MrDataset           intensity;
    ByteArray3          confidenceNorm;
    ConfidenceData      conData;
    int                 texWidth,
                        texHeight,
                        texDepth;

    // scene objects
    OrthographicCamera  orthoCams[3];
    ScaleTransform      voxelScale[4];
    Texture             volumeTex;
    Texture             transferTex;
    ConfidenceShader    conShader2d,
                        conShader3d,
                        conShaderHighlight;
    RigidTransform      volumeOrientation;
    CubeSlices          cubeSlices[4];
    LabelledAxes        labelledAxes;
    WireCube            boundingBox;
    SelectionAxes       selectionAxes;
    Material            selectionMat;
```

```
    Group               cubeGroup,
                        selectionAxes3D,
                        selectionGroup,
                        axisSliceGroup3D,
                        axisSlices3D[3];
    RigidTransform      axisSlicesPos3d[3];
    WireCube            selectionBox3D;
    Vector3             vTexCoordScale;
    Matrix3             mTexOrientation;

    // interface
    Fl_Window           *mainWin;
    char                szWinTitle[64];
    SceneWindow         *winScenes[4];
    Fl_Button           *btnReset,
                        *btnAddSelection,
                        *btnRemoveSelection,
                        *btnClearCurvesList;
    Fl_Round_Button     *btnSwizzles[6];
    Fl_Value_Slider     *sdrRiseThreshold,
                        *sdrFallThreshold,
                        *sdrRiseWeight,
                        *sdrConThreshold,
                        *sdrLinearRiseWeight,
                        *sdrTimeStep;
    WheelScroll         *scrCurvesList;
    Fl_Box              *boxPositionInfo,
                        *boxVolumeInfo;
    CallbackWindow      *winPlot;
    Fl_Menu_Item        popupMenu[11];
    std::list<Vector3>  curvesList;

    // settings
    float               window,
                        level;

public:
    MainDirector();

    // accessors
    static MainDirector& Instance();

    // interface creation
    virtual void CreateInterface();

    // director operations
    virtual void Changed(Fl_Widget& widget);

    // application operations
    void ResetSettings();
    void OpenDataset(const char *file);
    void PickPoint(const Vector3& vPoint);
    void AddPoint(const Vector3& vPoint);
    void RefreshViews();
    void ResetViews();

protected:
    // interface operations
    void CreateSceneGroup();
    void CreateControlGroup();
    void CreateNavigationGroup();
    void CreateMainMenu();
    void CreatePopupMenu();

    // updates
    void UpdateWindowLevel();
    void UpdateConfidence();
    void UpdateVolumeTex();
    void UpdateVolumeScale();
    void ReloadShaders();

    // menu callbacks
    static void MnuOpenDataset          (Fl_Widget*, void*);
    static void MnuCloseDataset         (Fl_Widget*, void*);
```

```
    static void MnuFileInformation      (Fl_Widget*, void*);
    static void MnuExitApplication      (Fl_Widget*, void*);

    // display tab callbacks
    static void ChkFastRender           (Fl_Widget*, void*);
    static void ChkLocalVolume          (Fl_Widget*, void*);
    static void ChkLabelledAxes         (Fl_Widget*, void*);
    static void ChkBoundingBox          (Fl_Widget*, void*);
    static void ChkSlicePositions       (Fl_Widget*, void*);
    static void ChkStoredSelections     (Fl_Widget*, void*);
    static void SdrOpacityChange        (Fl_Widget*, void*);
    static void SdrConOpacityChange     (Fl_Widget*, void*);
    static void SdrSliceCountChange     (Fl_Widget*, void*);
    static void SdrTimeStepChange       (Fl_Widget*, void*);
    static void BtnReloadShaders        (Fl_Widget*, void*);
    static void BtnResetViews           (Fl_Widget*, void*);

    // dataset tab callbacks
    static void BtnSetOrientation       (Fl_Widget*, void*);
    static void ChkFlipAxis             (Fl_Widget*, void*);

    // computation tab callbacks
    static void SdrSetConDirtyState     (Fl_Widget*, void*);
    static void BtnApplyConfidence      (Fl_Widget*, void*);
    static void BtnResetConfidence      (Fl_Widget*, void*);

    // curves tab callbacks
    static void BtnAddSelection         (Fl_Widget*, void*);
    static void BtnRemoveSelection      (Fl_Widget*, void*);
    static void BtnClearSelections      (Fl_Widget*, void*);
    static void BtnSelectCurve          (Fl_Widget*, void*);

    // view callbacks
    static void MouseDragWindowLevel    (CallbackWindow&, void*);
    static void ViewPostRender0         (CallbackWindow&, void*);
    static void ViewPostRender1         (CallbackWindow&, void*);
    static void ViewPostRender2         (CallbackWindow&, void*);
    static void CurveWindowRender       (CallbackWindow&, void*);
    static void LoadExtensions          (CallbackWindow&, void*);
    static void SetBlending             (CallbackWindow&, void*);
};
```