NIST Special Publication 1017-2

Smokeview (Version 5) - A Tool for Visualizing Fire Dynamics Simulation Data Volume II: Technical Reference Guide

Glenn P. Forney

National Institute of
Standards and Technology
U.S. Department of Commerce

NIST Special Publication 1017-2

Smokeview (Version 5) - A Tool for Visualizing Fire Dynamics Simulation Data Volume II: Technical Reference Guide

Glenn P. Forney
Fire Research Division
Building and Fire Research Laboratory

September 2, 2010
Smokeview Version 5.5.8
SVN Repository *Revision* : 6377

U.S. Department of Commerce
Gary Locke, Secretary

National Institute of Standards and Technology
Patrick Gallagher, Director

Certain commercial entities, equipment, or materials may be identified in this document in order to describe an experimental procedure or concept adequately. Such identification is not intended to imply recommendation or endorsement by the National Institute of Standards and Technology, nor is it intended to imply that the entities, materials, or equipment are necessarily the best available for the purpose.

National Institute of Standards and Technology Special Publication 1017-2
Natl. Inst. Stand. Technol. Spec. Publ. 1017-2, 70 pages (August 2009)
CODEN: NSPUE2

U.S. GOVERNMENT PRINTING OFFICE
WASHINGTON: 2009

For sale by the Superintendent of Documents, U.S. Government Printing Office
Internet: bookstore.gpo.gov – Phone: (202) 512-1800 – Fax: (202) 512-2250
Mail: Stop SSOP, Washington, DC 20402-0001

Preface

Smokeview is a software tool designed to visualize numerical calculations generated by the Fire Dynamics Simulator (FDS), a computational fluid dynamics (CFD) model of fire-driven fluid flow. This report documents some of the algorithms Smokeview uses to visualize fire dynamics data giving some of the technical and programming details. Details on the use of Smokeview may be found in the Smokeview User's Guide. Smokeview uses the 3D graphics library OpenGL to visualizing fire and smoke data. This library is used to specify the location, color and lighting of objects residing within a *3D world* defined by FDS. In the context of FDS, these objects may be used to represent geometry (such as blockages) or to visualize data. Smokeview presents fire modeling data using visualization techniques such as tracer particles, 2D shaded contours, iso-surfaces or flow vectors. Soot data or smoke is visualized using a variation of the 2D shaded contour technique where transparency rather than color is used to represent the opacity or optical thickness of smoke. The details used to implement various techniques for visualizing smoke will be discussed.

About the Author

Glenn Forney is a computer scientist at the Building and Fire Research Laboratory (BFRL) of NIST. He received a bachelors of science degree in mathematics from Salisbury State College in 1978 and a master of science and a doctorate in mathematics at Clemson University in 1980 and 1984. He joined the NIST staff in 1986 (then the National Bureau of Standards) and has since worked on developing tools that provide a better understanding of fire phenomena, most notably Smokeview, a software tool for visualizing Fire Dynamics Simulation data.

Disclaimer

The US Department of Commerce makes no warranty, expressed or implied, to users of Smokeview, and accepts no responsibility for its use. Users of Smokeview assume sole responsibility under Federal law for determining the appropriateness of its use in any particular application; for any conclusions drawn from the results of its use; and for any actions taken or not taken as a result of analysis performed using this tools.

Smokeview and the companion program FDS is intended for use only by those competent in the fields of fluid dynamics, thermodynamics, combustion, and heat transfer, and is intended only to supplement the informed judgment of the qualified user. These software packages may or may not have predictive capability when applied to a specific set of factual circumstances. Lack of accurate predictions could lead to erroneous conclusions with regard to fire safety. All results should be evaluated by an informed user.

Throughout this document, the mention of computer hardware or commercial software does not constitute endorsement by NIST, nor does it indicate that the products are necessarily those best suited for the intended purpose.

Contents

List of Figures

Chapter 1

Introduction

1.1 Basic Description of Smokeview

Smokeview is a software tool designed to visualize numerical predictions generated by the Fire Dynamics Simulator (FDS), a computational fluid dynamics (CFD) model of fire-driven fluid flow [1, 2]. Most of Smokeview is written in C using the 3D graphics library, OpenGL [3], for implementing visualization algorithms and GLUT [4] for interacting with both the user and the operating system. More specifically, OpenGL is used to specify the location, color and lighting of objects residing within a *3D world* defined by FDS. In the context of FDS, these objects may be used to represent geometry (such as blockages) or to visualize data. Smokeview uses these underlying techniques as building blocks to visualize data such as tracer particles, 2D shaded contours or 3D level iso-surfaces. Soot data or smoke may also be visualized using a variation of a 2D shaded contour, where transparency rather than color is used to represent the opacity or optical thickness of smoke. This report describes these algorithms giving some of the technical and programmatic details.

Smokeview is comprised of approximately 90,000 lines of code. Most of it is written in C [5]. A small but important part is written in Fortran 90 [6]. This part is used to read data generated by FDS. Additional software libraries (about 250,000 lines of code) are used for implementing dialogs, rendering images and decompressing data files. The use of portable libraries allows Smokeview to run on many platforms including Windows, Linux and OSX (for the Macintosh).

The fundamental purpose of visualization is to gain insight into the phenomena being studied. There is no one best method for visualizing data. Each visualization technique highlights a different aspect of the data. Smokeview visualizes fire dynamics data, typically results from the Fire Dynamics Simulator. This data takes many forms. Some data is static, whereas other data evolves with time. Some data represents geometric objects whereas other data represents the solution to the flow equations solved by FDS or a zone fire model such as CFAST. Smokeview displays fire dynamics data allowing quantitative assessment to be performed using visualization techniques such as animated tracer particles that follow the flow, animated shaded 2D and 3D contours that display flow quantities and animated flow vectors that display flow quantities and direction. Smokeview also visualizes smoke realistically by converting soot density to smoke opacity, displaying smoke as it would actually appear. Each of these visualization techniques highlight different aspects of the underlying flow phenomena.

For details on the use of Smokeview, the reader is advised to read the Smokeview User's Guide [7]. For details on how Smokeview is verified, see the Smokeview Verification Guide [8]. For details on setting up and running FDS cases, read the FDS User's Guide [2]. For details on the theory and algorithms implemented in FDS read the FDS Technical Guide [1].

1.2 Version History

Beginning in the early 1980s and continuing into the 1990s, Howard Baum and Ronald Rehm developed the basic flow solver that evolved into the Fire Dynamics Simulator which was publicly released in 2000[9]. Their solution technique, known as Large Eddy Simulation or LES, captures numerically very complicated fire plume dynamics. Unfortunately, the power of the methodology could not be appreciated without an effective way to view the calculation results. Early attempts to visualize the calculation results consisted of nothing more than little particles swirling about in a box. This was useful to the model developers, but hardly to anyone else. It just did not look like a fire.

Smokeview was written to address this problem. It is an advanced scientific visualization tool whose drawing algorithms are based on physics, not just a tool for drawing pretty pictures. Version 1 of Smokeview was publicly released in February 2000, version 2 in December 2001, version 3 in November 2002, and version 4 in July 2004. The present version of Smokeview is 5, officially released in September 2007.

Along with particle tracking as performed before, it visualized fire flow data by coloring and animating fire/smoke flow making it much easier to interpret FDS simulation results. Immediately after September 11, 2001, work began on both FDS and Smokeview to enable them to model and visualize much larger problems. As a result, fire scenarios with several million grid cells can now be modeled and visualized using a cluster of computers.

The next big step in Smokeview's development was the implementation of an algorithm for realistically visualizing smoke. The line between FDS which performs smoke flow computations and Smokeview which performs smoke flow visualization became blurred as Smokeview now performs physics-based computations (Beer's law) in order to visualize the smoke. The present algorithm for visualizing smoke only considers the effects of absorption - how much an object is obscured by smoke. Future work involves modeling the effects of scattering - how the interaction between light and smoke effects the visualization and coloring the fire more realistically based on physics principles such as the black body temperature curve. The challenge though is that how the eye perceives color is complicated. One complication is the fact that often the color one perceives depends on what one was just looking at.

1.3 Model Development

Currently, Smokeview is maintained by the Building and Fire Research Laboratory (BFRL) of the National Institute of Standards and Technology. Starting with Version 5, the FDS-Smokeview development team uses an Internet-based development environment called GoogleCode, a free service of the search engine company, Google. GoogleCode is a widely used service designed to assist open source software development by providing a repository for source code, revision control, program distribution, bug tracking, and various other very useful services.

The Smokeview manuals are typeset using LaTeX, specifically, PDF LaTeX. The LaTeX files are essentially text files that are under SVN (Subversion) control. The figures are either in the form of PDF or jpeg files, depending on whether they are vector or raster format. There are a variety of LaTeX packages available, including MiKTeX, a Windows version. The FDS-Smokeview developers edit the manuals as part of the day to day upkeep of the model. Different editions of the manuals are distinguished by date.

Changes made to Smokeview are tracked using revision control software. Not every change results in a change to the version number. The version number changes after it is judged that the source code changes warrant a version number change. For minor changes and bug fixes, incremental versions are released, referenced according to fractions of the integer version number. For example, version 5.1.4 would be a maintenance release of feature version 5.1, which in turn is an update within the major application release referred to as Smokeview 5. This happens every few weeks. A minor release from 5.2 to 5.3, for example,

might happen only a few times a year, when significant improvements have been made to the visualizations.

A suite of simple verification visualizations are routinely run to ensure that the daily changes have not altered any of the important visualization algorithms. The Smokeview verification suite is documented in Ref [8].

1.4 Capabilities

Smokeview is a program designed to visualize numerical calculations generated by the Fire Dynamics Simulator. Smokeview visualizes both dynamic and static data. Dynamic data is visualized by animating particle flow (showing location and *values* of tracer particles), 2D contour slices (both within the domain and on solid surfaces) and 3D iso surfaces. 2D contour slices can also be drawn with colored vectors that use velocity data to show flow direction, speed and value. Static data is visualized similarly by drawing 2D contours, vector plots and 3D level surfaces. Smokeview features in more detail include:

Particle Animations Lagrangian or moving particles can be used to visualize the flow field. Often these particles represent smoke or water droplets.

Color Contours Animated 2D shaded color contour plots are used to visualize gas phase information, such as temperature or density.

Animated Flow Vectors Flow vector animations, though similar to color contour animations (the vector colors are the same as the corresponding contour colors), are better than solid contour animations at highlighting flow features.

Animated Isosurfaces Isosurface or 3D level surface animations may be used to represent flame boundaries, layer interfaces and various other gas phase variables.

Realistic Smoke Smoke, fire and sprinkler spray are displayed realistically using a series of partially transparent planes.

This report describes some of the technical details involved in implementing these capabilities.

1.5 Overview

Smoke and other attributes of fire are visualized by Smokeview using both quantitative and realistic techniques. Smokeview uses quantitative display techniques such as 2D and 3D contouring. Realistic display of data refers to the intent of presenting the data in a form as it would actually appear. The first part of this document presents some of the details involved in using these techniques for displaying FDS generated data.

To implement these techniques, Smokeview uses various tools and techniques such as color, lighting, motion and transformation. These basic building blocks, discussed in Appendix 2, are used by each of the techniques Smokeview uses for visualizing data, in particular, smoke visualization. A more thorough treatment may be may be found in Martz's OpenGL Distilled [10], the OpenGL Programming Guide (Red book) [3] or the SuperBible [11]. Appendix A gives some of the programmatic details in particular giving the structure and naming conventions of some of the program source files and how the external libraries are organized and used. Appendix B gives the details of how the OpenGL components used by Smokeview interact with the host operating system. Finally, several miscellaneous topics such as image rendering and data compression are detailed.

Chapter 2

Basic Visualization Concepts

2.1 Defining Objects

Smokeview defines object locations in terms of a right handed coordinate system with the x axis pointing to the right, the y axis pointing to the back and the z axis pointing up (see Figure 2.1). Coordinate values as defined in an FDS input file for objects such as blockages, vents *etc.* are transformed internally within Smokeview to lie between 0.0 and 1.0. All Smokeview objects are drawn by grouping appropriately *OpenGL* vertices.

An OpenGL vertex has the same meaning as in geometry, a location. Vertices are specified in Smokeview using either **glVertex3f(x,y,z)** or **glVertex3fv(xv)**. The suffix for glVertex indicates the number and type of data value to be passed. For example, *3f* is used when passing 3 scalar floating points. The suffix *3fv* is used to when passing a pointer (or equivalently a memory address) to 3 floating point values.

To illustrate, suppose that `xyz` is a floating point array of size 3 containing a 3D coordinate. One could then use either of the two OpenGL calls:

```
glVertex3fv(xyz);
glVertex3f(xyz[0],xyz[1],xyz[2]);
```

to represent the vertex location.

Groups of vertices may be *built up* to form more complex geometric objects. Several objects are illustrated in Figure 2.2. They are formed by grouping vertices together and surrounding them with calls to `glBegin()` and `glEnd()`. The argument passed to `glBegin()` determines which higher level object is drawn. To draw a shaded triangle one would use

```
glBegin(GL_TRIANGLE);
glVertex3f(0.0,0.0,0.0);
glVertex3f(0.0,1.0,0.0);
glVertex3f(1.0,0.0,0.0);
glEnd();
```

Similarly, to draw points or to connect the vertices with lines (also shown in Figure 2.2) one would replace `GL_TRIANGLE` above with `GL_POINTS` or `GL_LINES` respectively.

The triangle is the fundamental construct Smokeview uses to visualize objects. To be useful though, these objects need to be colored, moved and projected onto a 2D terminal screen. These topics are discussed in the following sections.

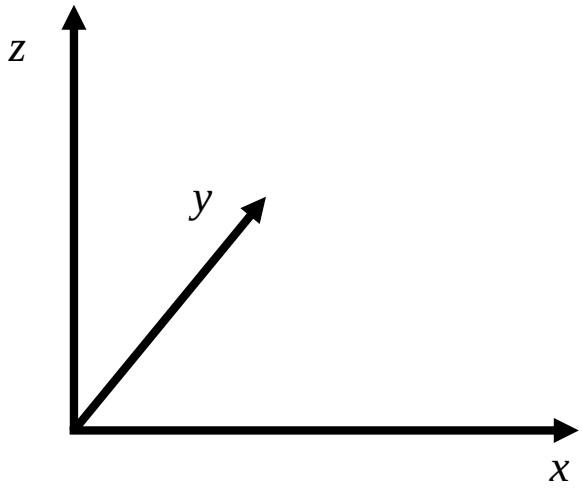


Figure 2.1: Right hand rule used by Smokeview for specifying a 3D vertex locations.

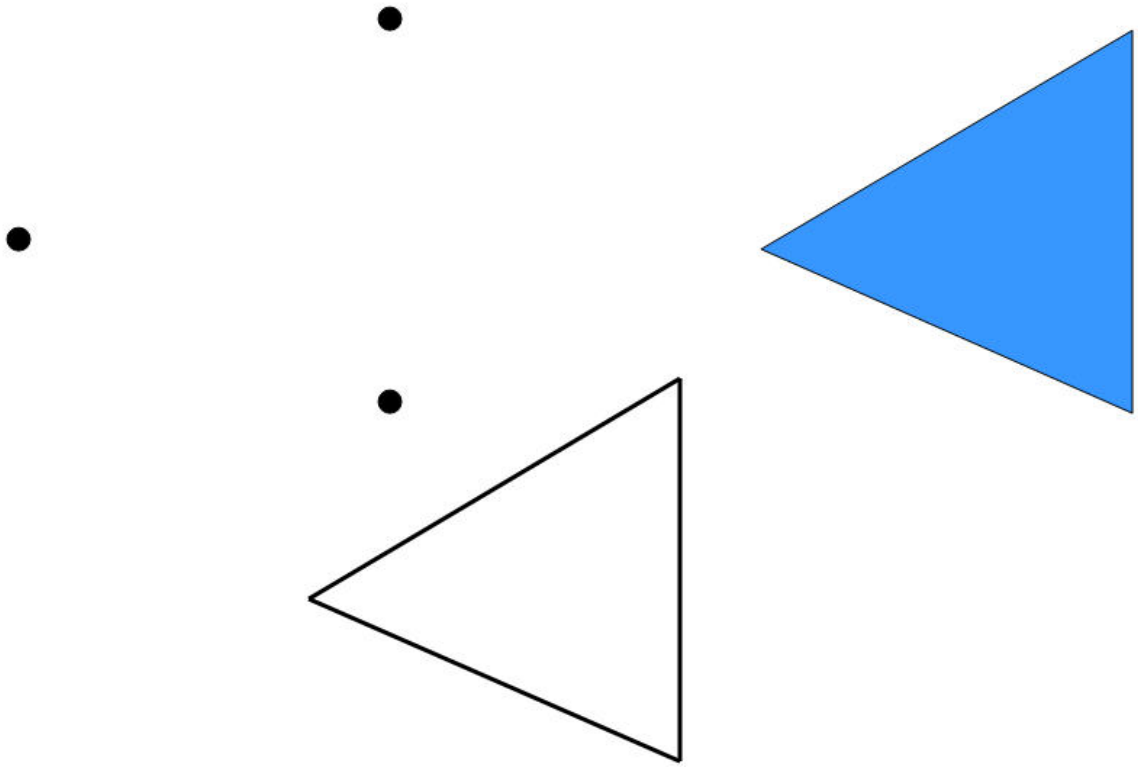


Figure 2.2: Points, lines and a shaded triangle drawn using OpenGL. Vertices are defined using `glVertex*` and the particular shapes are generated by passing `GL_POINTS`, `GL_LINES`, and `GL_TRIANGLE` to `glBegin()`

2.1.1 Setting the Scene

3D objects must be flattened or converted to 2D in order to display them on a computer screen. This occurs in two steps. The first step is called a projection. Smokeview uses three kinds of projections: perspective, orthographic and stereo. The second step takes whatever projection that has been applied and maps the resulting flattened geometry onto a subset of the computer screen. This window subset, usually a rectangle, is called a viewport. All viewports in Smokeview are rectangular.

2.1.2 Projections

Projections are used to flatten the 3D scene onto a two dimensional plane. Two common projections are orthographic and perspective. An orthographic projection is size preserving. Objects in the foreground take up the same amount of screen space as objects in the background. This projection is sometimes called a parallel projection because parallel lines in the 3D scene remain parallel when drawn on the screen.

A perspective projection creates an illusion of depth by causing objects in the background to take up less screen space than the same sized object drawn in the foreground. Both projection methods are available in Smokeview.

Smokeview uses `glFrustum` to perform perspective projections

```
glFrustum(
  (double)fleft,(double)fright,
  (double)fdown,(double)fup,
  (double)fnear,(double)ffar);
```

and `glOrtho` to perform orthographic projections

```
glOrtho(
  (double)fleft,(double)fright,
  (double)fdown,(double)fup,
  (double)fnear,(double)ffar);
```

where `fleft`, `fright`, `fdown`, `fup`, `fnear`, `ffar` are six clipping planes bounding the view frustum (truncated pyramid) for the perspective projection or the box in the orthographic projection. Drawing does not occur outside of the 3D region defined by these 6 clipping planes. An additional six clipping planes parallel to the x, y and z axes may be activated in Smokeview (using the clipping dialog box) to hide geometry making it easier to see interior objects or visualizations (hence the term clipping plane). OpenGL allows one to define clipping planes along arbitrarily oriented planes.

2.1.3 Stereo Projections

A stereo projection is simply a perspective applied twice making adjustments to simulate the viewpoint as seen for an observer's left and right eye. The two resulting versions of the scene can then be drawn in succession using shuttered glasses synched with the monitor to render the stereo effect. Alternatively, left and right versions of the scene may be drawn at the same time using stereo viewers to reveal the stereo effect. Figure 2.4 illustrates the two view frustums used for a stereo projection.

2.1.4 Viewports

A viewport is the particular portion of the screen where the drawing occurs. Smokeview defines separate viewports for drawing the title, time bar, color bar and the 3D scene. Figure 2.5 illustrates the relationship between the 3D scene and the 2D screen giving several viewport examples used by Smokeview.

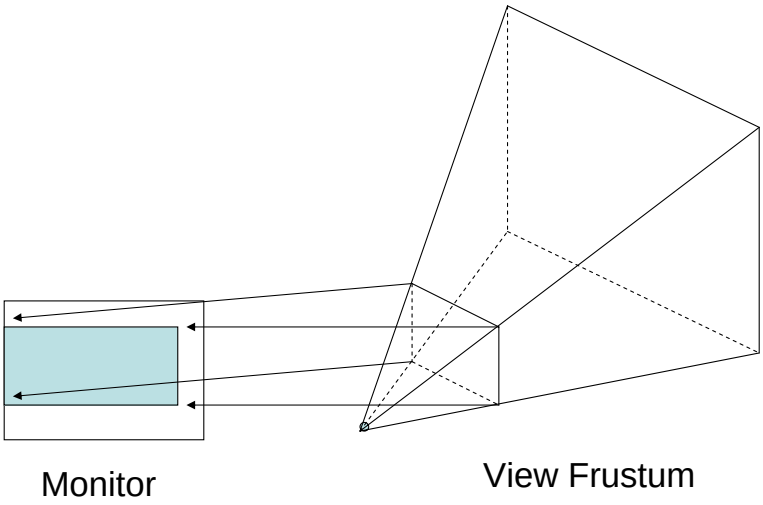


Figure 2.3: Example view frustum used to convert 3D scenes to 2D screen viewport.

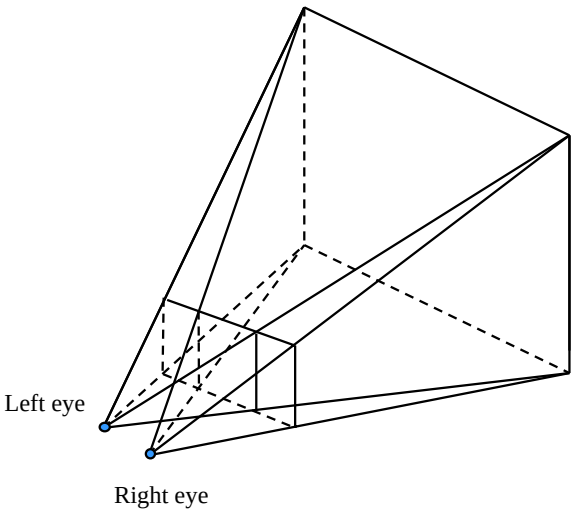


Figure 2.4: View frustums for stereo pairs.

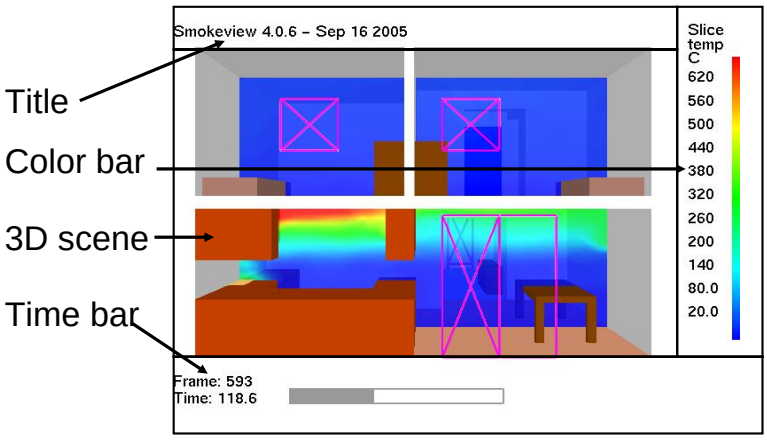


Figure 2.5: Examples of several viewports in a typical Smokeview scene.

Smokeview uses `glViewport()` to establish a window viewport. In particular, the call

```
glViewport(left,down,width,height);
```

is used to setup the portion of the screen where 3D drawing occurs where `left` and `down` are the bottom left screen coordinates of the viewport and `width` and `height` are the width and height of the viewport.

2.2 Coloring, Shading and Blending Objects

Color, shading and blending are three aspects of the same process used to draw objects residing in a three dimensional environment. Smokeview uses color to display and more importantly to distinguish between two or more geometric or data elements. Shading adds a further depth cue (besides perspective projections discussed earlier) to present the illusion of 3 dimensions. Using a virtual light source, colors are changed subtly based upon the relative orientation of the light source and the surface being drawn. Portions of the surface oriented away from the light are drawn darker than portions oriented towards the light. Blending is the process of combining the colors of two or more objects in such a way that objects in the background may be seen through semi-transparent objects drawn in the foreground. Together, color, shading and blending add to the illusion that the image drawn on a two dimensional terminal screen is three dimensional.

2.2.1 Color

Color in OpenGL is defined using four not three components. Besides the expected components of red, green and blue, OpenGL uses a component designated as *alpha* to represent opaqueness. Each component ranges from 0.0 to 1.0. An alpha component of 0.0 indicates a color that is completely transparent while an alpha component of 1.0 represents a color that is completely opaque.

Colors are specified using the OpenGL routine `glColorXXX()`. As with `glVertexXXX()`, XXX is replaced by a suffix used to indicate the number and type of argument passed. For the most part, Smokeview uses `glColor3f`, `glColor4f`, `glColor3fv` and `glColor4fv`. As before, the 3 or 4 indicates the number of data values used. If 3 is specified in the call then it is tacitly assumed that the fourth component, `alpha`, has value 1.0. In other words, the color is opaque. The `f` parameter indicates that floating point values are used and the `v` parameter indicates that the variable passed is a pointer to the actual data to be referenced.

2.2.2 Shading

OpenGL uses two shading models for drawing objects, flat and Gouraud. These models are specified using `glShadeModel(GL_FLAT)` and `glShadeModel(gl_SMOOTH)` respectively. Gouraud shading is also referred to as smooth shading. Flat shading assumes that objects are drawn in an environment with uniform lighting - light surrounds an object equally in all directions. Smooth lighting on the other hand assumes that light comes from a particular direction. This causes subtle changes in color to occur across an object's surface. Figure 2.6 shows examples of FDS blockages (the standard townhouse scenario) drawn using flat and smooth shading. Flat shading diminishes the three dimensionality of the scene.

A normal vector, another vertex attribute, is required to implement a smooth lighting scheme. A normal vector points in a direction perpendicular to the surface at the vertex. OpenGL uses this information to estimate the fraction of light from the light source reflected off of the given surface and intercepted by the observer. This is similar to a configuration factor calculation performed in fire modelling. The amount of light perceived by the observer depends on the relative orientation of the light source, the object (as specified by the location and normal vector of each vertex) and the observer.

For planar surfaces, the same normal vector is applied to each vertex defining the surface. For curved surfaces, normal vectors are determined using an average of the normal directions of faces surrounding the vertex. If normals are not averaged then the discontinuity in slope going from one face or triangle to another will result in a faceted or gem-like appearance. Figure 2.7 shows examples of drawing using non-averaged and averaged normals.

The Gouraud method for shading then determines a vertex color using the angle between the light source direction and the vertex normal vector. The color of the object being shaded is then determined by interpolating these colors.

Smokeview uses smooth shading or lighting to draw blockages and iso-surfaces. Particle, slice and boundary files are drawn without shading as are 3D smoke and Plot3D files.

2.2.3 Blending

Smokeview using OpenGL draws semi-transparent objects by combining or blending the color of the object currently being drawn with the color in the current background buffer.[1] The blending fraction is determined from the alpha color component of the object currently being drawn. A small alpha results in a small contribution from the currently drawn object color while a large alpha results in a large contribution. More precisely:

$$\text{updated background color} = \alpha \times \text{fragment color} + (1-\alpha) \times \text{original background color}$$

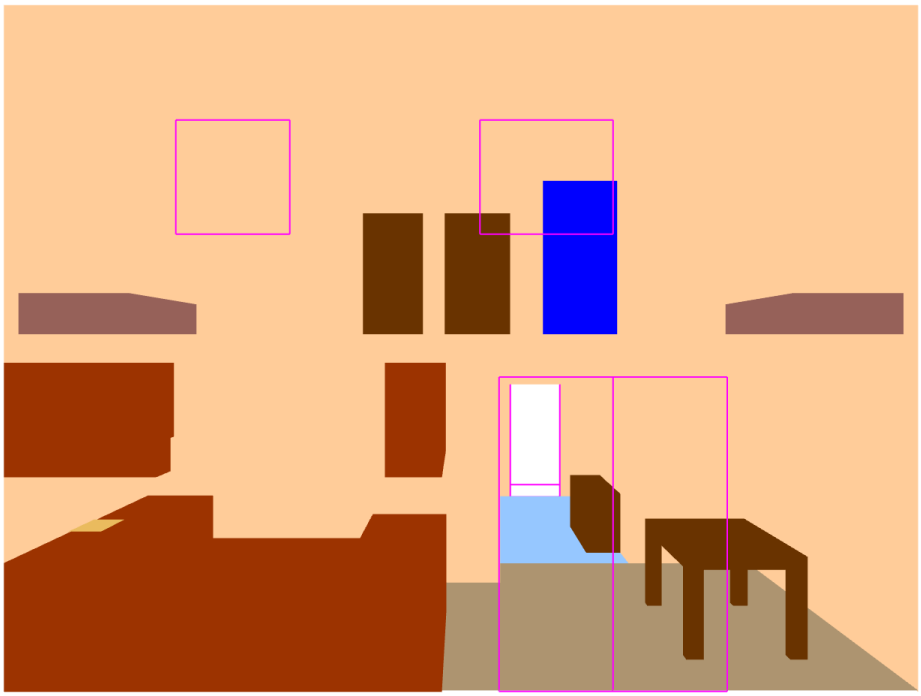
This particular blending model is the most common (there are others) and is activated in Smokeview using the OpenGL call

```
glBlendFunc(GL_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

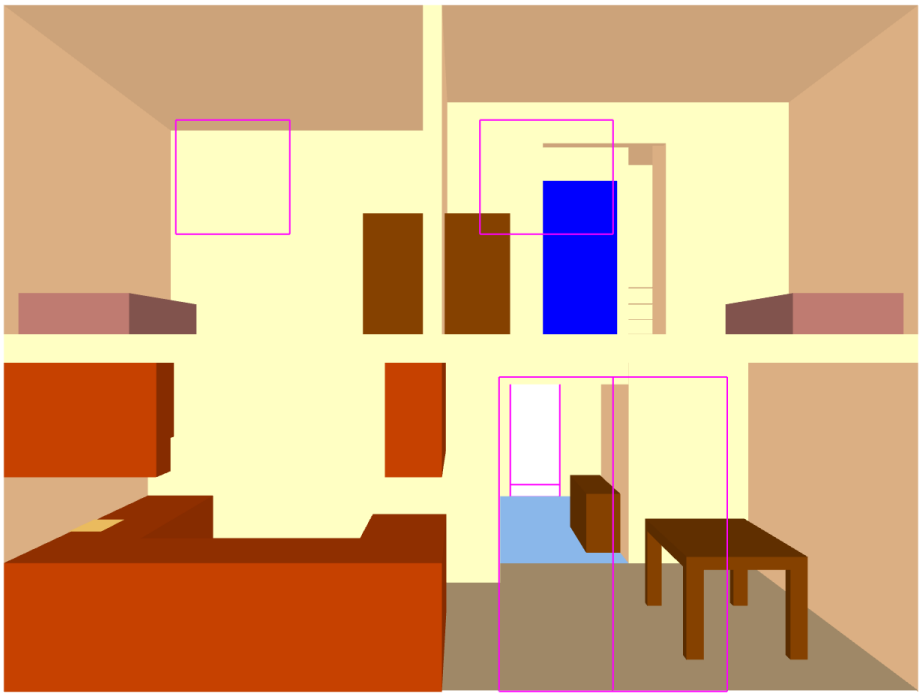
Choosing an alpha less than one allows one to *see through* objects. Smokeview uses this feature to implement partially transparent blockages and 2D animated slices. Slice files illustrated in Figure 2.8 are drawn using transparency. Smokeview also implements data chopping or hiding using blending or transparency. Data to be hidden is assigned an alpha value of 0.0 causing it to be completely transparent.

Complications arise, however, because this blending model is not commutative. The final color the user sees depends on the order in which the intermediate colors are drawn. To demonstrate this simply, consider

[1] The background buffer is updated as each object is drawn.



flat shading



smooth (Gouraud) shading

Figure 2.6: The FDS townhouse case drawn using flat and smooth shading. All blockage surfaces have identical colors when drawn with flat shading. When drawn with smooth shading, blockage colors change. Surfaces are darker when not in direct view of the light source adding to a sense of depth.

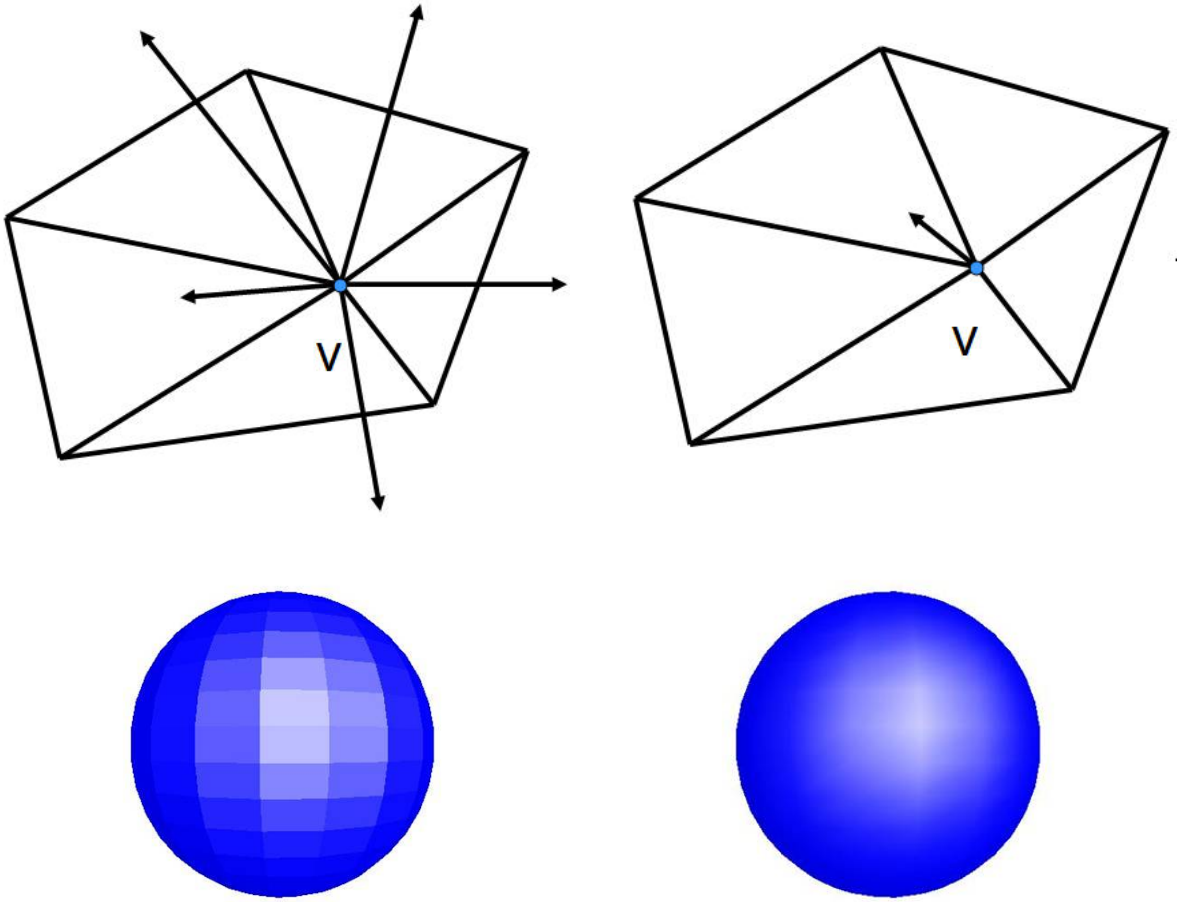
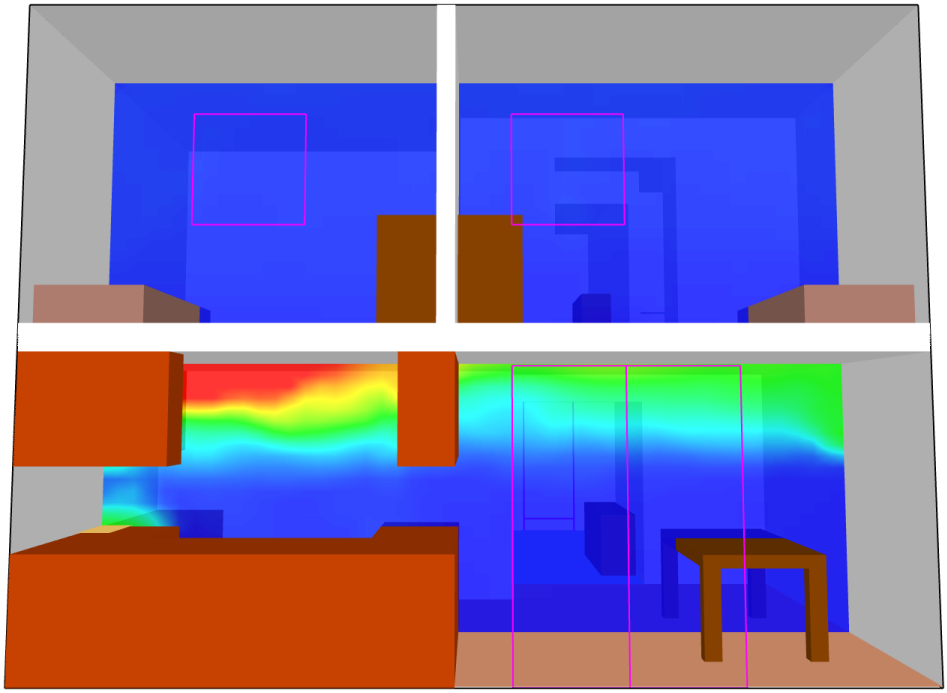
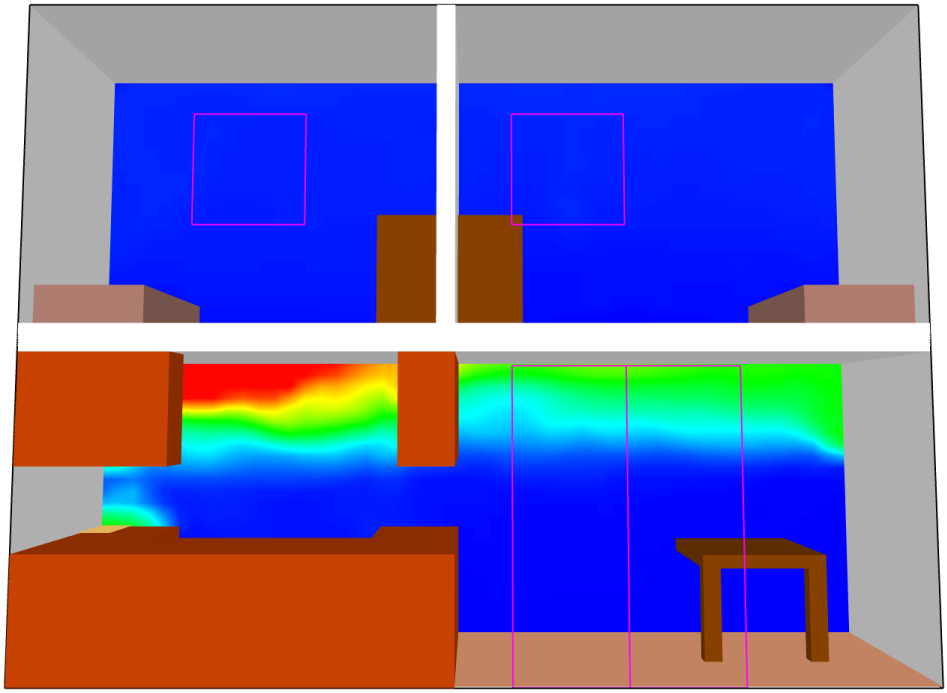


Figure 2.7: Two spheres drawn showing the effect of using averaged normals. Using non-averaged normals results in a faceted or gem-like appearance.



partially transparent slice plane



opaque slice plane

Figure 2.8: A slice file drawn transparently mixes or blends the slice colors with those in the background. When drawn opaquely, any portion of the scene behind the slice file is hidden.

two objects, one opaque and one semi-transparent. An opaque object drawn second and in front of a semi-transparent object will totaly obscure the semi-transparent object. A semi-transparent object drawn second and in front of an opaque object will blend with but not obscure the opaque object drawn previously.

To be more precise, consider two objects with colors c_1 and c_2 and alpha values α_1 and α_2. Let the background color be denoted b_0. Then blending object 1 with the background results in a new background color, b_1, given by

$$b_1 = (1-\alpha_1)b_0 + \alpha_1 c_1$$

Blending the color c_2 with b_1 results in a new background color, b_2, given by

$$\begin{aligned} b_2 &= (1-\alpha_2)b_1 + \alpha_2 c_2 \\ &= (1-\alpha_2)((1-\alpha_1)b_0 + \alpha_1 c_1) + \alpha_2 c_2 \\ &= (1-\alpha_2)(1-\alpha_1)b_0 + \alpha_1 c_1 + \alpha_2 c_2 - \alpha_2\alpha_1 c_1 \end{aligned}$$

Now blend colors c_1 and c_2 to the original background in the opposite order. Blending the color c_2 to with the background color b_0 results in a new background color, $\hat{b}_1$, given by

$$\hat{b}_1 = (1-\alpha_2)b_0 + \alpha_2 c_2$$

Blending c_1 to the interim background color $\hat{b}_1$ results in a new background color, $\hat{b}_2$, given by

$$\begin{aligned} \hat{b_2} &= (1-\alpha_1)\hat{b}_0 + \alpha_1 c_1 \\ \hat{b_2} &= (1-\alpha_1)((1-\alpha_2)b_0 + \alpha_2 c_2) + \alpha_1 c_1 \\ \hat{b_2} &= (1-\alpha_1)(1-\alpha_2)b_0 + \alpha_2 c_2 + \alpha_1 c_1 - \alpha_1\alpha_2 c_2 \end{aligned}$$

In general, $b_2 - \hat{b}_2 = \alpha_1\alpha_2(c_2 - c_1) \neq 0$, unless $c_1 = c_2$. Of course $b_2 = \hat{b}_2 = 0$ if $\alpha_1 = 0$ or $\alpha_2 = 0$ but this is a trivial case in which one of the two fragments is completely transparent.

The application to Smokeview is that since all smoke is drawn with the same color, smoke slice plane drawing is order independent. However, whenever fire is visualized along with smoke, multiple colors may exist in a given plane (black for smoke, organge for fire). In this case, the order that the planes are drawn becomes important.

In order then to prevent inconsistent drawing for the general case, opaque objects are drawn first then partially transparent objects are drawn next from back to front (from the point of view of the observer). Otherwise transparent objects may appear blended in front of objects when they should in fact be obscured. Of significance in Smokeview, is that slice plane ordering is not important when drawing 3D smoke since all smoke is drawn with the same color (but different opacities). However, order is important when drawing smoke and fire (two different colors) or if considering more elaborate lighting algorithms where in general every vertex may have a different color (due to lighting effects).

When lighting is not applied, colors within a triangle are determined in two steps using bi-linear interpolation. First, colors along a triangle edge are linearly interpolated using the two colors of the vertices bounding the edge. Second, the interior colors are determined along a horizontal scan line again using linear interpolation using colors previously interpolated on the triangle edge.

Flat shaded triangles may be drawn more efficiently but are not effective at visualizing a 3D effect. More sophisticated shading techniques are required and are discussed next.

2.3 Motion

Previous sections discussed how appearance is important in visualization. This section discusses how motion may be used to gain insight into fire phenomena. Motion may be thought of in two separate but equivalent ways - keeping the scene fixed and changing the observer's location and view direction or keeping the observer fixed and translating, rotating and/or scaling the scene.

Objects are moved or translated in OpenGL by applying a transformation to the current **modelview** matrix. The transformation is a matrix whose particular form depends on whether it is a translation, a rotation or a scaling.

A translation by (x,y,z) is performed by using `glTranslate3f(x,y,z)`. This OpenGL call generates the matrix

$$T = \begin{pmatrix} 1 & 0 & 0 & -x \\ 0 & 1 & 0 & -y \\ 0 & 0 & 1 & -z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

and applies it to the modelview matrix.

A rotation of θ degrees about the unit-vector axis, (x,y,z), is performed using `glRotatef(θ,x,y,z)`. A rotation of θ degrees about the x axis is performed using `glRotate(θ,1.0,0.0,0.0)`. This OpenGL call generates the matrix

$$R = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

and applies it to the modelview matrix.

It is of interest to rotate a scene between vectors u and v about an axis perpendicular to both vectors as illustrated in Figure 2.9. The rotation axis is formed by generating the cross-product of u and v, namely $u \times v$. The rotation amount θ is found using $cos(\theta) = u \cdot v/(||u|| \cdot ||v||)$. For the special case that u and v are unit vectors and $u = (0,0,1)$, a vector along the z axis, then θ and the rotation axis are given by

$$\begin{aligned} \theta &= \cos^{-1}(v_z) \\ u \times v &= (-v_y, v_x, 0) \end{aligned}$$

so that the OpenGL call to perform the desired rotation would be `glRotatef`$(180\cos^{-1}(v_z)/\pi, -v_y, v_x, 0)$.

A scaling of (a,b,c) meaning that x components are scaled by a, y components are scaled by b, and z components are scaled by c is performed using `glScale3f(a,b,c)`. This OpenGL call generates the matrix

$$R = \begin{pmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

and applies it to the modelview matrix. Smokeview uses scaling to view cases with large aspect ratios, tunnel fires for example.

This multiplication is usually performed in hardware by the video card. The modelview matrix is initialized to the identity matrix then multiplied by these translation, rotation or scaling transformation matrices as the scene is moved using the `glTranslate`, `glRotate` and `glScale` OpenGL calls.

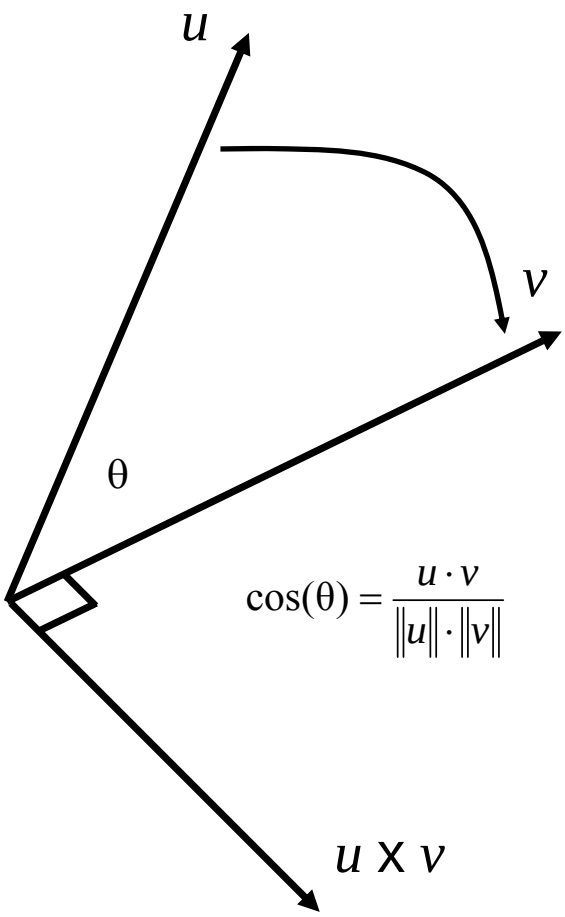


Figure 2.9: xxx

2.3.1 Smokeview Implementation

The desired translation and rotation amounts are communicated between the user and Smokeview using keyboard and/or mouse callback routines. Mouse motion is intercepted by the GLUT mouse callback routines in Smokeview and named `mouse` (first time mouse is clicked) and `motion` (when mouse is pressed and moving). GLUT *informs* the callback of the screen pixel coordinate which Smokeview uses to determine an elevation or azimuth angle or a translation amount depending on which control key (none, CTRL or ALT) is pressed. Smokeview translates and scales the coordinate system defined in an FDS scenario using

$$\begin{aligned}\hat{x} &= (x - x_{\min})/(xyz_{\max} - xyz_{\min}) \\ \hat{y} &= (y - y_{\min})/(xyz_{\max} - xyz_{\min}) \\ \hat{z} &= (z - z_{\min})/(xyz_{\max} - xyz_{\min})\end{aligned}$$

where $x_{\min}$ and $x_{\max}$ are the smallest and largest x coordinate considering all meshes. $y_{\min}$, $y_{\max}$, $z_{\min}$ and $z_{\max}$ are defined similarly.

The Smokeview scene is then translated in this coordinate system using `glTranslate` using the mouse position as passed to Smokeview by GLUT.

The Smokeview scene may be rotated about either a vertical or horizontal axis both passing through the scene center. The rotation angle about the horizontal axis (aligned perpendicular to the viewer's line of sight) is called an elevation angle and the rotation angle about the vertical axis is called an azimuth angle. These two angles and the translation amount are used by Smokeview to control the orientation of the scene. Rotations are then implemented in Smokeview using

```
glTranslatef(xcen,ycen,zcen);
glRotatef(YZ_AXIS_angle,cos(az_angle),-sin(az_angle),1.0);
glRotatef(XY_AXIS_angle,0.0,0.0,1.0);
glTranslatef(-xcen,-ycen,-zcen);
```

where `xcen, ycen, zcen` are the coordinates of the scene center, YZ_AXIS_angle is the elevation angle, and XY_AXIS_angle is the azimuth angle. The scene center (center of rotation) is translated to the origin, requested rotations are implemented, then the scene is translated back to its original location.

Chapter 3

Visualizing Data Quantitatively

Smokeview uses color and contours to visualize data quantitatively. Color is used to indicate *how much* of a quantity is at a given location whereas contours are used to identify *where* rather than *how much* a specified quantity occurs.

Coloring methods for visualizing data work by assigning colors based upon data values to various vertices within the scene. These vertices coincide with the FDS grid, either within a horizontal or vertical plane (slice or Plot3D file) or on a blockage exterior (boundary file). The video card then interpolates these colors at the interior of the figure, usually a triangle, formed by the vertices. Coloring variations result from differing methods for interpolating color between the vertices, either interpolating within the 3D color cube or using a 1D texture map (the colorbar).

Smokeview uses *marching* algorithms for generating contours, marching squares for generating 2D contours and marching cubes [12] for generating 3D contours. *Marching* algorithms work by reducing the general contouring problem to that of finding the contour in an elementary figure such as a square or a cube. Marching squares are used for finding 2D contours in a plane and marching cubes are used for finding isosurfaces in a volume. A contour for a region is then generated by splitting the region into a series of squares for 2D contours or a series of cubes for a 3D contour and finally assembling the contour acquired from all the parts.

3.1 Coloring Data

Several methods Smokeview uses to visualize data quantitatively involves converting data values to color. The basic procedure is to

1. obtain minimum and maximum data bounds to use in scaling, either through calculation or specification by the user,

2. map data onto integer indices between 0 to 255 ,

3. obtain colors using indices computed in step 2 to index into a color table (a numerical representation of the color bar)

4. display colors using 1D texture maps (or `glColorxx` in the case of particles).

These steps are detailed in the following sections. It is important to point out that the use of 1D texture maps in step 4 enables more detail to be obtained from the visualization due to the way that color interpolations between grid points are performed.

3.1.1 Determining Data Bounds

Smokeview uses three methods for setting data bounds. First, the bounds may be set by the user. This would be useful to ensure consistent coloring when several file types are displayed simultaneously (say slice and boundary files). A second method is to use *percentile* bounds (1st and 99th by default) which are useful when data outliers are present. To find percentile bounds, Smokeview scans the data computing a histogram. It then picks data bounds at a specified percentile levels, 1st and 99th by default. The third method for setting data bounds is to simply pick the global bounds for the data.

3.1.2 Converting data to a color

An index into a color table of size 256 is computed using

$$\text{color index} = \begin{cases} 0 & v < v_{min} \\ 1 + 253(v - v_{min})/(v_{max} - v_{min}) & v_{min} \le v \le v_{max} \\ 255 & v > v_{max} \end{cases}$$

where v_{min} and v_{max} are specified data bounds. Each table entry contains 4 components (red, green, blue and opacity). Each component is scaled from 0.0 to 1.0. The data in the colorbar data structure then defines colors (by default) as illustrated by the **bold path** in Figure 3.1. Other colorbars may be used, for example a color bar containing shades of grey from white to black.

3.1.3 Interpolating Colors

Consider the following code segment for drawing a shaded triangle with red, green and blue vertices:

```
glBegin(GL_TRIANGLE);
glColor3f(1.0,0.0,0.0);
glVertex3f(0.0,0.0,0.0);

glColor3f(0.0,1.0,0.0);
glVertex3f(0.0,1.0,0.0);

glColor3f(0.0,0.0,1.0);
glVertex3f(1.0,0.0,0.0);
glEnd();
```

OpenGL interpolates colors between the vertices and interior to the triangle using the color cube as in Figure 3.1. For example, if two vertices A and B are colored $(0.0, 1.0, 0.5)$ and $(1.0, 0.5, 0.0)$, then a point half way in between would be colored $(0.5, 0.75, 0.25)$, the average of the two colors. This color is the midpoint of the line segment AB interior to the color cube.

Smokeview version 4 (and earlier) uses this method, interpolating data within the 3D color cube (not the colorbar). As a result, near the fire or wherever there are large temperature gradients, interpolation artifacts occur. For example, if a red (1,0,0) region occurs near a blue (0,0,1) region, the interpolated color halfway in between would be (0.5,0.0,0.5), a shade of purple, not in the colorbar. In general, suppose that ci_j is an integer index between 0 and 255 and that $f(ci_j)$ is the ci_j'th color in the colorbar. Then the interpolated color between between the two colors $f(c_1)$ and $f(c_2)$ would be

$$\text{interpolated color} = (f(c_1) + f(c_2))/2$$

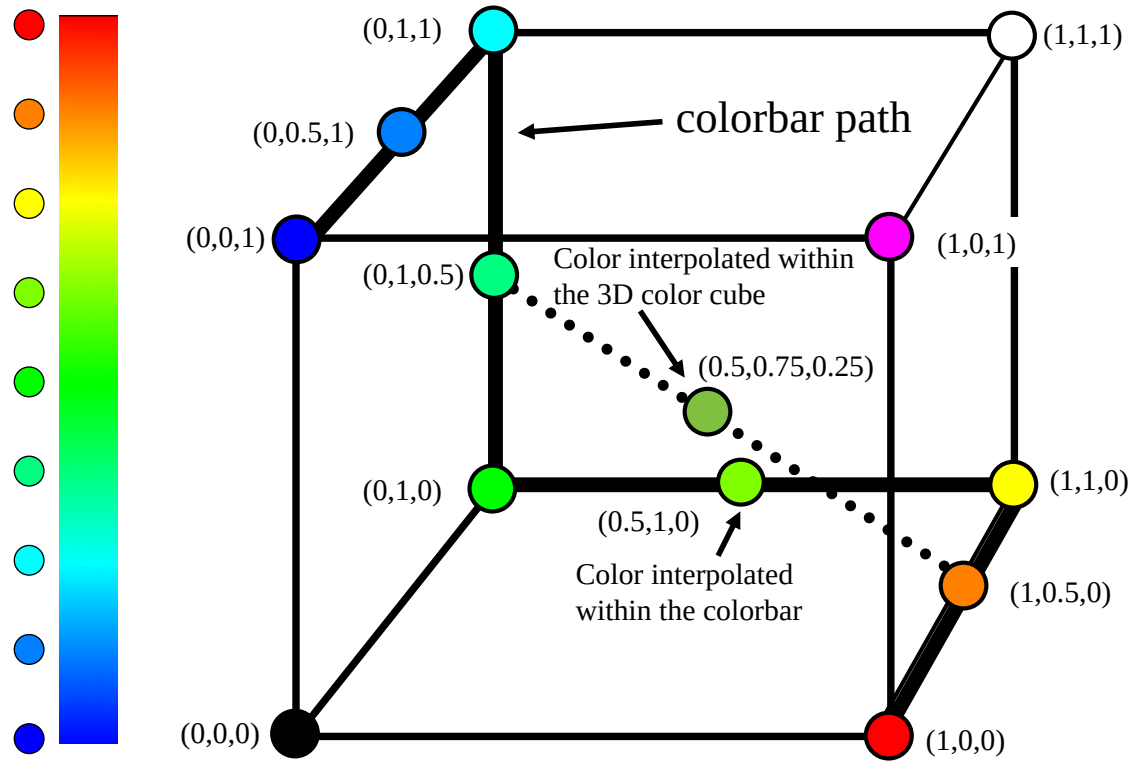


Figure 3.1: The 1D colorbar on the left is mapped onto the 3D color cube along the **bold path** from blue to cyan to green to yellow to red. Colors interpolated within the cube are different than colors interpolated within the colorbar.

This is not a good method for displaying colors related to data since only data on the colorbar have physical meaning. By using 1D texture maps, color indices are interpolated not colors. Therefore, colors displayed in data plots are always contained in the colorbar which is what we want.

Smokeview version 5 interpolates color indices not colors. As a result, interpolated colors are contained in the colorbar. This interpolation method is implemented using 1D texture maps. A 1D texture map is defined using the desired colorbar. A texture coordinate is assigned to each data vertex. Color indices at pixels between vertices are interpolated using the texture coordinate. In general, Smokeview version 5 uses the scheme,

$$\text{interpolated color} = f((c_1 + c_2)/2)$$

where c_1 and c_2 are color indices as before.

The following code segment sets up the use of 1D texture map.

```
glTexEnvf(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE,GL_REPLACE);
glEnable(GL_TEXTURE_1D);
glBindTexture(GL_TEXTURE_1D,texture_slice_colorbar_id);
```

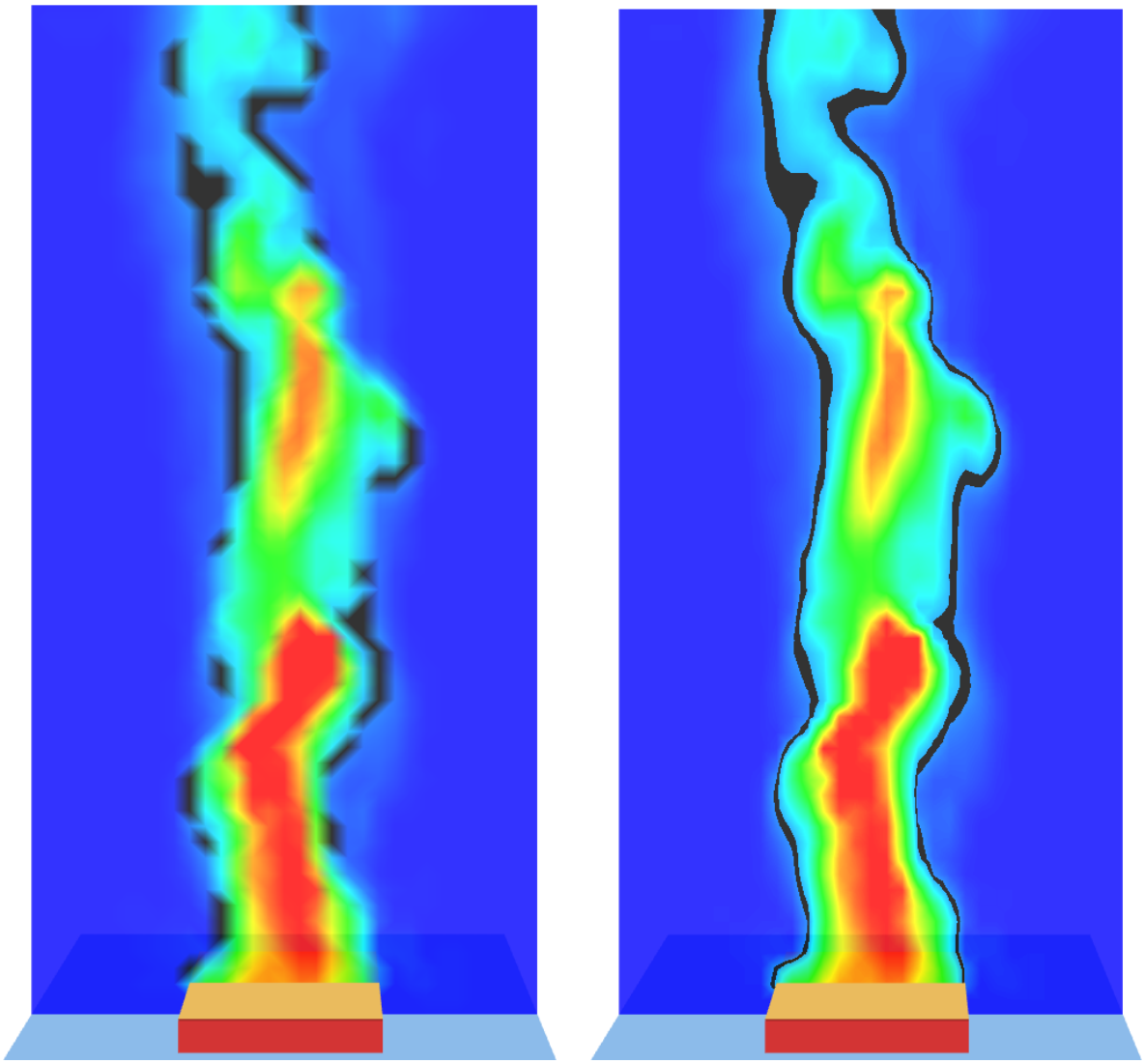
The following code segments (simplified) shows an example of drawing a slice in a YZ plane.

```
for(j=jmin; j<jmax; j++){
  for(k=kmin; k<kmax; k++){
    glTexCoord1f( r11); glVertex3f(xplane, yy1,  z1);
    glTexCoord1f( r31); glVertex3f(xplane,  y3,  z1);
    glTexCoord1f(rmid); glVertex3f(xplane,ymid,zmid);

    glTexCoord1f( r31); glVertex3f(xplane,  y3,  z1);
    glTexCoord1f( r33); glVertex3f(xplane,  y3,  z3);
    glTexCoord1f(rmid); glVertex3f(xplane,ymid,zmid);

    glTexCoord1f( r33); glVertex3f(xplane,  y3,  z3);
    glTexCoord1f( r13); glVertex3f(xplane, yy1,  z3);
    glTexCoord1f(rmid); glVertex3f(xplane,ymid,zmid);

    glTexCoord1f( r13); glVertex3f(xplane, yy1,  z3);
    glTexCoord1f( r11); glVertex3f(xplane, yy1,  z1);
    glTexCoord1f(rmid); glVertex3f(xplane,ymid,zmid);
  }
}
glEnd();
```

Figure 3.2 shows the old and new method for coloring. Note that the new method results in much crisper, clearer colors.

3.2 2D Contours

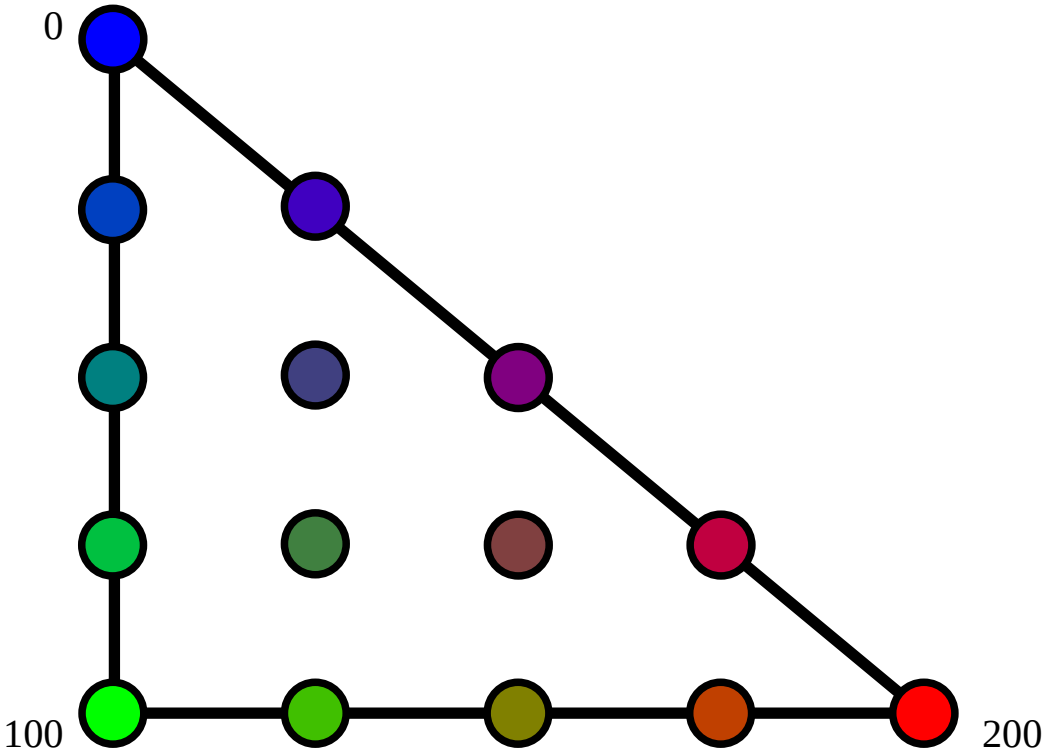
The *2D contouring problem* may be stated as: find the region in a 2D plane where a particular value exists *(line contour)* or an interval of values exist *(banded contour)*. In each case, the region to be contoured is divided into a series of squares. The problem is solved for each square and assembled to obtain the general solution. The square locations correspond to the grid set up in the FDS input file.



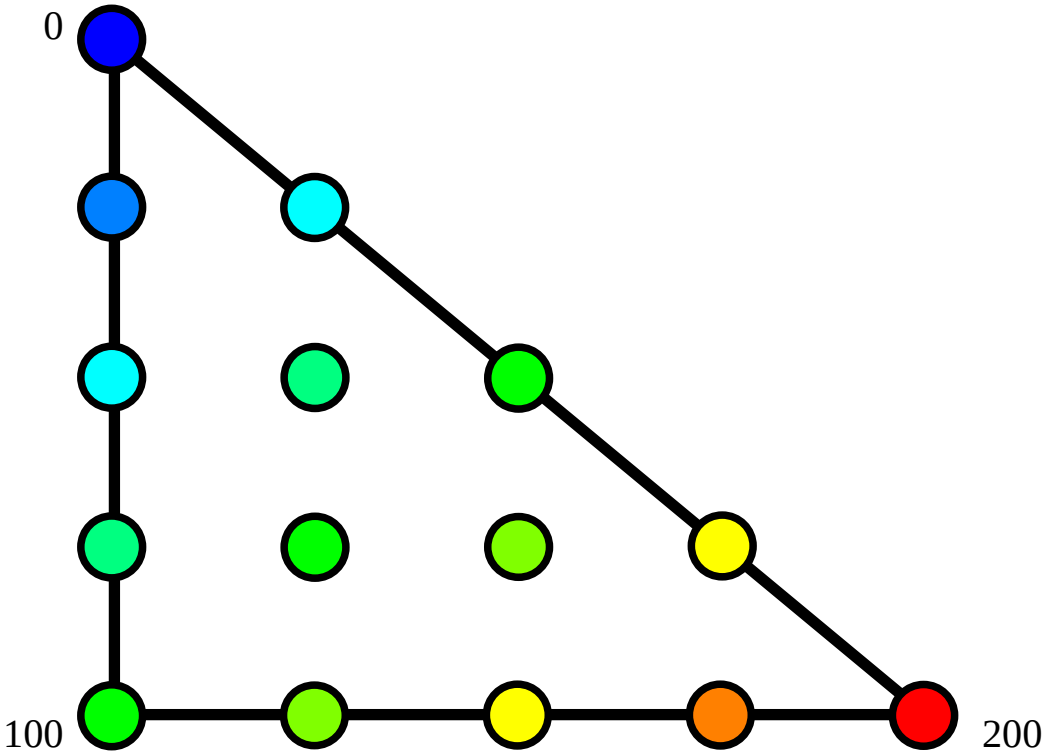
interpolate colors within a 3D color cube

interpolate colors within 1D texture color bar

Figure 3.2: Slice file snapshots illustrating old and new method for coloring data.



a) colors interpolated within the color cube



b) colors interpolated within the colorbar

Figure 3.3: Illustration showing colors representing data interpolated two different ways within a triangle: interpolated with the 3D color cube and interpolated with the colorbar

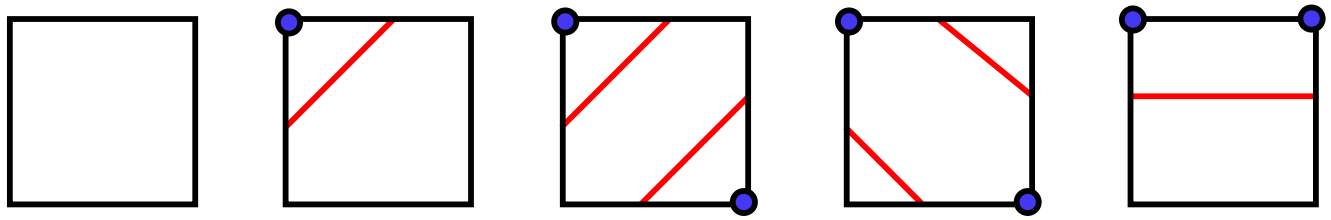


Figure 3.4: 2D line contour canonical forms.

3.2.1 Line contours

Mathematically, the 2D line contouring problem may be expressed as: for some level L, find the line(s) in the desired region where $f(x,y) = L$. Consider this problem for one square and assume that f is known at each square corner. The problem is solved by noting that the value of f at each of the four square corners is either greater than or less than or equal to the contour level L. There are then two states at each of the four corners. As a result, there are 16 cases to consider. These 16 cases may be reduced to 4 after accounting for various symmetries such as rotational, mirror and high/low. High/low symmetry refers to the observation that a case with one corner value above L will look the same as one with 3 corners above L. Figure 3.4 shows these four cases. Corners with blue dots indicate that the solution at this point is greater than L. Red lines indicate the line contouring solution. The algorithm may be summarized as follows:

1. Split the region to be contoured into a number of squares, each square aligned with the underlying FDS grid.

2. For each square:

   (a) Number the square corners as illustrated in Figure 3.4.

   (b) For each cell corner beginning with corner 0: assign 1 if its value exceeds L, 0 otherwise.

   (c) Use resulting 4 digit binary number $(0 \rightarrow 15)$ to determine the case number to be plotted. Using this numbering scheme, the cases are numbered from left to right 0 1, 5 and 3.

3.2.2 Banded contours

The banded contouring algorithm works similarly to the line contouring algorithm discussed previously. The 2D banded contouring problem may be expressed mathematically as given an interval $[L,H]$, find the 2D region where $L \leq f(x,y) \leq H$. The problem is solved by noting that the value of f at each of four square corners is either less than L, greater than H or in between. There are three states at each of four corners resulting in 81 cases to consider. These 81 cases may be reduced to 13 after accounting for various symmetries such as rotational, mirror and high/low. High/low symmetry is defined as before. Figure 3.5 shows these four cases. Each square corner is labeled with an L, M or H denoting that the value of f at that corner is below L, between L and H or above H respectively. The contoured region is bounded by black lines and contains a series of one or more red lines in the interior. The algorithm may be summarized as follows:

1. Split the region to be contoured into a number of squares, each square aligned with the underlying FDS grid.

2. For each square:

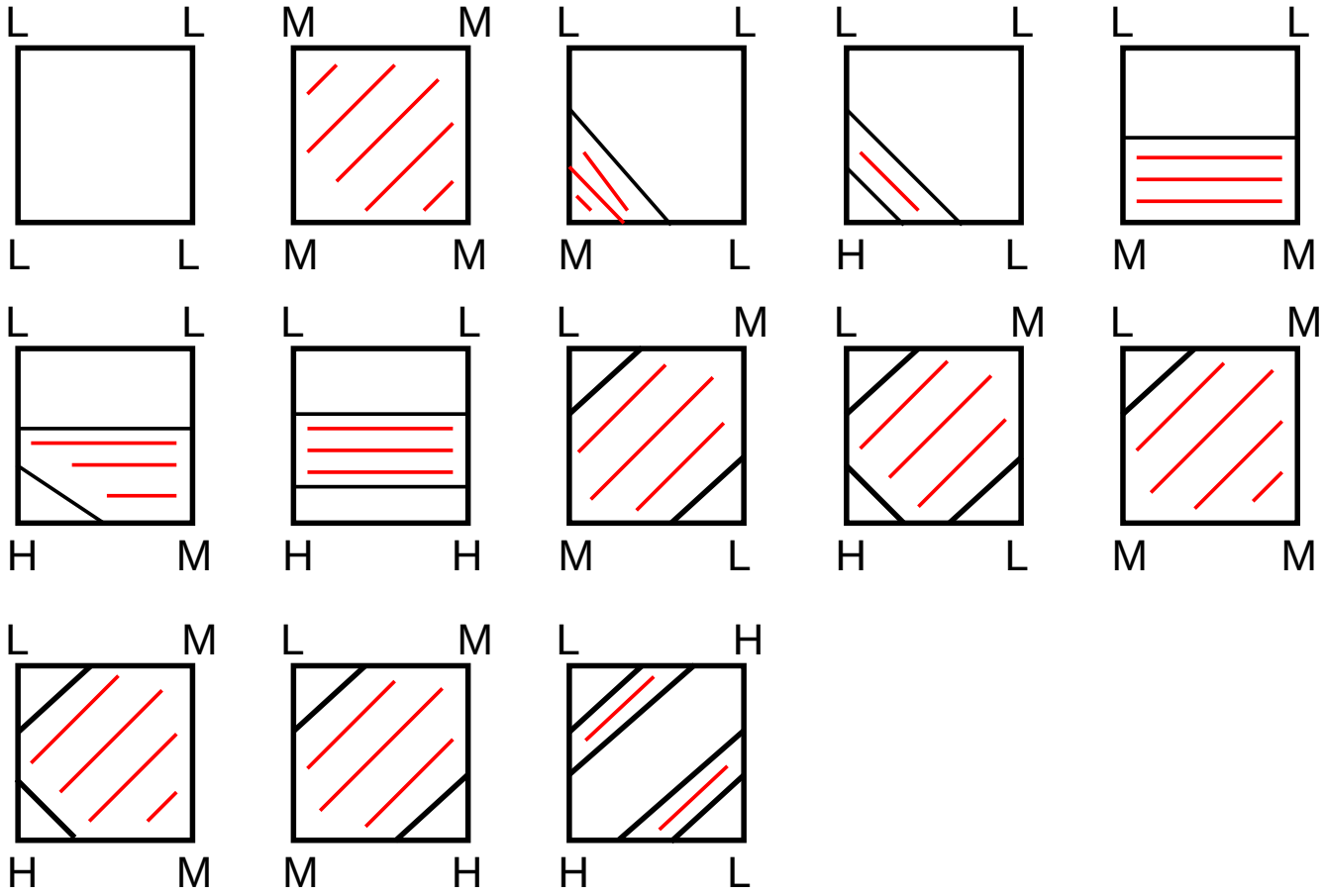


Figure 3.5: 2D band contour canonical forms.

(a) Number square corners as illustrated in Figure 3.5.

(b) For each cell corner beginning with corner 0: assign 0 if value is less than L, 1 if value is between L and H and 2 if value exceeds H.

(c) Use resulting 4 digit base 3 number $(0 \rightarrow 80)$ to determine the case number to be plotted. The case numbers in the first row are 0, 40, 1, 2 and 3. The case numbers in the second row are 5, 8, 10, 11 and 13. The case numbers in the last row are 14, 16 and 20.

3.3 3D Contours - Isosurfaces

An isosurface is a surface in 3-D space that defines constant values of a dependent variable. For example, FDS uses a mixture fraction model to simulate combustion. In FDS versions 2 through 4 this model considered a critical or stoichimetric mixture fraction value, such that regions greater than the critical value are fuel rich and regions less than the critical value are fuel lean. Burning then occurs, according to the model in versions 2 through 4, on the level surface where the mixture fraction equals this stoichimetric value. Therefore, for those versions of FDS it is of interest to visualize these locations. This is done using animated isosurfaces.

The isosurfaces are generated at each desired time step using a marching cube algorithm [12] modified to remove ambiguities. A decimation procedure is used to reduce the number of resulting triangles by collapsing nodes of triangles with large aspect ratios and re-triangulating. This makes the isosurface look better and also reduces storage requirements. Figures 3.6 and 3.7 illustrates the use of isosurfaces for visualizing the stoichimetric mixture fraction. These figures show different ways of drawing isosurfaces.

Isosurface uncertainty Data used to generate isosurfaces has uncertainty, therefore the location of the isosurface also has uncertainty. Bounds for this uncertainty may be expressed in terms of data uncertainty and data variation (the smaller the variation the larger the uncertainty). The Smokeview verification guide [8] shows that the uncertainty in isosurface location, Δx, between two node, data value pairs (x_1, T_1) and (x_2, T_2) may be bounded using

$$\left| \frac{\Delta x}{x_2 - x_1} \right| \leq \frac{\max(|\Delta T_1|, |\Delta T_2|)}{|T_2 - T_1|}$$

where ΔT_1 and ΔT_2 are the uncertainty in T_1 and T_2 at x_1 and x_2 respectively.

Algorithm for generating an Isosurface The algorithm for generating an isosurface may be summarized as follows:

1. Split region to be contoured into a number of cubes, each cube aligned with the underlying FDS grid.
2. For each cube:
   (a) Number cube corners from 0 to 7 as illustrated in Figure 3.8.
   (b) For each cube corner beginning with corner 0: assign 1 if value exceeds L, 0 otherwise.
   (c) Use resulting 8 digit binary number $(0 \rightarrow 255)$ to determine the case number to be plotted.
   (d) Triangulate cube surface, storing vertex locations and edge numbers according to case
3. Triangle decimation. Eliminate small triangles (any triangle with two or more vertices *closest* to the same node). Replace removed triangle vertices with a vertex with coordinates that are the average of the three removed vertices coordinates. Re-triangulate using this new vertex. This process is illustrated in Figure 3.9.
4. Estimate normal directions (to be associated with a vertex). For each vertex:
   (a) determine triangles sharing the vertex
   (b) determine the normal vector (normalized with length 1.0) of each triangle using the vertex
   (c) construct a harmonic weighted average of triangle normal vectors where all weights sum to 1.0 and each weight is inversely proportional to the corresponding triangle area
   (d) store the resulting average along with vertex data

Vertex normals in 3D An averaged normal vector, V_i, for vertex i may then be determined by summing over all triangles n connected to a vertex i as in

$$V_i = \sum_n \frac{U_n}{A_n} / \sum_n \frac{1}{A_n} = \sum_n \frac{u_n \times v_n}{||u_n \times v_n||^2} / \sum_n \frac{1}{||u_n \times v_n||}$$

where for the n'th triangle, u_n and v_n are two sides, $A_n = ||u_n \times v_n||$ is the triangle area and $U_n = u_n \times v_n / A_n$ is a unit vector normal to the triangle.

Vertex normals in 2D The use of inverse area weights or harmonic averages to construct average normal vectors at isosurface vertices is justified by the following two dimensional example. As illustrated in Figure 3.10, consider the quadratic $y(x) = A + Bx + Cx^2$ with $y(-\Delta x_2) = 0$, $y(0) = 1$ and $y(\Delta x_1) = 0$. The slope at $x = 0$ is given by $y'(0) = B$. Likewise, the slope of the vector perpendicular to this curve at $x = 0$ is $-1/B$. The coefficient B may be determined from the two simultaneous equations $y(\Delta x_1) = 0$ and $(y(-\Delta x_2) = 0$ (note that $A = 1$ since $y(0) = 1$) or

$$\begin{aligned} 1 + B\Delta x_1 + C\Delta x_1^2 &= 0 \\ 1 - B\Delta x_2 + C\Delta x_2^2 &= 0 \end{aligned}$$

which has solution

$$B = \frac{\Delta x_1^2 - \Delta x_2^2}{\Delta x_1 \Delta x_2 (\Delta x_1 + \Delta x_2)} = \frac{\Delta x_1 - \Delta x_2}{\Delta x_1 \Delta x_2}$$

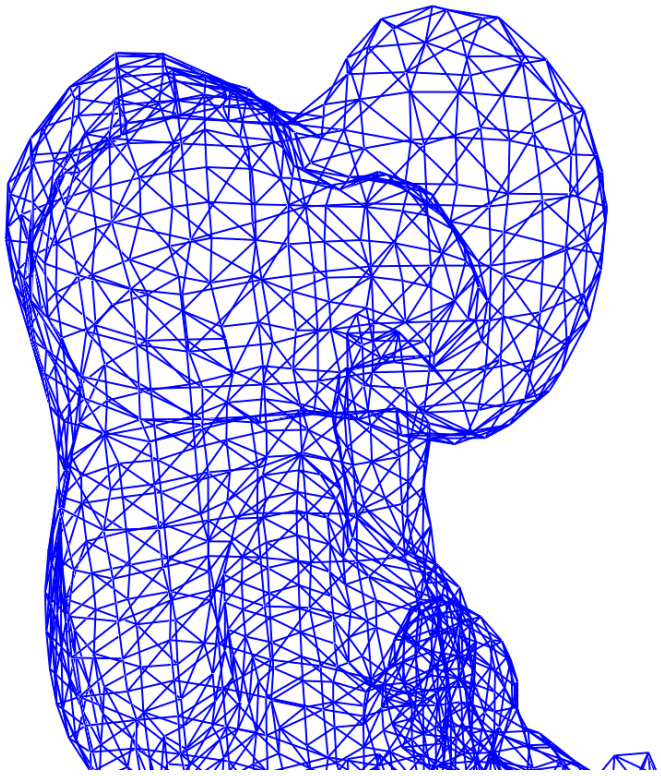
The slope N of the normal at $x = 0$ is then given by

$$N = -\frac{1}{B} = \frac{1}{1/\Delta x_1 - 1/\Delta x_2}$$

The slope N_1 of the normal to the line segment between $(0,1)$ and $(\Delta x_1, 0)$ is $N_1 = \Delta x_1$. The slope N_2 of the normal to the line segment between $(-\Delta x_2, 0)$ and (0,1) is $N_2 = -\Delta x_2$. Therefore the average slope N may be expressed in terms of N_1 and N_2 as

$$N = \frac{1}{1/N_1 + 1/N_2}$$

Figure 3.6: Snapshot of an isosurface of temperature at 100 °C (212 °F).



outline view

solid view

Figure 3.7: Snapshot of an isosurface of temperature at 100 °C (212 °F).

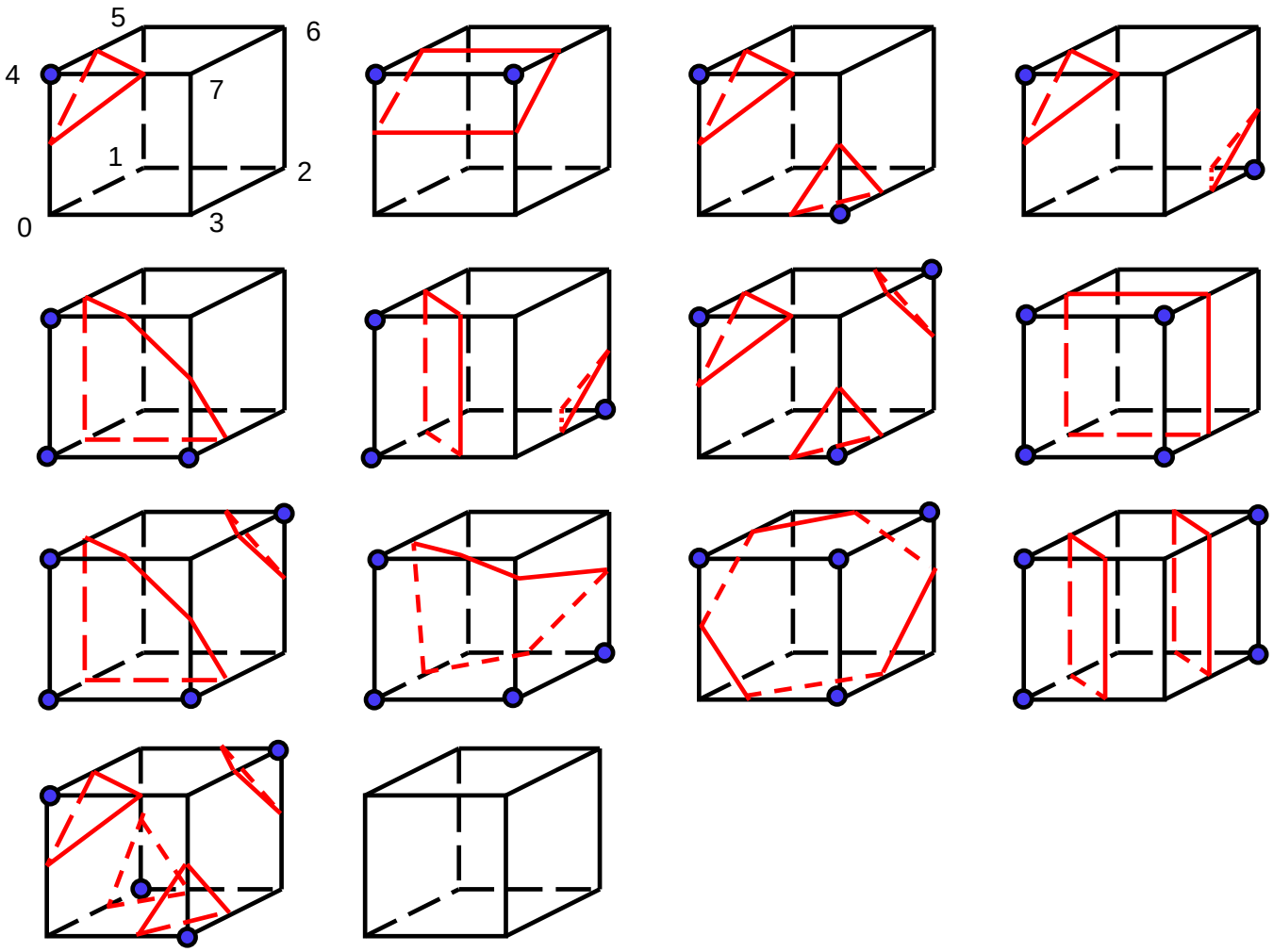
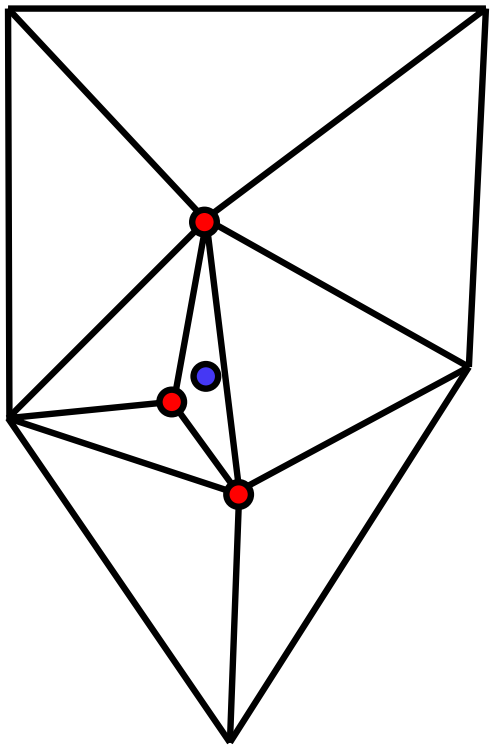
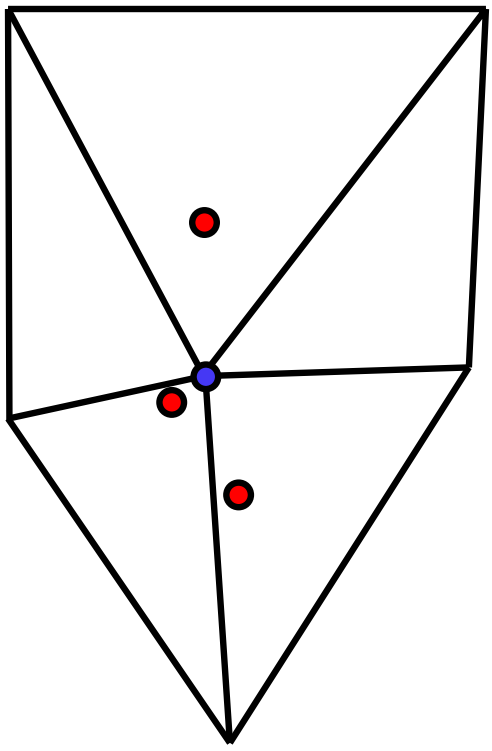


Figure 3.8: 3D isosurface canonical forms. Dots occur at corners where the data value is greater than the isosurface value. Other corners are below the isosurface value. Red polygons intersect cube edges at the isosurface value. The red polygon (isosurface) NEVER intersects an edge with two or zero dots on the ends.



before decimation



after decimation

Figure 3.9: Example of triangle decimation. Triangle with red dots is removed. Region is re-triangulated by replacing any edges connected to a red dot with the blue dot (average position of removed red dot).

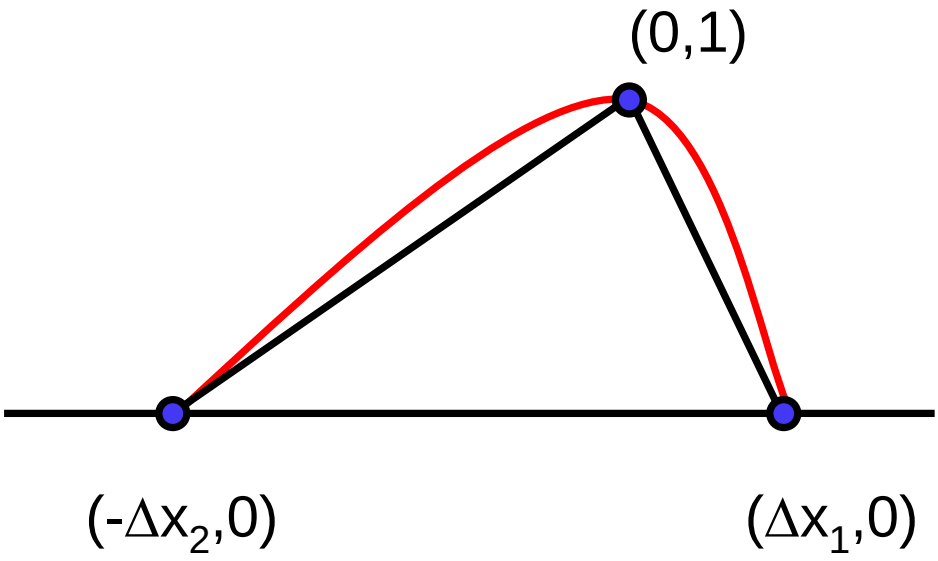


Figure 3.10: Setup for determining the slope of a smooth curve passing through three points.

Chapter 4

Visualizing Data Realistically

This chapter discusses two methods Smokeview uses for representing smokeview flow data realistically. The first involves particles, representing smoke flow by tracking tracer particles as they move through the simulation influenced by an underlying velocity field. This velocity field and the particle motion is computed by FDS and communicated to Smokeview using data files. This was the first method Smokeview implemented for visualizing smoke flow and was inspired by the software tool named Frames written by James Sims.

The second method for visualizing smoke flow involves a method known as volumetric rendering. Smoke flow is visualized using the optical properties of soot noting that that the amount of light obscured by smoke and hence its shade of grey depends on the amount of smoke between the observer and the background. Smokeview then integrates the smoke opacity one grid plane at a time using a simple implementation of Beer's law in the video hardware.

4.1 Particles Systems

4.1.1 Massless Particles

FDS uses particles as tracer elements to allow one to visualize flow. FDS also uses particles as droplets to model fire suppression. Particle positions are determined in FDS using the differential equation

$$\frac{dx_p}{dt} = V(x_p, t)$$

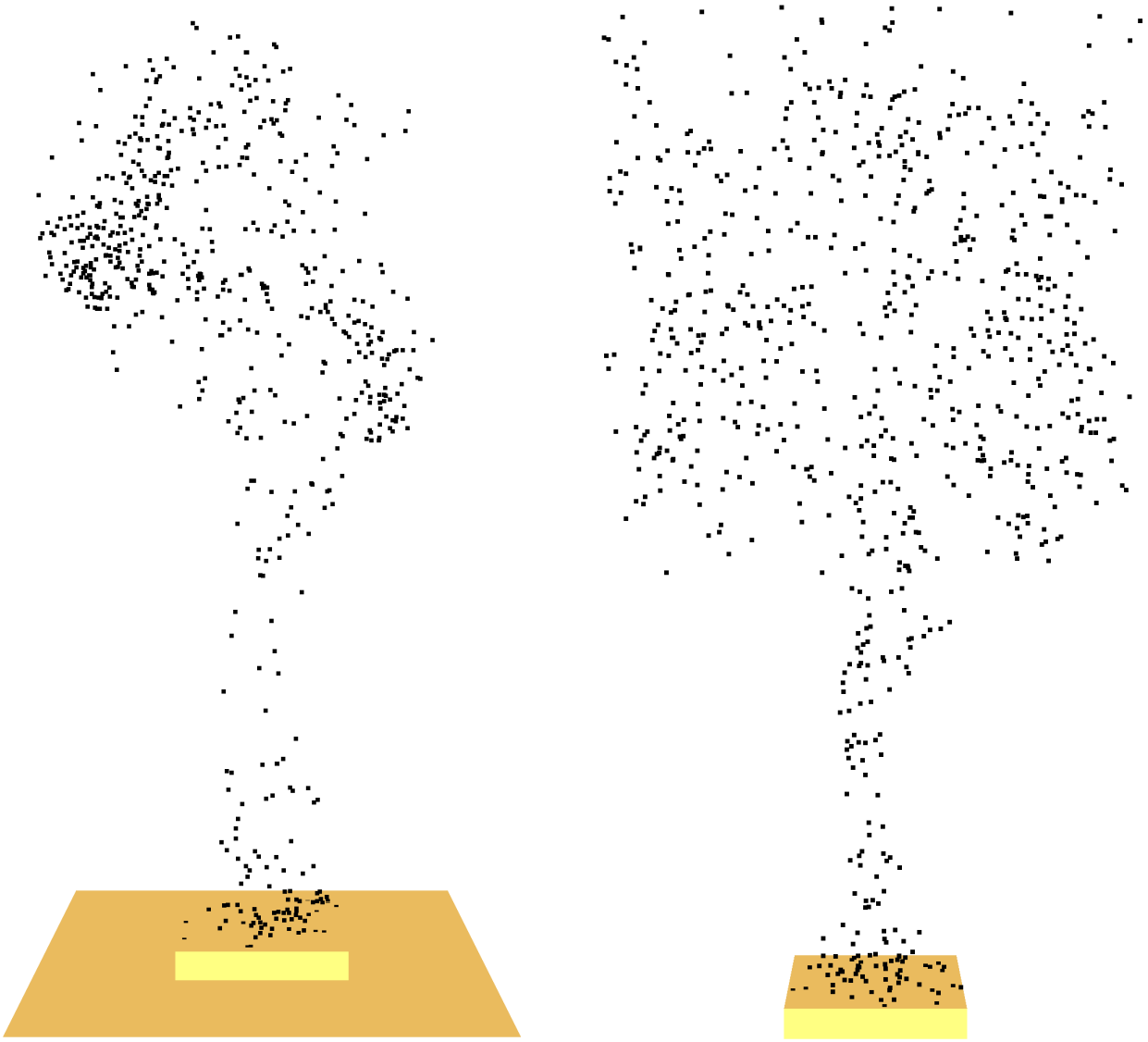
where x_p represents particle position and V represents the velocity field.

The assumption this model uses to compute particle flow is that the the drag force on the particle is large compared to mg, the force due to gravity.

Figures 4.1 and 4.2 shows particles represented as particles and as streaks. A particle streak represents where a particle is *now* and in the recent past. The streak length is specified as time. Static particle images are not effective at displaying motion. For example, the particle image in Figure 4.1 does not show particle motion. Streak lines, however, in Figure 4.2 shows curved motion due to the upper wall obstructions. Particle size is specified using

```
glPointSize(partpointsize);
```

The floating point value of `partpointsize` is specified using the `File/Bounds` dialog box. Streak line length is specified in terms of time, a certain number of seconds before the current display time. Streak length is also specified in the `File/Bounds` dialog box.



a) Open plume

b) Partially obstructed plume

Figure 4.1: Particle view of two plume flow cases. The case on the left is completely open. The upper half of the simulation on the right is obstructed.

a) Open plume

b) Partially obstructed plume

Figure 4.2: Streak view of two plume flow cases. Streak views show plume motion resulting from hidden obstructions while particle views do not. The case on the left is completely open. The upper half of the simulation on the right is obstructed.

4.1.2 Tracking Particles/Objects with Mass

4.2 Volumetric Methods

Visualizing smoke realistically is challenging for three reasons. First, the storage requirements for describing smoke throughout the simulation scene at every time step can easily exceed the disk size capacities of present 32 bit operating systems which would typically be 2 GB. Second, the computation required both by the CPU and the video card to display each frame can easily exceed 0.1 s, the time corresponding to a 10 frame/s display rate. Finally, the physics required to describe smoke and its interaction with itself and surrounding light sources is complex and computationally intensive. Approximations and simplifications are required.

Smoke visualization techniques described previously, such as the use of tracer particles or shaded 2D contours are useful for analyzing data quantitatively, but are not suitable for applications where realism is required. Some examples of such applications are using Smokeview as a virtual fire fighter trainer or using Smokeview to examine the obscuration effects of smoke on an outdoor environment.

The approach used by Smokeview for visualizing smoke realistically is similar to that taken in Fedkiw *et. al.* [13] except that interactions with smoke and light are not considered (only the effects of smoke obscuration are visualized). The video hardware is exploited to perform a simple obscuration calculation by using OpenGL to display a series of partially transparent parallel planes. The planes are oriented to be perpendicular to the viewer's line of sight. The transparencies are computed based on physics using data derived from an FDS calculation. Vertices in each plane are colored black or dark grey based on the estimated smoke albedo. The vertices are also assigned an OpenGL α opaqueness parameter. The assigned value depends on the optical smoke thickness, with 0.0 used for completely transparent smoke and 1.0 for completely opaque. Figure 4.3 gives a flowchart describing the process FDS uses to pre-compute smoke data and Smokeview uses update and draw the data.

4.2.1 Computing Opacity

The process of computing opacity is illustrated schematically in Figure 4.4. A ray travels from the background to the observer through intervening smoke. At each step towards the observer, light is absorbed, scattered or emitted by the smoke. Emission effects are ignored. Scattering effects presently are only accounted for in the value of the total mass extinction coefficient used. Contributions are assumed to be from both absorption and scattering. The obscuration is computed along each ray one grid plane at a time, using Beer's law as follows. The α values are pre-computed by FDS using Beer's law [14]

$$\alpha = 1 - \exp(-ks\Delta x) \tag{4.1}$$

for a particular view direction (down the x axis) where Δx is this distance between two nodes, k is the total soot mass extinction coefficient and s be the soot density. Beer's law is an empirical relationship relating light absorption to the material properties of the media the light is traveling through, in this case soot or smoke.

The α parameter in equation (4.1) is used by OpenGL to blend smoke planes with the current background, the same blending process discussed in Section 2.2.3. The α parameter used here also represents an opacity, 0.0 for completely transparent, 1.0 for completely opaque.

4.2.2 Adjusting Opacity

The absorption parameter, α, needs to be adjusted when the view direction is not aligned along the axis orthogonal to the viewing planes (as in Figure 4.5), the distance between adjacent smoke planes changes, or viewing planes are skipped.

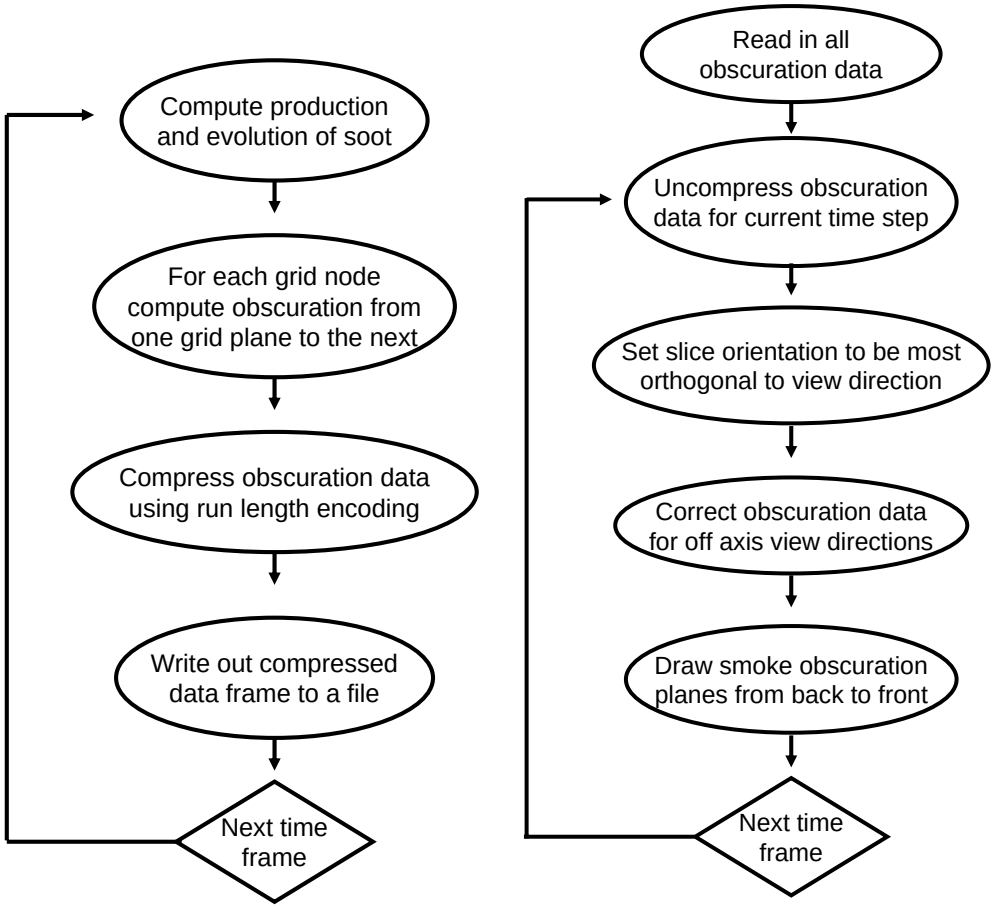


Figure 4.3: Flowchart of FDS and Smokeview computations for visualization smoke realistically. FDS computes obscuration parameters from one grid plane to the next. Smokeview corrects these parameters to account for non-axis aligned view directions.

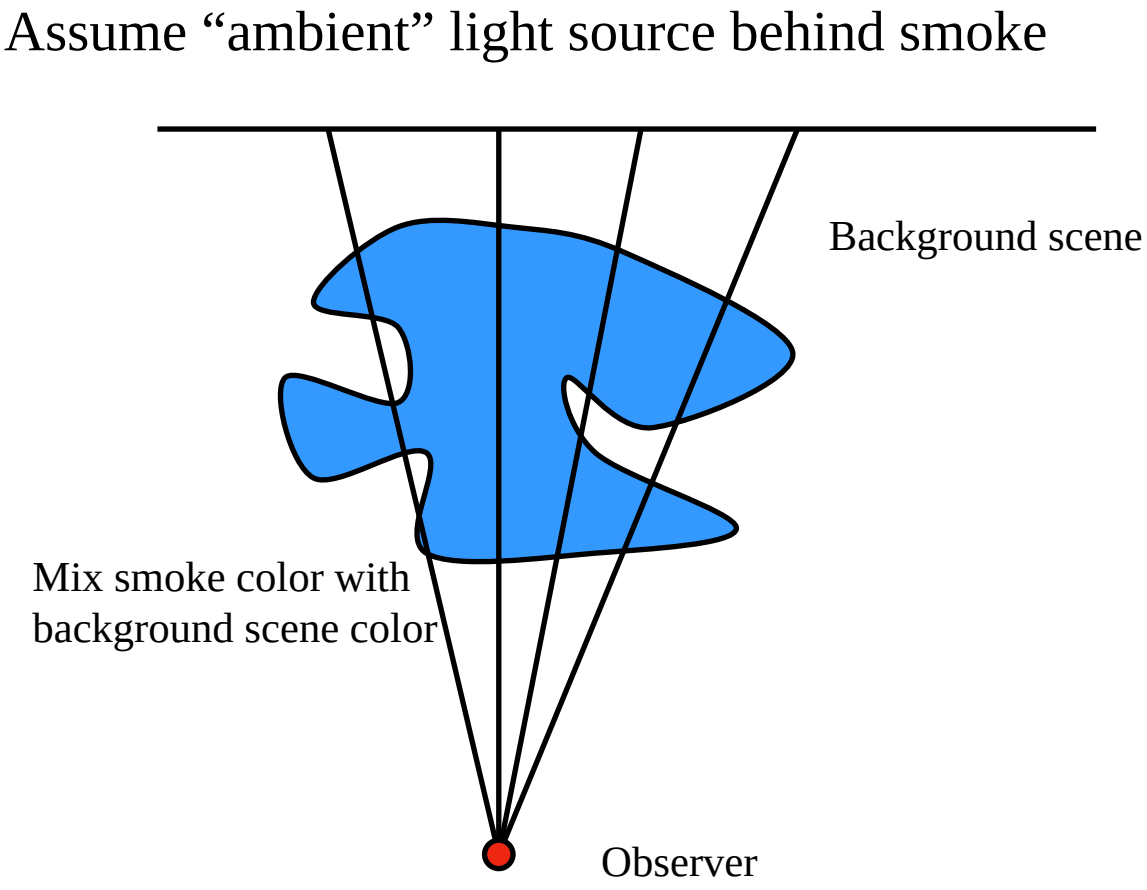


Figure 4.4: Smoke is drawn by blending the smoke color with the background color. The amount of blending depends on the amount of obscuration as determined from soot density and path length.

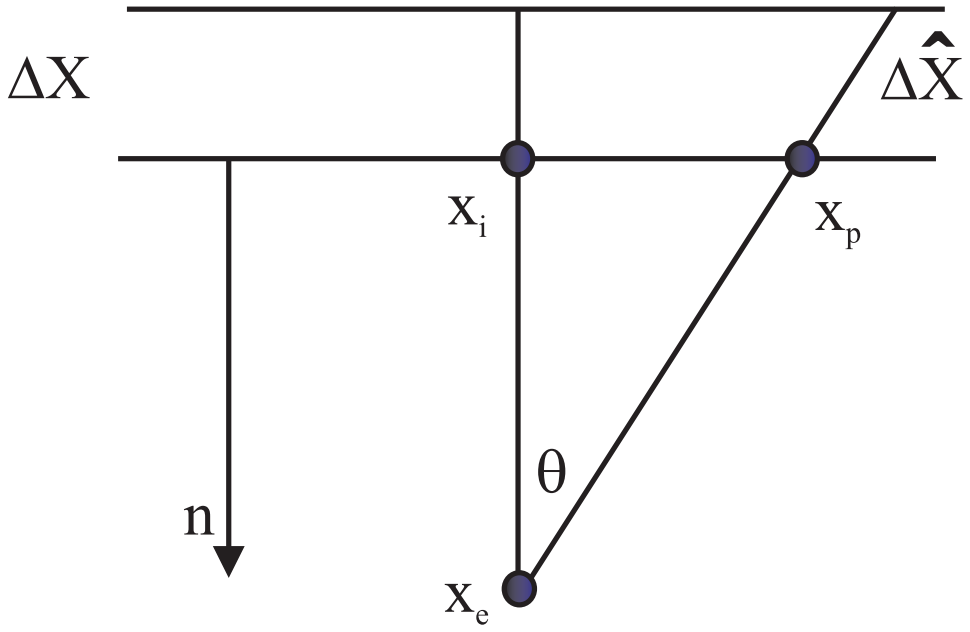


Figure 4.5: Diagram illustrating adjustment needed to opaqueness parameter, α, for non axis aligned views. The α value along the ray containing the $\hat{x}$ segment needs to be larger to account for the longer path length.

Ten million exponential operations per second are required to display smoke with corrected α's at 10 frames per second if the simulation has grid dimensions of $100 \times 100 \times 100$. Recent advances in CPU and video hardware makes these types of visualizations possible. These corrections may also be performed in the video card (GPU), resulting in increased display rates because the GPU performs the corrections simultaneously at all or many of the grid nodes rather than one at a time as the CPU would.

The α obscurations are pre-computed using the distance Δx between adjacent planes along the x-axis. The adjusted $\hat{\alpha}$ expressed in terms of $\Delta\hat{x}$ is given by

$$\hat{\alpha} = 1 - \exp(-ks\Delta\hat{x}) \tag{4.2}$$

where $\Delta\hat{x}$ is the distance between planes along the line of site. Equations (4.1) and (4.2) may be used to solve for $\hat{\alpha}$ in terms of α to obtain

$$\hat{\alpha} = 1 - (1-\alpha)^{\Delta\hat{x}/\Delta x} \tag{4.3}$$

after noting that

$$1 - \hat{\alpha} = \exp(-ks\Delta\hat{x}) = \exp(-ks\Delta x)^{\Delta\hat{x}/\Delta x} = (1-\alpha)^{\Delta\hat{x}/\Delta x}$$

The computation of equation (4.3) is expensive because the exponential is computed at each grid node for every time step. In addition, numerical cancellation may occur for small α leading to loss of significant digits. Both problems may be solved by expanding equation (4.3) in a Taylor series and keeping only the first few terms:

$$\hat{\alpha} \approx \alpha r - \frac{\alpha^2}{2} r(r-1) + \frac{\alpha^3}{6} r(r-1)(r-2)$$

where $r = \sec(\theta) = \Delta\hat{x}/\Delta x = ||x_p - x_e||/n \cdot (x_p - x_e)$, n is the unit vector normal to the current plane being drawn, θ is the angle between the view direction and n, x_e is the observers position and x_p is the vertex being drawn (along the view direction). These terms are illustrated in Figure 4.5.

When planes are skipped, equation (4.3) may be simplified. In particular, when every 2nd plane is skipped, $\Delta\hat{x}/\Delta x = 2$, so that equation (4.3) simplifies to

$$\hat{\alpha} = 1 - (1-\alpha)^2 = 2\alpha - \alpha^2$$

The video hardware uses α values contained in the smoke planes to obscure the background much like a camera uses a neutral density filter to darken a scene. Extending the analogy, Smokeview uses one spatial/time varying *numerical* neutral density filter for each plane of smoke data. On a node by node basis then, each smoke plane obscures the current image stored in the OpenGL back buffer by the amount $(1-\alpha)$ to form a new back buffer image. Figure 4.6 illustrates this process showing several snapshots of a fire plume. The final image in the lower right is the most realistic. A simplistic description of one step of this process is given by

$$\text{new buffer image} = (1-\alpha) \times \text{old buffer image}$$

This process is repeated for each smoke plane. Figure 4.7 illustrates this process showing smoke and fire in a townhouse kitchen fire.

The visualization is performed by displaying a series of partially transparent planes. For illustration, these planes are made more conspicuous (in Figure 4.7a) by skipping smoke planes (displaying every third plane) and orienting them along the 'x' axis. Figure 4.7b shows the visualization as it normally appears with all slice planes shown and oriented along a plane most perpendicular to the view direction.

4.2.3 Orienting smoke planes

Smoke opacity data computed as described in previous sections is stored in a 3D array. This array corresponds to the solution domain as set up in an FDS input file (or some other model). Smoke planes are drawn in Smokeview through this data. The orientation is chosen to be most perpendicular to the viewers line of sight. A plane orientation exactly perpendicular to the view direction could be drawn if one is willing to pay the added CPU cost of interpolating opacity values between grid nodes.

Figure 4.8 illustrates this process showing three view directions and the corresponding smoke plane orientations that would be used. Off-axis viewing is minimized by selecting the view planes orientation that minimizes the angle between the planes normal direction and the view direction. This angle, θ, is illustrated in Figure 4.9, and is given by

$$\cos(\theta) = \frac{n \cdot v_e}{||n||\ ||v_e||}$$

where n is normal vector for the candidate smoke plane, and v_e is the view direction vector. In OpenGL, the view direction vector, v_e, is computed by simply obtaining the modelview matrix, M and multiplying it by the vector, $(0,0,1)^T$ or equivalently the third row of M.

4.2.4 Compressing Smoke Data

The opacity parameters are computed at each node for all time steps. As noted earlier, the space required to store these values can easily exceed the 2GB file size limit found on 32 bit operating systems. Techniques are required to reduce storage requirements.

Compression for this application occurs in two steps. First, a four to one compression level is achieved by using Beer's law to convert soot density, a four byte floating point quantity, to opacity, a one byte quantity. Video cards presently use only one byte to represent opacity. Next, the sequence of opacity values are compressed using run-length encoding. Run length encoding works as follows.

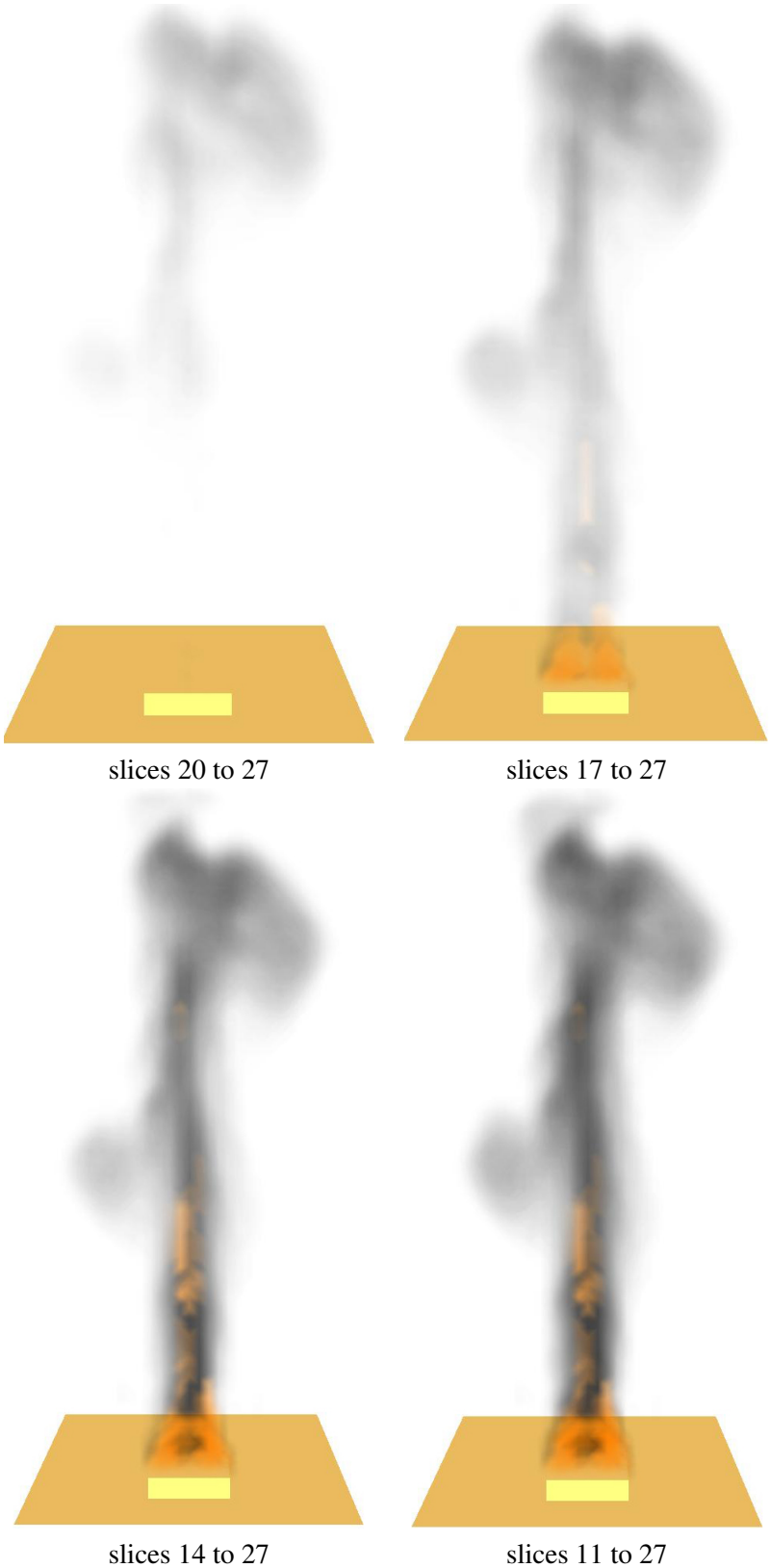
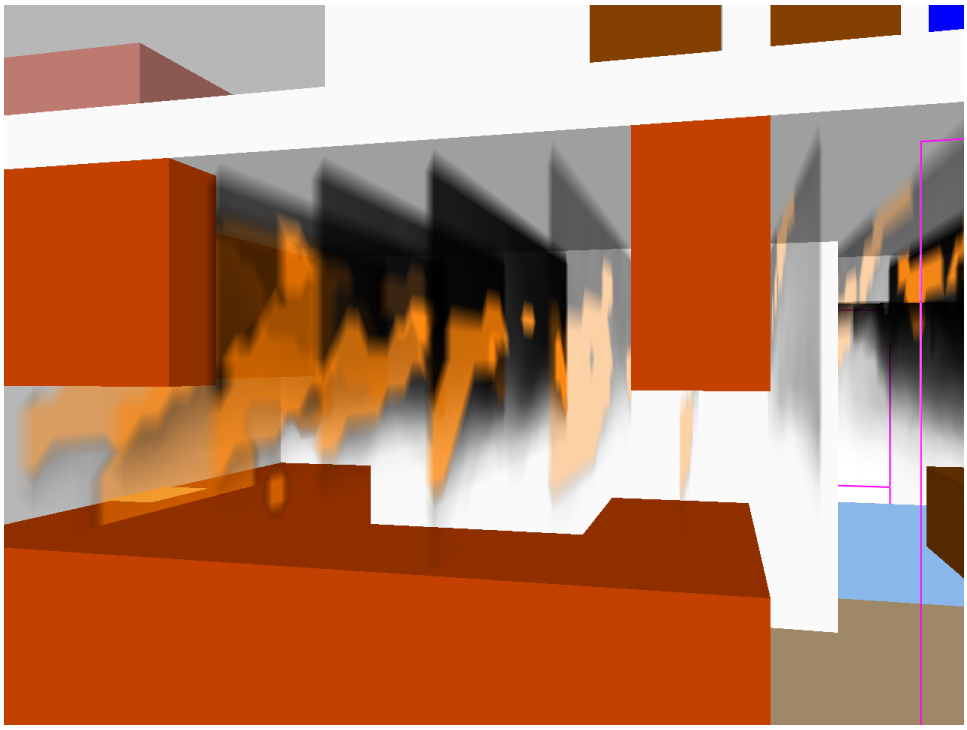
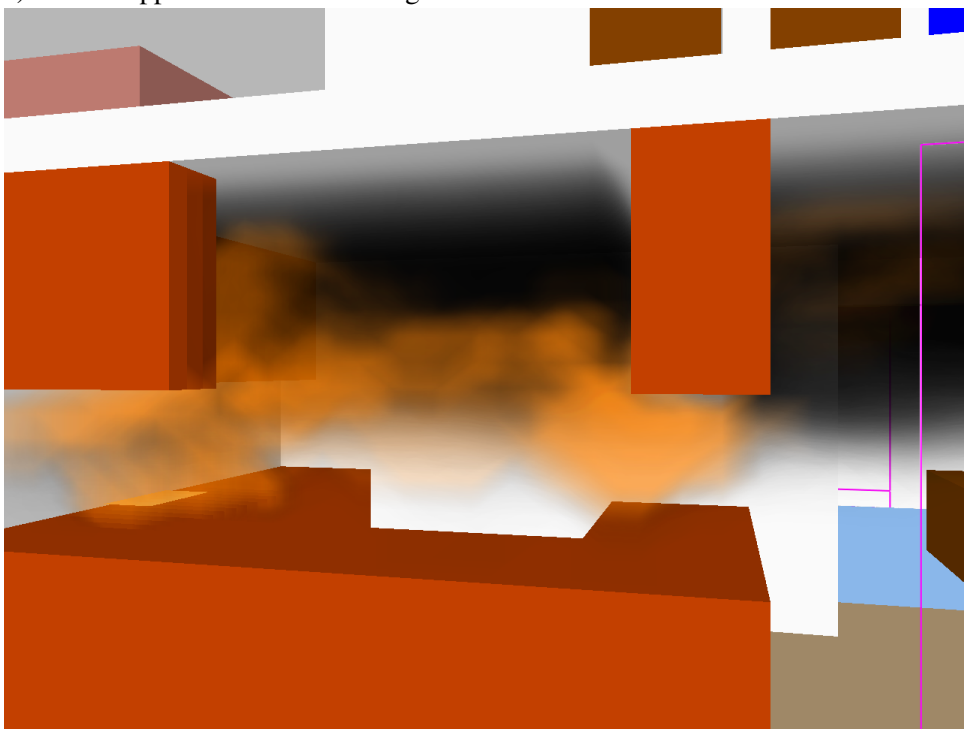


Figure 4.6: Smoke plume visualized using several vertical parallel partially transparent planes. The transparency is computing using soot density which is in turn computed using conservation equations for mass, momentum and energy.

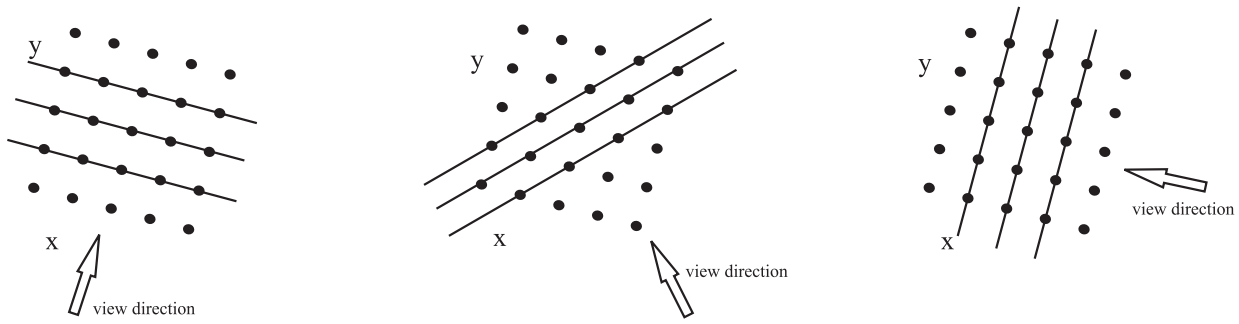


a) slices skipped and oriented along ‘x’ directions



b) all slices shown and oriented towards viewer

Figure 4.7: Realistic visualization of a townhouse kitchen fire simulated using FDS5. Planes in the top image are drawn to be conspicuous by skipping 2 out of every 3 planes and by aligning planes along the x axis. All planes in the bottom image are displayed (none are skipped) and they are aligned to be closest to perpendicular of all possible plane orientations.



a) smoke planes parallel to y axis b) smoke planes parallel to $y = x$ axis) c) smoke planes parallel to x axis

Figure 4.8: View of smoke planes from above. Smoke Planes are oriented so that they are *most perpendicular* to the line of sight

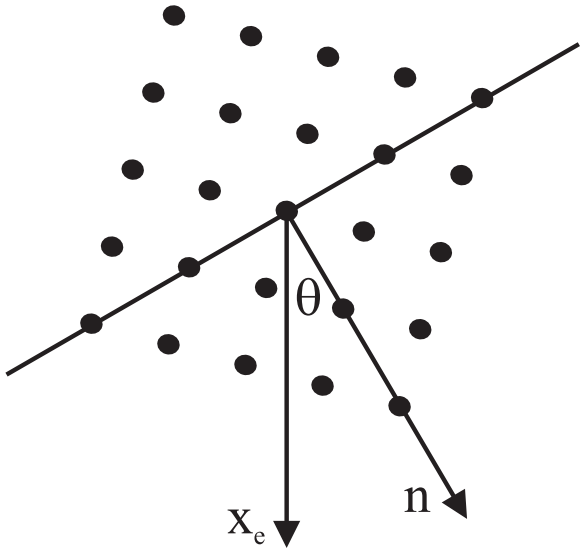


Figure 4.9: Diagram illustrating the angle between the line of sight and smoke plane normal vector. View planes are chosen to minimize this angle.

1. Represent four or more consecutive identical characters as #*nc* where # is a special character denoting the beginning of a repeated sequence, *n* is the number of repeats and *c* is the character repeated. *n* can be up to 254 (255 is used to represent the *special* character).

2. Represent characters not repeated four or more times as is.

So, the character string `aaaaaabbbbcc` would be encoded `#6b#4bcc`.

Run length encoding provides a reasonably good compression ratio, is simple to implement and more importantly can be decompressed quickly. This last property is important for any compression scheme chosen because it is a rate limiting step in the process that Smokeview uses to display smoke data. The CPU time required to compute the smoke flow can easily exceed one minute of CPU time per outputted time step, so extra time used to produce a more compact file is affordable. However, each data frame is decompressed *on the fly* so a compression format that can be rapidly decompressed is critical.

Smokezip is a software program developed as a companion to Smokeview and FDS to compress FDS data files. A second compression scheme is used by Smokezip to compress FDS files even more compactly. Smokezip uses the ZLIB compression library available at (`http://zlib.net/`).

Chapter 5

Future Work

Smokeview is a software tool used to gain insight into results generated by fire models such as the Fire Dynamics Simulator or CFAST. Two general areas of research need to be addressed in order to improve this tool. First, scenarios with a large number of grid cells need to be visualized more effectively and efficiently and second, some visualization algorithms need to be improved and others need to be added in order to more effectively interpret fire modeling data. Some areas of research to accomplish these goals are described in more detail in the following sections.

5.1 Visualize Cases more Realistically

Realism is a metric used when using Smokeview to gauge qualitatively the accuracy of the fire and smoke display. Realism itself is not the primary goal, however. Realism is a side effect of the application of more physics. The primary goal is to gain a better understanding of the data. This requires a more accurate representation of the underlying data which in turn requires the use of more physics.

- Investigate techniques for visualizing fire more realistically by using information about the fire such as its temperature and composition (soot and various gas species) to color it more realistically. Flame temperatures need to be modeled directly by FDS in order in order to use the black body temperature curve to obtain flame color. Further, the resolution of the computation needs to be consistent with the desired resolution of the flame image.

- Investigate algorithms for modeling the interaction of light and smoke. The *transport* of light into and out of the smoke/fire (*i.e. scattering*) is another important factor that effects its appearance. Light sources could consist of either man made lights or from the fire itself. Work also needs to be done to generalize the wavelength of light used to visualize the scene, *i.e.* implementing an algorithm in Smokeview to simulate a thermal imager or infrared viewer.

5.2 Fire Computations in Smokeview

The original conception of FDS and Smokeview was that FDS would perform fire computations and Smokeview would visualize them. This distinction became blurred with the addition in Smokeview of the 3D smoke visualization algorithms. Further blurring the distinction, pre-visualization steps are performed in FDS, to convert soot density to a smoke opacity.

Additional fire computations are planned for Smokeview. The user will gain insight into the fire phenomena much more quickly by being able to manipulate and solve their problem in real time. Two examples are detailed below.

- Investigate methods for modeling the motion of fire brands using either wind fields computed by FDS or wind fields defined in Smokeview. This allows one to define the initial position and distribution of fire brands and to note where they land.

- Implement a level set method proposed by Rehm and McDermott [15] for tracking a fire line. The method would take into account terrain (*i.e.* non-level ground) defined by the user.

5.3 Visualize Larger Cases

Techniques need to be investigated for visualizing larger cases more effectively. Updating Smokeview to allow 64 bit memory addressing is just one option. This is a brute force method. Techniques are also required for honing or zeroing in on data of interest. The fire line used to visualize WUI (wildland urban interface) simulations is a good example of this. A fire line in the context of Smokeview is simply a visual display of temperature only visible where the fire is located. Additional techniques for visualizing large cases that need to be investigated include:

- Investigate methods for making it easier to probe or mine data, *i.e.* to retrieve data of particular interest. Now FDS data is stored sequentially. For large data sets one needs to retrieve data efficiently at a particular time and region without reading the entire data set. User defined spatial and temporal data subsets need to be loaded efficiently in order to shorten the time required to visualize cases.

- Investigate standardized methods for storing data more efficiently and more effectively. Can we do it better and be practical? The idea would be not to change how FDS outputs data but to consider whether a separate *filter* program that would convert FDS generated data into a different format that would enable other tools besides Smokeview to be able to visualize data, for example the visualization tool kit VTK [16].

- Investigate parallelization methods for visualizing data. Smokeview presently has a limited ability to execute code in parallel using a technique known as multi-threading. For example, smooth blockages if present, are smoothed in parallel using the pthreads library. Smokeview in this respect is multi-threaded. Techniques will be investigated for performing the drawing or visualization in parallel. This may be necessary as cases get larger and larger. One technique for parallelizing the visualization is to use several video cards each one drawing a portion of the scene.

5.4 Tools and Techniques

- Investigate methods for using the video card to perform scientific computations. The computational power of the video card is already being exploited by Smokeview to perform simple smoke computations. It will need to be exploited even more in order to make the proposed fire coloring and smoke lighting computations practical. Techniques are being investigated for performing the computations needed to implement the more realistic fire and smoke computations discussed earlier, in particular CUDA [17]. Techniques such as these will be required it more complex visualization algorithms are to be effective.

- Investigate alternative methods for implementing a graphical user interfaces (GUI) for Smokeview. Smokeview uses GLUT for implementing a simple user interface. GLUT was not intended for developing user interfaces as complex as what Smokeview requires. Alternatives for implementing user interfaces will be investigated.

Bibliography

[1] National Institute of Standards and Technology, Gaithersburg, Maryland, USA, and VTT Technical Research Centre of Finland, Espoo, Finland. *Fire Dynamics Simulator, Technical Reference Guide*, 5th edition, October 2007. NIST Special Publication 1018-5 (Four volume set).

[2] K.B. McGrattan, S. Hostikka, and J.E. Floyd. Fire Dynamics Simulator (Version 5), User's Guide. NIST Special Publication 1019-5, National Institute of Standards and Technology, Gaithersburg, Maryland, October 2007.

[3] OpenGL Architecture Review Board, Dave Shreiner, Mason Woo, Jackie Neider, and Tom Davis. *OpenGL Programming Guide - The Official Guide to Learning OpenGL, Version 2.1*. Addison-Wesley, Stoughton, Massachussets, 6 edition, 2007.

[4] Mark J. Kilgard. *OpenGL Programming for the X Window System*. Addison-Wesley Developers Press, Reading, Massachussets, 1996.

[5] Brian W. Kernighan, Dennis Ritchie, and Dennis M. Ritchie. *C Programming Language (2nd Edition)*. Prentice Hall PTR, March 1988.

[6] T. M. Ellis, Ivor R. PHilips, and Thomas M. Lahey. *Fortran 90 Programming*. Addison-Wesley, 1994.

[7] G.P. Forney. Smokeview (Version 5), A Tool for Visualizing Fire Dynamics Simulation Data, Volume I: User's Guide. NIST Special Publication 1017-1, National Institute of Standards and Technology, Gaithersburg, Maryland, August 2007.

[8] G.P. Forney. Smokeview (Version 5), A Tool for Visualizing Fire Dynamics Simulation Data, Volume III: Verification Guide. NIST Special Publication 1017-3, National Institute of Standards and Technology, Gaithersburg, Maryland, May 2009.

[9] K.B. McGrattan, H.R. Baum, R.G. Rehm, A. Hamins, and G.P. Forney. Fire Dynamics Simulator, Technical Reference Guide. NISTIR 6467, National Institute of Standards and Technology, Gaithersburg, Maryland, January 2000.

[10] Paul Martz. *OpenGL Distilled*. Addison-Wesley, Upper Saddle River, NJ, 1st edition, 2006.

[11] Richard S. Wright Jr., Benjamin Lipchak, and Nicholas Haemel. *OpenGL Super Bible - Comprehensive Tutorial and Reference*. Addison-Wesley, Upper Saddle River, NJ, 4 edition, 2007.

[12] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 163–169. ACM Press, 1987.

[13] Ronald Fedkiw, Jos Stam, and Henrik Wann Jensen. Visual simulation of smoke. In Eugene Fiume, editor, *SIGGRAPH 2001, Computer Graphics Proceedings*, pages 15–22. ACM Press / ACM SIGGRAPH, 2001.

[14] Robert Siegel and John R. Howell. *Thermal Radiation Heat Transfer*. Taylor & Francis, Inc., New York, NY, 4 edition, 2001.

[15] R.G. Rehm and R. McDermott. Fire-Front Propagation Using the Level Set Method. NISTTN 1611, National Institute of Standards and Technology, Gaithersburg, Maryland, April 2009.

[16] Will Schroeder, Ken Martin, and Bill Lorensen. *The Visualization Toolkit, Third Edition*. Kitware Inc.

[17] What is CUDA, http://www.nvidia.com/object/cuda_what_is.html.

[18] Paul Rademacher. GLUI version 2.1, http://www.cs.unc.edu/ rademach/glui/.

[19] Thomas Boutell. GD version 2.0.7, http://www.boutell.com/gd/, November 2002.

[20] Guy Eric Schalnat, Andreas Dilger, and Glenn Randers-Pehrson. libpng version 1.2.5, http://www.libpng.org/pub/png/, November 2002.

[21] JPEG version 6b, http://www.ijg.org/.

[22] Jean loup Gailly and Mark Adler. zlib version 1.1.4, http://www.gzip.org/zlib/, November 2002.

Appendix A

Smokeview Program Structure

This chapter gives an overview of the program structure for Smokeview. Smokeview consists of about 85,000 lines of code. Most of it is written in C using standard libraries such as OpenGL [3] for implementing the graphics, GLUT [4] for providing a simple menu based user interface and interacting with the host operating system. Additional graphical user interface (GUI) elements are implemented as dialog boxes using the OpenGL based widget library GLUI [18]. The libraries GD [19], libpng [20], and libjpeg [21] are used for converting Smokeview scenes/results into images files. The library libzip [22] is used for compression. Smokeview uses this compression library when generating PNG files and when reading in 3D smoke files compressed by smokezip.

A small portion of Smokeview is written in Fortran 90 to input data generated by FDS. The use of portable libraries allows Smokeview to run on many platforms including Windows, and various versions of Unix such as IRIX (for the SGI), Linux and OSX (for the Macintosh).

Figure A.1 illustrates how these libraries and Smokeview are organized.

Since Smokeview is a large program and is constantly changing, it is not practical to give a line by line detailed description. The interested user, however, should be able to use this overview as a starting point to dig into those parts of Smokeview that may be of interest.

The program structure of Smokeview is fundamentally different than that of FDS in one important respect. Smokeview is an *event driven* program with a graphical user interface. The term *event driven* in this context means that the user controls the program flow using various means such as pressing a key, clicking or dragging the mouse, selecting a menu item *etc.* In other words, the user initiates events not the program. The Smokeview response then depends on which *event* has occurred.

The program structure may then be described as follows. Smokeview starts out by by performing initializations and setting up the OpenGL environment. This is followed by a call to a GLUT routine that implements the *event loop*. The *event loop* is literally a program loop without an exit which detects when the user invokes the various events, again these are key clicks, mouse clicks, menu selection *etc.*. Once an event is detected, a user *callback* routine is called. Note that user callback routines are Smokeview code and the event loop is GLUT library code. Part of Smokeview's initialization process is to define which C procedures are associated with which callbacks. This is performed in the Smokeview C routine `InitOpenGL` located in the source file `startup.c`.

There is a Smokeview callback routine for every action that is performed. For example, the user callback for loading a 3D smoke file is `Load3DMenu`. This is a menu callback because it is called after the user selects a menu item. All menu callbacks are located in the source file `menu.h`. All other callback routines are located in the source file `callback.c`.

Figure A.2 illustrates the program structure described in the previous paragraphs. Every OpenGL/GLUT program will use most of the callbacks named in this figure. Of course, the actions performed by Smokeview

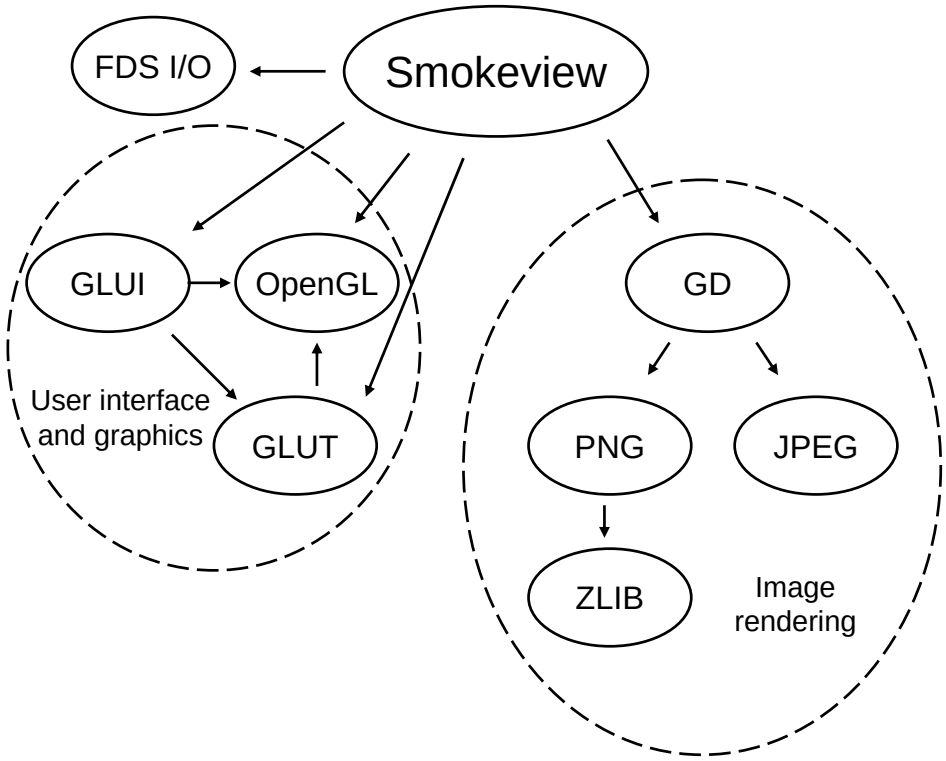


Figure A.1: Smokeview external library usage

will be different.

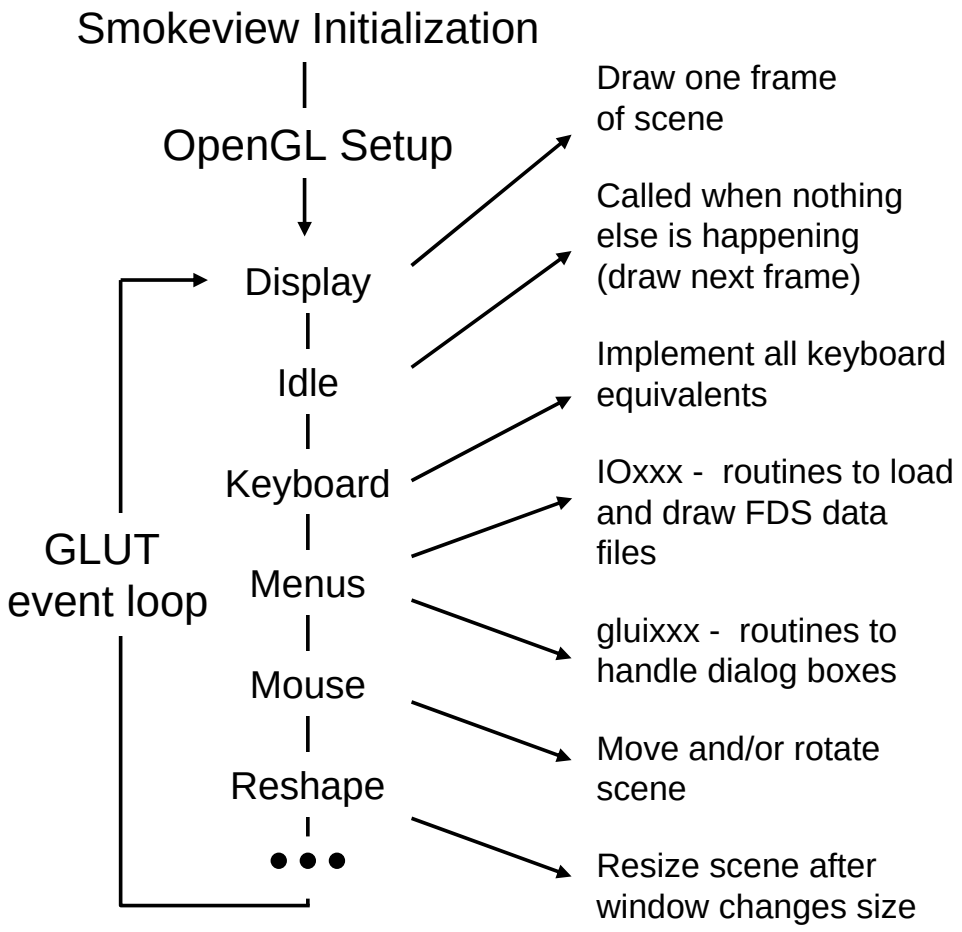


Figure A.2: Smokeview program structure

Appendix B

Interfacing OpenGL with the Host Operating System

OpenGL by design draws the 3D geometry but does not interact with the user or the operating system. Smokeview uses the graphics library utility toolkit (GLUT) for interacting with the user *via* the keyboard and mouse and for interacting with the operating system to swap display buffers, to display fonts, to set maximum frame rates *etc*. Though not as sophisticated as other libraries, GLUT is simple to use and is portable allowing Smokeview to be built on a number of different computer platforms including a PC running Windows or Linux, a Silicon Graphics workstation running IRIX or a Macintosh running OSX.

B.1 Buffers

Smokeview uses several buffers provided by OpenGL for visualization. GLUT is used to manipulate these buffers. Smokeview uses double buffering. Double buffering is the technique where drawing occurs in the **back buffer** while the scene is simultaneously displayed using the **front buffer**. Smokeview uses the GLUT routine `glutSwapBuffers();` to swap the front and back buffers once drawing is complete. Screen flickering occurs the if the display (front) buffer is updated while drawing occurs.

Hidden lines and surfaces are removed from a scene using the **depth buffer**. Each time Smokeview draws an object, its depth (distance from the observer) is compared to the value previously stored in the depth buffer. If the object's depth is less than the value stored in the depth buffer then the object is considered visible and the new depth value is store in the buffer. Otherwise the depth buffer remains unchanged and the object is considered hidden.

B.2 Initialization and Callback Routines

Initializations are performed by GLUT to set up windows and to define display modes. Smokeview defines the display to handle color, a depth buffer and double buffering by passing the OpenGL keywords `GLUT_RGB`, `GLUT_DEPTH` and `GLUT_DOUBLE` to `glutInitDisplayMode` as in

```
glutInitDisplayMode(GLUT_RGB|GLUT_DEPTH|GLUT_DOUBLE);
```

Smokeview creates a window with width `windW` and height `windH` using the GLUT calls

```
glutInitWindowSize(windW, windH);
glutCreateWindow("");
```

and defines callbacks with

```
glutSpecialUpFunc(specialkeyboard_up);
glutKeyboardUpFunc(keyboard_up);
glutKeyboardFunc(keyboard);
glutMouseFunc(mouse);
glutSpecialFunc(specialkeyboard);
glutMotionFunc(motion);
glutReshapeFunc(Reshape);
glutDisplayFunc(Display);
```

A callback is a routine that is called when a particular event occurs. For Smokeview these events would be when a keyboard key is depressed (or when the key is released), when the mouse is clicked or when the mouse is moved. Smokeview determines rotation and translation amounts using the `motion` callback defined with `glutMotionFunc(motion)`.

Appendix C

Rendering Smokeview Images

Smokeview uses the library GD [19] to convert the currently displayed scene into either a JPEG, PNG or GIF image. Smokeview reads in the OpenGL back buffer and then uses the GD routine, `gdImageSetPixel` to store the image data in GD's internal format one pixel at a time. Finally, Smokeview uses a GD routine to convert the image into the desired image format.

A summary of the steps in more detail are:

1. Allocate memory buffers and file pointers

```
RENDERfile = fopen(RENDERfilename, "wb");
pixels = (GLubyte *) malloc(width * height * sizeof(GLubyte) * 3);
```

2. Read pixel data from the back buffer

```
glReadPixels(x, y, width, height, GL_RGB, GL_UNSIGNED_BYTE, OpenGLimage);
```

3. Allocate the gd data structures used to hold image data

```
RENDERimage = gdImageCreateTrueColor(width,height);
```

4. Set pixel data

```
for (i = height-1 ; i>=0; i--) {
  for(j=0;j<width;j++){
    r=*p++; g=*p++; b=*p++;
    rgb = (r<<16)|(g<<8)|b;
    gdImageSetPixel(RENDERimage,j,i,rgb);

  }
}
```

5. Write out data to a JPEG image file

```
  gdImageJpeg(RENDERimage,RENDERfile,-1);
```

6. deallocate memory buffers and free file pointer

```
fclose(RENDERfile);
gdImageDestroy(RENDERimage);
free(OpenGLimage);
```