

University of Massachusetts Dartmouth

THREE DIMENSIONAL COMPUTER GRAPHICS:
TIME EFFICIENT DISPLAY OF
SURFACES OF REVOLUTION

A Thesis in
Computer Science
by
Jürgen Schulze-Döbold

Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

June 1998

I grant the University of Massachusetts Dartmouth the non-exclusive right to use the work for the purpose of making single copies of the work available to the public on a not-for-profit basis if the University's circulating copy is lost or destroyed.

Jürgen Schulze-Döbold

Date: _____

We approve the thesis of Jürgen Schulze-Döbold

Date of signature

Adam O. Hausknecht

Professor of Mathematics and Computer and Information Science
Thesis Advisor

Edmund Staples

Associate Professor of Computer and Information Science
Graduate Committee
Graduate Program Director, Computer and Information Science

Robert Green

Professor of Computer and Information Science
and Electrical and Computer Engineering
Graduate Committee

Boleslaw Mikolajczak

Chairperson, Computer and Information Science

Thomas J. Curry

Dean, College of Engineering

Richard J. Panofsky

Associate Vice Chancellor for Academic Affairs and Graduate Studies

Abstract

THREE DIMENSIONAL COMPUTER GRAPHICS:

TIME EFFICIENT DISPLAY OF SURFACES OF REVOLUTION

by Jürgen Schulze-Döbold

An algorithm for efficient shading of surfaces of revolution is developed. The surfaces can either be smoothly approximated by polygons, or they can be approximated by a number of cylinders. The algorithm is designed for computer systems which allow the direct access of pixels. All the stages in the displaying process have been optimized for speed.

An implementation of the algorithm is presented. It is written in the C programming language, and it is designed for Microsoft Windows computers. The core parts of the program are portable to other operating systems. In order to examine the efficiency of the new algorithm, programs for OpenGL and SPHIGS have been developed.

Table of Contents

1 INTRODUCTION.....	1
1.1 SHADING ALGORITHMS	3
<i>1.1.1 Shading in General</i>	<i>3</i>
<i>1.1.2 Flat Shading</i>	<i>5</i>
<i>1.1.3 Gouraud Shading.....</i>	<i>6</i>
<i>1.1.4 Phong Shading.....</i>	<i>7</i>
1.2 GRAPHICS ENVIRONMENTS	8
<i>1.2.1 Microsoft Windows</i>	<i>8</i>
<i>1.2.2 OpenGL.....</i>	<i>9</i>
<i>1.2.3 SPHIGS</i>	<i>13</i>
2 THE OPTIMIZED SHADING ALGORITHMS	15
2.1 DEFINITIONS	15
<i>2.1.1 Surface.....</i>	<i>15</i>
<i>2.1.2 Light Source and Viewpoint</i>	<i>18</i>
2.2 THE TECHNIQUES AND THEIR OPTIMIZATIONS.....	19
<i>2.2.1 Shading.....</i>	<i>19</i>
<i>2.2.2 Rotation Calculations</i>	<i>22</i>
<i>2.2.3 Normal Calculations.....</i>	<i>26</i>
<i>2.2.3.1 Points on the Surface.....</i>	<i>27</i>
<i>2.2.3.2 Points on Disks</i>	<i>29</i>
<i>2.2.4 Projection to 2D Screen</i>	<i>30</i>
<i>2.2.5 Back Face Culling.....</i>	<i>31</i>
<i>2.2.6 Painter's Algorithm.....</i>	<i>31</i>
<i>2.2.7 Creation of Quadrilaterals.....</i>	<i>33</i>
<i>2.2.8 Color Computation</i>	<i>34</i>
<i>2.2.9 Fixed-Point Arithmetic.....</i>	<i>35</i>
<i>2.2.10 Look-up Tables</i>	<i>36</i>
<i>2.2.11 Gradual Display of Rotation Process</i>	<i>37</i>
3 THE IMPLEMENTATIONS.....	39
3.1 THE OPTIMIZED ALGORITHM	39
<i>3.1.1 Create a New Vertex List.....</i>	<i>40</i>
<i>3.1.2 Create a New Polygon List.....</i>	<i>41</i>
<i>3.1.3 Draw the Surface</i>	<i>42</i>
<i>3.1.4 Rotate a Point</i>	<i>44</i>
<i>3.1.5 Project and Transform a Point to Screen Coordinates.....</i>	<i>45</i>
<i>3.1.6 Get a Vertex Color.....</i>	<i>46</i>
<i>3.1.7 Shade a Quadrilateral.....</i>	<i>47</i>

3.1.8 Calculate the Boundary Line Shades.....	47
3.1.9 Draw a Polygon	49
3.2 THE MICROSOFT WINDOWS VERSION	51
3.2.1 The Color Palette.....	51
3.2.2 The Graphics Engine	52
3.2.3 The Dialog Elements.....	53
3.2.3.1 The Function Selection.....	54
3.2.3.2 The Ambience Color Selection.....	54
3.2.3.3 The Rotation Parameters	54
3.2.3.4 The Approximation Modes.....	55
3.2.3.5 The Draw Modes	55
3.2.3.6 Animation	56
3.2.3.7 Display Speed Adjustments	56
3.3 THE OPENGL VERSION.....	57
3.4 THE SPHIGS VERSION	61
3.5 PORTING TO OTHER SYSTEMS	63
4 PERFORMANCE ANALYSIS	64
4.1 COMPARISON OF THE IMPLEMENTATIONS.....	64
4.2 PROFILER RESULTS	68
4.2.1 MS Windows Version	68
4.2.2 OpenGL Version	70
4.3 COMPLEXITY	72
4.4 FURTHER OPTIMIZATION	75
4.5 POSSIBLE FUTURE ENHANCEMENTS	76
5 REFERENCES.....	78
6 APPENDIX.....	79
6.1 LIST OF ALGORITHMS	79
6.2 COMPLETE PROFILER OUTPUT	80
6.2.1 Microsoft Windows Version.....	80
6.2.2 OpenGL Version	82
6.3 CONTENTS OF COMPUTER DISK	84
6.4 SOURCE CODE	86
6.4.1 Fast SOR Header File	86
6.4.2 Fast SOR Main Module.....	90
6.4.3 The OpenGL Version.....	106
6.4.4 The SPHIGS Version.....	122
7 CURRICULUM VITAE	142

List of Figures

Figure 1-1: Diffuse Reflection.....	5
Figure 1-2: Gouraud Shading	7
Figure 1-3: Block Diagram of OpenGL	11
Figure 2-1: Sample cylinder with 5 sectors	16
Figure 2-2: Sample Surface $f(x)=x$, 6 cylinders, 5 sectors.....	17
Figure 2-3: End Disks of a SOR	18
Figure 2-4: The MIN and MAX arrays.....	20
Figure 2-5: The Tangent Component	27
Figure 2-6: The Disk Components	29
Figure 2-7: The Vertices in a SOR	33
Figure 3-1: The Windows Dialog Box.....	53
Figure 3-2: The OpenGL implementation	58
Figure 3-3: The SPHIGS implementation.....	61

List of Tables

Table 2-1: Complexities of Rotations of n Points.....	25
Table 2-2: Comparison of Computation Methods.....	26
Table 4-1: Algorithm Speed Comparison	66
Table 4-2: Fast SOR Profiler Output.....	69
Table 4-3: Profiler Output of OpenGL version	71
Table 4-4: Complexities of the Fast SOR Routines	73
Table 4-5: Complexities by Type of Operation.....	74
Table 6-1: Files on Computer Disk	85

1 Introduction

At present, there are several general methods for rendering three dimensional objects on the two dimensional computer screen. These well known algorithms certainly work for surfaces of revolution as well. This thesis investigates the simplifications in rendering that emerge out of the special characteristics of surfaces of revolution. A specialized algorithm is developed, which is then compared to the general rendering algorithms of present 3D graphics environments. In developing the algorithm, an important goal was to create a Windows (short for "Microsoft Windows") version in the language of C++, which could be easily ported to the Macintosh OS. This is because the idea for the development of this algorithm came from the need for an efficient algorithm to rotate and modify an arbitrary surface of revolution in real time, that could be incorporated in the mathematics teaching software package "TEMATH" (available for Macintosh computers). This software is used in the Calculus and Physics class, for example, where surfaces of revolution are taught.

The algorithm is restricted to surfaces of revolution for a number of reasons. First of all, there is not much room anymore for an improvement of the general 3D rendering algorithms. A large amount of research was done on these, and there are countless implementations available on the market. But several simplifications apply to the

generation process of surfaces of revolution, which allow a significant improvement in rendering speed, compared to using a general algorithm. For example, its specific geometry allows the surface to be approximated by quadrilaterals, instead of triangles, which most 3D renderers would use. The developed algorithm, along with the characteristics of surfaces of revolution, makes sure that no intersections of quadrilaterals occur. This is a time consuming process which general rendering software has to deal with. Furthermore, the quadrilaterals' normal vectors can easily be calculated. They are needed for shading and backface culling. Finally, it is sufficient to use the painter's algorithm to do hidden surface removal, which is the fastest way to accomplish this.

For this thesis three different and independent programs were developed, running on three different computer systems. The main version, containing the optimized algorithm, is designed for Windows' systems, and it accesses the graphics output as directly as possible. In order to be able to compare the results of this algorithm, one program was written for SUN workstations using the 3D package SPHIGS, and another one was written in OpenGL, and it compiles on both Windows and Macintosh systems.

1.1 *Shading Algorithms*

In order to understand the features of the specialized algorithm, the general algorithms which are used when rendering 3D graphics have to be discussed. The three most common ones are described below. The flat shading algorithm is the fastest and easiest to describe. Gouraud shading needs much more computing power, but results in a great improvement in image quality. Phong shading is the most sophisticated algorithm of the three, and it produces the most realistic images. All of these algorithms require the subdivision of the surface to be displayed into triangles. All the algorithms process one triangle after another in a rendering loop.

1.1.1 Shading in General

According to (Foley et al. 1996, p. 722 ff.) the three most important components of illumination are ambient light, diffuse reflection, and specular reflection.

The ambient light component contributes to the base color value of the entire scene in which the surface of revolution is found. It is not dependent on the view angle or the location of light sources, and it is also present with surfaces which are facing away from the light source. Thus, this light component can be handled as a constant value which

is added to the other illumination components. The illumination equation for ambient light only is:

$$I = I_a \cdot k_a .$$

I is the resulting intensity, I_a is the intensity of the ambient light, and k_a is the amount of ambient light reflected from an object's surface (it depends on the surface).

Diffuse reflection depends on the angle between the surface normal and the direction of the light source. But it is independent of the viewpoint, since it describes the fraction of light which is reflected into all directions equally. The diffuse illumination equation is:

$$I = I_p \cdot k_d \cdot \cos(\alpha)$$

I_p is the intensity of the light source, k_d is the material dependent reflection coefficient for diffuse light, and α is the angle between the direction of the light source and the surface normal at a specific point on the surface. See Figure 1-1. This equation is known as *Lambert's Law*.

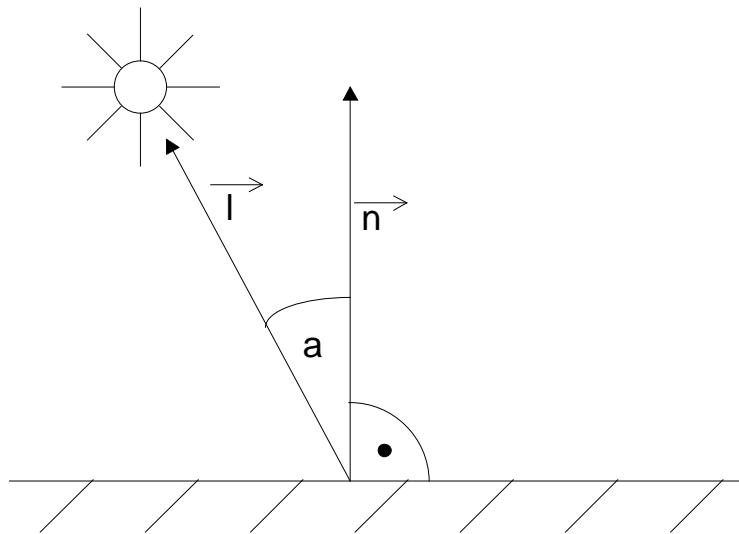


Figure 1-1: Diffuse Reflection

Finally, specular reflection depends on the viewpoint and the surface normal, it takes care of the part of light which is reflected by the surface as if it was a mirror. There are different illumination models for specular reflection, but for efficiency we will neglect this term in our illumination model.

1.1.2 Flat Shading

The triangle which is being rendered is colored with a single intensity. This intensity depends on the positions of the light source and the surface, which determine the angle a in the diffuse illumination equation ($I = I_p \cdot k_d \cdot \cos(\alpha)$). For this method of shading to be mathematically correct, both the light source and the viewer must be in a

fixed direction at infinity, so that their direction angles are constant. In this situation, a polygon's color only depends on the normal vector constructed for the polygon.

1.1.3 Gouraud Shading

This kind of shading considers not only the polygon face which has to be shaded, but also the neighboring faces. The key idea is to find the "real" angles from the polygon vertex normals to the light source. Usually the objects' polygons (which have to be planar) are just an approximation of a smooth surface, so the 'real' normals for the vertices can be found by averaging the face normals of the adjacent polygon faces. However, in case of a surface of revolution this is not necessary, since each vertex's angle can be computed directly. (see chapter 2.2.3.1).

After the calculation of all vertex normals, they are used to calculate the intensities at each vertex. Then the Gouraud shading algorithm interpolates the inner polygon colors linearly, usually using an algorithm which processes the polygon scan line by scan line. Figure 1-2 helps visualizing this process.

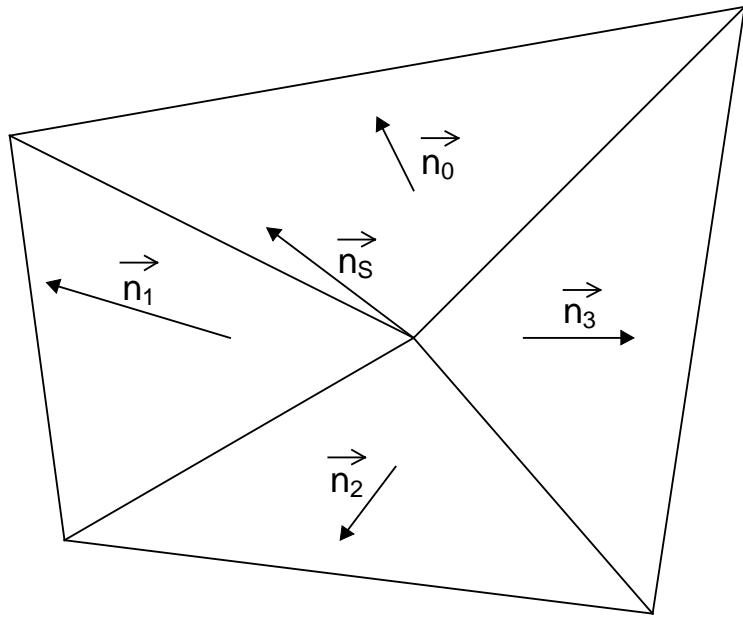


Figure 1-2: Gouraud Shading

1.1.4 Phong Shading

Phong shading is similar to Gouraud shading, except that it interpolates the normal vector instead of the light intensity at each pixel. This means that three values (one for each dimension) have to be interpolated, instead of just a single value. In order to obtain a correct intensity value, the interpolated normals must be normalized. This again means more computations. So Phong shading turns out to use significantly more computing power than Gouraud shading.

1.2 *Graphics Environments*

In order to be able to compare the specialized algorithm to the standard algorithms of rendering 3D graphics in both speed and programming effort, three different programs have been developed. The new specialized algorithm only runs on Windows systems. There are non-specialized versions in OpenGL for both PCs and Macintoshes, and finally, to examine one more 3D environment, there is a program for SUN workstations which uses the SPHIGS package.

These implementations illustrate the usefulness of graphics toolkits such as OpenGL and SPHIGS, in comparison to the Windows version, which does all of the 3D calculations on its own. This version directly accesses the pixel image which is copied to the output window.

1.2.1 Microsoft Windows

The Windows program uses direct pixel addressing. This means that it creates a duplicate image of the output window in memory. Thus pixel access is very fast, the performance difference to using the appropriate API functions is striking. The sample program uses 8 bit per pixel, the color values are chosen out of a color palette of 256 entries.

Mouse events are processed by the Windows system, which in turn sends messages to the application. These are then processed by the application's event loop.

The Windows implementation is the only one which allows the usage of dialog windows. Both OpenGL and SPHIGS do not use dialog window extensions. This gives the user easy access to all adjustable program parameters. Internally, dialog windows have their own window messages. There is a different message for each of the window elements that can be accessed by the user. And there are different callback routines for each dialog window, so it is fairly easy to distinguish the user events.

1.2.2 OpenGL

OpenGL (short for "Open Graphics Library") is a graphics toolkit which was developed by Silicon Graphics. It was derived from the existing graphics interface called IRIS GL. OpenGL was designed to be used on most of today's computer platforms, and there are implementations for SGI, SUN, Microsoft, Apple and DEC. There is also a freeware library available called Mesa. It is even built into the newer releases of Windows NT. OpenGL supports hardware graphics accelerators, but they are not necessary for its use. This platform independent 3D library turns out to be very powerful, and many commercial 3D applications use OpenGL. It is very successful in engineering and science applications, where it is mainly used on UNIX Systems. On the other hand, it has not been

widely accepted by computer game developers. In the Windows world, its biggest rival is Direct 3D by Microsoft.

OpenGL offers all kinds of help for the graphics programmer. To get an impression of the power of this graphics package (which is not so different from other 3D packages though), here are some of the most important features:

- geometric and raster primitives
- viewing and modeling transformations
- lighting and shading
- hidden surface removal (z-buffering)
- double buffering
- alpha blending (transparency)
- texture mapping
- atmospheric effects (fog, smoke, haze)
- selection of objects by screen position

The major disadvantage of toolkits like OpenGL is that the complicated graphics functions have to work for any kind of object, and thus the graphics output becomes very slow when a large number of polygons are used to approximate a smooth surface. Special acceleration graphics hardware keeps this negative side effect down, but at present there

are still applications which require more graphics power than most systems can provide, especially when cost is an issue.

Figure 1-3 shows the numerous steps involved in rendering an object.

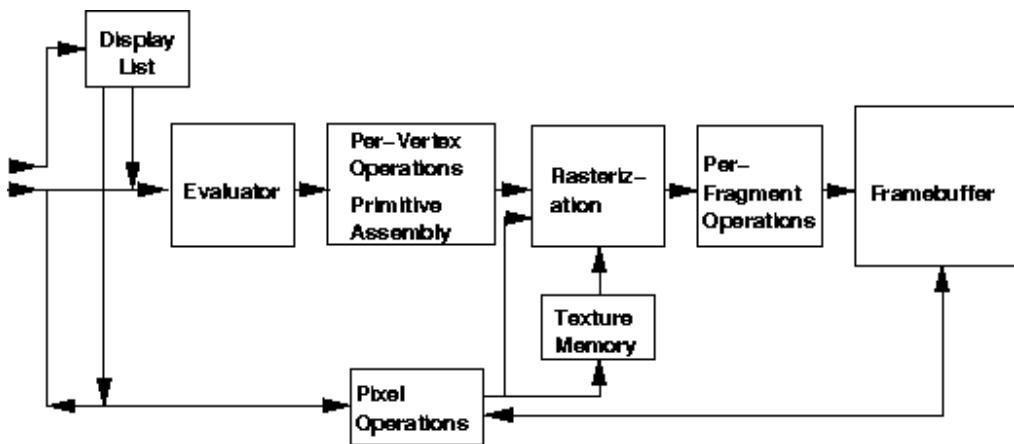


Figure 1-3: Block Diagram of OpenGL

In particular this figure shows that even the simple display of a 2D pixmap involves processing by all the stages. This means that it takes much more time to display pixels in OpenGL than in pixel based graphics toolkits, where pixels can be copied to the output screen buffer directly. However, if you want to do something like this you would not use a 3D graphics package, since a 2D equivalent would be sufficient. But on the other hand, this means that even if you try to speedup your OpenGL based application by writing specialized 3D code yourself and displaying the result on the output screen, you will by far not match the speed of direct graphics buffer addressing.

OpenGL is designed to be platform independent. But the original specification does not contain any functionality for the handling of the user interface. There are different toolkits available, the most widely known are AUX, GLUT, and TK. Their biggest difference is that they are all not implemented for as many systems as OpenGL itself. This thesis uses the GLUT toolkit for getting the new algorithm's comparison data, because it runs on both Macintosh and Windows systems. GLUT, designed by Mark Kilgard, is more powerful than AUX. In particular it provides fonts and a simple pop-up menu facility. The callback functions, called by the respective windows system, bear names like glutInitDisplayMode (initialize display window), or glutReshapeFunc (update window contents).

Typically an OpenGL program starts with creating a window, and a corresponding framebuffer, in which the screen picture will be drawn. Then the programmer can display basic geometric objects like points, lines, or polygons, and he or she can assign colors and materials to these. More complex objects can be put into a display list, which can be used as a single object later on. OpenGL does all the display processing by itself, it uses the objects' attributes and the 3D worlds' attributes (like light sources or orientation) to produce a 2D image in the framebuffer. Interestingly OpenGL does not provide any commands for direct manipulation of the framebuffer. This is to keep the entire image generation process under the control of OpenGL, so that the resulting programs are portable. Another goal in the development of OpenGL was to enable hardware

manufacturers to design graphics cards which support many of OpenGL's features on the circuit level.

1.2.3 SPHIGS

The SPHIGS toolkit, developed at Brown University, is a subset of the rather complex PHIGS package, which was developed at the MIT. SPHIGS (for "Simple Programmer's Hierarchical Interactive Graphics System") was designed for learning purposes in the development of [Foley et al. 1996]. Unfortunately, SPHIGS has not been updated in a while, and so it prevents the user from using features today's fast computers are capable of (e. g. alpha blending or texture mapping). But nevertheless it is still easy to learn 3D graphics using this environment, and one of its greatest advantages is that it was written using SRGP, which is a 2D package, introduced in the same text as SPHIGS.

In comparison to OpenGL, SPHIGS is not available for as many different computer systems. There are versions for Unix, Apple Macintosh Computers, and MS-DOS machines. This thesis uses the Unix implementation, compiled with the 'gcc' compiler. The biggest difference between SPHIGS and OpenGL is the representation of objects. SPHIGS maintains a database of graphic structures, in which each structure contains all the necessary rendering information for each object. The advantages are that you only need to modify a structure, in order to change its screen representation, and that this parts

database is memory efficient. If an application does not allow the usage of the internal structure, you can still design your own. But this would mean extra memory space cost. To the programmers, OpenGL provides a large amount of freedom for arranging the object data. But on the other hand this means that before writing an OpenGL application it is necessary to develop a data structure for the graphics data, and there is no memory space advantage.

2 The Optimized Shading Algorithms

This chapter is going to present the ideas of the new specialized algorithm. Only the mathematical background is given here, which will enable the reader to implement the algorithm on an arbitrary computer system.

Implementation specific issues are to be discussed in the next chapter. Pseudocode will only be used there.

2.1 Definitions

The used 3D model assumes a cartesian coordinate system. The output image is to be displayed on a pixel based device with individually addressable pixels. All variables represent real numbers, unless otherwise noted. To increase readability, the rest of this document may use the abbreviation "SOR" for "surface of revolution".

2.1.1 Surface

In the mathematical sense, we will assume that the surface which is to be displayed is perfectly smooth. But unless a raytracing algorithm is used, there is no way of avoiding its partition into a finite number of patches, just fine enough to create the impression of a

smooth surface. Most commercial graphics utilities use triangles, while this algorithm is based on planar quadrilaterals. This is because their symmetry best suits to the symmetry conditions of a SOR.

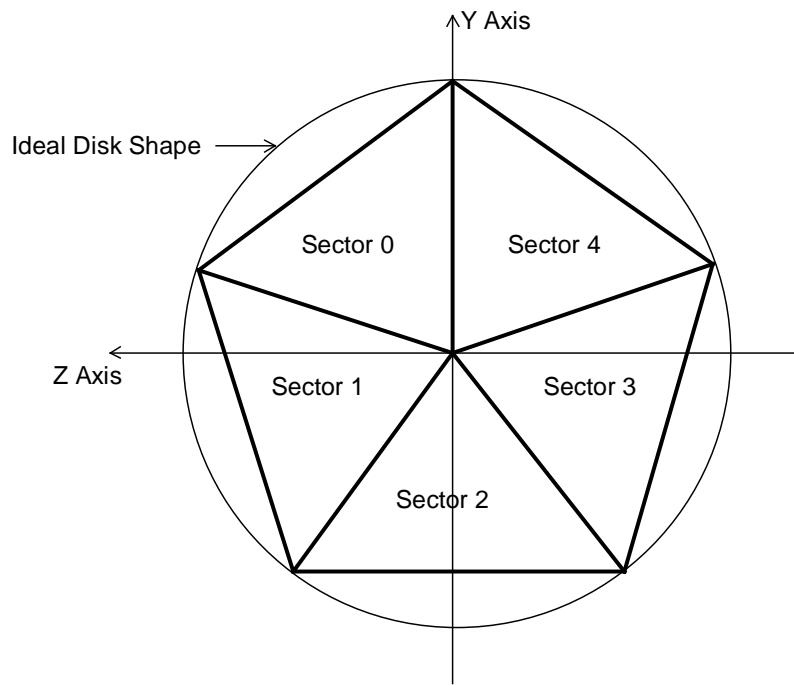


Figure 2-1: Sample cylinder with 5 sectors

There are two directions in which subdivisions of the surface occur. One is along the x -axis, the resulting objects are "cylinders". The cylinders are made up of a number of equally sized "sectors". Figure 2-1 shows a sample cylinder, and Figure 2-2 shows a sample SOR.

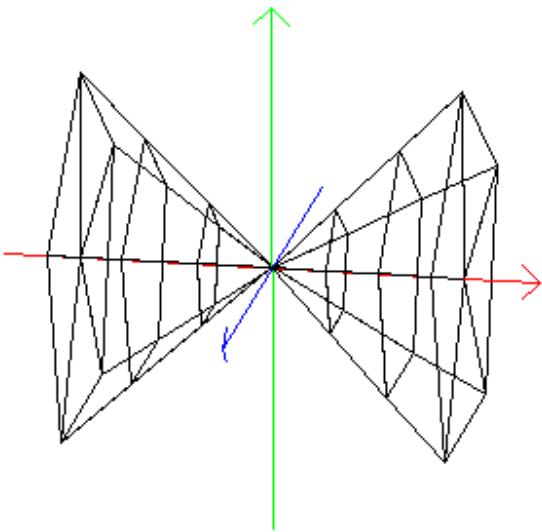


Figure 2-2: Sample Surface $f(x)=x$, 6 cylinders, 5 sectors

The surface of a SOR does not only consist of the quadrilaterals which were described above. The left and right ends of the SOR are covered by end disks (see Figure 2-3). Figure 2-2 shows that these end disks do not consist of quadrilaterals, rather they consist of triangles lying in a plane. In each triangle, one of the vertices is on the x -axis, while the other two vertices are identical with vertices of surface quadrilaterals.

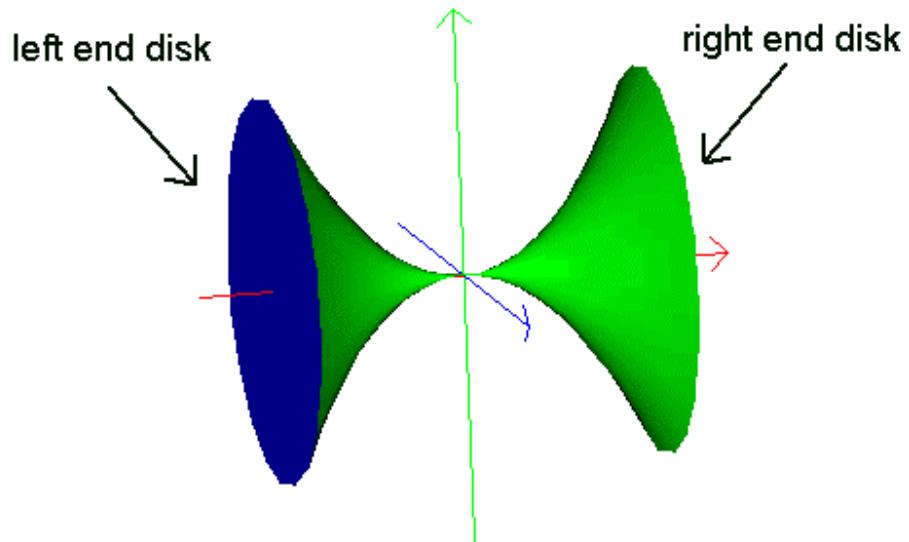


Figure 2-3: End Disks of a SOR

2.1.2 Light Source and Viewpoint

Kept in mind that the only goal of this algorithm is to provide a very fast SOR display algorithm for learning purposes, it does not matter how many light sources there are, which characteristics they have, and where they are located. Thus, both the simplest and the most efficient solution is to use just one light source, which is located at the position of the viewer, who is located at infinity in positive z -axis direction. So both

viewer and light source have the normalized direction vector $\vec{d} = \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix}$. This constant

vector is going to be implicitly used throughout this document. The light source emits white light in all directions equally. The light source position provides parallel light when hitting the SOR, the viewer's position results in parallel projection for the screen conversion, which is faster than perspective projection. The advantage of limiting the light color to white is that whatever surface color it hits, the reflected light's color remains the same. Only its intensity has to be adjusted for the impression of realistic shading.

It would require significant adjustments of the algorithm if different attributes for the above elements were desired.

2.2 *The Techniques and their Optimizations*

The following computations and display strategies are necessary to display a three dimensional SOR. Most of them could be optimized for this specific purpose.

2.2.1 Shading

The new shading algorithm is based on the idea of Gouraud shading. The elements which are derived from this classical algorithm are the following:

- The algorithm works on single polygons.
- Correct pixel colors are found for the vertices only.
- First the polygon edges are drawn using interpolated colors,
- Then the polygon interior is drawn scanline by scanline using interpolated colors.

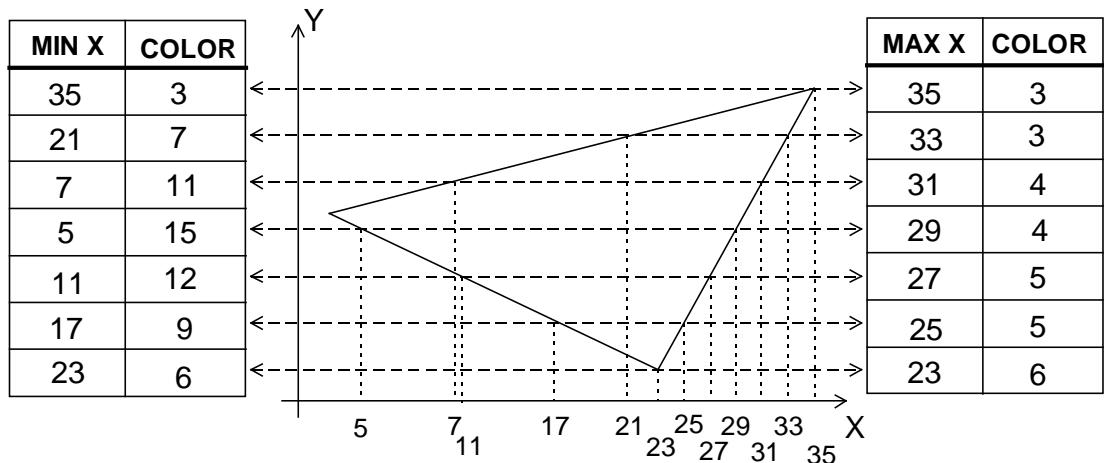


Figure 2-4: The MIN and MAX arrays

The new algorithm differs in the following decisive characteristic: The vertex colors do not have to be found using the neighboring polygons, instead they are computed by taking maximum advantage of the characteristics of the SOR. Since this part of the algorithm has to be executed for every frame and for each polygon separately, maximum performance is reached by using fixed-point variables for both of the interpolation parts.

This results in a little loss of accuracy, and it might not work correctly when displaying a very long but flat polygon, for example. However, it never draws beyond the borderlines. The limited accuracy only results in a slightly incorrect shading.

The new shading algorithm uses a MAX[y] and a MIN[y] array, both having as many elements as the screen has pixels in the y -direction (see Figure 2-4). Each element consists of an x value and a color. Initially all x values of the MAX[y] array get at most the value -1, all x values of the MIN[y] array get at least the value of one more than the largest possible x position. Then all edges of the polygon are processed from one end to the other, line by line. At every edge point the MAX[y] array gets the current x value if its current entry is smaller than that. The MIN[y] array works vice versa. In both cases the color value is adjusted as well. Moving along the line the color is interpolated linearly between the two end colors.

At the end of this process the MIN[y] array contains the left boundary, and the MAX[y] array contains the right boundary of the polygon. (This works for any number of edges!) The color values represent nicely shaded polygon edges.

The second part of the algorithm fills the polygon scan line by scan line, using the x values of the MIN[y] and MAX[y] arrays, linearly interpolating the colors for the pixels in a given scanline, from the colors at MIN[y] and MAX[y].

To make this algorithm fast, fixed-point arithmetic has to be used. The performance improvement over using floating-point arithmetic is substantial.

2.2.2 Rotation Calculations

The general rotation matrix for the rotation about the x -axis, rotating γ degrees, is:

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

There are similar matrices for the rotations about the y and z axes:

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

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

If not only a single rotation has to be applied, but all the three rotations have to be applied at the same time, we can multiply their respective rotation matrices, so that we get the composite rotation matrix

$$R_{\text{composite}}(\gamma, \beta, \alpha) = R_z(\gamma) \times R_y(\beta) \times R_x(\alpha) =$$

$$\begin{pmatrix} \cos \beta \cos \gamma & \sin \alpha \sin \beta \cos \gamma - \cos \alpha \sin \gamma & \cos \alpha \sin \beta \cos \gamma + \sin \alpha \sin \gamma & 0 \\ \cos \beta \sin \gamma & \sin \alpha \sin \beta \sin \gamma + \cos \alpha \cos \gamma & \cos \alpha \sin \beta \sin \gamma - \sin \alpha \cos \gamma & 0 \\ -\sin \beta & \sin \alpha \cos \beta & \cos \alpha \cos \beta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Applying $R_{\text{composite}}$ to a point P , we obtain:

$$R_{\text{composite}}(\gamma, \beta, \alpha) \cdot \begin{pmatrix} P_x \\ P_y \\ P_z \\ 1 \end{pmatrix} = \begin{pmatrix} P''_x \\ P''_y \\ P''_z \\ 1 \end{pmatrix} =$$

$$\begin{pmatrix} P_x \cos \beta \cos \gamma + P_y (\sin \alpha \sin \beta \cos \gamma - \cos \alpha \sin \gamma) + P_z (\cos \alpha \sin \beta \cos \gamma + \sin \alpha \sin \gamma) \\ P_x \cos \beta \sin \gamma + P_y (\sin \alpha \sin \beta \sin \gamma + \cos \alpha \cos \gamma) + P_z (\cos \alpha \sin \beta \sin \gamma - \sin \alpha \cos \gamma) \\ -P_x \sin \beta + P_y \sin \alpha \cos \beta + P_z \cos \alpha \cos \beta \\ 1 \end{pmatrix}$$

This case involves 16 multiplications, 4 additions, and 9 assignments to compute the composite rotation matrix. After that, as long as the rotation angles remain the same, 9 multiplications, 6 additions, and 3 assignments are needed to calculate the rotation of a point.

Another way to calculate the rotation about all the three axes would be to apply the rotations incrementally:

1. Rotation about the x -axis:

$$\begin{aligned}P'_y &= P_y \cdot \cos \alpha - P_z \cdot \sin \alpha \\P'_z &= P_y \cdot \sin \alpha + P_z \cdot \cos \alpha\end{aligned}$$

2. Rotation about the y -axis:

$$\begin{aligned}P'_x &= P_x \cdot \cos \beta + P_z \cdot \sin \beta \\P''_z &= P'_z \cdot \cos \beta - P_x \cdot \sin \beta\end{aligned}$$

3. Rotation about the z -axis:

$$\begin{aligned}P''_x &= P'_x \cdot \cos \gamma - P'_y \cdot \sin \gamma \\P''_y &= P'_x \cdot \sin \gamma + P'_y \cdot \cos \gamma\end{aligned}$$

For each rotation this involves 4 multiplications, 2 additions, and 2 assignments.

So for all three rotations, 12 multiplications, 6 additions, and 6 assignments are needed.

The complexities for the rotation computation of n points are summarized in Table

2-1:

	# Multiplications	# Additions	# Assignments
Pre-Computation of Composite Matrix	16	4	9
Rotation using Composite Matrix	$9n+16$	$6n+4$	$3n+9$
Rotation about one axis only	$4n$	$2n$	$2n$
Sequential computation	$12n$	$6n$	$6n$

Table 2-1: Complexities of Rotations of n Points

The decision about which technique to use depends on the specific application for which the rotation is needed. If there is only a rotation around one axis at a time, it is fastest to use the respective rotation matrix for a single angle. If the rotation is around all axes, it depends on how many rotations occur for the same combination of angles. Here only the multiplications need to be taken into consideration, since additions and assignments are much less complex. Table 2-2 lists the number of multiplications which are needed to rotate n points about all three axes. It shows that if there are less than six rotations, the sequential computation is fastest, because the pre-computation of the composite rotation matrix takes longer. But as soon as six or more computations are needed, it is faster to calculate the composite matrix.

Number of Points (n)	# Multiplications using Pre-Computation Method	# Multiplications using Sequential Computation
1	25	12
2	34	24
3	43	36
4	52	48
5	61	60
6	70	72
7	79	84
8	88	96

Table 2-2: Comparison of Computation Methods

The algorithm which is developed in this thesis will need one rotation for each point of the SOR at any rotation angle. Since many more than six rotations for each combination of angles are needed here, the fastest way to do this is to pre-compute the composite rotation matrix and to use it for each subsequent rotation.

2.2.3 Normal Calculations

Normal calculations have to be done for both the points on the surface itself, and for the two disks which cover the ends of the SOR (see Figure 2-3).

2.2.3.1 Points on the Surface

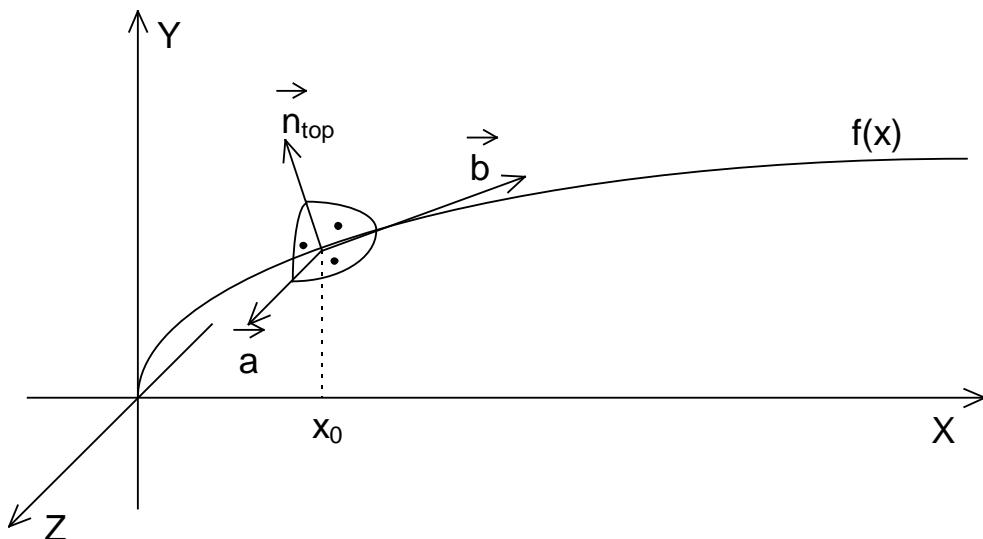


Figure 2-5: The Tangent Component

The only reason for computing the normals of all vertices is to find the color for the shading. In case of a SOR finding the vertex normals is fairly easy. Figure 2-5 and Figure 2-6 show the elements which have to be considered in order to find a surface normal. First the normal at the very top of the object is to be found for a specific x

coordinate x_0 . It is the cross product of vector $\vec{a} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$ pointing from the vertex into z

direction, and the vector $\vec{b} = \begin{pmatrix} 1 \\ f'(x_0) \\ 0 \end{pmatrix}$ pointing from the vertex and tangent to the curve,

thus its y -coordinate is the slope of the derivative of the function f at the coordinate x .

This cross product equals

$$\vec{n}_{top} = \vec{a} \times \vec{b} = \begin{pmatrix} -f'(x_0) \\ 1 \\ 0 \end{pmatrix}.$$

To find the normal vector at an arbitrary position around the disk, only the y and z values have to be adjusted. The x value remains the same, since the rotation occurs in the y,z -plane. The resulting formula for the normal vector at the angular position of φ is:

$$\vec{n}(\varphi, x_0) = \begin{pmatrix} -f'(x_0) \\ \cos(\varphi) \\ \sin(\varphi) \end{pmatrix}.$$

The shading formula assumes a normalized vector. So

$$\vec{n}_{norm}(\varphi, x_0) = \frac{1}{\sqrt{(-f'(x_0))^2 + \cos^2(\varphi) + \sin^2(\varphi)}} \begin{pmatrix} -f'(x_0) \\ \cos(\varphi) \\ \sin(\varphi) \end{pmatrix}.$$

This simplifies to:

$$\vec{n}_{norm}(\varphi, x_0) = \frac{1}{\sqrt{(-f'(x_0))^2 + 1}} \begin{pmatrix} -f'(x_0) \\ \cos(\varphi) \\ \sin(\varphi) \end{pmatrix}.$$

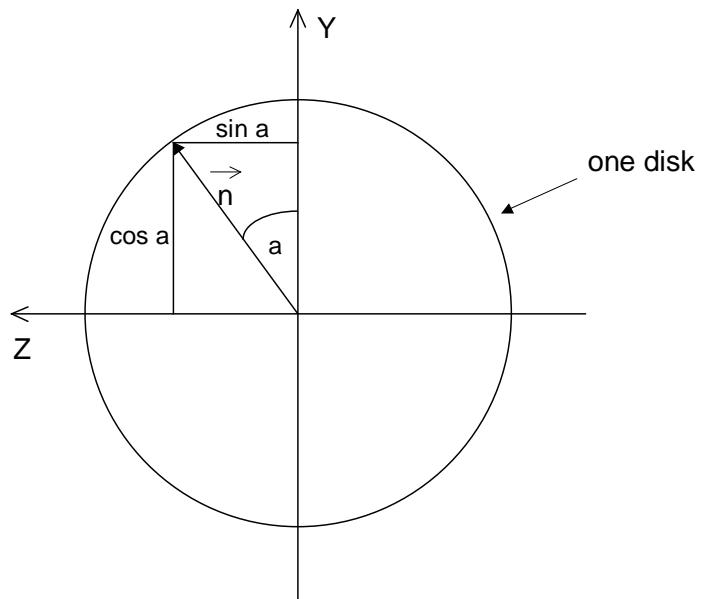


Figure 2-6: The Disk Components

2.2.3.2 Points on Disks

Finding the normal vectors for the points on the faces of the disks is easy. They have the form

$$\vec{n}_{disk}(x_0) = \begin{pmatrix} d \\ 0 \\ 0 \end{pmatrix},$$

where the value d can either be 1 for sides pointing into the positive x -axis direction, or -1 for disks pointing into the negative x -axis direction.

2.2.4 Projection to 2D Screen

The fastest way of performing projection, which produces realistic images, is parallel projection. Since in our 3D model, the viewer looks directly at the coordinate system's origin, and this vector is perpendicular to the screen plane, it suffices to take a 3D point, drop its z coordinate, and scale the other two coordinates so that they fit on the screen. Therefore the system has to know which coordinates the window boundaries represent. Let $WIDTH$ be the window width in pixels, $LEFT$ be the world coordinate's leftmost x value which is to be displayed, and $RIGHT$ is the rightmost one. Then a point P in 3D space has to be displayed at pixel coordinate Q in the window (assuming the window coordinates' origin is left). For maximum speed we use the following abbreviation:

$$WDTH_DIV_DX = \frac{WIDTH}{RIGHT - LEFT}$$

So for the resulting pixel coordinate we obtain:

$$Q_x = (P_x - LEFT) \cdot WDTH_DIV_DX$$

The y coordinate can be found analogously.

2.2.5 Back Face Culling

The technique of back face culling is used to determine which polygons are facing the viewer and are thus visible, and which are not. Ideally this requires the calculation of each polygon's normal, and rendering the polygon only if the normal's z coordinate is positive. However, this is inefficient. All four polygon vertex normals are already known, and we can sum their z coordinates instead. We compare this value to 0, and again only a value greater than 0 means the polygon is visible, otherwise it does not need to be drawn.

2.2.6 Painter's Algorithm

The painter's algorithm states that a 3D image can be drawn correctly if all its polygons are displayed starting with the one farthest from the view point, and drawing each polygon up to the one closest to the view point. This method assumes that polygons are convex, no two polygons intersect, no two polygons overlap in the z -axis direction,

and a decent depth sort algorithm has to exist. All of those conditions apply without modifications in the case of a SOR. The depth sort algorithm works like this:

First of all it has to be determined which end of the SOR is visible to the viewer.

This can be done by applying the current rotation transformation to a vector $\vec{v} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$

pointing in the positive x direction. The left edge of the SOR is farther away than the right one if the z value of the resulting vector is greater than 0. Otherwise it is vice versa.

Suppose the left edge is farther away. If the SOR is to be displayed without cylinders, all the surface polygons have to be drawn, starting with the leftmost ones, and a final end cover disk has to be displayed. If the SOR consists of cylinders, every other cylinder cover has to be drawn between the surface polygons. The polygons on a specific cylinder, as well as the triangles on a cylinder cover can be drawn in an arbitrary order, since they do not overlap.

2.2.7 Creation of Quadrilaterals

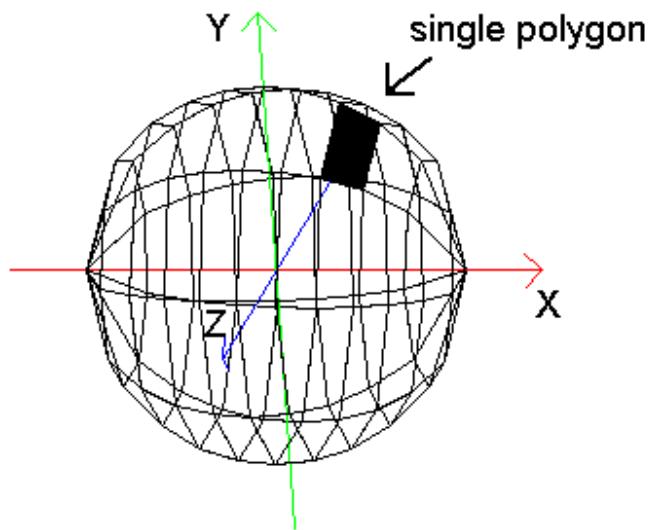


Figure 2-7: The Vertices in a SOR

Unlike many complex 3D objects, all the vertices which are necessary to draw a SOR could be computed while the application is running. However, it does make sense to calculate these points only once, which is after the user entered the surface formula, so that these calculations do not have to be done over and over again. These calculations should be fast, because one of the goals of the program will be to be able to change the number of SOR cylinders in realtime to create the impression of an animation. However, this requires the complete recalculation of all surface vertices. (see Figure 2-7)

There is a similar issue about the assignment of the vertices to polygons. Although this could also be done "on the fly", it is a little bit faster to make a list of the SOR's polygons and save it in memory. This list only needs to be changed when the number of vertices changes.

2.2.8 Color Computation

Usually a SOR consists of two basic colors. One for the surface, and another one for the cylinder covers. The system has to supply a number of shades in both basic colors. There is no requirement for a "true color" system, but in case of a palette based system, the shades of a color should have consecutive indices. This way there can be a direct conversion of surface normals into the visible shade. Each color can be characterized by its hue (base palette index) and its intensity (offset palette index).

In our lighting model, the visible shade consists of the ambient and the diffuse components. The ambient part insures that no color on the screen has a shade with an intensity below a minimum intensity, called the ambient intensity. According to Lambert's law, the diffuse component depends on the angle between the surface normal and the angle in which the light source hits the surface. The intensity is proportional to the cosine of this angle. Let \vec{n}_0 be the normalized surface normal, and \vec{l}_0 be the normalized vector to the light source. Then the cosine of the angle Θ between these vectors is:

$$\cos(\Theta) = \frac{\vec{n}_0 \cdot \vec{l}_0}{|\vec{n}_0| \cdot |\vec{l}_0|} = \frac{\begin{pmatrix} n_{0,x} \\ n_{0,y} \\ n_{0,z} \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}}{1 \cdot 1} = n_{0,z},$$

since both vectors are normalized, and the light source is in the direction $\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$ at infinity.

Thus, $\cos(\Theta) = n_{0,z}$. Since the diffuse reflection's intensity is proportional to $\cos(\Theta)$, it is also proportional to $n_{0,z}$. So as soon as a vertex normal's z coordinate is known, it can be mapped linearly to a shade, remembering to add the ambient light value, as well as cutting off at maximum intensity:

$$\text{color_index} = \min(I_{\text{ambient}} + I_{\text{diffuse}}, I_{\text{max}}).$$

2.2.9 Fixed-Point Arithmetic

The fact that floating-point arithmetic is slow, even when using special chips to speed them up, led to the idea of expressing floating-point numbers using integers. The integer is divided into an integer part and a fractional part. The numbers of bits used for the parts has to be determined according to the representation of the integer data type used. Using 32 bit integers, a sample division could be 24 bits for the integer part, 8 bits

for the fraction part. Having two fixed-point variables of the same type, they can be added using the normal addition operation. Multiplying however requires more attention, because the result has 16 fractional bits and thus only 16 integer bits. To convert it back to the original representation, the variable has to be shifted right by 8 bit in this case.

Fixed-point numbers make most sense in the core shading routines, which have to be called many times for each frame. Using them instead of all floating-point numbers throughout the program would speedup the result even more, but the price is a much harder to read source code and difficult debugging, since debuggers do not recognize fixed-point variables.

2.2.10 Look-up Tables

Look-up tables are usually used for the fast computation of trigonometric functions, mostly for sine and cosine. The creation of a look-up table is easy: all that is needed is a global array with as many entries as values that are needed. Typically 360 entries are used in graphics applications, since this means enough precision when displaying objects. Creating only the values for a fourth of a circle (for example), and getting the other results by negating the values in the correct way would save memory, but it needs a few more computations each time a value is looked up. The time for setting up

the look-up table may be four times as much for an entire circle, but since this is done only once at the beginning of an application, it is usually worth it.

Not only is the creation of a look-up table easy, but also its use. Instead of calling the mathematical function with the angle as a parameter, the array is addressed with the angle as a parameter.

In the SOR shading process there is one other point when a look-up table helps. This is when finding the correct color palette entry from the normal vector's z coordinate. This is a linear mapping process, and it can be implemented as a look-up table by converting the z coordinate to fixed-point, and creating just as many array entries as necessary to contain all the possible color shades for different values of the z coordinate.

2.2.11 Gradual Display of Rotation Process

To show how a SOR is generated by rotating the graph of a function around the x -axis, thus gradually displaying the SOR, there has to be a variable "gRevolution" in the system, which is globally accessible. This variable initially equals the number of cylinder sectors. The display routine contains a loop which draws one cylinder sector after another, up to "gRevolution". Finally the idle routine of the application checks if the variable "gRevolution" is smaller than its possible maximum "gSectors". If it is smaller, it is increased by one.

This way the user interface routine only has to set the variable "gRevolution" to the value of 1, whenever the user requests this kind of animation.

3 The Implementations

The last chapter introduced multiple 3D graphics techniques, which were explained and optimized for the special case of a SOR. This chapter will introduce an actual application, which uses all of those techniques for an optimized frame generation. To see both the performance differences, and the differences in programming effort compared to general 3D graphics toolkits, implementations in OpenGL and SPHIGS are going to be presented as well. In each case only the algorithmically relevant parts will be discussed.

3.1 *The Optimized Algorithm*

The optimized algorithm is platform independent. It will be described in pseudocode, in order to simplify both understanding and porting to other computer systems. This chapter will explain all the necessary steps for the SOR to be drawn on screen, as done in the sample program. All the algorithms are named according to the corresponding routines in the code. System dependent issues will not be discussed here. The actual algorithm is written in C, please refer to the appendix for the complete source code of the module "fast-sor.cpp".

3.1.1 Create a New Vertex List

Chapter 2.2.7 mentioned that the creation of a list of vertices is not absolutely necessary in case of a SOR. But it also says that it makes sense to get maximum speed. So this implementation does create such a list. Each list entry contains 3 vertex coordinates and 3 normal coordinates, both in world coordinates and for the current orientation. This is a pseudocode representation of the process:

<u>Inputs:</u>	- number of cylinders - number of sectors
<u>Outputs:</u>	- number of vertices - pointer to vertex list
<u>Algorithm:</u>	
1. compute the number of vertices needed	
2. allocate memory for the vertex structures	
3. create midpoints for left and right end disks	
4. for all cylinders do	
for all cylinder sectors do	
compute vertex coordinates	
compute vertex normals	
end for	
end for	
5. update polygon list	

Algorithm 1: NewVertexList

3.1.2 Create a New Polygon List

Although the program does not necessarily need a list of polygons (it could be generated at runtime), this is done to get both maximum speed and easier debugging. The polygon data structure mainly contains the array indices of the polygon vertices, and the polygon normals. The creation process looks like this:

Inputs:	- number of cylinders - number of sectors - number of vertices - pointer to vertex list
Outputs:	- number of polygons - pointer to polygon list
Algorithm:	
1. compute the number of polygons needed	
2. allocate memory for the polygon structures	
3. create left and right end disks	
4. for all cylinders do	
for all cylinder sectors do	
compute the array indices of the polygon vertices	
compute the polygon normals	
end for	
end for	

Algorithm 2: NewPolygonList

3.1.3 Draw the Surface

This is the main drawing routine. It computes the current coordinates of both vertices and vertex normals. It draws the coordinate axes, it shades the polygons, and it draws the function outline if necessary. Surface polygons are shaded using the algorithm from chapter 2.2.1, cover disks use flat shading with a previously computed color, since their orientation to the lightsource is the same for the entire object.

<u>Inputs:</u>	- number of vertices - pointer to vertex list - number of polygons - pointer to polygon list
<u>Output:</u>	- calls shading routines
<u>Algorithm:</u>	<ol style="list-style-type: none"> 1. for all vertices in the current surface orientation do <ol style="list-style-type: none"> compute vertex coordinates (rotation, see chapter 2.2.2) compute normals project vertices to current screen coordinates (see chapter 2.2.4) compute vertex colors (see chapter 2.2.8) end for 2. determine the polygons' drawing order (for painter's algorithm) 3. draw coordinate axes 4. for all polygons do <ol style="list-style-type: none"> if the current polygon faces the viewer (see chapter 2.2.5) then <ol style="list-style-type: none"> if the polygon is on the surface or a cover disk then <ol style="list-style-type: none"> draw the polygon (see chapter 2.2.1) end for 5. if in 0 degrees position then <ol style="list-style-type: none"> draw function outline

Algorithm 3: DrawSurface

3.1.4 Rotate a Point

<u>Inputs:</u>	<ul style="list-style-type: none"> - pointer to point to rotate - current rotation angles - sine look-up table - cosine look-up table
<u>Output:</u>	<ul style="list-style-type: none"> - pointer to rotated point
<u>Algorithm:</u>	
<ol style="list-style-type: none"> 1. for all the three possible rotation angles do <ol style="list-style-type: none"> if angle has changed then { <ol style="list-style-type: none"> memorize current angle precompute sine and cosine for this angle 	
<p style="padding-left: 20px;">}</p>	
<p style="padding-left: 20px;">end for</p>	
<ol style="list-style-type: none"> 2. if alpha or beta have changed then <ol style="list-style-type: none"> calculate alpha and beta dependent rotation matrix elements 	
<ol style="list-style-type: none"> 3. if beta or gamma have changed then <ol style="list-style-type: none"> calculate beta and gamma dependent rotation matrix elements 	
<ol style="list-style-type: none"> 4. if any rotation angle has changed then <ol style="list-style-type: none"> calculate rotation matrix elements which depend on all angles 	
<ol style="list-style-type: none"> 5. multiply rotation matrix and input point 	

Algorithm 4: Rotate

Chapter 2.2.2 develops the fastest way to rotate a 3D point. That is exactly how this routine is designed. It rotates a single 3D point about all the three coordinate axes using the rotation matrix. The elements of this matrix are only recalculated if the appropriate rotation angle has changed.

3.1.5 Project and Transform a Point to Screen Coordinates

The 3D to 2D projection is very simple. Since it is a parallel projection, it just maps the world x and y coordinate values to their respective screen coordinates, according to chapter 2.2.4:

<u>Inputs:</u>	- pointer to point to transform to the screen - screen width and screen height - world width and world height - left and bottom world edges
<u>Output:</u>	- pointer to point in screen coordinates
<u>Algorithm:</u>	
1. $\text{screen.x} = (\text{world.x} - \text{world.left_edge}) * \text{screen.width} / \text{world.width}$	
2. $\text{screen.y} = (\text{world.y} - \text{world.bottom_edge}) * \text{screen.height} / \text{world.height}$	

Algorithm 5: Project

3.1.6 Get a Vertex Color

This routine is passed a vertex and a base color. It then computes the color in the current lighting conditions. This is simply done by looking up the vertex normal's z coordinate in the lighting look-up table. The value found is added to the ambient light constant, and the result is checked for being within the valid range. Finally the base color value is added.

<u>Inputs:</u>	- pointer to normal vector - base index of color - color look-up table - ambience color value
<u>Output:</u>	- normal vector's color index
<u>Algorithm:</u>	
1. look vertex normal's z coordinate up in lighting look-up table	
2. add ambience color value	
3. if result is not in range then set color value to maximum value	
4. add base color value to set hue	

Algorithm 6: GetVertexColor

3.1.7 Shade a Quadrilateral

This is the main shading routine which processes a given quadrilateral. First the four boundary lines are found, then the interior is filled.

<u>Inputs:</u>	- pointer to polygon - height of output window
<u>Output:</u>	- calls basic shading routines
<u>Algorithm:</u>	
1. initialize boundary arrays	
2. for all four boundary lines do	
Process Boundary Line (see Chapter 3.1.8)	
end for	
3. draw the polygon (see Chapter 3.1.9)	

Algorithm 7: ShadeQuadrilateral

3.1.8 Calculate the Boundary Line Shades

This routine calculates the coordinates and colors for a pixel line between two projected points. The x -coordinate for each point on the line is checked to see if its value is between the current boundary array values. If it is not, the respective array values in both the x coordinate arrays, and in the color arrays are adjusted. However, no pixels on screen are affected by this algorithm. The pseudocode for this procedure is:

Inputs:	<ul style="list-style-type: none"> - two line end points - colors of the two line end points - minimum and maximum x coordinate arrays - two pixel color arrays
Outputs:	<ul style="list-style-type: none"> - updated minimum and maximum x coordinate arrays - two updated pixel color arrays

Algorithm:

1. if point a is below point b then
 - swap points
2. set x and y position counters to coordinates of point a
3. color counter = color value of point a
4. color increment = $(\frac{1}{b_y - a_y})$
5. x position increment = $(\frac{b_x - a_x}{b_y - a_y})$
6. while y position counter is above point b do
 - if new minimum x position found then
 - adjust minimum array
 - if new maximum x position found then
 - adjust maximum array
 - x position = x position + x increment
 - color = color + color increment
 - y = y + 1

end while

Algorithm 8: ProcessBoundaryLine

3.1.9 Draw a Polygon

This routine uses the data of the polygon boundary arrays to render a polygon on screen. It draws the polygon scanline by scanline, interpolating the color inbetween according to the colors in the previously calculated bounding arrays. This routine is the only one in the entire program which directly accesses the pixmap!

<u>Inputs:</u>	- minimum and maximum x coordinate arrays - two pixel color arrays - pointer to output window pixmap
<u>Outputs:</u>	- updated output window pixmap

Algorithm:

```

1. for all y positions in polygon do
    2. if no boundary is set then
        continue with next y position
    3. initialize  $x$  counter with minimum boundary value at this y position
    4. initialize color counter with its respective color value
    5. if boundary = 0 then
        color increment = 0
    else
        color increment = 1 / (width of this scanline)
    7. pixel pointer = screen coordinate of beginning of current scan line
    8. while  $x <$  right edge of this scanline do
        draw pixel in current color
        pixel pointer = pixel pointer + 1
        color value = color value + color increment
         $x = x + 1$ 
    end while
end for

```

Algorithm 9: ProcessScanLines

3.2 *The Microsoft Windows Version*

This version ("Fast SOR") is the only one which is based on the algorithm which was described in chapter 2. It runs on Microsoft Windows 95 and NT systems provided the WinG driver is installed properly (it is distributed by Microsoft). The source code was compiled with Microsoft Visual C++ 5.0.

The implementation uses direct addressing of the output graphics pixmap for maximum performance. Although pseudocode is used, all procedure names used below are also used in the source code. The chapter titles correspond to them as well.

3.2.1 The Color Palette

The size of the pixmap depends on the number of bytes each pixel uses. The smaller this number, the faster the algorithm, and the smallest acceptable number is one, which allows 256 different colors. Most computer systems do not have fixed color modes for 8 bit per pixel graphics modes, instead this byte is used as an index into a color palette. The graphics hardware uses the palette as a look-up table to find the actual colors it has to display. It is up to the programmer to setup this color palette, the Windows interface only requires the programmer not to use the first ten and the last ten entries, these are fixed. (So that the window elements do not change when an application changes the palette.)

If shading is involved there have to be a number of shades for every basic color which occurs on the object. So a good strategy is to restrict the number of different object colors, so that the number of shades of each color is at least 16. Otherwise the result looks rather ugly.

3.2.2 The Graphics Engine

It was not possible for Windows programmers to directly access single pixels in a window until the release of the WinG extension. WinG provides the programmer with several API extensions, of which the two most important are:

- a method for accessing the pointer to the pixmap which represents the contents of a graphics window
- a method quickly copying a pixmap to an on screen window (bit block transfer)

The sample program does not use the WinG functions directly, they are addressed through a library file ("engine") providing the Windows related routines. This way porting the algorithm to other systems should be rather easy, because only the library file has to be rewritten.

3.2.3 The Dialog Elements

Since the sample program is a true Windows program, it has a menu bar and a dialog window. These and the regular event handling use callback routines, which are all located in a separate module ("fast-win.cpp"), so that they could be adapted to a different operating system. Additionally there is a resource file ("fast-sor.rc") which contains all the information about the dialog elements, and which is also MS Windows specific.

Figure 3-1 shows the dialog box which is used in the Windows implementation.

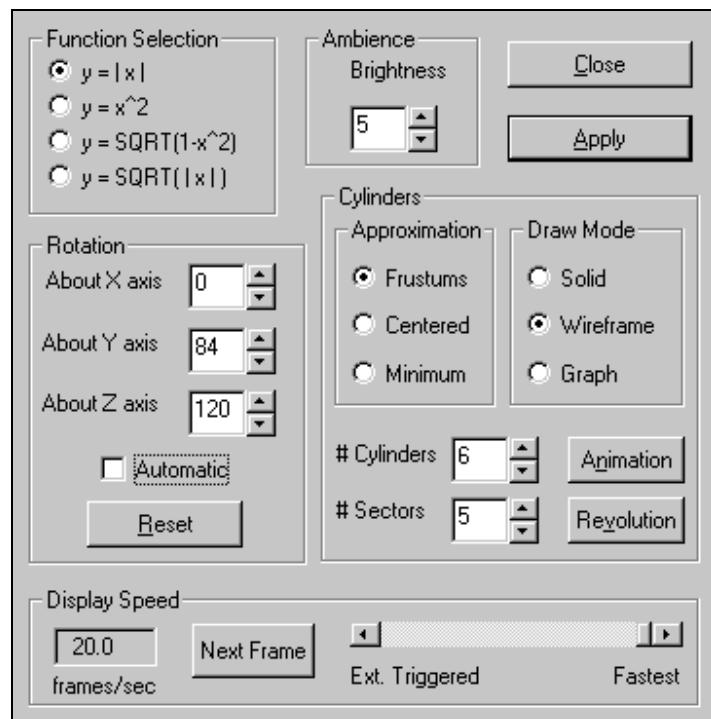


Figure 3-1: The Windows Dialog Box

3.2.3.1 The Function Selection

In the upper left corner of the dialog box the user can choose one of four available surface functions. They cannot be changed in the program, but it is easy to implement different functions in the source code.

3.2.3.2 The Ambience Color Selection

In the upper middle of the dialog box the user can adjust the ambience brightness. Usually the default value does not need to be changed, but this value shows that the ambience color can be chosen arbitrarily.

3.2.3.3 The Rotation Parameters

The left area of the dialog window enables the user to choose the angle from which he or she would like to look at the SOR. The values show the SOR's rotation around the coordinate axes in degrees. First the x -axis rotation is applied, then the object is rotated about the resulting y -axis, and finally it is rotated about the resulting z -axis.

When "Automatic" is checked, the SOR rotates automatically, so that the user gets a good impression of the three dimensional object.

3.2.3.4 The Approximation Modes

The dialog box in Figure 3-1 offers three different approximation modes: "Frustums", "Centered", and "Minimum".

- In "Frustums" mode, the SOR appears as smooth as possible, because the cylinders of which it consists, are not directly visible. They are represented as frustums, and as the number of cylinders increases, the frustums become smaller, and thus the SOR looks smooth. This is the three dimensional equivalent of a two dimensional function, which is approximated by a number of lines.
- In "Centered" mode the height of the cylinders is determined by the function value in the middle of the cylinders. Thus, only the middle of each cylinder has exactly the correct height for the respective x value of the SOR.
- In "Minimum" mode, the height of a cylinder depends on the minimum function value at their position. Thus, the cylinders are always smaller than the exact function graph.

3.2.3.5 The Draw Modes

Figure 3-1 offers three different draw modes for the SOR:

- "Solid" means that the SOR is rendered as a solid object in the three dimensional space.
- "Wireframe" draws only the polygon edges of the SOR. This can be used to show the polygons of which the SOR consists.
- "Graph" draws a planar polygon, of which the top edge is the function graph, the bottom edge is the x -axis, and the left and right edges are vertical lines at the clipping positions of the SOR. This draw mode is to show the function which is rotated to obtain the SOR.

3.2.3.6 Animation

In the lower right of the dialog window, the user can choose the number of cylinders and sectors of which the SOR consists. When values are chosen that give a SOR representation which is similar to the ideal one (e.g. 30 cylinders, 20 sectors), the buttons "Animation" and "Revolution" trigger animations starting with 1 cylinder or 3 sectors, and counting up to the selected values.

3.2.3.7 Display Speed Adjustments

If in the animation modes the display speed is too fast, the user can slide the speed control bar to the left, until he or she gets the desired animation speed. In the leftmost

position, the animation is stopped. In this case the user can switch to the next frame by clicking on the "Next Frame" button. If the scroll bar is in its rightmost position, the animation is executed without delay.

3.3 The OpenGL Version

The OpenGL implementation ("ShadeSOR") of the surfaces of revolution program compiles with the MS Visual C++ 5.0 compiler on the PC, and with the Code Warrior compiler on the Apple Macintosh. System specific lines of code are put between compiler directives, so that one source code file is sufficient for both systems. If the macro "WIN32" is defined, the file compiles on a Windows machine, otherwise it compiles on the Macintosh. The compiler directive disables the display of the rendered number of frames per second on the Macintosh. Figure 3-2 shows the MS Windows output.

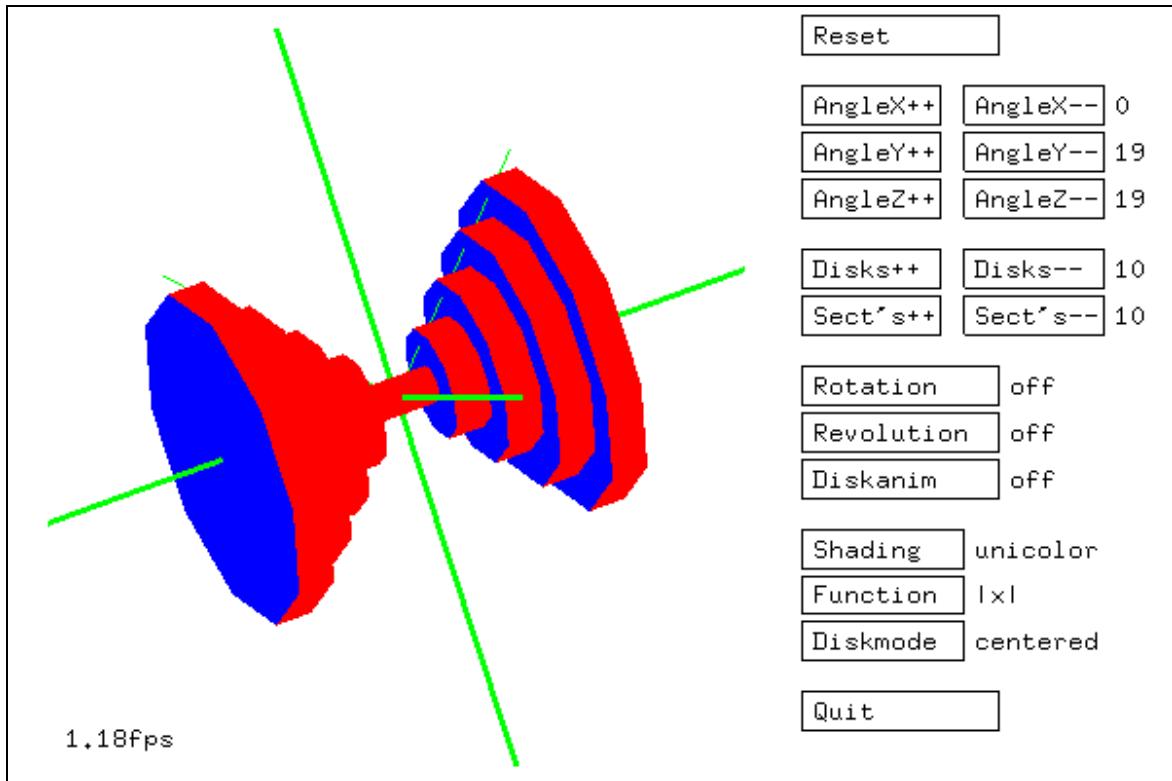


Figure 3-2: The OpenGL implementation

Since the OpenGL version is only designed to compare it to the new algorithm, it will not be discussed in the same depth as the MS Windows version was. The following paragraphs will give short descriptions of the most important routines, in processing order.

The main() function creates the output window, and it installs the callback routines. It initializes the surface parameters, and it sets up the lighting parameters.

The function NewVertexList() determines which display mode is active, and it branches to the generation routines of objects with or without cylinders. Since the basic rendering elements in OpenGL are triangles, both surface polygons, and end disk polygons are triangles. The new algorithm uses quadrilaterals for the surface polygons, and triangles for the end disks.

Since SmoothSOR() is a subset of CylindersSOR(), only the latter one will be described here. CylindersSOR() first allocates memory for the vertex list. Then it computes the vertices on the x -axis, which are used as midpoints for the cover disks. After that all the surface vertices are computed. Then the number of triangles needed is determined, and memory is allocated for them. The memory is filled with the lists of surface and cylinder covering triangles. Finally the normals for all the triangles are computed in the GenerateNormals() routine.

The NewPolygonList() function creates an OpenGL display list for the previously computed triangles, and for the function outline, which is generated here. From this point on the program can display the SOR only by addressing the display list. This is to save execution time.

If only the view angle changes, it is not necessary to recompute the display list. Instead, it is sufficient to call the RedrawScreen() function.

Another important routine is the MouseCallback() function. It processes all the mouse messages, which are triggered whenever the mouse button is pressed. This routine switches the various interaction possibilities, which can be seen on the screen (see Figure 3-2).

3.4 The SPHIGS Version

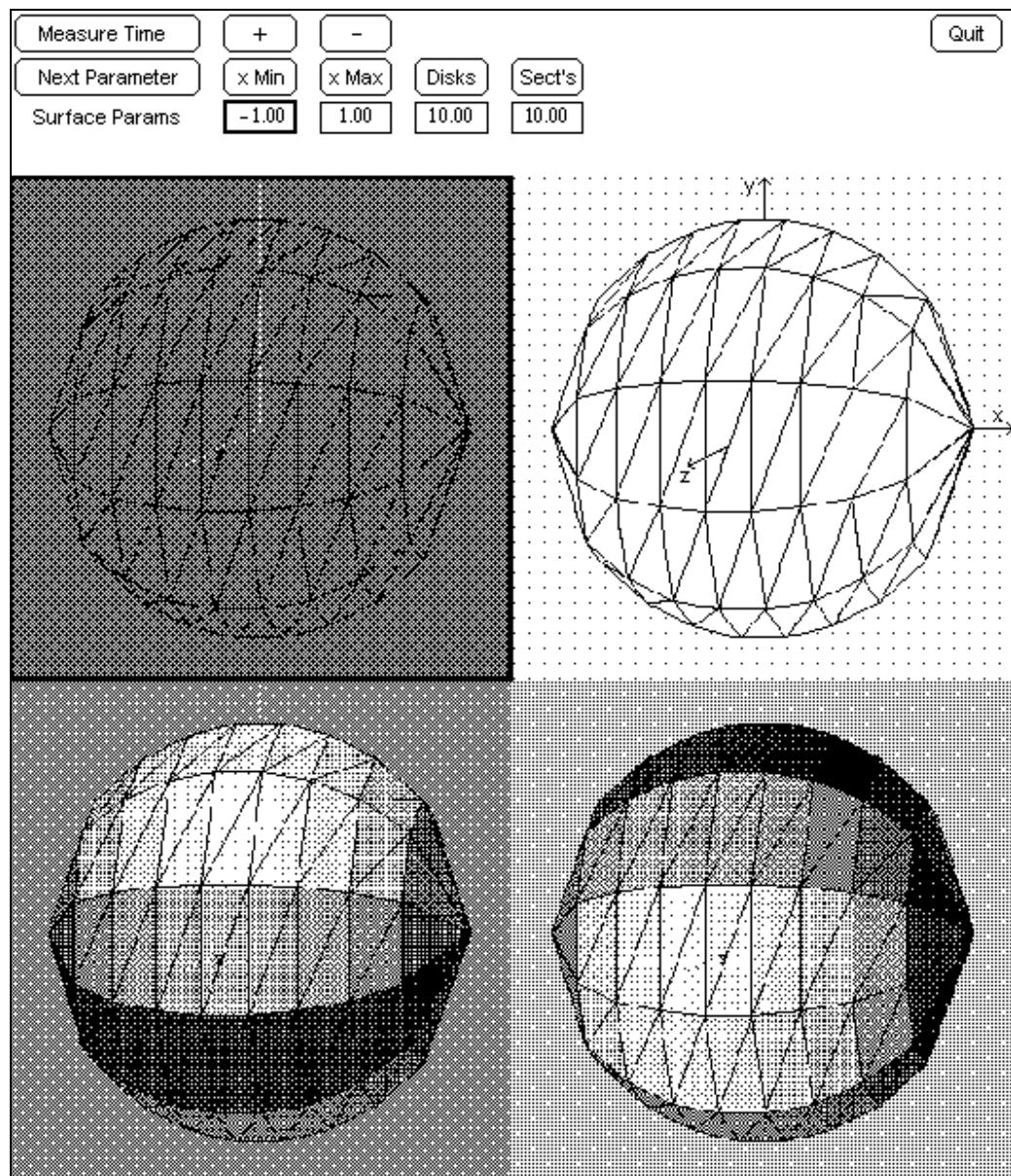


Figure 3-3: The SPHIGS implementation

Similar to the OpenGL version, the SPHIGS version ("Unix SOR") was made for performance comparison reasons only. It was compiled with the GNU C compiler "gcc" on a Linux PC.

There is only one surface function available, which represents a circle (see Figure 3-3). The output window contains four equally sized windows, each of them is individually accessible by the user. By default the top left window shows a wireframe representation, the top right window shows a flat shaded object, and the two bottom windows use light source dependent shading with different locations of the light source.

Internally the data is kept in a structure, of which the format is predefined by the SPHIGS graphics library. After the vertex coordinates have been calculated by the CreateSurfaceOfRevolution() function, the CreateSurfaceStructure() routine initializes the polygon structure it at the beginning of the program. Whenever the vertex positions change, this structure has to be updated by the UpdateSurfaceStructure() function.

The function DisplayInitialViews() shows the output windows for the first time. If the user chooses a different current output window, SelectView() is called. If the time for the generation of the output image is important, RedrawViewport() can be called, and it displays the required time on the screen. This is done whenever the "Measure Time" button is pressed.

Most of the other routines are either called only once, for initialization reasons, or they are used for the button handling, and therefore they will not be discussed here.

3.5 Porting to other Systems

The MS Windows version, which is the only one implementing the new algorithm, is designed to be easily portable to other computer systems. As long as a C compiler is available for the target system, only the Graphics Toolkit library ("GrafTool") has to be adapted, and the Windows message handling ("Fast-Win") has to be converted. The SOR manipulation routines are system independent and can thus be imported directly into the new source code file.

If it happens that the conversion of the Windows GUI (Graphic User Interface) message handling routines cannot be done, a new set of routines has to be written on the target system. But the interface to the main program's parameters is simple, because all the interaction parameters can be accessed as global variables.

This independence of the interaction mechanism also enables the program to be included in the TEMATH project. In this case it would receive all its input from the TEMATH interface.

4 Performance Analysis

This chapter will discuss the performance of the new algorithm in comparison to the current 3D graphics environments OpenGL and SPHIGS. A runtime comparison will be done, and profiler outputs showing the exact locations of speed traps will be presented. The complexity of the new algorithm will be determined. Finally possibilities for future extensions to the algorithm will be discussed.

4.1 Comparison of the Implementations

Table 4-1 shows the number of frames per second which can be displayed by the three different implementations of 3D surfaces of revolution. The following characteristics were the same for all measurements:

- the programs were executed on a Pentium 120 PC
- the new algorithm and the OpenGL version used MS Windows 95
- the output area was 300 x 300 pixels
- the surface function was: $f(x) = \sqrt{1 - x^2}$ (sphere)
- in case of light source shading, only one light source was used

The table shows frames per second for different combinations of cylinders, sectors, and rendering modes. The rendering modes are:

- WW = MS Windows version, wireframe, frustums
- WL = MS Windows version, lit shading, frustums
- WD = MS Windows version, lit shading, centered cylinders
- OF = OpenGL version, flat shading, frustums
- OL = OpenGL version, lit flat shading, frustums
- OD = OpenGL version, lit flat shading, centered cylinders
- LW = Linux version, wireframe, frustums
- LF = Linux version, flat shading, frustums
- LL = Linux version, lit flat shading, frustums

In Table 4-1, ">10.0" means that more than 10 frames can be displayed per second, but the exact number cannot be computed, because the timer resolution is too low. "n/a" means that no measuring could be done due to the enormous time consumption of the respective tests.

Cyl's	Sect's	WW	WL	WD	OF	OL	OD	LW	LF	LL
3	3	>10.0	>10.0	9.1	5.0	5.0	5.0	125.0	43.5	27.0
5	5	>10.0	>10.0	9.1	5.0	5.0	3.6	100.0	15.9	15.2
7	7	>10.0	9.1	8.3	3.7	3.6	3.3	83.3	7.9	9.8
9	9	9.1	9.1	8.3	3.6	3.5	3.0	58.8	3.8	6.7
11	11	9.1	8.3	8.3	3.7	3.6	2.7	45.5	2.4	5.1
13	13	8.3	8.3	7.7	3.3	3.1	2.5	35.7	1.6	3.9
15	15	8.3	7.7	7.1	3.2	3.0	2.3	29.4	1.1	3.0
17	17	8.3	7.7	6.7	3.1	2.9	2.0	23.3	0.8	2.5
19	19	7.7	7.1	6.3	3.0	2.8	1.9	19.2	0.6	2.1
21	21	7.1	7.1	5.9	2.9	2.6	1.8	15.9	0.5	1.8
50	50	3.6	3.8	2.5	1.6	1.2	0.6	n/a	n/a	n/a
99	99	1.3	1.5	0.8	n/a	n/a	n/a	n/a	n/a	n/a
3	60	8.3	7.7	7.1	3.7	3.5	2.9	38.5	15.4	4.2
9	20	8.3	8.3	7.7	3.5	3.2	2.6	35.7	7.8	4.0
20	9	8.3	8.3	7.1	3.2	3.0	2.3	34.5	4.8	3.4
60	3	8.3	7.7	6.7	2.9	2.8	2.0	31.3	2.8	2.4

Table 4-1: Algorithm Speed Comparison

The most interesting results the table shows are:

- The new algorithm is up to 3.5 times as fast as the OpenGL implementation.
- The new algorithm is up to 4 times as fast as the SPHIGS implementation.
- In the modes with centered cylinders, compared to frustums, the new algorithm loses less speed with more polygons than the OpenGL implementation.

- The new algorithm is the only one that can display 99 cylinders and 99 sectors in a reasonable time.
- Comparing the 3x3 (cylinders times sectors) and the 21x21 tests, the new algorithm retains 65 % of its performance, the OpenGL version retains 50 %, and the SPHIGS version retains only 10 % of its 3x3 performance.
- The SPHIGS version is the fastest of all when displaying objects with very few polygons.
- The SPHIGS version is the slowest of all when displaying objects with many polygons.

4.2 Profiler Results

Both the MS Windows version, introducing the new algorithm, and the OpenGL version were processed by the Visual C++ profiler. The following profiler outputs are shortened to show only the most interesting values. For a reference of the original profiler outputs, please refer to chapter 6.1. Both programs ran for approximately 45 seconds. During this time the Windows version displayed 374 frames (8.4 fps), while the OpenGL version generated only 233 frames (5.0 fps).

4.2.1 MS Windows Version

Table 4-2 shows the most important information of the profiler output of the MS Windows version ("Fast SOR"), containing the new algorithm. Please see the appendix for the complete profiler output.

Function Time	%	Function + Child Time	%	Hit Count	Function Name
36309.699	81.3	36309.699	81.3	377	_WinGBitBlt@32
3561.250	8.0	3601.149	8.1	15682	ProcessScanLines()
1462.713	3.3	1462.713	3.3	374	GT.Clear()
710.198	1.6	710.198	1.6	62728	ProcessBoundaryLine()
497.585	1.1	5844.597	13.1	374	DrawSurface()
398.233	0.9	398.233	0.9	94996	Rotate()
382.284	0.9	36691.982	82.2	377	GT.Paste()
294.661	0.7	3190.973	7.1	13085	ShadeQuadrilateral()
190.438	0.4	43995.080	98.5	374	RedrawSOR()
101.335	0.2	101.335	0.2	34034	Project()
80.306	0.2	80.306	0.2	4114	GT.Line()
77.839	0.2	77.839	0.2	29920	GetVertexColor()
57.774	0.1	1472.809	3.3	2597	ShadeQuadConst()
27.759	0.1	27.759	0.1	15360	GT.SetPixel()
25.517	0.1	105.823	0.2	1870	DrawArrow()
3.221	0.0	43881.998	98.3	373	IdleCallback()
0.230	0.0	0.230	0.0	70	ComputeNormal()
0.043	0.0	0.043	0.0	165	Func()

Table 4-2: Fast SOR Profiler Output

The table shows that 81.3 % of the time are used by the WinG function which copies the image buffer from memory to screen. Only 18.7 % of the entire runtime are needed by the image generation routines. Among these ProcessScanLines() is the most time consuming one, it uses 8 % of the time. Thus it was worth optimizing this routine. It

uses fixed point arithmetic, and shift operations instead of multiplications. The next one in the list is GT.Clear(). It uses 3.3 % of the running time only for clearing up the image buffer before a new image is drawn. This is an external routine, and therefore it cannot be optimized any more.

All the rest of the program needs only 7.4 % of the running time. Some frequently used routines, which might look complex at first, prove to be efficient (Rotate() or Project()). The profiler proves that the optimization was successful.

4.2.2 OpenGL Version

Table 4-3 shows the most important information of the profiler output of the OpenGL version ("ShadeSOR"), running on a Windows machine. Please refer to the appendix for the complete output.

Function Time	%	Function + Child Time	%	Hit Count	Function Name
26472.044	56.4	26472.044	56.4	233	DrawFunction()
17250.669	36.7	17250.669	36.7	233	_glutSwapBuffers
2826.549	6.0	2826.549	6.0	2330	_glutBitmapCharacter
197.383	0.4	46773.589	99.6	233	RedrawScreen()
26.944	0.1	2853.493	6.1	233	WriteText()
4.185	0.0	46219.506	98.5	230	IdleCallback()
0.936	0.0	0.936	0.0	355	Func()
0.332	0.0	0.332	0.0	140	VectorProduct()
0.151	0.0	0.151	0.0	2	SetViewParams()
0.067	0.0	0.067	0.0	140	SetColor()

Table 4-3: Profiler Output of OpenGL version

The OpenGL implementation uses 56.4 % of the time on the image drawing process, which mainly consists of calling the SOR display list. An additional 36.7 % of the time are used to swap the memory buffer to the screen. Surprisingly the bitmap font displaying routine is not very time efficient, it uses 6.0 % of the runtime. All the rest of the functions use only 0.9 % of the time.

The profiler output shows that there is no unnecessary bottleneck in the image generation process. Most of the time is used for OpenGL's internal routines, which cannot be omitted or replaced. At least all the rest of the functions are fast enough, so that they do not use a significant amount of time in the displaying process.

4.3 Complexity

This chapter will discuss the complexity of the program's core functions, which are the functions that are necessary to display the 3D object on the screen. The functions are going to be listed in the order of processing. First a list of vertices has to be created by NewVertexList(), which uses Func() to get the function values. These vertices are then combined to polygons by NewPolygonList(). In order to draw the object, the function DrawSurface() is called. For rotations it calls Rotate(), for 3D to 2D projections it uses Project(), it calls GetVertexColor() to find out about the colors, and depending on the type of polygon it calls ShadeQuadrilateral() or ShadeQuadConst() for the shading. The latter routines then call ProcessBoundaryLine() once for each polygon edge, and they call ProcessScanLines() to draw the polygons.

Table 4-4 lists the complexity values for all of the above functions, and it shows the sums of operations which are necessary to generate a single frame with c cylinders and s sectors. The disk mode is set to frustums, the function is $f(x) = x^2$. All the rotational angles are set to 0, so that exactly one half of the polygons is visible. For $c=10$ and $s=7$, an experimental value for the average y extension of polygon edges is $\bar{y}=16.87$, and polygon scanlines have an average x extension of $\bar{x}=12.49$. You can get these numbers, when shading a typical SOR, by adding up all x and y extensions (i.e. the number of pixels between the minimum and maximum coordinates in either the x or the y direction) of all

polygon edges on the screen, divided by the number of polygon edges. " H " is the output screen height in pixels (here: 300). "Mults" is the number of not only multiplications, but also divisions. "Adds" are not only additions, but also subtractions. "Others" are modulo values, square roots, absolute values, and trigonometric functions.

Function Name	# Calls	# Mults	# Adds	#Assgnmnts	#Comp's	#Others
NewVertexList()	1	$7cs+7s+5$	$7cs+7s+4c+14$	$9cs+9s+2c+23$	$cs+s+4c+22$	$5cs+5s+c+1$
Func()	$2cs+2s+c+1$	1	0	1	1	0
NewPolygonList()	1	2	$7cs+5c+10s+9$	$6cs+3c+18s+12$	$cs+5c+6s+13$	$cs+2s$
ComputeNormal()	cs	6	9	13	1	0
DrawSurface()	1	0	$cs+2s$	$3cs+5s+5$	$3.5cs+7s+1$	0
Rotate()	$3cs+4s+4$	9	6	3	4	0
Project()	$cs+s+2$	2	3	2	0	0
GetVertexColor()	$cs+s+3$	1	2	4	2	0
ShadeQuadrilateral()	$0.5cs+s$	2	0	16	0	0
ProcessBoundaryLine()	$2sc+4s$	$1.5 \bar{y} + 6$	$3 \bar{y} + 5$	$4 \bar{y} + 10$	$3 \bar{y} + 2$	0
ProcessScanLines()	$0.5sc+s$	$(3+\bar{x})H/9$	$(5+3\bar{x})H/9+2H+2$	$(5+\bar{x})H/9+H+4$	$(1+\bar{x})H/9+2H+2$	0

Table 4-4: Complexities of the Fast SOR Routines

In order to obtain the total number of operations, we have to multiply each function's number of calls with the respective number of operations for a specific operation type. To keep this calculation simple, we assume that the drawing of quadrilaterals is an atomic operation. (This means that `ProcessBoundaryLine()` and `ProcessScanLines()` are not part of the calculation.) Table 4-5 shows the total numbers of operations for each operation type, as well as an evaluation for $c=10$ and $s=7$.

Type of Operation	# Operations for any c and s	# Operations for $c=10, s=7$
Multiplications	$46cs + c + 50s + 51$	3631
Additions	$47cs + 9c + 48s + 59$	3775
Assignments	$56cs + 6c + 68s + 69$	4525
Comparisons	$22.5cs + 10c + 34s + 59$	1972
Others	$6cs + c + 7s + 1$	480

Table 4-5: Complexities by Type of Operation

The numbers of operations show that the connection between the number of cylinders and sectors in the SOR, and the number of operations necessary to render the SOR, is linear in each variable.

4.4 Further Optimization

What effort will be needed for a significant improvement of the algorithm? At present, the Fast SOR program spends 81.3 % (see section 6.2: Complete Profiler Output) of its run time in the operating system's routine which copies the image from memory to the screen. Suppose the computer system can draw a maximum of 20 frames per second. This means that it needs 50 ms (milliseconds) for the generation of one frame. The operating system uses 81.3 % of that time, which is 40.7 ms. This leaves 18.7 % = 9.3 ms for optimizations. In order to obtain one additional frame per second, the program would have to produce a frame in 1/21 seconds = 47.6 ms. The operating system again uses 40.7 ms, so the algorithm would have to do the same things as before in only 47.6 ms - 40.7 ms = 6.9 ms, which is about 3/4 of the original time.

This means in general: suppose the operating system uses s percent of the run time, and the current frame rate is f . If an optimization leaves a fraction of x percent of the original run time of the actual algorithm, the new frame rate f' is:

$$f' = \frac{f}{(1-s)x + s}.$$

The maximum frame rate that can be achieved from optimizing the routines can be calculated from the time the operating system needs to display a frame on screen. In the above example this is 40.7 ms. That means there can be a maximum number of

$\frac{1000 \text{ ms}}{40.7 \text{ ms}} = 24.6$ frames per second. In general, if a given operating system needs t

milliseconds to display a frame on screen, the maximum number of frames per second it

can display with a perfectly optimized algorithm is $\frac{1000 \text{ ms}}{t \text{ ms}}$ frames per second.

The above example shows that the developed algorithm cannot be improved substantially. This would only be obtained if the algorithm could manipulate the video memory directly, so that the operating system cannot interfere with the algorithm. But the resulting implementation would be highly machine specific, it would even depend on a specific version of the operating system. Currently, in most of the operating systems it is not even allowed to access video memory directly. This is to protect the other screen windows from being overwritten.

4.5 Possible Future Enhancements

Chapter 6.2.1 showed that the new algorithm uses much less time to generate the output image on an offscreen buffer, than Windows specific functions need to display this buffer on screen. Is it still possible to increase the speed of the image generation process, except by using a faster processor? One way would be to speed up the internal Windows functions. This could be done by a better display driver, which is the approach the Direct X extension uses. Or it can be done by hardware acceleration, which is provided by the 3D

acceleration cards on the graphics equipment market. Using a 3D card might well give OpenGL a good chance to outperform the new algorithm, since OpenGL is designed to take maximum advantage of this equipment. The new algorithm will not be able to use special 3D hardware, since it does all the 3D processing by itself.

Another area of algorithm enhancement would be the use of fixed-point arithmetic instead of the remaining floating-point computations. Although all the frequently called routines use fixed-point arithmetic already, there are still some less important routines which do not take advantage of it yet. Many of the high level vertex and normal calculations are still done with floating-point numbers.

The current version uses a core shading algorithm which is based on the Gouraud shading algorithm. Although it generates nicely shaded surfaces, it might be interesting to see if a Phong shading based algorithm would improve it.

Furthermore it might be interesting to figure out how a random position of the lightsource, instead of the fixed one in the new algorithm, would affect the speed of the image generation. It would certainly increase the image realism.

5 References

- James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, Reading, MA, 1996.
- Edward Angel. *Interactive Computer Graphics*. Addison-Wesley, Reading, MA, 1997.
- Mason Woo, Jackie Neider, and Tom Davis. *OpenGL Programming Guide, Second Edition*. Addison-Wesley, 1997.
- *OpenGL Reference Manual, Second Edition*. Addison-Wesley, 1997.
- Kevin Tieskoetter. *Black Art of Macintosh Game Programming*. Waite Group Press, 1995.

6 Appendix

6.1 List of Algorithms

Algorithm 1: NewVertexList	40
Algorithm 2: NewPolygonList	41
Algorithm 3: DrawSurface	43
Algorithm 4: Rotate.....	44
Algorithm 5: Project.....	45
Algorithm 6: GetVertexColor.....	46
Algorithm 7: ShadeQuadrilateral	47
Algorithm 8: ProcessBoundaryLine.....	48
Algorithm 9: ProcessScanLines.....	50

6.2 Complete Profiler Output

MS Visual C++ produces a more detailed output than the one given in chapter 4.2.

The complete output to both the WinG version, and the OpenGL version is presented in the following two chapters.

6.2.1 Microsoft Windows Version

Program Statistics

```
Total time: 44812.865 millisecond
Time outside of functions: 169.063 millisecond
Call depth: 7
Total functions: 53
Total hits: 308711
Function coverage: 88.7%
Overhead Calculated 14
Overhead Average 14
```

Module Statistics for fastsor.exe

```
Time in module: 44643.802 millisecond
Percent of time in module: 100.0%
Functions in module: 53
Hits in module: 308711
Module function coverage: 88.7%
```

Func		Func+Child		Hit	
Time	%	Time	%	Count	Function
<hr/>					
36309.699	81.3	36309.699	81.3	377	_WinGBitBlt@32 (wing32.def)
3561.250	8.0	3601.149	8.1	15682	ProcessScanLines() (fast-sor.obj)
1462.713	3.3	1462.713	3.3	374	TGT::Clear() (graftool.obj)
710.198	1.6	710.198	1.6	62728	ProcessBoundaryLine() (fast-sor.obj)
497.585	1.1	5844.597	13.1	374	DrawSurface() (fast-sor.obj)
398.233	0.9	398.233	0.9	94996	Rotate() (fast-sor.obj)
382.284	0.9	36691.982	82.2	377	TGT::Paste() (graftool.obj)
294.661	0.7	3190.973	7.1	13085	ShadeQuadrilateral() (fast-sor.obj)
190.438	0.4	43995.080	98.5	374	RedrawSOR() (fast-sor.obj)
150.936	0.3	44642.545	100.0	1	_WinMain@16 (fast-win.obj)
146.020	0.3	157.128	0.4	1	TGT::CreateUserPalette() (graftool.obj)
101.335	0.2	101.335	0.2	34034	Project() (fast-sor.obj)
80.306	0.2	80.306	0.2	4114	TGT::Line() (graftool.obj)
77.839	0.2	77.839	0.2	29920	GetVertexColor() (fast-sor.obj)
65.669	0.1	65.669	0.1	4	_WinGSetDIBColorTable@16 (wing32.def)
57.774	0.1	1472.809	3.3	2597	ShadeQuadConst() (fast-sor.obj)
33.779	0.1	300.528	0.7	61	WinProcParent() (fast-win.obj)
27.759	0.1	27.759	0.1	15360	TGT::SetPixel() (graftool.obj)
25.517	0.1	105.823	0.2	1870	DrawArrow() (fast-sor.obj)
22.432	0.1	22.432	0.1	15682	TGT::GetWidth() (graftool.obj)
17.467	0.0	17.467	0.0	15682	TGT::GetAddress() (graftool.obj)
10.519	0.0	10.519	0.0	1	TGT::ClearSystemPalette() (graftool.obj)
7.024	0.0	7.024	0.0	2	_WinGCreateBitmap@12 (wing32.def)
3.221	0.0	43881.998	98.3	373	IdleCallback() (fast-sor.obj)
1.680	0.0	2.883	0.0	1	NewVertexList() (fast-sor.obj)
1.594	0.0	1.594	0.0	1	InitLookupTables() (fast-sor.obj)
1.426	0.0	1.426	0.0	2	TGT::Delete() (graftool.obj)
0.969	0.0	0.969	0.0	377	TGT::Set() (graftool.obj)
0.930	0.0	1.160	0.0	1	NewPolygonList() (fast-sor.obj)
0.528	0.0	0.528	0.0	2	_WinGCreateDC@0 (wing32.def)
0.515	0.0	73.185	0.2	2	TGT::New() (graftool.obj)
0.450	0.0	0.450	0.0	1	TGT::RealizeUserPalette() (graftool.obj)
0.230	0.0	0.230	0.0	70	ComputeNormal() (fast-sor.obj)
0.212	0.0	2.936	0.0	7	WinProcChild() (fast-win.obj)
0.085	0.0	157.663	0.4	1	GeneratePalette() (fast-win.obj)
0.043	0.0	0.043	0.0	165	Func() (fast-sor.obj)
0.030	0.0	0.030	0.0	2	_WinGRecommendDIBFormat@4 (wing32.def)
0.000	0.0	0.000	0.0	1	TGT::TGT() (graftool.obj)
0.000	0.0	0.000	0.0	1	TGT::~TGT() (graftool.obj)

6.2.2 OpenGL Version

Program Statistics

```
-----
Total time: 47119.805 millisecond
Time outside of functions: 175.075 millisecond
Call depth: 9
Total functions: 76
Total hits: 4476
Function coverage: 64.5%
Overhead Calculated 14
Overhead Average 14
```

Module Statistics for shadesor.exe

```
-----
Time in module: 46944.730 millisecond
Percent of time in module: 100.0%
Functions in module: 76
Hits in module: 4476
Module function coverage: 64.5%
```

Func		Func+Child		Hit	
Time	%	Time	%	Count	Function
<hr/>					
26472.044	56.4	26472.044	56.4	233	DrawFunction() (shadesor.obj)
17250.669	36.7	17250.669	36.7	233	_glutSwapBuffers (glut_win.obj)
2826.549	6.0	2826.549	6.0	2330	_glutBitmapCharacter (glut_bitmap.obj)
197.383	0.4	46773.589	99.6	233	RedrawScreen() (shadesor.obj)
63.462	0.1	70.628	0.2	59	__glutWindowProc@16 (glut_win32.obj)
35.859	0.1	50.267	0.1	1	__glutCreateWindow (glut_win.obj)
26.944	0.1	2853.493	6.1	233	WriteText() (shadesor.obj)
12.253	0.0	12.253	0.0	1	__glutGetVisualInfo (glut_win.obj)
12.038	0.0	12.038	0.0	5	__glutSetWindow (glut_win.obj)
11.034	0.0	11.565	0.0	1	NewPolygonList() (shadesor.obj)
10.045	0.0	415.929	0.9	2	_processWindowWorkList (glut_event.obj)
8.709	0.0	46294.355	98.6	231	_idleWait (glut_event.obj)
4.185	0.0	46219.506	98.5	230	IdleCallback() (shadesor.obj)
2.992	0.0	4.145	0.0	1	SmoothSOR() (shadesor.obj)
2.772	0.0	65.601	0.1	6	_processEventsAndTimeouts (glut_event.obj)
2.533	0.0	46712.817	99.5	1	_glutMainLoop (glut_event.obj)
1.607	0.0	1.607	0.0	1	__glutOpenWin32Connection (glut_init.obj)

0.936	0.0	0.936	0.0	355 Func() (shadesor.obj)
0.528	0.0	2.248	0.0	1 _glutInit (glut_init.obj)
0.444	0.0	0.444	0.0	2 _glutSetCursor (glut_cursor.obj)
0.349	0.0	0.681	0.0	1 GenerateNormals() (shadesor.obj)
0.332	0.0	0.332	0.0	140 VectorProduct() (shadesor.obj)
0.189	0.0	50.784	0.1	1 _glutCreateWindow (glut_win.obj)
0.151	0.0	0.151	0.0	2 SetViewParams() (shadesor.obj)
0.134	0.0	46944.652	100.0	1 _main (shadesor.obj)
0.112	0.0	0.112	0.0	1 __glutInitTime (glut_init.obj)
0.086	0.0	4.232	0.0	1 NewVertexList() (shadesor.obj)
0.081	0.0	0.081	0.0	1 __glutDetermineMesaSwapHackSupport
0.067	0.0	0.067	0.0	140 SetColor() (shadesor.obj)
0.064	0.0	0.064	0.0	10 __glutGetWindow (glut_win.obj)
0.049	0.0	0.049	0.0	1 _getUnusedWindowSlot (glut_win.obj)
0.010	0.0	12.263	0.0	1 __glutDetermineVisual (glut_win.obj)
0.008	0.0	195.309	0.4	1 ReshapeCallback() (shadesor.obj)
0.007	0.0	0.007	0.0	1 __glutSetupColormap (glut_win.obj)
0.006	0.0	200.264	0.4	1 DisplayCallback() (shadesor.obj)
0.006	0.0	0.008	0.0	1 __glutPostRedisplay (glut_event.obj)
0.004	0.0	0.004	0.0	1 _glutMouseFunc (glut_win.obj)
0.003	0.0	0.003	0.0	1 _glutDisplayFunc (glut_win.obj)
0.003	0.0	0.003	0.0	1 _glutReshapeFunc (glut_win.obj)
0.002	0.0	0.002	0.0	1 __glutPutOnWorkList (glut_event.obj)
0.001	0.0	0.001	0.0	1 _glutIdleFunc (glut_event.obj)
0.001	0.0	0.001	0.0	1 _glutInitDisplayMode (glut_init.obj)
0.001	0.0	0.001	0.0	1 _glutInitWindowSize (glut_init.obj)
0.001	0.0	0.001	0.0	1 _glutInitWindowPosition (glut_init.obj)
0.000	0.0	0.000	0.0	1 _glutKeyboardFunc (glut_win.obj)

6.3 Contents of Computer Disk

The computer disk which comes with this thesis, contains the sourcecode to all the three programs in the MS-DOS file format. The MS Windows program (Fast SOR) can be compiled with MS Visual C++, the OpenGL program (ShadeSOR) can be compiled with either MS Visual C++ or CodeWarrior, and the Unix program (Unix-SOR) can be compiled with Gnu C. An OpenGL library is needed for ShadeSOR, SPHIGS must be installed when compiling the Unix version.

Table 6-1 lists all the files on the disk, along with short descriptions.

Directory	File	Description
FAST-SOR	FAST-SOR.EXE	MS Windows executeable
	FAST-SOR.H	main header file
	FAST-SOR.CPP	main module, Windows independent
	FAST-WIN.CPP	Windows specific routines
	GRAFTOOL.H	graphics library header file
	GRAFTOOL.CPP	graphics library
	FAST-SOR.RC	resource script
	ABOUTPIC.BMP	image file for resource script
	WING.H	Microsoft WinG header file
	WING32.LIB	Microsoft WinG library file
	WING32.DLL	Microsoft WinG dynamic link library
SHADESOR	SHADESOR.EXE	MS Windows executeable
	SHADESOR.CPP	source code
UNIX-SOR	UNIX-SOR.C	source code
	MAKEFILE	makefile
	SPHIGS_BUTTONS.C	buttons library
	SPHIGS_BUTTONS.H	header file for buttons library
	SRGP_BUTTONS.H	header file for buttons library

Table 6-1: Files on Computer Disk

6.4 Source Code

Below is the source code of the main module of the Windows program (fast-sor.cpp), including all the routines that belong to the new algorithm. It comes along with the header file of the project (fast-sor.h). The GT class (Graphics Toolkit), which provides the graphics routines, is located in a separate module (graftool.cpp and graftool.h). Since it does not contain any algorithm specific routines, it is not going to be printed out. Also the module containing the MS Windows specific callback routines (fast-win.cpp) is not listed below.

The OpenGL and SPHIGS versions are also listed below. Each of them consist of one source code file.

6.4.1 Fast SOR Header File

```
1 //*****  
2 // Filename:      FAST-SOR.H  
3 // Project:       Fast Shading of Surfaces of Revolution  
4 //               Master's Thesis Summer Term 1997  
5 // Module:        Main Header File  
6 // Programmer:   Juergen Schulze-Doebold  
7 //               Email: joe@studbox.uni-stuttgart.de  
8 // Advisor:       Prof. A. Hausknecht  
9 // Institution:  University of Massachusetts Dartmouth  
10 // Compiler:     MS Visual C++ 5.0  
11 // Environment:  MS Windows 95/NT  
12 // Requirements: WING32.DLL must be installed  
13 // Project Files: FAST-SOR.CPP, FAST-SOR.H, FAST-WIN.CPP,  
14 //                  FAST-SOR.RC, GRAFTOOL.CPP, GRAFTOOL.H,  
15 //                  RESOURCE.H, WING.H, WING32.LIB
```

```

16 // Last Changes: 12/03/97
17 //*****
18
19 #ifndef _FAST_SOR_H_ // prevent double inclusion
20 #define _FAST_SOR_H_
21
22 //=====
23 // Include Commands
24
25 #include "graftool.h"
26 #include "resource.h"
27
28 //=====
29 // Macros
30
31 #define WDTH 400           // width of parent window
32 #define HGHT 300           // height of parent window
33 #define WDTH_C 256          // width of child window
34 #define HGHT_C 60            // hight of child window
35
36 #define PI 3.141592654    // pi
37 #define KD 0.5             // material constant of SOR
38 #define IP 1.0              // intensity of lightsource
39 #define EPSILON 0.01        // used to determine derivative
40 #define ROTSTEP 2           // autorotation step (degrees)
41 #define DRAGSPEED 2         // pixel per degree
42 #define DEFAULTFPS 10.0     // default value for fps (double)
43 #define DELAYFACT 3          // delay factor for animation slowdown (short)
44
45 // View Window:
46 #define FROM_X (-2.0)       // smallest x value on screen
47 #define TO_X 2.0             // largest x value on screen
48 #define FROM_Y (-1.5)       // smallest y value on screen
49 #define TO_Y 1.5              // largest y value on screen
50 #define FROM_Z (-2.0)       // smallest z value on screen
51 #define TO_Z 2.0              // largest z value on screen
52 #define DX (TO_X-FROM_X)     // predefined macros for frequent computations
53 #define DY (TO_Y-FROM_Y)
54 #define DZ (TO_Z-FROM_Z)
55 #define XMIN (-1.0)           // left clipping plane of SOR
56 #define XMAX 1.0              // right clipping plane of SOR
57 #define XDIF (XMAX-XMIN)      // length of SOR
58
59 // Colors:
60 #define BASECOLOR 16          // first color to be used by the program
61 #define COLPERSHADE 64         // colors per shade
62 #define REDBASE (BASECOLOR)      // 64 shades of red
63 #define GREENBASE (BASECOLOR + COLPERSHADE)    // 64 shades of green
64 #define BLUEBASE (BASECOLOR + 2 * COLPERSHADE) // 64 shades of blue
65 #define BLACK 0
66 #define WHITE (BASECOLOR-1)
67 #define RED (REDBASE + 63)
68 #define GREEN (GREENBASE + 63)
69 #define BLUE (BLUEBASE + 63)
70
71 //=====
72 // Typedef Commands
73
74 typedef struct           // palette entry
75 {

```

```

76     UCHAR r,g,b;
77 } RGBCOLOR;
78
79 typedef struct           // 3D point
80 {
81     double x,y,z;
82 } VECTOR3D;
83
84 typedef struct           // 2D point
85 {
86     short x,y;
87 } VECTOR2D;
88
89 typedef struct
90 {
91     VECTOR3D vertex;      // point in world coordinates
92     VECTOR3D vertexrot;   // point's coordinates after rotation
93     VECTOR2D pixel;       // screen coordinates of projected pixel
94     UCHAR    color;        // pixel color = palette entry
95     VECTOR3D normal;      // 3D normal vector in world coordinates
96     VECTOR3D normalrot;   // 3D normal vector after rotation
97 } VERTEX;
98
99 typedef struct           // description for all polygons used in SOR
100 {
101     UCHAR    location;   // 0=surface polygon, 1=cover disk
102     short    sector;     // sector number for animation, starting with 0
103     short    a,b,c,d;    // clockwise indices into gVertices, representing
104     vertices of quadrilateral
105     VECTOR3D normal;    // normal vector in world coordinates, showing away from
106     SOR
107     VECTOR3D normalrot; // normal vector after rotation
108 } POLYGON;
109
110 //=====
111 // Variable Declarations for Module FAST-SOR.CPP
112
113 extern short    gFunction;
114 extern short    gAngle[3];
115 extern BOOL     gRotation;
116 extern BOOL     gShowPalette;
117 extern short    gCylmode;
118 extern short    gCyls;
119 extern short    gSectors;
120 extern short    gAmbience;
121 extern UCHAR    gSurfBaseColor;
122 extern UCHAR    gDiskBaseColor;
123 extern short    gDisplaySpeed;
124 extern short    gDrawmode;
125 extern RGBCOLOR gPalette[256];
126 extern short    gCylsSave;
127 extern short    gRevolution;
128 extern VERTEX*  gVertices;
129 extern short    gNumVertices;
130 extern POLYGON* gPolygons;
131 extern short    gNumPolygons;
132 extern short    gFillMinX[HGBT];
133 extern short    gFillMaxX[HGBT];
134 extern UCHAR    gFillColl[HGBT];
135 extern UCHAR    gFillColR[HGBT];

```

```
136 extern UCHAR      gColIndex[101];
137 extern double     gSine[361];
138 extern double     gCosine[361];
139 extern short      gSingleStep;
140 extern double     gFps;
141 extern VERTEX     gArrows[12];
142 //=====
143 // Forward Declarations for Module FAST-SOR.CPP
144
145
146 extern void ProcessBoundaryLine(short, short, short, short, short, short);
147 extern void ProcessScanLines(void);
148 extern void ShadeQuadrilateral(const POLYGON*);
149 extern void ShadeQuadConst(const POLYGON*, short);
150 extern double Func(double);
151 extern void ComputeNormal(short, short, short, short, VECTOR3D*);
152 extern void Rotate(const VECTOR3D*, VECTOR3D*);
153 extern void Project(const VECTOR3D*, VECTOR2D*);
154 extern short GetVertexColor(const VECTOR3D*, UCHAR);
155 extern void NewPolygonList(void);
156 extern short World2ScreenX(double);
157 extern short World2ScreenY(double);
158 extern double Screen2WorldX(short);
159 extern void DrawFunctionOutline(void);
160 extern void NewVertexList(void);
161 extern void DrawArrow(short);
162 extern void DrawSurface(void);
163 extern void RedrawSOR(void);
164 extern void IdleCallback(void);
165 extern void InitLookupTables(void);
166 extern void GeneratePalette(void);
167 extern void InitLookupTables(void);
168 //=====
169 // Variable Declarations for Module FAST-WIN.CPP
170
171
172 extern HWND gHwndParent, gHwndChild, gHwndParams;
173 extern int gHvbParent, gHvbChild;
174 extern BOOL gParamWindowActive;
175 //=====
176 // Forward Declarations for Module FAST-WIN.CPP
177
178
179 extern int WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int);
180 extern long WINAPI WinProcParent(HWND,UINT,UINT,LONG);
181 extern long WINAPI WinProcChild(HWND,UINT,UINT,LONG);
182 extern BOOL APIENTRY AboutDlgProc(HWND, UINT, WPARAM, LPARAM);
183 extern BOOL APIENTRY ParamsDlgProc (HWND, UINT, WPARAM, LPARAM);
184 extern void GeneratePalette(void);
185
186 #endif
187 //=====
188 // The End
189 //=====
190 //=====
```

6.4.2 Fast SOR Main Module

```

1 //*****
2 // Filename:      FAST-SOR.CPP
3 // Project:       Fast Shading of Surfaces of Revolution
4 //                Master's Thesis Summer Term 1997
5 // Module:        Core Routines
6 // Programmer:   Juergen Schulze-Doebold
7 //                Email: joe@studbox.uni-stuttgart.de
8 // Advisor:       Prof. A. Hausknecht
9 // Institution:  University of Massachusetts Dartmouth
10 // Compiler:     MS Visual C++ 5.0
11 // Environment:  MS Windows 95/NT
12 // Requirements: WING32.DLL must be installed
13 // Project Files: FAST-SOR.CPP, FAST-SOR.H, FAST-WIN.CPP,
14 //                  FAST-SOR.RC, GRAFTOOL.CPP, GRAFTOOL.H,
15 //                  RESOURCE.H, WING.H, WING32.LIB
16 // Last Changes: 12/04/97
17 //*****
18
19 //=====
20 // Include Commands
21
22 #include "fast-sor.h"
23 #include <stdio.h>
24 #include <math.h>
25 #include <time.h>
26
27 //=====
28 // Variable Definitions
29
30 // presettable variables:
31 short gFunction = 0;           // number of the function used for SOR
32 short gAngle[3] = {0, 0, 0};    // angle in degrees
33 BOOL gRotation = FALSE;       // toggle automatic rotation
34 BOOL gShowPalette = FALSE;    // palette display
35 short gCylmode = 0;           // cylinders display: 0=frustums, 1=centered,
36 2=minimum
37 short gCyls    = 10;          // number of cylinders in x direction
38 short gSectors = 7;           // number of vertices for each disc
39 short gAmbience = 5;          // minimum color index
40 UCHAR gSurfBaseColor = GREENBASE; // surface color
41 UCHAR gDiskBaseColor = BLUEBASE; // base color for disks
42 short gDisplaySpeed = 100;    // range: 0..100 (100=fastest)
43 short gDrawmode = 0;          // drawmode: 0=solid, 1=wireframe, 2=graph
44
45 // dependent variables:
46 RGBCOLOR gPalette[256];      // palette entries (RGB)
47 short gCylsSave = gCyls;      // cylinder animation; if gCyls<gCylsSave
48 then animation is active
49 short gRevolution = gSectors; // revolution animation; if <gSectors
50 animation is active
51 VERTEX* gVertices = NULL;
52 short gNumVertices;          // number of vertices

```

```

53 POLYGON* gPolygons = NULL;
54 short   gNumPolygons;           // number of polygons
55 short   gFillMinX[HGT];       // minimum x values of to-be-filled quadrilateral
56 short   gFillMaxX[HGT];       // maximum x values -"-"
57 UCHAR   gFillColL[HGT];       // fill color for left edge
58 UCHAR   gFillColR[HGT];       // fill color for right edge
59 UCHAR   gColIndex[101];        // lookup table for color indices
60 double  gSine[361];          // sine table for 0-360 degrees
61 double  gCosine[361];         // cosine table for 0-360 degrees
62 short   gSingleStep = 0;       // execute single animation step(s): 0=off, 1=on,
63 n=execute n-1 steps
64 double  gFps;                // frames per second
65 VERTEX  gArrows[12] =         // 3 coordinate axis arrows, 4 vertices each
66 { {-1.4, 0.0, 0.0}, {1.4,0.0,0.0}, { 1.3,0.1,0.0}, {1.3,-0.1,0.0},    // x axis
67 arrow
68 { 0.0,-1.4, 0.0}, {0.0,1.4,0.0}, {-0.1,1.3,0.0}, {0.1, 1.3,0.0},    // y axis
69 arrow
70 { 0.0, 0.0,-1.4}, {0.0,0.0,1.4}, { 0.0,0.1,1.3}, {0.0,-0.1,1.3} }; // z axis
71 arrow
72
73 //=====
74 // Function Definitions
75
76 -----
77 void ProcessBoundaryLine(short ax, short ay, short ac, short bx, short by,
78 short bc)
79 // computes the x and y values of one single line between the
80 // points ax/ay and bx/by, as well as the colors along the line
81 {
82     short y1, y2;           // y counters starting on top and on bottom
83     short x1, x2, dx;       // x counters starting on top and bottom, and stepsize,
84     unit: 1/64 (fixpoint arithmetic)
85     short xs;              // speedup for x
86     short c;                // current color value, unit: 1/128 (fixed-point
87     arithmetic)
88     short dc;              // color step per pixel, unit: 1/128 (fixed-point
89     arithmetic)
90     short h;                // dummy for swapping
91
92     if (ay > by) // make sure that vertex 1 is on top of vertex 2
93     {
94         h = ax; ax = bx; bx = h;
95         h = ay; ay = by; by = h;
96         h = ac; ac = bc; bc = h;
97     }
98
99     y1 = ay;
100    y2 = by;
101    x1 = ax;
102    x2 = bx;
103    c = ac;
104
105    x1 <= 6;
106    x2 <= 6;
107
108    if (y2 - y1 != 0) // division by zero?
109    {
110        dc = ((bc - c) << 7) / (y2 - y1);
111        dx = (x2 - x1) / (y2 - y1);
112    }

```

```

113     else
114         dc = dx = 0;
115
116     c <= 7;
117
118     do
119     {
120         xs = x1 >> 6;
121         if (xs < gFillMinX[y1])    gFillMinX[y1] = xs, gFillColL[y1] = c >> 7;
122         if (xs > gFillMaxX[y1])   gFillMaxX[y1] = xs, gFillColR[y1] = c >> 7;
123         x1 += dx;
124         c += dc;
125         ++y1;
126     } while(y1 <= y2);
127 }
128
129 //-----
130 void ProcessScanLines(void)
131 // uses the arrays of 'ProcessBoundaryLine' to fill a quadrilateral with
132 // interpolated colors
133 {
134     short x,y;      // pixel coordinates on screen
135     short c;        // current color value, unit: 1/128 (fixpoint arithmetic)
136     short dc;       // color step per pixel, unit: 1/128 (fixpoint arithmetic)
137     int vbw;        // VB-width
138     UCHAR* ptr;    // current pixel pointer
139     UCHAR* base;   // pointer to beginning of current line
140
141     vbw = GT.GetWidth(gHvbParent);
142     base = GT.GetAddress(gHvbParent);
143
144     for (y=0; y<HGBT; ++y, base += vbw)
145     {
146         if (gFillMaxX[y]==-1) continue;
147         x = gFillMinX[y];
148         c = gFillColL[y];
149         if (gFillMaxX[y] - x != 0)
150             dc = ((short)gFillColR[y] - c) * 128 / ((short)gFillMaxX[y] - x);
151         else dc = 0;
152         c <= 7; // c *= 128
153         ptr = base + x; // compute current pixel pointer
154         do
155         {
156             *ptr = c >> 7; // draw pixel
157             ++ptr;           // move pointer to next pixel
158             c += dc;         // add fraction of color
159             ++x;             // increase pixel counter
160         } while (x<=gFillMaxX[y]);
161     }
162 }
163
164 //-----
165 void ShadeQuadrilateral(const POLYGON* poly)
166 // Only use with surface quadrilaterals!
167 // display a filled quadrilateral, parameter is pointer to POLYGON
168 // quadrilateral edges are: ab, bc, cd, da
169 {
170     struct PIXEL // pixel databuffer
171     {
172         short x,y; // screen coordinates in pixels

```

```

173     short c;      // pixel color
174 } a,b,c,d;      // one struct for each vertex
175 short va, vb, vc, vd; // vertex vectors
176
177 va = poly->a;
178 vb = poly->b;
179 vc = poly->c;
180 vd = poly->d;
181
182 // initialize arrays:
183 memset(gFillMinX, 0x7F, sizeof(short) * HGBT);
184 memset(gFillMaxX, 0xFF, sizeof(short) * HGBT);
185
186 a.x = gVertices[va].pixel.x;
187 a.y = gVertices[va].pixel.y;
188 a.c = (short)gVertices[va].color;
189 b.x = gVertices[vb].pixel.x;
190 b.y = gVertices[vb].pixel.y;
191 b.c = (short)gVertices[vb].color;
192 c.x = gVertices[vc].pixel.x;
193 c.y = gVertices[vc].pixel.y;
194 c.c = (short)gVertices[vc].color;
195 d.x = gVertices[vd].pixel.x;
196 d.y = gVertices[vd].pixel.y;
197 d.c = (short)gVertices[vd].color;
198
199 if (gDrawmode==1)
200 {
201     GT.Line(a.x, a.y, b.x, b.y, BLACK);
202     GT.Line(b.x, b.y, c.x, c.y, BLACK);
203     GT.Line(c.x, c.y, d.x, d.y, BLACK);
204     GT.Line(d.x, d.y, a.x, a.y, BLACK);
205 }
206 else
207 {
208     ProcessBoundaryLine(a.x, a.y, a.c, b.x, b.y, b.c);
209     ProcessBoundaryLine(b.x, b.y, b.c, c.x, c.y, c.c);
210     ProcessBoundaryLine(c.x, c.y, c.c, d.x, d.y, d.c);
211     ProcessBoundaryLine(d.x, d.y, d.c, a.x, a.y, a.c);
212     ProcessScanLines();
213 }
214 }
215
216 -----
217 void ShadeQuadConst(const POLYGON* poly, short color)
218 // To be used with disk faces, uses a single color for shading
219 // display a filled quadrilateral, parameter is pointer to POLYGON
220 // quadrilateral edges are: ab, bc, cd, da
221 {
222     struct PIXEL // pixel databuffer
223     {
224         short x,y; // screen coordinates in pixels
225     } a,b,c,d; // one struct for each vertex
226     short va, vb, vc, vd; // vertex vectors
227
228 va = poly->a;
229 vb = poly->b;
230 vc = poly->c;
231 vd = poly->d;
232

```

```

233 // initialize arrays:
234 memset(gFillMinX, 0x7F, sizeof(short) * HGBT);
235 memset(gFillMaxX, 0xFF, sizeof(short) * HGBT);
236
237 a.x = gVertices[va].pixel.x;
238 a.y = gVertices[va].pixel.y;
239 b.x = gVertices[vb].pixel.x;
240 b.y = gVertices[vb].pixel.y;
241 c.x = gVertices[vc].pixel.x;
242 c.y = gVertices[vc].pixel.y;
243 d.x = gVertices[vd].pixel.x;
244 d.y = gVertices[vd].pixel.y;
245
246 if (gDrawmode==1)
247 {
248     GT.Line(a.x, a.y, b.x, b.y, BLACK);
249     GT.Line(b.x, b.y, c.x, c.y, BLACK);
250     GT.Line(c.x, c.y, d.x, d.y, BLACK);
251     GT.Line(d.x, d.y, a.x, a.y, BLACK);
252 }
253 else
254 {
255     ProcessBoundaryLine(a.x, a.y, color, b.x, b.y, color);
256     ProcessBoundaryLine(b.x, b.y, color, c.x, c.y, color);
257     ProcessBoundaryLine(c.x, c.y, color, d.x, d.y, color);
258     ProcessBoundaryLine(d.x, d.y, color, a.x, a.y, color);
259     ProcessScanLines();
260 }
261
262
263 //-----
264 double Func(double x)
265 // returns the function values
266 {
267     switch (gFunction)
268     {
269         case 0: return(fabs(x));
270         case 1: return(x * x);
271         case 2: return((x>=-1.0 && x<=1.0) ? sqrt(fabs(1.0-x*x)) : -2.0*x*x+2.0);
272         case 3: return(sqrt(fabs(x)));
273         default: return(x);
274     }
275 }
276
277 //-----
278 void ComputeNormal(short a, short b, short c, short d, VECTOR3D* n)
279 // important: points a,b,c,d must be in clockwise order, and they are indices
280 // in gVertices[]
281 // computes the normal vector to a triple of unequal vertices out of a,b,c,c
282 // by using the cross product.
283 // the result is returned in n.
284 // the returned normal is not necessarily normalized!
285 {
286     VECTOR3D *p1,*p2,*p3,*p4;
287     VECTOR3D v1,v2;
288
289     p1 = &gVertices[a].vertex;
290     p2 = &gVertices[b].vertex;
291     p3 = &gVertices[c].vertex;
292     p4 = &gVertices[d].vertex;

```

```

293
294     if (p1->x==p2->x && p1->y==p2->y && p1->z==p2->z) // a and b equal?
295         p1 = p4;
296
297     v1.x = p1->x - p2->x;
298     v1.y = p1->y - p2->y;
299     v1.z = p1->z - p2->z;
300
301     v2.x = p3->x - p2->x;
302     v2.y = p3->y - p2->y;
303     v2.z = p3->z - p2->z;
304
305     n->x = v1.y * v2.z - v1.z * v2.y;
306     n->y = v1.z * v2.x - v1.x * v2.z;
307     n->z = v1.x * v2.y - v1.y * v2.x;
308 }
309
310 //-----
311 void Rotate(const VECTOR3D* in, VECTOR3D* out)
312 // rotates a point from 'in' to 'out'
313 // uses the current values of the 'gAngle[]' array
314 // this routine first checks for which parameters have to be
315 // recomputed, and only recomputes these
316 // the actual rotation is done by the rotation matrix 'rm[][]'
317 {
318     static short angle[3] = {-1,-1,-1}; // saves current values of gAngle
319     static double rm[3][3]; // rotation matrix
320     static double s[3]; // precomputed sine value for all angles
321     static double c[3]; // precomputed cosine values for all angles
322     UCHAR recompute=0; // info about recomputation of rotation matrix
323     elements
324     short i; // loop counter
325
326     // if angles have changed recompute dependent sines and cosines:
327
328     for (i=0; i<3; ++i)
329     {
330         if (angle[i] != gAngle[i])
331         {
332             angle[i] = gAngle[i];
333             s[i] = gSine[angle[i]];
334             c[i] = gCosine[angle[i]];
335             recompute |= 1 << i;
336         }
337     }
338
339     // now the three least significant bits of 'recompute' give
340     // information about the angles that have changed
341
342     if (recompute != 0) // check for any angle
343     {
344         rm[0][1] = s[0] * s[1] * c[2] - c[0] * s[2];
345         rm[0][2] = c[0] * s[1] * c[2] + s[0] * s[2];
346         rm[1][1] = s[0] * s[1] * s[2] + c[0] * c[2];
347         rm[1][2] = c[0] * s[1] * s[2] - s[0] * c[2];
348
349         if ((recompute & 3) != 0) // check for alpha and beta
350         {
351             rm[2][0] = -s[1];
352             rm[2][1] = s[0] * c[1];

```

```

353     rm[2][2] = c[0] * c[1];
354 }
355
356 if ((recompute & 6) != 0) // check for beta and gamma
357 {
358     rm[0][0] = c[1] * c[2];
359     rm[1][0] = c[1] * s[2];
360 }
361 }
362
363 // multiply rotation matrix (RM) with vector 'in':
364 // | 00 01 02 | / x \ / 00x + 01y + 02z \
365 // | 10 11 12 | x | y | = | 10x + 11y + 12z |
366 // | 20 21 22 | \ z / \ 20x + 21y + 22z /
367
368 out->x = in->x * rm[0][0] + in->y * rm[0][1] + in->z * rm[0][2];
369 out->y = in->x * rm[1][0] + in->y * rm[1][1] + in->z * rm[1][2];
370 out->z = in->x * rm[2][0] + in->y * rm[2][1] + in->z * rm[2][2];
371 }
372
373 //-----
374 void Project(const VECTOR3D* in, VECTOR2D* out)
375 // project the input point to the 2d screen
376 {
377     static double wdth_div_dx = (double)WDTH / (double)DX;
378     static double hght_div_dy = (double)HHT / (double)DY;
379
380     out->x = (short)((in->x - FROM_X) * wdth_div_dx);
381     out->y = HHT - (short)((in->y - FROM_Y) * hght_div_dy);
382 }
383
384 //-----
385 // computes the color to a certain normal vector.
386 // the resulting vertex color value is returned
387 short GetVertexColor(const VECTOR3D* normal, UCHAR colorbase)
388 {
389     short index;
390     UCHAR color;
391
392     index = (short)(normal->z * 100.0); // only z coordinate decides color
393     if (index<0) index = 0; // index must be at least zero
394
395     color = gColIndex[index];
396
397     color += (UCHAR)gAmbience; // add ambient color
398     if (color > 63) // limit color values
399         color = 63;
400
401     return(color + colorbase);
402 }
403
404 //-----
405 void NewPolygonList(void)
406 {
407     short index; // index for polygons
408     short sect, cyl; // loop counters
409     short base; // base index for enumeration of vertices
410     short side; // 0=left/surface, 1=right/cylinder cover
411
412     // find number of polygons and allocate memory:

```

```

413     if (gPolygons != NULL) free(gPolygons);
414     if (gCylmode==0) // mode = frustums
415         gNumPolygons = (gCyls + 2) * gSectors; // + 2 for cylinder covers
416     else
417         gNumPolygons = (2 * gCyls + 1) * gSectors;
418     gPolygons = (POLYGON*)malloc(gNumPolygons * sizeof(POLYGON));
419     if (gPolygons==NULL) exit(-1); // memory allocation error?
420
421     // create left and right cylinder covers:
422     for (side=0; side<2; ++side) // 0=left, 1=right
423     {
424         base = (side==0) ? 1 : gNumVertices - 1 - gSectors;
425         index = (side==0) ? 0 : gNumPolygons - gSectors;
426
427         for (sect=0; sect<gSectors; ++sect, ++index)
428         {
429             gPolygons[index].location = 1;
430             gPolygons[index].sector = sect;
431             gPolygons[index].a = base + sect;
432             gPolygons[index].b = base + ((sect + 1) % gSectors);
433             gPolygons[index].c = gPolygons[index].d = (side==0) ? 0 : gNumVertices -
434             1;
435             gPolygons[index].normal.x = (side==0) ? -1.0 : 1.0;
436             gPolygons[index].normal.y = gPolygons[index].normal.z = 0.0;
437         }
438     }
439
440     // create the other polygons:
441     index = gSectors; // indices before are used for left cover disk
442     base = 1; // 0 is reserved for end cover disk
443     for (cyl=0; cyl<gCyls; ++cyl) // loop through all cylinders
444     {
445         for (side=0; side<((gCylmode==0) ? 1 : 2); ++side) // for surface and
446         disk part of each cylinder
447         {
448             if (side==0 || cyl != gCyls-1) // once only for last cylinder
449                 for (sect=0; sect<gSectors; ++sect, ++index)
450                 {
451                     gPolygons[index].location = (UCHAR)side;
452                     gPolygons[index].sector = sect;
453                     gPolygons[index].a = base + sect;
454                     gPolygons[index].b = base + ((sect + 1) % gSectors);
455                     gPolygons[index].c = gPolygons[index].b + gSectors;
456                     gPolygons[index].d = gPolygons[index].a + gSectors;
457                     ComputeNormal(gPolygons[index].d,
458                                     gPolygons[index].c,
459                                     gPolygons[index].b,
460                                     gPolygons[index].a,
461                                     &gPolygons[index].normal);
462                     base += gSectors;
463                 }
464             }
465
466     //-----
467     short World2ScreenX(double x)
468     // convert world x coordinates into screen x coordinates
469     {
470         return((short)((x - FROM_X) * (double)WDTH / (double)DX));
471     }
472

```

```

473 //-----
474 short World2ScreenY(double y)
475 // convert world x coordinates into screen x coordinates
476 {
477     return(HGHT - (short)((y - FROM_Y) * (double)HGHT / (double)DY));
478 }
479 //-----
480 double Screen2WorldX(short x)
481 // convert screen x coordinates into world x coordinates
482 {
483     return((double)x * (double)DX / (double)WDTH + FROM_X);
484 }
485 //-----
486 void GenerateFunctionOutline(void)
487 // draws the function outline
488 {
489     short strips;           // number of strips in x direction
490     short index,i,strip;   // loop indices
491     double height;         // strip height
492     double xstep, xpos;
493     short side, base;
494
495     strips = (World2ScreenX(XMAX) - World2ScreenX(XMIN) + 1) / 2;
496
497     // find number of vertices and allocate memory:
498     if (gVertices!=NULL) free(gVertices);
499     gNumVertices = (strips + 1) * 2 + 1;
500     gVertices = (VERTEX*)malloc(gNumVertices * sizeof(VERTEX));
501     if (gVertices==NULL) exit(-1); // memory allocation error?
502
503     // create the vertices:
504
505     xstep = XDIF / (double)strips;
506     xpos = XMIN;           // initialize
507     index = 0;
508     for (strip=0; strip<strips+1; ++strip, xpos += xstep) // loop through all
509     strips
510     {
511         height = fabs(Func(xpos));
512
513         for (i=0; i<2; ++i) // points at bottom and top of strip
514         {
515             // compute vertex coordinates:
516             gVertices[index].vertex.x = xpos;
517             gVertices[index].vertex.y = (i==0) ? 0.0 : height;
518             gVertices[index].vertex.z = 0.0;
519             gVertices[index].normal.x = 0.0;
520             gVertices[index].normal.y = 0.0;
521             gVertices[index].normal.z = 1.0;
522             ++index;
523         }
524     }
525
526     // last point is special:
527     gVertices[index].vertex.x = xpos;
528     gVertices[index].vertex.y = gVertices[index].vertex.z = 0.0;
529     gVertices[index].normal.x = gVertices[index].normal.y = 0.0;
530     gVertices[index].normal.z = -1.0;
531
532 }
```

```

533
534     // create the polygons:
535
536     // find number of polygons and allocate memory:
537     if (gPolygons != NULL) free(gPolygons);
538     gNumPolygons = strips * 2;
539     gPolygons = (POLYGON*)malloc(gNumPolygons * sizeof(POLYGON));
540     if (gPolygons==NULL) exit(-1); // memory allocation error?
541
542     // create polygons:
543     for (strip=index=base=0; strip<strips; ++strip, base+=2) // loop through
544     all strips
545     {
546         for (side=0; side<2; ++side, ++index) // front and back
547         {
548             gPolygons[index].location = 1;
549             gPolygons[index].sector = 0;
550             gPolygons[index].a = base;
551             gPolygons[index].b = base + 2;
552             gPolygons[index].c = base + 3;
553             gPolygons[index].d = base + 1;
554             gPolygons[index].normal.x = gPolygons[index].normal.y = 0.0;
555             gPolygons[index].normal.z = (side==0) ? 1.0 : -1.0;
556         }
557     }
558 }
559
560 //-----
561 void DrawFunctionOutline(void)
562 // draws the function outline
563 // this works only correctly if all angles are 0!
564 {
565     short x1, x2; // start and end pixel coordinates
566     short x,y; // current x and y positions
567     short prev_y; // previous y value
568
569     x1 = World2ScreenX(XMIN);
570     x2 = World2ScreenX(XMAX);
571
572     for (x=x1; x<=x2; ++x) // for all pixel x coordinates
573     {
574         y = World2ScreenY(Func(Screen2WorldX(x)));
575         if (x>x1) // no line possible at first x position
576             GT.Line(x-1, prev_y, x, y, BLUE);
577         prev_y = y;
578     }
579 }
580
581 //-----
582 void NewVertexList(void)
583 // creates the list of vertices
584 // order of vertices:
585 // [0] = midpoint of left cover disk
586 // [1] to [gNumVertices-2] = surface vertices from left to right
587 // [gNumVertices-1] = midpoint of right cover disk
588 {
589     short cyl, sect; // loop counters
590     short side; // 0=left, 1=right
591     double xpos, angle; // current vertex position in world coordinates
592     double xstep, anglestep; // stepsizes

```

```

593     short index;           // current vertex array index
594     double deriv;          // first derivative
595     double divider;        // shortcut for speed
596     double height;         // column height
597
598     if (gDrawmode==2)           // use different routine for outline
599     {
600         GenerateFunctionOutline();
601         return;
602     }
603
604     // find number of vertices and allocate memory:
605     if (gVertices!=NULL) free(gVertices);
606     if (gCylmode==0) // mode = frustums?
607         gNumVertices = (gCyls + 1) * gSectors + 2; // + 2 for left and right end
608     covers
609     else // with columns:
610         gNumVertices = 2 * gCyls * gSectors + 2; // + 2 for left and right end
611     covers
612     gVertices = (VERTEX*)malloc(gNumVertices * sizeof(VERTEX));
613     if (gVertices==NULL) exit(-1); // memory allocation error?
614
615     xstep = XDIF / (double)gCyls;
616     anglestep = 2.0 * PI / (double)gSectors; // step in radians
617
618     xpos = XMIN; // initialize
619
620     // create midpoints for left and right end cover:
621     for (side=0; side<2; ++side) // 0=left, 1=right
622     {
623         index = (side==0) ? 0 : gNumVertices - 1;
624         gVertices[index].vertex.x = (side==0) ? XMIN : XMIN + gCyls * xstep;
625         gVertices[index].vertex.y = 0.0;
626         gVertices[index].vertex.z = 0.0;
627         gVertices[index].normal.x = (side==0) ? -1.0 : 1.0;
628         gVertices[index].normal.y = 0.0;
629         gVertices[index].normal.z = 0.0;
630     }
631
632     index = 1;
633     // create all vertices:
634     for (cyl=0; cyl < gCyls + ((gCylmode==0) ? 1 : 0); ++cyl) // loop through
635     all cylinders
636     {
637         switch (gCylmode) // different height computations
638         {
639             case 0: height = fabs(Func(xpos)); break; // frustums
640             case 1: height = (fabs(Func(xpos)) + fabs(Func(xpos+xstep))) / 2.0;
641         break; // centered
642             case 2: height = min(fabs(Func(xpos)), fabs(Func(xpos+xstep))); break;
643         // minimum
644         }
645
646         for (side=0; side < ((gCylmode==0) ? 1 : 2); ++side) // for left and
647         right side of each cylinder
648         {
649             for (angle=sect=0; sect<gSectors; ++sect, ++index, angle+=anglestep)
650             {
651                 // compute vertex coordinates:
652                 gVertices[index].vertex.x = xpos;

```

```

653     gVertices[index].vertex.y = height * cos(angle);
654     gVertices[index].vertex.z = height * sin(angle);
655
656     // compute vertex normals:
657     deriv = (Func(xpos + EPSILON) - Func(xpos)) / EPSILON;
658     divider = sqrt(deriv * deriv + 1.0);
659     gVertices[index].normal.x = - deriv / divider;
660     gVertices[index].normal.y = cos(angle) / divider;
661     gVertices[index].normal.z = sin(angle) / divider;
662 }
663 if (side==0 || gCylmode==0) xpos += xstep; // move on to next x
664 position
665 }
666 }
667
668 NewPolygonList(); // _must_ be done now
669 }
670
671 //-----
672 void DrawArrow(short type)
673 // draw axis arrows. parameters:
674 // 0=left x arrow, 1=middle x arrow, 2=right x arrow, 3=y arrow, 4=z arrow
675 {
676     short x1,y1,x2,y2; // line end points
677
678     switch(type)
679     {
680         case 0: // left part of arrow pointing into x direction
681             x1 = gArrows[0].pixel.x;
682             y1 = gArrows[0].pixel.y;
683             x2 = gVertices[0].pixel.x;
684             y2 = gVertices[0].pixel.y;
685             GT.Line(x1,y1,x2,y2,RED);
686             break;
687
688         case 1: // middle part of arrow pointing into x direction
689             x1 = gVertices[0].pixel.x;
690             y1 = gVertices[0].pixel.y;
691             x2 = gVertices[gNumVertices-1].pixel.x;
692             y2 = gVertices[gNumVertices-1].pixel.y;
693             GT.Line(x1,y1,x2,y2,RED);
694             break;
695
696         case 2: // right part of arrow pointing into x direction
697             x1 = gVertices[gNumVertices-1].pixel.x;
698             y1 = gVertices[gNumVertices-1].pixel.y;
699             x2 = gArrows[1].pixel.x;
700             y2 = gArrows[1].pixel.y;
701             GT.Line(x1,y1,x2,y2,RED); // shaft
702             x1 = gArrows[1].pixel.x;
703             y1 = gArrows[1].pixel.y;
704             x2 = gArrows[2].pixel.x;
705             y2 = gArrows[2].pixel.y;
706             GT.Line(x1,y1,x2,y2,RED); // top arrow line
707             x1 = gArrows[1].pixel.x;
708             y1 = gArrows[1].pixel.y;
709             x2 = gArrows[3].pixel.x;
710             y2 = gArrows[3].pixel.y;
711             GT.Line(x1,y1,x2,y2,RED); // bottom arrow line
712             break;

```

```

713
714     case 3: // arrow pointing into y direction
715         x1 = gArrows[4].pixel.x;
716         y1 = gArrows[4].pixel.y;
717         x2 = gArrows[5].pixel.x;
718         y2 = gArrows[5].pixel.y;
719         GT.Line(x1,y1,x2,y2,GREEN); // shaft
720         x1 = gArrows[5].pixel.x;
721         y1 = gArrows[5].pixel.y;
722         x2 = gArrows[6].pixel.x;
723         y2 = gArrows[6].pixel.y;
724         GT.Line(x1,y1,x2,y2,GREEN); // left arrow line
725         x1 = gArrows[5].pixel.x;
726         y1 = gArrows[5].pixel.y;
727         x2 = gArrows[7].pixel.x;
728         y2 = gArrows[7].pixel.y;
729         GT.Line(x1,y1,x2,y2,GREEN); // right arrow line
730         break;
731
732     case 4: // arrow pointing into z direction
733         x1 = gArrows[8].pixel.x;
734         y1 = gArrows[8].pixel.y;
735         x2 = gArrows[9].pixel.x;
736         y2 = gArrows[9].pixel.y;
737         GT.Line(x1,y1,x2,y2,BLUE); // shaft
738         x1 = gArrows[9].pixel.x;
739         y1 = gArrows[9].pixel.y;
740         x2 = gArrows[10].pixel.x;
741         y2 = gArrows[10].pixel.y;
742         GT.Line(x1,y1,x2,y2,BLUE); // top arrow line
743         x1 = gArrows[9].pixel.x;
744         y1 = gArrows[9].pixel.y;
745         x2 = gArrows[11].pixel.x;
746         y2 = gArrows[11].pixel.y;
747         GT.Line(x1,y1,x2,y2,BLUE); // bottom arrow line
748         break;
749
750     default: break;
751 }
752 }
753
754 //-----
755 void DrawSurface(void)
756 {
757     short i;           // loop counter
758     short index;       // loop counter, self adjusted
759     short step;        // stepsize for polygon counter
760     short diskcolor;  // color for cover disks
761
762     // compute dynamic vertex values for current orientation:
763     for (i=0; i<gNumVertices; ++i)
764     {
765         Rotate(&gVertices[i].vertex, &gVertices[i].vertexrot);
766         Rotate(&gVertices[i].normal, &gVertices[i].normalrot);
767         Project(&gVertices[i].vertexrot, &gVertices[i].pixel);
768         gVertices[i].color      = (UCHAR)GetVertexColor(&gVertices[i].normalrot,
769 gSurfBaseColor);
770     }
771
772     // compute current coordinates for arrows:

```

```

773     for (i=0; i<12; ++i)
774     {
775         Rotate(&gArrows[i].vertex, &gArrows[i].vertexrot);
776         Project(&gArrows[i].vertexrot, &gArrows[i].pixel);
777     }
778
779     // compute dynamic polygon values:
780     for (i=0; i<gNumPolygons; ++i)
781         Rotate(&gPolygons[i].normal, &gPolygons[i].normalrot);
782
783     // draw surface polygons:
784     if (gPolygons[0].normalrot.z <= 0.0) // find order for painter's algorithm
785     using left cover disk
786     {
787         index = 0;
788         step = 1;
789         diskcolor = GetVertexColor(&gVertices[gNumVertices-1].normalrot,
790 gDiskBaseColor);
791     }
792     else
793     {
794         index = gNumPolygons-1;
795         step = -1;
796         diskcolor = GetVertexColor(&gVertices[0].normalrot, gDiskBaseColor);
797     }
798
799     // draw back part of x axis:
800     if (step==1) DrawArrow(0);
801     else DrawArrow(2);
802
803     // draw middle part of x axis:
804     DrawArrow(1);
805
806     while (index>=0 && index<gNumPolygons) // draw all polygons
807     {
808         if (index==(gNumPolygons>>1)) // draw y and z arrows halfway through
809         polygons
810         {
811             DrawArrow(3);
812             DrawArrow(4);
813         }
814         if (gPolygons[index].sector < gRevolution) // is polygon to be shown in
815         current animation phase?
816         {
817             if (gDrawmode==1 || gRevolution<gSectors || gPolygons[index].normalrot.z
818 > 0.0) // is polygon visible?
819             {
820                 if (gPolygons[index].location == 0) // draw surface polygons
821                     ShadeQuadrilateral(&gPolygons[index]);
822                 else
823                     ShadeQuadConst(&gPolygons[index], diskcolor);
824             }
825         }
826         index += step;
827     }
828
829     // draw front part of x axis:
830     if (step==1) DrawArrow(2);
831     else DrawArrow(0);
832

```

```

833     // draw function outline if in original position:
834     if (gRotation==FALSE && gAngle[0]==0.0 && gAngle[1]==0.0 && gAngle[2]==0.0)
835         DrawFunctionOutline();
836     }
837
838 //-----
839 void RedrawSOR(void)
840 {
841     clock_t start,end,diff;
842     char buf[20];
843
844     start = clock();
845     GT.Set(gHvbParent);
846     GT.Clear(WHITE); // erase previous image
847     DrawSurface();
848     GT.Paste(0,0,WDTH,HGHT,0,0,gHvbParent,gHwndParent); // display GT
849     if (gDisplaySpeed<100 && gDisplaySpeed>0) // delay
850         Sleep((100-gDisplaySpeed)*DELAYFACT);
851     end = clock();
852     diff = end - start;
853     if (diff == 0) diff = 1; // division by zero?
854     gFps = (double)CLOCKS_PER_SEC / (double)diff;
855     if (gParamWindowActive) // continue only if params window is active
856     {
857         sprintf(buf, "%6.1f", gFps);
858         SetDlgItemText(gHwndParams, IDC_FPS, buf);
859     }
860 }
861
862 //-----
863 void IdleCallback(void)
864 // function to be called during idle time
865 {
866     short i;
867     BOOL redraw=FALSE;
868     BOOL recompute=FALSE;
869
870     if (gSingleStep==1) return;
871     if (gSingleStep>1) --gSingleStep;
872     if (gRotation) // rotation about axes
873     {
874         for (i=1; i<=2; ++i)
875         {
876             gAngle[i] += ROTSTEP;
877             if (gAngle[i] >= 360) gAngle[i] -= 360;
878         }
879         redraw = TRUE;
880     }
881     if (gRevolution < gSectors) // sector animation
882     {
883         ++gRevolution;
884         redraw = TRUE;
885     }
886     if (gCyls < gCylsSave)
887     {
888         ++gCyls;
889         recompute = redraw = TRUE;
890     }
891     if (recompute) NewVertexList();
892     if (redraw) RedrawSOR();

```

```
893     }
894 //-----
895 void InitLookupTables(void)
896 {
897     short i;
898
899     for (i=0; i<101; ++i)
900         gColIndex[i] = (UCHAR)((double)COLPERSHADE * acos(1.0 - (double)i/100.0) /
901 (PI/2.0));
902
903     for (i=0; i<=360; ++i)
904     {
905         gSine[i]   = sin(i * PI / 180.0);
906         gCosine[i] = cos(i * PI / 180.0);
907     }
908 }
909
910 //=====
911 // The End
912 //=====
913
```

6.4.3 The OpenGL Version

```

1 //*****
2 // Filename:      SHADESOR.CPP
3 // Project:       Shading of Surfaces of Revolution with OpenGL
4 //                Master's Thesis Summer Term 1997
5 // Programmer:    Juergen Schulze-Doebold
6 //                Email: joe@studbox.uni-stuttgart.de
7 // Advisor:       Prof. A. Hausknecht
8 // Institution:   University of Massachusetts Dartmouth
9 // Compilers:     MS Visual C++ 5.0 (PC) or Code Warrior (Apple)
10 // Environment:  MS Windows 95/NT or Apple Finder
11 // Requirements: OpenGL must be installed
12 // Project Files: PC: SHADESOR.CPP, GLU32.LIB, GLUT.LIB,
13 //                  OPENGL32.LIB
14 // Last Changes:  12/01/97
15 //*****
16
17 #ifdef WIN32
18     #include <windows.h>
19     #include <conio.h>
20     #include <time.h>
21 #endif
22 #include <stdlib.h>
23 #include <stdio.h>
24 #include <stdarg.h>
25 #include <string.h>
26 #include <math.h>
27 #include <memory.h>
28 #ifdef WIN32
29     #include <gl\gl.h>
30     #include <gl\glu.h>
31     #include <gl\glut.h>
32 #else           // for use with the Macintosh
33     #include "gl.h"
34     #include "glu.h"
35     #include "glut.h"
36 #endif
37
38 //=====
39 // Constants Definitions
40 //=====
41
42 #undef BENCHMARK      // switch off buttons for benchmarking
43
44 #define WDTH 600        // window size
45 #define HGHT 400
46
47 #define BUTTONS 18       // number of clickable buttons
48 #define FUNCNUM 4         // number of different functions
49 #define CYLNUM 3          // number of different cylinder display modes
50
51 #define MIN(x,y) ( (y<x) ? (y) : (x) )
52
53 #ifndef WIN32

```

```

54     #define TRUE    (0==0)
55     #define FALSE   (!TRUE)
56 #endif
57
58 #define BLUE     0x00FF0000
59 #define GREEN    0x0000FF00
60 #define RED      0x000000FF
61 #define BLACK    0x00000000
62 #define WHITE    0x00FFFFFF
63 #define MAGENTA  0x00FFFF00
64 #define YELLOW   0x0000FFFF
65 #define PINK     0x00FF00FF
66
67 #define PI 3.141592654 // pi
68 #define SOR_LIST 1      // display list entry for surface of revolution (0
69 doesn't work)
70
71 // View Window:
72 #define FROM_X (-2.0)    // smallest x value
73 #define TO_X     4.0       // largest x value
74 #define FROM_Y (-2.0)    // smallest y value
75 #define TO_Y     2.0       // largest y value
76 #define FROM_Z (-3.0)    // smallest z value
77 #define TO_Z     3.0       // largest z value
78 #define DX (TO_X-FROM_X) // predefined macros for frequent occurences
79 #define DY (TO_Y-FROM_Y)
80 #define DZ (TO_Z-FROM_Z)
81
82 // SOR parameters:
83 #define XMIN (-1.0)      // left clipping plane of SOR
84 #define XMAX  1.0        // right clipping plane of SOR
85 #define XDIF (XMAX-XMIN) // length of SOR
86
87 #define ROTSTEP 5.0      // rotation angle per klick (degrees!)
88 #define BENCHFRAMES 200
89
90 #ifndef WIN32
91     typedef char BOOL;
92 #endif
93 typedef unsigned char UCHAR;
94 typedef unsigned short USHORT;
95 typedef unsigned long ULONG;
96
97 typedef double POINT4[4];
98 typedef float COLOR[3];
99
100 struct TRIANGLE
101 {
102     short a,b,c; // corners of the triangle
103     POINT4 n;     // normal vector
104     COLOR col;   // color, elements: [0]=red, [1]=green, [2]=blue
105 };
106
107 //=====
108 // Variables Definitions
109 //=====
110
111 struct BUTTON
112 {
113     double x,y; // x/y coordinates in range 0..1

```

```

114     double w,h; // width and height, range 0..1
115     short id; // button ID
116     char text[15+1];
117 } gButton[BUTTONS] =
118 { {.68, .94, .17, .05, 0, "Reset"}, .
119 { .68, .85, .12, .05, 1, "AngleX++"}, .
120 { .82, .85, .12, .05, 2, "AngleX--"}, .
121 { .68, .79, .12, .05, 3, "AngleY++"}, .
122 { .82, .79, .12, .05, 4, "AngleY--"}, .
123 { .68, .73, .12, .05, 5, "AngleZ++"}, .
124 { .82, .73, .12, .05, 6, "AngleZ--"}, .
125 { .68, .64, .12, .05, 7, "Cyl's++"}, .
126 { .82, .64, .12, .05, 8, "Cyl's--"}, .
127 { .68, .58, .12, .05, 9, "Sect's++"}, .
128 { .82, .58, .12, .05, 10, "Sect's--"}, .
129 { .68, .49, .17, .05, 11, "Rotation"}, .
130 { .68, .43, .17, .05, 12, "Revolution"}, .
131 { .68, .37, .17, .05, 13, "Cyl.Anim"}, .
132 { .68, .28, .14, .05, 14, "Shading"}, .
133 { .68, .22, .14, .05, 15, "Function"}, .
134 { .68, .16, .14, .05, 16, "Cyl.Mode"}, .
135 { .68, .07, .17, .05, 17, "Quit"} };
136
137 float gLightpos[] = {0.0, 0.0, 5.0, 1.0};
138 float gDiffuse[] = {1.0, 1.0, 1.0, 1.0};
139 float gAmbient[] = {1.0, 1.0, 1.0, 1.0};
140 float gSpecular[] = {1.0, 1.0, 1.0, 1.0};
141 float gBlack[] = {0.0, 0.0, 0.0, 1.0};
142 float gShininess[] = {50.0};
143 float gAxiscolor[] = {0.0, 1.0, 0.0, 1.0};
144 char gOffOn[2][3+1] = {"off", "on"};
145 char gNoYes[2][3+1] = {"no", "yes"};
146 char gAniname[2][11+1] = {"off", "active"};
147 char gCylname[CYLNUM][10+1] = {"frustums", "centered", "minimum"};
148 char gShadename[2][10] = {"unicolor", "flat"};
149 char gFuncname[4][14] = {"|x|", "x*x", "sqrt(|1-x*x|)", "sqrt(|x|)"};
150 POINT4* gVertices=NULL; // all used vertices of the SOR
151 struct TRIANGLE* gTri=NULL; // list of triangles
152 double gAngle[3] = {0.0,0.0,0.0}; // angle in degrees
153 short gTris; // number of triangles in the SOR
154 BOOL gRotation = FALSE; // toggle automatic rotation
155 short gFunction=0; // number of function used for SOR
156 short gShading=1; // shading model used for SOR
157 short gCylmode = 0; // cylinders mode: 0=frustums, 1=centered,
158 2=minimum
159 short gCyls = 10; // number of cylinders
160 short gCylsSave = gCyls; // memorizes gCyls for cylinder animation
161 short gSectors = 7; // number of sectors for each disc
162 BOOL gAnimation = gSectors; // revolution animation; if <gSectors
163 animation is active
164
165 //=====
166 // Functions Definitions
167 //=====
168
169 double Func(double x)
170 // returns function values which are already abs'd
171 {
172     switch (gFunction)
173 {

```

```

174     case 0: return(fabs(x));
175     case 1: return(x * x);
176     case 2: return((x>=-1.0 && x<=1.0) ? sqrt(fabs(1.0-x*x)) : -2.0*x*x+2.0);
177     case 3: return(sqrt(fabs(x)));
178     default: return(x);
179   }
180 }
181
182 //-----
183
184 double CylinderHeight(double x1, double x2)
185 // returns height of a column between 2 x values
186 {
187   switch (gCylmode)
188   {
189     case 1: return((Func(x1)+Func(x2))/2.0); break; // centered
190     case 2: return(MIN(Func(x1),Func(x2))); break; // minimum rule
191     default: return(Func(x1)); break;
192   }
193 }
194
195 //-----
196
197 void KeyboardCallback(UCHAR key, int x, int y)
198 {
199   if (key==27) exit(1); // exit on ESCAPE
200 }
201
202 //-----
203
204 // Computes vector product of a,b and c and returns it in n
205 // Call example: POINT4 a,b,c,n; VectorProduct(a,b,c,n);
206 void VectorProduct(POINT4 a, POINT4 b, POINT4 c, POINT4 n)
207 {
208   n[0] = (c[1]-b[1]) * (a[2]-b[2]) - (c[2]-b[2]) * (a[1]-b[1]);
209   n[1] = (c[2]-b[2]) * (a[0]-b[0]) - (c[0]-b[0]) * (a[2]-b[2]);
210   n[2] = (c[0]-b[0]) * (a[1]-b[1]) - (c[1]-b[1]) * (a[0]-b[0]);
211 }
212
213 //-----
214
215 void SetColor(COLOR col, float r, float g, float b)
216 // set color type variable to rgb color values
217 {
218   col[0] = r;
219   col[1] = g;
220   col[2] = b;
221 }
222
223 //-----
224
225 void NewPolygonList(void)
226 // create output body and assign it to a display list
227 {
228   short i,x,d,index;
229   double xstep;
230   double xpos;
231   short samples; // number of sample points on outline function
232
233   glNewList(SOR_LIST, GL_COMPILE);

```

```

234     glLineWidth(3.0); // thick axes lines
235     glBegin(GL_LINES); // draw coordinate system
236         glColor4fv(gAxiscolor);
237         glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, gAxiscolor);
238         glVertex3d( FROM_X, 0.0, 0.0); // x axis
239         glVertex3d(-FROM_X, 0.0, 0.0);
240         glVertex3d(0.0, FROM_X, 0.0); // y axis
241         glVertex3d(0.0,-FROM_X, 0.0);
242         glVertex3d(0.0, 0.0, FROM_X); // z axis
243         glVertex3d(0.0, 0.0,-FROM_X);
244     glEnd();
245     glLineWidth(1.0); // reset to default
246
247     glBegin(GL_TRIANGLES);
248     if (gCylmode==0) // no cylinders
249     {
250         for (x=0; x<gCyls; ++x)
251             for (d=0; d<gAnimation; ++d)
252             {
253                 for (index=0; index<2; ++index)
254                 {
255                     i = index + 2 * (x * gSectors + d);
256                     glColor3fv(gTri[i].col);
257                     glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, gTri[i].col);
258                     glNormal3dv(gTri[i].n);
259                     glVertex3dv(gVertices[gTri[i].a]);
260                     glVertex3dv(gVertices[gTri[i].b]);
261                     glVertex3dv(gVertices[gTri[i].c]);
262                 }
263             }
264     }
265     else // cylinders
266     {
267         for (x=0; x<gCyls; ++x)
268         {
269             for (d=0; d<gAnimation; ++d)
270             {
271                 for (index=0; index<2; ++index) // draw surface triangles
272                 {
273                     i = index + 2 * (x * gSectors + d);
274                     glColor3fv(gTri[i].col);
275                     glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, gTri[i].col);
276                     glNormal3dv(gTri[i].n);
277                     glVertex3dv(gVertices[gTri[i].a]);
278                     glVertex3dv(gVertices[gTri[i].b]);
279                     glVertex3dv(gVertices[gTri[i].c]);
280                 }
281             // draw vertical triangles:
282             index = 2 * gSectors * gCyls; // base index of vertical triangles
283             i = index + x*gSectors + d;
284             glColor3fv(gTri[i].col);
285             glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, gTri[i].col);
286             glNormal3dv(gTri[i].n);
287             glVertex3dv(gVertices[gTri[i].a]);
288             glVertex3dv(gVertices[gTri[i].b]);
289             glVertex3dv(gVertices[gTri[i].c]);
290
291             // draw rightmost vertical triangle:
292             i = index + gCyls*gSectors + d;
293             glColor3fv(gTri[i].col);

```

```

294     glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, gTri[i].col);
295     glNormal3dv(gTri[i].n);
296     glVertex3dv(gVertices[gTri[i].a]);
297     glVertex3dv(gVertices[gTri[i].b]);
298     glVertex3dv(gVertices[gTri[i].c]);
299   }
300 }
301 }
302
303 glEnd();
304
305 // draw function outline:
306 glBegin(GL_LINE_STRIP);
307   glLineWidth(3.0); // thick axes lines
308   glColor4fv(gAxiscolor);
309   glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, gAxiscolor);
310   glNormal3d(0.0,0.0,1.0);
311   xpos = XMIN;
312   samples = WDTH / 3;
313   xstep = XDIF / (double)samples;
314   for (x=0; x<=samples; ++x) // draw function outline
315   {
316     glVertex3d(xpos, Func(xpos), 0.0);
317     xpos += xstep;
318   }
319   glLineWidth(1.0); // reset to default
320 glEnd();
321
322 glEndList();
323 }
324
325 //-----
326
327 void GenerateNormals(void)
328 // generates normal vectors. important: order of cornerpoints must be
329 // counterclockwise!
330 {
331   short index;
332
333   for (index=0; index<gTris; ++index)
334     VectorProduct(gVertices[gTri[index].a], gVertices[gTri[index].b],
335                   gVertices[gTri[index].c], gTri[index].n);
336 }
337
338 //-----
339
340 void SmoothSOR(void)
341 // create vertices and triangles of output shape
342 {
343   short d,x; // loop counters
344   double xpos, dpos; // current vertex position in world coordinates
345   double xstep, dstep; // stepsizes
346   short index;
347   short base;
348
349   if (gVertices!=NULL) free(gVertices);
350   gVertices = (POINT4*)malloc((gCyls+1) * gSectors * sizeof(POINT4));
351   if (gVertices==NULL) exit(-1); // memory allocation error
352
353   xstep = XDIF / (double)gCyls;

```

```

354     dstep = 2.0 * PI / (double)gSectors; // step in radians
355
356     for (index=x=0; x<gCyls+1; ++x)
357     {
358         xpos = XMAX - (double)(gCyls - x) * xstep;
359         for (d=0; d<gSectors; ++d)
360         {
361             dpos = (double)d * dstep;
362             gVertices[index][0] = xpos;
363             gVertices[index][1] = Func(xpos) * cos(dpos);
364             gVertices[index][2] = Func(xpos) * sin(dpos);
365             ++index;
366         }
367     }
368
369     gTris = 2 * gSectors * gCyls;
370
371     // compose vertices to triangles:
372     if (gTri!=NULL) free(gTri);
373     gTri = (struct TRIANGLE*)malloc(gTris * sizeof(struct TRIANGLE));
374     if (gTri==NULL) exit(-1); // memory allocation error
375
376     for (x=0; x<gCyls; ++x)
377     {
378         for (d=0; d<gSectors; ++d)
379         {
380             index = 2 * (x * gSectors + d);
381             base = x * gSectors;
382
383             // compose upper left triangle:
384             gTri[index].a = base + d;
385             gTri[index].b = base + ((d + 1) % gSectors);
386             gTri[index].c = base + d + gSectors;
387             SetColor(gTri[index].col, 1.0, 0.0, 0.0);
388
389             // compose lower right triangle:
390             ++index;
391             gTri[index].a = base + d + gSectors;
392             gTri[index].b = base + ((d + 1) % gSectors);
393             gTri[index].c = base + ((d + 1) % gSectors) + gSectors;
394             SetColor(gTri[index].col, 1.0, 0.0, 0.0);
395         }
396     }
397
398     GenerateNormals();
399 }
400
401 //-----
402
403 void CylindersSOR(void)
404 // generate SOR with cylinders symbolizing approximation (from the inside)
405 {
406     short d,x;           // loop counters
407     double xpos, dpos;   // current vertex position in world coordinates
408     double xstep, dstep; // stepsizes
409     short index;
410     short base;
411     double radius;      // current radius
412     double x1,x2,x3;   // 3 sample x positions
413     BOOL leftside;      // boolean variable to mark left sides of cylinders

```

```

414     short  help;      // buffer for swap
415
416     if (gVertices!=NULL) free(gVertices);
417     gVertices = (POINT4*)malloc((2 * gCyls * gSectors + gCyls+1) * sizeof(POINT4));
418     if (gVertices==NULL) exit(-1); // memory allocation error
420
421     xstep = XDIF / (double)gCyls;
422     dstep = 2.0 * PI / (double)gSectors; // step in radians
423
424     // compute vertices on x axis:
425     index = 0;
426     for (x=0; x<gCyls+1; ++x)
427     {
428         gVertices[index][0] = XMIN + (double)x * xstep;
429         gVertices[index][1] = gVertices[index][2] = 0.0;
430         ++index;
431     }
432
433     // compute the rest of the vertices:
434     xpos = XMIN;
435     for (x=0; x<gCyls; ++x)
436     {
437         radius = CylinderHeight(xpos, xpos+xstep);
438         dpos = 0.0;
439         for (d=0; d<gSectors; ++d)
440         {
441             gVertices[index][0] = xpos;
442             gVertices[index][1] = radius * cos(dpos);
443             gVertices[index][2] = radius * sin(dpos);
444             dpos += dstep;
445             ++index;
446         }
447         xpos += xstep;
448         dpos = 0.0;
449         for (d=0; d<gSectors; ++d)
450         {
451             gVertices[index][0] = xpos;
452             gVertices[index][1] = radius * cos(dpos);
453             gVertices[index][2] = radius * sin(dpos);
454             dpos += dstep;
455             ++index;
456         }
457     }
458
459     gTris = 2 * gSectors * gCyls + (gCyls+1) * gSectors;
460
461     // compose vertices to triangles:
462     if (gTri!=NULL) free(gTri);
463     gTri = (struct TRIANGLE*)malloc(gTris * sizeof(struct TRIANGLE));
464     if (gTri==NULL) exit(-1); // memory allocation error?
465
466     // generate triangles parallel to x axis:
467     index = 0;
468     for (x=0; x<gCyls; ++x)
469     {
470         base = gCyls + 1 + 2 * x * gSectors;
471         for (d=0; d<gSectors; ++d)
472         {
473             // compose upper left triangle:

```

```

474     gTri[index].a = base + d;
475     gTri[index].b = base + ((d + 1) % gSectors);
476     gTri[index].c = base + d + gSectors;
477     SetColor(gTri[index].col, 1.0, 0.0, 0.0);
478     ++index;
479
480     // compose lower right triangle:
481     gTri[index].a = base + d + gSectors;
482     gTri[index].b = base + ((d + 1) % gSectors);
483     gTri[index].c = base + ((d + 1) % gSectors) + gSectors;
484     SetColor(gTri[index].col, 1.0, 0.0, 0.0);
485     ++index;
486   }
487 }
488
489 // generate triangles parallel to yz plane:
490 index = 2 * gSectors * gCyls;
491 for (x=0; x<gCyls; ++x)
492 {
493   leftside = FALSE;
494   x1 = XMIN+(double)(x-1)*xstep; // compute previous, current and next x
495   positions
496   x2 = XMIN+(double) x *xstep;
497   x3 = XMIN+(double)(x+1)*xstep;
498   if (x>0 && CylinderHeight(x1,x2) > CylinderHeight(x2,x3))
499     base = gCyls + 1 + (2 * x - 1) * gSectors;
500   else
501   {
502     base = gCyls + 1 + 2 * x * gSectors;
503     leftside = TRUE;
504   }
505
506   for (d=0; d<gSectors; ++d)
507   {
508     gTri[index].a = x;
509     gTri[index].b = base + d;
510     gTri[index].c = base + ((d + 1) % gSectors);
511     if (leftside==TRUE)
512     { help=gTri[index].b; gTri[index].b=gTri[index].c; gTri[index].c=help; }
513     SetColor(gTri[index].col, 0.0, 0.0, 1.0);
514     ++index;
515   }
516 }
517
518 // generate rightmost yz plane cover:
519 base = gCyls + 1 + 2 * gCyls * gSectors - gSectors; // jump to last
520 cylinder
521 for (d=0; d<gSectors; ++d)
522 {
523   gTri[index].a = x;
524   gTri[index].b = base + d;
525   gTri[index].c = base + ((d + 1) % gSectors);
526   SetColor(gTri[index].col, 0.0, 0.0, 1.0);
527   ++index;
528 }
529
530 GenerateNormals();
531 }
532
533 //-----

```

```

534
535 void NewVertexList(void)
536 {
537     if (gCylmode>0) CylindersSOR();
538     else SmoothSOR();
539 }
540
541 //-----
542
543 void WriteText(double x, double y, const char* text, ...)
544 // x/y are of range 0..1
545 {
546     short length,i;
547     char output[80+1]; // buffer for output text
548     va_list arglist;
549
550     va_start(arglist,text);
551     length = vsprintf(output, text, arglist);
552     if (length>80) exit(-1); // text array too small
553     va_end(arglist);
554     output[length] = '\n';
555     glRasterPos3d(x*DX+FROM_X, y*DY+FROM_Y, TO_Z-DZ/100.0);
556     for (i=0; i<length; ++i)
557         glutBitmapCharacter(GLUT_BITMAP_8_BY_13, output[i]);
558 }
559
560 //-----
561
562 short TextPixelWidth(const char* text)
563 // returns the width of a text in pixels
564 {
565     short i,width=0;
566
567     for(i=0; i<(short)strlen(text); ++i)
568         width += glutBitmapWidth(GLUT_BITMAP_8_BY_13, text[i]);
569     return(width);
570 }
571
572 //-----
573
574 void DrawButtons(void)
575 {
576     short i;
577     double x1,x2,y1,y2;
578
579     for (i=0; i<BUTTONS; ++i)
580     {
581         x1 = gButton[i].x;
582         x2 = x1 + gButton[i].w;
583         y1 = gButton[i].y;
584         y2 = y1 + gButton[i].h;
585
586         x1 = x1 * DX + FROM_X;
587         x2 = x2 * DX + FROM_X;
588         y1 = y1 * DY + FROM_Y;
589         y2 = y2 * DY + FROM_Y;
590
591         glBegin(GL_LINE_LOOP);
592             glColor4fv(gBlack);
593             glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, gBlack);

```

```

594     glNormal3d(0.0, 0.0, 1.0);
595     glVertex3d(x1, y1, TO_Z-DZ/100.0);
596     glVertex3d(x1, y2, TO_Z-DZ/100.0);
597     glVertex3d(x2, y2, TO_Z-DZ/100.0);
598     glVertex3d(x2, y1, TO_Z-DZ/100.0);
599     glEnd();
600
601     WriteText(gButton[i].x + .01,
602               gButton[i].y + gButton[i].h/2.0 - .01,
603               gButton[i].text);
604 }
605 }
606
607 //-----
608
609 void SetViewParams(short w, short h)
610 // w,h = screen windows size
611 {
612     glViewport(0, 0, w, h); // x,y,wdth,hght of view window
613
614     glMatrixMode(GL_PROJECTION);
615     glLoadIdentity();
616     glOrtho(FROM_X, TO_X, FROM_Y, TO_Y, FROM_Z, TO_Z);
617 //    gluLookAt(0.0, 0.0, 700.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
618     glMatrixMode(GL_MODELVIEW);
619 }
620
621 //-----
622
623 short GetButtonID(short x, short y)
624 // returns button ID. if no button pressed return -1
625 // x/y are the mouse coordinates as pixels
626 {
627     short i;
628     double sx,sy; // screen values in fraction of screen size
629
630     sx = (double)x / WDTH;
631     sy = (double)y / HGBT;
632
633     for (i=0; i<BUTTONS; ++i)
634         if (sx>=gButton[i].x && sx<gButton[i].x+gButton[i].w &&
635             sy>=gButton[i].y && sy<gButton[i].y+gButton[i].h)
636             return(gButton[i].id);
637     return(-1);
638 }
639
640 //-----
641
642 void DrawStatus(void)
643 {
644     short i;
645     double x,y;
646
647     for (i=0; i<BUTTONS; ++i)
648     {
649         x = gButton[i].x + gButton[i].w + .01;
650         y = gButton[i].y + .015;
651         switch(i)
652         {
653             case 2: WriteText(x, y, "%3.0f", gAngle[0]); break;

```

```

654     case  4: WriteText(x, y, "%3.0f", gAngle[1]); break;
655     case  6: WriteText(x, y, "%3.0f", gAngle[2]); break;
656     case  8: WriteText(x, y, "%d", gCyls); break;
657     case 10: WriteText(x, y, "%d", gSectors); break;
658     case 11: WriteText(x, y, "%s", gOffOn[gRotation]); break;
659     case 12: WriteText(x, y, "%s", gAniname[gAnimation<gSectors]); break;
660     case 13: WriteText(x, y, "%s", gAniname[gCyls<gCylsSave]); break;
661     case 14: WriteText(x, y, "%s", gShadename[gShading]); break;
662     case 15: WriteText(x, y, "%s", gFuncname[gFunction]); break;
663     case 16: WriteText(x, y, "%s", gCylname[gCylmode]); break;
664     default: break;
665   }
666 }
667 }
668
669 //-----
670
671 void DrawFunction(void)
672 {
673   glRotated(gAngle[0],1.0,0.0,0.0); // rotation in degrees!
674   glRotated(gAngle[1],0.0,1.0,0.0);
675   glRotated(gAngle[2],0.0,0.0,1.0);
676   glCallList(SOR_LIST);
677 }
678
679 //-----
680
681 void RedrawScreen(void)
682 {
683 #ifdef WIN32
684   clock_t start,end,diff;
685   char    buf[20];
686   double  fps;
687
688   start = clock();
689 #endif
690   glLoadIdentity();
691   glClear(GL_COLOR_BUFFER_BIT);
692   glClear(GL_DEPTH_BUFFER_BIT);
693 #ifndef BENCHMARK
694   DrawButtons();
695   DrawStatus();
696 #endif
697   DrawFunction();
698 #ifdef WIN32
699   glLoadIdentity();
700   glColor4fv(gBlack);
701   glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, gBlack);
702   end = clock();
703   diff = end - start;
704   if (diff==0) diff=1;      // prevent division by zero error
705   fps = (double)CLOCKS_PER_SEC / (double)diff;
706   sprintf(buf, "%6.2f fps", fps);
707   WriteText(0.05, 0.05, buf);
708 #endif
709   glFlush();
710   glutSwapBuffers();
711 }
712
713 //-----

```

```

714
715 void DisplayCallback(void)
716 {
717     RedrawScreen();
718 }
719
720 //-----
721
722 void ReshapeCallback(GLsizei w, GLsizei h)
723 // clears window
724 {
725     SetViewParams(w, h);
726     RedrawScreen();
727 }
728
729 //-----
730
731 void IdleCallback(void)
732 // is called if no other action takes place
733 {
734     short i;
735     BOOL vertices=FALSE, polygons=FALSE, redraw=FALSE; // marks what has to be
736     redone
737
738     if (gRotation)
739     {
740         for (i=1; i<=2; ++i)
741         {
742             gAngle[i] += ROTSTEP/5.0;
743             if (gAngle[i] >= 360.0) gAngle[i] -= 360.0;
744         }
745         redraw = TRUE;
746     }
747     if (gAnimation < gSectors)
748     {
749         ++gAnimation;
750         polygons = redraw = TRUE;
751     }
752     if (gCyls < gCylsSave)
753     {
754         ++gCyls;
755         vertices = polygons = redraw = TRUE;
756     }
757     if (vertices)
758         NewVertexList();
759     if (polygons)
760         NewPolygonList();
761     if (redraw)
762         RedrawScreen();
763 }
764
765 //-----
766
767 void MouseCallback(int button, int state, int x, int y)
768 {
769     short axis=0; // 0=x, 1=y, 2=z axis
770
771     // convert y coordinate to screen coordinates:
772     y = HIGHT - y;
773

```

```

774     if (button==GLUT_RIGHT_BUTTON & state==GLUT_DOWN)
775         exit(1);
776
777     if (button==GLUT_LEFT_BUTTON & state==GLUT_DOWN)
778     {
779         switch (GetButtonID(x,y))
780         {
781             case -1: // no button pressed
782                 break;
783             case 0: // Reset
784                 gAngle[0] = gAngle[1] = gAngle[2] = 0.0;
785                 gRotation = FALSE;
786                 gAnimation = gSectors;
787                 gCyls = gCylsSave;
788                 NewVertexList();
789                 NewPolygonList();
790                 RedrawScreen();
791                 break;
792             case 5: ++axis; // angle z ++
793             case 3: ++axis; // angle y ++
794             case 1: // angle x ++
795                 gAngle[axis] += ROTSTEP;
796                 if (gAngle[axis] >= 360.0) gAngle[axis] -= 360.0;
797                 RedrawScreen();
798                 break;
799             case 6: ++axis; // angle z --
800             case 4: ++axis; // angle y --
801             case 2: // angle x --
802                 gAngle[axis] -= ROTSTEP;
803                 if (gAngle[axis] < 0.0) gAngle[axis] += 360.0;
804                 RedrawScreen();
805                 break;
806             case 7: // Cylinders ++
807                 gCyls++;
808                 if (gCyls>99) gCyls = 99;
809                 gCylsSave = gCyls;
810                 NewVertexList();
811                 NewPolygonList();
812                 RedrawScreen();
813                 break;
814             case 8: // Cylinders --
815                 gCyls--;
816                 if (gCyls<1) gCyls = 1;
817                 gCylsSave = gCyls;
818                 NewVertexList();
819                 NewPolygonList();
820                 RedrawScreen();
821                 break;
822             case 9: // Sectors ++
823                 gSectors++;
824                 if (gSectors>99) gSectors = 99;
825                 gAnimation = gSectors;
826                 NewVertexList();
827                 NewPolygonList();
828                 RedrawScreen();
829                 break;
830             case 10: // Sections --
831                 gSectors--;
832                 if (gSectors<3) gSectors = 3;
833                 gAnimation = gSectors;

```

```

834         NewVertexList();
835         NewPolygonList();
836         RedrawScreen();
837         break;
838     case 11: // auto rotation
839         gRotation = !gRotation;
840         RedrawScreen();
841         break;
842     case 12: // revolution animation
843         gAnimation = 0;
844         NewPolygonList();
845         RedrawScreen();
846         break;
847     case 13: // Cylinder animation
848         gCyls = 2;
849         NewVertexList();
850         NewPolygonList();
851         RedrawScreen();
852         break;
853     case 14: // Shading model
854         ++gShading;
855         if (gShading>=2) gShading=0;
856
857         switch(gShading)
858         {
859             case 0: glDisable(GL_LIGHTING); break;
860             case 1: glEnable(GL_LIGHTING); break;
861         }
862         NewPolygonList();
863         RedrawScreen();
864         break;
865     case 15: // Function
866         ++gFunction;
867         if (gFunction>=FUNCNUM) gFunction=0;
868         NewVertexList();
869         NewPolygonList();
870         RedrawScreen();
871         break;
872     case 16: // Cylinders
873         ++gCylmode;
874         if (gCylmode>=CYLNUM) gCylmode=0;
875         NewVertexList();
876         NewPolygonList();
877         RedrawScreen();
878         break;
879         case 17: // quit
880         exit(1);
881         break;
882     }
883 }
884
885 //-----
886
887 int main(int argc, char* argv[])
888 {
889     glutInit(&argc, argv);
890     glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
891     glutInitWindowSize(WDTH, HGHT);
892

```

```
894     glutInitWindowPosition(100, 100);
895     glutCreateWindow("Shading SOR");
896
897     NewVertexList();
898     NewPolygonList();
899
900     SetViewParams(WDTW, HGHT);
901     glClearColor((GLclampf)1.0, (GLclampf)1.0, (GLclampf)1.0, (GLclampf)0.0); // white
902     RedrawScreen();
903
904     glutDisplayFunc(DisplayCallback); // same as WM_PAINT => is also called after window creation
905     glutReshapeFunc(ReshapeCallback);
906     glutIdleFunc(IdleCallback);
907
908     glutMouseFunc(MouseCallback);
909     glutKeyboardFunc(KeyboardCallback);
910
911     glShadeModel(GL_SMOOTH);
912     glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, gSpecular);
913     glMaterialfv(GL_FRONT_AND_BACK, GL_SHININESS, gShininess);
914     glLightfv(GL_LIGHT0, GL_POSITION, gLightpos);
915     glEnable(GL_LIGHT0);
916     glEnable(GL_DEPTH_TEST);
917     if (gShading==1)
918         glEnable(GL_LIGHTING);
919
920     glutMainLoop(); // starts event-processing loop
921
922     free(gTri);
923     free(gVertices);
924     return(0);
925 }
926
927 //=====
928 // The End
929 //=====
930
```

6.4.4 The SPHIGS Version

```

1  ****
2  Filename:      UNIX-SOR.CPP
3  Project:       Shading of Surfaces of Revolution with SPHIGS
4  :           Master's Thesis Summer Term 1997
5  Programmer:    Juergen Schulze-Doebold
6  :           Email: joe@studbox.uni-stuttgart.de
7  Advisor:       Prof. A. Hausknecht
8  Institution:   University of Massachusetts Dartmouth
9  Compiler:      GNU C
10 Environment:   Unix or Linux
11 Requirements:  SPHIGS must be installed
12 Project Files: UNIX-SOR.C, MAKEFILE, SPHIGS_BUTTONS.C,
13 :           SPHIGS_BUTTONS.H, SRGP_BUTTONS.H
14 Last Changes:  12/03/97
15 ****
16
17 /*-----*/
18 /* Include Files */
19 /*-----*/
20
21 #include "sphigslocal.h"
22 #include "sphigs_buttons.h"
23 #include <stdio.h>
24 #include <stdlib.h>
25 #include <string.h>
26 #include <math.h>
27 #include <sys/timeb.h>
28
29 /*-----*/
30 /* Macros */
31 /*-----*/
32
33 #undef srgp_point
34
35 /* Define View Cube */
36 #define kViewLeft   -1.2
37 #define kViewBottom -1.2
38 #define kViewRight  1.2
39 #define kViewTop    1.2
40 #define kViewFront  8
41 #define kViewBack   -8
42
43 #define kNumberOfPlanes 8
44 #define kBlackColor  SRGP_BLACK
45 #define kWhiteColor   SRGP_WHITE
46 #define kRedColor     2
47 #define kGreenColor   3
48 #define kBlueColor    4
49 #define kGreyColor    5
50 #define kPinkColor    6
51 #define kTanColor     7
52 #define kLavenderColor 8

```

```
53 #define kDefaultFont      0
54 #define kStdFont       1
55 #define kButtonFont  2
56
57 #define kScreenWidthWidth 600
58 #define kScreenWindowHeight 700
59
60
61 #define vector vector_3D
62 #define matrix matrix_4x4
63 #define srgp_rectangle rectangle
64 #define MaxStringLength 255
65
66
67 /*-----*/
68 /* Variable Definitions */
69 /*-----*/
70
71 char *gFontNames[] =
72 { "-adobe-times-bold-r-normal--*-12-*-*-*-*-*-*",
73   "-adobe-times-medium-r-normal--*-12-*-*-*-*-*-*-*",
74   "-adobe-helvetica-bold-r-normal--*-12-*-*-*-*-*-*-*"
75 };
76 NDC_rectangle gScreenViewRect;
77
78 /*-----*/
79 /* Buttons */
80 /*-----*/
81
82 enum ButtonIDs {   QuitButtonID =           0,
83                  RedrawButtonID =        1,
84                  IncButtonID =          2,
85                  DecButtonID =          3,
86                  ParameterButtonID =    4,
87                  FirstCoordinateButtonID = 5,
88                  SecondCoordinateButtonID = 6,
89                  ThirdCoordinateButtonID = 7,
90                  FourthCoordinateButtonID = 8
91 };
92 #define kLastButtonID FourthCoordinateButtonID
93 int gActiveButton;
94 enum buttonDims { stdButtonWidth = 42,
95                   wideButtonWidth = 110,
96                   stdButtonHeight = 22,
97                   stdMargin = 4,
98                   stdButtonGap = 15,
99                   quitButtonRightOffset = 50
100 };
101
102 /*-----*/
103 /* Surface parameters */
104 /*-----*/
105 #define DX (gXmax-gXmin)
106 #define kSurfaceID 1
107 double gXmin=-1.0, gXmax=1.0; /* Surface parameters */
108 int gM=10, gN=10;
109 point* gVertices=NULL;
110 int gNumberOfTris;
111 vertex_index* gTri=NULL;
112
```

```

113  /*-----*/
114  /* Axes */
115  /*-----*/
116  #define kDefautAxisLength 2
117  point gXAxis[3], gYAxis[3], gZAxis[3];
118
119  /*-----*/
120  /* 3D Viewing */
121  /*-----*/
122  typedef struct{double uMin, vMin, uMax, vMax;} viewPlaneWindowBounds;
123  typedef struct{point xyzMin, xyzMax;} viewCubicRegion;
124  vector vectorText;
125  typedef struct
126  { point viewRefPt;
127   vector viewPlaneNormal, viewUpVector;
128   matrix VOMatrix;
129   int projectionType;
130   point projRefPt;
131   viewPlaneWindowBounds viewPlaneWindow;
132   double frontClippingPlaneDist, backClippingPlaneDist;
133   viewCubicRegion stdViewVolume, stdViewClip;
134   matrix VMMatrix;
135   NDC_rectangle screenViewBounds;
136   int viewBackgroundColor;
137 } viewInfo;
138
139 viewInfo gTheViewInfo[4];
140 int gSelectedView = 0;
141
142 /*-----*/
143 /* View Parameters */
144 /*-----*/
145 #define ParameterDelta 0.8
146 #define kNumParameters 8 /* number of parameters */
147 enum ParameterIDs
148 {
149   ViewRefPtID = 0,
150   ViewRefNormaID = 1,
151   ViewUpVectorID = 2,
152   ProjRefPtID = 3,
153   ViewPlaneWindowID = 4,
154   ViewPlaneWidnowSizeCenterID = 5,
155   ViewClippingPlanesID = 6,
156   SurfaceParamsID = 7
157 };
158 int gSelectedParameter = ProjRefPtID;
159 char *gParameterNames[] =
160 {
161   "View Ref Point",
162   "View Ref Normal",
163   "View-Up Vector",
164   "Projection Ref Pt",
165   "View Plane Window",
166   "VPW Size & Center",
167   "Front & Back Planes",
168   "Surface Params"
169 };
170 srgp__rectangle gCoordinateOutputRects[4];
171 srgp__rectangle gParameterTextRect;

```

```

173    srgp__rectangle      gButtonRect;
174
175    char CoordinateNames[ ][4][9] =
176    {
177        {"x", "Y", "z", "unused" },
178        {"x", "Y", "z", "unused" },
179        {"x", "Y", "z", "unused" },
180        {"u", "v", "n", "unused" },
181        {"uMin", "vMin", "uMax", "vMax" },
182        {"Width", "Height", "u", "v" },
183        {"Front", "Back", "unused", "unused" },
184        {"xMin", "xMax", "m", "n" }
185    };
186
187    int gSelectedCoordinate = 0;
188
189    /*-----*/
190    /* Function Declarations */
191    /*-----*/
192
193    void SetUpViewReferenceCoordinateSystem(int);
194    void UpdateSurfaceStructure(double, double, int, int);
195    void CreateSurfaceOfRevolution(double, double, int, int);
196    void CreateSurfaceStructure(double, double, int, int);
197    void DrawCoordinates(void);
198
199    /*-----*/
200    /* Function Definitions */
201    /*-----*/
202
203    boolean pt3DinRect (point pt, NDC_rectangle rect)
204    {
205        return ( (rect.bottom_left.x <= pt[0]) && (pt[0]< rect.top_right.x) &&
206                (rect.bottom_left.y <= pt[1]) && (pt[1]< rect.top_right.y) );
207    }
208
209    /*-----*/
210
211    void setBackgroundColor(int color)
212    {
213        srgp__attribute_group savedAttributes;
214
215        SRG_P_inquireAttributes(&savedAttributes);
216        savedAttributes.background_color = color;
217        SRG_P_setAttributes(&savedAttributes);
218    }
219
220    /*-----*/
221
222    void eraseTheRect (srgp__rectangle theRect)
223    {
224        srgp__attribute_group savedAttributes;
225
226        setBackgroundColor(kWhiteColor);
227        SRG_P_inquireAttributes(&savedAttributes);
228        SRG_P_setFillStyle(SOLID);
229        SRG_PSetColor(savedAttributes.background_color);
230        SRG_P_fillRectangle(theRect);
231        SRG_P_setAttributes(&savedAttributes);
232    }

```

```

233 /*-----*/
234 void drawStr(srgp_point screenLocation, char *theStr)
235 {
236     int delta = 1;
237     int indent = 2;
238     int width, height, descent;
239     srgp_rectangle textRect;
240     srgp_attribute_group savedAttributes;
241
242     setBackgroundColor(kWhiteColor);
243     SRGP_inquireAttributes(&savedAttributes);
244     SRGPSetColor(kWhiteColor);
245     if ( strlen(theStr) > 0 )
246     {
247         SRGP_inquireTextExtent(theStr, &width, &height, &descent);
248         screenLocation = SRGP_defPoint(screenLocation.x + indent,
249                                         screenLocation.y + indent);
250         textRect = SRGP_defRectangle( screenLocation.x,
251                                     screenLocation.y,
252                                     screenLocation.x + width + delta,
253                                     screenLocation.y + height + descent + delta);
254         screenLocation.y = screenLocation.y + descent;
255         SRGP_setClipRectangle (textRect);
256         eraseTheRect(textRect);
257         SRGPSetColor(kBlackColor);
258         SRGP_setWriteMode(WRITE_REPLACE);
259         SRGP_text(screenLocation, theStr);
260     }
261     SRGP_setAttributes(&savedAttributes);
262 }
263 /*-----*/
264 void drawReal(srgp_point screenLocation, double theReal)
265 {
266     char numStr[MaxStringLength];
267     int width, height, descent;
268
269     sprintf(numStr,"%7.2f",theReal);
270     drawStr(screenLocation,numStr);
271 }
272 /*-----*/
273 void CreateAxes(double length)
274 {
275     SPH_defPoint(gXAxis[0],0, 0, 0);
276     SPH_defPoint(gXAxis[1],length, 0, 0);
277     SPH_defPoint(gXAxis[2],length,0.1, 0); /* Label position */
278
279     SPH_defPoint(gYAxis[0],0, 0, 0);
280     SPH_defPoint(gYAxis[1],0, length, 0);
281     SPH_defPoint(gYAxis[2],0.1, length*.9, 0); /* Label position */
282
283     SPH_defPoint(gZAxis[0],0, 0, 0);
284     SPH_defPoint(gZAxis[1],0, 0, length);
285     SPH_defPoint(gZAxis[2],0.1, 0, length*.9); /* Label position */
286 }
```

```

293 /*-----*/
294 void SetUpViewReferenceCoordinateSystem(int viewIndex)
295 {
296     SPH_evaluateViewOrientationMatrix(gTheViewInfo[viewIndex].viewRefPt,
297                                     gTheViewInfo[viewIndex].viewPlaneNormal,
298                                     gTheViewInfo[viewIndex].viewUpVector,
299                                     gTheViewInfo[viewIndex].VOMatrix);
300 }
301
302 /*-----*/
303 void SetUpProjection(int viewID)
304 {
305     SPH_evaluateViewMappingMatrix(
306         gTheViewInfo[viewID].viewPlaneWindow.uMin,
307         gTheViewInfo[viewID].viewPlaneWindow.uMax,
308         gTheViewInfo[viewID].viewPlaneWindow.vMin,
309         gTheViewInfo[viewID].viewPlaneWindow.vMax,
310         gTheViewInfo[viewID].projectionType,
311         gTheViewInfo[viewID].projRefPt,
312         gTheViewInfo[viewID].frontClippingPlaneDist,
313         gTheViewInfo[viewID].backClippingPlaneDist,
314         gTheViewInfo[viewID].stdViewVolume.xyzMin[0],
315         gTheViewInfo[viewID].stdViewVolume.xyzMax[0],
316         gTheViewInfo[viewID].stdViewVolume.xyzMin[1],
317         gTheViewInfo[viewID].stdViewVolume.xyzMax[1],
318         gTheViewInfo[viewID].stdViewVolume.xyzMin[2],
319         gTheViewInfo[viewID].stdViewVolume.xyzMax[2],
320         gTheViewInfo[viewID].stdViewVolume.xyzMin[2],
321         gTheViewInfo[viewID].stdViewVolume.xyzMax[2],
322         gTheViewInfo[viewID].VMMatrix);
323 }
324
325 /*-----*/
326 void SetUpView(int viewID)
327 {
328     int viewColor;
329
330     SPH_setViewRepresentation(viewID,
331                               gTheViewInfo[viewID].VOMatrix,
332                               gTheViewInfo[viewID].VMMatrix,
333                               gTheViewInfo[viewID].stdViewClip.xyzMin[0],
334                               gTheViewInfo[viewID].stdViewClip.xyzMax[0],
335                               gTheViewInfo[viewID].stdViewClip.xyzMin[1],
336                               gTheViewInfo[viewID].stdViewClip.xyzMax[1],
337                               gTheViewInfo[viewID].stdViewClip.xyzMin[2],
338                               gTheViewInfo[viewID].stdViewClip.xyzMax[2],
339                               gTheViewInfo[viewID].stdViewClip.xyzMin[2],
340                               gTheViewInfo[viewID].stdViewClip.xyzMax[2]);
341
342     switch (viewID)
343     {
344         case 0: SPH_setRenderingMode (viewID,WIREFRAME); break;
345         case 1: SPH_setRenderingMode (viewID,FLAT); break;
346         case 2: SPH_setRenderingMode (viewID,LIT_FLAT); break;
347         case 3: SPH_setRenderingMode (viewID,LIT_FLAT);
348             SPH_removePointLightSource(viewID, 0); /* remove default lightsource */
349             SPH_setViewPointLightSource(viewID, -20, -20, 100);
350             break;
351     }
352     SPH_setViewBackgroundColor(viewID,gTheViewInfo[viewID].viewBackgroundColor);

```

```

353     }
354
355     /*-----*/
356
357     void DrawCoordinate(boolean selected, srgp_rectangle itsRect,
358                         double theCoordinate)
359     {
360         srgp_attribute_group att_group;
361         srgp_rectangle clipRect;
362         srgp_point tempPt;
363         SRG_P_inquireAttributes(&att_group);
364
365         clipRect = SRG_P_defRectangle(itsRect.bottom_left.x - 1,
366                                     itsRect.bottom_left.y,
367                                     itsRect.top_right.x,
368                                     itsRect.top_right.y + 1);
369         SRG_P_setClipRectangle(clipRect);
370         if (!selected)
371             eraseTheRect(clipRect);
372         tempPt.x = itsRect.bottom_left.x;
373         tempPt.y = itsRect.bottom_left.y;
374         drawReal(tempPt, theCoordinate);
375         if (selected) SRG_P_setLineWidth(2);
376         else SRG_P_setLineWidth(1);
377         SRG_P_rectangle(itsRect);
378         SRG_P_setLineWidth(1);
379         SRG_P_setAttributes(&att_group);
380     }
381
382     /*-----*/
383
384
385     /*-----*/
386
387     void SelectCoordinate(int newSelectedCoordinate)
388     {
389         gSelectedCoordinate = newSelectedCoordinate;
390         DrawCoordinates();
391     }
392
393     /*-----*/
394
395     void DrawCoordinates(void)
396     {
397         int nextCoordinate;
398         srgp_rectangle rect;
399
400         rect = SRG_P_defRectangle(gCoordinateOutputRects[0].bottom_left.x-2,
401                                   gCoordinateOutputRects[0].bottom_left.y-2,
402                                   gCoordinateOutputRects[3].top_right.x+2,
403                                   gCoordinateOutputRects[3].top_right.y+2);
404         SRG_P_setClipRectangle(rect);
405         SRG_P_setFillStyle(SOLID);
406         SRG_PSetColor(kWhiteColor);
407         SRG_P_fillRectangle(rect);
408         SRG_PSetColor(kBlackColor);
409
410         switch (gSelectedParameter)
411         {
412             case ViewRefPtID:

```

```

413     for (nextCoordinate = 0; nextCoordinate < 3; nextCoordinate++)
414         DrawCoordinate( nextCoordinate==gSelectedCoordinate,
415                         gCoordinateOutputRects[nextCoordinate],
416                         gTheViewInfo[gSelectedView].viewRefPt[nextCoordinate]);
417     break;
418 case ViewRefNormalID:
419     for (nextCoordinate = 0; nextCoordinate < 3; nextCoordinate++)
420         DrawCoordinate( nextCoordinate==gSelectedCoordinate,
421                         gCoordinateOutputRects[nextCoordinate],
422
423                         gTheViewInfo[gSelectedView].viewPlaneNormal[nextCoordinate]);
424     break;
425 case ViewUpVectorID:
426     for (nextCoordinate = 0; nextCoordinate < 3; nextCoordinate++)
427         DrawCoordinate( nextCoordinate==gSelectedCoordinate,
428                         gCoordinateOutputRects[nextCoordinate],
429                         gTheViewInfo[gSelectedView].viewUpVector[nextCoordinate]);
430     break;
431 case ProjRefPtID:
432     for (nextCoordinate = 0; nextCoordinate < 3; nextCoordinate++)
433         DrawCoordinate( nextCoordinate==gSelectedCoordinate,
434                         gCoordinateOutputRects[nextCoordinate],
435                         gTheViewInfo[gSelectedView].projRefPt[nextCoordinate]);
436     break;
437 case ViewPlaneWindowID:
438     DrawCoordinate( 0==gSelectedCoordinate,
439                     gCoordinateOutputRects[0],
440                     gTheViewInfo[gSelectedView].viewPlaneWindow.uMin);
441     DrawCoordinate( 1==gSelectedCoordinate,
442                     gCoordinateOutputRects[1],
443                     gTheViewInfo[gSelectedView].viewPlaneWindow.vMin);
444     DrawCoordinate( 2==gSelectedCoordinate,
445                     gCoordinateOutputRects[2],
446                     gTheViewInfo[gSelectedView].viewPlaneWindow.uMax);
447     DrawCoordinate( 3==gSelectedCoordinate,
448                     gCoordinateOutputRects[3],
449                     gTheViewInfo[gSelectedView].viewPlaneWindow.vMax);
450     break;
451 case ViewPlaneWidnowSizeCenterID:
452     DrawCoordinate( 0==gSelectedCoordinate,
453                     gCoordinateOutputRects[0],
454                     gTheViewInfo[gSelectedView].viewPlaneWindow.uMax-
455                     gTheViewInfo[gSelectedView].viewPlaneWindow.uMin);
456     DrawCoordinate( 1==gSelectedCoordinate,
457                     gCoordinateOutputRects[1],
458                     gTheViewInfo[gSelectedView].viewPlaneWindow.vMax-
459                     gTheViewInfo[gSelectedView].viewPlaneWindow.vMin);
460     DrawCoordinate( 2==gSelectedCoordinate,
461                     gCoordinateOutputRects[2],
462                     (gTheViewInfo[gSelectedView].viewPlaneWindow.uMax+
463                     gTheViewInfo[gSelectedView].viewPlaneWindow.uMin)/2.0);
464     DrawCoordinate( 3==gSelectedCoordinate,
465                     gCoordinateOutputRects[3],
466                     (gTheViewInfo[gSelectedView].viewPlaneWindow.vMax+
467                     gTheViewInfo[gSelectedView].viewPlaneWindow.vMin)/2.0);
468     break;
469 case ViewClippingPlanesID:
470     DrawCoordinate( 0==gSelectedCoordinate,
471                     gCoordinateOutputRects[0],
472                     gTheViewInfo[gSelectedView].frontClippingPlaneDist);

```

```

473     DrawCoordinate( 1==gSelectedCoordinate,
474                     gCoordinateOutputRects[1],
475                     gTheViewInfo[gSelectedView].backClippingPlaneDist);
476     break;
477     case SurfaceParamsID:
478         DrawCoordinate(      0==gSelectedCoordinate,      gCoordinateOutputRects[0],
479                         gXmin);
480         DrawCoordinate(      1==gSelectedCoordinate,      gCoordinateOutputRects[1],
481                         gXmax);
482         DrawCoordinate(  2==gSelectedCoordinate, gCoordinateOutputRects[2], gM);
483         DrawCoordinate(  3==gSelectedCoordinate, gCoordinateOutputRects[3], gN);
484         break;
485     }
486 }
487 /*-----*/
488 void DrawParameterName(char *itsName)
489 {
490     srgp_attribute_group att_group;
491
492     setBackgroundColor(kWhiteColor);
493     SRGP_inquireAttributes(&att_group);
494     SRGP_setClipRectangle(gParameterTextRect);
495     SRGPSetColor(kWhiteColor);
496     SRGP_fillRectangle(gParameterTextRect);
497     SRGP_setColor(kBlackColor);
498     SRGPSetFont(kDefaultFont);
499     drawStr(gParameterTextRect.bottom_left, itsName);
500     SRGP_setAttributes(&att_group);
501 }
502 /*-----*/
503 void DrawSelectedParameter(void)
504 {
505     DrawParameterName(gParameterNames[gSelectedParameter]);
506     DrawCoordinates();
507 }
508 /*-----*/
509 void UpdateButtonTitles(int param)
510 {
511     int buttonID;
512
513     setBackgroundColor(kWhiteColor);
514     SRGP_setLineWidth(1);
515     for (buttonID = FirstCoordinateButtonID;
516          buttonID <= FourthCoordinateButtonID; buttonID++) {
517         if ((strcmp(CoordinateNames[param][buttonID-
518 FirstCoordinateButtonID], "unused") ==
519             0)
520             SPH_hideButton(buttonID);
521         else {
522             SPH_setButtonTitle(buttonID, CoordinateNames[param][buttonID -
523                                         FirstCoordinateButtonID ]);
524             SPH_showButton(buttonID);
525         }
526     }
527 }
```

```

533     }
534
535     /*-----*/
536
537 void CycleToNextParameter(void)
538 {
539     ++gSelectedParameter;
540     gSelectedParameter %= kNumParameters;
541     gSelectedCoordinate = 0;
542     UpdateButtonTitles(gSelectedParameter);
543     DrawSelectedParameter();
544 }
545
546     /*-----*/
547
548 void HiliteSelectedView(void)
549 {
550     srgp_rectangle rect;
551     srgp_attribute_group att_group;
552
553     SRGP_inquireAttributes(&att_group);
554     SRGPSetColor(kBlackColor);
555     SRGP_setLineWidth(5);
556     rect = SRGP_defRectangle(
557         gTheViewInfo[gSelectedView].screenViewBounds.bottom_left.x *
558         kScreenWidthWidth,
559         gTheViewInfo[gSelectedView].screenViewBounds.bottom_left.y *
560         kScreenWidthWidth,
561         gTheViewInfo[gSelectedView].screenViewBounds.top_right.x *
562         kScreenWidthWidth,
563         gTheViewInfo[gSelectedView].screenViewBounds.top_right.y *
564         kScreenWidthWidth);
565     SRGP_setClipRectangle (rect);
566     SRGP_rectangle(rect);
567     SRGP_setAttributes(&att_group);
568 }
569
570 void SelectView(int newViewID)
571 {
572     if (gSelectedView != newViewID)
573     {
574         SPH_postRoot(gSelectedView, gSelectedView); /* Dims old view */
575         gSelectedView = newViewID;
576         HiliteSelectedView();
577         DrawSelectedParameter();
578     }
579 }
580
581     /*-----*/
582
583 void RedrawViewport(void)
584 {
585     struct timeb timestruct;
586     time_t time1, time2;
587     unsigned short milli1, milli2;
588     char buf[30];
589     srgp_point location = {450, 620};
590     double timendif;
591
592     /* start time measurement: */

```

```

593     ftime(&timestruct);
594     time1 = timestruct.time;
595     milli1 = timestruct.millitm;
596
597     SetUpProjection(gSelectedView);
598     SetUpView(gSelectedView);
599     HilitSelectedView();
600
601     /* end time measurement: */
602     ftime(&timestruct);
603     time2 = timestruct.time;
604     milli2 = timestruct.millitm;
605
606     timedif = (double)time2 + (double)milli2 / 1000.0 - ((double)time1 +
607     (double)milli1 / 1000.0);
608     sprintf(buf,"%5.2f frames per second", 1.0 / timedif);
609     drawStr(location, buf);
610 }
611
612 /*-----*/
613
614 void ChangeViewParamter(double factor)
615 {
616     switch(gSelectedParameter)
617     {
618         case ViewRefPtID:
619             gTheViewInfo[gSelectedView].viewRefPt[gSelectedCoordinate] += factor *
620             ParameterDelta;
621             SetUpViewReferenceCoordinateSystem(gSelectedView);
622             break;
623         case ViewRefNormaID:
624             gTheViewInfo[gSelectedView].viewPlaneNormal[gSelectedCoordinate] +=
625             factor * ParameterDelta;
626             SetUpViewReferenceCoordinateSystem(gSelectedView);
627             break;
628         case ViewUpVectorID:
629             gTheViewInfo[gSelectedView].viewUpVector[gSelectedCoordinate] += factor *
630             * ParameterDelta;
631             SetUpViewReferenceCoordinateSystem(gSelectedView);
632             break;
633         case ProjRefPtID:
634             gTheViewInfo[gSelectedView].projRefPt[gSelectedCoordinate] += factor *
635             ParameterDelta;
636             break;
637         case ViewPlaneWindowID:
638             switch (gSelectedCoordinate)
639             {
640                 case 0: gTheViewInfo[gSelectedView].viewPlaneWindow.uMin += factor *
641                 ParameterDelta;
642                 break;
643                 case 1: gTheViewInfo[gSelectedView].viewPlaneWindow.vMin += factor *
644                 ParameterDelta;
645                 break;
646                 case 2: gTheViewInfo[gSelectedView].viewPlaneWindow.uMax += factor *
647                 ParameterDelta;
648                 break;
649                 case 3: gTheViewInfo[gSelectedView].viewPlaneWindow.vMax += factor *
650                 ParameterDelta;
651                 break;
652             }

```

```

653         break;
654     case ViewPlaneWidnowSizeCenterID:
655         switch(gSelectedCoordinate)
656         {
657             case 0:
658                 gTheViewInfo[gSelectedView].viewPlaneWindow.uMin -= factor * ParameterDelta / 2.0;
659                 gTheViewInfo[gSelectedView].viewPlaneWindow.uMax += factor * ParameterDelta / 2.0;
660                 break;
661             case 1:
662                 gTheViewInfo[gSelectedView].viewPlaneWindow.vMin -= factor * ParameterDelta / 2.0;
663                 gTheViewInfo[gSelectedView].viewPlaneWindow.vMax += factor * ParameterDelta / 2.0;
664                 break;
665             case 2:
666                 gTheViewInfo[gSelectedView].viewPlaneWindow.uMin += factor * ParameterDelta;
667                 gTheViewInfo[gSelectedView].viewPlaneWindow.uMax += factor * ParameterDelta;
668                 break;
669             case 3:
670                 gTheViewInfo[gSelectedView].viewPlaneWindow.vMin += factor * ParameterDelta;
671                 gTheViewInfo[gSelectedView].viewPlaneWindow.vMax += factor * ParameterDelta;
672                 break;
673             case 4:
674                 gTheViewInfo[gSelectedView].viewPlaneWindow.uMin -= factor * ParameterDelta;
675                 gTheViewInfo[gSelectedView].viewPlaneWindow.uMax += factor * ParameterDelta;
676                 break;
677             case 5:
678                 gTheViewInfo[gSelectedView].viewPlaneWindow.vMin -= factor * ParameterDelta;
679                 gTheViewInfo[gSelectedView].viewPlaneWindow.vMax += factor * ParameterDelta;
680                 break;
681             }
682             break;
683         case ViewClippingPlanesID:
684             switch (gSelectedCoordinate)
685             {
686                 case 0: gTheViewInfo[gSelectedView].frontClippingPlaneDist += factor * ParameterDelta;
687                 break;
688                 case 1: gTheViewInfo[gSelectedView].backClippingPlaneDist += factor * ParameterDelta;
689                 break;
690             }
691             break;
692         case SurfaceParamsID:
693             switch (gSelectedCoordinate)
694             {
695                 case 0: if (factor>0) gXmin+=0.1; else gXmin-=0.1; break;
696                 case 1: if (factor>0) gXmax+=0.1; else gXmax-=0.1; break;
697                 case 2: if (factor>0) ++gM; else if (gM>2) --gM; break;
698                 case 3: if (factor>0) ++gN; else if (gN>3) --gN; break;
699             }
700             CreateSurfaceOfRevolution(gXmin, gXmax, gM, gN);
701             UpdateSurfaceStructure(gXmin, gXmax, gM, gN);
702             break;
703         }
704     }
705 }
706
707 SetUpProjection(gSelectedView);
708 SetUpView(gSelectedView);
709 HiliteSelectedView();
710 DrawCoordinates();
711
712

```

```

713  /*-----*/
714
715 void InitializeProjections(void)
716 {
717     int viewID;
718     double viewLeft, viewRight, viewBottom, viewTop;
719
720     for (viewID = 0; viewID <=3; viewID++)
721     {
722         SPH_defPoint(gTheViewInfo[viewID].viewRefPt,0,0,0);
723         SPH_defPoint(gTheViewInfo[viewID].viewPlaneNormal,0,0,1);
724         SPH_defPoint(gTheViewInfo[viewID].viewUpVector,0, 1,0);
725         SetUpViewReferenceCoordinateSystem(viewID);
726
727         /* View Window */
728         gTheViewInfo[viewID].viewPlaneWindow.uMin = kViewLeft;
729         gTheViewInfo[viewID].viewPlaneWindow.vMin = kViewBottom;
730         gTheViewInfo[viewID].viewPlaneWindow.uMax = kViewRight;
731         gTheViewInfo[viewID].viewPlaneWindow.vMax = kViewTop;
732
733         /* View Projection */
734         gTheViewInfo[viewID].projectionType = PERSPECTIVE;
735         SPH_defPoint(gTheViewInfo[viewID].projRefPt, 12.0, 6.0, 70.0);
736
737         gTheViewInfo[viewID].frontClippingPlaneDist = kViewFront;
738         gTheViewInfo[viewID].backClippingPlaneDist = kViewBack;
739
740         viewLeft = (viewID % 2) * 0.5;
741         viewBottom = 0.5 - (viewID / 2) * 0.5;
742         viewRight = viewLeft + 0.5;
743         viewTop = viewBottom + 0.5;
744
745         gTheViewInfo[viewID].stdViewVolume.xyzMin[0] = viewLeft;
746         gTheViewInfo[viewID].stdViewVolume.xyzMax[0] = viewRight;
747         gTheViewInfo[viewID].stdViewVolume.xyzMin[1] = viewBottom;
748         gTheViewInfo[viewID].stdViewVolume.xyzMax[1] = viewTop;
749         gTheViewInfo[viewID].stdViewVolume.xyzMin[2] = 0;
750         gTheViewInfo[viewID].stdViewVolume.xyzMax[2] = 0.5;
751
752         gTheViewInfo[viewID].stdViewClip.xyzMin[0] = viewLeft;
753         gTheViewInfo[viewID].stdViewClip.xyzMax[0] = viewRight;
754         gTheViewInfo[viewID].stdViewClip.xyzMin[1] = viewBottom;
755         gTheViewInfo[viewID].stdViewClip.xyzMax[1] = viewTop;
756         gTheViewInfo[viewID].stdViewClip.xyzMin[2] = 0;
757         gTheViewInfo[viewID].stdViewClip.xyzMax[2] = 1;
758
759         gTheViewInfo[viewID].viewBackgroundColor = kGreyColor + viewID;
760         gTheViewInfo[viewID].screenViewBounds.bottom_left.x = viewLeft;
761         gTheViewInfo[viewID].screenViewBounds.bottom_left.y = viewBottom;
762         gTheViewInfo[viewID].screenViewBounds.top_right.x = viewRight;
763         gTheViewInfo[viewID].screenViewBounds.top_right.y = viewTop;
764     }
765 }
766 /*-----*/
767
768 void DisplayInitialViews(void)
769 {
770     int viewID;
771
772

```

```

773     for (viewID = 0; viewID <= 3; viewID++)
774     {
775         SetUpProjection(viewID);
776         SetUpView(viewID);
777     }
778
779     for (viewID = 0; viewID <= 3; viewID++)
780         SPH_postRoot(kSurfaceID, viewID);
781     gSelectedView = 0;
782     HiliteSelectedView();
783 }
784
785 /*-----*/
786
787 void CreateControlButtons(void)
788 /* Creates the buttons to control the viewing of the 3D object.*/
789 {
790     int buttonID;
791     rectangle buttonRect;
792
793     SRGP_loadFont(kButtonFont,gFontNames[kButtonFont]);
794     SPH_initializeButtons();
795
796     SRGP_setLineWidth(1);
797
798     /* First row buttons */
799     buttonRect.bottom_left.x = gScreenViewRect.top_right.x -
800         quitButtonRightOffset;
801     buttonRect.bottom_left.y = gScreenViewRect.top_right.y -
802         stdButtonHeight - stdMargin,
803     buttonRect.top_right.x = buttonRect.bottom_left.x + stdButtonWidth;
804     buttonRect.top_right.y = buttonRect.bottom_left.y + stdButtonHeight;
805     SPH_createButton(QuitButtonID, buttonRect, "Quit\0",
806                     kBlackColor, kRedColor,kButtonFont, TRUE);
807
808     buttonRect.bottom_left.x = gScreenViewRect.bottom_left.x + stdMargin;
809     buttonRect.top_right.x = buttonRect.bottom_left.x + wideButtonWidth;
810     SPH_createButton(RedrawButtonID, buttonRect, "Measure Time\0",
811                      kBlackColor, kGreyColor,kButtonFont, TRUE);
812
813     buttonRect.bottom_left.x = buttonRect.top_right.x + stdButtonGap;
814     buttonRect.top_right.x = buttonRect.bottom_left.x + stdButtonWidth;
815     SPH_createButton(IncButtonID, buttonRect, "+\0",
816                      kBlackColor, kGreenColor,kButtonFont, TRUE);
817
818     buttonRect.bottom_left.x = buttonRect.top_right.x + stdButtonGap;
819     buttonRect.top_right.x = buttonRect.bottom_left.x + stdButtonWidth;
820     SPH_createButton(DecButtonID, buttonRect, "-\0",
821                      kBlackColor, kGreenColor,kButtonFont, TRUE);
822
823     /* Second row buttons */
824
825     buttonRect.bottom_left.x = gScreenViewRect.bottom_left.x + stdMargin;
826     buttonRect.bottom_left.y = gScreenViewRect.top_right.y -
827         2*(stdButtonHeight + stdMargin);
828     buttonRect.top_right.x = buttonRect.bottom_left.x + wideButtonWidth;
829     buttonRect.top_right.y = buttonRect.bottom_left.y + stdButtonHeight;
830     SPH_createButton(ParameterButtonID, buttonRect, "Next Parameter\0",
831                      kBlackColor, kLavenderColor,kButtonFont, TRUE);
832     gParameterTextRect = SRGP_defRectangle(buttonRect.bottom_left.x,

```

```

833                                buttonRect.bottom_left.y - stdButtonHeight,
834                                buttonRect.top_right.x + 10,
835                                buttonRect.top_right.y - stdButtonHeight -
836                                2);
837        for (buttonID = FirstCoordinateButtonID;
838             buttonID <= FourthCoordinateButtonID; buttonID++)
839        {
840            buttonRect.bottom_left.x = buttonRect.top_right.x + stdButtonGap;
841            buttonRect.top_right.x = buttonRect.bottom_left.x + stdButtonWidth;
842            SPH_createButton( buttonID, buttonRect,
843                CoordinateNames[ ProjRefPtID ][ buttonID -
844                    FirstCoordinateButtonID ],
845                kBlackColor, kGreenColor,kButtonFont, buttonID!=3 );
846            gCoordinateOutputRects[ buttonID-FirstCoordinateButtonID ] =
847            SRGP_defRectangle( buttonRect.bottom_left.x,
848                buttonRect.bottom_left.y -
849                stdButtonHeight,
850                buttonRect.top_right.x,
851                buttonRect.top_right.y -
852                stdButtonHeight - stdMargin );
853        }
854
855        UpdateButtonTitles(ProjRefPtID); /* This ensures the correct display of the
856 buttons */
857    }
858
859 /*-----*/
860
861 void InitializeProgram(void)
862 {
863     NDC_rectangle      tempRect;
864     int next;
865
866     SPH_begin(kScreenWidthWidth, kScreenWidthHeight,kNumberOfPlanes, 0);
867     SPH_setDoubleBufferingFlag(FALSE);
868     SPH_setImplicitRegenerationMode(ALLOWED);
869     SRGP_tracing(FALSE);
870     SRGP_disableDebugAids();
871     SPH_loadCommonColor(kRedColor,"red");
872     SPH_loadCommonColor(kBlueColor,"blue");
873     SPH_loadCommonColor(kGreenColor,"green");
874     SPH_loadCommonColor(kGreyColor,"gray");
875     SPH_loadCommonColor(kLavenderColor,"lavender");
876     SPH_loadCommonColor(kTanColor,"tan");
877     SPH_loadCommonColor(kPinkColor,"pink");
878
879     SPH_loadFont(kStdFont,gFontNames[kStdFont]);
880     SPH_loadFont(kDefaultFont,gFontNames[kDefaultFont]);
881
882     gScreenViewRect.bottom_left.x = 0;
883     gScreenViewRect.bottom_left.y = kScreenWidthHeight -
884                                         kScreenWidthWidth ;
885     gScreenViewRect.top_right.x = kScreenWidthWidth;
886     gScreenViewRect.top_right.y = kScreenWidthHeight;
887
888     CreateControlButtons();
889
890     SPH_setInputMode(LOCATOR, EVENT);
891     SPH_setInputMode(KEYBOARD, EVENT);
892     SPH_setKeyboardProcessingMode(RAW);

```

```

893     CreateAxes(kDefautAxisLength);
894     CreateSurfaceOfRevolution(gXmin, gXmax, gM, gN);
895     CreateSurfaceStructure(gXmin, gXmax, gM, gN);
896     InitializeProjections();
897     DisplayInitialViews();
898     DrawSelectedParameter();
899 }
900 }
901 /*-----*/
902 void CloseDown (int theLastButton)
903 {
904     int nextButton;
905
906     free(gTri);
907     free(gVertices);
908     for (nextButton = 0; nextButton <= theLastButton; nextButton++)
909         SPH_DisposeButton(nextButton);
910
911     SPH_end();
912 }
913 /*-----*/
914 void DoButton (int theButtonID)
915 {
916     SRG_P_setLineWidth(1);
917     switch (theButtonID) {
918         case QuitButtonID:
919             break;
920         case RedrawButtonID:
921             RedrawViewport();
922             break;
923         case ParameterButtonID:
924             CycleToNextParameter();
925             break;
926         case IncButtonID:
927             ChangeViewParamter(1.0);
928             break;
929         case DecButtonID:
930             ChangeViewParamter(-1.0);
931             break;
932         case FirstCoordinateButtonID:
933             SelectCoordinate(0);
934             break;
935         case SecondCoordinateButtonID:
936             SelectCoordinate(1);
937             break;
938         case ThirdCoordinateButtonID:
939             SelectCoordinate(2);
940             break;
941         case FourthCoordinateButtonID:
942             SelectCoordinate(3);
943             break;
944         default:
945             break;
946     }
947 }
948
949
950
951
952

```

```

953 /*-----*/
954
955 void DoViewClick (point where)
956 {
957     if (pt3DinRect(where, gTheViewInfo[0].screenViewBounds))
958         SelectView(0);
959     else if (pt3DinRect(where, gTheViewInfo[1].screenViewBounds))
960         SelectView(1);
961     else if (pt3DinRect(where, gTheViewInfo[2].screenViewBounds))
962         SelectView(2);
963     else if (pt3DinRect(where, gTheViewInfo[3].screenViewBounds))
964         SelectView(3);
965     else
966         SPH_beep();
967 }
968 /*-----*/
969
970 void EventLoop(void)
971 {
972     inputDevice      theInputDevice;
973     char theInput[MaxStringLength+1];
974     locator_measure theLocMeasure;
975     short theButtonID;
976     boolean quitting;
977     srgp_point pos2d;    /* position in 2D */
978
979     quitting = FALSE;
980     do {
981         theInputDevice = SPH_waitEvent(0);
982         switch (theInputDevice) {
983
984             case KEYBOARD:
985                 SPH_getKeyboard(theInput,MaxStringLength);
986                 quitting = (theInput[0] == 'q') || (theInput[0] == 'Q');
987                 break;
988
989             case LOCATOR:
990                 SPH_getLocator(&theLocMeasure);
991                 SRG_P_setLineWidth(1);
992                 pos2d.x = (double)kScreenWidthWidth * theLocMeasure.position[0];
993                 pos2d.y = (double)kScreenWidthWidth * theLocMeasure.position[1];
994                 if (SPH_findButton(pos2d, &theButtonID) )
995                 {
996                     if (SPH_trackButton(theButtonID) )
997                     {
998                         if (theButtonID == QuitButtonID)
999                             quitting = TRUE;
1000                         else
1001                             DoButton(theButtonID);
1002                     }
1003                 }
1004             else if ( (theLocMeasure.button_chord[LEFT_BUTTON] == DOWN) )
1005                 DoViewClick(theLocMeasure.position);
1006             break;
1007
1008             default: /*Do nothing*/
1009                 break;
1010             }
1011     } while (!quitting);

```

```

1013 }
1014 /*-----
1015 /* Surface of Revolution Routines */
1016 /*-----*/
1017
1018 double f(double x)
1019 {
1020     if (1-x*x < 0.01) return(0.01);
1021     else return(sqrt(1-x*x));
1022 }
1023
1024 /*-----*/
1025
1026 void CreateSurfaceOfRevolution(double xMin, double xMax, int m, int n)
1027 {
1028     int d,x;           /* loop counters */
1029     double xpos, dpos; /* current vertex position in world coordinates */
1030     double xstep, dstep; /* stepsizes */
1031     int index;
1032     int base;
1033
1034     if (gVertices!=NULL) free(gVertices);
1035     if (gTri!=NULL) free(gTri);
1036
1037     gVertices = (point*)malloc(m * n * sizeof(point));
1038     if (gVertices==NULL) exit(-1);
1039
1040     xstep = DX / (double)(m-1);
1041     dstep = 2.0 * M_PI / (double)n; /* step in radians */
1042
1043     index = 0;
1044
1045     for (x=0; x<m; ++x)
1046     {
1047         xpos = xMax - (double)(m-x-1) * xstep;
1048         for (d=0; d<n; ++d)
1049         {
1050             dpos = (double)d * dstep;
1051             SPH_defPoint(gVertices[index++], xpos, f(xpos) * cos(dpos), f(xpos) *
1052 sin(dpos));
1053         }
1054     }
1055
1056     gNumberOfTris = 2 * n * (m-1);
1057
1058     /* compose vertices to triangles:*/
1059     gTri = (vertex_index*)malloc(gNumberOfTris * 4 * sizeof(vertex_index));
1060     if (gTri==NULL) exit(-1);
1061
1062     index = 0;
1063     for (x=0; x<m-1; ++x)
1064     {
1065         for (d=0; d<n; ++d)
1066         {
1067             base = x * n;
1068
1069             /* compose upper left triangle: */
1070             gTri[index++] = base + d;
1071             gTri[index++] = base + ((d + 1) % n);
1072

```

```

1073     gTri[index++] = base + d + n;
1074     gTri[index++] = -1; /* face end marker */
1075
1076     /* compose lower right triangle: */
1077     gTri[index++] = base + d + n;
1078     gTri[index++] = base + ((d + 1) % n);
1079     gTri[index++] = base + ((d + 1) % n) + n;
1080     gTri[index++] = -1;
1081 }
1082 }
1083 printf("Number of Triangles: %d\n", gNumberOfTris);
1084 }
1085
1086 /*-----*/
1087
1088 void UpdateSurfaceStructure(double xMin, double xMax, int m, int n)
1089 {
1090     SPH_openStructure(kSurfaceID);
1091     SPH_setElementPointer(0);
1092     SPH_moveElementPointerToLabel(0);
1093     SPH_offsetElementPointer(1);
1094     SPH_deleteElement();
1095     SPH_polyhedron(m * n, 2 * n * (m-1), gVertices, gTri);
1096     SPH_closeStructure();
1097 }
1098
1099 /*-----*/
1100
1101 void CreateSurfaceStructure(double xMin, double xMax, int m, int n)
1102 {
1103     SPH_openStructure(kSurfaceID);
1104     /* first add axes */
1105     SPH_setTextFont(kStdFont);
1106     SPH_setLineColor(kRedColor);
1107     SPH_polyLine(2,gXAxis);
1108     SPH_text(gXAxis[2],"x\0");
1109     SPH_setLineColor(kGreenColor);
1110     SPH_polyLine(2,gYAxis);
1111     SPH_text(gYAxis[2],"y\0");
1112     SPH_setLineColor(kBlueColor);
1113     SPH_polyLine(2,gZAxis);
1114     SPH_text(gZAxis[2],"z\0");
1115
1116     SPH_setInteriorColor(kWhiteColor);
1117     SPH_label(0);
1118     SPH_polyhedron(m * n, 2 * n * (m-1), gVertices, gTri);
1119     SPH_closeStructure();
1120 }
1121
1122 /*-----*/
1123 /* Main Program */
1124 /*-----*/
1125
1126 void main (void)
1127 {
1128     InitializeProgram();
1129     EventLoop();
1130     CloseDown(kLastButtonID);
1131 }
1132

```

```
1133 /*-----*/  
1134 /* The End */  
1135 /*-----*/
```

7 Curriculum Vitae

Name:	Jürgen Schulze-Döbold
Date of birth:	December 16, 1973
Place of birth:	Dortmund
High school graduation:	1993 from Geschwister-Scholl-Gymnasium, Stuttgart (Germany)
Undergraduate studies:	1995 undergraduate degree ("Vordiplom") at the University of Stuttgart (Germany) in computer science
Employment record:	1993 (3 months): internship at Fraunhofer Institute of Production Technology and Automation (IPA) Stuttgart
	1994 (10 weeks): internship at Mercedes-Benz, Stuttgart
	1993-1994 (12 months): research assistant at IPA
	since 1994: Programming of multi media applications