Fast Shading Algorithms

Björn Nystedt

Marko Pirttioja

Department of Mathematics, Natural- and Computer Sciences
University of Gävle,

Kungsbäcksvägen 47
S-801 76 Gävle, Sweden

E-mail
na99bnt@student.hig.se
na98mpa@student.hig.se

7th June 2001

Abstract

One main concern when one is producing three-dimensional computer graphics is the trade-off between realism and speed. One wants to incorporate more physics principles into three-dimensional graphics algorithms to better simulate complex interactions between objects and the lightning environment. The calculation of how light interacts with surfaces is a technique called shading and the aim of this paper is to implement and compare the efficiency of different shading algorithms. The different algorithms are timed and compared to each other in order to see which is the fastest one. All algorithms use the Phong illumination model for light-material interaction calculations. The important result of the test is that both Fast Phong shading and Reflection shading is faster then Phong shading, especially when large polygons are used. Both algorithms are variants of the Phong shading algorithm and we could not spot any differences in the picture they produced. However they becomes slower when the polygons becomes small.

Keywords: Shading, Illumination, Algorithm, Geometric, Graphics

Contents

1 Introduction

Since three-dimensional graphics was introduced in computer graphics one of the main problems has been to produce, realistic pictures fast, preferably in real-time. It is the calculation of how light interacts with surfaces, which is very time consuming. The technique is called shading.

One can divide shading into two groups, polygon shading and environment shading. Environment shading produces by far the best result but is very time consuming. Polygon shading, further only referred as shading, on the other hand can produce much faster shading with a very convincing result.

Now when computers has become much faster then they were before and with the use of 3D hardware accelerators one can use more and more complex shading algorithms to produce three-dimensional pictures in real-time or almost in real-time. Something that once used to take hours, could now be done in real-time.

There are several shading algorithms available, with variations in realism of the image they produce and calculation speed. The most common shading algorithms are Flat shading [Angel00], Gouraud shading [Gouraud71] and Phong shading [Phong75]. Of them Phong shading gives the best result and is also the most time consuming. Faster methods are Fast Phong shading [Duff79], Reflection shading [HastSEG01] and Bishop's shading method [Bishop86], which are different ways of doing Phong shading. Bishop method is very hard to implement and it is difficult to find a good explanation of the algorithm.

1.1 Problem Definition

There has been a lot of research on developing algorithms for realistic and fast shading. Some algorithms are faster and some produce better results. It is the trade-off between the speed at which a picture could be produced and the realism of its picture. There have also been made improvements on excisting algorithms.

Many shading algorithms that before were considered to slow for producing real-time pictures are now or in a near future on the verge of doing it.

It has as mentioned above been a lot of research on producing faster shading algorithms, but there has not been, to our knowledge any complete comparison between all existing shading algorithms, especially not among the newer ones.

1.2 Aim

Our aim of this paper is to compare the speed of different shading algorithms and produce data that could be used for choosing method when implementing fast or real-time shaders, today or in the future.

Another goal for us with the paper is to learn and show how to implement the different shading algorithms.

1.3 Questions at Issue

The issue of the paper is very simple, it is only a matter of clocking the speed of the different shading algorithm and compare their times to each other. Then we should draw conclusions on how good the algorithms are for fast shading. The picture that the algorithms produce should also be comment.

1.4 Hypotheses

Our hypothesis is that if one uses approximations of a curve instead of calculating all the interpolated normals, faster shading can be achieved. Then we get a square root calculation less per pixel.

1.5 Expected Results

We expect that the more complex shading algorithms to be faster than the simpler ones. The more complex algorithms are Fast Phong- and Reflection shading. The setup, pre-calculations are rather large for these methods. It will effect the speed according to the size of the polygons. Smaller polygons need less calculation than larger ones, so the setup calculations for small polygons can take unnecessary time. It is because smaller polygons have shorter scan lines and the setup is calculated for every scan line.

In other words, we think that the Fast Phong- and Reflection shading should be faster for normal and larger polygons. Reflection shading is expected to be faster than Fast Phong shading.

2 Theoretical Background

The theoretical background that is of importance for this paper. We will introduce brief- and more advanced theoris for three-dimensional graphics in this section.

2.1 Brief Introduction to Three Dimensional Graphics

A brief introduction to three dimensional graphic that can serve as a reminder or an introduction, different opperations on vector and theoris about light.

2.1.1 Polygons, Triangles and Vertices

To be able to model three-dimensional objects in an efficient way in computers one use polygons. A polygon is built up by vertices, it's corners. Each vertex has three coordinates, x, y and z, which give its position in three-dimensional space.

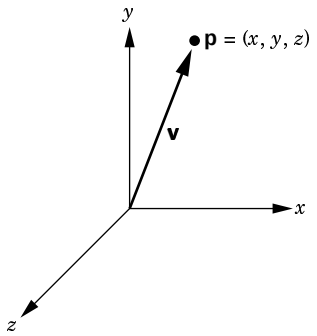


Figure 1: Axlar x,y,z vertice p vector v

The calculations becomes easier if polygons that only uses three vertices are used, triangles.

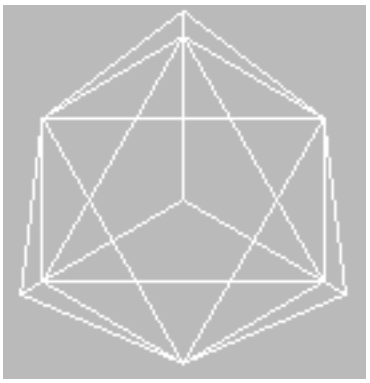


Figure 2: Example of an approximation of a sphere with 16 vertices.

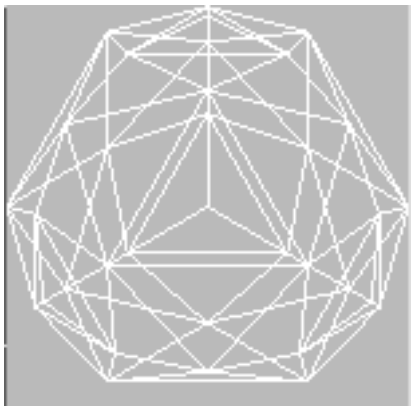


Figure 3: Example of an approximation of a sphere with 48 vertices.

2.1.2 Vectors and Vector Calculations

Vectors are often used for calculations in 3D graphics. The difference between vertices / points and vectors is that a vector is the difference between two points and therefore has a direction.

There are operations for addition, subtraction and multiplication with a scalar and so on for vectors. Two other very use full calculations on vectors are the scalar product and the cross product.

2.1.3 The Scalar Product

The scalar product, also called the dot product is calculated:

$$\mathbf{u} \cdot \mathbf{v} = \parallel \mathbf{u} \parallel \parallel \mathbf{v} \parallel \cdot cos\alpha$$

2.1.4 The Cross Product

The cross product of two vectors is a vector in a direction perpendicular to the two original vectors.

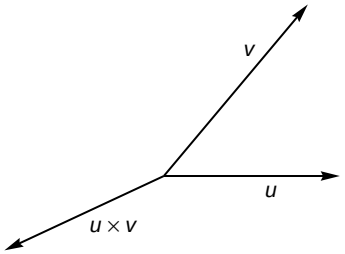


Figure 4: The cross product

If $\mathbf{u}$ has componets u_1, u_2, u_3 and $\mathbf{v}$ has componets v_1, v_2, v_3, the crossproduct is calculated

$$\mathbf{u}\ x\ \mathbf{v} = [u_2v_3 - u_3v_2, u_3v_1 - u_1v_3, u_1v_2 - u_2v_1]$$

The absolute value of the cross product gives the area of a parallelogram built by two vectors. The sign of the cross product is very useful, because with it one sees on which side one point is according to two others.

2.1.5 Calculate vertex normal

Multiple polygons meet at interior vertices. The normal at the vertex can be defined in a way to achieve smoother shading through interpolation. The interior vertex, each with its own normal, is defined as the average of the normals of the polygons that share the vertex.

Equation for the normal at a point:

$$\mathbf{n} = \frac{\mathbf{n_1} + \mathbf{n_2} + \mathbf{n_3} + \mathbf{n_4}}{\parallel \mathbf{n_1} + \mathbf{n_2} + \mathbf{n_3} + \mathbf{n_4} \parallel}$$

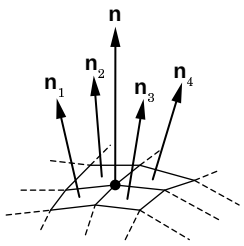


Figure 5: Interior vertex

2.1.6 Projections and Frame Buffer

If three-dimensional objects are to be displayed on a screen they must first be transferred into two-dimensional objects, this process is called projection. Perspective or parallel projection can be used. A special case of parallel projection is orthogonal or orthographic projection; it has no visible depth. Orthogonal projection is very simple one only has to get rid of the z value. It is the fastest projection algorithm.

During the projection process, the two-dimensional picture is stored in a buffer called frame buffer. The frame buffer has the same size as the screen or a window and colors are stored in it for each pixel.

2.1.7 Shading

Shading is a technique used for making objects in two-dimensional look as they are three-dimensional. It is done by positioning a light source that is directed towards the object and cast shades on the object.

Figure 6: Example of a shaded sphere

There are different shading techniques available, that produces a varying convincing result. An example of that could be the use of highlights, that makes the three dimensional appearance more convincing.

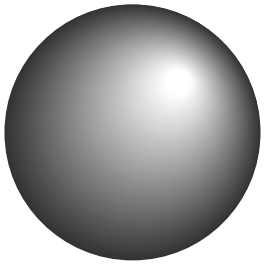


Figure 7: Example of a sphere with highlight

2.1.8 Light Source

In order to calculate the shading effects one must take in to account differences of lights. Light can leave a surface through two fundamental processes: self-emission and reflection. Light sources are self-emitted. Light can be divided into four groups: ambient light, point sources, spotlights and distant light sources.

2.1.9 Ambient Light

Ambient light can also be described as background light. Ambient light is the result of inter reflections between objects, light that "always" is in the room and is not directly coming from a light source. It is like the light in the forest or in a classroom.

2.1.10 Point Sources

An example of a point source in the real world could be a streetlight. An ideal point source emits light equally in all direction and is positioned at a fixed point in the space.

2.1.11 Spotlights

Spotlights are lights that emit light in a narrow range of angles. It is like a desk lamp or car lights.

2.1.12 Distant Light Source

If the light source is moved far away from the object its light rays become almost parallel. An example of a distant light source would be the sun shining on a house. If distant light sources are used in the shading model, the calculations become much easier.

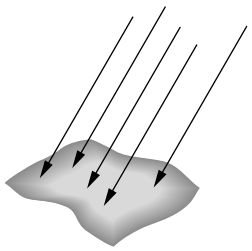


Figure 8: Parallel light rays

2.1.13 Material

It is not only the type of light that is of importance when the shading model is calculated, so is also the material.

2.2 Important Theories

There are important theories that one should know about before implementing shading algorthms.

2.2.1 The Phong Reflection Model

Shading models apply some reflection model to approximate the complex physics of light-material interaction in (physical 3D world) models. One model that leads to efficient computations is the model that was introduced by Phong, the Phong illumination model [Phong75]. His model supports three types of material-light interactions, specular, ambient and diffuse.

The Phong illumination model uses four vectors to calculate a color for a random point on a surface. The vector **n** is the normal vector at the point p, the vector **v** is the direction from the point p to the viewer, the vector l is the direction from point p to a light source and vector **r** is the direction that a perfectly reflected ray from l would make.

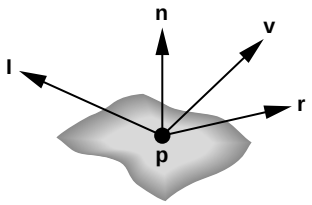


Figure 9: Vectors used by the Phong model

The Phong equation as mentioned above supports three types of material-light interactions, specular, ambient and diffuse, so the equation can be

divided into three parts.

2.2.2 Ambient reflection

The intensity of the ambient light L_a is the same at every point on the model. This light is reflected or absorbed by the surface of the model. The amount of reflection and absorption is given by the ambient reflection coefficient k_a. Because only positive fractions of lights is reflected k_a must have a value between 0 and 1.

The equation for ambient light is:

$$I_a = k_a L_a$$

2.2.3 Diffuse Reflection

The diffuse reflection depends both on the material of the surface and the direction of the light. One can see diffuse reflections as reflections on a rough surface. A perfect diffuse reflection scatters the lights in all directions.

The equation for diffuse reflection is:

$$I_d = \frac{1}{a + bd + cd^2} k_d L_d (\mathbf{l} \cdot \mathbf{n})$$

$\mathbf{l}$ is the direction vector of the light source and $\mathbf{n}$ is the normal for the point. L_d is the intensity of the light and k_d is the material coefficient. The rest of the equation is for the distance of the light source.

2.2.4 Specular Reflectio

When objects are shaded with only ambient and diffuse light, they appear a bit dull, it could be fixed by adding specular reflection. Specular reflection appears like highlights and is also called that.

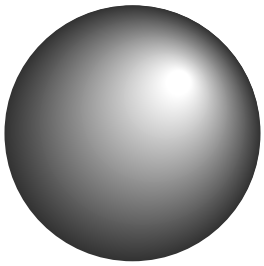


Figure 10: Specular highlights on a sphere

The equation for specular light is:

$$I_s = \frac{1}{a + bd + cd^2} k_s L_s (\mathbf{r} \cdot \mathbf{v})^\alpha$$

k_s is the material coefficient, L_s the intensity of the light, $\mathbf{r}$ is the direction of a perfectly reflected ray from the light source, $\mathbf{v}$ is the vector to the viewer and α is a shininess coefficient. The rest of the equation is for the distance of the light source.

2.2.5 Phong Equation

The whole Phong illumination equation model is written:

$$I = \frac{1}{a + bd + cd^2} (k_d L_d (\mathbf{l} \cdot \mathbf{n}) + k_s L_s (\mathbf{r} \cdot \mathbf{v})^\alpha) + k_a L_a$$

If one uses a distant light source the equation can be simplified to:

$$I = (k_d L_d (\mathbf{l} \cdot \mathbf{n}) + k_s L_s (\mathbf{r} \cdot \mathbf{v})^\alpha) + k_a L_a$$

Future simplifications can be done by not dealing with the material and light coefficients.

2.2.6 Flat Shading

Flat shading is the fastest shading algorithm but, its result is by far the least appealing of the algorithms that are available. Flat shading assume that we have the normal of the polygon. The normal can be given by the data structure of the model or can be calculated. By using the normal in the Phong illumination model, we get the color for the whole polygon.

Figure 11: A flat shaded sphere

2.2.7 Interpolative and Gouraud Shading

Interpolative and Gouraud shading [Gouraud71] takes the colors at the vertices of the polygon, resulting from the illumination model, and interpolate these colors across the edges and across the scan lines.

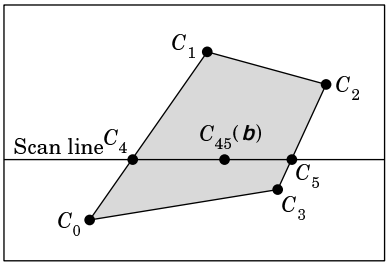


Figure 12: Scan line

2.2.8 Phong Shading

Phong shading [Phong75] will take the normals at the vertices of the polygon and interpolate these across the edges of the polygon and across the scan lines. Then the color is calculated by applying the Phong illumination model to each point on the scan line. This method will need more calculations then Gouraud, but it will produce a more accurate shading of the polygon, since the illumination model is applied to every point on the polygon, instead of interpolating the colors at the vertices.

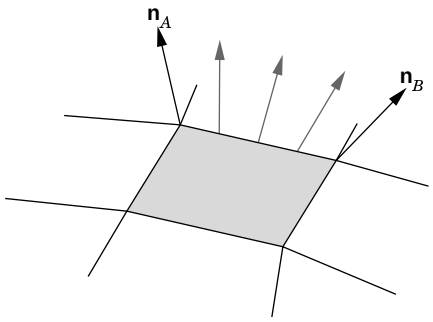


Figure 13: Edge normals

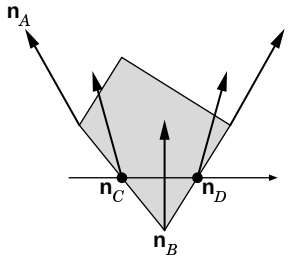


Figure 14: Interpolation of normals in Phong shading

2.2.9 Fast Phong Shading, Duff

Duff shading [Duff79] is a different way of doing Phong shading. The method will take the normals at the vertices of the polygon and interpolate these across the edges of the polygon and across the scan lines. Then the color is calculated for each point by calculating $I_d = p/\sqrt{q}$. We need to calculate setup data where $\mathbf{n} \cdot \mathbf{l} = Ax + B = p$, $\mathbf{n} \cdot \mathbf{n} = Cx^2 + Dx + E = q$, $A = \mathbf{k} \cdot \mathbf{l}$, $B = \mathbf{m} \cdot \mathbf{l}$, $C = \mathbf{k}^2$, $D = 2(\mathbf{k} \cdot \mathbf{m})$, $E = \mathbf{m}^2$ $\mathbf{k} = \frac{\mathbf{n_2} - \mathbf{n_1}}{\mathbf{n}}$, $\mathbf{n} = x_2 - x_1$ and $\mathbf{m} = \mathbf{n_1}$. $\mathbf{n_1}$, $\mathbf{n_2}$ are the normals at the edges and $\mathbf{l}$ is the light source direction in an unit vector.

The whole formula:

$$\frac{\mathbf{n} \cdot \mathbf{l}}{\| \mathbf{n} \|} = \frac{Ax + B}{\sqrt{Cx^2 + Dx + E}} = \frac{p}{\sqrt{q}}$$

This values are calculated by recurrence as follows:

$p_{i+1} = p_i + dp_i$
$q_{i+1} = q_i + dq_i$
$dq_{i+1} = dq_i + d^2q$

where $p_0 = B$, $dp = A$, $q_0 = E = 1$, $dq_0 = C + D$ and $d^2q = 2C$.

The recurrence is evaluated in the inner loop for the scan line, along with $I_d = p/\sqrt{q}$, which is the diffuse light intensity for the pixel. The setup data A, B, C, D, p_0, q_0, dq_0 and d^2q must be calculated for each new scan line.

2.2.10 Reflection shading

Reflection shading [HastSEG01] works almost like Duff but with different setup data which makes the calculation of the color easier, we get rid of one square root calculation in the inner loop. The color intensity is $I_d = p/q$ where $\mathbf{n} \cdot \mathbf{l} = Ax + B = p$, $\mathbf{h} \cdot \mathbf{h} = Dx^2 + Ex + F = q$, $D = A - GD$, $E = B - GE$, $F = C - GF$ and $G = \mathbf{n} \cdot \mathbf{l}$.

The whole fomula:

$$\frac{\mathbf{n} \cdot \mathbf{l}}{\| \mathbf{n} \|} = \frac{Ax + B}{(A - GD)x^2 + (B - GE)x + (C - CF)} = \frac{p}{q}$$

This values are calculated by recurrence as follows:

$p_{i+1} = p_i + dp_i$
$q_{i+1} = q_i + dq_i$
$dq_{i+1} = dq_i + d^2q$

where $p_0 = C$, $dp = A + B$, $q_0 = F$, $dq_0 = D + E$ and $d^2q = 2D$.

The recurrence is evaluated in the inner loop for the scan line, along with $I_d = p/q$, which is the diffuse light intensity for the pixel. The setup data A, B, C, D, E, F, G, p_0, q_0, dq_0 and d^2q must be calculated for each new

scan line.

2.2.11 Hidden surface removal

Once we have got lines, points, line segments and polygons we have to determine which of them that should be displayed. Some of these may hide others. The hidden surface removal problem must be solved. There are two main methods, the object space approach and the image space approach.

Object space approach An object space method compares objects and part of objects to each other to determine which surfaces, as a whole, we should label as visible.

Image space approach In an image space algorithm visibility is decided point by point at each pixel position on the projection plane.

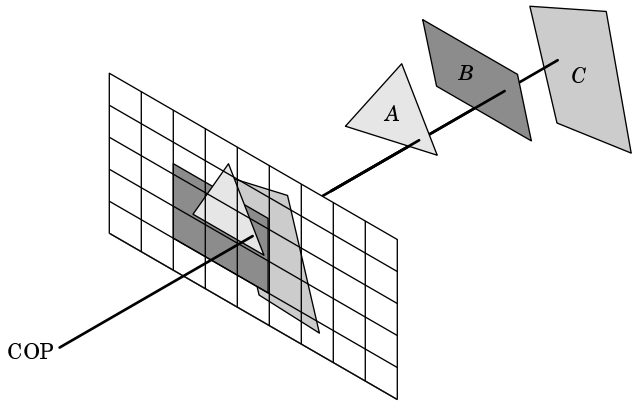


Figure 15: Image space hidden surface removal

2.2.12 Back-Face Culling

One can reduce the work required for hidden surface removal by eliminating all back-face polygons before one apply any other hidden surface removal algorithm. For a solid 3D model one can see the front of a polygon if the polygon's normal is directed towards the viewer. So polygons with normals directed into the screen can be ignored when the scan conversion is done.

2.2.13 Z-buffer algorithm

The z-buffer algorithm is a direct implementation of the image space approach, it loops over the polygons, rather than over the pixels and therefore

it can be a part of the scan conversion process.

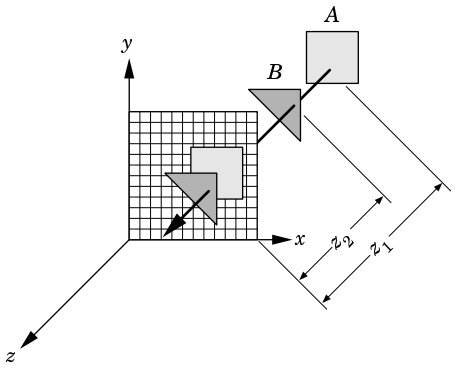


Figure 16: Z-buffer algorithm

The z-buffer is initialised to zero, which is represented by the back clipping planes z value, and the frame buffer is initialised to the background color. For each point of intersection between a ray that leaves the centre of projection, COP and a pixel one can calculate the color for it by using a shading model, but one must also determine if this point is visible or not. It will be visible if it is the closest intersection along the ray.

In addition to the frame buffer, one needs a z-buffer, with the same resolution as the frame buffer where one stores the depth information, the z value. During the scan conversion one updates the z-buffer and the color in the frame buffer, if the point's z value is higher than the value stored in the z-buffer. If not, one do nothing.

2.2.14 Scan Conversion

A technique to fill a polygon with a color is called scan conversion. There are several methods for travelling between two points on a polygon. One method uses the k-value, which is $\frac{(y2-y1)}{(x2-x1)}$ where $(x1, y1)$ and$(x2, y2)$ are the points, for travelling between two points. For each new point on the line between the points, one increases the k-value to step forward. Another method used for travel between to point, or along a line, uses the formula $p(\alpha) = (1-\alpha)p_1 + \alpha p_2$ where p_1 is the start point and p_2 the end point. To travel from the start- to the end point the alfa value is increased from 0 to 1 in equal steps.

3 Method

Methods that one should use when one obtain information and analyse results.

3.1 Choice of Method

In order to obtain information of shading algorithms and other theories of importance for implementation, two main methods will be used. A thorough study of books, previous research papers, and articles within polygon shading. And an empirical analysis of the results will also be done.

3.2 Description of Method

The literature study is unavoidable in order to achieve a solid theoretical background, which is needed to construct the shading algorithms. The tests and analysis of the results from the tests is a very important part so the hypotheses can be verified or declined.

4 System Design

The design of shading algorithms can be divided in two main steps. The first step is the designing of the algorithms, which have the theoretical part as its important part. In order to be able to compare the speed and result of the different shading algorithms they have to be implemented, the second part. The other part is where theories become reality and are tested.

The comparison could be done in theory but it would be difficult. The implementation is very time consuming, because in addition to the core shading algorithms we have to make additional programming for scanning polygons, back face removal, loading and producing models and shell program. Another issue is how we should time the different algorithms.

4.1 Design and Implementation of Main Program

To be able to implement and time the different shading algorithms, we have to make a shell program. The shell program should be able to project three-dimensional models on the screen that are represented in a data structure. In other words the program should be able to show shaded 3D models on the computer display. We use orthogonal projection.

Such a program should at least consists of these major parts: an internal data representation for three dimensional objects, a frame buffer that stores the picture yet to be shown, a function for showing the picture stored in the frame buffer, some kind of hidden surface removal and scan conversion functions that scans the objects to the frame buffer. It is also good if the program has a loader for 3D objects. Another good idea is to have a graphic user interface.

Another issue is how we should time the different shading algorithms. We also have to choose work platform and development language for our program.

One of our goals with the program is that it should be fast and readable, finding a balance for the trade-off between speed and readability.

The main program is called **PolLac** (Polygon Lacquer).

4.1.1 Platform and Language

We chose to develop our program on the UNIX platform and use C as our development language. The reason for why we chose UNIX is because it is used at our high school and C is a simple and fast language. Although we used a Ms Windows system for the timing of the algorithms, why we did so is decried in section *Timing the Algorithms*. We considered C++, but we had questions if uses of classes could make the timing process of the algorithms difficult.

4.1.2 Data representation

For representation of vertices, we made a vector structure, called Vector. It was implemented in a header file that consists of functions for vectors. The functions are vector addition, subtraction, multiplication with scalar, dot product and so on.

The vertices of the models are stored in a structure called point, which consists of a Vector and storage for a color and a normal. In similar way we have a structure for triangles data called face that store a color and a normal.

We made a frame buffer of points, with the size of a display window and we have a z-buffer.

4.1.3 Producing the Picture on the Screen

In order to be able to produce pictures from our frame buffer on the screen, one must use a interface between our application program and the graphic system. Such a program is called an application programmer's interface, API. We chose OpenGL as our API, because we were familiar with it.

With OpenGL and an extension to it called GLUT, one can create windows, show pixels in it and create menus. OpenGL is a lot more power full that that, it is almost a complete API for making real time 3D programs, but that are we not interested of for this work. We only use it for setting pixels.

4.1.4 Hidden Surface Removal

We chose to use the z-buffer algorithm as our hidden surface removal algorithm. As mentioned above we have a z-buffer, in which we store corresponding z values to the frame buffer. Values are stored in the z-buffer while we do the scan conversion. The z-buffer is initialized to 0.0. Before the z-buffering we do back-face culling.

4.1.5 Scan Conversion

For the three-dimensional object we scan convert each polygon, triangle separately. We decided to use edge lists for the task. For each triangle, we make two edge lists one left and one right. One list consists of interpolated x, y and z value from the highest- to the lowest point and the other from the highest- to the third- to the lowest point of a triangle, se figure.

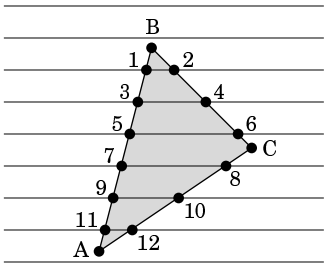


Figure 17: Edge lists

According to which shading algorithm we use, normals and colors are also interpolated. When we interpolate between two corners of a triangle we step pixel steps in y direction and get the values for x and z by using the k value, which is described earlier in this paper. We interpolate in similar way normals and colors. We could have used the alpha-value for interpolation, described earlier, but it is more compute intensive and is more difficult to use if one wants to make steps in pixel size.

After we have done the edge lists, we interpolate the values in the lists. We travel from the left list to the right list in pixel steps in x direction. It is described more in detail in the pseudocode for each of the different shading algorithms.

The scan conversion of the triangles could have been done without the use of edge lists, in a more faster and efficient way, but it would have been a lot more complicated.

4.1.6 Graphic User Interface

To make it easy for the users of our program we have menus. We used the functions for menus that were supported by OpenGL. There are menus for different shading types, directions and a special menu. In the special menu, one can choose if one wants to see vertex normals, face normals for the 3D models or see the model as a wire model. Of course, there is also an exit item in the menu structure.

4.1.7 Loader for 3D Models

When we decided how we should load models into our internal representation of the model, we look at different formats. We chose one that was very similar to our internal data structure and it was the RAW format. RAW stands for raw triangles and the format is described in [Rule 96]. It could some how be difficult to find three-dimensional objects that are stored in the RAW format, but luckily there are converters from other formats available.

4.1.8 Making Spheres

To test our program and the different shading algorithms we need a simple object that is also good for analysis. We think a sphere built up by triangles is a good choice. We found a program that made a approximation of a sphere by recursive subdivision in [Angel00]. The program starts with a tetrahedron. A tetrahedron is composed of four equilateral triangles, determined by four vertices. The triangles of the tetrahedron are recursive divide into smaller triangles. After the subdivision the four new triangles will still be in the same plane as the original triangle, but then we move out the middle triangle a bit.

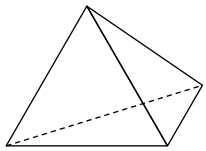


Figure 18: A tetrahedron

With more subdivisions the model becomes closer and closer to the shape of a sphere. One thing to notice with this method is that the triangles that build up the sphere are not all of the same size. It is a good feature when we later will test and time the different shading algorithms.

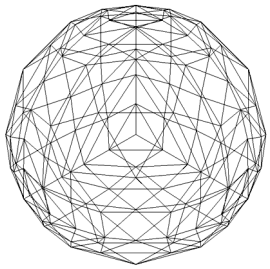


Figure 19: Sphere approximation using subdivision

We modified the existing program, so it makes an output file with the vertices in the RAW file format.

4.1.9 Timing the Algorithms

When we did the timing of all the shading algorithms we used the C library function clock(). It can be found in the library time.h. The clock() function returns a value for used processor time. And when we tried to use the function on the UNIX system, which is a shared system, the return value had very low resolution so we decided to do the timing on a Ms Windows system which worked better.

By calculate the difference between the returned value before and after the algorithm we get a time value, which can be used for comparing different shading algorithms. The time value we get from the timing process difference a little even if we time the same algorithm, so we decided to make a couple of measures and calculate an average.

We will use the sphere, described in section *Making Spheres* as our shading object. We will do measurements on spheres with different number of vertices, 4, 64 and 1024 triangles. They are called sphere 0, sphere 2 and sphere 4. In order to see how the different shading algorithms works on models with large and small polygons. As mention in section *Making Spheres* the spheres has triangles of different size, so we will do another set of measurements with the spheres rotated 180 degrees. We have tried to optimize all the shading algorithms equally for speed, but small differences can be found.

4.2 Design and Implementation of Shading Algorithms

The designed shading algorithms are all based on the Phong illumination model (described in the *Theoretical Background* part). We have choosen to only calculate the diffuse reflections of the three-dimensional objects, use a distant light source and we do not use any material coefficients, so the illumination formula can be reduced.

The reduced Pong illuminadion moddel.

$$I_d = \mathbf{l} \cdot \mathbf{n}$$

4.2.1 Flat Shading Algorithm

The flat shading algorithm is the simplest of the shading algorithms and therefore the easiest one to implement. After we have load a model into our internal data structure, we calculate a normal for each triangle. It is done by making two vectors of a triangle and then the normal is given by the cross product between these vectors. Then by apply the illumination model on the normals we get the shaded color for each polygon. The triangles are drawn by interpolating our two edge lists values, in other words we travels from the left to the right list. The step size is one pixel in the x direction.

Pseudocode:

```
Place 0 in faceCounter
Place 0 in pointCounter
DOUNTIL faceCounter is equal to NUM_OF_FACES
        Place calculated values in edgeList
        Place number of values in edgeList in pointInEdgeList
        SetColour for the triangle
        DOUNTIL pointCounter is equal to pointsInEdgeList
                x=edgeListLeft[x]
                DOUNTIL x is equal to edgeListRight[x]
                        Save point in framebuffer
                        Add 1 to x
                ENDDO
                Add 1 to pointCounter
        ENDDO
        Add 1 to faceCounter
ENDDO
```

4.2.2 Gouraud Shading Algorithm

Gouraud shading is a little bit harder to implement then flat shading. In similar way as for flat shading we interpolate the values in the edge lists. The edge lists for Gouraud shading contains in addition to the x, y and z value also the color. In other words lines is drawn in x direction between the two edge lists, with right interpolated color.

Pseudocode:

```
Place 0 in faceCounter
Place 0 in pointCounter
DOUNTIL faceCounter is equal to NUM_OF_FACES
        Place calculated values in edgeList
        Place number of values in edgeList in pointInEdgeList
        DOUNTIL pointCounter is equal to pointsInEdgeList
                x=edgeListLeft[x]
                DOUNTIL x is equal to edgeListRight[x]
                        SetColor for the point by interpolating between
                        the colors at the edge lists
                        Save point in framebuffer
                        Add 1 to x
                ENDDO
                Add 1 to pointCounter
        ENDDO
        Add 1 to faceCounter
ENDDO
```

4.2.3 Phong Shading Algorithm

The Phong shading algorithm works like the one for Gouraud shading , but instead of interpolating colors, normals are interpolated. The Phong illumination model is applied for every point on the triangle.

Pseudocode:

```
Place 0 in faceCounter
Place 0 in pointCounter
DOUNTIL faceCounter is equal to NUM_OF_FACES
        Place calculated values in edgeList
        Place number of values in edgeList in pointInEdgeList
        DOUNTIL pointCounter is equal to pointsInEdgeList
                x=edgeListLeft[x]
                DOUNTIL x is equal to edgeListRight[x]
                        Calculate the normal by interpolating
                        between the normals of the edge lists
                        Calculate the color for the point
                        Save point in framebuffer
                        Add 1 to x
                ENDDO
                Add 1 to pointCounter
        ENDDO
        Add 1 to faceCounter
ENDDO
```

4.2.4 Fast Phong Shading Algorithm, Duff

The Fast Phong shading algorithm has a set up for each scan line that is drawn.

Pseudocode:

```
Place 0 in faceCounter
Place 0 in pointCounter
DOUNTIL faceCounter is equal to NUM_OF_FACES
        Place calculated values in edgeList
        Place number of values in edgeList in pointInEdgeList
        DOUNTIL pointCounter is equal to pointsInEdgeList
                Calculate setup data
                x=edgeListLeft[x]
                DOUNTIL x is equal to edgeListRight[x]
                        Calculate the color for the point
                        Save point in framebuffer
                        Refresh setup data
                        Add 1 to x
                ENDDO
                Add 1 to pointCounter
        ENDDO
        Add 1 to faceCounter
ENDDO
```

4.2.5 Reflection Shading

The Reflection shading algorithm has like the Fast Phong shading algorithm a set up for each scan line, although it is bigger.

Pseudocode:

```
Place 0 in faceCounter
Place 0 in pointCounter
DOUNTIL faceCounter is equal to NUM_OF_FACES
        Place calculated values in edgeList
        Place number of values in edgeList in pointInEdgeList
        DOUNTIL pointCounter is equal to pointsInEdgeList
                Calculate setup data
                x=edgeListLeft[x]
                DOUNTIL x is equal to edgeListRight[x]
                        Calculate the color for the point
                        Save point in framebuffer
                        Refresh setup data
                        Add 1 to x
                ENDDO
                Add 1 to pointCounter
        ENDDO
        Add 1 to faceCounter
ENDDO
```

5 Run the program

The program is very simple to use. Run the program by typing **PolLac**. It loads the file **3dmodels/sphere3.raw** as default model. If one wants to load any other model just type the name and path for the file as an argument for example **PolLac 3dmodels/cow.raw**.

When the program starts it load the three-dimensional model and does some initial calculations, such as calculations of normals. It can take awhile if the loaded model consists of many vertices. Then a Phong shaded model of the loaded object appears in the window.

The menus appear when the right button is clicked. Then the user can choose different types of shading algorithms and other special functions. Exit the program by choosing the exit item in the menu.

The program can be found at ~/na98mpa/xjobb/ or ~/na99bnt/avd/ at ours school's network or downloaded from http://www.student.hig.se/~ na98mpa, http://www.student.hig.se/~ na99bnt or http://www.bjorn.nystedt.net.

6 Result

Phong- Fast Phong- and Reflection shading produce to our eyes no visual difference. One can see differences between these algorithms and Gouraud shading especially on sphere 0 and sphere 2. Flat shading gives the least appealing results and is not comparable to the others in visual aspects. Sphere 0 is an extream model that gives bad result for all but Flat shading, because it has few vertices.

The result of the measuring of the different shading algorithms along with a plotted curve can be found in the tables and the graf.

6.1 Tables, sphere 0

Sphere 0 front side, 4 triangles, 12 vertices.
Shaded 100 times.

Flat	Gouraud	Phong	Duff	Reflection
3380	3410	6860	4340	4180
3300	3350	6920	4340	4120
3300	3450	6920	4340	4120
3300	3410	6860	4340	4170
3300	3410	6870	4340	4170

Sphere 0 back side, 4 triangles, 12 vertices.
Shaded 100 times.

Flat	Gouraud	Phong	Duff	Reflection
1920	2200	2640	2580	2700
1980	2200	2630	2630	2690
1930	2140	2580	2640	2690
1930	2190	2640	2640	2690
1930	2190	2640	2650	2700

Avarage times for sphere 0.

Flat	Gouraud	Phong	Duff	Reflection
2630	2800	4760	3480	3420

6.2 Tables, sphere 2

Sphere 2 front side, 64 triangles, 192 vertices.
Shaded 100 times.

Flat	Gouraud	Phong	Duff	Reflection
8950	9000	19770	12140	11860
8950	9000	19780	12190	11810
8900	9010	19780	12140	11810
8900	9010	19770	12150	11800
8850	9000	19770	12140	11810

Sphere 2 back side, 64 triangles, 192 vertices.
Shaded 100 times.

Flat	Gouraud	Phong	Duff	Reflection
8240	8510	17460	11590	11920
8240	8460	17410	11640	11860
8260	8460	17460	11650	11860
8240	8460	17410	11640	11920
8240	8510	17470	11640	11860

Avarage times for sphere 2.

Flat	Gouraud	Phong	Duff	Reflection
8580	8740	18610	11890	11850

6.3 Tables, sphere 4

Sphere 4 front side, 1024 triangles, 3072 vertices.
Shaded 100 times.

Flat	Gouraud	Phong	Duff	Reflection
11310	11760	26030	19500	23890
11250	11810	26030	19490	23900
11250	11810	26040	19500	23900
11310	11760	26040	19490	23890
11310	11790	26030	19490	23880

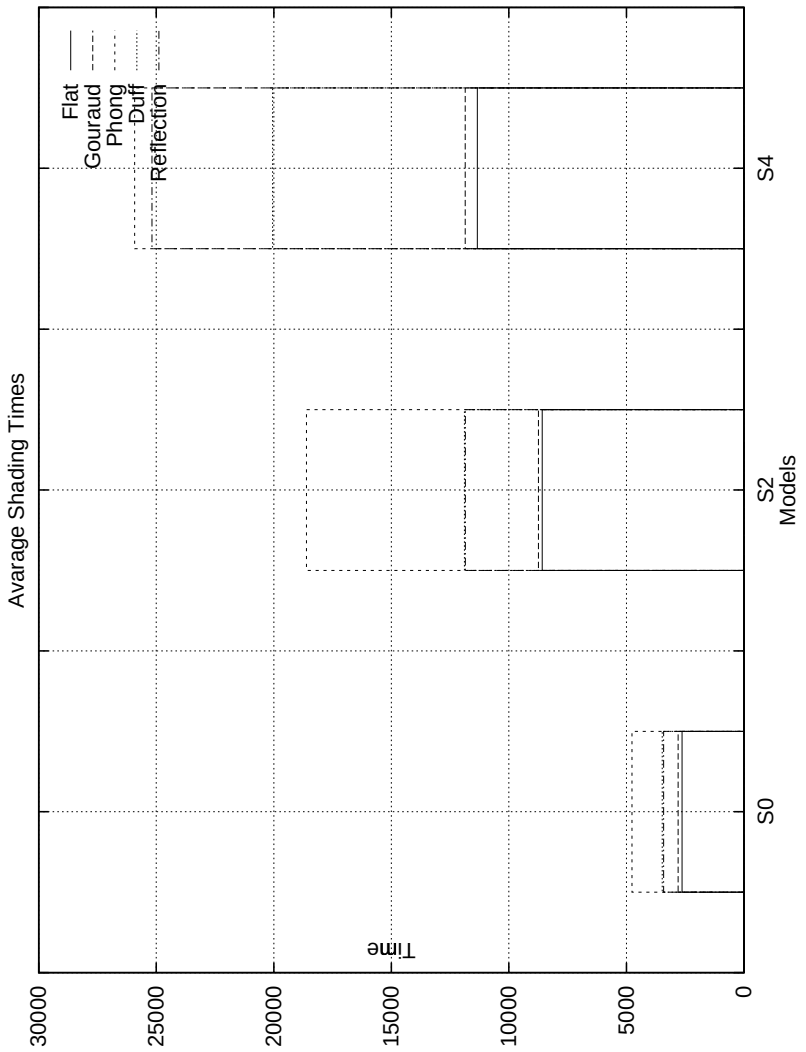
Sphere 4 back side, 1024 triangles, 3072 vertices.
Shaded 100 times.

Flat	Gouraud	Phong	Duff	Reflection
11370	11970	25820	20590	26470
11430	11970	25810	20600	26470
11400	11910	25810	20600	26480
11390	11910	25820	20600	26480
11400	11950	25810	20590	26470

Avarage times for sphere 4.

Flat	Gouraud	Phong	Duff	Reflection
11340	11860	25920	20050	25180

6.4 Graf


Avarage Shading Times
Time
Models
S0
S2
S4
Flat
Gouraud
Phong
Duff
Reflection
0
5000
10000
15000
20000
25000
30000

7 Discussion

In this section we are to discuss the results from our mesurements of the different shading algorithms. By examination of the tables and the diagram in the result section one can see that Flat- and Gouraud shading are fastest in all occasions. Flat is off course the fastest of them, but the time difference is not so big that one can expect and that is because of the scan conversion of the polygons takes up a major part of the timed value. Phong shading is the slowest in all occasions and should rightfully be so. Fast Phong- and Reflection shading's times are in the middle of the tested algorithms.

As mentioned in earlier sections the test is done on three approximated spheres with different number of polygons. Sphere 0 has least polygons, Sphere 4 most and Sphere 2 is in between. On the test on Sphere 0 and Sphere 2 Reflection shading is a little faster than Fast Phong shading, but on Sphere 4 the situation is reversed. That is because Reflection shading has more pre-calculations than Fast Phong shading and therefore becomes slower when the polygons are small.

The difference in speed between these two algorithms is not so big, so it is difficult to know which one is the best. As mentioned earlier Fast Phong shading is faster for small polygons, which is good for rendering detailed objects. But those objects could be placed in an environment made of larger polygons. So one has to examine what one wants to shade before choosing the fastest of them.

The key to these two's advantage over Phong shading is that they use less computative calculations than Phong shading. For example calculating the square rot take a lot of processor time compared to simpler operations like multiplications.

If one compare the shadings of what these three methods produce on the screen, no visual difference can be seen. It is no surprise because Fast Phong Shading is, as mentioned earlier, a different way of calculation the values for Phong shading. Reflection shading is a very close approximation of Phong shading, so it should be impossible to spot any difference.

Flat shading produce the least realistic shading and can only be recommended for special models, like sphere 0, mentioned in the result section, that gives unexpected result with the other methods or if one wants shading with a retro feeling. Gouraud shading which produces good looking shading and is easy to implement is a good choice of algorithm, if one compare speed and quality. Gouraud shading is already implemented in many 3D hardware accelerator cards.

8 Conclusion

It is proved by the test results and further discussed in the section above that our initial hypothese was correct. It is possible to achieve faster shading by using approximations of a curve instead of calculate the interpolated normals. One can get a square rot calculation less per pixel. Methods that uses approximations or other ways of doing is Fast Phong- , Reflection- and the methods mentioned in the further work section.

8.1 Further work

Our work has by no means been a complete timing test of the shading algorithms available, but we think it can give good indications on which algorithms that can be fast in certain situations. There are many other old, new and yet discovered algorithms to be tested. The ones we have been looking at but have not implemented are: Approximated Phong Shading by using the Euler Method [Hast01], Quadratic interpolation for near-Phong quality shading [Seiler98], which both seems to be fast.

References

[Duff79] T. Duff, *Smoothly Shaded Renderings of Polyhedral Objects on Raster Displays* ACM, Computer Graphics, Vol. 13, 1979, 270-275.

[Bishop86] G. Bishop, D. M. Weimer, *Fast Phong Shading* Computer Graphics vol. 20, No 4, 1986.

[Foley97] J. D. Foley, A. van Dam, S. K. Feiner, J. F. Hughes, *Computer Graphics - Principles and Practice* Addison-Wesley, 1997.

[Phong75] B. T. Phong, *Illumination for Computer Generated Pictures* Communications of the ACM, Vol. 18, No 6, June 1975.

[Gouraud71] H. Gouraud, *Continous Shading of Curved Surfaces* IEEE transcriptions in computers Vol. c-20, No 6, June 1971.

[HastSEG01] A. Hast, T. Barrera, E. Bengtsson, *Imporved Shading Performance by Avoiding Vector Normalization* VWSCG'2001.

[Hast01] A. Hast, T. Barrera, E. Bengtsson, *Approximated Phong Shading by using Euler* to be published at Eurographics 2001.

[Angel00] E. Angel, *Computer Graphics: a top down approach with OpenGL* 2^{nd} *ed.* Addison-Wesley, 2000.

[Rule96] K. Rule, *3D Graphics File Formats: a Programmer's Reference* Addison-Wesley, 1996.

[Hearn97] D. Hearn, *Computer Graphics* 2^{nd} *ed.* Pretice Hall, 1997.

[O'Rourke99] J. O'Rourke, *Computarional Geometry in C* 2^{nd} *ed.* Cambridge, 1999.

[Adams99] R. A. Adams, *Calculus a Complete Course* 4^{th} *ed.* Addison-Wesley, 1999.

[Mart00] B. Mårtensson, T. Nilstun, *Praktisk vetenskapsteori* Studentlitteratur, 2000.

[Woo97] M. Woo, J. Neider, T. Davis, *OpenGL Programming Guide: the official guide to learning OpenGL version 1.1* 2^{nd} *ed.* Addison-Wesley, 1997.

[Seiler98] L. Seiler, *Quadratic Interpolation for Near-Phong Quality Shading* SIGGraph, 1998.

9 Appendix

9.1 main.h

```
/**
 * File: main.h
 *
 * Auther: Björn Nystedt and Marko Pirttioja
 *
 * Last Modified: 20010516
 *
 * Version: 1.0
 *
 **/

#ifndef _MAIN_H_
#define _MAIN_H_
#include <GL/glut.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include "data.h"
#include "vector.h"

/**
 * Type definition of a boolean variable
 **/
typedef int Boolean;

/**
 * Data structure for a vertex
 **/
struct VertexStruct {
  Vector pos;
  Vector normal;
  Vector color;
};
typedef struct VertexStruct Vertex;

/**
 * Data structure for a face
 **/
struct FaceStruct {
  Vector normal;
```

```
  Vector midpoint;
  Vector color;
  Vector vertices[3];
};
typedef struct FaceStruct Face;

/**
 * Datastucture for the frame buffer
 **/
struct bufferStruct {
  Vector color;
  int z;
};
typedef struct bufferStruct buffer;

/**
 * Pre :
 * Post:
 **/
void rotation(int);

/**
 * Pre :
 * Post:
 **/
void scaling(float);

/**
 * Pre :
 * Post:
 **/
void translation(int, int, int);

/**
 * Pre :
 * Post: initialize the frame buffer to zero
 **/
void initfbuffer();

/**
 * Pre :
 * Post: load a model to the program
 **/
void loadModel();
```

```
/**
 * Pre :
 * Post: calculates the face normals
 **/

/**
 * Pre :
 * Post: scale and translate the model so it fits the window
 **/
void scaleModel();

/**
 * Pre :
 * Post:
 **/
void calcFaceMidPoint();

/**
 * Pre :
 * Post:
 **/
void calcFaceNormals(Vertex, Vertex, Vertex, Vector);

/**
 * Pre :
 * Post: calculate a shade for a point
 **/
void calcShading();

/**
 * Pre :
 * Post: initialize the face normal, midpoint and color to zero
 **/
void initFace();

/**
 * Pre :
 * Post: calculate the normal for the vertex
 **/
void calcVertexNormals();

/**
 * Pre :
```

```
 * Post: make edgelists for flat shading and
 *       returns the number of points in the lists
 **/
int makeFlatLists(int);

/**
 * Pre :
 * Post: make edgelists for Goudraud shading and
 *       returns the number of points in the lists
 **/
int makeGouraudLists(int);

/**
 * Pre :
 * Post: make edgelists for Phong, Duff and reflection shading and
 *       returns the number of points in the lists
 **/
int makePhongLists(int);

/**
 * Pre :
 * Post: draw the frame buffer on the screen
 **/
void drawBuffer();

/**
 * The programs most important function
 * Calculates and displays the shaded models on screen
 **/
void display();

/**
 * Pre :
 * Post:
 **/
void menu(int);

/**
 * Pre :
 * Post:
 **/
void shadingMenu(int);

/**
```

```
 * Pre :
 * Post:
 **/
void specialMenu(int);

/**
 * Pre :
 * Post:
 **/
void rotateMenu(int);

/**
 * Pre :
 * Post:
 **/
void transMenu(int);

/**
 * Pre :
 * Post:
 **/
void lightMenu(int);

/**
 * Intitialize OpenGl
 **/
void myinit();

/**
 * Main function
 **/
int main(int, char**);

#endif
```

9.2 main.c

```
/**
 * Author: Björn Nystedt and Marko Pirttioja
 **/

#include "main.h"
#define FLAT 1
#define GOURAUD 2
#define PHONG 3
#define DUFF 4
#define REFLECTION 5
#define TRUE 1
#define FALSE 0
#define FBUFFERSIZE 600

char modelfile[40];
int shadingLoopCounter=1;
int lowest;
float trans = 0.0; /*100-SCALEFACTOR;*/
float scalefactor;
buffer fBuffer[FBUFFERSIZE][FBUFFERSIZE];
int rotated=FALSE;

/* Points is a pointer on a Vertex */
Vertex *points;

/* Faces is a pointer on a Face */
Face *faces;

/* Faces is a pointer on a Face */
Vertex listLeft[99000];
Vertex listRight[99000];

/*Default light direction - from viewer */
Vector lightDir={0.0,0.0,1.0};

/*Default viewer direction - from viewer */
Vector viewDir={0.0,0.0,1.0};

/*Half vector */
Vector halfVector={0.0,0.0,0.0};

/* Alternetive light directions */
```

```
/* From viewer */
Vector lightAlt1={0.0, 0.0, 1.0};

/* From viewer, upper right */
Vector lightAlt2={1.0, 1.0, 1.0};

/* From back, upper right, but more to the center */
Vector lightAlt3={0.5, 0.5, 1.0};

/* Shading algorithm that is used 1=Flat 2=Gouraud 3=Phong */
int shadingType=FLAT;

/* Special */
Boolean faceNormals=FALSE;
Boolean vertexNormals=FALSE;
Boolean wireModel=FALSE;
Boolean switchDir=FALSE;

int NUM_OF_VERTICES=0;
int NUM_OF_FACES=0;

long timeBefore, timeAfter;

void rotation(int v) {
  int i;
  float tmp;
  if(v==90)
    tmp=M_PI/2;
  if(v==180)
    tmp=M_PI;
  for(i=0; i<NUM_OF_VERTICES; ++i) {
    points[i].pos[0]=points[i].pos[0]*cos(tmp)-points[i].pos[2]*sin(tmp);
    points[i].pos[2]=points[i].pos[0]*sin(tmp)+points[i].pos[2]*cos(tmp);
  }
}

void scaling(float factor) {
  int i;
  for(i=0; i<NUM_OF_VERTICES; ++i)
    divVectorScaler(points[i].pos,factor);
}

void translation(int x, int y, int z) {
  int i;
```

```
  for(i=0; i<NUM_OF_VERTICES; ++i) {
    points[i].pos[0]+=x;
    points[i].pos[1]+=y;
    points[i].pos[2]+=z;
  }
}

void initfbuffer() {
  int i,j;
  for(i=0; i<FBUFFERSIZE; i++)
    for(j=0; j<FBUFFERSIZE; j++) {
      fBuffer[i][j].z=0;
      fBuffer[i][j].color[0]=1.0;
      fBuffer[i][j].color[1]=1.0;
      fBuffer[i][j].color[2]=1.0;
    }
}

void loadModel(char *fileName) {
  FILE *fp;
  int i=0, highest=0;
  float f0, f1, f2;

  if( (fp = fopen(/*"3dmodels/sphere3.raw"*/fileName, "r")) != 0) {
    printf("Loading model...");
    while( (fscanf(fp, "%f %f %f", &f0, &f1, &f2)) != EOF) {
      points[i].pos[0]=f0;
      points[i].pos[1]=f1;
      points[i].pos[2]=f2;
      i++;

      if(f0<lowest)
lowest=(int)(f0-1.0);
      if(f1<lowest)
lowest=(int)(f1-1.0);

      if(f0>highest)
highest=(int)(f0+1.0);
      if(f1>highest)
highest=(int)(f1+1.0);
    }
    if(highest>-lowest)
      lowest=-highest;
    fclose(fp);
```

```
    printf("Finished\n");
    NUM_OF_VERTICES=i;
    NUM_OF_FACES=i/3;

    printf("Finished\n");
    printf("Lowest: %i, Highest %i\n", lowest, highest);
    if((lowest*-1)>highest)
      scalefactor=FBUFFERSIZE/2/(-1*lowest);
    else
      scalefactor=FBUFFERSIZE/2/(highest);
    printf("scalefactor: %f\n", scalefactor);
  }
  else
    printf("File not found\n");
}

void scaleModel() {
  int i;
  for(i=0; i<NUM_OF_VERTICES; ++i) {
    points[i].pos[0]=(int)(0.5+(points[i].pos[0]-lowest)*scalefactor);
    points[i].pos[1]=(int)(0.5+(points[i].pos[1]-lowest)*scalefactor);
    points[i].pos[2]=(int)(0.5+(points[i].pos[2]-lowest)*scalefactor);
  }
}

void calcFaceMidPoint() {
  int i;
  for(i=0; i<NUM_OF_FACES; ++i)
    calcMidPoint(points[3*i+0].pos,
 points[3*i+1].pos,
 points[3*i+2].pos,
 faces[i].midpoint);
}

void calcFaceNormal(Vertex v1, Vertex v2, Vertex v3, Vector v) {
  Vector vTemp1, vTemp2;
  subVector(v1.pos, v2.pos, vTemp1);
  subVector(v1.pos, v3.pos, vTemp2);
  crossProd(vTemp1, vTemp2, v);
  normalize(v);
}

void calcShading(Vector v, Vector c) {
```

```
  float tmp=dotProduct(v, lightDir);
  if(tmp<0)
    c[0]=0.0;
  /* Color sets to 0 if it is negative */
  else
    c[0]=tmp;
  c[1]=0.0;
  c[2]=0.0;
}

void initFace() {
  int i, j;
  for(i=0; i<NUM_OF_FACES; ++i) {
    for(j=0; j<3; ++j) {
      faces[i].normal[j]=0.0;
      faces[i].midpoint[j]=0.0;
      faces[i].color[j]=0.0;
    }
  }
  for(i=0; i<NUM_OF_FACES; ++i) {
    calcFaceNormal(points[3*i+0],points[3*i+1],points[3*i+2],faces[i].normal);
    calcMidPoint(points[3*i+0].pos, points[3*i+1].pos, points[3*i+2].pos,
    faces[i].midpoint);
  }
}

void calcVertexNormal() {
  int i,j;
  /* This loop take long time if it is many vertices */
  for(i=0; i<NUM_OF_VERTICES; ++i) {
    for(j=0; j<NUM_OF_VERTICES; ++j) {
      if(isEqual(points[i].pos,points[j].pos)) {
addVector(points[i].normal,faces[(int)(j/3)].normal);
      }
    }
    normalize(points[i].normal);
  }
}

int makeFlatLists(int faceNr) {
  int ileft, iright, ydiff, y;
  float xstep, zstep, x, z;
  Vector tempPos;
  Vertex pointHigh, pointThree, pointLow;
```

```
/* Set pointHigh to the first point */
copyVector(pointHigh.pos, points[0+3*faceNr].pos);

/* Set pointHigh to the second point if it has greater y-value */
if(points[1+3*faceNr].pos[1]>pointHigh.pos[1]) {
  copyVector(pointLow.pos, pointHigh.pos);
  copyVector(pointHigh.pos, points[1+3*faceNr].pos);
}
else {
  copyVector(pointLow.pos, points[1+3*faceNr].pos);
}
/* Set pointHigh to the third point if it has greater y-value */
if(points[2+3*faceNr].pos[1]>pointHigh.pos[1]) {
  copyVector(pointThree.pos, pointHigh.pos);
  copyVector(pointHigh.pos, points[2+3*faceNr].pos);
}
else {
  copyVector(pointThree.pos, points[2+3*faceNr].pos);
}
/* If pointLow has higher y-value switch pointLow with pointThree */
if(pointLow.pos[1]>=pointThree.pos[1]) {
  copyVector(tempPos, pointLow.pos);
  copyVector(pointLow.pos, pointThree.pos);
  copyVector(pointThree.pos, tempPos);
}

/* case 1
   x
   xx
   xxx
   xxxx
   xxx
   xx
   x
*/

/* Test on which side pointThree is according to pointHigh and Low */
if(area(pointHigh.pos, pointLow.pos, pointThree.pos) > 0) {

  /* Travel from pointHigh to pointLow */

  x=pointHigh.pos[0];
  y=pointHigh.pos[1];
```

```
z=pointHigh.pos[2];

/* Calculate pixel steps in y direction */
ydiff=pointHigh.pos[1]-pointLow.pos[1];

/* Calculate step size in x and y direction */
if((pointHigh.pos[1]-pointLow.pos[1]) != 0) {
  xstep=(pointHigh.pos[0]-pointLow.pos[0]) / ydiff;
  zstep=(pointHigh.pos[2]-pointLow.pos[2]) / ydiff;
}
else {
  xstep=0.0;
  zstep=0.0;
}

/* Gives the first to the last - 1 values to the list */
for(ileft=0; ileft<ydiff; ++ileft) {

  /* Give the list pixelvalues for x, y and x */
  listLeft[ileft].pos[0]=(int)(0.5+x);
  listLeft[ileft].pos[1]=y;
  listLeft[ileft].pos[2]=(int)(0.5+z);

  x-=xstep;
  --y;
  z-=zstep;
}

/* Values for the last point in the list */
listLeft[ileft].pos[0]=(int)(0.5+x);
listLeft[ileft].pos[1]=y;
listLeft[ileft].pos[2]=(int)(0.5+z);

/* Travle from pointHigh to pointThree */

x=pointHigh.pos[0];
y=pointHigh.pos[1];
z=pointHigh.pos[2];

/* Calculate pixel steps in y direction */
ydiff=pointHigh.pos[1]-pointThree.pos[1];

/* Calculate step size in x and y direction */
if((pointHigh.pos[1]-pointThree.pos[1]) != 0) {
```

```
  xstep=(pointHigh.pos[0]-pointThree.pos[0]) / ydiff;
  zstep=(pointHigh.pos[2]-pointThree.pos[2]) / ydiff;
}
else {
  xstep=0.0;
  zstep=0.0;
}

/* Gives the first to the last - 1 values to the list */
for(iright=0; iright<ydiff; ++iright) {

  /* Give the list pixelvalues for x, y and x */
  listRight[iright].pos[0]=(int)(0.5+x);
  listRight[iright].pos[1]=y;
  listRight[iright].pos[2]=(int)(0.5+z);

  x-=xstep;
  --y;
  z-=zstep;
}

/* Values for the last point in the list */
listRight[iright].pos[0]=(int)(0.5+x);
listRight[iright].pos[1]=y;
listRight[iright].pos[2]=(int)(0.5+z);

/* Travel from pointThree to pointLow */

x=pointThree.pos[0];
y=pointThree.pos[1];
z=pointThree.pos[2];

/* Calculate pixel steps in y direction */
ydiff=pointThree.pos[1]-pointLow.pos[1];

/* Calculate step size in x and y direction */
if((pointThree.pos[1]-pointLow.pos[1]) != 0) {
  xstep=(pointThree.pos[0]-pointLow.pos[0]) / ydiff;
  zstep=(pointThree.pos[2]-pointLow.pos[2]) / ydiff;
}
else {
  xstep=0.0;
  zstep=0.0;
}
```

```
  /* Gives the first to the last - 1 values to the list */
  for(iright=iright; iright<ileft; ++iright) {

    /* Give the list pixelvalues for x, y and x */
    listRight[iright].pos[0]=(int)(0.5+x);
    listRight[iright].pos[1]=y;
    listRight[iright].pos[2]=(int)(0.5+z);

    x-=xstep;
    --y;
    z-=zstep;
  }

  /* Values for the last point in the list */
  listRight[iright].pos[0]=(int)(0.5+x);
  listRight[iright].pos[1]=y;
  listRight[iright].pos[2]=(int)(0.5+z);
}

/* case 2
   x
   xx
   xxx
   xxxx
   xxx
   xx
   x
*/

else {

  /* Travel from pointHigh to pointLow */

  x=pointHigh.pos[0];
  y=pointHigh.pos[1];
  z=pointHigh.pos[2];

  /* Calculate pixel steps in y direction */
  ydiff=pointHigh.pos[1]-pointLow.pos[1];

  /* Calculate step size in x and y direction */
  if((pointHigh.pos[1]-pointLow.pos[1]) != 0) {
    xstep=(pointHigh.pos[0]-pointLow.pos[0]) / ydiff;
```

```
  zstep=(pointHigh.pos[2]-pointLow.pos[2]) / ydiff;
}
else {
  xstep=0.0;
  zstep=0.0;
}

/* Gives the first to the last - 1 values to the list */
for(iright=0; iright<ydiff; ++iright) {

  /* Give the list pixelvalues for x, y and x */
  listRight[iright].pos[0]=(int)(0.5+x);
  listRight[iright].pos[1]=y;
  listRight[iright].pos[2]=(int)(0.5+z);

  x-=xstep;
  --y;
  z-=zstep;
}

/* Values for the last point in the list */
listRight[iright].pos[0]=(int)(0.5+x);
listRight[iright].pos[1]=y;
listRight[iright].pos[2]=(int)(0.5+z);

/* Travle from pointHigh to pointThree */
x=pointHigh.pos[0];
y=pointHigh.pos[1];
z=pointHigh.pos[2];

/* Calculate pixel steps in y direction */
ydiff=pointHigh.pos[1]-pointThree.pos[1];

/* Calculate step size in x and y direction */
if((pointHigh.pos[1]-pointThree.pos[1]) != 0) {
  xstep=(pointHigh.pos[0]-pointThree.pos[0]) / ydiff;
  zstep=(pointHigh.pos[2]-pointThree.pos[2]) / ydiff;
}
else {
  xstep=0.0;
  zstep=0.0;
}

/* Gives the first to the last - 1 values to the list */
```

```
for(ileft=0; ileft<ydiff; ++ileft) {

  /* Give the list pixelvalues for x, y and x */
  listLeft[ileft].pos[0]=(int)(0.5+x);
  listLeft[ileft].pos[1]=y;
  listLeft[ileft].pos[2]=(int)(0.5+z);

  x-=xstep;
  --y;
  z-=zstep;
}

/* Values for the last point in the list */
listLeft[ileft].pos[0]=(int)(0.5+x);
listLeft[ileft].pos[1]=y;
listLeft[ileft].pos[2]=(int)(0.5+z);

/* Travel from pointThree to pointLow */

x=pointThree.pos[0];
y=pointThree.pos[1];
z=pointThree.pos[2];

/* Gives the first to the last - 1 values to the list */
ydiff=pointThree.pos[1]-pointLow.pos[1];

/* Calculate step size in x and y direction */
if((pointThree.pos[1]-pointLow.pos[1]) != 0) {
  xstep=(pointThree.pos[0]-pointLow.pos[0]) / ydiff;
  zstep=(pointThree.pos[2]-pointLow.pos[2]) / ydiff;
}
else {
  xstep=0.0;
  zstep=0.0;
}

/* Gives the first to the last - 1 values to the list */
for(ileft=ileft; ileft<iright; ++ileft) {

  /* Give the list pixelvalues for x, y and x */
  listLeft[ileft].pos[0]=(int)(0.5+x);
  listLeft[ileft].pos[1]=y;
  listLeft[ileft].pos[2]=(int)(0.5+z);
```

```
      x-=xstep;
      --y;
      z-=zstep;
    }

    /* Values for the last point in the list */
    listLeft[ileft].pos[0]=(int)(0.5+x);
    listLeft[ileft].pos[1]=y;
    listLeft[ileft].pos[2]=(int)(0.5+z);
  }
  return iright;
}

int makeGouraudLists(int faceNr) {
  int ileft, iright, ydiff, y;
  float xstep, zstep, cstep, x, z, c;
  Vector tempPos, tempNormal;
  Vertex pointHigh, pointThree, pointLow;

  /* Set pointHigh to the first point */
  copyVector(pointHigh.pos, points[0+3*faceNr].pos);
  copyVector(pointHigh.normal, points[0+3*faceNr].normal);

  /* Set pointHigh to the second point if it has greater y-value */
  if(points[1+3*faceNr].pos[1]>pointHigh.pos[1]) {
    copyVector(pointLow.pos, pointHigh.pos);
    copyVector(pointLow.normal, pointHigh.normal);

    copyVector(pointHigh.pos, points[1+3*faceNr].pos);
    copyVector(pointHigh.normal, points[1+3*faceNr].normal);
  }
  else {
    copyVector(pointLow.pos, points[1+3*faceNr].pos);
    copyVector(pointLow.normal, points[1+3*faceNr].normal);
  }
  /* Set pointHigh to the third point if it has greater y-value */
  if(points[2+3*faceNr].pos[1]>pointHigh.pos[1]) {
    copyVector(pointThree.pos, pointHigh.pos);
    copyVector(pointThree.normal, pointHigh.normal);

    copyVector(pointHigh.pos, points[2+3*faceNr].pos);
    copyVector(pointHigh.normal, points[2+3*faceNr].normal);
  }
  else {
```

```
  copyVector(pointThree.pos, points[2+3*faceNr].pos);
  copyVector(pointThree.normal, points[2+3*faceNr].normal);
}
/* If pointLow has higher y-value switch pointLow with pointThree */
if(pointLow.pos[1]>=pointThree.pos[1]) {
  copyVector(tempPos, pointLow.pos);
  copyVector(tempNormal, pointLow.normal);

  copyVector(pointLow.pos, pointThree.pos);
  copyVector(pointLow.normal, pointThree.normal);

  copyVector(pointThree.pos, tempPos);
  copyVector(pointThree.normal, tempNormal);
}

/*calcColors in pointHigh,Low,Three - ONLY FOR -  Gouraud shading*/
normalize(pointHigh.normal);
normalize(pointLow.normal);
normalize(pointThree.normal);

calcShading(pointHigh.normal,pointHigh.color);
calcShading(pointLow.normal,pointLow.color);
calcShading(pointThree.normal,pointThree.color);

/* case 1
   x
   xx
   xxx
   xxxx
   xxx
   xx
   x
*/
if(area(pointHigh.pos, pointLow.pos, pointThree.pos) > 0) {
  /* Travel from pointHigh to pointLow */
  x=pointHigh.pos[0];
  y=pointHigh.pos[1];
  z=pointHigh.pos[2];
  c=pointHigh.color[0];

  ydiff=pointHigh.pos[1]-pointLow.pos[1];

  if((pointHigh.pos[1]-pointLow.pos[1]) != 0) {
    xstep=(pointHigh.pos[0]-pointLow.pos[0]) / ydiff;
```

```
  zstep=(pointHigh.pos[2]-pointLow.pos[2]) / ydiff;
  cstep=(pointHigh.color[0]-pointLow.color[0]) / ydiff;
}
else {
  xstep=0.0;
  zstep=0.0;
  cstep=0.0;
}

for(ileft=0; ileft<ydiff; ++ileft) {
  listLeft[ileft].pos[0]=(int)(0.5+x);
  listLeft[ileft].pos[1]=y;
  listLeft[ileft].pos[2]=(int)(0.5+z);
  listLeft[ileft].color[0]=c;

  x-=xstep;
  --y;
  z-=zstep;
  c-=cstep;
}

listLeft[ileft].pos[0]=(int)(0.5+x);
listLeft[ileft].pos[1]=y;
listLeft[ileft].pos[2]=(int)(0.5+z);
listLeft[ileft].color[0]=c;

/* Travle from pointHigh to pointThree */
x=pointHigh.pos[0];
y=pointHigh.pos[1];
z=pointHigh.pos[2];
c=pointHigh.color[0];

ydiff=pointHigh.pos[1]-pointThree.pos[1];
if((pointHigh.pos[1]-pointThree.pos[1]) != 0) {
  xstep=(pointHigh.pos[0]-pointThree.pos[0]) / ydiff;
  zstep=(pointHigh.pos[2]-pointThree.pos[2]) / ydiff;
  cstep=(pointHigh.color[0]-pointThree.color[0]) / ydiff;
}
else {
  xstep=0.0;
  zstep=0.0;
  cstep=0.0;
}
```

```
for(iright=0; iright<ydiff; ++iright) {
  listRight[iright].pos[0]=(int)(0.5+x);
  listRight[iright].pos[1]=y;
  listRight[iright].pos[2]=(int)(0.5+z);
  listRight[iright].color[0]=c;

  x-=xstep;
  --y;
  z-=zstep;
  c-=cstep;
}

listRight[iright].pos[0]=(int)(0.5+x);
listRight[iright].pos[1]=y;
listRight[iright].pos[2]=(int)(0.5+z);
listRight[iright].color[0]=c;

/* Travel from pointThree to pointLow */
x=pointThree.pos[0];
y=pointThree.pos[1];
z=pointThree.pos[2];
c=pointThree.color[0];

ydiff=pointThree.pos[1]-pointLow.pos[1];
if((pointThree.pos[1]-pointLow.pos[1]) != 0) {
  xstep=(pointThree.pos[0]-pointLow.pos[0]) / ydiff;
  zstep=(pointThree.pos[2]-pointLow.pos[2]) / ydiff;
  cstep=(pointThree.color[0]-pointLow.color[0]) / ydiff;
}
else {
  xstep=0.0;
  zstep=0.0;
  cstep=0.0;
}

for(iright=iright; iright<ileft; ++iright) {
  listRight[iright].pos[0]=(int)(0.5+x);
  listRight[iright].pos[1]=y;
  listRight[iright].pos[2]=(int)(0.5+z);
  listRight[iright].color[0]=c;

  x-=xstep;
  --y;
  z-=zstep;
```

```
    c-=cstep;
  }
  listRight[iright].pos[0]=(int)(0.5+x);
  listRight[iright].pos[1]=y;
  listRight[iright].pos[2]=(int)(0.5+z);
  listRight[iright].color[0]=c;
}

/* case 2
   x
   xx
   xxx
   xxxx
   xxx
   xx
   x
*/
else {
  /* Travel from pointHigh to pointLow */
  x=pointHigh.pos[0];
  y=pointHigh.pos[1];
  z=pointHigh.pos[2];
  c=pointHigh.color[0];

  ydiff=pointHigh.pos[1]-pointLow.pos[1];
  if((pointHigh.pos[1]-pointLow.pos[1]) != 0) {
    xstep=(pointHigh.pos[0]-pointLow.pos[0]) / ydiff;
    zstep=(pointHigh.pos[2]-pointLow.pos[2]) / ydiff;
    cstep=(pointHigh.color[0]-pointLow.color[0]) / ydiff;
  }
  else {
    xstep=0.0;
    zstep=0.0;
    cstep=0.0;
  }

  for(iright=0; iright<ydiff; ++iright) {
    listRight[iright].pos[0]=(int)(0.5+x);
    listRight[iright].pos[1]=y;
    listRight[iright].pos[2]=(int)(0.5+z);
    listRight[iright].color[0]=c;

    x-=xstep;
    --y;
```

```
  z-=zstep;
  c-=cstep;
}

listRight[iright].pos[0]=(int)(0.5+x);
listRight[iright].pos[1]=y;
listRight[iright].pos[2]=(int)(0.5+z);
listRight[iright].color[0]=c;

/* Travle from pointHigh to pointThree */
x=pointHigh.pos[0];
y=pointHigh.pos[1];
z=pointHigh.pos[2];
c=pointHigh.color[0];

ydiff=pointHigh.pos[1]-pointThree.pos[1];
if((pointHigh.pos[1]-pointThree.pos[1]) != 0) {
  xstep=(pointHigh.pos[0]-pointThree.pos[0]) / ydiff;
  zstep=(pointHigh.pos[2]-pointThree.pos[2]) / ydiff;
  cstep=(pointHigh.color[0]-pointThree.color[0]) / ydiff;
}
else {
  xstep=0.0;
  zstep=0.0;
  cstep=0.0;
}

for(ileft=0; ileft<ydiff; ++ileft) {
  listLeft[ileft].pos[0]=(int)(0.5+x);
  listLeft[ileft].pos[1]=y;
  listLeft[ileft].pos[2]=(int)(0.5+z);
  listLeft[ileft].color[0]=c;

  x-=xstep;
  --y;
  z-=zstep;
  c-=cstep;
}

listLeft[ileft].pos[0]=(int)(0.5+x);
listLeft[ileft].pos[1]=y;
listLeft[ileft].pos[2]=(int)(0.5+z);
listLeft[ileft].color[0]=c;
```

```
    /* Travel from pointThree to pointLow */
    x=pointThree.pos[0];
    y=pointThree.pos[1];
    z=pointThree.pos[2];
    c=pointThree.color[0];

    ydiff=pointThree.pos[1]-pointLow.pos[1];
    if((pointThree.pos[1]-pointLow.pos[1]) != 0) {
      xstep=(pointThree.pos[0]-pointLow.pos[0]) / ydiff;
      zstep=(pointThree.pos[2]-pointLow.pos[2]) / ydiff;
      cstep=(pointThree.color[0]-pointLow.color[0]) / ydiff;
    }
    else {
      xstep=0.0;
      zstep=0.0;
      cstep=0.0;
    }

    for(ileft=ileft; ileft<iright; ++ileft) {
      listLeft[ileft].pos[0]=(int)(0.5+x);
      listLeft[ileft].pos[1]=y;
      listLeft[ileft].pos[2]=(int)(0.5+z);
      listLeft[ileft].color[0]=c;

      x-=xstep;
      --y;
      z-=zstep;
      c-=cstep;
    }
    listLeft[ileft].pos[0]=(int)(0.5+x);
    listLeft[ileft].pos[1]=y;
    listLeft[ileft].pos[2]=(int)(0.5+z);
    listLeft[ileft].color[0]=c;
  }

  return iright;
}

int makePhongLists(int faceNr) {
  int ileft, iright, ydiff, y;
  float xstep, zstep, x, z, n0step, n1step, n2step;
  Vector tempPos, tempNormal, nVector;
  Vertex pointHigh, pointThree, pointLow;
```

```
/* Set pointHigh to the first point */
copyVector(pointHigh.pos, points[0+3*faceNr].pos);
copyVector(pointHigh.normal, points[0+3*faceNr].normal);

/* Set pointHigh to the second point if it has greater y-value */
if(points[1+3*faceNr].pos[1]>pointHigh.pos[1]) {
  copyVector(pointLow.pos, pointHigh.pos);
  copyVector(pointLow.normal, pointHigh.normal);

  copyVector(pointHigh.pos, points[1+3*faceNr].pos);
  copyVector(pointHigh.normal, points[1+3*faceNr].normal);
}
else {
  copyVector(pointLow.pos, points[1+3*faceNr].pos);
  copyVector(pointLow.normal, points[1+3*faceNr].normal);
}
/* Set pointHigh to the third point if it has greater y-value */
if(points[2+3*faceNr].pos[1]>pointHigh.pos[1]) {
  copyVector(pointThree.pos, pointHigh.pos);
  copyVector(pointThree.normal, pointHigh.normal);

  copyVector(pointHigh.pos, points[2+3*faceNr].pos);
  copyVector(pointHigh.normal, points[2+3*faceNr].normal);
}
else {
  copyVector(pointThree.pos, points[2+3*faceNr].pos);
  copyVector(pointThree.normal, points[2+3*faceNr].normal);
}
/* If pointLow has higher y-value switch pointLow with pointThree */
if(pointLow.pos[1]>=pointThree.pos[1]) {
  copyVector(tempPos, pointLow.pos);
  copyVector(tempNormal, pointLow.normal);

  copyVector(pointLow.pos, pointThree.pos);
  copyVector(pointLow.normal, pointThree.normal);

  copyVector(pointThree.pos, tempPos);
  copyVector(pointThree.normal, tempNormal);
}

/* case 1
   x
   xx
   xxx
```

```
   xxxx
   xxx
   xx
   x
*/

if(area(pointHigh.pos, pointLow.pos, pointThree.pos) > 0) {
  /* Travel from pointHigh to pointLow */
  x=pointHigh.pos[0];
  y=pointHigh.pos[1];
  z=pointHigh.pos[2];
  copyVector(nVector,pointHigh.normal);

  ydiff=pointHigh.pos[1]-pointLow.pos[1];

  if(ydiff != 0) {
    xstep=(pointHigh.pos[0]-pointLow.pos[0]) / ydiff;
    zstep=(pointHigh.pos[2]-pointLow.pos[2]) / ydiff;
    n0step=(pointHigh.normal[0]-pointLow.normal[0]) / ydiff;
    n1step=(pointHigh.normal[1]-pointLow.normal[1]) / ydiff;
    n2step=(pointHigh.normal[2]-pointLow.normal[2]) / ydiff;
  }
  else {
    xstep=0.0;
    zstep=0.0;
    n0step=0.0;
    n1step=0.0;
    n2step=0.0;
  }

  for(ileft=0; ileft<ydiff; ++ileft) {
    listLeft[ileft].pos[0]=(int)(0.5+x);
    listLeft[ileft].pos[1]=y;
    listLeft[ileft].pos[2]=(int)(0.5+z);
    copyVector(listLeft[ileft].normal,nVector);
    normalize(listLeft[ileft].normal);

    x-=xstep;
    --y;
    z-=zstep;
    nVector[0]-=n0step;
    nVector[1]-=n1step;
    nVector[2]-=n2step;
  }
```

```
listLeft[ileft].pos[0]=(int)(0.5+x);
listLeft[ileft].pos[1]=y;
listLeft[ileft].pos[2]=(int)(0.5+z);
copyVector(listLeft[ileft].normal,nVector);
normalize(listLeft[ileft].normal);

/* Travle from pointHigh to pointThree */
x=pointHigh.pos[0];
y=pointHigh.pos[1];
z=pointHigh.pos[2];
copyVector(nVector,pointHigh.normal);

ydiff=pointHigh.pos[1]-pointThree.pos[1];
if((pointHigh.pos[1]-pointThree.pos[1]) != 0) {
  xstep=(pointHigh.pos[0]-pointThree.pos[0]) / ydiff;
  zstep=(pointHigh.pos[2]-pointThree.pos[2]) / ydiff;
  n0step=(pointHigh.normal[0]-pointThree.normal[0]) / ydiff;
  n1step=(pointHigh.normal[1]-pointThree.normal[1]) / ydiff;
  n2step=(pointHigh.normal[2]-pointThree.normal[2]) / ydiff;
}
else {
  xstep=0.0;
  zstep=0.0;
  n0step=0.0;
  n1step=0.0;
  n2step=0.0;
}

for(iright=0; iright<ydiff; ++iright) {
  listRight[iright].pos[0]=(int)(0.5+x);
  listRight[iright].pos[1]=y;
  listRight[iright].pos[2]=(int)(0.5+z);
  copyVector(listRight[iright].normal,nVector);
  normalize(listRight[iright].normal);

  x-=xstep;
  --y;
  z-=zstep;
  nVector[0]-=n0step;
  nVector[1]-=n1step;
  nVector[2]-=n2step;
}
```

```
listRight[iright].pos[0]=(int)(0.5+x);
listRight[iright].pos[1]=y;
listRight[iright].pos[2]=(int)(0.5+z);
copyVector(listRight[iright].normal,nVector);
normalize(listRight[iright].normal);

/* Travel from pointThree to pointLow */
x=pointThree.pos[0];
y=pointThree.pos[1];
z=pointThree.pos[2];
copyVector(nVector,pointThree.normal);

ydiff=pointThree.pos[1]-pointLow.pos[1];
if((pointThree.pos[1]-pointLow.pos[1]) != 0) {
  xstep=(pointThree.pos[0]-pointLow.pos[0]) / ydiff;
  zstep=(pointThree.pos[2]-pointLow.pos[2]) / ydiff;
  n0step=(pointThree.normal[0]-pointLow.normal[0]) / ydiff;
  n1step=(pointThree.normal[1]-pointLow.normal[1]) / ydiff;
  n2step=(pointThree.normal[2]-pointLow.normal[2]) / ydiff;
}
else {
  xstep=0.0;
  zstep=0.0;
  n0step=0.0;
  n1step=0.0;
  n2step=0.0;
}

for(iright=iright; iright<ileft; ++iright) {
  listRight[iright].pos[0]=(int)(0.5+x);
  listRight[iright].pos[1]=y;
  listRight[iright].pos[2]=(int)(0.5+z);
  copyVector(listRight[iright].normal,nVector);
  normalize(listRight[iright].normal);

  x-=xstep;
  --y;
  z-=zstep;
  nVector[0]-=n0step;
  nVector[1]-=n1step;
  nVector[2]-=n2step;
}
listRight[iright].pos[0]=(int)(0.5+x);
listRight[iright].pos[1]=y;
```

```
  listRight[iright].pos[2]=(int)(0.5+z);
  copyVector(listRight[iright].normal,nVector);
  normalize(listRight[iright].normal);
}

/* case 2
   x
   xx
   xxx
   xxxx
   xxx
   xx
   x
*/
else {
  /* Travel from pointHigh to pointLow */
  x=pointHigh.pos[0];
  y=pointHigh.pos[1];
  z=pointHigh.pos[2];
  copyVector(nVector,pointHigh.normal);

  ydiff=pointHigh.pos[1]-pointLow.pos[1];

  if(ydiff != 0) {
    xstep=(pointHigh.pos[0]-pointLow.pos[0]) / ydiff;
    zstep=(pointHigh.pos[2]-pointLow.pos[2]) / ydiff;
    n0step=(pointHigh.normal[0]-pointLow.normal[0]) / ydiff;
    n1step=(pointHigh.normal[1]-pointLow.normal[1]) / ydiff;
    n2step=(pointHigh.normal[2]-pointLow.normal[2]) / ydiff;
  }
  else {
    xstep=0.0;
    zstep=0.0;
    n0step=0.0;
    n1step=0.0;
    n2step=0.0;
  }

  for(iright=0; iright<ydiff; ++iright) {
    listRight[iright].pos[0]=(int)(0.5+x);
    listRight[iright].pos[1]=y;
    listRight[iright].pos[2]=(int)(0.5+z);
    copyVector(listRight[iright].normal,nVector);
    normalize(listRight[iright].normal);
```

```
  x-=xstep;
  --y;
  z-=zstep;
  nVector[0]-=n0step;
  nVector[1]-=n1step;
  nVector[2]-=n2step;
}

listRight[iright].pos[0]=(int)(0.5+x);
listRight[iright].pos[1]=y;
listRight[iright].pos[2]=(int)(0.5+z);
copyVector(listRight[iright].normal,nVector);
normalize(listRight[iright].normal);

/* Travle from pointHigh to pointThree */
x=pointHigh.pos[0];
y=pointHigh.pos[1];
z=pointHigh.pos[2];
copyVector(nVector,pointHigh.normal);

ydiff=pointHigh.pos[1]-pointThree.pos[1];
if((pointHigh.pos[1]-pointThree.pos[1]) != 0) {
  xstep=(pointHigh.pos[0]-pointThree.pos[0]) / ydiff;
  zstep=(pointHigh.pos[2]-pointThree.pos[2]) / ydiff;
  n0step=(pointHigh.normal[0]-pointThree.normal[0]) / ydiff;
  n1step=(pointHigh.normal[1]-pointThree.normal[1]) / ydiff;
  n2step=(pointHigh.normal[2]-pointThree.normal[2]) / ydiff;
}
else {
  xstep=0.0;
  zstep=0.0;
  n0step=0.0;
  n1step=0.0;
  n2step=0.0;
}

for(ileft=0; ileft<ydiff; ++ileft) {
  listLeft[ileft].pos[0]=(int)(0.5+x);
  listLeft[ileft].pos[1]=y;
  listLeft[ileft].pos[2]=(int)(0.5+z);
  copyVector(listLeft[ileft].normal,nVector);
  normalize(listLeft[ileft].normal);
```

```
  x-=xstep;
  --y;
  z-=zstep;
  nVector[0]-=n0step;
  nVector[1]-=n1step;
  nVector[2]-=n2step;
}

listLeft[ileft].pos[0]=(int)(0.5+x);
listLeft[ileft].pos[1]=y;
listLeft[ileft].pos[2]=(int)(0.5+z);
copyVector(listLeft[ileft].normal,nVector);
normalize(listLeft[ileft].normal);

/* Travel from pointThree to pointLow */
x=pointThree.pos[0];
y=pointThree.pos[1];
z=pointThree.pos[2];
copyVector(nVector,pointThree.normal);

ydiff=pointThree.pos[1]-pointLow.pos[1];
if((pointThree.pos[1]-pointLow.pos[1]) != 0) {
  xstep=(pointThree.pos[0]-pointLow.pos[0]) / ydiff;
  zstep=(pointThree.pos[2]-pointLow.pos[2]) / ydiff;
  n0step=(pointThree.normal[0]-pointLow.normal[0]) / ydiff;
  n1step=(pointThree.normal[1]-pointLow.normal[1]) / ydiff;
  n2step=(pointThree.normal[2]-pointLow.normal[2]) / ydiff;

}
else {
  xstep=0.0;
  zstep=0.0;
  n0step=0.0;
  n1step=0.0;
  n2step=0.0;
}

for(ileft=ileft; ileft<iright; ++ileft) {
  listLeft[ileft].pos[0]=(int)(0.5+x);
  listLeft[ileft].pos[1]=y;
  listLeft[ileft].pos[2]=(int)(0.5+z);
  copyVector(listLeft[ileft].normal,nVector);
  normalize(listLeft[ileft].normal);
  x-=xstep;
```

```
      --y;
      z-=zstep;
      nVector[0]-=n0step;
      nVector[1]-=n1step;
      nVector[2]-=n2step;
    }
    listLeft[ileft].pos[0]=(int)(0.5+x);
    listLeft[ileft].pos[1]=y;
    listLeft[ileft].pos[2]=(int)(0.5+z);
    copyVector(listLeft[ileft].normal,nVector);
    normalize(listLeft[ileft].normal);
  }
  return iright;
}

void drawBuffer() {
  int i, j;
  glBegin(GL_POINTS);
  for(i=0; i<FBUFFERSIZE; ++i)
    for(j=0; j<FBUFFERSIZE; ++j) {
      glColor3fv(fBuffer[i][j].color);
      glVertex2i(i+1,j+1);
    }
  glEnd();
}

void display(void) {
  int i,j,k,pointsInList,x,y,xdiff;
  float C,dp,dq,d2q,p,q,z,zstep,cstep,n0step,n1step,n2step,c;
  float A, B, d2p;
  Vector colorpoint, nVector, tempVector, K, M, n;

  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glLoadIdentity();
  glTranslatef(0.0,0.0,-400.0);
  initfbuffer();

  /*draw face normals*/
  if(faceNormals) {
    printf("Drawing face normals...\n");
    glBegin(GL_LINES);
    glColor3f(0.0,0.0,1.0);
    for(i=0; i<NUM_OF_FACES; ++i)
      if(faces[i].midpoint[2]>=(FBUFFERSIZE/2)) {
```

```
glVertex2f(faces[i].midpoint[0],faces[i].midpoint[1]);
glVertex2f((faces[i].midpoint[0]+50*faces[i].normal[0]),
   (faces[i].midpoint[1]+50*faces[i].normal[1]));
      }
    glEnd();
    printf("Finished\n");
  }

  if(vertexNormals) {
    printf("Drawing vertex normals...\n");
    /*draw vertex normals*/
    glBegin(GL_LINES);
    glColor3f(0.0,1.0,0.0);
    for(i=0; i<NUM_OF_VERTICES; ++i)
      if(points[i].pos[2]>=(FBUFFERSIZE/2)) {
glVertex2f(points[i].pos[0],points[i].pos[1]);
glVertex2f((points[i].pos[0]+50*points[i].normal[0]),
   (points[i].pos[1]+50*points[i].normal[1]));
      }
    glEnd();
    printf("Finished\n");
  }

  if(!wireModel) {
    switch(shadingType) {
    case FLAT:
      printf("Scanning polygons...\n");
      timeBefore=clock();
      for(k=0; k<shadingLoopCounter; ++k) {
/* draw triangles FLAT SHADING*/
for(i=0; i<NUM_OF_FACES; ++i) {
  /* Back-Face Removal */
  if(rotated && faces[i].normal[2] > 0)
    continue;
   if(!rotated && faces[i].normal[2] < 0)
    continue;
  /* Calculate color for the face */
  calcShading(faces[i].normal,colorpoint);


  pointsInList=makeFlatLists(i);

  for(j=0; j<=pointsInList; ++j) {
    xdiff=listRight[j].pos[0]-listLeft[j].pos[0];
```

```
    if(xdiff != 0)
      zstep=(listRight[j].pos[2]-listLeft[j].pos[2]) / xdiff;
    else
      zstep=0.0;

    y=listLeft[j].pos[1];
    z=listLeft[j].pos[2]-zstep;

    for(x=listLeft[j].pos[0];x <=listRight[j].pos[0]; ++x) {
      z+=zstep;
      if(fBuffer[x][y].z<z) {
fBuffer[x][y].z=(int)(0.5+z);
fBuffer[x][y].color[0]=colorpoint[0];
fBuffer[x][y].color[1]=0.0;
fBuffer[x][y].color[2]=0.0;
      }
    }
  }
}
      }
      timeAfter=clock();
      printf("Flat executiontime is %ld\n", (timeAfter-timeBefore));
      printf("Finished\n");
      printf("Drawing frame buffer...\n");
      drawBuffer();
      printf("Finished\n\n");
      break;

    case GOURAUD:
      printf("Scanning polygons...\n");
      timeBefore=clock();
      for(k=0; k<shadingLoopCounter; ++k) {
/* draw triangles GOURAUD SHADING*/
for(i=0; i<NUM_OF_FACES; ++i) {

  /* Back-Face Removal */
  if(rotated && faces[i].normal[2] > 0)
    continue;
  if(!rotated && faces[i].normal[2] < 0)
    continue;

  pointsInList=makeGouraudLists(i);
  for(j=0; j<=pointsInList; ++j) {
    xdiff=listRight[j].pos[0]-listLeft[j].pos[0];
```

```
    if(xdiff != 0) {
      zstep=(listRight[j].pos[2]-listLeft[j].pos[2]) / xdiff;
      cstep=(listRight[j].color[0]-listLeft[j].color[0]) / xdiff;
    }
    else {
      zstep=0.0;
      cstep=0.0;
    }
    y=listLeft[j].pos[1];
    z=listLeft[j].pos[2]-zstep;
    c=listLeft[j].color[0]-cstep;
    for(x=listLeft[j].pos[0];x <=listRight[j].pos[0]; ++x) {
      z+=zstep;
      c+=cstep;
      if(fBuffer[x][y].z<z) {
fBuffer[x][y].z=(int)(0.5+z);
fBuffer[x][y].color[0]=c;
fBuffer[x][y].color[1]=0.0;
fBuffer[x][y].color[2]=0.0;
      }
    }
  }
}
      }
      timeAfter=clock();
      printf("Gouraud executiontime is %ld\n", (timeAfter-timeBefore));
      printf("Finished\n");
      printf("Drawing frame buffer...\n");
      drawBuffer();
      printf("Finished\n\n");
      break;

    case PHONG:
      printf("Scanning polygons...\n");
      timeBefore=clock();
      for(k=0; k<shadingLoopCounter; ++k) {
/* draw triangles PHONG SHADING*/
for(i=0; i<NUM_OF_FACES; ++i) {

  /* Back-Face Removal */
  if(rotated && faces[i].normal[2] > 0)
    continue;
  if(!rotated && faces[i].normal[2] < 0)
    continue;
```

```
  pointsInList=makePhongLists(i);
  for(j=0; j<=pointsInList; ++j) {
    xdiff=listRight[j].pos[0]-listLeft[j].pos[0];
    if(xdiff != 0) {
      zstep=(listRight[j].pos[2]-listLeft[j].pos[2]) / xdiff;
      n0step=(listRight[j].normal[0]-listLeft[j].normal[0]) / xdiff;
      n1step=(listRight[j].normal[1]-listLeft[j].normal[1]) / xdiff;
      n2step=(listRight[j].normal[2]-listLeft[j].normal[2]) / xdiff;
    }
    else {
      zstep=0.0;
      n0step=0.0;
      n1step=0.0;
      n2step=0.0;
    }
    y=listLeft[j].pos[1];
    z=listLeft[j].pos[2]-zstep;
    nVector[0]=listLeft[j].normal[0]-n0step;
    nVector[1]=listLeft[j].normal[1]-n1step;
    nVector[2]=listLeft[j].normal[2]-n2step;
    for(x=listLeft[j].pos[0];x <=listRight[j].pos[0]; ++x) {
      z+=zstep;
      nVector[0]+=n0step;
      nVector[1]+=n1step;
      nVector[2]+=n2step;
      if(fBuffer[x][y].z<z) {
copyVector(tempVector, nVector);
normalize(tempVector);
calcShading(tempVector,colorpoint);
fBuffer[x][y].z=(int)(0.5+z);
fBuffer[x][y].color[0]=colorpoint[0];
fBuffer[x][y].color[1]=0.0;
fBuffer[x][y].color[2]=0.0;
      }
    }
  }
}
      }
      timeAfter=clock();
      printf("Phong executiontime is %ld\n", (timeAfter-timeBefore));
      printf("Finished\n");
      printf("Drawing frame buffer...\n");
      drawBuffer();
```

```
      printf("Finished\n\n");
      break;

    case DUFF:
      printf("Scanning polygons...\n");
      timeBefore=clock();
      for(k=0; k<shadingLoopCounter; ++k) {
/* draw triangles DUFF SHADING*/
for(i=0; i<NUM_OF_FACES; ++i) {

  /* Back-Face Removal */
  if(rotated && faces[i].normal[2] > 0)
    continue;
  if(!rotated && faces[i].normal[2] < 0)
    continue;

  pointsInList=makePhongLists(i);
  for(j=0; j<=pointsInList; ++j) {
    copyVector(M,listLeft[j].normal);
    subVector(listLeft[j].normal,listRight[j].normal,K);
    xdiff=listRight[j].pos[0]-listLeft[j].pos[0];
    divVectorScaler(K,xdiff);

    dp=dotProduct(lightDir,K); /* dp=A=K dot L */
    p=dotProduct(M,lightDir);  /* p=N dot L  */
    C=dotProduct(K,K);
    /*   D=2*dotProduct(K,M);*/
    /*   E=1;*/
    q=1;
    dq=C+2*dotProduct(K,M);
    d2q=2*C;

    if(xdiff != 0) {
      zstep=(listRight[j].pos[2]-listLeft[j].pos[2]) / xdiff;
    }
    else {
      zstep=0.0;
    }
    y=listLeft[j].pos[1];
    z=listLeft[j].pos[2]-zstep;
    for(x=listLeft[j].pos[0];x <=listRight[j].pos[0]; ++x) {
      z+=zstep;
      if(fBuffer[x][y].z<=z) {
colorpoint[0]=p/(sqrt(q));
```

```
if(colorpoint[0]<0)
  colorpoint[0]=0;
fBuffer[x][y].z=(int)(0.5+z);
fBuffer[x][y].color[0]=colorpoint[0];
fBuffer[x][y].color[1]=0.0;
fBuffer[x][y].color[2]=0.0;
      }
      p+=dp;
      q+=dq;
      dq+=d2q;
    }
  }
}
      }
      timeAfter=clock();
      printf("Duff executiontime is %ld\n", (timeAfter-timeBefore));
      printf("Finished\n");
      printf("Drawing frame buffer...\n");
      drawBuffer();
      printf("Finished\n\n");
      break;

    case REFLECTION:
      printf("Scanning polygons...\n");
      timeBefore=clock();
      for(k=0; k<shadingLoopCounter; ++k) {
/* draw triangles REFLECTION SHADING*/
for(i=0; i<NUM_OF_FACES; ++i) {

  /* Back-Face Removal */
  if(rotated && faces[i].normal[2] > 0)
    continue;
  if(!rotated && faces[i].normal[2] < 0)
    continue;

  pointsInList=makePhongLists(i);
  copyVector(n,faces[i].normal);
  for(j=0; j<=pointsInList; ++j) {
    copyVector(M,listLeft[j].normal);
    subVector(listLeft[j].normal,listRight[j].normal,K);
    xdiff=listRight[j].pos[0]-listLeft[j].pos[0];
    divVectorScaler(K,xdiff);
    addVector(M,n);
```

```
  /*   A1=dotProduct(K,lightDir);
A2=dotProduct(K,n);
B1=dotProduct(M,lightDir);
B2=dotProduct(M,n);*/

  /*   Aa1=K[2]; *//*K.z*/
  /*Bb1=M[2];*/ /*M.z*/

  A=2*dotProduct(K,lightDir)*dotProduct(K,n); /* A=2*A1*A2 */
  B=2*(dotProduct(K,lightDir)*dotProduct(M,n)+dotProduct(K,n)
  *dotProduct(M,lightDir)); /* B=2*(A1*B2+A2*B1) */
  C=2*dotProduct(M,lightDir)*dotProduct(M,n); /* C=2*B1*B2 */

  /* Aa=2*K[2]*dotProduct(K,n);
     Bb=2*(K[2]*dotProduct(M,n)+dotProduct(K,n)*M[2]);
     Cc=2*M[2]*dotProduct(M,n);
  */

  /*
D=dotProduct(K,K);
E=2*dotProduct(K,M);
F=dotProduct(M,M);
  */

  /*   G=dotProduct(n,lightDir);*/

  A-=(dotProduct(n,lightDir)*dotProduct(K,K)); /* A-=G*D */
  B-=(dotProduct(n,lightDir)*2*dotProduct(K,M)); /* B-=G*E */
  C-=(dotProduct(n,lightDir)*dotProduct(M,M)); /* C-=G*F */

  /*Gg=n[2];*/ /*n.z*/

  /*
    Aa-=n[2]*dotProduct(K,K);
    Bb-=n[2]*2*dotProduct(K,M);
    Cc-=n[2]*dotProduct(M,M);
  */

  p=C;
  dp=A+B;
  d2p=2*A;

  q=dotProduct(M,M); /* q=F */
  dq=dotProduct(K,K)+2*dotProduct(K,M); /* dq=D+E */
```

```
    d2q=2*dotProduct(K,K); /* d2q=2*D */

    if(xdiff != 0) {
      zstep=(listRight[j].pos[2]-listLeft[j].pos[2]) / xdiff;
    }
    else {
      zstep=0.0;
    }
    y=listLeft[j].pos[1];
    z=listLeft[j].pos[2]-zstep;
    for(x=listLeft[j].pos[0];x <=listRight[j].pos[0]; ++x) {
      z+=zstep;
      if(fBuffer[x][y].z<=z) {
colorpoint[0]=p/q;
if(colorpoint[0]<0)
  colorpoint[0]=0;
fBuffer[x][y].z=(int)(0.5+z);
fBuffer[x][y].color[0]=colorpoint[0];
fBuffer[x][y].color[1]=0.0;
fBuffer[x][y].color[2]=0.0;
      }
      p+=dp;
      dp+=d2p;
      q+=dq;
      dq+=d2q;
    }
  }
}
      }
      timeAfter=clock();
      printf("Reflection executiontime is %ld\n", (timeAfter-timeBefore));
      printf("Finished\n");
      printf("Drawing frame buffer...\n");
      drawBuffer();
      printf("Finished\n\n");
      break;
    case 0:
      printf("Program finished\n");
      exit(0);
      break;
    }
  }

  /*draw wire model*/
```

```
  if(wireModel) {
    printf("Drawing wire model...\n");
    j=0;
    for(i=0; i<NUM_OF_VERTICES; i+=3) {
      glBegin(GL_LINE_LOOP);
      glColor3fv(faces[j].color);
      glVertex2f(points[i+0].pos[0],points[i+0].pos[1]);
      glVertex2f(points[i+1].pos[0],points[i+1].pos[1]);
      glVertex2f(points[i+2].pos[0],points[i+2].pos[1]);
      glEnd();
      ++j;
    }
    printf("Finished\n");
  }
  glFlush();
  glutSwapBuffers();
}

void menu(int id) {
  shadingType=id;
  switch(id) {
  case 0:
    free(points);
    free(faces);
    exit(0);
    break;
  }
  glutPostRedisplay();
}
void shadingMenu(int id) {
  shadingType=id;
  glutPostRedisplay();
  shadingLoopCounter=1;
}

void specialMenu(int id) {
  switch(id) {
  case 1:
    if(faceNormals==FALSE)
      faceNormals=TRUE;
    else
      faceNormals=FALSE;
    break;
  case 2:
```

```
    if(vertexNormals==FALSE)
      vertexNormals=TRUE;
    else
      vertexNormals=FALSE;
    break;
  case 3:
    if(wireModel==FALSE)
      wireModel=TRUE;
    else
      wireModel=FALSE;
    break;
  }
  glutPostRedisplay();
}

void rotateMenu(int id) {
  switch(id) {
  case 2:
    /* The model is loaded again, scaled and rotated,
       but the old normals calculations are used */
    initfbuffer();
    loadModel(modelfile);
    rotation(180);
    scaleModel();
    rotated=TRUE;
    break;
  case 3:
    /* The model is loaded again, scaled,
       but the old normals calculations are used */
    initfbuffer();
    loadModel(modelfile);
    scaleModel();
    rotated=FALSE;
    break;
  }
  switchDir=FALSE;
  copyVector(lightDir, lightAlt1);
  normalize(lightDir);
  glutPostRedisplay();
}

void timeMenu(int id) {
  shadingType=id;
  glutPostRedisplay();
```

```
  shadingLoopCounter=10;
}

void lightMenu(int id) {
  Vector temp={1.0, 1.0, 1.0};
  switch(id) {
  case 1:
    copyVector(lightDir, lightAlt1);
    if(switchDir==TRUE)
      setVector(temp,-1.0, -1.0, -1.0);
    break;
  case 2:
    copyVector(lightDir, lightAlt2);
    if(switchDir==TRUE)
      setVector(temp, -1.0, -1.0, -1.0);
    break;
  case 3:
    copyVector(lightDir, lightAlt3);
    if(switchDir==TRUE)
      setVector(temp, -1.0, -1.0, -1.0);
    break;
  case 4:
    if(switchDir==FALSE)
      switchDir=TRUE;
    else
      switchDir=FALSE;
    setVector(temp, -1.0, -1.0, -1.0);
    break;
  }

  multVector(lightDir, temp);
  normalize(lightDir);
  glutPostRedisplay();
}

void myinit() {
  glClearColor(1.0, 1.0, 1.0, 1.0);
  glColor3f(1.0, 0.0, 0.0);

  /* set up viewing */
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho(0.0, FBUFFERSIZE, 0.0, FBUFFERSIZE, 1.0, 800.0);
  glMatrixMode(GL_MODELVIEW);
```

```
}

int main(int argc, char** argv) {
  /* Standard GLUT initialization */
  int lMenu, rMenu, mMenu, sMenu, specMenu, tMenu;

  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
  glutInitWindowSize(FBUFFERSIZE, FBUFFERSIZE);
  glutInitWindowPosition(0, 0);
  glutCreateWindow("PolLac");
  glutDisplayFunc(display);
  myinit();
  sMenu=glutCreateMenu(shadingMenu);
  glutAddMenuEntry("Flat", 1);
  glutAddMenuEntry("Guraud", 2);
  glutAddMenuEntry("Phong", 3);
  glutAddMenuEntry("Duff", 4);
  glutAddMenuEntry("Reflection", 5);

  specMenu=glutCreateMenu(specialMenu);
  glutAddMenuEntry("Face normals", 1);
  glutAddMenuEntry("Vertex nornals", 2);
  glutAddMenuEntry("Wire model on / off", 3);

  tMenu=glutCreateMenu(timeMenu);
  glutAddMenuEntry("Flat 100 times", 1);
  glutAddMenuEntry("Guraud 100 times", 2);
  glutAddMenuEntry("Phong 100 times", 3);
  glutAddMenuEntry("Duff 100 times", 4);
  glutAddMenuEntry("Reflection 100 times", 5);

  lMenu=glutCreateMenu(lightMenu);
  glutAddMenuEntry("From viewer", 1);
  glutAddMenuEntry("From viewer, upper right", 2);
  glutAddMenuEntry("From viewer, upper right more to center", 3);
  glutAddMenuEntry("Switch light direction to the opposite", 4);

  rMenu=glutCreateMenu(rotateMenu);
  glutAddMenuEntry("Rotate 180", 2);
  glutAddMenuEntry("Front", 3);

  mMenu=glutCreateMenu(menu);
  glutAddSubMenu("Shading types", sMenu);
```

```
  glutAddSubMenu("Rotate", rMenu);
  glutAddSubMenu("Timing",tMenu);
  glutAddSubMenu("Light", lMenu);
  glutAddSubMenu("Special menu", specMenu);
  glutAddMenuEntry("Exit", 0);

  glutAttachMenu(GLUT_RIGHT_BUTTON);

  points = (Vertex *) malloc(990000*sizeof(Vertex));
  if(points==NULL)
    exit(0);

  faces = (Face *)malloc(330000*sizeof(Face));
  if(faces==NULL)
    exit(0);

  initfbuffer();
  if(argc==2)
    strcpy(modelfile, argv[1]);
  /* No raw file as argument - load sphere2 as default model */
  else
    strcpy(modelfile,"3dmodels/sphere3.raw");
  loadModel(modelfile);
  printf("Init face...");
  initFace();
  printf("Finished\n");
  printf("Calc vertex normals...");
  calcVertexNormal();
  printf("Finished\n");
  scaleModel();
  calcFaceMidPoint();
  glEnable(GL_DEPTH_TEST);
  glutMainLoop();
  exit(0);
}
```

9.3 vector.h

```
/**
 * File: vector.h, headerfile to vector.c
 *
 * Auther: Björn Nystedt and Marko Pirttioja
 *
 * Last modified: 20010516
 *
 * Version: 1.0
 *
 **/

#ifndef _VECTOR_H_
#define _VECTOR_H_
#include <math.h>

/**
 * Data structure for a vector
 **/
typedef float Vector[3];

/**
 * Pre :
 * Post: returns the dotproduct of two vectors
 **/
float dotProduct(Vector, Vector);

/**
 * Pre :
 * Post: returns the lenght of the vector
 **/
float lenght(Vector);

/**
 * Pre :
 * Post: normalize the vector
 **/
void normalize(Vector);

/**
 * Pre :
 * Post: calculate the crossproduct of two vectors,
 *       stored in the third arguemnt
```

```
 **/
void crossProd(Vector, Vector, Vector);

/**
 * Pre :
 * Post: return an area of a triangle
 **/
float area(Vector, Vector, Vector);

/**
 * Pre :
 * Post: subtract two vectors, stored in the third argument
 **/
void subVector(Vector, Vector, Vector);

/**
 * Pre :
 * Post: returns 1 if equal otherwise 0
 **/
int isEqual(Vector, Vector);

/**
 * Pre :
 * Post: add two vectors, stored in the first argument
 **/
void addVector(Vector, Vector);

/**
 * Pre :
 * Post: multiplies a scaler to the vector
 **/
void multVectorScaler(Vector, float);

/**
 * Pre :
 * Post: div a scaler with the vector
 **/
void divVectorScaler(Vector, float);

/**
 * Pre :
 * Post: copy a vector, stored in first argument
 **/
void copyVector(Vector, Vector);
```

```
/**
 * Pre :
 * Post: sets a vector
 **/
void setVector(Vector, float, float, float);

/**
 * Pre :
 * Post: calculate the midpoint of a triangle,
 *       stored in the fourth argument
 **/

void calcMidPoint(Vector, Vector, Vector, Vector);
/**
 * Pre :
 * Post: multiplicate two vectors,
 *       stored in the first argument
 **/
void multVector(Vector v1, Vector v2);

#endif
```

9.4 vector.c

```
/**
 * File: vector.c
 *
 * Auther: Björn Nystedt and Marko Pirttioja
 *
 * Last modified: 20010516
 *
 * Version: 1.0
 *
 **/

#include "vector.h"

float dotProduct(Vector v1, Vector v2) {
  float sum=0;
  sum+=v1[0]*v2[0];
  sum+=v1[1]*v2[1];
  sum+=v1[2]*v2[2];
  return sum;
}

float lenght(Vector v) {
  return sqrt(v[0]*v[0]+
      v[1]*v[1]+
      v[2]*v[2]);
}

void normalize(Vector v) {
  float len=lenght(v);
  float scalfactor=1.0/len;
  v[0]=v[0]*scalfactor;
  v[1]=v[1]*scalfactor;
  v[2]=v[2]*scalfactor;
}

void crossProd(Vector v1, Vector v2, Vector v) {
  v[0]=(v1[1]*v2[2])-(v1[2]*v2[1]);
  v[1]=(v1[2]*v2[0])-(v1[0]*v2[2]);
  v[2]=(v1[0]*v2[1])-(v1[1]*v2[0]);
}

float area(Vector v1, Vector v2, Vector v3) {
```

```
  return ((v3[1] - v1[1])*(v2[0] - v1[0])-
  (v3[0] - v1[0])*(v2[1] - v1[1]));
}

void subVector(Vector v1, Vector v2, Vector v) {
  v[0]=v2[0]-v1[0];
  v[1]=v2[1]-v1[1];
  v[2]=v2[2]-v1[2];
}

int isEqual(Vector v1, Vector v2) {
  int i, equal=1;
  for(i=0; i<3; i++)
    if(v1[i]!=v2[i])
      equal=0;
  return equal;
}

void addVector(Vector v1, Vector v2) {
  v1[0]+=v2[0];
  v1[1]+=v2[1];
  v1[2]+=v2[2];
}

void multVectorScaler(Vector v1, float v2) {
  v1[0]*=v2;
  v1[1]*=v2;
  v1[2]*=v2;
}

void divVectorScaler(Vector v1, float v2) {
  v1[0]/=v2;
  v1[1]/=v2;
  v1[2]/=v2;
}

void copyVector(Vector v1, Vector v2) {
  v1[0]=v2[0];
  v1[1]=v2[1];
  v1[2]=v2[2];
}

void setVector(Vector v1, float f0, float f1, float f2) {
  v1[0]=f0;
```

```
  v1[1]=f1;
  v1[2]=f2;
}

void calcMidPoint(Vector v1, Vector v2, Vector v3, Vector v) {
  v[0]=(v1[0]+v2[0]+v3[0])/3;
  v[1]=(v1[1]+v2[1]+v3[1])/3;
  v[2]=(v1[2]+v2[2]+v3[2])/3;
}
void multVector(Vector v1, Vector v2) {
  v1[0]*=v2[0];
  v1[1]*=v2[1];
  v1[2]*=v2[2];
}
```

9.5 sp.c

```
/**
 * Recursive subdivision of cube.
 **/

#include <stdio.h>
#include <stdlib.h>
#include <GL/glut.h>


typedef float point[4];

/* initial tetrahedron */
point v[]={{0.0, 0.0, 1.0}, {0.0, 0.942809, -0.33333},
   {-0.816497, -0.471405, -0.333333},
   {0.816497, -0.471405, -0.333333}};

static GLfloat theta[] = {0.0,0.0,0.0};

int n=6;
int mode=0;

FILE *fp;

void triangle( point a, point b, point c)

/* display one triangle using a line loop for wire frame, a single
normal for constant shading, or three normals for interpolative shading */
{
  fprintf(fp, "%0.5f ", a[0]);
  fprintf(fp, "%0.5f ", a[1]);
  fprintf(fp, "%0.5f ", a[2]);
  fprintf(fp, "%0.5f ", b[0]);
  fprintf(fp, "%0.5f ", b[1]);
  fprintf(fp, "%0.5f ", b[2]);
  fprintf(fp, "%0.5f ", c[0]);
  fprintf(fp, "%0.5f ", c[1]);
  fprintf(fp, "%0.5f\n", c[2]);

if (mode==0) glBegin(GL_LINE_LOOP);
    else glBegin(GL_POLYGON);
       if(mode==1) glNormal3fv(a);
       if(mode==2) glNormal3fv(a);
```

```
       glVertex3fv(a);
       if(mode==2) glNormal3fv(b);
       glVertex3fv(b);
       if(mode==2) glNormal3fv(c);
       glVertex3fv(c);
    glEnd();

}

void normal(point p)
{
/* normalize a vector */
    double sqrt();
    float d =0.0;
    int i;
    for(i=0; i<3; i++) d+=p[i]*p[i];
    d=sqrt(d);
    if(d>0.0) for(i=0; i<3; i++) p[i]/=d;
}

void divide_triangle(point a, point b, point c, int m)
{
/* triangle subdivision using vertex numbers
righthand rule applied to create outward pointing faces */
    point v1, v2, v3;
    int j;
    if(m>0)
    {
        for(j=0; j<3; j++) v1[j]=a[j]+b[j];
        normal(v1);
        for(j=0; j<3; j++) v2[j]=a[j]+c[j];
        normal(v2);
        for(j=0; j<3; j++) v3[j]=b[j]+c[j];
         normal(v3);
        divide_triangle(a, v1, v2, m-1);
        divide_triangle(c, v2, v3, m-1);
        divide_triangle(b, v3, v1, m-1);
        divide_triangle(v1, v3, v2, m-1);
    }
    else(triangle(a,b,c)); /* draw triangle at end of recursion */
}

void tetrahedron( int m)
{
```

```
/* Apply triangle subdivision to faces of tetrahedron */
    divide_triangle(v[0], v[1], v[2], m);
    divide_triangle(v[3], v[2], v[1], m);
    divide_triangle(v[0], v[3], v[1], m);
    divide_triangle(v[0], v[2], v[3], m);
}

void display(void)
{

/* Displays all three modes, side by side */
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glLoadIdentity();
  mode=0;
  tetrahedron(n);
  fclose(fp);
  glFlush();
}


void myinit()
{
  glClearColor(1.0, 1.0, 1.0, 1.0);
  glColor3f(1.0, 0.0, 0.0);

  /* set up viewing */
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho(-2.0, 2.0, -2.0, 2.0, -10.0, 10.0);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
}


int main(int argc, char** argv)
{
  /* Standard GLUT initialization */
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
  glutInitWindowSize(500, 500);
  glutInitWindowPosition(0, 0);
  glutCreateWindow("My window!");
  myinit();
```

```
  glutDisplayFunc(display);
  /* enable depth test */
  glEnable(GL_DEPTH_TEST);
  fp=fopen("3dmodels/sphere6.raw", "w");
  glutMainLoop();
}
```

9.6 sphere0.raw

```
0.00000 0.00000 1.00000 0.00000 0.94281 -0.33333 -0.81650 -0.47140 -0.33333
0.81650 -0.47140 -0.33333 -0.81650 -0.47140 -0.33333 0.00000 0.94281 -0.33333
0.00000 0.00000 1.00000 0.81650 -0.47140 -0.33333 0.00000 0.94281 -0.33333
0.00000 0.00000 1.00000 -0.81650 -0.47140 -0.33333 0.81650 -0.47140 -0.33333
```

9.7 sphere1.raw

```
0.00000 0.00000 1.00000 0.00000 0.81650 0.57735 -0.70711 -0.40825 0.57735
-0.81650 -0.47140 -0.33333 -0.70711 -0.40825 0.57735 -0.70711 0.40825 -0.57735
0.00000 0.94281 -0.33333 -0.70711 0.40825 -0.57735 0.00000 0.81650 0.57735
0.00000 0.81650 0.57735 -0.70711 0.40825 -0.57735 -0.70711 -0.40825 0.57735
0.81650 -0.47140 -0.33333 0.00000 -0.81650 -0.57735 0.70711 0.40825 -0.57735
0.00000 0.94281 -0.33333 0.70711 0.40825 -0.57735 -0.70711 0.40825 -0.57735
-0.81650 -0.47140 -0.33333 -0.70711 0.40825 -0.57735 0.00000 -0.81650 -0.57735
0.00000 -0.81650 -0.57735 -0.70711 0.40825 -0.57735 0.70711 0.40825 -0.57735
0.00000 0.00000 1.00000 0.70711 -0.40825 0.57735 0.00000 0.81650 0.57735
0.00000 0.94281 -0.33333 0.00000 0.81650 0.57735 0.70711 0.40825 -0.57735
0.81650 -0.47140 -0.33333 0.70711 0.40825 -0.57735 0.70711 -0.40825 0.57735
0.70711 -0.40825 0.57735 0.70711 0.40825 -0.57735 0.00000 0.81650 0.57735
0.00000 0.00000 1.00000 -0.70711 -0.40825 0.57735 0.70711 -0.40825 0.57735
0.81650 -0.47140 -0.33333 0.70711 -0.40825 0.57735 0.00000 -0.81650 -0.57735
-0.81650 -0.47140 -0.33333 0.00000 -0.81650 -0.57735 -0.70711 -0.40825 0.57735
-0.70711 -0.40825 0.57735 0.00000 -0.81650 -0.57735 0.70711 -0.40825 0.57735
```

9.8 Shaded Models