Introduction to Computer Graphics:
Visualization of Parametric Surfaces

010582 Dirk Hasselbalch 250963 Michael Neidhardt 300679 Morten Poulsen

October 12, 2006
Abstract

Computer graphics is widely used, and seems to grow in importance. The present report attempts to analyse the central aspects of 3D visualisation, to find good design solutions and to create code that might possibly be of use to us later, but as such it contains nothing new. The main areas of interest have been scan conversion of triangles, how to handle surfaces in 3D, the projection of 3D objects onto 2D planes and how to apply light and shading to objects. We have been able to create all the required visualisations: The general parametric surface and the 2 objects defined by Bezier patches. We think that illumination and shading has been done with a certain success, at least based on visual inspection. There are minor errors in the final images that we would have preferred to remove, but time has not allowed us to do that.
Introduction

This report details our solution to the assignment in the course "Introduktion til grafik", at Department of Computer Science, Copenhagen University. The task in [Hen06] is to analyse, construct, present and implement a 3D visualisation system in C++. This is reflected in the report, in that we have analysed the particular topics in detail and chosen an appropriate solution in each case. We describe design considerations and an actual implementation. The field of computer graphics is vast, so this is by no means an exhaustive analysis, but we have tried to include as much as we could find the time for. We assume the reader knows nothing about 3D visualisation, but is familiar with programming, linear algebra and geometry. It is our intention that the analysis and design parts should in themselves be enough to give the reader an understanding of the problems and solutions we have encountered. Most of the theory is well documented in [FvDFH97].
1 Analysis

1.1 Polygons and triangles

A polygon, in the general case, is a series of points and edges, where the edges interconnect the points and thereby enclose some region. A polygon can be convex or concave. A convex polygon is a polygon, where lines between all pairs of points are all inside the polygon. A concave polygon is a polygon where at least one line between a pair of points is outside the polygon.

![Figure 1.1: Left: A concave polygon. Middle: A convex polygon. Right: The convex polygon divided into triangles.](image)

All polygons can be converted into a number of triangles. In the case where the polygon is convex this is a trivial task. In the field of computer graphics this means that the term polygon is often synonymous with triangle. From this point forward we will also consider the two terms equal since the polygons in this project are so simple that they in fact are triangles. We could have chosen not to do this, but that would make our polygon filling algorithm more complex since we would have to take the number of edges into account along with the possibility of concave polygons.

1.1.1 Types of edges and polygons

Polygons can be divided into different types depending on their edges. We will define the following types of edges:

**Top edges** are horizontal edges, where the inside of the polygon is below the edge.

**Bottom edges** are horizontal edges, where the inside of the polygon is above the edge.

**Left edges** are edges, that are neither top nor bottom edges, and where the inside of the polygon is to the right of the edge.
CHAPTER 1. ANALYSIS

Right edges are edges, that are neither top nor bottom edges, and where the inside of the polygon is to the left of the edge.

With this handy, we will define the following types of polygons:

Bottom edge polygon is a polygon with one bottom edge.

Top edge polygon is a polygon with one top edge.

Left edge dominated polygon is a polygon with two left edges and one right edge.

Right edge dominated polygon is a polygon with two right edges and one left edge.

Note that these types are mutually exclusive when we limit our polygons to have non-parallel edges.

1.1.2 Inside or outside

Since polygons are now all considered triangles, all polygons are convex. Thus the distinction between inside and outside becomes easy: The inside of a polygon is a point where another point, traveling around the perimeter of the polygon, wraps around the point in question exactly one time.

What if the point in question is on an edge? In that case we will have to make a consistent decision, since eventually we will have to define whether a point is inside or outside a polygon in order to color the corresponding pixel on the screen accordingly. Using the definitions of section 1.1.1 we define the following:

- Points located exactly on a left or bottom edge of a polygon is considered to be inside the polygon.

- Points located exactly on a right or top edge of a polygon is considered to be outside the polygon.

- When a clash of the two definitions above occur, for instance a point located exactly on the intersection between a top edge and a left edge, the point is considered to be outside the polygon.

1.2 Filling polygons

Filling polygons is the process of coloring the pixels that are inside a polygon. The main challenge is to design an efficient and simple algorithm that covers the most general case. Efficient in this case means the use of integer operations only since floating point operations are too expensive, and not really necessary.

When working with points and lines in a two dimensional coordinate system, it seems inevitable not to use discrete numbers such as integers. We realize, however, that we do not necessarily need a real point field such as \( \mathbb{R}^2 \) to model these things, but can do just fine with a rational point field such as \( \mathbb{Q}^2 \). Since any rational number consists of an integral numerator and an integral denominator we might have a way to avoid floating point arithmetics!
1.2.1 The Scan Line Algorithm

We will use the following scheme to fill polygons:

1. We make a horizontal scan-line traverse the polygon from bottom to top.

2. At each step we will determine where the scan-line intersects the left and right edge of the polygon.

3. We will then color the points between the right and left intersection along the scan-line according to some shading model, which we will describe in detail in section 1.6.

This is basically the scheme as presented in [FvDFH97, pp. 92 ff], except that in our case polygons are all triangles as mentioned earlier.

1.2.1.1 Finding edge points

When calculating intersections between the scan-line and the left and right edges of our polygon, we wish to exploit the fact that the intersection points are near the intersection points of the scan-line immediately below. By keeping track of $x$’s, $y$’s and the numerator and denominator of the slopes’ coefficients of the left and right edges of our polygon this is possible.

Keeping this in mind, the idea is to find the points on or to the immediate right of an edge. We will pinpoint one point per edge for each $y$-value. Thus, if the line’s slope is less than one and greater than minus one, there will be gaps in the line. This does not concern us, however, since the identified points are solely used as end points in drawing purely horizontal lines between left and right edges.
Figure 1.3: Finding edge points. Crosses are points on or to the immediate right of the polygon's left edge. Circles are points on or to the immediate right of the polygon’s right edges. The grey discs is the result when we color pixels between crosses and circles and include the crosses but omit the circles.

Figure 1.3 shows the idea: The crosses are points on or to the immediate right of edge AC and the circles are points on or to the immediate right of edges AB and BC. The current scan line \( s \) is at \( y = 4 \). When drawing horizontal lines from left to right, we include the left point but exclude the right point. If the two points are the same we do \textit{not} include it.

This is consistent with our choices of when points are considered inside or outside a polygon, cf. section 1.1.2.

Generally, we define the beginning point \( A \) of an edge to be the point with the lowest \( y \)-value. Thus, we define the beginning point \( A = (x_{\text{begin}}, y_{\text{begin}}) \) and the end point \( B = (x_{\text{end}}, y_{\text{end}}) \), where \( y_{\text{begin}} < y_{\text{end}} \). Both points are presumed to have integral \( x \) and \( y \) values. For now, we consider the case where also \( x_{\text{begin}} < x_{\text{end}} \). The edge \( AB \) in figure 1.3 satisfy these conditions and can be used as a case study.

The slope of the edge is \( m = (y_{\text{end}} - y_{\text{begin}}) / (x_{\text{end}} - x_{\text{begin}}) > 0 \). We start by pinpointing the point \( (x_0, y_0) = (x_{\text{begin}}, y_{\text{begin}}) \), which in figure 1.3 corresponds to point \( A = (3, 2) \). In order to find the next point, we first increment our \( y \)-value by one: \( y_1 = y_{\text{begin}} + 1 \). The corresponding \( x \)-value \textit{following the edge} is \( x'_1 = x_{\text{begin}} + 1/m \). Generally, for all \( n \), \( x'_n = x'_{n-1} + 1/m \), where \( x' \) is the \( x \)-coordinate of the intersection between the edge and the scan-line at position \( y_n \). It is clear that the \( x'_n \)s are not necessarily integers.

The \( x_n \) we need to find, equals \( x'_n \textit{ rounded up} \) to the nearest integer. When following the right edge \( AB \) in figure 1.3, for example, we have \( x'_2 = 10 \frac{1}{2} \) (which is the intersection between the depicted scan-line at \( y_2 = 4 \) and the edge \( AB \)), while \( x_2 = 11 \) (which is the circle to the right of the line).

If the slope of the edge is steep such that \( 1/m < 1 \), \( y_n \) might be incremented two or more times without \( x_n \) changing value. If \( 1/m > 1 \) we will need to increment \( x_n \) several times.
without $y_n$ changing value. So when $y_n$ gets incremented, $x_n$ can be incremented basically any number of times. How do we know when to increment $x_n$?

The answer is that we increment $x_n$ the moment the fractional part of $x'_n$ overruns. At the same time, we subtract one from the fraction.

The fraction we are adding to $x'_n$ at each step can be split into a numerator $n$ and a denominator $d$:

$$n/d = 1/m \Rightarrow n = x_{\text{end}} - x_{\text{begin}}, \quad d = y_{\text{end}} - y_{\text{begin}}. \quad (1.1)$$

The numerator of $x'_n$’s fractional part can be used to check whether the fractional part exceeds one by comparing it to the fraction’s denominator. We name this the incrementor $i$ and initialize it with the value of $d$. At all times $x'_n = x'_{n,\text{int}} + x'_{n,\text{frac}} = x'_{n,\text{int}} + i/d$. I.e., if $i/d > 1$ which is the same as asking if $i > d$, then we will need to increment $x_n$ and subtract $d$ from $i$. We repeat these steps until $i$ is once again less than $d$. Then we increment $y_n$ and start over.

In the case where the slope is negative, i.e. $x_{\text{begin}} > x_{\text{end}}$, we need to change some things:

1. Since $x'_n$ will be less than $x'_{n-1}$ we will be decrementing $x_n$ instead of incrementing it.
2. The fractional part signified by the variable $i$ will for the same reason get smaller with each iteration (since $n < 0$). We will therefore not be checking for overruns, but for underruns. Therefore the check on $i$ will consist of checking whether it is less than or equal to zero.
3. After $i$ has underrun, we add $d$ to $i$ until $i$ is once again greater than 0.

All this sums up to the following algorithm:

```c
/* Initializing variables */
x ← x_{\text{begin}}
n ← x_{\text{end}} - x_{\text{begin}}
δx ← sign(n) /* δx determines if we increment or decrement x and i. */
d ← y_{\text{end}} - y_{\text{begin}}
i ← d
/* Scan line traversing */
for y ← y_{\text{begin}}, y ≤ y_{\text{end}}, y ← y + 1 do
  Mark (x, y) as edge point
  i ← i + n
  while i > d ∨ i ≤ 0 do
    x ← x + δx
    i ← i - δx × d
  end while
end for
```

### 1.2.1.2 Right and left edges

One scan line intersects both a left and a right edge of a polygon as it is seen in figure [1.3](#). Merging two edge point algorithms as the one just outlined into a single algorithm simply requires that all variables except $y$ (which will be the same for the whole line) be duplicated.
CHAPTER 1. ANALYSIS

So instead of keeping track of only \( x_n, n, d, \) and \( i \), we keep track of \( x_{n,\text{right}} \) and \( x_{n,\text{left}} \), \( n_{\text{right}} \) and \( n_{\text{left}} \), \( d_{\text{right}} \) and \( d_{\text{left}} \), and \( i_{\text{right}} \) and \( i_{\text{left}} \).

1.2.1.3 Changing edges

When scanning the polygon from bottom to the top we notice that the polygon usually has either two left edges or two right edges. This means that somewhere along the way we will need to change edges. For instance, in figure 1.3 when the scan line reaches \( y = 5 \) we need to make sure that the edge \( BC \) is followed at the right side the rest of the way.

One way to do this is to make the scan line traverse the polygon in two phases. In the first phase, we make the scan line run from the bottom point to the middle point (from \( A \) to \( B \) in figure 1.3). In the second phase we let it run from there to the top point (from \( B \) to \( C \) in the figure). Before we begin, we determine whether it is the right or the left edge which ends at the middle point. If the right edge ends there, we know that the variables associated with the right edge need to be changed between the phases. If the left edge ends there we change the variables associated with the left edge.

The variables we need to update are those we use to determine the next edge point from the current edge point. In figure 1.3 for instance, we start with \( x_{0,\text{right}} = x_{\text{begin}} = 3 \) and when the first phase is done, \( x_{5,\text{right}} = 14 \). When we continue with phase two, we do not need a new \( x_{5,\text{right}} \) – it is still 14. We only need to know how to get from \( x_{5,\text{right}} \) to \( x_{6,\text{right}} \) and so on following the new edge \( BC \). Thus the variables we need to update are \( n_{\text{right}}, d_{\text{right}} \) and \( i_{\text{right}} \) but not \( x_{n,\text{right}} \).

We determine the new updated values the same way as we initialized them in equation 1.1 with \( i = n \), except that now the begin and end coordinates equals the beginning and end point of the new edge.

1.2.1.4 Sorting points and edges

It is clear that we need to distinguish correctly between which edges are left edges and which are right edges. Also, we will need to determine which point is the bottom, middle and top point respectively. Beginning with the latter, we need to sort the points by \( y \)-value. We decide that if two points have the same \( y \)-value, we sort them by \( x \)-value. This will obviously only become relevant if a polygon has a top or bottom edge. In those cases, the middle point will have the same \( y \)-value as one of the two other points.

We name the points \( P_{\text{bottom}}, P_{\text{middle}} \) and \( P_{\text{top}} \). With these known, the task is to determine which edge is the left edge and which is the right – in other words to figure out if the angle from the vector from \( P_{\text{bottom}} \) to \( P_{\text{middle}} \) to the vector from \( P_{\text{bottom}} \) to \( P_{\text{top}} \) is positive or negative. This can be done by determining the sign of the determinant \( d = (x_{\text{middle}} - x_{\text{bottom}})(y_{\text{top}} - y_{\text{bottom}}) - (y_{\text{middle}} - y_{\text{bottom}})(x_{\text{top}} - x_{\text{bottom}}) \). If \( d < 0 \) then the angle is positive, and the vector from \( P_{\text{bottom}} \) to \( P_{\text{middle}} \) is left edge. If \( d > 0 \) then the angle is negative, and the vector from \( P_{\text{bottom}} \) to \( P_{\text{top}} \) is left edge. If \( d = 0 \) the two edges coincide.

With the edges sorted out, the last thing to resolve is if the polygon is left edge dominated or right edge dominated. This is simple: if the top point of the left edge is higher (e.g. if its \( y \) value is greater) than the top point of the right edge, then the polygon is right edge dominated, and we will need to change right edge variables between the phases in the scan line algorithm. Otherwise we can consider the polygon left edge dominated.
1.2.1.5 Summary

To sum up: drawing a polygon is done like this:

1. Sort points and edges
   - Sort points
   - Determine left and right edges
   - Determine if the polygon is right edge or left edge dominated

2. Phase 1: Scan line left and right edge points simultaneously up to the middle point.
   - At each $y$ step interpolate colors or points and normals depending on the shading model according to an interpolation difference function.
   - Draw pixels from the left edge point to the right edge point at the given $y$ position with a shading scheme. An interpolation differencing function will need to be calculated at each $y$ position.

3. When reaching the middle point, update the variables associated with the left or right side depending on left edge or right edge domination.

4. Phase 2: Scan line edge points from the middle point to the top point.

For the sake of example, figure 1.4 depicts how figure 1.3 has been colored when the scan line algorithm is done.

Figure 1.4: The result of the scan line algorithm.
1.3 Curves and surfaces

In most graphics software, some form of primitives are needed for modeling objects. In our case these primitives are curves and surfaces. With these we can model all necessary objects.

1.3.1 Curves

Drawing a curve is particularly easy if it goes straight from one point to another. If several such line segments are connected, they form a polyline. This can be modelled by a sequence of ordinary functions (e.g. \( y = ax + b \)) or by a sequence of pairs of points. There are problems with this approach, however. Firstly, vertical lines cannot be modelled by this type of function and secondly, the number of points to store can become large. An alternative way of expressing the line \( L \) from \( P_1 \) to \( P_2 \) is this:

\[
L(t) = P_1 + t(P_2 - P_1), \quad 0 \leq t \leq 1
\]

More explicitly for a line in three dimensions, this is:

\[
L(t) = \begin{cases} 
  x(t) &= x_1 + t(x_2 - x_1) \\
  y(t) &= y_1 + t(y_2 - y_1) \\
  z(t) &= z_1 + t(z_2 - z_1) 
\end{cases}, \quad 0 \leq t \leq 1
\]

This is called a parametric curve and it gets around the problem with vertical lines. However, we would still need one such parametric curve per line segment. If we want to draw, say, a sine curve, and impress more than just the most impressionable, we will need a lot of line segments, which could easily complicate things. It is better to use a more powerful curve, a parametric cubic curve. This type of curve is defined by 3 cubic polynomials. Cubics are attractive because they are the lowest degree polynomials that allow us to interpolate 2 end points, that each have a specified derivative. Further, they are also the lowest degree polynomials that are non-planar in 3D. In the following, unless otherwise stated, the parameter \( t \) is restricted to the interval \([0, 1]\). Formally, a parametric cubic curve, \( Q(t) \), has the form:

\[
Q(t) = \begin{cases} 
  x(t) &= a_xt^3 + b_xt^2 + c_xt + d \\
  y(t) &= a_yt^3 + b_yt^2 + c_yt + d \\
  z(t) &= a_zt^3 + b_zt^2 + c_zt + d 
\end{cases}
\]
We can split this into coefficients and parameters:

\[
Q(t) = \begin{pmatrix} a_x & b_x & c_x & d \\ a_y & b_y & c_y & d \\ a_z & b_z & c_z & d \end{pmatrix} \begin{pmatrix} t^3 \\ t^2 \\ t^1 \\ t^0 \end{pmatrix}
\]

If we call the coefficient matrix \( C \) and the parameter matrix \( T \), we get \( Q(t) = CT \). If we also split \( C \) into \( G \) and \( M \), we get \( Q(t) = GMT \). \( G \) is called the geometry matrix and \( M \) is called the basis matrix. This is a piecewise polynomial curve, which implies that we can join curves together. When doing that, it is useful to know the kind of continuity that is obtainable:

- Two curves have \( G^0 \) geometric continuity in a join point if the end point of the first is identical to the start point of the second.
- To have \( G^1 \) geometric continuity, the tangents in the join point must have the same direction.
- To have \( C^1 \) continuity, the tangents in the join point must have the same direction and the same magnitude.

These kinds of continuities are depicted in figure 1.6.

Below we describe 2 examples of parametric cubic curves, Hermite and Bezier curves.
1.3.1.1 Hermite curves

A Hermite curve is described by its start point, its end point and the tangents for the curve in each of these.

![Hermite curve diagram](image)

Figure 1.7: A Hermite curve.

Reusing the terminology from the general case, the Hermite curve is defined by this expression:

\[ Q_H(t) = G_H M_H T = G_H M_H (t^3 \ t^2 \ t^1 \ t^0)T. \]

We know what the geometry matrix, \( G_H \), contains, namely the geometric elements of the curve:

\[ G_H = (P_1 \ P_4 \ R_1 \ R_4) \]

Differentiating \( Q_H(t) \) with respect to \( t \), we get:

\[ Q'_H(t) = G_H M_H (3t^2 \ 2t \ 1 \ 0)^T \]

This gives us tangents in the start and end points, i.e. for \( t = 0 \) and \( t = 1 \):

\[ R_1 = Q'_H(0) = G_H M_H (0 \ 0 \ 1 \ 0)^T \]
\[ R_4 = Q'_H(1) = G_H M_H (3 \ 2 \ 1 \ 0)^T \]

We also know the start and end points themselves:

\[ P_1 = Q_H(0) = G_H M_H (0 \ 0 \ 0 \ 1)^T \]
\[ P_4 = Q_H(1) = G_H M_H (1 \ 1 \ 1 \ 1)^T \]

Now we have:

\[ (P_1 \ P_4 \ R_1 \ R_4) = G_H = G_H M_H \begin{pmatrix} 0 & 1 & 0 & 3 \\ 0 & 1 & 0 & 2 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 \end{pmatrix} \]
It is clear that $M_H$ must be the inverse of the last matrix. Therefore the final expression is:

$$Q_H(t) = G_H \begin{pmatrix} 2 & -3 & 0 & 1 \\ -2 & 3 & 0 & 0 \\ 1 & -2 & 1 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} t^3 \\ t^2 \\ t^1 \\ t^0 \end{pmatrix}$$

### 1.3.1.2 Bezier curves.

A Bezier curve is described by a start point, 2 control points and an end point. The vector formed by subtracting $P_1$ from $P_2$ is tangent to the curve in point $P_1$, and $P_4 - P_3$ is tangent to the curve in point $P_4$. The 4 points make up the Bezier geometry matrix:

$$G_B = \begin{pmatrix} P_1 & P_2 & P_3 & P_4 \end{pmatrix}$$

From this, it is easy to calculate the Hermite curve: Simply form the vector from $P_1$ to $P_2$ and the vector from $P_3$ to $P_4$:

$$G_H = \begin{pmatrix} P_1 \\ P_4 \\ R_1 \\ R_4 \end{pmatrix} = \begin{pmatrix} P_1 \\ P_4 \\ 3(P_2 - P_1) \\ 3(P_4 - P_3) \end{pmatrix}$$

Getting the Bezier basis matrix is also straightforward:

$$G_H = G_B M_{HB} = G_B \begin{pmatrix} 1 & 0 & -3 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & -3 \\ 0 & 1 & 0 & 3 \end{pmatrix}$$

$$Q_H(t) = G_H M_H T = G_B M_{HB} M_H T \Rightarrow M_B = M_{HB} M_H$$

And finally:

$$Q_B(t) = G_B M_B T$$
Apart from being very intuitive, a Bezier curve is also distinguished by the fact that it is always contained within the convex hull defined by the start, end and 2 control points.

### 1.3.1.3 Drawing curves

We know of 3 methods, to draw a parametric cubic curve: sampling, forward differences and subdivision. Sampling is simple but quite expensive, whereas forward differences is faster, but more complicated. Subdivision is elegant and simple, which is why we focus on it. It works by recursively halving the curve, and when it reaches a level where the curve is short enough, we simply draw a line from start to end point. For Bezier curves, we can decide when that level is reached by checking the flatness, i.e. the distance from the line \( P_1 P_4 \) to the highest point on the curve. A second method is to always let the recursion run a fixed number of times.

**Figure 1.9:** Checking the flatness by the distance \( d \).

Subdivision is visualised below:

**Figure 1.10:** Subdividing a Bezier curve. ©Knud Henriksen, DIKU.

The new points are easily obtained:

\[
\begin{align*}
L_2 &= (G_1 + G_2)/2, & H &= (G_2 + G_3)/2, & L_3 &= (L_2 + H)/2, \\
R_3 &= (G_3 + G_4)/2, & R_2 &= (H + R_3)/2, & L_4 &= R_1 = (L_3 + R_2)/2.
\end{align*}
\]

### 1.3.2 Surfaces

When modeling a surface, it is tempting to simply form a grid and represent it with a collection of points and edges. For anything but very simple figures, however, this will quickly grow
complicated. If, say, we were modeling a teapot and wanted a slightly taller and more slender form, there would be lots of individual points that needed to be changed. Fortunately, there are alternatives, and one of these is the Bezier surface. It is an extension of Bezier curves, in such a way that we form a grid with the curves. More precisely, we model our surface with a collection of 16 points, called a patch. Where a curve was parameterized on a single value, the surface is parameterized on two, \( s \) and \( t \). These correspond to the two dimensions of the grid seen in figure 1.11. Each row and each column is in fact a Bezier curve.

![Figure 1.11: A Bezier surface.](image)

Like curves, surfaces can be joined with different levels of continuity, occurring across edges. \( G^0 \) and \( C^0 \) continuity occurs across edges when the four common control points are (pairwise) identical. \( C^1 \) continuity occurs when control points are identical and the direction and magnitude of the tangents are also identical.

### 1.3.2.1 Drawing a surface

As for curves, we mention 3 ways of drawing a surface: Sampling, forward differences and subdivision. For the same reason as above, we also focus on subdivision here. It is done in two stages: A patch is divided in two subpatches, and these are each again divided in two, giving us a total of four new patches.

![Figure 1.12: Subdividing a Bezier surface.](image)

This continues until a required flatness has been reached, exactly as for curves. When doing this, we risk getting cracks between 2 patches if they were not both divided to the same
level. To avoid this, we could subdivide to a very small flatness level, or we could subdivide to a fixed level for each patch. In either case, we solve the problem with cracks at the cost of unnecessary subdivisions.

1.4 Projection

Projection is the art of taking a 3D scene and visualizing it in 2D, so it seems to be 3D. [FvDFH97, pp. 237] mentions many types of projections. In this section we will concentrate on perspective projection.

1.4.1 Homogeneous coordinates

In two dimensions a point can be described by two coordinates, \((x, y)\). Sometimes (as soon will become apparent), however, it’s convenient to add an extra coordinate, \(W\). We define that two points \((x, y, W)\) and \((\tilde{x}, \tilde{y}, \tilde{W})\) represents the same point in 2D if and only if \((x, y, W) = a \cdot (\tilde{x}, \tilde{y}, \tilde{W})\), for any real \(a\). To find the point in 2D, we just divide all coordinates by \(W\) and remove the third coordinate. The points in which \(W = 0\) are defined to be at infinity.

Similarly, in 3D, we can add an extra coordinate, yielding similar attributes, which will prove to be very convenient.

1.4.2 Rotation, Translation, Scaling and Shearing

Rotation, translation, scaling and shearing are operations that can be performed on any 2D image, and most image editors support at least some of these operations. An example of the effects of these operations on an image is shown in figure 1.13. However, we’re not going to use them on 2D systems. We are going to use them on 3D systems - or rather, 4D as we are going to use homogeneous coordinates.

When one of the four operations is called on an image (in 2D), each of them can be said to work on each point \(P\) in the image, and change it into \(P’\). In a simple translation the contents of the image are moved by some vector, so each component of the vector is simply added to the corresponding coordinate of each point of the image. Scaling is done by multiplying each coordinate of each point by a scale, which doesn’t have to be the same value on each coordinate. To do this, however, we will need to use a \(2 \times 2\) matrix. To rotate a point by an angle \(\theta\) around the origin, we would also have to multiply the point by a \(2 \times 2\) matrix. Shearing is also best done by multiplying the point with a matrix.

![Figure 1.13: The geometrical transformations. a) is the original scene, and the following figures show different transformations on this scene: b) translation, c) scaling, d) rotation, and e) shearing.](image-url)
So scaling, rotation, and shearing is done by multiplication with a $2 \times 2$ matrix, while translation is done by adding a vector. This is unfortunate, as we would like to represent all four transformations in a consistent and compatible way, for ease and convenience of use.

In order to achieve this, we switch to homogeneous coordinates (i.e. add an extra coordinate), so each point, previously described by $(x, y)$, must now be described by $(x, y, W)$. This will allow us to represent the translation operation by a $3 \times 3$ matrix that can be multiplied with the point.

These matrices will be shown as they are needed, as seeing them will not bring any increased understanding\(^1\).

1.4.3 The view system

In order to make any sort of projection, we must start by defining variables and assumptions we will use. First, we have a set of points, lines, and polygons in three dimensions. This set is the scene we want to view. Second, we have the position from which we want to visualize the scene - that is, the eye of the beholder. And third, we must define the view plane, that is the plane that we project onto. This plane will later be drawn to the screen.

The actual construction of the scene will not be made in this section, but it is assumed that it is constructed from points that we will project onto the view plane.

In order to be able to change the view plane we wish to describe the view plane in its own unique coordinate system, that we call *view coordinates*. Because of this we will have to define a few extra variables and vectors, but it will be well worth the effort! We will call the coordinate system of the scene *world coordinates*.

- **VRP** - the view reference point - this point is the origin of the view coordinate system in world coordinates.
- **VPN** - the view plane normal - this vector is the normal of the view plane - if we normalize it, we will have one of the axes of the view coordinate system. This vector is given in the world coordinate system.
- **VUP** - the view up vector - this vector defines the “up” direction of the view coordinate system, by projecting this vector onto the view plane, we can find a second axis (actually done by vector cross product $\mathbf{VUP} \times \mathbf{VPN}$). From this vector, we can calculate the third vector to define the view coordinate system. This vector is given in the world coordinate system.
- **CW** - center of window, a point - usually one cannot see all of the world, one could say that we look at the world through a window. We assume that the window is rectangular, and thus, only need two points to define it. The center of window is the center of this window. This point is in view coordinates.
- **PRP** - the projection reference point - also known as the eye of the beholder, given in view coordinates.
- **DOP** - the direction of projection - this vector will be used to find out whether or not a given point is on the same side of the view plane as the **PRP**.

In the “nice” situation, the **CW** and **VRP** is the same point, **VPN** parallel with the $z$ axis, **VUP** parallel with the $y$ axis, and **PRP** somewhere on the $z$ axis. In general, however, we can’t assume that. This can be realized by an analogy. Suppose you are in a room with a window. When you look out the window, you can stand up, sit on a chair or lie in bed. So the **PRP** is independent of the center of window. Maybe the window can be tilted and

\(^1\)The interested reader could check out [FvDFH97, pg. 215 and 216].
moved relative to you and the world coordinate system, changing both the VRP and VPN. Neither does the window have to be square - any rectangular shape will do.

Figure 1.14 shows the coordinate systems, points and vectors in a more general case.

While the preceding presentation has tried to give an intuitive understanding of these definitions, we will need more than that. We will need to be able to calculate a number of vectors. The basis vectors of the view plane can be found by:

\[ \hat{n} = \frac{\mathbf{VPN}}{|\mathbf{VPN}|} \]
\[ \hat{u} = \frac{\mathbf{VUP} \times \mathbf{VPN}}{|\mathbf{VUP} \times \mathbf{VPN}|} \]
\[ \hat{v} = \frac{\mathbf{VPN} \times (\mathbf{VUP} \times \mathbf{VPN})}{|\mathbf{VPN} \times (\mathbf{VUP} \times \mathbf{VPN})|} \]

The center of window is

\[ \mathbf{CW} = \left( \frac{u_{\text{min}} + u_{\text{max}}}{2}, \frac{v_{\text{min}} + v_{\text{max}}}{2} \right) \]

And the direction of projection is:

\[ \mathbf{DOP} = \mathbf{PRP} - \mathbf{CW} \]

We note that we will need to know the VRP in world coordinates, the vectors VPN and VUP to setup the view coordinate system. We will need the PRP in view coordinates, and two points to specify the boundaries of the window.

1.4.4 Construction of the normalization matrix

The first step of the projection is to normalize the scene. This is done in order to make a convenient representation of the scene in the view plane. The central goals are to transform our scene so that the PRP is in the origin, the view plane is orthogonal on the z axis, and...
the CW is placed on the z-axis. If we were to implement clipping, the last step would be extremely useful, as clipping is easier to do against simple planes.

This section is based on "The projection cookbook" by Knud Henriksen, and follows these basic steps:

1. Translate VRP to the origin
2. Rotate the view coordinate system so it is axis parallel to the the world coordinate system.
3. Translate PRP to the origin
4. Shear so that the line from PRP to CW becomes equal to the z axis
5. Scale to the canonical perspective view volume

In the following each of these five steps will be described, and finally we will put it all together.

1.4.4.1 Translate VRP to the origin

In order to rotate the view coordinate system to have axes parallel to the world coordinate system, we would have to place VRP in the origin of the world coordinate system. This can be achieved by a simple translation. If VRP = (vrpx, vrpz, vrpz) the translation matrix TVRP is:

\[
TVRP = \begin{pmatrix}
1 & 0 & 0 & -vrpx \\
0 & 1 & 0 & -vrpy \\
0 & 0 & 1 & -vprpz \\
0 & 0 & 0 & 1
\end{pmatrix}
\]

1.4.4.2 Rotate view coordinate system

After translation of VRP into the origin of the world coordinate system, it’s now possible to rotate the view system, so that it has axis parallel to the world coordinate system. This could be done by projecting one view axis onto a plane spanned by two world axes, and calculating the angle between the projection of the view axis and one of the two world axes that span this plane, and rotate the view axis by this angle around the third world axis. This should then be repeated for the other two view axis. This is, however, a very involved process (not to mention three matrix multiplications), and we would very much like to find an easier path.

Fortunately this process can be simplified considerably, by noting that the resulting rotational matrix, R, is an orthogonal matrix. This allows us, by knowing the basis of the system we wish to rotate into (the simple world coordinate system), to insert the basis vectors of the first system directly into the rows of the rotation matrix. If the basis vectors of the view coordinate system are (\(\hat{u}, \hat{v}, \hat{n}\)), rotated into the basis vectors of the world coordinate system, (\(\hat{x}, \hat{y}, \hat{z}\), the rotation matrix is:

\[
R = \begin{pmatrix}
u_1 & u_2 & u_3 & 0 \\
v_1 & v_2 & v_3 & 0 \\
n_1 & n_2 & n_3 & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\]
CHAPTER 1. ANALYSIS

Note the basic assumption that we rotate into a “traditional” \((\hat{x}, \hat{y}, \hat{z})\) basis.

1.4.4.3 Translate PRP to the origin

After the rotation of the world coordinate system, we wish to have PRP in the origin of our coordinate system. Why this is clever, will be apparent later. As PRP is given in view coordinates, it’s given relative to VRP (at present our origin), so a simple translation will do:

\[
T_{PRP} = \begin{pmatrix}
1 & 0 & 0 & -prp_x \\
0 & 1 & 0 & -prp_y \\
0 & 0 & 1 & -prp_z \\
0 & 0 & 0 & 1
\end{pmatrix}
\]

1.4.4.4 Shearing

In this step we wish to shear so that the center line of the view volume becomes the z axis. This is where CW (center of window) comes into play, as this point and PRP lie on the center line. From these two points, we can calculate the direction of the center line (by calculating the DOP - direction of projection). We wish to shear so that DOP is parallel to the z axis. Since we wish to shear in the x and y direction, but not in the z direction, we must use the \(xy\)-shear matrix. The shearing matrix can be calculated by solving this equation for the matrix \(Sh_{xy}\):

\[
\begin{pmatrix}
0 \\
0 \\
dop_n
\end{pmatrix}
= Sh_{xy} \cdot \begin{pmatrix}
dop_u \\
dop_v \\
dop_n
\end{pmatrix}
= \begin{pmatrix}
dop_u + sh_x dop_n \\
-dop_u - sh_x dop_n \\
0
\end{pmatrix}
\]

By simple algebraic manipulations we get that the shearing matrix \(Sh_{xy}\) must be:

\[
Sh_{xy} = \begin{pmatrix}
1 & 0 & -dop_x & 0 \\
0 & 1 & -dop_y & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\]

1.4.4.5 Scale to the canonical perspective view volume

In this step we wish to scale from the perspective view volume into the canonical perspective view volume. The scaling matrix is calculated in two steps. First we make sure that the slope of the planes bounding the view volume is \(\pm 45^\circ\). Then we scale all the three axis uniformly so that the back clipping plane is scaled from \(z = vrp_z' + B\) to \(z = -1\), where \(vrp_z'\) is the z coordinate of VRP after translation (step 3) and shearing (step 4), and \(B\) is the distance of the back clipping plane from PRP. As the shearing operation in step 4 doesn’t affect the z coordinates, \(vrp_z' = -prp_z\).

At the z coordinate of CW, \(-prp_n\), the y coordinate must be \(y = \frac{v_{max} - v_{min}}{2}\). We wish this to be equal to \(-prp_n\). This gives:
\[-prp_n = s_y \frac{v_{\text{max}} - u_{\text{min}}}{2} \Rightarrow s_y = \frac{-2prp_n}{v_{\text{max}} - u_{\text{min}}}\]

Similarly, the scaling factor for \(x\) is:

\[s_x = \frac{-2prp_n}{u_{\text{max}} - v_{\text{min}}}\]

The second factor in the scaling is the uniform scaling of all axis. Here we wish that the \(z\) value on the back clipping plane is equal to -1. \(B\) is the signed distance from VRP to the back clipping plane. So in order for the elements between the back clipping plane and the view plane to be “visible”, \(B\) must be negative (the same goes for the distance to the front clipping plane, \(F\)). This yields:

\[s_{\text{all}}(B - prp_n) = -1 \Rightarrow s_{\text{all}} = -\frac{1}{B - prp_n}\]

This gives the scaling matrix \(S\):

\[
S = \begin{pmatrix}
    s_{\text{all}}s_x & 0 & 0 & 0 \\
    0 & s_{\text{all}}s_y & 0 & 0 \\
    0 & 0 & s_{\text{all}} & 0 \\
    0 & 0 & 0 & 1
\end{pmatrix}
\]

\[
= \begin{pmatrix}
    \frac{-2prp_n}{(u_{\text{max}} - v_{\text{min}})(B - prp_n)} & 0 & 0 & 0 \\
    0 & 0 & \frac{-2prp_n}{(v_{\text{max}} - u_{\text{min}})(B - prp_n)} & 0 \\
    0 & 0 & 0 & \frac{-1}{prp_n + B} \\
    0 & 0 & 0 & 1
\end{pmatrix}
\]

Putting it all together

Now we know all the matrices needed to transform the objects in our scene in world coordinates into the canonical perspective view volume. The transformation matrix \(N\) is:

\[
N = S \cdot Sh_{xy} \cdot T_{PRP} \cdot R \cdot T_{VRP}
\]

So if we have a point \(P\) in world coordinates, its coordinates in the canonical perspective view volume \(P'\) can be found by:

\[
P' = N \cdot P
\]

1.4.5 Construction of the projection matrix

In this section, we will show the basic math involved in calculating a matrix that can take a point \(P\) in the 4D homogeneous coordinates and project it onto the view plane and give us back the \((x_p, y_p)\) coordinates of the projected point \(P_p\). We will assume that we have started by calculating and applying the \(N\) matrix (as specified in the previous section), as this will place us in the “nice” situation. This makes the math significantly simpler.

As mentioned above, we assume that PRP is in the origin, that the view plane is orthogonal to the \(z\) axis in a distance \(d\) from PRP. Furthermore, we assume that CW is on the \(z\) axis, and that there is no clipping left to be done. The basic situation is shown in figure 1.15, but it’s more convenient to split it into two figures, as shown in figure 1.16. Given the point \(P\) and the value of \(d\), and wish to calculate the coordinates \(x_p\) and \(y_p\) of the point \(P_p\). This can be done by noting that \(P\) and \(P_p\) act as a delimiter of two similar triangles. These two
triangles must have the same angles. The angle $\theta$ at $\text{PRP}$ in the $XZ$ plane must be given by:

$$\tan \theta = \frac{X}{Z} = \frac{x_p}{d} \Rightarrow x_p = \frac{Xd}{Z} = \frac{X}{Z/d}$$

And the angle $\phi$ at $\text{PRP}$ in the $YZ$ plane must be given by:

$$\tan \phi = \frac{Y}{Z} = \frac{y_p}{d} \Rightarrow y_p = \frac{Yd}{Z} = \frac{Y}{Z/d}$$

So what we have is a scale factor $Z/d$ that is applied to the $x$ and $y$ coordinates. As we are working in homogeneous coordinates the scaling factor can be applied simply by equating the $W$ coordinate to $1/d$, assuming $W = 1$, as this will correspond to dividing all coordinates by $z/d$.

This yields the projection matrix $P$: 

$$P = (X, Y, Z)^T$$
\[ P = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{d} & 0 \end{pmatrix} \]

Now all we need is to calculate \( d \). We know that \( CW \) and \( VRP \) has the same \( z \) coordinate as \( d \). So we must be able to find \( d \) by applying \( N \) to \( VRP \) (as \( CW \) is in view coordinates, applying the normalization wouldn’t give the desired result).

### 1.4.6 Transformation of view volumes

An alternative to the geometric projection of the previous section is to transform the canonical perspective view volume into the canonical parallel view volume. The two view volumes are shown in figure 1.17. In this view volume the values stored in the \( x \) and \( y \) coordinates are the same as we would, get if we made a geometric projection. There is, however, a crucial difference: We will still have access to the relative \( z \) coordinate. In order to find the transformation matrix \( C \), we note that the two points furthest away from \( PRP \) must stay unchanged, and a point in the corner of the front clipping plane \((\pm z_{\text{max}}, \pm z_{\text{max}}, z_{\text{max}})\) must be transformed into the corresponding corner of the canonical parallel view volume, i.e. \((\pm 1, \pm 1, 0)\). This transformation can be accomplished by the conversion matrix \( C \):

\[ C = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{1+z_{\text{max}}} & -z_{\text{max}} \\ 0 & 0 & -1 & \frac{-z_{\text{max}}}{1+z_{\text{max}}} \end{pmatrix} \]
Note that $z_{\text{max}}$ is given by:

$$z_{\text{max}} = -\frac{F - \text{prp}_n}{B - \text{prp}_n},$$

where $B$ and $F$ is the signed distance from VRP front and back clipping plane, respectively, as defined in section 1.4.4.5.

### 1.4.7 Mapping into device coordinates

After normalization and transformation, the $x$ and $y$ coordinates of our points have values between -1 and 1. However, we wish to show the scene on some device that uses pixels to show the scene, and usually it’ll also have a different coordinate system, sometimes only supporting positive integer coordinates. Thus, we need to translate the $x$ and $y$ coordinates into the new coordinate system, and scale into the device window (and round of to the nearest integer value).

If the coordinate system of the device is a regular cartesian $(x, y)$ coordinate system, such as the one used by DotDevice, the translation can be done by moving the origin of the canonical view volume from $(0, 0, 0)$ to $(1, 1, 1)$, and thus, all coordinates would have positive values. Then the translation is:

$$T_v = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

If the device has dimensions $x_{\text{max}} \times y_{\text{max}}$ pixels, then we must use these as scale factors, and get the scale matrix:

$$S_v = \begin{pmatrix} x_{\text{max}} & 0 & 0 & 0 \\ 0 & y_{\text{max}} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Note, that there is no scaling of the $z$ coordinate, as we’ll only need to know the relative values of the $z$ coordinates in the $z$ buffer algorithm. So the mapping can be done by the mapping matrix $M$:

$$M = S_v \cdot T_v$$

In the more complicated case, in which we paint into a window defined by $x_{\text{min}} \leq x \leq x_{\text{max}}$ and $y_{\text{min}} \leq y \leq y_{\text{max}}$, the scaling factors will be the average of the min and max coordinates, and we must end by translating the scene into the lower left corner of the window.

### 1.4.8 The process of projection

The full process of projection is:

1. Normalization
2. Projection
CHAPTER 1. ANALYSIS

3. Mapping into device coordinates

If we are projecting the point $P$ into the point $P'$ on the view device, we would do the calculation:

$$P' = M \cdot C \cdot N \cdot P$$

This would give us $P'$ in homogeneous coordinates. So in order to get the "real" points, we would have to divide by $W$, discard $z$ and $W$, and round (in order to get integer values).

1.5 Illumination

The color of an object is seldom just one color. Depending on lighting and the position of the viewer, the perceived color of an object can not only change tremendously, there can even be color-changes over the surface of the object, i.e. highlights and shadows. The focus of this section is to build a model that can color an object so it’ll look good.

The best result may come from building a physical model, but this is a very complex problem. One would have to calculate light intensity from the light source on any surface, the light reflected from each surface (depending on the material properties of this surface), the reflected light intensity on the surfaces "visible" from the reflecting surfaces, the reflection of the reflected light and so on. But not only that: The light source may not be a perfect mathematical point (that is, have zero volume), but will often have some extent, which a physical model will have to take into account. And there may very well be more than one light source, with intensities and colors changing over their extent.

One should also note that if an object is visible, it’s because light is reflected from the surface of the object towards the eye of the viewer.

This complexity is the central reason why researchers have tried to find simpler models that work well in practice. These are, however, built more on trial and error ("what looks good?") than on a physical understanding.

1.5.1 Ambient light

Ambient light adds a constant amount of light everywhere (see the left triangle in figure 1.18). That means that each object is displayed as having a single color, that is intrinsic to it, regardless of light sources. If one assumes a constant and all-present light source with intensity $I_a$, then one could define a material constant $k_a$, called the ambient reflection coefficient, that controls the fraction of light reflected from the surface (and thus, is visible to the viewer). So the intensity $I$ of the light on a surface could be described by:

$$I = k_a I_a$$

One could view ambient light as a way of modeling the reflected light in the physical model above. In this context, the ambient intensity should be dependent on the number and intensity of the light sources. In a brightly lit scene (a summers’ day), the intensity of the ambient light should be quite high, but in a dark cave, lit only by a flashlight, it should be very low (or nil).
1.5.2 The source of light

In order to make good looking scenes, we must take the sources of light into account. In the following we will look at the case of just one source of light, and assume this to be shaped like a point.

1.5.2.1 Diffuse reflection

A point on matte surfaces appear equally bright regardless of the position of the viewer (see center triangle in figure 1.18). This can be modeled by assuming that the intensity is only dependent on the angle \( \theta \) between the normal \( \mathbf{N} \) of the matte surface and the vector \( \mathbf{L} \) from the point in question to the light source. If \( \mathbf{N} \) and \( \mathbf{L} \) have been normalized, then \( \cos \theta = \mathbf{N} \cdot \mathbf{L} \), and so the equation of diffuse illumination must be:

\[
I = I_p k_d (\mathbf{N} \cdot \mathbf{L}),
\]

where \( I_p \) is the intensity of the light source, and \( k_d \) is the diffuse reflection coefficient and is a constant of the material. Note that if \( \mathbf{N} \cdot \mathbf{L} \) is negative, the surface faces away from the source of light, and thus, the light doesn’t illuminate the surface in question. From this it follows that the light from the light source shouldn’t be added to the light reflected from the point in question, and thus, \( \mathbf{N} \cdot \mathbf{L} \) is defined as being zero. This could be expressed as:

\[
\mathbf{N} \cdot \mathbf{L} = \max(\mathbf{N} \cdot \mathbf{L}, 0)
\]

In the remainder of this section, we will assume that any vector dot product is either positive or zero, even if it’s not explicitly written.

There is, however, a central problem in this model. If we assume that the light doesn’t meet any obstacles, the amount of light at a distance \( d_L \) from the light source is a constant, as the light doesn’t disappear. So the amount of light passing through the surface of a sphere
with radius $d_L$ at a given point in time is a constant. From this it follows that the intensity$^2$ $I_p$ of the light at a point on the sphere must be

$$I_p = \frac{I}{4\pi d_L^2}.$$ 

Using this to model the real intensity of the light at a given point has a few drawbacks. First, at small $d_L$ (say $d_L = 10^{-3}$) the fraction blows up. Close to the light source there is great variation, even if the angle between $N$ and $L$ is unchanged. Second, far away from the light source there is too little variation.

Therefore, the attenuation due to distance from the light source can be given greater range of effects by defining the attenuation factor $f_{att}$ as:

$$f_{att} = \min\left(\frac{1}{c_1 + c_2 d_L + c_3 d_L^2}, 1\right),$$

where $c_1$, $c_2$ and $c_3$ are just user defined constants. Thus, the equation for the diffuse reflection is:

$$I = f_{att} I_p k_d (N \cdot L)$$

### 1.5.2.2 Color of light and surfaces

Light and surfaces are not generally monochromatic. In the real world light consists of many different wavelengths. We will assume that light consists of just three “wavelengths” - red, green and blue.

This leads to giving light sources an intensity for each color, so each light source is represented by a triple $(I_r, I_g, I_b)$. Similarly the material can be said to have a diffuse color, given by the triple $(O_{dr}, O_{dg}, O_{db})$. For a color, $\lambda$, the intensity is:

$$I_\lambda = k_{a\lambda} O_{d\lambda} I_{a\lambda} + f_{att} k_d O_{d\lambda} (N \cdot L) I_{p\lambda}$$

### 1.5.3 Phong illumination

On a shiny surface we get highlights that move with the viewer, as seen on figure 1.18. This is called specular reflection. Phong illumination is a way to model specular reflection. On a reflecting surface the direction of the “strongest” highlight will be given by the direction of the reflected light $R$. Thus the maximum highlight happens when the angle $\alpha$ between the direction $V$ to the viewpoint and $R$ is zero, and falls off when $\alpha$ increases. Phong Bui-Tuong approximated this falloff to $\cos^n \alpha$, where $n$ is called the specular reflection exponent. The higher the value of $n$, the more sharp is the falloff. If both $R$ and $V$ is normalized, $\cos \alpha = R \cdot V$. As earlier, this dot product must be positive or zero, as the total amount of light can’t be reduced and. Furthermore, we must also require that $N \cdot L \geq 0$.

Even though the angle of incidence $\theta$ does have an influence on the amount of specular reflected light, it’s typically set to the product of $k_s$, the specular reflection constant, and $O_{s\lambda}$, the objects specular color. This gives the total equation for illumination:

$$I_\lambda = k_{a\lambda} O_{d\lambda} I_{a\lambda} + f_{att} I_{p\lambda} [k_d O_{d\lambda} (N \cdot L) + k_s O_{s\lambda} (R \cdot V)^n]$$

$^2$Remember that the intensity is the amount of light pr area.
CHAPTER 1. ANALYSIS

Note, that we haven’t given an explicit expression for $\mathbf{R}$. This vector can be calculated by mirroring $\mathbf{L}$ around $\mathbf{N}$. This gives the following equation for $\mathbf{R}$:

$$\mathbf{R} = 2\mathbf{N}(\mathbf{N} \cdot \mathbf{L}) - \mathbf{L}$$

Now, we can calculate the color of each point on every object in the world! However, this calculation is quite involved, with many floating point operations. So we would like to do the calculation as few times as possible, and this is the topic of the next section.

1.6 Shading and Coloring

Using an illumination model such as the one described in section 1.5 we can determine the final color of every point on the screen. This process is called shading. The brute force method would be to determine every point and normal of every surface on the screen and then apply the desired illumination model to these data. This is the correct and precise way to do it, but it is also very cumbersome and slow. As usual when discussing computer graphics we will need to make some sensible approximations that will reduce the work needed to be done by the computer.

1.6.1 Flat shading

The easiest way to approximate shading would be to divide surfaces in the model world into polygons and simply shade each polygon according to its normal. This takes little computational effort but it looks it. Particularly, edges between polygons are visible as a sudden change of color and the whole scene looks edged. To smooth things out the number of polygons can be increased, but this increases the number of calculations considerably and it does not really pay off in the results. Ideally we would like to obtain a nicer and smoother picture with not too many polygons.

1.6.2 Interpolated shading

Instead of coloring the whole polygon with a single color, we can make the color change gradually across polygons. To do this we interpolate data from the polygon’s vertices across the inside of the polygon. Exactly what data from the vertices we interpolate determine the shading model. As long as we use linear interpolation we can make use of an iterative interpolation method which lessens the calculation burden substantially. Doing this we can get away with drawing fewer polygons and still achieve good pictures.

1.6.2.1 Interpolation

Generally speaking, interpolation is the process of constructing new data points from a discrete set of points. In computer graphics this is typically used to color something where only a very limited number of pairs of colors and points are known. For example, we might want to color a line from point $A$ to $B$, where point $A$ is red and point $B$ is blue. We want the line to gradually change its color from red to blue when we traverse the line from $A$ to $B$. We use interpolation to deduce the intermediate colors of the points between $A$ and $B$. Note that the intermediate colors are dependent on the color model used.
Intuitively the colors of the two end points are considered of equal weight, thus leading to a scale of colors where the color is represented linearly dependent on the distance to the end points. This is called linear interpolation and it will be the main focus in this project.

Colors are not the only thing that can be interpolated. We will later need to interpolate vectors along a line for example. Generally an interpolation function takes a point, in which we would like to calculate the corresponding subject, and returns the subject (e.g. colors or vectors) we are seeking. The two points and two subjects to interpolate between are constant for a given interpolation function. Thus it can be described as a function of the form

\[ f_{\mathcal{S} \times \mathcal{P}^2} : \mathcal{P} \mapsto \mathcal{S}, \]

where \( \mathcal{S} \) is the set of subjects and \( \mathcal{P} \) is the set of points.

When interpolating colors from \( x_{\text{begin}} \) to \( x_{\text{end}} \) where \( x_{\text{begin}} \) has color \( c_{\text{begin}} \) and \( x_{\text{end}} \) has color \( c_{\text{end}} \) using linear interpolation, the function looks as follows:

\[ f_{c_{\text{end}}, c_{\text{begin}}, x_{\text{end}}, x_{\text{begin}}}(x) = c_{\text{end}} - (c_{\text{end}} - c_{\text{begin}}) \frac{x_{\text{end}} - x}{x_{\text{end}} - x_{\text{begin}}}. \]

From this point forward we will drop the subscripts to \( f \) for brevity.

We can derive an iteration method so that when we know \( f(x) \) we can calculate \( f(x + \Delta x) \) provided we know the difference \( \Delta f(x) \) between the two. The huge advantage with the linear interpolation function over other interpolation functions is that the difference function is constant. We need to calculate it only once for given end points and colors. Since this property is exactly what constitutes “linear functions” it is really no surprise, but let us calculate it anyway and see for ourselves:

\[
\Delta f(x) = f(x + \Delta x) - f(x) \\
= c_{\text{end}} - (c_{\text{end}} - c_{\text{begin}}) \frac{x_{\text{end}} - x - \Delta x}{x_{\text{end}} - x_{\text{begin}}} - c_{\text{end}} + (c_{\text{end}} - c_{\text{begin}}) \frac{x_{\text{end}} - x}{x_{\text{end}} - x_{\text{begin}}} \\
= (c_{\text{end}} - c_{\text{begin}}) \left( \frac{x_{\text{end}} - x}{x_{\text{end}} - x_{\text{begin}}} - \frac{x_{\text{end}} - x - \Delta x}{x_{\text{end}} - x_{\text{begin}}} \right) \\
= \Delta x \frac{c_{\text{end}} - c_{\text{begin}}}{x_{\text{end}} - x_{\text{begin}}}. 
\]

We notice that \( \Delta f(x) \) depends only on end points and on the incrementing parameter \( \Delta x \), but not on \( x \).

For a computer the iterative method going from one point to the next is much faster than repeatedly calculating the interpolation function for each point on the path. The scheme is of course similar when interpolating vectors or other subjects different from colors.

### 1.6.2.2 Gouraud shading

In Gouraud shading we apply an illumination model to the three vertices of each polygon, and then interpolate the obtained colors to determine the color of the inside of the polygon. We thus avoid applying the illumination model to each and every point of the inside of the polygon.

This presumes that each vertex has a normal and that each of these normals is determined from the original surface which the collection of polygons model. If the original surface is not known, each normal can be determined as an average of the surrounding polygon normals.
Since colors are determined solely from the polygon’s end point colors, any spot in the middle of the polygon will not be visible. A highlight will need to be near one of the polygon end points to be visible.

If we use Gouraud shading, we will need to apply the illumination model to each of the original polygon’s vertices and map the calculated colors to the screen polygon’s vertices. Using these colors and the difference in y-coordinates we calculate an interpolation differencing function which determines the change in color per y-step for each edge. This is not much different from what we did in section 1.6.2.1 only this time we use the y-coordinate instead of the x-coordinate and we know that our ∆y is 1. The interpolation function becomes

$$f(y) = c_{\text{end}} + (c_{\text{end}} - c_{\text{begin}}) \frac{y_{\text{end}} - y}{y_{\text{end}} - y_{\text{begin}}}$$

and with ∆y = 1 the differencing color (i.e. the constant differing function) becomes:

$$\Delta c ( = \Delta f(y)) = \frac{c_{\text{end}} - c_{\text{begin}}}{y_{\text{end}} - y_{\text{begin}}}$$

Thus beginning with the color $c_{\text{begin}}$ of the start point $(x_0, y_0) = (x_{\text{begin}}, y_{\text{begin}})$, the color at $(x_1, y_1)$ becomes $c_1 = c_0 + \Delta c = c_{\text{begin}} + \Delta c$. Generally we have $c_n = c_{n-1} + \Delta c$.

Now we know the colors of the edge points of the right and the left edges, we can interpolate between right and left color for a given scan line when drawing the horizontal line. The same scheme as before applies: at height $y = y_n$ we have color $c_{n, \text{left}}$ at the left point $(x_{n, \text{left}}, y_n)$ and color $c_{n, \text{right}}$ at the right point $(x_{n, \text{right}}, y_n)$. So to find the color at some point $(x, y_n)$ in between we interpolate. This interpolation function looks as follows:

$$f(x) = c_{n, \text{right}} + (c_{n, \text{right}} - c_{n, \text{left}}) \frac{x_{n, \text{right}} - x}{x_{n, \text{right}} - x_{n, \text{left}}}$$

with a difference color of

$$\Delta c = \frac{c_{n, \text{right}} - c_{n, \text{left}}}{x_{n, \text{right}} - x_{n, \text{left}}}$$

All this can be integrated in the scan line algorithm from section 1.2.1 such that we calculate the difference color per y-step when we initialize the other variables before the first phase. During the iterations we make sure to update the color as well by adding the difference color. When changing edges we update the difference color also.

### 1.6.2.3 Phong shading

Another method is to interpolate the normals of the vertices across the inside of the polygon, and then apply the illumination model to each point and normal. This is known as Phong shading. This borders the brute force method except that we determine normals and points using interpolation, which can be boosted using an interpolation differencing function as sketched above.

Phong shading requires that we interpolate the end points and their normals of the original world polygon across the screen polygon, and then let the illumination model determine the color of each point on the screen polygon from the interpolated world point and normal.
1.6.2.4 Problems with interpolated shading

When interpolating linearly we do not take into consideration the perspective distortion caused by the projection from the three dimensional world coordinate system onto the two dimensional screen coordinate system. For instance, the world middle point of a rectangle which is seen from an angle is not the middle point of the figure seen on the screen. This is because a constant change in screen coordinates does not map to constant changes in world coordinates.

Another problem is that the orientation of the polygon in question matters. For instance, when rotating the viewing plane, a point or color calculated with interpolation might differ before and after the rotation. This is a consequential error of perspective distortion.

Another obvious issue is that since we are keen on not increasing the number of polygons too much, the silhouette of a surface will still look edged even though the inside looks smooth.

1.7 Visible surface determination

We want to determine the visible surfaces of objects in a scene, in order to avoid drawing those that are not visible. This process is also called hidden surface elimination. In principle it is not complicated, but it can be time consuming. [FvDFH97] talks about image- and object-precision algorithms. Image-precision means that we handle the task from a pixel level, i.e. for each pixel in the window we determine which object is the closest. Object-precision means that we handle it from an object level, i.e. for each object in the scene, we determine which parts of it are unobscured by other objects, and then draw those.

1.7.1 The Z-buffer algorithm.

This image-precision algorithm has at its center a two dimensional array with the same dimensions as the window we display our scene in. Each element in the array contains a z-value and a color. Initially the buffer contains the z-value of the back clipping plane and an arbitrary color. When we scan-convert the polygons, we calculate a z-value, \( z_{xy} \), for each pixel (by interpolation). This is compared to the z-value in the same \((x,y)\)-position in the buffer. If the buffer’s z-value is closer to the viewer than \( z_{xy} \), we skip to the next pixel. If \( z_{xy} \) is closer, then we overwrite this buffer element with \( z_{xy} \) and its color. When we have scan converted the entire scene, the contents of the buffer can be drawn on the screen. One attractive thing about this is, that we can draw in any order we like. We can be sure that the pixels with the z-value closest to the viewer is the one whose color will be drawn last.

1.7.2 List-priority algorithms.

This is a different type of algorithm. The central aspect is to create a list of the objects to draw, sorted on z-value. This way, objects can be drawn such that the one closest to the viewer is drawn last, thereby ensuring that it overwrites what it hides. In certain cases, there is no order that will solve the problem, in which case it is necessary to split polygons into smaller pieces, sort the new polygons and then draw.

An example of a list-priority algorithm is the binary space partitioning algorithm, BSP. The idea is as follows: By splitting the plane into two half planes, front of \( \textbf{P}_1 \) and back of \( \textbf{P}_1 \), we know that if viewed from the front of \( \textbf{P}_1 \), no object from the back of \( \textbf{P}_1 \) can obscure
an object from the front side. The splitting of planes continues recursively, so the back of $P_1$ is split into front and back of $P_2$. This results in a binary tree. The internal nodes of the binary tree are splitting lines; the left child of a node $N$ contains what is on the front side of this line, and the right child contains what is on the back side of this line. The leaves contains the regions that result from the partitioning. BSP works at object-precision, and is good for scenes with static objects and changing viewpoint.

1.7.3 Area subdivision.
This type of algorithm works by examining an area of the projected image. If the visible polygons in this area are easily found, then they are displayed, otherwise the area is subdivided until the decision becomes easy.

1.7.4 Ray tracing.
Ray tracing in its simplest form consists of tracing a ray from the viewer’s eye into the scene. The color of the object that is hit first is the color we will draw at this pixel.
2 Design

2.1 Doing the projection

In this section different ways of implementing a way to do the projection will be discussed.

In section 1.4 it became clear that the calculation of the projection matrices are only
dependent on the definitions of the view plane and projection reference point. So it seems
reasonable to encapsulate the calculations in a class that can either access these definitions or
contain these definitions. So the calculation of the projection matrices is dependent on how
we chose to represent this data.

One way could be to setup the view plane in one class, and the projection reference point
in another. This would, however, make it complicated to make changes to the view plane
and PRP. Thus, it seems convenient to setup the whole view coordinate system in one class.
This class could be called Camera, and could also contain methods to calculate the projection
matrices.

2.1.1 The Camera class

From section 1.4 we note that we need VRP in world coordinates, PRP in view coordinates,
the vectors VPN and VUP, and two points to specify the boundaries of the window in order
to setup the system (and later to calculate the projection matrices). These values should be
specified and set in the constructor. In order to increase ease of use, one could also define
some sensible default values.

From these we would like to calculate the unit vectors of the view coordinate system,
(\(\hat{u}, \hat{v}, \hat{n}\)), the center of window, CW, and the direction of projection, DOP. Should these be
made into specific fields? The class would need more memory, but the fields would only have
to be recalculated when the values from the constructor is reset. On the other hand they are
only needed to calculate the projection matrices, which is only done once anyway.

The Camera class shouldn't know anything about the device that will show the scene.
However, we will need to calculate a mapping matrix from the canonical parallel view volume.
This mapping matrix need to know the height and width of the screen and the coordinate
system of the device (to make sure the whole calculated scene is shown). If one expected
to alternate between different devices to show the scene, it would be prudent to place this
functionality in the device class.

As we’ll only be using the class DotDevice, and the device will be known when the
projection matrix will be calculated, it does make sense to let the camera calculate both the
projection matrix and the mapping matrix in one go.
2.1.1.1 Moving the camera

It would be nice to be able to change the camera after the program has started, so the same scene can be visualized differently, such as zooming or panning. While this is beyond the scope of this assignment, the Camera class should be constructed so that it would be easy to implement. Changing the camera does give some possible problems. A number of vectors and points and a single matrix is calculated from the original values. So if these change, these values might be inconsistent with the new values. So a method to update the system after each change will be needed.

There are a lot of calculations in calculating the total projection matrix, so recalculating every time the user changes something might not be very efficient. Having a boolean value showing whether the projection matrix is in an inconsistent state or not is preferable. This advantage becomes even more pronounced because of the fact that we’ll need the total projection matrix many, many times. So making sure it’s always in a consistent state, and not having to recalculate it every time it’s needed is a definite advantage. As we only have one camera the increased memory usage will be insignificant.

From this discussion we get an outline of the Camera class:

<table>
<thead>
<tr>
<th>Camera</th>
<th>Know ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 × point</td>
<td>Position of VRP, PRP, CW and the two corner points of the view window.</td>
</tr>
<tr>
<td>6 × vector</td>
<td>The following vectors: VPN, VUP and the three basis vectors u, v and n of the view coordinate system.</td>
</tr>
<tr>
<td>matrix</td>
<td>The projection matrix.</td>
</tr>
<tr>
<td>calculate projection matrix</td>
<td>Given the necessary vectors and points, the Camera should be able to calculate the projection matrix.</td>
</tr>
</tbody>
</table>

2.2 Design of illumination

2.2.1 Source of Light

In this assignment we make the simplifying assumption that we use a single light source, shaped like a point. The light source can be represented by a class, containing the intensity of each of the three colors (RGB) and the coordinates of the light source. If the light source were to have specific properties (we may want to model it as a flashlight or the moon) these should be placed here.

If we had more than one light source, we would need to make some sort of container class, that should contain attenuation functionality and ambient light intensities, as these are “global” values, and only should be defined in one place. As we’ll only support one light source, all this functionality might as well be placed in the light source.
2.2.2 Properties of the object

The object has some properties, such as the specular reflection coefficient. These properties will have to be contained somewhere. As we know that we’ll be making a lot of polygons, it’ll be a waste of memory to let each polygon contain their own copy of these properties. If we contain all the object properties in a material class, all the polygon will need to know is a pointer to its instance material-class. This will also make it easy to represent scenes of different objects with different properties.

For each material, we’ll need a diffuse color, $O_{d\lambda}$, and a specular color, $O_{s\lambda}$. On top of this, we’ll need an ambient reflection coefficient $k_a$, a diffuse reflection coefficient $k_d$, and a specular reflection coefficient $k_s$. Furthermore, we’ll need a specular reflection coefficient $n$ that we might as well define as an integer.

2.2.3 The illumination function

As stated in the assignment, we must support Phong illumination. This means that we must be able to calculate the color of a given point. The function to do the calculation could be placed in the light source class. Then it would take a material and a point as arguments and do the calculations. Another possibility would be to make it as a function. This would, however, be in violation of the object oriented way of thinking, and as there is no advantage to implement it like this, it will not be done.

<table>
<thead>
<tr>
<th>LightSource</th>
</tr>
</thead>
<tbody>
<tr>
<td>Know ...</td>
</tr>
<tr>
<td>3 real numbers</td>
</tr>
<tr>
<td>2 colors</td>
</tr>
<tr>
<td>point</td>
</tr>
<tr>
<td>calculate attenuation factor</td>
</tr>
<tr>
<td>calculate Phong illumination</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Material</th>
</tr>
</thead>
<tbody>
<tr>
<td>Know ...</td>
</tr>
<tr>
<td>3 real numbers</td>
</tr>
<tr>
<td>1 integer</td>
</tr>
<tr>
<td>2 colors</td>
</tr>
</tbody>
</table>

2.3 Visible surface determination

Due to its simplicity, we have chosen to implement the Z-buffer as our hidden surface removal method. In our design, the actual Z-buffer simply contains the buffer structure, and a function that, given a $z$-value, determines if this is closer to the existing one. If so, the existing $z$-value and color is overwritten. The interpolation of the $z$-values takes place in the screen polygon, so there is nothing more to this.
2.4 Polygons

Eventually all surfaces will be divided into polygons which then will be drawn on the screen. Conceptually the polygons in our virtual world differs from the polygons on the screen. For example, the polygons in the world are three dimensional, they have normals and it would probably also be wise to associate some sort of material with each polygon. Since they reside in our virtual world, we would like to separate them from the whole drawing process which involves viewing angle, a light source and other things. The polygon on the screen, on the other hand, is tightly connected with the viewing angle, the light source and how things in the end will be drawn. It makes good sense to separate these two kinds of polygons into what we hereafter will name world polygons and screen polygons.

2.4.1 Polygons in the world

World polygons should model a three dimensional polygon, and it resides in our virtual world. It should have three points determining the position of its vertices, and it should have three normals, one for each vertex. Of course, an ordinary triangle has only one normal, but in our case the polygons will often model some surface, and we would like the normals of the polygons to be identical to the normals of the surface which the polygons model. These normals will be used for shading purposes.

Our world polygon should also know what material it is made of, which also will be used for shading purposes. Finally the world polygon should know how to react to projections, translations, shearing, rotation and other geometrical operations given a matrix.

In an object oriented language the world polygon’s interface would look as follows:

<table>
<thead>
<tr>
<th>Polygon</th>
<th>Know ...</th>
<th>Know how to ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 × point</td>
<td>Position of vertices</td>
<td>Given a matrix, the polygon should</td>
</tr>
<tr>
<td>3 × vector</td>
<td>Normals in vertices</td>
<td>be able to perform geometrical operations on itself.</td>
</tr>
<tr>
<td>material</td>
<td>Material which the polygon is made of.</td>
<td></td>
</tr>
</tbody>
</table>

2.4.2 Polygons on the screen

Polygon objects on the screen should have different properties than polygons in our virtual world. Since we will be using a Z-buffer as visible-surface algorithm, the polygon’s vertices have integral $x$ and $y$ coordinates and floating point $z$ coordinates. The $x$ and $y$ determine the polygon vertices’ position on the screen (which have no such thing as fractional coordinates) while $z$ determines the depth position of the vertices in the Z-buffer.

We imagine that the screen polygon should be able to determine its screen and Z-buffer coordinates given a world polygon, a light source, and information about how it is viewed from the spectator, i.e. a camera. This is done by determining the projection matrix which projects the world onto a parallel view volume, and then tell the world polygon to project itself with this matrix. The coordinates of the projected world polygon equal the screen coordinates if the view volume we have chosen also considers the screen dimensions.
CHAPTER 2. DESIGN

This should enable the screen polygon to draw itself to a Z-buffer given a shading model. Both Gouraud and Phong shading models require the points and normals from the world polygon’s vertices from which the screen polygon has been created, so it is reasonable that the screen polygon should keep a reference of some sort to the original polygon. Another way would be for it to keep references to the points and normals and to the material of the original polygon.

Other than this reference, we stress that the screen polygon does not reside in the virtual world and that it is generally considered another kind of object than the world polygon.

Our screen polygon object could look similar to this:

<table>
<thead>
<tr>
<th>ScreenPolygon</th>
</tr>
</thead>
<tbody>
<tr>
<td>Know ...</td>
</tr>
<tr>
<td>3 × point</td>
</tr>
<tr>
<td>World polygon</td>
</tr>
<tr>
<td>Know how to ...</td>
</tr>
<tr>
<td>construct itself</td>
</tr>
<tr>
<td>draw itself</td>
</tr>
</tbody>
</table>

2.5 Surfaces

We are required to implement parametric surfaces as both Bezier surfaces and as general parametric surfaces. For the Bezier surfaces, we get data (i.e. the patches and points) using a function provided by Knud Henriksen. We store one patch in a collection of points, and another vector then holds these pointvectors. To create displayable polygons from these, we go through each pointvector (i.e. patch) and subdivide it. We have chosen subdivision as the display method primarily because it is easy to implement. When a patch is divided to the maximum level, we display it. Assuming it is a well-formed patch, we calculate normals for each corner point. The normal for corner a is simply the cross product between a1 and a2, and the other normals are made similarly. The sign of the normals might be wrong, in which case we simply invert all normals by multiplying with -1. At this stage we create two triangles, abd and bcd, and pass these and their corresponding normals on to the next stage, creating world and screen polygons, where lighting, shading and scan conversion takes place.

![Figure 2.1: A patch with vectors from corner a to its two control points.](image)
There are cases where a patch is not well-formed, so we have made a check for deformities. As it is, we only check whether one side of the patch is collapsed as seen in figure 2.2; ideally we would also check for other deformities, such as having more than two points collapsed, or if three (or four) of the corner points are collinear.

![Figure 2.2: A patch with one side collapsed.](image)

If we find that one side has collapsed, we treat the patch as a triangle. This means calculating the normals of the three corners accordingly. In figure 2.2 it is the side \(ad\) that is collapsed. This means that the tangent from \(a\) to the first control point towards \(d\), must be calculated using the first control point towards \(c\) instead.

Our BezierPatch class reflects the fact that the method is simple, and has only a constructor and a method to draw it. The draw-method takes care of initiating subdivision of the patch. This in turn means subdividing each curve.

### 2.5.1 General parametric surfaces

As part of the assignment, we must visualise what is called a general parametric surface. This is a surface defined solely by four functions of the type \(f : \mathbb{R}^2 \to \mathbb{R}^3\). As suggested, we will draw the surface using sampling. This is done by looping over the two parameters, \(u\) and \(v\) (as you would over a 2D array). In the inner loop, we calculate values in four points, \((u, v), (u + \Delta u, v), (u + \Delta u, v + \Delta v), \text{and } (u, \Delta v)\). This gives us a rectangle, and to get the normals of each corner, we have made 2 functions for each of the four surfaces, that calculate the derivative in a given point. These two vectors are crossed, giving us a normal. Now the procedure is exactly as for the maximally subdivided Bezier patch above. To fix any problem that might occur with the sign of the normals here, we found that simply inverting it for one of the function gives us a good looking Klein bottle. It might be pure luck, but it seems to work in this case, i.e. for this camera, stepsize etc.

<table>
<thead>
<tr>
<th>BezierPatch</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Know ...</strong></td>
</tr>
<tr>
<td>16 × point material</td>
</tr>
<tr>
<td>16 points defining the Bezier patch.</td>
</tr>
<tr>
<td>The material the patch is made of.</td>
</tr>
<tr>
<td><strong>Know how to ...</strong></td>
</tr>
<tr>
<td>Subdivide itself</td>
</tr>
</tbody>
</table>
### ParametricSurface

<table>
<thead>
<tr>
<th>Know ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>12 vector functions</td>
</tr>
<tr>
<td>material</td>
</tr>
<tr>
<td>Sample itself</td>
</tr>
</tbody>
</table>
3 Implementation

We have implemented our visualization system in C++ with the help of the DotDevice class and other tools provided to us. We have used the template functionality of C++ to try and make the implementation generic. For instance, the internal data container in all of our classes is a template parameter (actually, it is a template template parameter, cf. [VJ03, pp. 102–103]) which can easily be changed. Also the default floating point type of most classes can be changed to something different from `float` if need be.

We have seperated the classes of the program into namespaces on the basis of what the classes model and what they do. In the following we will provide an overview of these namespaces and classes, and how the chain of events leading from the reading of a data file to the final output to the screen occur. A printout of the source files are provided in appendix B starting on page 50.

3.1 Namespaces

The classes we have created ourselves are all in the `graphic` namespace. From the `main` function in the global namespace we use our classes to create objects such as Bezier patches, Z-buffers, etc.

3.1.1 The global namespace

The global namespace contains the `main` function along with the helper functions and classes provided to us, namely the `ReadDataFile` function in `readdata.cpp` source file and the `DotDevice` class in the `DotDevice.cpp` and `DotDevice.h` source files.

The `main` function dispatches control to other functions and member functions of objects and is where the strings connect, so to speak.

3.1.2 The `graphic` namespace

The `graphic` namespace is divided into three sub-namespaces, `tools`, `world` and `draw`. The header file `graphic.h` bring these together by including the source files for the three sub-namespaces.

3.1.2.1 The `graphic::tools` namespace

The `graphic::tools` namespace contains the basic building blocks used throughout the classes. Here reside the `Vector`, `Point` and `Matrix` template classes along with `Color`. The `Vector` and `Point` template classes share a common base class named `BasicPoint` which
implement the common methods used by both, such as the subscript operator \texttt{operator[]}\texttt{.} Also, a base class to \texttt{Matrix, BasicMatrix} is defined, so that it can be used as a simple container. We do not use this class directly, though.

All of the above templates use the \texttt{Container} template class, which defines what kind of internal container all the classes use to store data. As default the \texttt{Container} template implements a standard C array with bounds check.

\subsection{3.1.2.2 The graphic::world namespace}

The \texttt{graphic::world} namespace holds the classes for the objects which we intuitively would place in a virtual world. This includes the \texttt{Polygon, BezierPatch} and \texttt{ParametricSurface} classes, but also the \texttt{Camera, LightSource, and Material} classes. All these classes use objects from the \texttt{graphic::tools} namespace, and some also use objects or pointers to objects from the \texttt{graphic::draw} namespace.

\subsection{3.1.2.3 The graphic::draw namespace}

The \texttt{graphic::world} namespace includes the \texttt{ScreenPolygon} and \texttt{Zbuffer} classes which does not really belong in the world since they are both specific as to how we want to visualize a scene.

\section{3.2 The pipeline}

Generally our program will read data from a file with the \texttt{ReadDataFile} to create a series of instances of \texttt{BezierPatch(es)}. The \texttt{BezierPatch} class has a draw function which first subdivides the patch into \texttt{Polygons} which are then used to create \texttt{ScreenPolygons} which is then drawn to a \texttt{Zbuffer}. To create \texttt{ScreenPolygons} we will need to provide a \texttt{Camera} object, a \texttt{Zbuffer} object and a \texttt{LightSource} object as well. The \texttt{Zbuffer} object will need a \texttt{DotDevice} object so that it knows where to draw to.

We do not save the intermediate steps from \texttt{BezierPatch} to the update of the \texttt{Zbuffer}. In other words, we provide the \texttt{BezierPatch} with the objects needed, and then temporary objects such as the \texttt{Polygons} and \texttt{ScreenPolygons} are created and drawn directly to the \texttt{Zbuffer}. There are a maximum of two \texttt{Polygons} and two \texttt{ScreenPolygons} in scope at a time. This lessens the memory requirements.

In the case where we visualize a surface given a set of parametric functions the pipeline differs. In this case we are not reading from a file and we are not creating Bezier surfaces either. Instead we create the polygons from the parametric functions directly by hard coding the functions into our program. In the same way as above, the \texttt{Polygons} are converted to \texttt{ScreenPolygons} and thereafter drawn to the \texttt{Zbuffer} before going out of scope.
4 Test and Visualizations

Programmers run tests on their programs in order to document what the program can and cannot do.

4.1 Building the test

A thorough test of a program should contain a test of each class and method (for each set of possible input) to investigate if they work as intended (by means of equivalency classes), and tests of the programs execution as a whole (black-box testing).

It is, however, a very big task to construct a full test of this program and it’s really not the point of this assignment. Thus, we will make a less thorough testing. The program can be divided into parts that can be tested separately. These parts could be the math functions, drawing of polygons, reading data and constructing Bezier surfaces, testing the z-buffer, and testing the combined program.

The test functions can be found in appendix B starting at page 114.

4.1.1 Math functions

While these functions are central to the execution of the program, each method will not be tested for each possible input. In most cases each method will be called once or twice, and if these calls give the expected results, the functionality will be assumed to work. Due to lack of time, the tests will not be described closer, but the test function testMath.cpp is included in appendix B. Prints from the test run can be found in appendix A.1.

4.1.2 Drawing polygons

Being able to correctly draw a polygon is another central feature of the program. This will be tested by usage of the DotDevice method TestLine to investigate if the polygons are drawn as expected. Seven different polygons will be drawn and investigated (see figure 1.1). On top of this it must be investigated if two polygons, placed side by side, draw the same pixels. This is best done by drawing them with a little distance, so it will become obvious. This test is included in the mentioned test.

4.1.3 Bezier surfaces

The more times the Bezier surface is subdivided, the more continuous it should appear. This can be tested by drawing a figure with more and more subdivisions. The results of this test is
CHAPTER 4. TEST AND VISUALIZATIONS

shown in figure 4.10. A second way of testing the Bezier functionality is to draw the outlines of the shapes, as seen in figure 4.11 and 4.12.

4.1.4 Z-buffer

The Z-buffer can be tested by placing one polygon in front of another and viewing the result. A more difficult test is to investigate if intersection between two polygons can be handled correctly, i.e. if one polygon can cut through the surface of another.

4.1.5 The combined program

Testing the whole is done by running the data-files given in the assignment. These will be made using the same source of light and reflection constants. They will be visualised from three different directions, from the side, from the top and from the front.

4.2 Results of the test

The results of the test runs are shown in the following pages. Most of the tests act as expected. There are, however, a few deviations.

One problem happened while testing if one polygon can cut through another. The black polygon in figure 4.4 should be a triangle cutting through another triangle. It most certainly shouldn’t appear be round! A similar problem can be seen in the test run of smart.data (figure 4.10), for less than two subdivisions of the bezier surface. In these test runs, black and dark polygons are shown in the figure. The problem seems to disappear when the number of subdivisions are increased - or maybe they simply get so small, that they aren’t visible.

There also seem to be some problems with visualizing the disc. There doesn’t seem to be any shading effects on the disc, regardless of the shading function used. This could happen if the disc is in fact just a plane. Thus there would be no difference in normal vectors, and the highlights non-existing. If it also were small, the difference from one edge to the other would be neglegent, and thus Gouraud shading wouldn’t add much.

In the visualization of the Klein bottle, viewed from the bottom, one can see a highlighted tube. This tube is highlighted as shadows are not implemented (so that was kind of expected).

All in all, we get som nice visualizations and much of the functionality seems to work. There are some problems that need to be solved, though.
CHAPTER 4. TEST AND VISUALIZATIONS

Figure 4.1: Test output from testing of polygon drawing.

Figure 4.2: Test output from testing of Gouraud shading.
CHAPTER 4. TEST AND VISUALIZATIONS

Figure 4.3: Test output from testing of Phong shading.

Figure 4.4: Test output from z-buffer testing.

Figure 4.5: Drawing Cone.data from different PRPs. The center one is drawn used Gouraud shading, the others with Phong shading.
Figure 4.6: Drawing Disk.data from different PRPs. The left is drawn using Phong shading, the right with Gouraud shading.

Figure 4.7: Drawing Normal1.data from different PRPs. The center one is drawn using Phong shading, the others with Gouraud shading.

Figure 4.8: Drawing Normal2.data from different PRPs. The center one is drawn using Gouraud shading, the others with Phong shading.
Figure 4.9: Drawing pain.data from different PRPs. The one on the left is drawn using Gouraud shading, the others using Phong shading.

Figure 4.10: Drawing from Smarts.data. The first four figures are subdivided 0, 1, 2 and 3 times, respectively. The bottom center is using Phong shading and the bottom right is using Gouraud shading.
CHAPTER 4. TEST AND VISUALIZATIONS

Figure 4.11: Drawing Sphere.data from different PRPs. The left one is drawn using Gouraud shading, the center one using Phong shading and the right is an outline of the polygons.

Figure 4.12: Drawing teapot.data from different PRPs. The top left uses Gouraud shading, the top and bottom right uses Phong shading, and the bottom left draws outlines of the polygons.
Figure 4.13: Drawing the parametric curves presented in the assignment. From left, the first and third uses line art, the second and the fifth uses Phong shading and the fourth uses Gouraud shading.
5 Conclusion

We believe that the final result is satisfactory. We have analysed the important and necessary aspects of 3D visualizations and have all acquired a good understanding of the basics. We believe that we have made a useful design which we might be able to build on in the future. The implementation runs and correctly reads the required data files and produces the required images. Not all tests went as expected, so there is still room for improvement.
A Test Runs

A.1 Output from testMath.cpp

Point constructors and assignment worked as expected
Vector operations worked as expected
Basic matrix operations worked as expected
Rotation, Scaling, Shearing and Translation works as expected
Color constructor ran as expected
Methods setRed, setGreen, and setBlue worked as expected!
Projection matrix of C1 and worldPRP as expected
Projection matrix and worldPRP of C2 as expected
Material constructor and get functionality works as expected
Illumination calculated as expected
Press ENTER to continue ...
B Source files

B.1 graphic_tools.hpp

/**************
Toolbox
**************/
#ifndef _GRAPHICTOOLS_HPP
#define _GRAPHICTOOLS_HPP

#include <vector>
#include <string>

namespace graphic{
    namespace tools{

    Container template "Container<T,N>" – where T is
    the type of elements – to be used in Vectors,
    Points and matrices need to have at least the following
    header (return and arg types omitted):
    Container(); // standard ctor
    Container(T); // value ctor
    operator[](i); // subscript operator
    operator[](i) const; // const subscript operator

    T is presumed to be a numeral type which works with
    standard arithmetics such as +, -, /, *.
*/

/* Simple container template using arrays. */
/* Simple container template using arrays. */

template<typename T, unsigned int N> class Container {
    protected:
        T data[N];
    public:
        Container() { // standard ctor
            for (unsigned int i = 0; i < N; ++i){
                data[i] = T();
            }
        }
    }
Container(T value) { // value ctor
    for (unsigned int i = 0; i < N; ++i)
        data[i] = value;
}

Container(Container<T,N> const& c) { // copy ctor
    for (unsigned int i = 0; i < N; ++i)
        data[i] = c.data[i];
}

// assignment operator:
Container<T,N>& operator=(Container<T,N> const& c) {
    for (unsigned int i = 0; i < N; ++i)
        data[i] = c.data[i];
    return *this;
}

// subscript operator:
T& operator[](unsigned int i){
    return data[i];
}

// subscript operator:
T const& operator[](unsigned int i) const {
    return data[i];
};

/**
 * BasicPoint template declaration */
/**
 * template<typename T = float,
 *          template<unsigned int, typename> class C = Container>
 * class BasicPoint
 * {
  protected:
  //Member variables
  C<T,N> mCoords;
  public:
  BasicPoint(); //Construct
  // Constructor from member type:
  BasicPoint(C<T,N> newVal);
  // c’tor from array:
  BasicPoint(T newVal[N]);
  //subscript operator:
  T& operator[](unsigned int i);
  //subscript operator:
  T const & operator[](unsigned int i) const;
*/
// BasicPoint equality operator:
bool operator==(BasicPoint<N,T,C> const & p) const;

// sets BasicPoint to value in type array:
void set(T newVal[N]);

// prints BasicPoint to screen:
void print() const;

// this prints string prior to printing the BasicPoint:
void print(std::string str) const;
};

// Vector template declaration

// Vector template declaration

template<unsigned int N, 
type T = float, 
template<typename, unsigned int> class C = Container >
class Vector : public BasicPoint<N,T,C>
{
public:
  Vector() : BasicPoint() { } // Constructor

  // Constructor from member type:
  Vector(C<T,N> newVal) : BasicPoint(newVal) { };

  // c'tor from array:
  Vector(T newVal[N]) : BasicPoint(newVal) { };

  // dot-product (Vector-Vector):
  T operator*(const Vector<N,T,C>& v) const;

  // Vector-addition:
  Vector<N,T,C> operator+(Vector<N,T,C> const& v) const;

  // Vector-addition:
  Vector<N,T,C> & operator+=(Vector<N,T,C> const& v);

  // Vector-subtraction:
  Vector<N,T,C> operator-(Vector<N,T,C> const& v) const;

  // the Vector-scalar and scalar-Vector multiplication
  // will be defined in global scope.

  // functions
  T length() const; // calculates/returns length of the Vector
  void normalize(); // normalizes the Vector
  // we need a Vector crossproduct, but only for 3D Vectors,
  // so it'll be defined in global scope.
}; // Vector template

// Point template declaration */
APPENDIX B. SOURCE FILES

157 //**************************************************************/
158 // Point is inherited from BasicPoint to avoid implicit cast from
159 // Vector to Point.
160 template<unsigned int N, typename T = float,           
161           template <typename, unsigned int> class C = Container >
162 class Point : public BasicPoint<N,T,C>
163 {
164     public:
165         Point() : BasicPoint() {}
166         // Constructor from member type:
167         Point(C<T,N> newVal) : BasicPoint(newVal) {}
168         // c'tor from array:
169         Point(T newVal[N]) : BasicPoint(newVal) {}
170         // moving point:
171         void move(Vector<N,T,C> const & v);
172     } // Point template
173
174} //**************************************************************/
175// Basic matrix template declaration */
176//**************************************************************/
177template<unsigned int N, typename T = float,           
178           template <typename, unsigned int> class C = Container >
179class BasicMatrix
180{
181     protected:
182     // members
183     C<C<T,N>,N> mElements;
184     public:
185         BasicMatrix(); // constructor, constructs identity Matrix
186         BasicMatrix(T newVal[N][N]); // constructor from 2-dim array
187         void set(T newVal[N][N]); // value assignment from array
188         // subscript: Matrix[i] returns i'th row. Matrix[i][j]
189         // returns element on i'th row in j'th column.
190         C<T,N>& operator [] (unsigned int i); // subscript operator
191         C<T,N> const& operator [] (unsigned int i) const; // subscript op.
192         bool operator ==(BasicMatrix<N,T,C> const & m) const; // Equality op.
193         void print() const; // instruction to print Vector to screen
194         // this prints string prior to printing the Vector:
195         void print(std::string str) const;
196     }
197
198} //**************************************************************/
199// Matrix template declaration */
200template<unsigned int N, typename T = float,           
201           template <typename, unsigned int> class C = Container >
202class Matrix : public BasicMatrix<N,T,C>
203{
204     public:
205         // constructor, constructs identity Matrix:
206         Matrix() : BasicMatrix() {}
207         // constructor from 2-dim array:
208         Matrix(T newVal[N][N]) : BasicMatrix(newVal) {};
209     } // Matrix template declaration */
210
APPENDIX B. SOURCE FILES

// Matrix multiplication:
Matrix<N,T,C> operator*(Matrix<N,T,C> const & m) const;

// Matrix-vector product:
Vector<N,T,C> operator*(Vector<N,T,C> const & v) const;

// Matrix addition:
Matrix<N,T,C> operator+(Matrix<N,T,C> const & m) const;

// Matrix subtraction:
Matrix<N,T,C> operator-(Matrix<N,T,C> const & m) const;

void transpose(); // Matrix transpose on this Matrix
Vector<N,T,C> row(unsigned int) const;
Vector<N,T,C> col(unsigned int) const;

void rotateX(const T angle); // rotation around the x-axis
void rotateY(const T angle); // rotation around the y-axis
void rotateZ(const T angle); // rotation around the z-axis

// Translation by the specified amount:
void translate(const Vector<N,T,C> v);
void translate(const Point<N,T,C> p);

// mVec contains the values for each xyz-scaling:
void scale(const Vector<N,T,C> v);
void scale(const Point<N,T,C> p);

// xy shearing:
void shearXY(const float shx, const float shy);

// xz shearing:
void shearXZ(const float shx, const float shy);

// yz shearing:
void shearYZ(const float shx, const float shy);

};

// template<typename T = double>
class Color {
protected:
  double mRed, mGreen, mBlue;

public:
  Color(); // default c'tor

  // c'tor initializing with doubles [0,1]:
  Color(double red, double green, double blue);
  // c'tor initializing with ints [0,255]
  //Color(unsigned short int red, unsigned short int green,
  //unsigned short int blue);
  Color const & setColor(Color const & color);
double setRed(double value);  // echoes value as return value.
double setGreen(double value);
double setBlue(double value);
// echoes mRed/mGreen/mBlue as return value.
double changeRed(double value);
double changeGreen(double value);
double changeBlue(double value);
double getRed() const;
double getGreen() const;
double getBlue() const;
bool operator==(Color const & c) const;  // color equality op.
void print() const;
void print(std::string str) const;
};

Color const green(0, 1.0, 0);
Color const red(1.0, 0, 0);
Color const blue(0, 0, 1.0);
Color const orange(1.0, 0.5, 0);
Color const yellow(1.0, 1.0, 0);
Color const magenta(1.0, 0.0, 0.5);
Color const violet(1.0, 0, 1.0);
Color const cyan(0, 1.0, 1.0);
Color const white(1.0, 1.0, 1.0);
Color const grey(0.5, 0.5, 0.5);
Color const black(0.0, 0.0, 0.0);

template <typename T>
inline T sign(T value) {
    if (value < 0)
        return (T) -1;
    else if (value == 0)
        return (T) 0;
    return (T) 1;
}

#include <cmath>  // sqrt(x)
#include <stdexcept>
#include <iostream>
#include "graphic_tools.hpp"
#define PI 3.14159265358979323846
using namespace graphic::tools;

B.2 graphic_tools.cpp

#pragma once

#include <cmath>  // sqrt(x)
#include <stdexcept>
#include <iostream>
#include "graphic_tools.hpp"
#define PI 3.14159265358979323846
using namespace graphic::tools;

/***********************************************************/
/* BasicPoint template implementation */
// BasicPoint template default constructor
template<typename N, typename T, typename C>
inline BasicPoint<N,T,C>::BasicPoint() : mCoords() {}

// BasicPoint template member container type constructor
template<typename N, typename T, typename C>
inline BasicPoint<N,T,C>::BasicPoint(C<T,N> c) : mCoords(c) {}

// BasicPoint template array constructor
template<typename N, typename T, typename C>
inline BasicPoint<N,T,C>::BasicPoint(T newVal[N]) {
  for(unsigned int i = 0; i < N; ++i)
    mCoords[i] = newVal[i];
}

// BasicPoint subscript operators
template<typename N, typename T, typename C>
inline T & BasicPoint<N,T,C>::operator[](unsigned int i) {
  return mCoords[i];
}

// BasicPoint equality operator
template<typename N, typename T, typename C>
inline bool BasicPoint<N,T,C>::operator==(BasicPoint<N,T,C> const & p) const {
  for(unsigned int i = 0; i < N; ++i) {
    T compareVal = abs(mCoords[i]/10000.);
    if(compareVal < 0.0001)
      compareVal = 0.0001;
    if( abs(mCoords[i] - p[i]) > compareVal ){
      //std::cout << "Error in coordinate " << i << ", got " << p[i] << " and expected " << mCoords[i] << std::endl;*/
      return false;
    }
    std::cout << abs(mElements[i][j]/10.) << std::endl; */
  }
  return true;
}

// sets value of BasicPoint coordinates to values given in array
template<typename N, typename T, typename C>
template<typename N, typename T, typename C>
inline BasicPoint<N,T,C>::BasicPoint(T newVal[N]) {  //BasicPoint template default constructor
inline void BasicPoint<N,T,C>::set(T newVal[N]) {
    for(unsigned int i = 0; i<N; ++i){
        mCoords[i] = newVal[i];
    }
}

// prints the BasicPoint to the screen
template<typename int N, typename T, template<typename, unsigned int> class C>
inline void BasicPoint<N,T,C>::print() const{
    std::cout << " ( ";
    for(unsigned int i = 0; i<N-1; ++i)
        std::cout << mCoords[i] << ",";
    std::cout << mCoords[N-1] << ") " << std::endl;
}

// prints str to screen before printing BasicPoint
template<typename int N, typename T, template<typename, unsigned int> class C>
inline void BasicPoint<N,T,C>::print(std::string str) const{
    std::cout << str;
    this->print();
}

/*********************
* Point template implementation *
***********************
// moving point
template<typename int N, typename T, template<typename, unsigned int> class C>
inline void Point<N,T,C>::move(Vector<N,T,C> const & v){
    for(int i = 0; i < N; ++i) {
        mCoords[i] += v[i];
    }
}

/*********************
* Vector template implementation *
***********************
// dot-product (Vector*Vector)
template<typename int N, typename T, template<typename, unsigned int> class C>
inline T Vector<N,T,C>::operator*(const Vector<N,T,C>& v) const {
    T result = 0;
    for(unsigned int i=0; i<N ; i++){
        result += mCoords[i]*v[i];
    }
    return result;
}

/**************************
* The first function allows scalar-Vector multiplication *
**************************
template<typename int N, typename T1, typename T2,
template<typename, unsigned int> class C>
inline Vector<N,T1,C> operator*(T2 const & s, Vector<N,T1,C> const & v) {
    Vector<N,T1,C> result;
    for(unsigned int i=0 ; i<N ; i++){
        result.mCoords[i] = s*v.mCoords[i];
    }
    return result;
}
result[i] = s*v[i];
}
return result;
}

// the second function allows Vector−scalar multiplication
template<unsigned int N, typename T1, typename T2,
    template <typename, unsigned int> class C
    inline Vector<N,T1,C> operator*(Vector<N,T1,C> const& v, T2 const& s) {
    return s*v;
}

// Vector−Vector addition
template<unsigned int N, typename T, template <typename, unsigned int> class C
    inline Vector<N,T,C> Vector<N,T,C>::operator+(Vector<N,T,C> const& v) const{
    Vector<N,T,C> result;
    for(unsigned int i =0; i<N ; i++){
        result[i] = mCoords[i]+v[i];
    }
    return result;
}

template<unsigned int N, typename T, template <typename, unsigned int> class C
    inline Vector<N,T,C> & Vector<N,T,C>::operator+=(Vector<N,T,C> const& v) {
    mCoords[i] += v[i];
}
    return *this;
}

// Vector−Vector addition
template<unsigned int N, typename T, template <typename, unsigned int> class C
    inline Vector<N,T,C> Vector<N,T,C>::operator−(Vector<N,T,C> const& v) const{
    Vector<N,T,C> result;
    for(unsigned int i =0; i<N ; i++){
        result[i] = mCoords[i]−v[i];
    }
    return result;
}

// Vector template length function
// this function calculates the length of the Vector
// the length is the square root of the sum over the square of all elements
template<unsigned int N, typename T, template <typename, unsigned int> class C
    inline T Vector<N,T,C>::length() const{
    T length = 0;
    for(unsigned int i = 0; i<N ; ++i){
        length += mCoords[i] * mCoords[i];
    }
    length = sqrt(length);
    return length;
}

// Vector normalize function
// this function normalizes the Vector, so it has a length of 1
template<unsigned int N, typename T, template <typename, unsigned int> class C
    inline void Vector<N,T,C>::normalize(){
T length = this->length();
for (unsigned int i = 0; i < N; ++i){
  mCoords[i] /= length;
}

// the Vector cross product is only allowed for 3d Vectors
template<typename T, template<typename, unsigned int> class C>
inline Vector<3,T,C> cross(Vector<3,T,C>& v1, Vector<3,T,C>& v2){
  Vector<3,T,C> newVector;
  newVector[1] = v1[2]*v2[0] - v1[0]*v2[2];
  newVector[2] = v1[0]*v2[1] - v1[1]*v2[0];
  return newVector;
}

// calculates the vector pointing from p1 to p2
template<unsigned int N, typename T, template<typename, unsigned int> class C>
inline Vector<N,T,C> vectorFromPoints(const Point<N,T,C>& p1, const Point<N,T,C>& p2){
  Vector<N,T,C> newVector;
  for (int i = 0; i < N; ++i)
    newVector[i] = p2[i] - p1[i];
  return newVector;
}

/* *****************************************************
   Basic matrix implementation *
*****************************************************/

// BasicMatrix default constructor
template<unsigned int N, typename T, template<typename, unsigned int> class C>
inline BasicMatrix<N,T,C>::BasicMatrix() : mElements(C<T,N>()){
  for (unsigned int i = 0; i < N; ++i){
    mElements[i][i] = 1;
  }
}

// BasicMatrix constructor from array
template<unsigned int N, typename T, template<typename, unsigned int> class C>
inline BasicMatrix<N,T,C>::BasicMatrix(T value[N][N]) {
  set(value);
}

template<unsigned int N, typename T, template<typename, unsigned int> class C>
inline void BasicMatrix<N,T,C>::set(T value[N][N]){ 
  for (unsigned int i = 0; i < N; ++i){
    for (unsigned int j = 0; j < N; ++j){
      mElements[i][j] = value[i][j];
    }
  }
}
// BasicMatrix subscript operators
template<unsigned int N, typename T, 
    template<typename, unsigned int> class C>
inline C<N,F> BasicMatrix<N,T,C>::operator[](unsigned int i) { 
    return mElements[i];
}

// BasicMatrix equality operator, returns true
// if all values in the matrix are equal
// used in testing by assert
template<unsigned int N, typename T, 
    template<typename, unsigned int> class C>
inline bool BasicMatrix<N,T,C>::operator==(BasicMatrix<N,T,C> const & m) const { 
    for (unsigned int i = 0; i < N; ++i) { 
        for (unsigned int j = 0; j < N; ++j) { 
            T compareVal = abs(mElements[i][j]/10000.);
            compareVal = 0.0001;
            // aka floating point ==
            if( abs(mElements[i][j] - m[i][j]) > compareVal ){
                std::cout << Error.in(" ") << i << "", 
                    << j << ", got ", 
                    << m[i][j] << "and expected ", 
                    << mElements[i][j] << std::endl;
                /*std::cout << " error allowed: " << (mElements[i][j]/10.
                    << " 
                    << abs(mElements[i][j] - m[i][j]) << std::endl;*/
                return false;
            }
        }
    }
    return true;
}

// print *this BasicMatrix
template<unsigned int N, typename T, 
    template<typename, unsigned int> class C>
inline void BasicMatrix<N,T,C>::print() const { 
    std::cout << " [ [ ";
    for (unsigned int j = 0; j < N-1; ++j) { 
        std::cout << mElements[0][j] << ", ";
    }
    std::cout << mElements[0][N-1] << "] " << std::endl;
    for (unsigned int i = 1; i < N-1; ++i) { 
        std::cout << " [ [ ";
        for (unsigned int j = 0; j < N-1; ++j) { 
            std::cout << mElements[i][j] << ", ";
        }
        std::cout << mElements[i][N-1] << "] " << std::endl;
    }
}
APPENDIX B. SOURCE FILES

```cpp
std::cout << mElements[i][N-1] << "\"" << std::endl;
}

std::cout << ";\";
for (unsigned int j = 0; j < N-1; ++j) {
  std::cout << mElements[N-1][j] << ";\";
}
std::cout << mElements[N-1][N-1] << "]\]" << std::endl;
```

```cpp
// print *this BasicMatrix with prefix.
template<unsigned int N, typename T,
inline void BasicMatrix<N,T,C>::print (std::string str) const
  std::cout << str << std::endl;
print();
}

/** Matrix implementation */
/***************************************************************************/
// Matrix multiplication
template<unsigned int N, typename T,
#endif
```
inline Matrix<N,T,C> Matrix<N,T,C>::operator+(Matrix<N,T,C> const & m) const{
    Matrix<N,T,C> returnMatrix;
    for(unsigned int i = 0; i < N; ++i) {
        for(unsigned int j = 0; j < N; ++j) {
            returnMatrix[i][j] = mElements[i][j] + m[i][j];
        }
    }
    return returnMatrix;
}

// Matrix subtraction
inline Matrix<N,T,C> Matrix<N,T,C>::operator-(Matrix<N,T,C> const & m) const{
    Matrix<N,T,C> returnMatrix;
    for(unsigned int i = 0; i < N; ++i) {
        for(unsigned int j = 0; j < N; ++j) {
            returnMatrix[i][j] = mElements[i][j] - m[i][j];
        }
    }
    return returnMatrix;
}

// Matrix transpose
inline void Matrix<N,T,C>::transpose() {
    Matrix<N,T,C> copy(*this);
    for(unsigned int i = 0; i < N; ++i) {
        for(unsigned int j = 0; j < N; ++j) {
            mElements[i][j] = copy[j][i];
        }
    }
}

template<Unsigned int N, typename T, template<typename, unsigned int> class C>
inline Vector<N,T,C> Matrix<N,T,C>::row(unsigned int i) const{
    if(i < N){
        return Vector<N,T,C>(mElements[i]);
    } else {
        throw std::out_of_range("Matrix_row_index_out_of_range!");
    }
}

template<Unsigned int N, typename T, template<typename, unsigned int> class C>
inline Vector<N,T,C> Matrix<N,T,C>::col(unsigned int j) const{
    if(j < N){
        C<T> mColumn;
        for(unsigned int i = 0; i < N; ++i) {
            mColumn[i] = mElements[i][j];
        }
        return Vector<N,T,C>(mColumn);
    } else {
        throw std::out_of_range("Matrix_column_index_out_of_range!");
    }
}

// rotates the matrix around the x-axis, by the value in angle
template<unsigned int N, typename T, template<typename, unsigned int> class C>
inline void Matrix<N,T,C>::rotateX(const T angle) {
if (N == 4) {
    T rad = angle * PI/180;  // converting from angle to radian
    T rotArray[4][4] = {{1,0,0,0}, {0,cos(rad),-sin(rad),0},
                        {0,sin(rad),cos(rad),0}, {0,0,0,1}};
    Matrix<N,T,C> rotationMatrix(rotArray);
    mElements = (rotationMatrix * (*this)).mElements;
} else {
    throw std::invalid_argument("May only rotate 4x4 matrices!");
}
}

// rotates the matrix around the y-axis, by the value in angle
template<unsigned int N, typename T, template<typename, unsigned int> class C>
inline void Matrix<N,T,C>::rotateY(const T angle) {
if (N == 4) {
    T rad = angle * PI/180;  // converting from angle to radian
    T rotArray[4][4] = {{cos(rad),0,sin(rad),0}, {0,1,0,0},
                        {-sin(rad),0,cos(rad),0}, {0,0,0,1}};
    Matrix<N,T,C> rotationMatrix(rotArray);
    mElements = (rotationMatrix * (*this)).mElements;
} else {
    throw std::invalid_argument("May only rotate 4x4 matrices!");
}
}

// rotates the matrix around the z-axis, by the value in angle
template<unsigned int N, typename T, template<typename, unsigned int> class C>
inline void Matrix<N,T,C>::rotateZ(const T angle) {
if (N == 4) {
    T rad = angle * PI/180;  // converting from angle to radian
    T rotArray[4][4] = {{cos(rad),-sin(rad),0,0}, {sin(rad),cos(rad),0,0},
                        {0,0,1,0}, {0,0,0,1}};
    Matrix<N,T,C> rotationMatrix(rotArray);
    mElements = (rotationMatrix * (*this)).mElements;
} else {
    throw std::invalid_argument("May only rotate 4x4 matrices!");
}
}

// translates by the values stored in vector
template<unsigned int N, typename T, template<typename, unsigned int> class C>
inline void Matrix<N,T,C>::translate(const Vector<N,T,C> v) {
if (N == 4) {
    T transArray[4][4] = {{1,0,0,v[0]}, {0,1,0,v[1]}, {0,0,1,v[2]},
                          {0,0,0,1}};
    Matrix<N,T,C> translationMatrix(transArray);
    mElements = (translationMatrix * (*this)).mElements;
} else {
    throw std::invalid_argument("May only rotate 4x4 matrices!");
}
```cpp
// translates by the values stored in point
inline void Matrix<N,T,C>::translate(const Point<N,T,C>& p){
    if(N == 4){
        T transArray[4][4] = {
            {1,0,0,p[0]}, {0,1,0,p[1]}, {0,0,1,p[2]},
            {0,0,0,1}};
        Matrix<N,T,C> translationMatrix(transArray);
        mElements = (translationMatrix * (*this)).mElements;
    } else{
        throw std::invalid_argument("May only rotate 4x4 matrices!");
    }
}

// mVec contains the values for each xyz-scaling
inline void Matrix<N,T,C>::scale(const Vector<N,T,C>& v){
    if(N == 4){
        T scaleArray[4][4] = {
            {v[0],0,0,0}, {0,v[1],0,0}, {0,0,v[2],0},
            {0,0,0,1}};
        Matrix<N,T,C> scaleMatrix(scaleArray);
        mElements = (scaleMatrix * (*this)).mElements;
    } else{
        throw std::invalid_argument("May only rotate 4x4 matrices!");
    }
}

// mVec contains the values for each xyz-scaling
inline void Matrix<N,T,C>::scale(const Point<N,T,C>& p){
    if(N == 4){
        T scaleArray[4][4] = {
            {p[0],0,0,0}, {0,p[1],0,0}, {0,0,p[2],0},
            {0,0,0,1}};
        Matrix<N,T,C> scaleMatrix(scaleArray);
        mElements = (scaleMatrix * (*this)).mElements;
    } else{
        throw std::invalid_argument("May only rotate 4x4 matrices!");
    }
}

// xy shearing
inline void Matrix<N,T,C>::shearXY(const float shx, const float shy){
    if(N == 4){
        T shearArray[4][4] = {
            {1,0,shx,0}, {0,1,shy,0}, {0,0,1,0},
            {0,0,0,1}};
        Matrix<N,T,C> shearMatrix(shearArray);
        mElements = (shearMatrix * (*this)).mElements;
    } else{
        throw std::invalid_argument("May only rotate 4x4 matrices!");
    }
}

// xz shearing, lecture slides
```
template<unsigned int N, typename T, template<typename, unsigned int> class C>
inline void Matrix<N,T,C>::shearXZ(const float shx, const float shz)
{
    if(N == 4)
    {
        T shearArray[4][4] = {
            {1, shx, 0, 0}, {0, 1, 0, 0}, {0, shz, 1, 0},
            {0, 0, 0, 1}
        };
        Matrix<N,T,C> shearMatrix(shearArray);
        mElements = (shearMatrix * (*this)).mElements;
    }
    else
    {
        throw std::invalid_argument("May only rotate 4x4 matrices!");
    }
}

// xy shearing, lecture slides
template<unsigned int N, typename T, template<typename, unsigned int> class C>
inline void Matrix<N,T,C>::shearYZ(const float shy, const float shz)
{
    if(N == 4)
    {
        T shearArray[4][4] = {
            {1, 0, 0, 0}, {shy, 1, 0, 0}, {shz, 0, 1, 0},
            {0, 0, 0, 1}
        };
        Matrix<N,T,C> shearMatrix(shearArray);
        mElements = (shearMatrix * (*this)).mElements;
    }
    else
    {
        throw std::invalid_argument("May only rotate 4x4 matrices!");
    }
}

/* *******************************************************
 * Color implementation *
 *******************************************************/

// default c'tor
Color::Color() : mRed(), mGreen(), mBlue() {}

// c'tor initializing with doubles [0, 1]
inline Color::Color(double red, double green, double blue)
{
    // setting red
    if( red > 1)
        mRed = 1;
    else if( red < 0)
        mRed = 0;
    else
        mRed = red;

    // setting green
    if( green > 1)
        mGreen = 1;
    else if( green < 0)
        mGreen = 0;
    else
        mGreen = green;

    // blue
    if( blue > 1)
        mBlue = 1;
    else if( blue < 0)
        mBlue = 0;
APPENDIX B. SOURCE FILES

```cpp
else
    mBlue = blue;
}

// sets the color in this object as the color-value in the
inline Color const & Color::setColor(Color const & newColor) { 
    mRed = newColor.red();
    mGreen = newColor.green();
    mBlue = newColor.blue();
    return newColor;
}

// sets the value in mRed to value
inline double Color::setRed(double red) { 
    // setting red
    if (red > 1)
        mRed = 1;
    else if (red < 0)
        mRed = 0;
    else
        mRed = red;
    return mRed;
}

// sets the value in mGreen to value
inline double Color::setGreen(double green) { 
    // setting green
    if (green > 1)
        mGreen = 1;
    else if (green < 0)
        mGreen = 0;
    else
        mGreen = green;
    return mGreen;
}

// sets the value in mBlue to value
inline double Color::setBlue(double blue) { 
    // setting blue
    if (blue > 1)
        mBlue = 1;
    else if (blue < 0)
        mBlue = 0;
    else
        mBlue = blue;
    return mBlue;
}

// changes mRed with value
inline double Color::changeRed(double value) { 
    mRed += value;
    if (mRed < 0.0)
        mRed = 0.0;
    if (mRed > 1.0)
        mRed = 1.0;
    return mRed;
}

// changes mGreen with value
```
inline double Color::changeGreen(double value){
    mGreen += value;
    if(mGreen < 0.0) mGreen = 0.0;
    if(mGreen > 1.0) mGreen = 1.0;
    return mGreen;
}

// changes mBlue with value
inline double Color::changeBlue(double value){
    mBlue += value;
    if(mBlue < 0.0) mBlue = 0.0;
    if(mBlue > 1.0) mBlue = 1.0;
    return mBlue;
}

inline double Color::getRed() const{ return mRed; }

inline double Color::getGreen() const{ return mGreen; }

inline double Color::getBlue() const{ return mBlue; }

// Color equality operator
bool Color::operator==(Color const & c) const{
    if( abs(mRed - c.getRed()) > 0.001 ||
        abs(mGreen - c.getGreen()) > 0.001 ||
        abs(mBlue - c.getBlue()) > 0.001 )
    {
        std::cout << "Error in color equality! Had: (" << mRed
        << ", " << mGreen << ", " << mBlue << ") Got: (" 
        << c.getGreen() << ", " << c.getBlue() 
        << ")" << std::endl;
        //this->print("Error in color equality! Had: ");
        //c.print("got ");
        return false;
    }
    return true;
}

void Color::print() const
{
    std::cout << "(" << mRed << ", " << mGreen << ", " 
    << mBlue << ")" << std::endl;
}
B.3 graphic_world.hpp

```cpp
*/
'*∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗

#include <string>
#include <vector>

using namespace graphic;

namespace draw {
    template<typename T>
    class Zbuffer;
}
using namespace draw;

namespace world {
    template<typename T = float>
    class Material;
    template<typename T = float>
    class Polygon {
    protected:
        Point<3,T> mPoints[3];
        Vector<3,T> mNormals[3];
        Material<T> const* mMaterial;
    public:
        Polygon(Point<3,T> const& p0, Point<3,T> const& p1,
                Point<3,T> const& p2, Vector<3,T> const& v0,
                Vector<3,T> const& v1, Vector<3,T> const& v2,
                Material<T> const& m);
        Polygon(Polygon<T> const & p);
        Polygon() {};
        Point<3,T> const& point(unsigned int i, Point<3,T> const& p);
        Point<3,T> const& point(unsigned int i) const;
        Vector<3,T> const& normal(unsigned int i, Vector<3,T> const& v);
        Vector<3,T> const& normal(unsigned int i) const;
        Material<T> const& material(Material<T> const& m);
        Material<T> const& material() const;
        void projectPoints(Matrix<4,T> const& matrix);
        void projectNormals(Matrix<4,T> const& matrix);
        void print(std::string str = "");
    };
```
APPENDIX B. SOURCE FILES

template<typename T = float> class Camera{
protected:
    // these are set by the constructor, and can be accessed
    // and modified by get and set functionality.
    // the origin of view coordinate system in world coordinates:
    Point<3,T> VRP;
    // the eye of the beholder in view coordinates:
    Point<3,T> PRP;
    // the eye of the beholder in world coordinates:
    Point<3,T> worldPRP;
    // the view plane normal:
    Vector<3,T> VPN;
    // view up vector:
    Vector<3,T> VUP;
    // delimiter of window, must have minimum values (u_min, v_min):
    Point<3,T> w1;
    // delimiter of window, must have maximum values (u_max, v_max):
    Point<3,T> w2;
    T B;  // the signed distance from VRP to back clipping plane
    T F;  // the signed distance from VRP to front clipping plane
    // these can only be modified by modifying the fields above
    // (in which case the fields below must be recalculated
    Point<3,T> CW;  // center of window
    Vector<3,T> DOP;  // direction of projection
    Vector<3,T> u;  // "x" axis
    Vector<3,T> v;  // "y" axis
    Vector<3,T> n;  // "z" axis
    Matrix<4,T> ProjMatr;  // the projection matrix
    // making sure if the projection matrix is in a valid state:
    bool projMatrixIsValid;

public:
    // default constructor
    // Camera();
    // explicit constructor
    Camera(Point<3,T> mVRP, Point<3,T> mPRP, Vector<3,T> mVPN,
           Vector<3,T> mVUP, Point<3,T> mw1, Point<3,T> mw2, T mB, T mF);
    // handy constructor
    // position: position (point) of perspective reference point in world coords
    // aim: point camera is aiming for
    // up: up-vector
    // size: vertical and horizontal size of view plane
    // zoom:
    Camera(Point<3,T> position, Point<3,T> aim, Vector<3,T> up,
           T size = 1, T zoom = 1, T frontClippingPlane = 0,
           T backClippingPlane = -20);
    // set functionality – note that these methods will need
    // to update a number of fields
    void setVRP(Point<3,T> newVRP);
    void setPRP(Point<3,T> newPRP);
    void setVPN(Vector<3,T> newVPN);
    void setVUP(Vector<3,T> newVUP);
    void setW1(Point<3,T> newW1);
    void setW2(Point<3,T> newW2);
// if pan, zoom and similar functionality is to be added,
// this is the place!

// this method forces an update, i.e. after use of a set function:
void update();

// this method calculates the normalization matrix N:
Matrix<4,T> calculateNormalization();

// this method calculates the projection matrix M x N:
Matrix<4,T> calculateProjection();

// this method calculates the conversion matrix C from canonical
// perspective to canonical parallel view volume:
Matrix<4,T> calculateParPalConv();

// this method calculates mapping matrix MAP from canonical
// parallel view volume into DotDevice coordinates in width x
// height system:
Matrix<4,T> calculateMapping(int width, int heigth);

// this method calculates MAP x C x N:
Matrix<4,T> calculateTotalProjection(int width, int heigth);

// this method returns ProjMat if it exists, or recalculates
// if needed:
Matrix<4,T> getProjectionMatrix(int width, int heigth);

// returns the PRP in world coordinates:
Point<3,T> const & getWorldPRP() const;

// prints the fields of the Camera to the screen:
void print();

// preceeds the printing of Camera fields by printing str:
void print(std::string str);

};

// template< name T = float >
class Material {
protected:
    Color diffuseColor; // O_d
    Color specularColor; // O_s
    T ka; // ambient reflection coefficient
    T kd; // diffuse reflection coefficient
    T ks; // specular reflection coefficient
    int n; // specular reflection exponent

public:
    // constructor
    Material(Color mDColor, Color mSColor, T mKa, T mKd, T mKs, int mN);
    Color getDiffuseColor() const;
    Color getSpecularColor() const;
    T getKa() const;
    T getKd() const;
    T getKs() const;

    // returns specular reflection exponent:
    int getN() const;

    // prints material fields to screen:
    void print();

    // prints str to screen and then material fields to screen:
```cpp
void print(std::string str);
};

template<typename T = float>
class LightSource {
protected:
  // the coordinates of the light source:
  Point<3,T> coordinates;
  // the coefficients of the attenuation function:
  Point<3,T> attenuation;
  // the level of ambient light:
  Color ambient;
  // the intensities of the light source:
  Color intensities;
public:
  // LightSource();
  // constructor:
  LightSource(Point<3,T> const& mCoordinates,
               Point<3,T> const& mAttenuation, Color const& mAmbient,
               Color const& mIntensities = white);
  LightSource(Point<3,T> const& position, T c1 = 0,
               T c2 = 0.1, T c3 = 0);

  // returns the coordinates of the light source:
  Point<3,T> const & getCoordinates() const;
  // returns the coefficients of the attenuation function:
  Point<3,T> const & getAttenuation() const;
  // returns the attenuation factor, given the distance
  // from the light source:
  T getAttenuationFactor(T distance) const;
  // returns the level of ambient light:
  Color const & getAmbient() const;
  // returns the intensities of the light source:
  Color const & getIntensities() const;

  // calculates illumination at given point:
  Color PhongIllumination(const Material<T>& mMaterial,
                          Point<3,T>& mPoint, const Vector<3,T>& normal,
                          Camera<T>& mCamera) const;

  // prints light source fields to screen:
  void print() const;

  // prints str to screen and then light source
  // fields to screen:
  void print(std::string str) const;
};

// The BezierPatch class handles subdivision of patches.
// When subdivision ends, this class creates a WorldPolygon
// and a ScreenPolygon, and calls draw on the ScreenPolygon.
```
template<typename T>
class BezierPatch {
    public:
        BezierPatch (std::vector<Point<3, T>> patch, Material<T> const &material, int maxlvl, char shading, int sign);
        void drawPatch(Zbuffer<T> &zbuffer, Camera<T> &camera,
                        LightSource<T> const &light);
    
    private:
        void subdividePatch (Point<3, T> patch[4][4], int count,
                              Zbuffer<T> &zbuffer, Camera<T> &camera,
                              LightSource<T> const &light);

        // Subdividing a patch means subdividing
        // the individual curves of the patch:
        void subdivideCurve (Point<3, T> curve[4],
                              Point<3, T> returnVal[7]);

        Point<3, T> mPatch[4][4];
        Material<T> const *mMaterial;
        int maxlevel; // How many times we let the recursive
                      // subdivision run.
        char mShading; // type of shading
        T mSign; // sign of normals
};

// The ParametricSurface class handles calculating and
// drawing the general parametric surface (a Klein bottle in this case).
// This general surface is defined by 4 different function.
template<typename T = float>
class ParametricSurface {
    public:
        ParametricSurface (Material<T> const &material,
                            char shading, T sign);
        void drawBottom(Zbuffer<T> &zbuffer, Camera<T> &camera,
                         LightSource<T> const &light);
        void drawHandle(Zbuffer<T> &zbuffer, Camera<T> &camera,
                         LightSource<T> const &light);
        void drawTop(Zbuffer<T> &zbuffer, Camera<T> &camera,
                     LightSource<T> const &light);
        void drawMiddle(Zbuffer<T> &zbuffer, Camera<T> &camera,
                         LightSource<T> const &light);

    private:
        Point<3, T> bottom(float u, float v);
        Point<3, T> handle(float u, float v);
        Point<3, T> top(float u, float v);
        Point<3, T> middle(float u, float v);

        // The du_bottom function is the derivative of the
        // bottom function above here with respect to u.
APPENDIX B. SOURCE FILES

276  // The du_bottom function is the derivative of the bottom
277  // function above here with respect to v.
278  // The 2 vectors together define the tangent plane to the
279  // function, and hence the can be crossed
280  // to give us a normal in a given point.
281  // The rest below here follow the same pattern.
282  Vector<3, T> du_bottom(float u, float v);
283  Vector<3, T> dv_bottom(float u, float v);
284  Vector<3, T> du_handle(float u, float v);
285  Vector<3, T> dv_handle(float u, float v);
286  Vector<3, T> du_top(float u, float v);
287  Vector<3, T> dv_top(float u, float v);
288  Vector<3, T> du_middle(float u, float v);
289  Vector<3, T> dv_middle(float u, float v);

290  Material<T> const* mMaterial;
291  char mShading;  // Gouraud or Phong or ?
292  T mSign;  // We may want to invert the sign for
293  // our normals.
294  // This makes it easier to do that.
295  float deltaU;  // Step size.
296  float deltaV;  // Step size.

297  };
298
300
301
302 } #endif // GRAPHIC_WORLD_HPP

B.4 graphic_world.cpp

1 #include <stdexcept>
2 #include <cmath>  // pow
3 #include <vector>
4 #include "graphic_world.hpp"
5
6 using namespace graphic::tools;
7 using namespace graphic::world;
8 using namespace graphic::draw;
9
10 /* **************************************************** */
11 /* Polygon implementation */
12 /* **************************************************** */
13
14 // c'tor
15 template<typename T>
16 Polygon<T>::Polygon(Point<3,T> const& p0, Point<3,T> const& p1,
17  Point<3,T> const& p2, Vector<3,T> const& v0,
18  Vector<3,T> const& v1, Vector<3,T> const& v2,
19  Material<T> const& m) : mMaterial(&m){
20  mPoints[0] = p0;
21  mPoints[1] = p1;
22  mPoints[2] = p2;
23  mNormals[0] = v0;
24  mNormals[1] = v1;
25  mNormals[2] = v2;
APPENDIX B. SOURCE FILES

template<typename T>
Polygon<T>::Polygon(Polygon<T> const & p) {
  for (int i = 0; i < 3; ++i) {
    mPoints[i] = p.mPoints[i];
    mNormals[i] = p.mNormals[i];
  }
  mMaterial = p.mMaterial;
}

// point mutator
template<typename T>
Point<3,T> const& Polygon<T>::point(unsigned int i, Point<3,T> const & p) {
  if (i >= 3)
    throw std::out_of_range("Polygon point index out of range!");
  return mPoints[i] = p;
}

// point accessor
template<typename T>
Point<3,T> const Polygon<T>::point(unsigned int i) const {
  if (i >= 3)
    throw std::out_of_range("Polygon point index out of range!");
  return mPoints[i];
}

// normal vector mutator
template<typename T>
Vector<3,T> const& Polygon<T>::normal(unsigned int i, Vector<3,T> const & v) {
  if (i >= 3)
    throw std::out_of_range("Polygon normal index out of range!");
  return mNormals[i] = v;
}

// normal vector accessor
template<typename T>
Vector<3,T> const Polygon<T>::normal(unsigned int i) const {
  if (i >= 3)
    throw std::out_of_range("Polygon normal index out of range!");
  return mNormals[i];
}

// color mutator
template<typename T>
Material<T> const& Polygon<T>::material(Material<T> const & m) {
  mMaterial = &m;
  return *mMaterial;
}

// color accessor
template<typename T>
Material<T> const Polygon<T>::material() const {
  return *mMaterial;
}

}
APPENDIX B. SOURCE FILES

void Polygon<T>::projectPoints(Matrix<4,T> const& matrix){
    Vector<4,T> v;
    Point<3,T> p[3];
    for(int i = 0; i < 3; ++i) {
        v[0] = mPoints[i][0];
        v[1] = mPoints[i][1];
        v[2] = mPoints[i][2];
        v[3] = (T) 1;
        v = matrix * v;
        v = 1/v[3] * v;
        mPoints[i][0] = v[0];
        mPoints[i][1] = v[1];
        mPoints[i][2] = v[2];
    }
}

template<typename T>
void Polygon<T>::projectNormals(Matrix<4,T> const& matrix){
    Vector<4,T> v;
    for(int i = 0; i < 3; ++i) {
        v[0] = mNormals[i][0];
        v[1] = mNormals[i][1];
        v[2] = mNormals[i][2];
        v[3] = (T) 1;
        v = matrix * v;
        mNormals[i][0] = v[0];
        mNormals[i][1] = v[1];
        mNormals[i][2] = v[2];
        mNormals[i].normalize;
    }
}

// printing out data
template<typename T>
void Polygon<T>::print(std::string str = ""){
    if (str != "")
        std::cout << str << std::endl;
    for (unsigned int i = 0; i < 3; ++i){
        std::cout << "Point #" << i << ":\n";
        mPoints[i].print();
        std::cout << "Normal at that point is: \n";
        mNormals[i].print();
    }
}

// ************************************************** *
// BezierPatch implementation *
// ************************************************** *
template<typename T>
BezierPatch<T>::BezierPatch(std::vector<Point<3, T>> patch,
    Material<T> const &material, int maxlvl, char shading, int sign) {
    mMaterial = &material;
    maxlevel = maxlvl;
    mShading = shading;
}
mSign = sign;

for (int j = 0; j < 4; j++) { mPatch[0][j] = patch[j]; }
for (int j = 4; j < 8; j++) { mPatch[1][j - 4] = patch[j]; }
for (int j = 8; j < 12; j++) { mPatch[2][j - 8] = patch[j]; }
for (int j = 12; j < 16; j++) { mPatch[3][j - 12] = patch[j]; }

template<typename T>
void BezierPatch<T>::drawPatch (Zbuffer<T> &zbuffer, Camera<T> &camera, LightSource<T> const &light) {
    subdividePatch (mPatch, 0, zbuffer, camera, light);
}

template<typename T>
void BezierPatch<T>::subdividePatch (Point<3, T> patch[4][4], int count, Zbuffer<T> &zbuffer, Camera<T> &camera, LightSource<T> const &light) {
    // When count reaches maxlevel, recursion ends.
    // At that point, the corner points are the points we use
    // for drawing, and they are placed like this:
    //  
    // a d
    // o o
    // b c
    //−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
    // if (count >= maxlevel) {
        char shading = 'p';
        T sign = -1;
        // a & b collapsed.
        if (patch[0][0] == patch[1][0] && patch[0][0] == patch[2][0] && patch[0][0] == patch[3][0]) {
            Vector<3, T> a1 = vectorFromPoints(patch[0][0], patch[3][1]);
        Vector<3, T> a2 = vectorFromPoints(patch[0][0], patch[0][1]);
        Vector<3, T> c1 = vectorFromPoints(patch[3][3], patch[2][3]);
        Vector<3, T> c2 = vectorFromPoints(patch[3][3], patch[3][2]);
        Vector<3, T> d1 = vectorFromPoints(patch[0][3], patch[0][2]);
        Vector<3, T> d2 = vectorFromPoints(patch[0][3], patch[1][3]);
        Vector<3, T> Na = mSign * cross(a1, a2); Na.normalize();
        Vector<3, T> Nc = mSign * cross(c1, c2); Nc.normalize();
        Vector<3, T> Nd = mSign * cross(d1, d2); Nd.normalize();
        Polygon<T> worldpoly(patch[0][0], patch[3][3], patch[0][3],
            Na, Nc, Nd, *mMaterial);
        ScreenPolygon<T> screenpoly(worldpoly, zbuffer, camera, light);
        screenpoly.draw(mShading);
APPENDIX B. SOURCE FILES 77

}  // b & c collapsed.

  patch[3][0] == patch[3][3]) {
  Vector<3, T> a1 = vectorFromPoints(patch[0][0], patch[1][0]);
  Vector<3, T> a2 = vectorFromPoints(patch[0][0], patch[0][1]);
  Vector<3, T> b1 = vectorFromPoints(patch[3][0], patch[2][3]);
  Vector<3, T> b2 = vectorFromPoints(patch[3][0], patch[2][0]);
  Vector<3, T> d1 = vectorFromPoints(patch[0][3], patch[0][2]);
  Vector<3, T> d2 = vectorFromPoints(patch[0][3], patch[1][3]);

  Vector<3, T> Na = mSign * cross(a1, a2);  Na.normalize();
  Vector<3, T> Nb = mSign * cross(b1, b2);  Nb.normalize();
  Vector<3, T> Nd = mSign * cross(d1, d2);  Nd.normalize();

  Polygon<T> worldpoly(patch[0][0], patch[3][0], patch[0][3],
    Na, Nb, Nd, *mMaterial);
  ScreenPolygon<T> screenpoly(worldpoly, zbuffer, camera, light);
  screenpoly.draw(mShading);
}

}  // c & d collapsed.

  patch[3][3] == patch[0][3]) {
  Vector<3, T> a1 = vectorFromPoints(patch[0][0], patch[1][0]);
  Vector<3, T> a2 = vectorFromPoints(patch[0][0], patch[0][1]);
  Vector<3, T> b1 = vectorFromPoints(patch[3][0], patch[3][1]);
  Vector<3, T> b2 = vectorFromPoints(patch[3][0], patch[2][0]);
  Vector<3, T> c1 = vectorFromPoints(patch[3][3], patch[0][2]);
  Vector<3, T> c2 = vectorFromPoints(patch[3][3], patch[3][2]);

  Vector<3, T> Na = mSign * cross(a1, a2);  Na.normalize();
  Vector<3, T> Nb = mSign * cross(b1, b2);  Nb.normalize();
  Vector<3, T> Nc = mSign * cross(c1, c2);  Nc.normalize();

  Polygon<T> worldpoly(patch[0][0], patch[3][0], patch[3][3],
    Na, Nb, Nc, *mMaterial);
  ScreenPolygon<T> screenpoly(worldpoly, zbuffer, camera, light);
  screenpoly.draw(mShading);
}

}  // d & a collapsed.

else if (patch[0][3] == patch[0][2] & patch[0][3] == patch[0][1] &
  patch[0][3] == patch[0][0]) {

  Vector<3, T> a1 = vectorFromPoints(patch[0][0], patch[1][0]);
  Vector<3, T> a2 = vectorFromPoints(patch[0][0], patch[1][3]);
  Vector<3, T> b1 = vectorFromPoints(patch[3][0], patch[3][1]);
  Vector<3, T> b2 = vectorFromPoints(patch[3][0], patch[2][0]);
  Vector<3, T> c1 = vectorFromPoints(patch[3][3], patch[2][3]);
  Vector<3, T> c2 = vectorFromPoints(patch[3][3], patch[3][2]);

  Vector<3, T> Na = mSign * cross(a1, a2);  Na.normalize();
  Vector<3, T> Nb = mSign * cross(b1, b2);  Nb.normalize();
  Vector<3, T> Nc = mSign * cross(c1, c2);  Nc.normalize();

  Polygon<T> worldpoly(patch[0][0], patch[3][0], patch[3][3],
    Na, Nb, Nc, *mMaterial);
  ScreenPolygon<T> screenpoly(worldpoly, zbuffer, camera, light);
  screenpoly.draw(mShading);
// Here we assume all 4 corner points are distinct.

else {

    Vector<3, T> a1 = vectorFromPoints(patch[0][0], patch[1][0]);
    Vector<3, T> a2 = vectorFromPoints(patch[0][0], patch[0][1]);
    Vector<3, T> b1 = vectorFromPoints(patch[3][0], patch[3][1]);
    Vector<3, T> b2 = vectorFromPoints(patch[3][0], patch[2][0]);
    Vector<3, T> c1 = vectorFromPoints(patch[3][3], patch[2][3]);
    Vector<3, T> c2 = vectorFromPoints(patch[3][3], patch[3][2]);
    Vector<3, T> d1 = vectorFromPoints(patch[0][3], patch[0][2]);
    Vector<3, T> d2 = vectorFromPoints(patch[0][3], patch[1][3]);

    Vector<3, T> Na = mSign*cross(a1, a2); Na.normalize();
    Vector<3, T> Nb = mSign*cross(b1, b2); Nb.normalize();
    Vector<3, T> Nc = mSign*cross(c1, c2); Nc.normalize();
    Vector<3, T> Nd = mSign*cross(d1, d2); Nd.normalize();

    Polygon<T> worldpoly1(patch[0][0], patch[3][0], patch[0][3],
                          Na, Nb, Nd, *mMaterial);
    ScreenPolygon<T> screenpoly1(worldpoly1, zbuffer, camera, light);
    screenpoly1.draw(mShading);

    Polygon<T> worldpoly2(patch[3][0], patch[3][3], patch[0][3],
                          Nb, Nc, Nd, *mMaterial);
    ScreenPolygon<T> screenpoly2(worldpoly2, zbuffer, camera, light);
    screenpoly2.draw(mShading);
}

else {

    Point<3, T> left[4][4];
    Point<3, T> right[4][4];
    Point<3, T> topleft[4][4];
    Point<3, T> topright[4][4];
    Point<3, T> botleft[4][4];
    Point<3, T> botright[4][4];
    Point<3, T> thiscurve[4];

    // Subdivide patch in left and right.
    for (int i = 0; i < 4; i++) {

        thiscurve[0] = patch[i][0];  // The
        thiscurve[1] = patch[i][1];  // curve
        thiscurve[2] = patch[i][2];  // we

        Point<3, T> newcurves1[7];

        // The 2 new curves after dividing are in newcurves:
        subdivideCurve(thiscurve, newcurves1);

        left[i][0] = newcurves1[0];
        left[i][1] = newcurves1[1];
        left[i][2] = newcurves1[2];
        left[i][3] = newcurves1[3];
        right[i][0] = newcurves1[3];
        right[i][1] = newcurves1[4];
    }
}
APPENDIX B. SOURCE FILES

right[i][2] = newcurves1[5];
right[i][3] = newcurves1[6];
}

// Now both left and right must each be subdivided again,
// into top and bottom.
for (int i = 0; i < 4; i++) {
  thiscurve[0] = left[0][i];
  thiscurve[1] = left[1][i];
  thiscurve[2] = left[2][i];
  thiscurve[3] = left[3][i]; // The curve we divide.
  Point<3, T> newcurves2[7];
  // The 2 new curves after dividing are in newcurves:
  subdivideCurve(thiscurve, newcurves2);
  topleft[0][i] = newcurves2[0];
  topleft[1][i] = newcurves2[1];
  topleft[2][i] = newcurves2[2];
  topleft[3][i] = newcurves2[3];
  botleft[0][i] = newcurves2[3];
  botleft[1][i] = newcurves2[4];
  botleft[2][i] = newcurves2[5];
  botleft[3][i] = newcurves2[6];
  thiscurve[0] = right[0][i];
  thiscurve[1] = right[1][i];
  thiscurve[2] = right[2][i];
  thiscurve[3] = right[3][i]; // The curve we divide.
  Point<3, T> newcurves3[7];
  // The 2 new curves after dividing are in newcurves:
  subdivideCurve(thiscurve, newcurves3);
  topright[0][i] = newcurves3[0];
  topright[1][i] = newcurves3[1];
  topright[2][i] = newcurves3[2];
  topright[3][i] = newcurves3[3];
  botright[0][i] = newcurves3[3];
  botright[1][i] = newcurves3[4];
  botright[2][i] = newcurves3[5];
  botright[3][i] = newcurves3[6];
}

// Here the 4 recursive calls:
subdividePatch(topleft, count+1, zbuffer, camera, light);
subdividePatch(botleft, count+1, zbuffer, camera, light);
subdividePatch(topright, count+1, zbuffer, camera, light);
subdividePatch(botright, count+1, zbuffer, camera, light);

} } } } } 

template<typename T>
void BezierPatch<T>::subdivideCurve(Point<3, T> curve[4],
Point<3, T> returnVal[7]) {

}
\[ T_{Hx} = \frac{\text{curve}[1][0] + \text{curve}[2][0]}{2}; \]
\[ T_{Hy} = \frac{\text{curve}[1][1] + \text{curve}[2][1]}{2}; \]
\[ T_{Hz} = \frac{\text{curve}[1][2] + \text{curve}[2][2]}{2}; \]

// Left subdivision here:

// X-values.
returnVal[0][0] = curve[0][0];
returnVal[1][0] = \( \frac{\text{curve}[0][0] + \text{curve}[1][0]}{2} \);
returnVal[2][0] = \( \frac{T_{Hx} + \text{returnVal}[1][0]}{2} \);

// Y-values.
returnVal[0][1] = curve[0][1];
returnVal[1][1] = \( \frac{\text{curve}[0][1] + \text{curve}[1][1]}{2} \);
returnVal[2][1] = \( \frac{\text{Hy} + \text{returnVal}[1][1]}{2} \);

// Z-values.
returnVal[0][2] = curve[0][2];
returnVal[1][2] = \( \frac{\text{curve}[0][2] + \text{curve}[1][2]}{2} \);
returnVal[2][2] = \( \frac{T_{Hz} + \text{returnVal}[1][2]}{2} \);

// Right subdivision here:

// X-values.
returnVal[5][0] = \( \frac{\text{curve}[2][0] + \text{curve}[3][0]}{2} \);
returnVal[4][0] = \( \frac{T_{Hx} + \text{returnVal}[5][0]}{2} \);
returnVal[3][0] = \( \frac{\text{returnVal}[2][0] + \text{returnVal}[4][0]}{2} \);
returnVal[6][0] = curve[3][0];

// Y-values.
returnVal[5][1] = \( \frac{\text{curve}[2][1] + \text{curve}[3][1]}{2} \);
returnVal[4][1] = \( \frac{\text{Hy} + \text{returnVal}[5][1]}{2} \);
returnVal[3][1] = \( \frac{\text{returnVal}[2][1] + \text{returnVal}[4][1]}{2} \);
returnVal[6][1] = curve[3][1];

// Z-values.
returnVal[5][2] = \( \frac{\text{curve}[2][2] + \text{curve}[3][2]}{2} \);
returnVal[4][2] = \( \frac{T_{Hz} + \text{returnVal}[5][2]}{2} \);
returnVal[3][2] = \( \frac{\text{returnVal}[2][2] + \text{returnVal}[4][2]}{2} \);
returnVal[6][2] = curve[3][2];

}
template<typename T>
void ParametricSurface<T>::drawBottom(Zbuffer<T> & zbuffer, Camera<T> & camera, LightSource<T> const & light) {
  for (float u = 0.0; u < 2*PI; u += deltaU) {
    for (float v = 0.0; v < PI; v += deltaV) {
      Point<3, T> p1 = bottom(u, v);
      Point<3, T> p2 = bottom(u+deltaU, v);
      Point<3, T> p3 = bottom(u+deltaU, v+deltaV);
      Point<3, T> p4 = bottom(u, v+deltaV);

      Vector<3, T> N1 = -1*mSign*cross(du_bottom(u, v), dv_bottom(u, v));
      N1.normalize();
      Vector<3, T> N2 = -1*mSign*cross(du_bottom(u+deltaU, v), dv_bottom(u+deltaU, v));
      N2.normalize();
      Vector<3, T> N3 = -1*mSign*cross(du_bottom(u+deltaU, v+deltaV), dv_bottom(u+deltaU, v+deltaV));
      N3.normalize();
      Vector<3, T> N4 = -1*mSign*cross(du_bottom(u, v+deltaV), dv_bottom(u, v+deltaV));
      N4.normalize();

      Polygon<T> worldpoly1(p1, p3, p4, N1, N3, N4, *mMaterial);
      ScreenPolygon<T> screenpoly1(worldpoly1, zbuffer, camera, light);
      screenpoly1.draw(mShading);

      Polygon<T> worldpoly2(p1, p2, p3, N1, N2, N3, *mMaterial);
      ScreenPolygon<T> screenpoly2(worldpoly2, zbuffer, camera, light);
      screenpoly2.draw(mShading);
    }
  }
}

// END OF FILE
APPENDIX B. SOURCE FILES

N1. normalize();
Vector<3, T> N2 = -1*mSign*cross(du_handle(u+deltaU, v),
dv_handle(u+deltaU, v));
N2. normalize();
Vector<3, T> N3 = -1*mSign*cross(du_handle(u+deltaU, v+deltaV),
dv_handle(u+deltaU, v+deltaV));
N3. normalize();
Vector<3, T> N4 = -1*mSign*cross(du_handle(u, v+deltaV),
dv_handle(u, v+deltaV));
N4. normalize();

Polygon<T> worldpoly1(p1, p3, p4, N1, N3, N4, *mMaterial);
ScreenPolygon<T> screenpoly1(worldpoly1, zbuffer, camera, light);
screenpoly1.draw(mShading);

Polygon<T> worldpoly2(p1, p2, p3, N1, N2, N3, *mMaterial);
ScreenPolygon<T> screenpoly2(worldpoly2, zbuffer, camera, light);
screenpoly2.draw(mShading);

// std::cout << ” u: ” << u << ”, v: ” << v << std::endl;
}
}
}
#endif

template<typename T>
void ParametricSurface<T>::drawTop(Zbuffer<T> & zbuffer, Camera<T> & camera,
LightSource<T> const & light) {
    for (float u = 0.0; u < 2*PI; u += deltaU) {
        for (float v = 0.0; v < PI; v += deltaV) {
            Point<3, T> p1 = top(u, v);
            Point<3, T> p2 = top(u+deltaU, v);
            Point<3, T> p3 = top(u+deltaU, v+deltaV);
            Point<3, T> p4 = top(u, v+deltaV);
            Vector<3, T> N1 = mSign*cross(du_top(u, v),
dv_top(u, v));
            N1. normalize();
            Vector<3, T> N2 = mSign*cross(du_top(u+deltaU, v),
dv_top(u+deltaU, v));
            N2. normalize();
            Vector<3, T> N3 = mSign*cross(du_top(u+deltaU, v+deltaV),
dv_top(u+deltaU, v+deltaV));
            N3. normalize();
            Vector<3, T> N4 = mSign*cross(du_top(u, v+deltaV),
dv_top(u, v+deltaV));
            N4. normalize();
            Polygon<T> worldpoly1(p1, p3, p4, N1, N3, N4, *mMaterial);
            ScreenPolygon<T> screenpoly1(worldpoly1, zbuffer, camera, light);
screenpoly1.draw(mShading);
APPENDIX B. SOURCE FILES

539 Polygon<T> worldpoly2(p1, p2, p3, N1, N2, N3, *mMaterial);
540 ScreenPolygon<T> screenpoly2(worldpoly2, zbuffer, camera, light);
541 screenpoly2.draw(mShading);
542 // std::cout << " u: " << u << " v: " << v << std::endl;
543 }
544 }
545 }
546 }
547 }
548 template<typename T>
549 void ParametricSurface<T>::drawMiddle(Zbuffer<T> & zbuffer, Camera<T> & camera, LightSource<T> const & light) {
550 for (float u = 0.0; u < 2*PI; u += deltaU) {
551 for (float v = 0.0; v < PI; v += deltaV) {
552 Point<3, T> p1 = middle(u, v);
553 Point<3, T> p2 = middle(u+deltaU, v);
554 Point<3, T> p3 = middle(u+deltaU, v+deltaV);
555 Point<3, T> p4 = middle(u, v+deltaV);
556 Vector<3, T> N1 = mSign*cross(du_middle(u, v), dv_middle(u, v));
557 N1.normalize();
558 Vector<3, T> N2 = mSign*cross(du_middle(u+deltaU, v), dv_middle(u+deltaU, v));
559 N2.normalize();
560 Vector<3, T> N3 = mSign*cross(du_middle(u+deltaU, v+deltaV), dv_middle(u+deltaU, v+deltaV));
561 N3.normalize();
562 Vector<3, T> N4 = mSign*cross(du_middle(u, v+deltaV), dv_middle(u, v+deltaV));
563 N4.normalize();
564 Polygon<T> worldpoly1(p1, p3, p4, N1, N3, N4, *mMaterial);
565 ScreenPolygon<T> screenpoly1(worldpoly1, zbuffer, camera, light);
566 screenpoly1.draw(mShading);
567 Polygon<T> worldpoly2(p1, p2, p3, N1, N2, N3, *mMaterial);
568 ScreenPolygon<T> screenpoly2(worldpoly2, zbuffer, camera, light);
569 screenpoly2.draw(mShading);
570 }
571 }
572 template<typename T>
573 Point<3, T> ParametricSurface<T>::bottom(float u, float v) {
574 Point<3, T> point;
575 point[0] = (2.5 + 1.5*cos(v))*cos(u);
576 point[1] = (2.5 + 1.5*cos(v))*sin(u);
577 point[2] = -2.5*sin(v);
template<typename T>
Point<3, T> ParametricSurface<T>::handle(float u, float v) {
    Point<3, T> point;
    point[0] = 2.0 - 2.0*cos(v) + sin(u);
    point[1] = cos(u);
    point[2] = 3.0*v;
    return point;
}

template<typename T>
Point<3, T> ParametricSurface<T>::top(float u, float v) {
    Point<3, T> point;
    point[0] = 2.0 + (2.0 + cos(u))*cos(v);
    point[1] = sin(u);
    point[2] = 3.0*PI + (2.0 + cos(u))*sin(v);
    return point;
}

template<typename T>
Point<3, T> ParametricSurface<T>::middle(float u, float v) {
    Point<3, T> point;
    point[0] = (2.5 + 1.5*cos(v))*cos(u);
    point[1] = (2.5 + 1.5*cos(v))*sin(u);
    point[2] = 3*v;
    return point;
}

// Here are the 8 derivatives of the above 4 functions:
// duXXX is derivative with respect to u.
// dvXXX is derivative with respect to v.

template<typename T>
Vector<3, T> ParametricSurface<T>::du_bottom(float u, float v) {
    Vector<3, T> tangent;
    tangent[0] = -0.5*sin(u)*(3.0*cos(v) + 5);
    tangent[1] = cos(u)*(1.5*cos(v) + 2.5);
    tangent[2] = 0.0;
    return tangent;
}

template<typename T>
Vector<3, T> ParametricSurface<T>::dv_bottom(float u, float v) {
    Vector<3, T> tangent;
    tangent[0] = -1.5*sin(v)*cos(u);
    tangent[1] = -1.5*sin(v)*sin(u);
tangent[2] = -2.5*cos(v);
return tangent;
}
template<typename T>
Vector<3, T> ParametricSurface<T>::du_handle(float u, float v) {
Vector<3, T> tangent;
tangent[0] = cos(u);
tangent[1] = -sin(u);
tangent[2] = 0.0;
return tangent;
}
template<typename T>
Vector<3, T> ParametricSurface<T>::dv_handle(float u, float v) {
Vector<3, T> tangent;
tangent[0] = 2.0*sin(v);
tangent[1] = 0.0;
tangent[2] = 3.0;
return tangent;
}
template<typename T>
Vector<3, T> ParametricSurface<T>::du_top(float u, float v) {
Vector<3, T> tangent;
tangent[0] = -sin(u)*cos(v);
tangent[1] = cos(u);
tangent[2] = -sin(u)*sin(v);
return tangent;
}
template<typename T>
Vector<3, T> ParametricSurface<T>::dv_top(float u, float v) {
Vector<3, T> tangent;
tangent[0] = -sin(v)*(cos(u) + 2.0);
tangent[1] = 0.0;
tangent[2] = cos(v)*(cos(u) + 2.0);
return tangent;
}
template<typename T>
Vector<3, T> ParametricSurface<T>::du_middle(float u, float v) {
Vector<3, T> tangent;
tangent[0] = -0.5*sin(u)*(3.0*cos(v) + 5.0);
tangent[1] = cos(u)*(1.5*cos(v) + 2.5);
tangent[2] = 0.0;
APPENDIX B. SOURCE FILES

```cpp
return tangent;
}

template<typename T>
Vector<3, T> ParametricSurface<T>::dv_middle(float u, float v) {
  Vector<3, T> tangent;
  tangent[0] = -1.5*sin(v)*cos(u);
  tangent[1] = -1.5*sin(v)*sin(u);
  tangent[2] = 3.0;
  return tangent;
}
```
APPENDIX B. SOURCE FILES

767 w1 = pW1;
768 w2 = pW2;
769 B = backClippingPlane;
770 F = frontClippingPlane;
771 update();
772 }
773
774 // This method updates CW,DOP, u, v, n
775 // It is used to secure that these values are always correct
776 template<typename T>
777 void Camera<T>::update()
778 {
779    // update center of window
780    CW[0] = (w1[0] + w2[0]) / 2;
781    CW[1] = (w1[1] + w2[1]) / 2;
782    CW[2] = 0;
783
784    // update DOP
785    DOP[0] = PRP[0] − CW[0];
786    DOP[1] = PRP[1] − CW[1];
788
789    // update n
790    n = VPN;
791    n.normalize();
792
793    // update u
794    u = cross(VUP,VPN);
795    u.normalize();
796
797    // update v
798    Vector<3,T> tmp = cross(VUP,VPN);
799    v = cross(VPN, tmp);
800    v.normalize();
801
802    // calculate WorldPRP
803    // This is done by making the rotation matrix R as specified
804    T valR[4][4] = {{u[0], u[1], u[2], 0}, {v[0], v[1], v[2], 0},
805                  {n[0], n[1], n[2], 0}, {0, 0, 0, 1}};
806    Matrix<4,T> R(valR);
807    T valPRP[4] = {PRP[0], PRP[1], PRP[2], 1};
808    Vector<4,T> wPRP(valPRP);
809    wPRP = R * wPRP;
812    worldPRP.set(valWPRP);
813    projMatrixIsValid = false;
814 }
815
816 // sets VRP to the value specified
817 template<typename T>
818 void Camera<T>::setVRP(Point<3,T> newVRP)
819 {
820    VRP = newVRP;
821    this−>update();
822 }
823
// sets PRP to the value specified
template<typename T>
void Camera<T>::setPRP(Point<3,T> newPRP)
{
    PRP = newPRP;
    this->update();
}

// sets VPN to the value specified
template<typename T>
void Camera<T>::setVPN(Vector<3,T> newVPN)
{
    VPN = newVPN;
    this->update();
}

// sets VUP to the value specified
template<typename T>
void Camera<T>::setVUP(Vector<3,T> newVUP)
{
    VUP = newVUP;
    this->update();
}

// sets w1 to the value specified
template<typename T>
void Camera<T>::setW1(Point<3,T> newW1)
{
    w1 = newW1;
    this->update();
}

// sets w2 to the value specified
template<typename T>
void Camera<T>::setW2(Point<3,T> newW2)
{
    w2 = newW2;
    this->update();
}

// this method calculates the normalization matrix
template<typename T>
Matrix<4,T> Camera<T>::calculateNormalization()
{
    // First, we make sure all values are updated
    this->update();
    // Second, we make the Normalization matrix and
    // initialize it to identity
    T identity[4][4] = {{1,0,0,0}, {0,1,0,0}, {0,0,1,0}, {0,0,0,1}};
    Matrix<4,T> N(identity);
    // step 1: Translate VRP to the origin
    T valTVRP[4][4] = {{1,0,0,-1*VRP[0]} , {0,1,0,-1*VRP[1]} ,
    {0,0,1,-1*VRP[2]} , {0,0,0,1} };
    Matrix<4,T> TVRP(valTVRP);
    //TVRP.print("TVRP: ");
APPENDIX B. SOURCE FILES

881 // step 2: Rotate
882 // This is done by making the rotation matrix R as specified
883 T valR[4][4] = {{u[0],u[1],u[2],0}, {v[0],v[1],v[2],0},
884 {n[0],n[1],n[2],0}, {0,0,0,1}};
885 Matrix<4,T> R(valR);
886 //R.print("R: ");
887
888 // step 3: Translate PRP
889 T valTPRP[4][4] = {{1,0,0,-1*PRP[0]},{0,1,0,-1*PRP[1]},
890 {0,0,1,-1*PRP[2]},{0,0,0,1}};
891 Matrix<4,T> TPRP(valTPRP);
892 //TPRP.print("TPRP: ");
893
894 // step 4: Shear
895 T valSh[4][4] = {{1,0,−DOP[0]/DOP[2],0},{0,1,−DOP[1]/DOP[2],0},
896 {0,0,1,0},{0,0,0,1}};
897 Matrix<4,T> Sh(valSh);
898 //Sh.print("Sh: ");
899
900 // step 5: Scale
901 T sx = (2*PRP[2]) / (w2[0]−w1[0]);
902 T sy = (2*PRP[2]) / (w2[1]−w1[1]);
903 T sall = (−1) / (B−PRP[2]);
904 T valSc[4][4] = {{sx*sall,0,0,0},{0,sy*sall,0,0},
905 {0,0,sall,0},{0,0,0,1}};
906 Matrix<4,T> Sc(valSc);
907 //Sc.print("Sc: ");
908
909 //Matrix<4,T> m1 = R * TVRP;
910 //m1.print("R * TVRP: ");
911 //Matrix<4,T> m2 = TPRP * R * TVRP;
912 //m2.print("TPRP * R * TVRP: ");
913 //Matrix<4,T> m3 = Sh * TPRP * R * TVRP;
914 //m3.print("Sh * TPRP * R * TVRP: ");
915 //Matrix<4,T> m4 = Sc * Sh * TPRP * R * TVRP;
916 //m4.print("Sc * Sh * TPRP * R * TVRP: ");
917
918 // calculating N
919 N = Sc * Sh * TPRP * R * TVRP;
920 //N.print("N: ");
921 return N;
922 }
923
924 // calculates the projection matrix
template<typename T>
925 Matrix<4,T> Camera<T>::calculateProjection()
926 {
927 // first, make sure all values are in a legal state
928 this->update();
929
930 // calculate d, done by finding N, and letting it work on CW
931 T arVRP[4] = { VRP[0], VRP[1], VRP[2], 1 };
932 Vector<4,T> vecVRP(arVRP);
933 Matrix<4,T> N = this->calculateNormalization();
934 vecVRP = N * vecVRP;
935 T d = vecVRP[2];
// std::cout << "d is: " << d << std::endl;

// now, we make the projection matrix
T valM[4][4] = {{1, 0, 0, 0}, {0, 1, 0, 0}, {0, 0, 1, 0}, {0, 0, 1/d, 0}};
Matrix<4,T> M(valM);

//M.print("M: ");
Matrix<4,T> P = M * N;
return P;

// this method calculates the transformation matrix from canonical
// perspective view volume into canonical parallel view volume
template<typename T>
Matrix<4,T> Camera<T>::calculateParPalConv()
{
    // first, make sure all values are in a legal state
    this->update();

    // now, calculate zmax
    T zmax = -1 * (F - PRP[2])/(B - PRP[2]);
    T valC[4][4] = {{1, 0, 0, 0}, {0, 1, 0, 0}, {0, 0, 1/(1+zmax), -zmax/(1+zmax)},
                    {0, 0, -1, 0}};
    Matrix<4,T> C(valC);
    //C.print("C: ");

    return C;
}

// this method calculates the mapping matrix from projected points into
// screen points it assumes usage of DotDevice
template<typename T>
Matrix<4,T> Camera<T>::calculateMapping(int width, int heigth)
{
    // first, make sure all values are in a legal state
    this->update();

    // second, the translation matrix
    T valT1[4][4] = {{1, 0, 0, 1}, {0, 1, 0, 1}, {0, 0, 1, 1}, {0, 0, 0, 1}};
    Matrix<4,T> T1(valT1);

    // third, the scale matrix
    T scx = width/2;
    T scy = heigth/2;
    T scz = 1; // we’re only interested in relative z anyway
    T valSc[4][4] = {{scx, 0, 0, 0}, {0, scy, 0, 0}, {0, 0, 1, 0}, {0, 0, 0, 1}};
    Matrix<4,T> Sc(valSc);

    // fourth, combine to calculate the mapping matrix MAP
    Matrix<4,T> MAP = Sc * T1;
    //MAP.print("MAP: ");

    return MAP;
// this method calculates the total projection matrix
// that is from world points into screen coordinates

// first, make sure all values are in a legal state
this->update();

// calculate the matrices
Matrix<4,T> N = this->calculateNormalization();
Matrix<4,T> C = this->calculateParPalConv();
Matrix<4,T> MAP = this->calculateMapping(width, heigth);

// calculate the projection matrix
Matrix<4,T> P = MAP * C * N;

// if there is a correct projection matrix, this will be returned,
// in not, calculate a new one

Matrix<4,T> Camera<T>::getProjectionMatrix(int width, int heigth) {
    if(!projMatrixIsValid)
        {
            ProjMatr = this->calculateTotalProjection(width, heigth);
            projMatrixIsValid = true;
        }
    return ProjMatr;
}

// prints the worldPRP
Point<3,T> Camera<T>::getWorldPRP() const
    { return worldPRP; }

// prints all the fields in Camera to the screen
void Camera<T>::print() {
    VRP.print("VRP: ");
    PRP.print("PRP: ");
    worldPRP.print("worldPRP: ");
    VPN.print("VPN: ");
    VUP.print("VUP: ");
    w1.print("w1: ");
    w2.print("w2: ");
    CW.print("CW: ");
    DOP.print("DOP: ");
    u.print("u: ");
    v.print("v: ");
    n.print("n: ");
}
template<typename T>
void Camera<T>::print(string str)
{
    cout << str << endl;
}

// Prints str to screen and then all the fields in Camera to the screen

template<typename T>
void Camera<T>::print(const string str)
{
    cout << str << endl;
    this->print();
}

// ***************************************************** */
// */ Material template implementation */
// ***************************************************** */

// constructor

template<typename T>
Material<T>::Material(Color mDColor, Color mSColor, T mKa,
        T mKd, T mKs, int mN)
{
    diffuseColor = mDColor;
    specularColor = mSColor;
    // ka
    if ( 1 >= mKa && mKa >= 0)
        ka = mKa;
    else{
        ka = 0.1;
        cout << "WARNING: k\alpha set to illegal value. Set to 0.1" << endl;
    }
    // kd
    if ( 1 >= mKd && mKd >= 0)
        kd = mKd;
    else{
        kd = 0.1;
        cout << "WARNING: k\delta set to illegal value. Set to 0.1" << endl;
    }
    // ks
    if ( 1 >= mKs && mKs >= 0)
        ks = mKs;
    else{
        ks = 0.1;
        cout << "WARNING: k\sigma set to illegal value. Set to 0.1" << endl;
    }
    n = mN;
}

// returns red constants

template<typename T>
Color Material<T>::getDiffuseColor() const
{
    return diffuseColor;
}

// returns green constants

template<typename T>
Color Material<T>::getSpecularColor() const
{
    return specularColor;
}

// returns ka
template<typename T>
T Material<T>::getKa() const { return ka; }

// returns kd
template<typename T>
T Material<T>::getKd() const { return kd; }

// returns ks
template<typename T>
T Material<T>::getKs() const { return ks; }

// returns n
template<typename T>
int Material<T>::getN() const { return n; }

void Material<T>::print()
{
    std::cout << "Diffuse Color: " ;
    diffuseColor.print();
    std::cout << "Specular Color: " ;
    specularColor.print();
    std::cout << "ka: " << ka << " , kd: " << kd << " , ks: "
               << ks << " , n: " << n << std::endl;
}

void Material<T>::print(std::string str)
{
    std::cout << str << std::endl;
    print();
}

/* *********************************************** */
/* LightSource template implementation */
/* *********************************************** */

// constructor
template<typename T>
LightSource<T>::LightSource(const Point<3,T>& mCoordinates,
                             const Point<3,T>& mAttenuation,
                             const Color& mAmbient,
                             const Color& mIntensities)
{
    coordinates = mCoordinates;
    attenuation = mAttenuation;
    ambient = mAmbient;
appendix b. source files

1166  intensities = mIntensities;
1167 }
1168
1169 template< typename T>
1170 LightSource<T>::LightSource(Point<3,T> const& position,
1171 T c1 = 0, T c2 = 0.1, T c3 = 0)
1172 Point<3,T> p;
1173 p[0] = c1;
1174 p[1] = c2;
1175 p[2] = c3;
1176 coordinates = position;
1177 attenuation = p;
1178 ambient = grey;
1179 intensities = white;
1180 }
1181
1182 // returns the coordinates of the light source
1183 template< typename T>
1184 Point<3,T> const & LightSource<T>::getCoordinates() const
1185 {
1186  return coordinates;
1187 }
1188
1189 // returns the coefficients of the attenuation function
1190 template< typename T>
1191 Point<3,T> const & LightSource<T>::getAttenuation() const
1192 {
1193  return attenuation;
1194 }
1195
1196 // returns attenuation factor, given the distance from the light source
1197 template< typename T>
1198 T LightSource<T>::getAttenuationFactor(T distance) const
1199 {
1200  T fatt = 1 / (attenuation[0] + attenuation[1]*distance +
1201            attenuation[2]*distance*distance);
1202  if( fatt > 1)
1203    return 1;
1204  else
1205    return fatt;
1206 }
1207
1208 // returns the level of ambient light
1209 template< typename T>
1210 Color const & LightSource<T>::getAmbient() const
1211 {
1212  return ambient;
1213 }
1214
1215 // returns the intensities of the light source
1216 template< typename T>
1217 Color const & LightSource<T>::getIntensities() const
1218 {
1219  return intensities;
1220 }
1221 // calculates illumination at given point
1222 // note that normal is expected to be normalized
template<typename T>
Color LightSource<typename T>::PhongIllumination(const Material<typename T>& mMateri,
const Point<typename T>& mPoint, const Vector<typename T>& N,
const Camera<typename T>& myCamera) const
{
    // for each of the three colors, we will have to build the illumination I
    // to lessen the number of calculations, we start by calculating a few
    // constants making normal non-const NB: A hack is taking place
    Point<typename T> worldPRP = myCamera.getWorldPRP();
    //worldPRP.print("WorldPRP: ");
    // calculating the vector L and distance d
    Vector<typename T> L = vectorFromPoints(mPoint, coordinates);
    T distance = L.length();
    L.normalize();
    //L.print("L: ");
    //N.print("N: ");
    // calculating N*L, must be positive or zero
    T NdotL = N*L;
    if( NdotL < 0 )
    {
        // std::cout << "NdotL set to 0, were " << NdotL << std::endl;
        NdotL = 0;
    }
    // calculating V, vector from mPoint to viewpoint
    Vector<typename T> V = vectorFromPoints(mPoint, worldPRP);
    V.normalize();
    //V.print("V: ");
    // calculating R*V, must be positive or zero
    T RdotV = 0;
    // only calculate RdotV if lightsource is on the same side of
    // surface as viewer
    if( NdotL > 0.0001 )
    {
        RdotV = ((2*NdotL) * N - L) * V;
        if( RdotV < 0 )
        {
            RdotV = 0;
        }
    }
    T RdotVn = pow(RdotV, mMateri.getN());
    // the attenuation factor
    T fatt = this->getAttenuationFactor(distance);
    //N.print("N: ");
    //L.print("L: ");
    /* std::cout << "N*L = " " NdotL << ", R*V = " " RdotV
    << " , RdotV = " " distance << " , fatt = 
    << fatt << std::endl; */
    // red
    T I AmbientRed = ambient.getRed() * mMateri.getKa() *
    mMateri.getDiffuseColor().getRed();
    T IDiffuseRed = intensities.getRed() * mMateri.getKd() *
    mMateri.getDiffuseColor().getRed() * NdotL;
    T ISpecularRed = intensities.getRed() * mMateri.getKs() *
    mMateri.getSpecularColor().getRed() * RdotVn;
    /* std::cout << "Iar = " " IAmbientRed " IDiffuseRed; */
// green
T IAmbientGreen = ambient.getGreen() * mMaterial.getKa() *
mMaterial.getDiffuseColor().getGreen();
T IDiffuseGreen = intensities.getGreen() * mMaterial.getKd() *
    mMaterial.getDiffuseColor().getGreen() * NdotL;
T ISpecularGreen = intensities.getGreen() * mMaterial.getKs() *
    mMaterial.getSpecularColor().getGreen() * RdotVN;
/* std::cout << "Iag = " << IAmbientGreen << ", Idg = " << IDiffuseGreen << ", Isg = " << ISpecularGreen
   << std::endl; */

// blue
T IAmbientBlue = ambient.getBlue() * mMaterial.getKa() *
mMaterial.getDiffuseColor().getBlue();
T IDiffuseBlue = intensities.getBlue() * mMaterial.getKd() *
    mMaterial.getDiffuseColor().getBlue() * NdotL;
T ISpecularBlue = intensities.getBlue() * mMaterial.getKs() *
    mMaterial.getSpecularColor().getBlue() * RdotVN;
/* std::cout << "Iab = " << IAmbientBlue << ", Idb = " << IDiffuseBlue << ", Isb = " << ISpecularBlue << std::endl; */

// a dummy for now
Color phongIIColor(0, 0, 0);
phongIIColor.setRed(IAmbientRed + fatt * (IDiffuseRed + ISpecularRed));
phongIIColor.setGreen(IAmbientGreen + fatt * (IDiffuseGreen + ISpecularGreen));
phongIIColor.setBlue(IAmbientBlue + fatt * (IDiffuseBlue + ISpecularBlue));
return phongIIColor;

// prints light source fields to screen
template<typename T>
void LightSource<T>::print() const
{
    std::cout << "Coordinates:\n";
    coordinates.print();
    std::cout << "Attenuation constants:\n";
    attenuation.print();
    std::cout << "Ambient light:\n";
    ambient.print();
    std::cout << "Color(intensity) of light:\n";
    intensities.print();
}

// prints str to screen and then light source fields to screen
template<typename T>
void LightSource<T>::print(std::string str) const
{
    std::cout << str << std::endl;
    print();
B.5 graphic_draw.hpp

```cpp
// ****************
'Draw' classes
****************/
#ifndef _GRAPHIC_DRAW_HPP
#define _GRAPHIC_DRAW_HPP

#include <vector>
#include <string>
#include "DotDevice/DotDevice.h"

using namespace graphic::world;

namespace graphic {
  namespace draw {
    class ScreenLine {
      protected:
        Point<2, int> mStartPoint;
        Color mStartColor;
        Point<2, int> mEndPoint;
        Color mEndColor;
      public:
        ScreenLine(); // default ctor
        ScreenLine(Point<2, int> const & startPoint,
                    Point<2, int> const & endPoint);
        Point<2, int> const & setStartPoint(
                        Point<2, int> const & newStartPoint);
        Point<2, int> const & setEndPoint(
                        Point<2, int> const & newEndPoint);
        // sets the start and end point to newColor:
        Color const & setColor(Color const & newColor);
        void setColor(Color const & newStartColor,
                      Color const & newEndColor);
        Point<2, int> const & getStartPoint() const;
        Point<2, int> const & getEndPoint() const;
        // draws the line to the screen:
        void drawMonocolor(DotDevice* device) const;
    };
  }
}
```

// The ZBuffer class implements our z-buffer (not surprisingly).
// It simply contains an array of ColorPoints, i.e. a z-coord
// and a color.

```
class Zbuffer {
public:
    Zbuffer();
    Zbuffer(DotDevice& device, Color const & color = grey);

    // If z is closer then write new z and new color.
    void setPixel(int x, int y, int z, Color const & c);

    // Checks if z is closer. As above, but without writing
    // to the buffer.
    bool checkPixel(int x, int y, int z);

    // When done with all objects, this will draw the z-buffer
    // to screen.
    void drawBuffer();

    int height;
    int width;

private:
    DotDevice* mDevice;

    struct ColorPoint {
        T zcoord;
        Color clr;
    };

    // The actual z-buffer.
    std::vector<std::vector<ColorPoint>> mBuffer;
};

template<typename T>
class ScreenPolygon {
protected:
    // struct containing projected polygon end point rounded
    // to integers for use on screen, reference to z-coord
    // from projected end point, reference to
    // point and normal in original polygon.
    struct Vertexdata {
        int x;
        int y;
        T z;
        Point<3,T> const * p;
        Vector<3,T> const * n;
        // Vertexdata(Point<3,T> const & p, Vector<3,T> const & n)
        // : p(& p), n(& n) }
        } v0, v1, v2;
    Camera<T> * mCamera;
    Zbuffer<T>* mZbuffer;
    LightSource<T> const* mLight;
    Polygon<T> const* mOriginalPolygon;
    Polygon<T> mProjected;
    // draw helper functions:
    void hLineGouraud(int const & x0, int const & x1, int const & y, T const & z0, T const & z1, Color const & c0, Color const & c1);
    void hLinePhong(int const & x0, int const & x1, int const & y,
T const k z0, T const & z1,
Vector<3,T> const & n0, Vector<3,T> const & n1,
Point<3,T> const & p0, Point<3,T> const & p1;
void hLineFlat(int const & xLeft, int const & xRight, int const & y,
T const & z0, T const & z1, Color flatColor);
void hLineLine(int const & xLeft, int const & xRight, int const & y,
T const & z0, T const & z1, Color lineColor);

public:
// c 'tor
ScreenPolygon(Polygon<T> const & polygon, Zbuffer<T> & zbuffer,
Camera<T>& camera, LightSource<T> const & light);
void forceXYs(int x0, int y0, int x1, int y1, int x2, int y2); // for testing ONLY
void draw(char shading); // draws the line to the screen
void print(); // prints the fields of the screen polygon to the screen

};

B.6 graphic_draw.cpp

# include <stdexcept>
# include <cmath>
# include <assert.h>
# include "graphic_draw.hpp"

using namespace graphic::draw;
using namespace graphic::tools;

/* Line implementation */

// default c 'tor
ScreenLine::ScreenLine() {}

// c 'tor taking two Point<2, int> s as argument
ScreenLine::ScreenLine(Point<2, int> const & startPoint,
Point<2, int> const & endPoint)
{ mStartPoint = startPoint;
  mEndPoint = endPoint;
}

Point<2, int> const & ScreenLine::setStartPoint(
  Point<2, int> const & newStartPoint)
{ return mStartPoint = newStartPoint;
}

Point<2, int> const & ScreenLine::setEndPoint(
  Point<2, int> const & newEndPoint) 
{ return mEndPoint = newEndPoint;
}
APPENDIX B. SOURCE FILES

34  // sets the start and end Point<2, int> to newColor
35  Color const & ScreenLine::setColor(Color const & newColor) {
36    mStartColor = newColor;
37    mEndColor = newColor;
38    return newColor;
39  }
40
41  void ScreenLine::setColor(Color const & newStartColor,
42   Color const & newEndColor) {
43    mStartColor = newStartColor;
44    mEndColor = newEndColor;
45  }
46
47  Point<2, int> const & ScreenLine::getStartPoint() const {
48    return mStartPoint;
49  }
50
51  Point<2, int> const & ScreenLine::getEndPoint() const {
52    return mEndPoint;
53  }
54
55  // Z buffer implementation
56  template<typename T>
57  Zbuffer<T>::Zbuffer(DotDevice& device, Color const& color = grey) {
58    height = std::floor(T(device.Height()) / device.UnitLength());
59    width = std::floor(T(device.Width()) / device.UnitLength());
60    mBuffer.resize(width);
61    for (int x=0; x < mBuffer.size(); x++) {
62      mBuffer[x].resize(height);
63    }
64    for (int y=0; y < mBuffer.size(); y++) {
65      for (int x=0; x < mBuffer.size(); x++) {
66        mBuffer[x][y].zcoord = 0;
67      }
68    }
69  }
for (int y=0; y< height; y++) {
    mBuffer[x][y].zcoord = 0;
    mBuffer[x][y].clr = color;
}
}

mDevice = &device;
}

/**
 * ScreenPolygon implementation
 */

template<typename T>
void Zbuffer<T>::setPixel(int x, int y, T z, Color const& c) {
    //only draw id coordinates inside z buffer
    if (checkPixel(x, y, z)) {
        mBuffer[x][y].zcoord = z;
        mBuffer[x][y].clr = c;
    }
}

template<typename T>
bool Zbuffer<T>::checkPixel(int x, int y, T z) {
    return x>=0 && x<width && y>=0 && y<height && z>=mBuffer[x][y].zcoord && z<=1;
}

template<typename T>
void Zbuffer<T>::drawBuffer() {
    for (int x = 0; x < width; x++) {
        for (int y = 0; y < height; y++) {
            mDevice->SetPixel(
                x, y,
                mBuffer[x][y].clr.getRed(),
                mBuffer[x][y].clr.getGreen(),
                mBuffer[x][y].clr.getBlue() );
        }
    }
    device->Update();
}

ScreenPolygon<T>::ScreenPolygon(Polygon<T> const& polygon, Zbuffer<T>& zbuffer, Camera<T>& camera, LightSource<T> const& light) {
    //mCamera(&camera), mLigh跟你 (With) {mZbuffer = &zbuffer;
mCamera = &camera;
mLight = &light;
mOriginalPolygon = &polygon;
mProjected = polygon;
v0.p = &polygon.point(0);
v0.n = &polygon.normal(0);
v1.p = &polygon.point(1);
v1.n = &polygon.normal(1);
v2.p = &polygon.point(2);
v2.n = &polygon.normal(2);

// projection matrix calculated from camera
Matrix<4,T> projectionMatrix =
camera.getProjectionMatrix(zbuffer.width, zbuffer.height);

// polygon's points projected with projection matrix –
// normals are unchanged.
mProjected.projectPoints(projectionMatrix);

// bottom vertex
v0.x = int(floor(mProjected.point(0)[0] + 0.5)), // x
v0.y = int(floor(mProjected.point(0)[1] + 0.5)), // y
v0.z = mProjected.point(0)[2]; // z

// middle vertex (might be top or bottom)
v1.x = int(floor(mProjected.point(1)[0] + 0.5)), // x
v1.y = int(floor(mProjected.point(1)[1] + 0.5)), // y
v1.z = mProjected.point(1)[2]; // z

// top vertex
v2.x = int(floor(mProjected.point(2)[0] + 0.5)), // x
v2.y = int(floor(mProjected.point(2)[1] + 0.5)), // y
v2.z = mProjected.point(2)[2]; // z

// sort vertices ascending by y-value.
// if y of position 1 > y of position 2 –OR– if they are equal
// but x of position 1 > x of position 2 then swap positions.
if(v1.y > v2.y || (v1.y == v2.y && v1.x > v2.x))
    std::swap(v1, v2);

// ditto for positions 0 and 1
if(v0.y > v1.y || (v0.y == v1.y && v0.x > v1.x)) {
    std::swap(v0, v1);
    // if position 0 and 1 has been swapped,
    // it might be necessary to swap 1 and 2 again.
    if(v1.y > v2.y || (v1.y == v2.y && v1.x > v2.x))
        std::swap(v1, v2);
}

// This method forces the xy values into specific values.
// It's for testing ONLY.

template<typename T>
void ScreenPolygon<T>::forceXYs(int x0, int y0, int x1,
                                int y1, int x2, int y2) {
    v0.x = x0;
v1.x = x1;
v2.x = x2;
v0.y = y0;
v1.y = y1;
v2.y = y2;

    // sort vertices ascending by y-value.
    // if y of position 1 > y of position 2 –OR– if they are equal
    // but x of position 1 > x of position 2 then swap positions.
    if(v1.y > v2.y || (v1.y == v2.y && v1.x > v2.x))
        std::swap(v1, v2);
// ditto for positions 0 and 1
if (v0.y > v1.y || (v0.y == v1.y && v0.x > v1.x)) {
    std::swap(v0, v1);
    // if position 0 and 1 has been swapped,
    // it might be necessary to swap 1 and 2 again.
    if (v1.y > v2.y || (v1.y == v2.y && v1.x > v2.x))
        std::swap(v1, v2);
}

template<typename T>
void ScreenPolygon<T>::hLineGouraud(int const & x0, int const & x1,
    int const & y, T const & z0, T const & z1,
    Color const & c0, Color const & c1) {
    if (x0 == x1) // nothing needs to be drawn
        return;
    // total change of x
    int deltaX = x1 - x0;
    // change of z per x-step
    T dZ = (z1 - z0)/deltaX;
    // change of color per x-step
    double dRed = (c1.getRed() - c0.getRed())/deltaX;
    double dGreen = (c1.getGreen() - c0.getGreen())/deltaX;
    double dBlue = (c1.getBlue() - c0.getBlue())/deltaX;
    // loop vars
    int x = x0; T z = z0; Color c = c0;
    for (x = x0, z = z0, c = c0; x < x1; ++x) {
        mZbuffer->setPixel(x, y, z, c);
        c.changeRed(dRed);
        c.changeGreen(dGreen);
        c.changeBlue(dBlue);
        z += dZ;
    }
}

template<typename T>
void ScreenPolygon<T>::hLinePhong(int const & x0, int const & x1,
    int const & y, T const & z0, T const & z1,
    Vector<3,T> const & n0, Vector<3,T> const & n1,
    Point<3,T> const & p0, Point<3,T> const & p1) {
    if (x0 == x1) // nothing needs to be drawn
        return;
    // total change of x
    int deltaX = x1 - x0;
    // reproc
    T rDeltaX = 1/T(deltaX);
    // change of z per x-step
    T dZ = (z1 - z0)/T(deltaX);
    // change of normal per x-step
    Vector<3,T> dN = (n1 - n0) * rDeltaX;
    // change of point per x-step
    Vector<3,T> dP = vectorFromPoints(p0, p1) * rDeltaX;
    // loop vars
    int x, T z, Vector<3,T> n, Point<3,T> p;
for (x = x0; x = x0, n = n0, p = p0; x < x1; ++x) {
    Color c = mLight->PhongIllumination(mProjected.material(),
    p, n, *mCamera);
    // we check if we want to draw in setPixel
    mZbuffer->setPixel(x, y, z, c);
    n += dN;
    n.normalize();
    p.move(dP);
    z += dZ;
}

template<typename T>
void ScreenPolygon<T>::hLineFlat(int const & xLeft, int const & xRight,
    int const & y, T const & z0, T const & z1, Color flatColor) {
    // total change of x
    int deltaX = xRight - xLeft;
    T dZ = (z1 - z0)/(deltaX);
    int x;
    T z = z0;
    /* std::cout << y << "," << xLeft << "," << xRight
     *<",",(xRight - xLeft) << std::endl;*/
    for (x = xLeft; x < xRight; ++x) {
        mZbuffer->setPixel(x, y, z, flatColor);
        z += dZ;
    }
    // if (z != z1)
    // std::cout << "error in hLineFlat, z is " << z << ", z0 is "
    // << z0 << "," << z1 << "," << z1 << "," << dZ << std::endl;
}

template<typename T>
void ScreenPolygon<T>::hLineLine(int const & xLeft, int const & xRight,
    int const & y, T const & z0, T const & z1, Color lineColor) {
    mZbuffer->setPixel(xLeft, y, z0, lineColor);
    mZbuffer->setPixel(xRight - 1, y, z1, lineColor);
}

template<typename T>
void ScreenPolygon<T>::draw(char shading) {
    // determining shading model
    bool flat = false;
    bool gouraud = false;
    bool line = false;
    bool phong = false;
    if (shading == 'g' || shading == 'G')
        gouraud = true;
    else if (shading == 'p' || shading == 'P')
        phong = true;
    else if (shading == 'f' || shading == 'F')
        flat = true;
    else if (shading == 'l' || shading == 'L')
        line = true;
// colors for use with gouraud shading
Color colorTopLeft, colorTopRight, colorBottomLeft, colorBottomRight;

// determining vertices of polygon
Vertexdata∗ topLeft, ∗ topRight, ∗ bottomLeft, ∗ bottomRight;

if ( v0 . y == v1 . y ) { // if polygon has bottom edge ...
    topLeft = &v2;  
    topRight = &v2; // ... side edges have common top point...
    colorTopLeft = colorTopRight = mLight−>PhongIllumination(mProjected . material (), *
        (v2 . p) , *(v2 . n) , *mCamera);
    bottomLeft = &v0; // ... but different bottom points
    colorBottomLeft = mLight->PhongIllumination(mProjected . material () , *
        (v0 . p) , *(v0 . n) , *mCamera);
    bottomRight = &v1;
    colorBottomRight = mLight->PhongIllumination(mProjected . material () , *
        (v1 . p) , *(v1 . n) , *mCamera); }

else { // if polygon has 'pointy' bottom
    // make the vectors from v0 to v1 and v2
    T vecVAR0[2] = {(T)v0 . x, (T)v0 . y};
    T vecVAR1[2] = {(T)v1 . x, (T)v1 . y};
    T vecVAR2[2] = {(T)v2 . x, (T)v2 . y};
    Point<2,T> poiP0(vecVAR0);
    Point<2,T> poiP1(vecVAR1);
    Point<2,T> poiP2(vecVAR2);
    Vector<2,T> vecV01 = vectorFromPoints(poiP0, poiP1);
    Vector<2,T> vecV02 = vectorFromPoints(poiP0, poiP2);
    // calculate determinant
    T determinant = vecV01[0] * vecV02[1] − vecV01[1] * vecV02[0];

    if ( determinant < 0 ) { // determinant negative ⇒ vecV01 left edge
        // side edges have different top points
        topLeft = &v1;
        colorTopLeft = mLight−>PhongIllumination(mProjected . material (), *
            (v1 . p) , *(v1 . n) , *mCamera);
        topRight = &v2;
        colorTopRight = mLight->PhongIllumination(mProjected . material () , *
            (v2 . p) , *(v2 . n) , *mCamera); }

    } else { // side edges have different top points
        topLeft = &v2;
        colorTopLeft = mLight->PhongIllumination(mProjected . material () , *
            (v2 . p) , *(v2 . n) , *mCamera);
        topRight = &v1;
        colorTopRight = mLight->PhongIllumination(mProjected . material () , *
            (v1 . p) , *(v1 . n) , *mCamera); }

    } // if ( determinant >= 0 ) { // determinant positive or zero ⇒ vecV02
    // left edge, catches line!.
    else { // side edges have different top points
        topLeft = &v2;
        colorTopLeft = mLight->PhongIllumination(mProjected . material () , *
            (v2 . p) , *(v2 . n) , *mCamera);
        topRight = &v1;
        colorTopRight = mLight->PhongIllumination(mProjected . material () , *
            (v1 . p) , *(v1 . n) , *mCamera); }

    } // if polygon has bottom edge...
APPENDIX B. SOURCE FILES

376 // side edges have common bottom point
377 bottomLeft = bottomRight = &v0;
378 colorBottomLeft = colorBottomRight
379 = mLight->PhongIllumination(mProjected.material(),
380 *(v0.p), *(v0.n), *mCamera);
381 }
382 // x-coords of starting points
383 int xLeft = bottomLeft->x;
384 int xRight = bottomRight->x;
385 // numerators of inverse slope coefficients
386 int numLeft = topLeft->x - bottomLeft->x;
387 int numRight = topRight->x - bottomRight->x;
388 // delta x = sign of numerators
389 int dXLeft = graphic::tools::sign(numLeft);
390 int dXRight = graphic::tools::sign(numRight);
391 // denominators of inverse slope coefficients
392 int denLeft = topLeft->y - bottomLeft->y;
393 int denRight = topRight->y - bottomRight->y;
394 // incrementors controlling overrun/underrun
395 int incLeft = denLeft;
396 int incRight = denRight;
397 // z-coords of starting points
398 T zLeft = bottomLeft->z;
399 T zRight = bottomRight->z;
400 // change of z-coords per y-step
401 T dZLeft = (topLeft->z - bottomLeft->z)/denLeft;
402 T dZRight = (topRight->z - bottomRight->z)/denRight;
403 // normals for use with phong shading
404 // bottomLeft->p) print("bottomLeft: ");
405 // topLeft->p) print("topLeft: ");
406 // bottomRight->p) print("bottomRight: ");
407 // topRight->p) print("topRight: ");
408 Vector<3,T> normalLeft = *(bottomLeft->n);
409 Vector<3,T> normalRight = *(bottomRight->n);
410 // change of normals per y-step
411 Vector<3,T> dNormalLeft = (*(topLeft->n) - *(bottomLeft->n))*
412 (1/(T)denLeft);
413 Vector<3,T> dNormalRight = (*(topRight->n) - *(bottomRight->n))*
414 (1/(T)denRight);
415 // points for use with phong shading
416 Point<3,T> pointLeft = *bottomLeft->p;
417 Point<3,T> pointRight = *bottomRight->p;
418 // change of points per y-step
419 Vector<3,T> dPointLeft
420 = vectorFromPoints(*(bottomLeft->p), *(topLeft->p)) * (1/(T)denLeft);
421 Vector<3,T> dPointRight
422 = vectorFromPoints(*(bottomRight->p), *(topRight->p)) * (1/(T)denRight);
423 // colors for use with gouraud shading
424 Color colorLeft = colorBottomLeft;
425 Color colorRight = colorBottomRight;
426 // change of color per y-step
427 double dRedLeft
428 = (colorTopLeft.getRed() - colorBottomLeft.getRed()) / denLeft;
429 double dGreenLeft
430 = (colorTopLeft.getGreen() - colorBottomLeft.getGreen()) / denLeft;
double dBlueLeft
= (colorTopLeft.getBlue() - colorBottomLeft.getBlue()) / denLeft;
double dRedRight
= (colorTopRight.getRed() - colorBottomRight.getRed()) / denRight;
double dGreenRight
= (colorTopRight.getGreen() - colorBottomRight.getGreen()) / denRight;
double dBlueRight
= (colorTopRight.getBlue() - colorBottomRight.getBlue()) / denRight;

// scan-line loop, drawing lines from y-height v0 to v1
for (int y = v0.y; y < v1.y; ++y) {
  if (gouraud) { // gouraud shading: interpolate colors
    hLineGouraud(xLeft, xRight, y, zLeft, zRight, colorLeft, colorRight);
    colorLeft.changeRed(dRedLeft);
    colorLeft.changeGreen(dGreenLeft);
    colorLeft.changeBlue(dBlueLeft);
    colorRight.changeRed(dRedRight);
    colorRight.changeGreen(dGreenRight);
    colorRight.changeBlue(dBlueRight);
  }
  else if (phong) { // phong shading: interpolate world points and normals:
    hLinePhong(xLeft, xRight, y, zLeft, zRight, normalLeft, normalRight, pointLeft, pointRight);
    normalLeft += dNormalLeft;
    normalLeft.normalize();
    pointLeft.move(dPointLeft);
    normalRight += dNormalRight;
    normalRight.normalize();
    pointRight.move(dPointRight);
  }
  else if (flat) { // flat shading: color in one color
    Vector<3,T> avrNormal = (v0.n + v1.n + v2.n) * (1/3.0);
    avrNormal.normalize();
    Color flatColor = mLight->PhongIllumination(mProjected.material(), *(v1.p),
                                             avrNormal, +mCamera);
    // assert (xLeft <= xRight);
    if(xLeft > xRight) {
      std::cout << xLeft << " ," << xRight
       << std::endl;
      std::cout << "Bottom:\(" << bottomRight->x << ",\"
              << bottomRight->y << ")" << std::endl;
      std::cout << "TopRight:\(" << topRight->x << ",\"
              << topRight->y << ")" << std::endl;
      std::cout << "TopLeft:\(" << topLeft->x << ",\"
              << topLeft->y << ")" << std::endl;
      hLineFlat(xLeft, xRight, y, zLeft, zRight, flatColor);
    }
    else if (line) { // line shading: color in one color
      Color lineColor = mProjected.material().getDiffuseColor();
      hLineLine(xLeft, xRight, y, zLeft, zRight, lineColor);
    }
    incLeft += numLeft;
  while(incLeft > denLeft || 0 >= incLeft ) {

APPENDIX B. SOURCE FILES

xLeft += dXLeft;
incLeft -= dXLeft * denLeft;
}

zLeft += dZLeft;
incRight += numRight;
while(incRight > denRight || 0 >= incRight) {
    xRight += dXRight;
    incRight -= dXRight * denRight;
}

zRight += dZRight;

// Initializing vars for next scan-line loop.
// If both of the following conditions are false:
// either polygon has top edge and has already been drawn, or
// polygon has bottom edge and no variables has changed.
// In either case no vars need to be updated.
if (topLeft->y < topRight->y) { // Left edge dominated polygon
    // Left edge loop parameters need to be updated
    // Right edge loop parameters are unchanged
    topLeft = &v2;
    colorTopLeft = mLight->PhongIllumination(mProjected.material(),
        *(v2.p), *(v2.n), *mCamera);
    bottomLeft = &v1;
    colorBottomLeft = mLight->PhongIllumination(mProjected.material(),
        *(v1.p), *(v1.n), *mCamera);
    numLeft = topLeft->x - bottomLeft->x;
    dXLeft = graphic::tools::sign(numLeft);
    dZLeft = (topLeft->z - bottomLeft->z)/denLeft;
    dNormalLeft = (*topLeft->n - *(bottomLeft->n)) * (1/(T) denLeft);
    dPointLeft = vectorFromPoints(*(bottomLeft->p), *(topLeft->p)) * (1/(T) denLeft);
    dRedLeft = (colorTopLeft.getRed() - colorBottomLeft.getRed()) / denLeft;
    dGreenLeft = (colorTopLeft.getGreen() - colorBottomLeft.getGreen()) / denLeft;
    dBlueLeft = (colorTopLeft.getBlue() - colorBottomLeft.getBlue()) / denLeft;
}
else if (topLeft->y > topRight->y) { // Right edge dominated polygon
    // Right edge loop parameters need to be updated
    // Left edge loop parameters are unchanged
    topRight = &v2;
    colorTopRight = mLight->PhongIllumination(mProjected.material(),
        *(v2.p), *(v2.n), *mCamera);
    bottomRight = &v1;
    colorBottomRight = mLight->PhongIllumination(mProjected.material(),
        *(v1.p), *(v1.n), *mCamera);
    numRight = topRight->x - bottomRight->x;
    dXRight = graphic::tools::sign(numRight);
    denRight = topRight->y - bottomRight->y;
}
incRight = denRight;

dZRight = (topRight->z - bottomRight->z)/denRight;

dNormalRight = ((topRight->n) - (bottomRight->n)) * 
    (1/(T)denRight);

dPointRight = vectorFromPoints(*bottomRight->p, *topRight->p) * 
    (1/(T)denRight);

dRedRight = (colorTopRight.getRed() - colorBottomRight.getRed()) / 
    denRight;

dGreenRight = (colorTopRight.getGreen() - colorBottomRight.getGreen()) / 
    denRight;

dBlueRight = (colorTopRight.getBlue() - colorBottomRight.getBlue()) / 
    denRight;

} // scan-line loop, drawing lines from y-height v1 to v2
for (int y = v1.y; y < v2.y; ++y) {
    if (gouraud) { // gouraud shading: interpolate colors
        hLineGouraud(xLeft, xRight, y, zLeft, zRight, 
                colorLeft, colorRight);
        colorLeft.changeRed(dRedLeft);
        colorLeft.changeGreen(dGreenLeft);
        colorLeft.changeBlue(dBlueLeft);
        colorRight.changeRed(dRedRight);
        colorRight.changeGreen(dGreenRight);
        colorRight.changeBlue(dBlueRight);
    }
    else if (phong) { // phong shading: interpolate world points and normals:
        hLinePhong(xLeft, xRight, y, zLeft, zRight, 
                normalLeft, normalRight, pointLeft, pointRight);
        normalLeft += dNormalLeft;
        normalLeft.normalize();
        pointLeft.move(dPointLeft);
        normalRight += dNormalRight;
        normalRight.normalize();
        pointRight.move(dPointRight);
    } else if (flat) { // flat shading: color in one color
        Vector<3,T> avrNormal = (*v0.n + *v1.n + *v2.n) * (1/3.0);
        avrNormal.normalize();
        if(xLeft > xRight)
        {
            std::cout << xLeft << ";" << xRight << std::endl;
            std::cout << "Bottom: (" << bottomRight->x << "," 
                << bottomRight->y << "," 
                    << bottomRight->z)" << std::endl;
            std::cout << "TopRight: (" << topRight->x << "," 
                << topRight->y << "," 
                    << topRight->z)" << std::endl;
            std::cout << "TopLeft: (" << topLeft->x << "," 
                << topLeft->y << "," 
                    << topLeft->z)" << std::endl;
        }
        Color flatColor = mLight->PhongIllumination(mProjected.material(), 
            *v1.p, avrNormal, *mCamera);
        hLineFlat(xLeft, xRight, y, zLeft, zRight, flatColor);
    } else if (line) {
        Color lineColor = mProjected.material().getDiffuseColor();
APPENDIX B. SOURCE FILES

604     hLineLine(xLeft, xRight, y, zLeft, zRight, lineColor);
605 }
606     incLeft += numLeft;
607     while(incLeft > denLeft || 0 >= incLeft) {
608         xLeft += dXLeft;
609         incLeft -= dXLeft * denLeft;
610     }
611     zLeft += dZLeft;
612     incRight += numRight;
613     while(incRight > denRight || 0 >= incRight) {
614         xRight += dXRight;
615         incRight -= dXRight * denRight;
616     }
617     zRight += dZRight;
618 }
619 }
620 }
621 template<typename T>
622     void ScreenPolygon<T>::print() {
623         (*v0.p).print("v0 point:\n");
624         (*v0.n).print("n0 point:\n");
625         (*v1.p).print("v1 point:\n");
626         (*v1.n).print("n1 point:\n");
627         (*v2.p).print("v2 point:\n");
628         (*v0.n).print("n2 point:\n");
629 }

B.7 readdata.cpp

1 #include <cstdio>
2 #include <iostream>
3 #include <iomanip>
4 #include <fstream>
5 #include <vector>
6
7 /*****************************************************************************
8 * R e a d D a t a F i l e ( c o n s t c h a r * ) * *
9 * The file contains the specification of a number of Bezier patches *
10 * This function ‘ReadDataFile(const char* filename)’ which can read *
11 * the specific file ‘patches.data’ are provided by knud henriksen *
12 * The function is by no means general! It can only read the *
13 * data file ‘patches.data’, and that is also the intention! *
14 * The purpose of the this function is to make it easy for the *
15 * students to get the control points of the specific *
16 * Bezier surfaces, which are obligatory in the report. *
17 * *
18 \*****************************************************************************/
19 //typedef graphics::tools::Point Point;
20 typedef double T;
int ReadDataFile(const char* filename, 
    std::vector<std::vector<Point<T>>> &patches) { 
    const int NVERTEX = 0; 
    const int READ_VERTICES = 1; 
    const int PATCHNAME = 2; 
    const int SEARCH_PATCHES = 3; 
    const int READ_PATCHES = 4; 

    std::vector<Point<T>> vertices; 
    int patchCounter = 0; 
    char ch; 
    const int MAX_BUFFER = 256; 
    char InputBuffer[MAX_BUFFER]; 
    int NumberOfVertices; 
    int VertexNumber; 
    double x; 
    double y; 
    double z; 
    char PatchName[MAX_BUFFER]; 
    int PatchNumber; 
    int index_11, index_12, index_13, index_14; 
    int index_21, index_22, index_23, index_24; 
    int index_31, index_32, index_33, index_34; 
    int index_41, index_42, index_43, index_44; 

    ifstream data_file(filename); 
    if (!data_file) { 
        cerr << "Cannot open data file: " 
             << filename << endl << flush; 
        return -1; 
    } 

    int state = NVERTEX; 
    while (data_file.get(InputBuffer, MAX_BUFFER, '\n')) { 
        if (data_file.get(ch) && ch != '\n') { 
            data_file.close(); 
            cerr << "Eof on data file: " << filename 
                 << endl << flush; 
            return -1; 
        } 
    } 

    // Now one line of data is in InputBuffer 
    // Use sscanf to extract the numeric values 
    // Build a data structure which represents the different 
    // vertices and patches 
    // There are several possibilities for the input lines:
APPENDIX B. SOURCE FILES

// 1: a comment line, i.e. InputBuffer[0] == '#'
// 2: a 'number of vertices line', i.e. just one number
sscannf(InputBuffer, "%d", &NumberOfVertices)
// 3: a 'vertex line', i.e. 4 numbers
sscannf(InputBuffer, "%d%f%f%f", &VertexNumber, &x, &y, &z)
// 4: a 'patch line', i.e. 17 numbers
sscannf(InputBuffer,
"%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d",
&PatchNumber,
&index[0], &index[1], &index[2], &index[3], &index[4],
&index[5], &index[6], &index[7], &index[8], &index[9],
&index[10], &index[11], &index[12], &index[13],
&index[14], &index[15], &index[16], &index[17],
&index[18], &index[19], &index[20], &index[21])

switch (state) {
case NVERTEX:
if (InputBuffer[0] != '#') {
  if (sscannf(InputBuffer, "%d",
             &NumberOfVertices) != 1) {
    cerr << "number of vertices not found in data file: "
         << filename << endl << flush;
    return -1;
  }
  vertices.resize(NumberOfVertices);
  cout << "Number of Vertices = "
       << NumberOfVertices
       << endl << endl;
  state = READ_VERTICES;
}
break;
case READ_VERTICES:
if (InputBuffer[0] != '#') {
  if (sscannf(InputBuffer, "%d%f%f%f", &VertexNumber, &x, &y, &z) != 4) {
    cerr << "vertex not found in data file: "
         << filename << endl << flush;
    return -1;
  }
  else {
// insert the vertex in a
// data structure
  vertices[VertexNumber-1][0] = x;
  vertices[VertexNumber-1][1] = y;
  vertices[VertexNumber-1][2] = z;
  if (VertexNumber == NumberOfVertices) {
    state = PATCHNAME;
  }
  }
break;
case PATCHNAME:
  if (InputBuffer[0] == '#') {
    if (strlen(InputBuffer) > 2) {
      // read the name of the patch
      if (sscanf(InputBuffer, "#\%s", PatchName) != 1) {
        cerr << "patch name not found"
        << endl << flush;
        return -1;
      }
      cout << "patch name: " << PatchName
      << endl << flush;
      state = SEARCH_PATCHES;
    }
  }
break;
case SEARCH_PATCHES:
case READ_PATCHES:
if (InputBuffer[0] == '#') {
  if (state == READ_PATCHES) {
    state = PATCHNAME;
  }
}
else {
  state = READ_PATCHES;
  if (sscanf(InputBuffer, "%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d", &PatchNumber, &index_11, &index_12, &index_13, &index_14, &index_21, &index_22, &index_23, &index_24, &index_31, &index_32, &index_33, &index_34, &index_41, &index_42, &index_43, &index_44) != 17) {
    cerr << "No patch found in data file:"
    << filename
    << endl << flush;
    return -1;
  }
else {
  // insert patch in a data structure
  ++patchCounter;
  patches.resize(patchCounter);
  patches[patchCounter-1].resize(16);
  patches[patchCounter-1][0] = vertices[index_11-1];
  patches[patchCounter-1][1] = vertices[index_12-1];
  patches[patchCounter-1][2] = vertices[index_13-1];
  patches[patchCounter-1][3] = vertices[index_14-1];
  patches[patchCounter-1][4] = vertices[index_21-1];
  patches[patchCounter-1][5] = vertices[index_22-1];
  patches[patchCounter-1][6] = vertices[index_23-1];
B.8 showObligatory.cpp

```cpp
// #include "DotDevice/DotDevice.h"
#include <vector>
#include "graphic.h"
#include "readdata.cpp"

typedef double T;

DotDevice* device;
Zbuffer<T>* zbuf;

void showObligatory(char shading, int sign, char filename[], int depth, T zoom, T pos[], Color matCol)
{
  using namespace graphic;

  std::vector<std::vector<Point3<T>>> patches;
  int result = ReadDataFile(filename, patches);

  // zbuffer
  zbuf = new Zbuffer<T>(*device, white);

  // setting up the world system
```
APPENDIX B. SOURCE FILES

```cpp
world::Material<T> material(matCol, white, 0.4, 0.6, 0.7, 3);

//camera
T aim[] = {0,0,0};
T up[] = {0,0,-1};
Point<3,T> pAim(aim);
Point<3,T> pPos(pos);
Vector<3,T> pUp(up);
Camera<T> camera(pos, aim, up, 1.2, zoom);

// lightsource
T tLightPoint[] = {0,5,7};
Point<3,T> lightPoint(tLightPoint);
LightSource<T> light(lightPoint);

// print the bezier surface
for (int i=0; i<patches.size(); i++) {
    BezierPatch<T> bp(patches[i], material, depth, shading, sign);
    bp.drawPatch(*zbuf, camera, light);
    std::cout << "patch " << i << std::endl;
}

void showParametric(char shading, int sign, T zoom, T pos[], Color matCol) {
    using namespace graphic;

    // zbuffer
    zbuf = new Zbuffer<T>(*device, white);

    // setting up the world system
    world::Material<T> material(matCol, white, 0.4, 0.6, 0.7, 3);

    //camera
    T aim[] = {-6,-6,0};
    T up[] = {0,0,-3};
    Point<3,T> pAim(aim);
    Point<3,T> pPos(pos);
    Vector<3,T> pUp(up);
    Camera<T> camera(pos, aim, up, 1.2, zoom, 0, -100);

    // lightsource
    T tLightPoint[] = {0,5,7};
    Point<3,T> lightPoint(tLightPoint);
    LightSource<T> light(lightPoint);

    ParametricSurface<T> ps(material, shading, sign);
    std::cout << "Here comes bottom:" << std::endl;
    ps.drawBottom(*zbuf, camera, light);
    std::cout << "Here comes handle:" << std::endl;
    ps.drawHandle(*zbuf, camera, light);
    std::cout << "Here comes top:" << std::endl;
    ps.drawTop(*zbuf, camera, light);
    std::cout << "Here comes middle:" << std::endl;
    ps.drawMiddle(*zbuf, camera, light);
}``
void DisplayCallBack()
{
    cout << ""encialDisplayCallback" << endl << flush;
    int unitlength = device->UnitLength(1); // clearing display
    device->Clear();
    device->Update();
    device->UnitLength(unitlength); // restoring unitlength
    zbuf->drawBuffer();
    device->Update();
}

int main(int argc, char** argv)
{
    using std::cout;
    using std::endl;
    using std::flush;
    device = new DotDevice(700, 700, "Obligatory");
    device->UnitLength(1);
    device->Clear();
    char tea[] = "teapot.data";
    char nor1[] = "Normal1.data";
    char nor2[] = "Normal2.data";
    char cone[] = "Cone.data";
    char sma[] = "Smarty.data";
    char sph[] = "Sphere.data";
    char pain[] = "pain.data";
    char disk[] = "disk.data";
    T front[3] = {8.0, 0.0, 4.0}; // frontal position
    T side[3] = {3.0, 6.0, 5.0}; // side position
    T top[3] = {3.0, 0.0, 10.0}; // top position
    T topD[3] = {3.0, 2.0, 10.0}; // top position
    //type of shade, sign of formal, file, depth, zoom factor, position of eye, color
    //showObligatory('g', -1, tea, 3, 0.8, front, blue);
    showObligatory('g', -1, nor1, 3, 2.5, front, green);
    //showObligatory('p', -1, tea, 3, 0.8, side, blue);
    //showObligatory('l', -1, tea, 2, 0.9, side, blue);
    //showObligatory('p', -1, tea, 3, 0.8, top, blue);
    //smart used to comment on bezier—surface usage
    //showObligatory('f', -1, sma, 0, 2.5, side, violet);
    //showObligatory('f', -1, sma, 1, 2.5, side, violet);
APPENDIX B. SOURCE FILES

B.9 testPolygons.cpp

```cpp
#include <vector>
#include "graphic.h"
#include "readdata.cpp"

// showObligatory ( 'f', -1, sma, 2, 2.5, side, violet);
// showObligatory ( 'p', -1, sma, 3, 2.5, side, violet);
// showObligatory ( 'g', -1, sma, 2, 2.5, side, violet);

// note the line art
// showObligatory ( 'g', -1, sph, 3, 2.5, front, magenta);
// showObligatory ( 'p', -1, sph, 3, 2.5, side, magenta);
// showObligatory ( 'l', -1, sph, 3, 2.5, top, magenta);
// showObligatory ( 'g', 1, pain, 3, 2.5, front, cyan );
// showObligatory ( 'p', 1, pain, 3, 2.5, side, cyan );
// showObligatory ( 'p', 1, pain, 3, 2.5, top, cyan );

// showObligatory ( 'g', 1, disk, 3, 2.5, front, green ); // noshow
// showObligatory ( 'p', 1, disk, 3, 2.5, side, green ); // disk1
// showObligatory ( 'g', -1, disk, 3, 2.5, topD, green );
// showObligatory ( 'g', -1, disk, 3, 2.5, top, green ); // noshow

/*
preferred signs of normals:
pain: 1
all others: -1
*/

T bottomP[3] = { -15.0, 30.0, -50.0 }; // top
T frontP[3] = { 40.0, 0, 20.0 }; // frontal position
T sideP[3] = { 15.0, 30.0, 25.0 }; // side position
T topP[3] = { 15.0, 0, 50.0 }; // top
// showParametric ( 'l', 1, 1.5, frontP, red);
// showParametric ( 'p', 1, 1.5, sideP, red );
// showParametric ( 'l', 1, 1.5, sideP, red );
// showParametric ( 'g', 1, 1.5, topP, red );
// showParametric ( 'p', 1, 2.5, bottomP, red );

glutDisplayFunc ( DisplayCallBack );

cout << "For this DisplayCallBack function you should see:");

cout << "A very nice figure"

cout << "g -- to toggle the grid on and off.");

cout << "d -- to redisplay the scene.");

cout << "q -- to quit the program."

cout << endl;

glutMainLoop ();

return 0;
```
// this function takes 3 points and draw the triangle
void drawTriangle(int x0, int y0, int x1, int y1, int x2, int y2){
    device->TestLine(x0,y0, x1,y1);
    device->TestLine(x1,y1, x2,y2);
    device->TestLine(x2,y2, x0,y0);
}

void testPolygons() {
    using namespace graphic;
    using namespace graphic;
    // first we build the camera, material, light and z-buffer
    world::Material< float > material1( grey, white, 1, 0, 0, 2 ); // purely ambient
    world::Material< float > material2( black, white, 1, 0, 0, 2 ); // purely ambient
    float aim[] = {1,1,0};
    float pos[] = {0,0,10.0};
    float up[] = {0,-1,0};
    Point<3,float> pAim( aim );
    Point<3,float> pPos ( pos );
    Vector<3,float> pUp(up);
    Camera< float > camera ( pos , aim , up , 1 , 0.1 , 0 , -100 ) ;
    float tLightPoint[] = {0.0,0.0,2.0};
    Point<3,float> lightPoint ( tLightPoint );
    LightSource< float > light ( lightPoint );
    zbuf = new Zbuffer< float >( *device , white ) ;
    // setting up the first triangle
    float t1[] = {10.0,10.0,0};
    float t2[] = {80.0,10.0,0};
    float t3[] = {40.0,80.0,0};
    Point<3> p1(t1);
    Point<3> p2(t2);
    Point<3> p3(t3);
    // setting up the second triangle
    float t4[] = {20,20,3};
    float t5[] = {80,85,-2};
    float t6[] = {95,30,-2};
    Point<3> p4(t4);
    Point<3> p5(t5);
    Point<3> p6(t6);
    // just a few dummy values — don’t matter
    float nA1[3] = {0,0,1};
    float nA2[3] = {0,0,1};
    float nA3[3] = {0,0,1};
    Vector<3,float> n1(nA1);
    Vector<3,float> n2(nA2);
    Vector<3,float> n3(nA3);
    n1.normalize ( );
    n2.normalize ( );
    n3.normalize ( );
    // world polygon
world::Polygon<float> polygon1(p1, p2, p3, n1, n2, n3, material1);
world::Polygon<float> polygon2(p4, p5, p6, n1, n2, n3, material2);
ScreenPolygon<float> screenPoly1(polygon1, *zbuf, camera, light);
ScreenPolygon<float> screenPoly2(polygon2, *zbuf, camera, light);
screenPoly1.draw('f');
screenPoly2.draw('f');
}

// the value in sign is multiplied on the normal vectors
// in effect one can invert all normals in one go
void testShading(int sign, char shading){
using namespace graphic;
float t1[] = {2.0, 1.0, -1.0};
float t2[] = {-1.0, -2.0, -2.0};
float t3[] = {0.5, -0.5, 1.0};
Point<3,float>p1(t1);
Point<3,float>p2(t2);
Point<3,float>p3(t3);
float nA1[3] = {0.7, -1.3, -0.3};
float nA2[3] = {1.3, -0.7, -0.3};
float nA3[3] = {1, -1, 0.3};
Vector<3,float>n1(nA1);
Vector<3,float>n2(nA2);
Vector<3,float>n3(nA3);
n1.normalize();
n2.normalize();
n3.normalize();
world::Material<float> material1(red, white, 0.5, 0.9, 0.8, 5);
world::Polygon<float> polygon1(p1, p2, p3, sign*n1, sign*n2, sign*n3, material1);
float aim[] = {0, 0, 0};
float pos[] = {4.0, -8.0, 0.0};
float up[] = {0, 0, 1};
Point<3,float>pAim(aim);
Point<3,float>pPos(pos);
Vector<3,float>pUp(up);
Camera<float> camera(pos, aim, up, 1, 1.5);
float tLightPoint[] = {2, -1.0};
Point<3,float>tLightPoint(tLightPoint);
LightSource<float> light(lightPoint, 0.1, 0.1, 0.1);
Zbuffer<float> zbuf = new Zbuffer<float>(*device, white);
ScreenPolygon<float> screenPoly1(polygon1, *zbuf, camera, light);
screenPoly1.draw(shading);
}

void DisplayCallBack(){
cout << "---DisplayCallBack" << endl << flush;
int unitlength = device->UnitLength(1); // clearing display
device->Clear();
device->Update();
device->UnitLength(unitlength); // restoring unitlength
// draw z buffer

APPENDIX B. SOURCE FILES

119  zbuf->drawBuffer();
120
121  // adding grids as the very last
122  device->Update();
123 }
124
125 int main(int argc, char** argv)
126 {
127  using std::cout;
128  using std::endl;
129  using std::flush;
130  device = new DotDevice(400, 300, "Shading");
131  device->Clear();
132  device->Update();
133  device->UnitLength(1);
134
135  // calling test function
136  // testPolygons();
137  // testShading(1, 'g'); // gouraud
138  // testShading(-1, 'g'); // gouraud, inverted normals
139  // testShading(1, 'p'); // phong
140  testShading(-1, 'p'); // phong, inverted normals
141
142  glutDisplayFunc(DisplayCallBack);
143
144  cout << "================================" << endl;
145  cout << "This will show a whole bunch of polygons" << endl;
146  cout << "In the window type:" << endl;
147  cout << "g - to toggle the grid on and off." << endl;
148  cout << "d - to redisplay the scene."
149  cout << "q - to quit the program."
150  cout << endl;
151  cout << endl;
152
153  // Let GLUT take over and rely on the call-back functions
154  glutMainLoop();
155
156  return 0;
157 }

B.10 testZbuffer.cpp

1 #include <vector>
2 #include "graphic.h"
3 #include "readdata.cpp"
4
5 DotDevice* device;
6 Zbuffer<float>* zbuf;
7
8 // this function takes 3 points and draw the triangle
9 void drawTriangle(int x0, int y0, int x1, int y1, int x2, int y2){
10  device->TestLine(x0,y0, x1,y1);
11  device->TestLine(x1,y1, x2,y2);
12  device->TestLine(x2,y2, x0,y0);
13 }
void testZBuffer() {
  using namespace graphic;

  // initializing consistent variables: materials polygons, cameras and lighting
  float t1[] = {2.0, 1.0, 0.0};
  float t2[] = {-1.0, -2.0, -0.2};
  float t3[] = {0.5, -0.5, 0.0};
  float t4[] = {2.0, 0.1, 0.0};
  float t5[] = {-1.0, -2.0, -0.5};
  float t6[] = {0.5, -0.5, -1};
  Point<3> p1(t1);
  Point<3> p2(t2);
  Point<3> p3(t3);
  Point<3> p4(t4);
  Point<3> p5(t5);
  Point<3> p6(t6);
  float nA1[3] = {0,0,-1};
  float nA2[3] = {0,0,-1};
  float nA3[3] = {0,0,-1};
  Vector<3,float> n1(nA1);
  Vector<3,float> n2(nA2);
  Vector<3,float> n3(nA3);
  n1.normalize();
  n2.normalize();
  n3.normalize();
  world::Material<float> material1(grey, white, 1, 0, 0, 2);
  world::Material<float> material2(violet, white, 1, 0, 0, 2);
  world::Polygon<float> polygon1(p1, p2, p3, n1, n2, n3, material1);
  world::Polygon<float> polygon2(p4, p5, p6, n1, n2, n3, material2);
  float aim[] = {6, 6, 0};
  float pos[] = {6, 6, 2};
  float up[] = {0,1,0};
  Point<3,float> pAim(aim);
  Point<3,float> pPos(pos);
  Camera<float> camera(pos, aim, up);
  float tLightPoint[] = {5, 5, 2};
  Point<3,float> lightPoint(tLightPoint);
  LightSource<float> light(lightPoint);

  // the z buffer
  zbuf = new Zbuffer<float>(*device, white);

  // front polygon
  ScreenPolygon<float> polyH1(polygon1, *zbuf, camera, light);
  polyH1.forceXYs(1,1,7,2,5,8);
  polyH1.draw('f');

  // back polygon
  ScreenPolygon<float> polyH2(polygon2, *zbuf, camera, light);
  polyH2.forceXYs(12,1,4,3,8,9);
  polyH2.draw('f');
}
void DisplayCallBack(){
APPENDIX B. SOURCE FILES

122

cout << "---DisplayCallBack" << endl << flush;
int unitlength = device->UnitLength(1); // clearing display
device->Clear();
device->Update();
device->UnitLength(unitlength); // restoring unitlength

//draw z buffer
zbuf->drawBuffer();

//draw lines of the triangles
drawTriangle(1,1,7,2,5,8); // polyH1
drawTriangle(12,1,4,3,8,9); // polyH2

// adding grids as the very last
device->Grid(1,1);
device->Update();
}

int main(int argc, char** argv)
{
  using std::cout;
  using std::endl;
  using std::flush;
  
  device = new DotDevice(400, 300, "DotDevice_Example");
device->Clear();
device->Update();
device->UnitLength(25);

  // calling test function
testZBuffer();
  glutDisplayFunc(DisplayCallBack);

  cout << "= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
  This will show a whole bunch of polygons" << endl;
cout << "In the window type:"
  cout << "g-to toggle the grid on and off."
  cout << "d-to redisplay the scene."
  cout << "q-to quit the program."
  cout << endl;

  // Let GLUT take over and rely on the call-back functions
  glutMainLoop();

  return 0;
}

B.11 testPolyDrawing.cpp

1 #include <vector>
2 #include "graphic.h"
3 #include "readdata.cpp"
4
5 DotDevice* device;
6 Zbuffer<float>* zbuf;

8 // this function takes 3 points and draw the triangle
9 void drawTriangle(int x0, int y0, int x1, int y1, int x2, int y2){
10 device->TestLine(x0,y0, x1,y1);
11 device->TestLine(x1,y1, x2,y2);
12 device->TestLine(x2,y2, x0,y0);
13 }

15 void testPolygon() {
16     using namespace graphic;
17     // we start out by clearing the display
18     // device->Clear();
19     // initializing consistent variables: materials polygons, cameras and lighting
20     float t1[] = {2.0, 1.0, -1.0};
21     float t2[] = {-1.0, -2.0, -2.0};
22     float t3[] = {0.5, -0.5, 1.0};
23     Point<3> p1(t1);
24     Point<3> p2(t2);
25     Point<3> p3(t3);
26     float nA1[3] = {0.7, -1.3, -0.3};
27     float nA2[3] = {1.3, -0.7, -0.3};
28     float nA3[3] = {1, -1, 0.3};
29     Vector<3,float> n1(nA1);
30     Vector<3,float> n2(nA2);
31     Vector<3,float> n3(nA3);
32     n1.normalize();
33     n2.normalize();
34     n3.normalize();
35     world::Material<float> material(green, white, 1, 0, 0, 2);
36     world::Polygon<float> polygon(p1, p2, p3, n1, n2, n3, material);
37     // Point<3> pVRP; pVRP[0] =
38     float aim[] = {0, 0, 0};
39     float pos[] = {4.0, -8.0, 0.0};
40     float up[] = {0, 0, 1};
41     Point<3,float> pAim(aim);
42     Point<3,float> pPos(pos);
43     Vector<3,float> pUp(up);
44     Camera<3,float> camera(pos, aim, up);
45     float tLightPoint[] = {2, -1, 0};
46     Point<3,float> lightPoint(tLightPoint);
47     LightSource<3,float> light(lightPoint);
48     // the z buffer
49     zbuf = new Zbuffer<float>(*device, white);
50     // polygon with horizontal bottom
51     ScreenPolygon<float> polyH1(polygon, *zbuf, camera, light);
52     polyH1.forceXYs(1, 2, 7, 2, 3, 10);
53     polyH1.draw('f');
54     // polygon with horizontal top
55     ScreenPolygon<float> polyH2(polygon, *zbuf, camera, light);
56     polyH2.forceXYs(9, 2, 5, 10, 12, 10);
57     polyH2.draw('f');
// stretched polygon 1
ScreenPolygon<float> polyS1(polygon, *zbuf, camera, light);
polyS1.forceXYs(11, 2, 13, 7, 19, 10);
polyS1.draw('f');

// stretched polygon 2
ScreenPolygon<float> polyS2(polygon, *zbuf, camera, light);
polyS2.forceXYs(10, 12, 14, 15, 15, 20);
polyS2.draw('f');

// regular polygon 1
ScreenPolygon<float> polyR1(polygon, *zbuf, camera, light);
polyR1.forceXYs(14, 2, 20, 2, 20, 8);
polyR1.draw('f');

// regular polygon 2
ScreenPolygon<float> polyR2(polygon, *zbuf, camera, light);
polyR2.forceXYs(5, 12, 1, 15, 5, 20);
polyR2.draw('f');

// regular polygon 3
ScreenPolygon<float> polyR3(polygon, *zbuf, camera, light);
polyR3.forceXYs(6, 12, 10, 15, 6, 20);
polyR3.draw('f');
}

void DisplayCallBack()
{
  cout << "−−DisplayCallBack" << endl << flush;
  int unitlength = device->UnitLength(1); // clearing display
  device->Clear();
  device->Update();
  device->UnitLength(unitlength); // restoring unitlength

  // draw z buffer
  zbuf->drawBuffer();

  // draw lines of the triangles
  drawTriangle(1, 2, 7, 2, 3, 10); // polyH1
  drawTriangle(9, 2, 5, 10, 12, 10); // polyS2
  drawTriangle(11, 2, 13, 7, 19, 10); // polyS1
  drawTriangle(10, 12, 14, 15, 15, 20); // polyS2
  drawTriangle(14, 2, 20, 2, 20, 8); // polyR1
  drawTriangle(5, 12, 1, 15, 5, 20); // polyR2
  drawTriangle(6, 12, 10, 15, 6, 20); // polyR3

  // adding grids as the very last
  device->Grid(1, 1);
  device->Update();
}

int main(int argc, char** argv)
{
  using std::cout;
  using std::endl;
  using std::flush;

  device = new DotDevice(600, 600, "DotDevice_Example");
device->Clear();
device->Update();
device->UnitLength(25);

// calling test function
testPolygon();
glutDisplayFunc(DisplayCallBack);

cout << "= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =" << endl;
cout << " This will show a whole bunch of polygons" << endl;
cout << " In the window type:" << endl;
cout << " grid on and off."" << endl;
cout << " display the scene."" << endl;
cout << " quit the program."" << endl;
cout << endl;

// Let GLUT take over and rely on the callback functions
 glutMainLoop();
return 0;

B.12 testMath.cpp

#include "DotDevice/DotDevice.h"
#include <vector>
#include "graphic.h"
#include "readdata.cpp"

void pause(std::string str) {
    std::string dummy;
    std::cout << " Press ENTER to continue...");
    std::getline(std::cin, dummy);
}

int testTools() {
    /** testing Point class***/
    // here follow a basic test of the Point class
    // first we make a few Points
    Point<2, double> p1;
    Point<2, double> p2;
    Point<2, double> p3;

    p1[0] = 1;
    p1[1] = 2;
    assert ( p1[0] == 1 && p1[1] == 2 );
    p2[0] = 4;
    p2[1] = 5;
    assert ( p2[0] == 4 && p2[1] == 5 );
    p2 = p1;
    p1[0] = 42;
    assert ( p1[0] == 42 && p1[1] == 2 );
    assert ( p2[0] == 1 && p2[1] == 2 );

    double values1[] = {3,3};
    p3.set(values1);
APPENDIX B. SOURCE FILES

```
# include <assert.h>
# include <iostream>

// first we make a few 3D Vectors
Vector<3, double> v1;
Vector<3, double> v2;
Vector<3, double> v3;

// normalization
assert( abs(v3.length() - 9.64365) < 0.001);
V3 = V1.normalize();
double v3nExpAr[3] = { 0.20739, 0.518476, 0.829561 };
Vector<3,double> v3nExp(v3nExpAr);
assert( V3 == v3nExp && abs(V3.length()-1) < 0.001 );

// dot product
double result = (V1 * V2);
assert( abs(V1 * V2 - 36) < 0.001 );

// vector−skalar multiplication
double multi = 3;
v3 = V1 * multi;
assert( V3[0] == 3 && V3[1] == 6 && V3[2] == 9 );
v3 = 2.0 * v1;
assert( V3[0] == 2 && V3[1] == 4 && V3[2] == 6 );

// vector addition
v3 = v1 + v2;
assert( V3[0] == 3 && V3[1] == 7 && V3[2] == 11 );
v3 = v1 - v2;
assert( V3[0] == -1 && V3[1] == -3 && V3[2] == -5 );

// vector cross product
v3 = cross(v1,v2);
assert( V3[0] == 1 && V3[1] == -2 && V3[2] == 1 );
v3 = cross(v1,v1);
assert( V3[0] == 0 && V3[1] == 0 && V3[2] == 0 );

// vector from points
float arP1[3] = {4,4,4};
Point<3,float> P1(arP1);
float arP2[3] = {4,4,4};
Point<3,float> P2(arP2);
Vector<3,float> V1 = vectorFromPoints(P1,P2);
assert( V1[0] == 0 && V1[1] == 0 && V1[2] == 0 );
float arP12[3] = {0,0,0};
```
APPENDIX B. SOURCE FILES

127

P1.set(arP12);
V1 = vectorFromPoints(P1,P2);
assert( V1[0] == 4 && V1[1] == 4 && V1[2] == 4 );
std::cout << "Vector_operations_worked_as_expected" << std::endl;

/******** testing Matrix class******/
float mVal[2][2] = {{1.2},{3.4}};
Matrix<2> m1(mVal);
assert( m1[0][0] == 1 && m1[1][0] == 3 && m1[1][1] == 4 );

// transposing
m1.transpose();
assert( m1[0][0] == 1 && m1[0][1] == 3 && m1[1][0] == 2 && m1[1][1] == 4 );

// set
Matrix<2> m2;
float mVal2[2][2] = {{5,6},{7,8}};
m2.set(mVal2);
assert( m2[0][0] == 5 && m2[0][1] == 6 && m2[1][0] == 7 && m2[1][1] == 8 );

// matrix multiplication
Matrix<2> m3 = m1 * m2;
assert( m3[0][0] == 26 && m3[0][1] == 30 && m3[1][0] == 38 && m3[1][1] == 44 );

// matrix addition and subtraction
Matrix<2> m4 = m1 + m2;
Matrix<2> m5 = m1 - m2;
assert( m4[0][0] == 6 && m4[1][0] == 9 && m4[1][1] == 12 );
assert( m5[0][0] == 4 && m5[0][1] == 3 && m5[1][0] == 5 && m5[1][1] == -4 );

// constructor initialization
double mVal3[3][3] = {{1,2,3},{4,5,6},{7,8,9}};
Matrix<3, double> m6(mVal3);
assert( m6[0][0] == 1 && m6[0][1] == 2 && m6[0][2] == 3 && m6[1][0] == 4 && m6[1][1] == 5 && m6[1][2] == 6 && m6[2][0] == 7 && m6[2][1] == 8 && m6[2][2] == 9 );

// matrix vector multiplication
Vector<3, double> v4 = m6 * v2;

// row and col
assert( m6.col(0)[0] == 1 && m6.col(0)[1] == 4 && m6.col(0)[2] == 7 );
assert( m6.row(1)[0] == 4 && m6.row(1)[1] == 5 && m6.row(1)[2] == 6 );
std::cout << "Basic_matrix_operations_worked_as_expected" << std::endl;

/******** testing rotation******/
float matVal1[4][4] = {{1,2,3,4}, {5,6,7,8}, {9,1,2,3}, {4,5,6,7}};
float matVal2[4][4] = {{1,0,0,0}, {0,1,0,0}, {0,0,1,0}, {0,0,0,1}}; // identity
Matrix<4> mat1(matVal1);
Matrix<4> mat2;

// RotateX
float expectedX1[4][4] = {{1,2,3,4}, {-2.82843,3.53553,3.53553,3.53553}, {9.89949,4.94975,6.94975,8.94975}};
float expectedX2[4][4] = {{1,0,0,0}, {0,-1,0}, {0,1,0}, {0,0,1}}; // identity
Matrix<4> matExp1(expectedX1);
Matrix<4> matExp2(expectedX2);
mat1.rotateX(45);
assert(mat1 == matExp1);
mat2.rotateX(90);
assert(mat2 == matExp2);
//std::cout << "RotateX worked as expected" << std::endl;

// RotateY
mat1.set(matVal1);
mat2.set(matVal2);
float expectedY1[4][4] = {{7.07107, 2.12132, 3.53553, 4.94975}, {5.6, 7.8}, {5.65685, -0.707107}, {4.24264, 5.65685, 7.07107}};
float expectedY2[4][4] = {{0, 0, 1, 0}, {0, 1, 0, 0}, {0, 0, 1, 0}, {0, 0, 0, 1}};
matExp1.set(expectedY1);
matExp2.set(expectedY2);
mat1.rotateY(45);
mat2.rotateY(90);
assert(mat1 == matExp1);
assert(mat2 == matExp2);
//std::cout << "RotateY worked as expected" << std::endl;

// RotateZ
mat1.set(matVal1);
mat2.set(matVal2);
float expectedZ1[4][4] = {{-2.82843, -2.82843, -2.82843, -2.82843}, {4.24264, 5.65685, 7.07107}, {9, 1, 2, 3}, {4, 5, 6, 7}};
float expectedZ2[4][4] = {{0, -1, 0, 0}, {1, 0, 0, 0}, {0, 0, 1, 0}, {0, 0, 0, 1}};
matExp1.set(expectedZ1);
matExp2.set(expectedZ2);
mat1.rotateZ(45);
mat2.rotateZ(90);
assert(mat1 == matExp1);
assert(mat2 == matExp2);
//std::cout << "RotateZ worked as expected" << std::endl;

// translating
mat1.set(matVal1);
mat2.set(matVal2);
float vecVal1[4] = {1, 2, 3, 1};
float vecVal2[4] = {0.5, 0.5, 0.5, 1};
Vector<4,float> vec1(vecVal1);
Vector<4,float> vec2(vecVal2);
float expectedT1[4][4] = {{5, 7, 9, 11}, {13, 16, 19, 22}, {21, 24}, {0.5, 0.5, 0.5, 1}};
matExp1.set(expectedT1);
matExp2.set(expectedT2);
mat1.translate(vec1);
mat2.translate(vec2);
assert(mat1 == matExp1);
assert(mat2 == matExp2);
//std::cout << "translation worked as expected" << std::endl;

// translating – the vectors used are defined under translation above, but shown below
mat1.set(matVal1);
mat2.set(matVal2);
float vecVal1[4] = {1, 2, 3, 1};
float vecVal2[4] = {0.5, 0.5, 0.5, 1};
Vector<4,float> vec1(vecVal1);
Vector<4,float> vec2(vecVal2);
float expectedSc1[4][4] = {{2, 1, 3, 4}, {10, 12, 14, 16}, {27, 3, 6, 9}, {4, 5, 6, 7}};
APPENDIX B. SOURCE FILES

float expectedSc2[4][4] = \{ \{0.5,0,0,0\}, \{0,0.5,0,0\}, \{0,0,0.5,0\}, \{0,0,0,1\}\};
matExp1.set(expectedSc1);
matExp2.set(expectedSc2);
mat1.scale(vec1);
mat2.scale(vec2);
assert(mat1 == matExp1);
assert(mat2 == matExp2);
//std::cout << "scaling worked as expected" << std::endl;

//xy shearing
mat1.set(matVal1);
mat2.set(matVal2);
float expectedShXY1[4][4] = \{ \{46,7,13,19\}, \{23,8,11,14\}, \{9,1,2,3\}, \{4,5,6,7\}\};
float expectedShXY2[4][4] = \{ \{1,0,5,0\}, \{0,1,2,0\}, \{0,0,1,0\}, \{0,0,0,1\}\};
matExp1.set(expectedShXY1);
matExp2.set(expectedShXY2);
mat1.shearXY(5,2);
mat2.shearXY(5,2);
assert(mat1 == matExp1);
assert(mat2 == matExp2);
//std::cout << "xy shearing worked as expected" << std::endl;

//xz shearing
mat1.set(matVal1);
mat2.set(matVal2);
float expectedShXZ1[4][4] = \{ \{26,32,38,44\}, \{5,6,7,8\}, \{19,13,16,19\}, \{4,5,6,7\}\};
float expectedShXZ2[4][4] = \{ \{1,5,0,0\}, \{0,1,0,0\}, \{0,2,1,0\}, \{0,0,0,1\}\};
matExp1.set(expectedShXZ1);
matExp2.set(expectedShXZ2);
mat1.shearXZ(5,2);
mat2.shearXZ(5,2);
assert(mat1 == matExp1);
assert(mat2 == matExp2);
//std::cout << "xz shearing worked as expected" << std::endl;

//yz shearing
mat1.set(matVal1);
mat2.set(matVal2);
float expectedShYZ1[4][4] = \{ \{1,2,3,4\}, \{10,16,22,28\}, \{11,5,8,11\}, \{4,5,6,7\}\};
float expectedShYZ2[4][4] = \{ \{1,0,0,0\}, \{5,1,0,0\}, \{2,0,1,0\}, \{0,0,0,1\}\};
matExp1.set(expectedShYZ1);
matExp2.set(expectedShYZ2);
mat1.shearYZ(5,2);
mat2.shearYZ(5,2);
assert(mat1 == matExp1);
assert(mat2 == matExp2);
//std::cout << "yz shearing worked as expected" << std::endl;

std::cout << "Rotation, Scaling, Shearing and Translation works as expected" << std::endl;
return 0;

//Testing some color functionality
int testColor(){
    //testing constructor correctness
    Color cl1(2.5,0.2,0.5);
    Color cl1s(1.0,0.2,0.5);
APPENDIX B. SOURCE FILES

259  Color c2(0.2,2.6,0.3);
260  Color c2s(0.2,1.0,0.3);
261  Color c3(0.3,0.5,500);
262  Color c3s(0.3,0.5,1.0);
263  Color c4(-1.0,4.0,6);
264  Color c4s(0.0,4.0,6);
265  Color c5(0.5,-0.1,0.6);
266  Color c5s(0.5,0.0,6);
267  Color c6(0.3,0.6,-700);
268  Color c6s(0.3,0.6,0);
269  assert( c1 == c1s && c2 == c2s && c3 == c3s && c4 == c4s && c5 == c5s && c6 == c6s );
270  std::cout << "Color constructor ran as expected" << std::endl;

272  // setRed
273  c1.setRed(-0.1);
274  Color c11(0.0,0.2,0.5);
275  assert( c1 == c11 );
276  c1.setRed(10.0);
277  Color c12(1.0,0.2,0.5);
278  assert( c1 == c12 );
279  c1.setRed(0.3);
280  Color c13(0.3,0.2,0.5);
281  assert( c1 == c13 );
282  std::cout << "Methods setRed, setGreen, setBlue worked as expected!" << std::endl;

288  // setGreen
289  c2.setGreen(-0.1);
290  Color c21(0.2,0.0,0.3);
291  assert( c2 == c21 );
292  c2.setGreen(10.0);
293  Color c22(0.2,1.0,0.3);
294  assert( c2 == c22 );
295  c2.setGreen(0.3);
296  Color c23(0.2,0.3,0.3);
297  assert( c2 == c23 );
298  std::cout << "setGreen, setBlue worked as expected!" << std::endl;

305  // setBlue
306  c3.setBlue(-1);
307  Color c31(0.3,0.5,0);
308  assert( c3 == c31 );
309  c3.setBlue(1.1);
310  Color c32(0.3,0.5,1.0);
311  assert( c3 == c32 );
312  c3.setBlue(0.4);
313  Color c33(0.3,0.5,0.4);
314  assert( c3 == c33 );
315  std::cout << "setBlue worked as expected!" << std::endl;

319  return 0;

321  // Testprogram for Camera – just the math
322  int testCameraMath()
323  {
324    // first we test the constructor
325    // using nice coordinates
326    float arVRP1[3] = {0,0,2};

327    return 0;
328  }
APPENDIX B. SOURCE FILES

Point<3,float> VRP1(arVRP1);
float arPRP1[3] = {0,0,4};
Point<3,float> PRP1(arPRP1);
float arVPN1[3] = {0,0,1};
Vector<3,float> VPN1(arVPN1);
float arVUP1[3] = {0,1,0};
Vector<3,float> VUP1(arVUP1);
float arW11[3] = {-2,-2,0};
Point<3,float> w11(arW11);
float arW21[3] = {2,2,0};
Point<3,float> w21(arW21);
float B1 = -10;
float F1 = -1;

Camera<float> C1(VRP1, PRP1, VPN1, VUP1, w11, w21, B1, F1);
//C1.print("C1 is: ");

Matrix<4,float> P1 = C1.getProjectionMatrix(300,300);
float expProj1[4][4] = {
    {21.4286, 0, -10.7143, 64.2857 },
    {0, 21.4286, -10.7143, 64.2857 },
    {0, 0, 0.039683, 0.317459 },
    {0, 0, -0.071429, 0.428571 } };
Matrix<4,float> expMat1(expProj1);

Matrix<4,float> expProj2[4][4] = {
    {40.6701, 4.35405, -15.3111, -205.625 },
    {8.64045, 28.0491, -1.10715, -115.495 },
    {-0.021347, -0.021347, 0.042694, 0.565562 },
    {0.024015, 0.024015, -0.04803, 0.363739 } };
Matrix<4,float> expMat2(expProj2);

Camera<float> C2(VRP2, PRP2, VPN2, VUP2, w12, w22, B2, F2);
//C2.print("C2 is: ");

Matrix<4,float> P2 = C2.getProjectionMatrix(300,300);
float expProj2[4][4] = {
    {40.6701, 4.35405, -15.3111, -205.625 },
    {8.64045, 28.0491, -1.10715, -115.495 },
    {-0.021347, -0.021347, 0.042694, 0.565562 },
    {0.024015, 0.024015, -0.04803, 0.363739 } };
Matrix<4,float> expMat2(expProj2);
Point<3, float> expWPRP2(expWPRPAr2);
assert (P2 == expMat2 && expWPRP2 == C2.getWorldPRP());
assert (expMat2 == C2.getProjectionMatrix(300,300));
std::cout << "Projection\_matrix\_and\_worldPRP\_of\_C2\_as\_expected" << std::endl;
return 0;
}

int testMaterialAndLight()
{
  Color difCol1(0.1,0.2,0.3);
  Color specCol1(0.4,0.5,0.6);
  float ka1 = 0.7;
  float kd1 = 0.8;
  float ks1 = 0.9;
  int n1 = 4;

  Material<float> material1(difCol1,specCol1,ka1,kd1,ks1,n1);
  // material1.print("Material1 is: ");
  assert (difCol1 == material1.getDiffuseColor() && specCol1 == material1.getSpecularColor() &&
          abs(ka1 - material1.getKa()) < 0.001 && abs(kd1 - material1.getKd()) < 0.001 &&
          abs(ks1 - material1.getKs()) < 0.001 && n1 == material1.getN());
  std::cout << "Material\_constructor\_and\_get\_functionality\_works\_as\_expected" << std::endl;

  float lightCoordsAr1[3] = {5,5,5};
  float attenAr1[3] = {1,2,3};
  Point<3, float> lightCoords1(lightCoordsAr1);
  Point<3, float> atten1(attenAr1);
  Color ambia1(0.3,0.4,0.5);
  Color intens1(0.1,0.5,0.9);
  LightSource<float> lightSource1(lightCoords1,atten1,ambia1,intens1);
  // lightSource1.print("LightSource1 is: ");

  // testing illumination functionality
  // first we make a camera
  float arVRP1[3] = {0,0,2};
  Point<3, float> VRP1(arVRP1);
  float arPRP1[3] = {0,0,4};
  Point<3, float> PRP1(arPRP1);
  float arVPN1[3] = {0,0,1};
  Vector<3, float> VPN1(arVPN1);
  float arVUP1[3] = {0,1,0};
  Vector<3, float> VUP1(arVUP1);
  float arW11[3] = {-2,-2,0};
  Point<3, float> w11(arW11);
  float arW21[3] = {2,2,0};
  Point<3, float> w21(arW21);
  float B1 = -10;
  float F1 = -1;
  Camera<float> C1(VRP1,PRP1,VPN1,VUP1,w11,w21,B1,F1);

  // then we make the other variables
  float pointAr1[3] = {4.0,3.0,2.0};
  float normalAr1[3] = {0.1,0.3,0.2};
APPENDIX B. SOURCE FILES

430 Point<3, float> p1(pointAr1);
431 Vector<3, float> normal1(normalAr1);
432 normal1.normalize();
433 // calling the illumination function
434 Color illum1 = lightSource1.PhongIllumination(material1, p1, normal1, C1);
435 // illum1.print();
436 Color illumExp1(0.021147, 0.057471, 0.108973);
437 assert(illum1 == illumExp1);
438 // a new test to make sure R*V is positive
439 float normalAr2[3] = {-7, 8, 5};
440 Vector<3, float> normal2(normalAr2);
441 normal2.normalize();
442 Color illum2 = lightSource1.PhongIllumination(material1, p1, normal2, C1);
443 // illum2.print();
444 Color illumExp2(0.02109, 0.056887, 0.107384);
445 assert(illum2 == illumExp2);
446 std::cout << "Illumination calculated as expected" << std::endl;
447 return 0;
448 }
Bibliography

