Exercises Computer Graphics

Prof. Dr.sc.hum. Hans-Heino Ehricke

Exercise: Visibility and Shading

In this exercise visibility and shading algorithms are used in order to display a teapot (a standard model in computer graphics). The 3D display is optimized step by step by removal of hidden lines/surfaces, application of a simple lighting mechanism and shading of polygonal surfaces with different techniques. Although all the algorithms are available as methods of graphics libraries, such as OpenGL, and thus are ready for direct usage, we will implement them for ourselfs with C++.

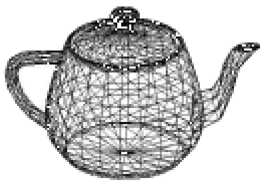

Fig. 1: Wire frame display of the teapot.

The wireframe display of fig.1 shows the polygonal edges of the model. It is not very realistic, for two reasons:

1. Objects in our real world usually have non-transparent surfaces. Thus, in a three-dimensional projection image back surfaces have to be eliminated. If in a wireframe display all polygonal edges are drawn, the viewer sometimes cannot distinguish between front and back parts of the object (fig 2).

   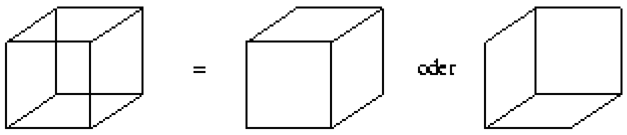

   Fig. 2: Interpretation problems with wireframe displays.

2. The model of the object is a polygonal approximation of the real curved surface. For the teapot triangles have been used to approximate the surface. In order to generate a realistic display of the model, not triangles, but a continuosly shaded surface must be drawn. This can be achieved with different shading techniques, such as Gouraud and Phong shading.

Backface Culling

Backface culling is a very simple, but efficient technique for hidden line/surface removal. When an object has been modelled with surface patches, such as triangles, their normals may be used for the visibility decision. As a precondition, normals pointing outwards have to be determined. Triangles with normals pointing to the back (not to the viewer) are not visible. The decision is made by analysis of the angle between normal and projection ray.

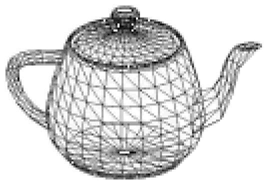

Fig. 3: Result of backface culling.

As can be seen in fig. 3 backface culling is not a perfect visibility technique. Only in objects with a simple convex shape, all non-visible surface patches are eliminated. In the case of non-convex objects or scenes with many objects, problems may occur (see fig. 4). At any rate backface culling may be used to reduce the number of polygons which have to be considered for the following computation steps.

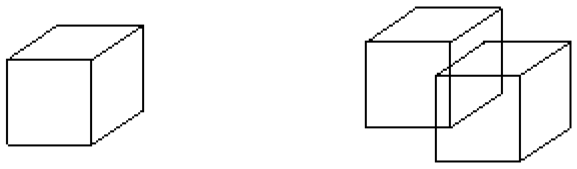

Fig 4: Result of backface culling in a scene with 1 and 2 objects.

In order to understand the details of the backface culling algorithm, let's look at fig. 5:

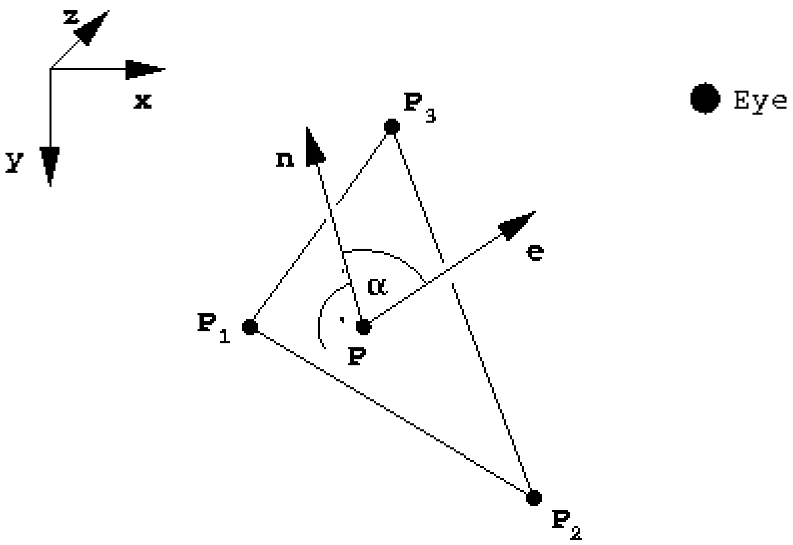

Fig. 5: Triangle with normal **n** in point **P**.

If the triangle $\mathbf{P_1P_2P_3}$ is visible from the position of the viewer *Eye*, then the following is true: $\alpha \leq 90°$ or $\cos \alpha \geq 0$. All triangles with $\alpha > 90°$ or $\cos \alpha < 0$ are not visible and must not be drawn.

Since $\cos \alpha$ is the scalar product of the normalized vectors **n** and **e** and **n** is calculated as the cross product of two edge vectors of the triangle, we may perform the following calculations in object space (the points **P, P_1, P_2** and **P_3** are considered as local position vectors **p, p_1, p_2** and **p_3**):

```
/* calculate normal n*/
n = (p2 - p1) × (p3 - p1);
Normalize (n);
/* choose p1 as reference point and compute viewer vector e */
p := p1;
e = Eye - p;
Normalize (e);
/* decide visibility of tiangle */
cos α := n · e;
if (cos α > 0)
     front
else back;
```

Shading

A lighting model is an instruction to compute the intensities of light emitted from one or several light sources and reflected at an object surface point towards the viewer. Frequently used lighting models in computergraphics are those of Lambert (diffuse reflection) or Blinn (ambient light, diffuse and specular reflection). **Shading** includes the application of a lighting model to object surfaces in a scene. It is obvious that a we can shade a surface by computation of the surface normal and application of the lighting model in any surface point. However, this brute-force mechanism is extremely inefficient and wastes a lot of computation power.

The three shading methods, we describe below, are much more efficient. However, they only approximate the correct calculation of the brute-force technique. Nevertheless, the quality of the results, especially of Phong and Gouraud shading is sufficient in most cases.

Flat Shading

Flat shading calculates for each triangle a single surface normal and applys the lighting model to it only once. The colour determined in this manner is used to fill the whole triangle. Flat shading is computationally inexpensive and thus very fast. However, it generates projection images, in which triangles (which are only approximations of curved surfaces) are still visible by intensity leaps at triangle edges (fig. 6). The teapot has originally been modelled with squares, which subsequently have been subdivided into triangles. Therefore two neighbouring triangles have identical shading values and you usually will see squares, not triangles.

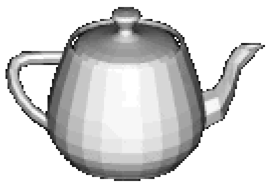

Fig. 6: Result of flat shading.

For the triangle filling procedure *scanconversion* is necessary. The main problem is to find those pixels on a scan line, which belong to the triangle (fig. 7).

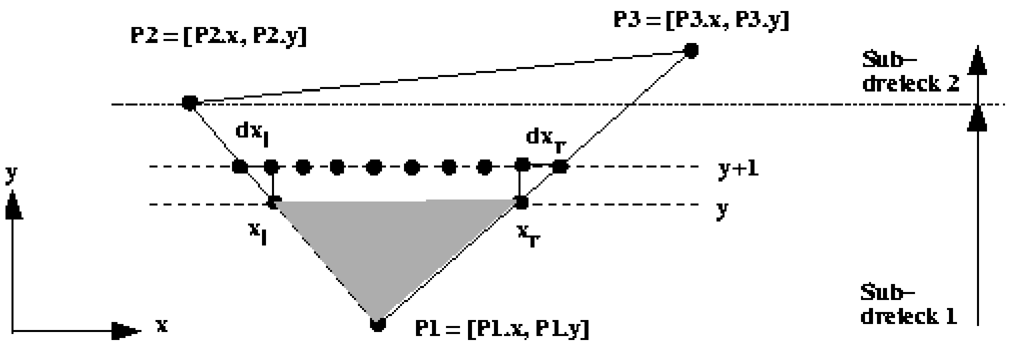

Bild 7: Scanconversion of a triangle.

In our C++ implementation framework the scanconversion routine is implemented in function **void Scan_Convert_Triangle2(ImagePoint P1, ImagePoint P2, ImagePoint P3)**. P1, P2 und P3 are the triangle vertices, as shown in fig. 7. A triangel is drawn scanline by scanline from bottom to top. For this purpose the triangle is split into two sub-triangles. In each sub-triangle the left and the right offset dx_l, dx_r of the scanline is calculated. These are used to calculate the left and right edge points when proceeding to the next scanline. The scanlines x_l - x_r are drawn.

```
sort vertices according to  ascending y coordinate→ P1.y ≤ P2.y ≤ P3.y;
$x_l$ := P1.x;
$dx_l$ := (P2.x - P1.x) / (P2.y - P1.y);
$x_r$ := P1.x;
$dx_r$ := (P3.x - P1.x) / (P3.y - P1.y);
/* draw sub-triangle between P1.y and P2.y */
for ScanLine = P1.y to P2.y do begin
    draw scanline from $x_l$ to $x_r$;
    $x_l$ += $dx_l$;
    $x_r$ += $dx_r$;
end
/* draw sub-triangle between P2.y and P3.y */
$dx_l$ := (P3.x - P2.x) / (P3.y - P2.y);
$x_l$ := P2.x + $dx_l$;
for ScanLine = P2.y to P3.y do begin
    draw scanline from $x_l$ to $x_r$;
    $x_l$ += $dx_l$;
    $x_r$ += $dx_r$;
end
```

In order to compute a colour for a triangle, we apply the very simple Lambert lighting model to the surface normal. The following steps are proposed:

1. Compute the triangle's surface normal **n**.
2. We assume identical positions for the light source and the viewer and calculate the projection/light direction vector as follows: *ProjDir* := *Eye* - $\mathbf{P_1}$.
3. Calculate the cos of the angle α between *ProjDir* and **n** as with backface culling; scale this value to a range between 0 and NumColors-1.

Gouraud Shading

Gouraud shading applies the lighting model to the three vertices of each triangle. The necessary three vertex normals are already included in the teapot database. The resulting three intensity values are used for bilinear interpolation of all shading values of the triangle surface.

In order to avoid intensity leaps at triangle edges, neigbouring triangles must have identical normal vectors in common vertices. Since vertex normals are usually calculated by the cross-product of two edge vectors, three triangles with a common vertex will usually yield three different vertex normals. By averaging a single vertex normal for all three triangles may be calculated. This has already been done for you, and you may use the vertex normals from the teapot database.

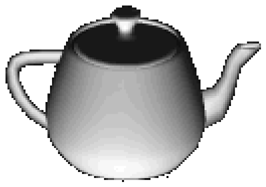

Fig. 8: Result of Gouraud shading.

Fig 9 and the algorithm below describe the mechanism of Gouraud shading, integrated with triangle scanconversion:

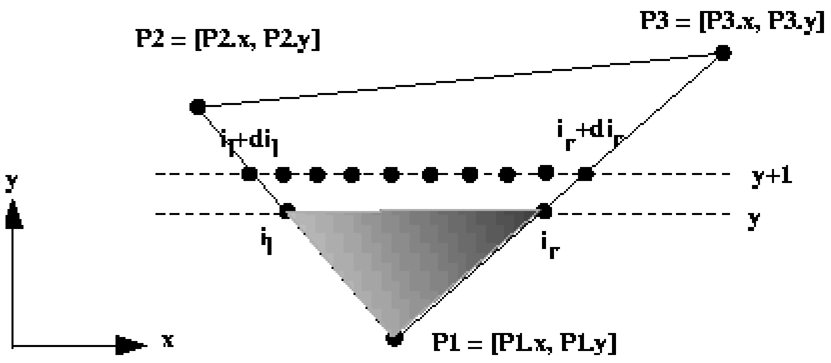

Fig. 9: Bilinear interpolation of shading values and scanconversion.

```
for each triangle of the model do begin
    apply lighting model to each of three vertices -> vertex shading
values;
    for each scanline do begin
        interpolate shading value at left and right triangle edge from
vertex values;
        for each pixel of the scanline do
            interpolate shading value from left and right edge values;
    end
end
```

The two inner loops of the above algorithm are an extension of the scanconversion algorithm for flat shading.

Phong Shading

Phong shading interpolates normal vectors for each point of the triangle surface from the vertex normals. At each point the lighting model is applied. It is obvious, that this is computationally more expensive then Gouraud shading. The main advantage is, that objects with specular reflection may be displayed in more realistic manner. The reason is, that specular highlights are reduced by intensity interpolation mechanisms.

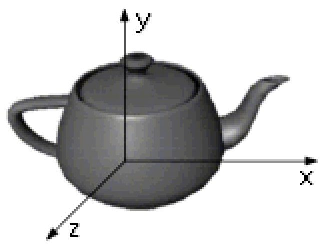

Fig. 9: Result of Phong shading with integrated coordinate system of the model.

The Teapot Model

In this exercise we use a simple teapot model, consisting of 1504 triangles. The database includes 3D coordinates and normal vectors for all triangle vertices. You find the data in **teapotdata.c**. Data structures are defined in **math3d.h** and **projection.h**:

```
typedef struct
    {
        int x, y;
    } ImagePoint ; /* 2D point in image space */

typedef struct
    {
        float x, y, z, w;
    } ObjectPoint ; /* point in homogenous coordinates */

typedef ObjectPoint Vector;

typedef struct
    {
        ObjectPoint p1, p2, p3;
        Vector n1, n2, n3;
    } Triangle3dType ; /* triangle in object space */

typedef struct
    {
        ImagePoint p1, p2, p3;
    } Triangle2dType ; /* projected triangle in image space */

#define NumTriangle 1504

extern TriangleType teapot3d[NumTriangle];
```

Each triangle is defined via three vertices p_1, p_2, p_3 and three vertex normals n_1, n_2, n_3 with length(n_i)=1.0. The data are defined with homogenous coordinates as local vectors with p_i.w = 1.0. For the vertex normals n_i.w = 0.0, that means, the normals are defined as direction vectors. Neighbouring triangles possess identical normals in their common vertices.

The transformation from 3D space to the 2D image plane (projection and viewing transformation) is predefined with static values for eyepoint *Eye*, viewing point *View* and viewup vector *ViewUp* (defines the upward direction of the viewing plane:

```
ObjectPoint Eye = { 10.0, 100.0, 200.0 };
ObjectPoint View = { 0.0, 0.0, 0.0 };
Vector ViewUp = { 0.0, 1.0, 0.0 };
```

Additionally, in **math3d.c** functions for vector and matrix handling have been implemented:

```
/* calculates vec := p2 - p1 fuer for each of x,y,z and sets vec.w = 0.0 */

extern void MakeVector(ObjectPoint p1, ObjectPoint p2, Vector *vec) ;

/* calculates vector cp as cross product of u, v.
   cp is orthogonal to u and v */

extern void CrossProd (Vector u, Vector v, Vector *cp) ;

/* calculates the scalar product of u, v.
   in the case of normalized vectors, the cos of the angle between u and v
is returned */

extern Scalar SkalProd (Vector u, Vector v) ;

/* normalizes vec; length (vec) = 1.0 */

extern void Normalize (Vector *vec) ;
```

Preparation

1. Reconsider your knowledge of hidden line/surface removal, lighting models and shading techniques.
2. Carefully analyze the source code made available in the framework of this exercise.
3. Answer the following questions:
   - What is the correlation between surface normal vector and visibility?
   - How can surface normal vectors be determined for a triangle?
   - How could be checked, whether a normal vector points outside of the modeled object?
   - Why do we have to subdivide a triangle into two sub-triangles during scanconversion and filling?
   - In Gouraud shading, how is the normal vector at a vertex with neighboring triangles computed?
   - Why are the triangle edges no longer visible in a Gouraud shaded display?
   - When is it worthwhile to use Phong shading?
   - Which are the three components of the Blinn-Phong lighting model and how are they calculated?

Exercises

1. WireFrame: Compile, link and run the project *teapot*. Analyze **teapot.cpp** and **wireframe.cpp**. Make yourself familiar with the use of teapot model data.
2. Backface culling: The function **Backface()** within *backface.cpp* treats sequentially all triangles of the model database, projects them onto the image plane and draws their edges. Add to this procedure the backface culling mechanism so, that only those triangles are projected and drawn, which are visible from the eyepoint *Eye*. Save your program to the file **backface.c**.
3. Flat shading: Implement flat shading together with Lambert lighting in the function **Flatshading()** within **flatshading.cpp**. Compute the necessary normal vectors by yourself.
4. Gouraud shading: Test your implementation of Gouraud shading with the program **trifill.cpp**, which draws a triangle with pre-defined intensity values at its vertices. Use the intensity values **i** and **di** of the points with type **ScanPoint**. Take care for special topology cases of the triangles to be filled.

5. Implement your Gouraud shading algorithm in the program **gouraudshading.cpp** within the function **Gouraudshading()** The vertex normal vectors may be extracted from the teapot database.
6. For Gouraud shading the lighting calculation is performed within the function **Intensitaet**. Extend this function by implementation of the specular reflection term. Calculate the necessary vectors and apply specular reflection parameters so that a metallic teapot is modeled. The reflection vector R can be calculated from the normal vector N and the vector to the light source L as follows:
   $R = 2 \cdot N \cdot (N \cdot L) - L$
7. (Optional) Implement the Phong shading mechanism. Apply a lighting model with parameters such, that a specular highlight on the teapot surface becomes visible.

Hints

Download and unpack the file teapot.zip. You will find all necessary files in the current directory.