

From Graphics to Visualization



Motivation

In both visualization and computer graphics, we take as input some data and produce an image that reflects several aspects of the input data.

Hence, there are similarities between visualization and computer graphics. As a simple example, we can visualize a function f(x, y) = z by simply plotting it. This then results in the classical solution, the **height plot**. The function f is sampled at various sample points, typically aligned in a grid pattern, and then plotted by using OpenGL triangles for example. (Note: while quads could be used OpenGL may not draw them if the vertices are not planar.)



Height plot

Height plot for the function $f(x, y) = e^{-(x^2 + y^2)}$ sampled at 30x30 points:





Height plot (continued)

Obviously, the sampling rate greatly influences the quality of the visualization. To low a sampling rate can result in a misrepresentation of the results. Hence, the sampling density must be proportional to the local frequency of the original continuous function that we want to approximate. According to the Nyquist theorem, the sampling rate must be at least twice as high as the signal's frequency. In practice, an adaptive sampling rate proportional to the curvature, i.e. second derivative, provides good results.



Height plot (continued)

Height plot for a sinusoidal function $g(x, y) = \sin\left(\frac{1}{x^2 + y^2}\right)$



RIGHT STATE De

Lighting

Using correct lighting improves the three-dimensional impression.





OpenGL allows you to specify locations for up to eight light sources. In addition, material properties can be specified that describe the characteristics of on object's surface, e.g. how shiny it is.

The framework provided for the assignments already defines a light source and sets reasonable material properties for your objects. If you want to change those you can use the OpenGL function glMaterial as described on the next slide. Keep in mind that changing the material properties often comes with a loss in rendering performance.



Material properties

Different kind of materials can be generated with regard to, for example, their shininess using glMaterial*:

```
GLfloat diffuse []
    = \{ 0.2, 0.4, 0.9, 1.0 \};
GLfloat specular []
    = \{ 1.0, 1.0, 1.0, 1.0 \};
glMaterialfv (GL FRONT AND BACK,
   GL AMBIENT AND DIFFUSE, diffuse);
glMaterialfv (GL FRONT AND BACK,
   GL SPECULAR, specular);
glMaterialf (GL FRONT AND BACK, GL SHININESS,
   25.0);
```



The lighting in OpenGL is then calculated based on the Phong Illumination model, i.e. OpenGL computes the reflection direction of the light to determine the specular and diffuse lighting.



1-9

Entire mathematical model in detail:



_E Departn

UNIVERSITY

Light sources

OpenGL supports up toe eight light source (GL_LIGHT0 **through** GL_LIGHT7**). To enable lighting you need to issue:**

```
glEnable (GL_LIGHTING)
```

Each light source can be enabled using, for example:

```
glEnable (GL LIGHTO)
```

Properties of light sources can be changed using the command:



Properties

Different properties are available:

Location:

GLfloat position [] = { 0.0, 0.0, 0.0 };

glLightfv (GL LIGHTO, GL POSITION, position);

Color:

GLfloat color [] = { 1.0, 1.0, 1.0 }; glLightfv (GL_LIGHT0, GL_AMBIENT, color); glLightfv (GL_LIGHT0, GL_DIFFUSE, color); glLightfv (GL_LIGHT0, GL_SPECULAR, color);



Shading model

In OpenGL, two shading models are available: flat shading and Gouraud shading (default)

You can specify which of the shading models to use by using the following function

glShadeModel (mode);

Where mode can assume the values GL_SMOOTH and GL FLAT.





Flat

Gouraud

Phong

Types of Shading

Flat, Gouraud and Phong shading are the three most common types of shading used on 3D objects. (Image courtesy of Intergraph Computer Systems.)



Since the Phong illumination model requires normal vectors for the object, these need to be specified. Typically, this is done per vertex so that the light intensity can be interpolated across a polygon (if there is only one normal vector per polygon, only flat shading is possible).



Normal vectors

Normal vectors can be provided by using the command glNormal*:

GLfloat normal $[] = \{ 1.0, 1.0, 1.0 \};$

```
GLfloat vertex [] = \{ 2.0, 1.0, 3.0 \};
```

glNormal3fv (normal);

```
glVertex3fv (vertex);
```

Make sure that the normal vector is provided **before** the vertex since OpenGL is a state machine!

If your normal vectors are not normalized OpenGL can do that for you if you issue:

```
glEnable (GL NORMALIZE);
```

Polygonal normal generation

Gouroud and Phong shading can improve the appearance of rendered polygons. Both techniques require point normals. Unfortunately, polygonal meshes do not always contain point normals, or data file formats may not support point normals. Examples include the marching cubes algorithm for general data sets which typically will not generate surface normals.



Example





Computing normal vectors

Ideally, the exact normal of the object should be used. However, these data may not be available so that we need to compute the normal vector ourselves. In order to compute the normal for a single vertex, all adjacent polygons need to be considered. The average normal of the normal vectors of all those polygons is then used as the normal vector for the vertex.

Finding all adjacent polygons can be tedious and computationally intensive. It is easier and faster to just loop through the faces:



Computing normal vectors

- Create an array of normal vectors (same size as array of vertices)
- Loop through the polygons
 - Compute normal vector for the polygon
 - Add this normal vector to the current ones associated with all vertices of the current polygon
- Normalize normal vectors

This algorithm computes the normal vectors relatively fast. However, this naïve approach may fail in some cases:



Naïve approach that just averages normals between neighboring polygons can have unsatisfactory results:





First, neighboring polygons should be oriented in the same way. Otherwise normal may cancel each other out since they are facing in opposite directions, i.e. the scalar product is negative.

To orient edges consistently, we use a recursive neighbor traversal. An initial polygon is selected and marked "consistent". For each edge neighbor of the initial polygon, the ordering of the neighbor polygon points is checked – if not consistent, the ordering is reversed. The neighbor polygon is then marked "consistent." This process repeats recursively for each edge neighbor until all neighbors are marked "consistent."



Consistent polygons





Feature edges

A similar traversal method splits sharp edges. A sharp edge is an edge shared by two polygons whose normals vary by a used specified *feature angle*. The feature angle between two polygons is the angle between their normals. When sharp edges are encountered during the recursive traversal, the points along the edge are duplicated, effectively disconnecting the mesh along that edge. Then, when shared polygon normals are computed later in the process, contributions to the average normal across sharp edges is prevented.



Splitting of edges



a) Feature edge

b) Exaggerated splitting of feature edge



Result





Texture mapping

Extra realism can be added to rendered 3D models by using a technique called **texture mapping**. The rendering mode described so far assigns a single color to a each vertex of the polygon (for Gouraud and Phong shading). Such techniques are limited in conveying the large amount of small-scale detail present in real-life objects, such as surface ruggedness or fiber structure that are particular to various materials, such as wood, stone, brick, sand, or textiles. Texture mapping is an effective technique that can simulate a wide range of appearances of such materials on surface by mapping image data onto the surface.



To map the image (texture) onto the surface, a local coordinate system (texture coordinates) is defined on the surface which defines the mapping function:





Create the texture and copy it to the graphics memory:

Enable textures:

glEnable (GL_TEXTURE_2D);

If you use OpenGL prior to version 2.0 width and height have to be powers of two!

Texture coordinates

Provide a texture coordinate for every vertex of you polygonal mesh:

```
GLfloat texcoord = \{ 1.0, 1.0 \};
```

```
GLfloat vertex = { 2.0, 1.0, 3.0 };
```

glTexCoord2fv (texcoord);

```
glVertex3fv (vertex);
```

Provide the texture coordinate before the vertex!

When using vertex arrays, texture coordinates can also be provided as a single array:

GLfloat texcoordarray;

```
glEnableClientState (GL_TEXTURE_COORD_ARRAY);
```

Naming textures

If you use more than one texture you need to provide names in order to be able to switch between the provided textures.

```
GLuint texname;
```

```
glGenTextures (1, texname);
```

Then, you can change between them using these names:

glBindTextures (GL_TEXTURE_2D, texname); Remember, OpenGL is a state machine so it will use this texture from now on for every texture related commands!

Textures, just as colors, can also be used to visualize additional information as we will see later. For example, in order to visualize a 3D vector field, a surface can be created based on the vector field and a texture can be used to visualize the vector field locally.





Transparency and Blending

So far, we have used fully opaque shapes. In many cases, rendering half-transparent (translucent) shapes can add extra value to visualization. For instance, in our height plot, we may be interested in seeing both the gridded domain and the height plot in the same image and from any viewpoint. We can achieve this effect by first rendering the grid graphics, followed by rendering the height plot, as described previously, but using halftransparent primitives.



Transparency and Blending (continued)

In OpenGL, transparency-related effects can be achieved by using a special graphics mode called **blending**. Note: you have to request a visual that supports an alpha channel; for example, when using GLUT you need to initialize your display mode using something like this:

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

Then, we must enable blending using the following function call:

glEnable (GL_BLEND);

Transparency and Blending (continued)

We can define the way the color of a newly rendered polygon is combined with the one currently in the frame buffer by defining the blending function:

The colors then typically are specified as four instead of just three components where the last coefficient describes the transparency:

```
glColor4f (0.3, 0.3, 0.3, 0.5);
```

Beware: the depth buffer OpenGL uses for solving the visibility problem does not handle transparency.



Transparency and Blending (continued)

The height plot with transparency to make the underlying sample grid more visible can then look like the following image:



