Computer graphics

Linus Falk

March 8, 2023

Contents

Lecture 1: Introduction monday 16 jan 10:15

1 Introduction to computer graphics

What can it be used for

- Entertainment
  - Games, pushing graphic card development forward..
- User interfaces
- Applications
- Data visualization
  - Information Visualization
    - Charts diagrams etc
    - Plots of N-d data after dimensionality
  - Data visualization
    - ...
- Image processing
  - Extracts information
  - produces outputs like: classifications and segmented objects
- Computer vision
  - feature tracking
  - Tracking
  - **3D reconstruction**
- Photogrammetry?
  - Reconstruct 3D models from photos taken from different viewpoints. How do the different points align. Use **CG** to view result.
- Cultural heritage?
  - preserve art works, 3D scanning and other techniques.
  - Display the result for the audience with **CG**
- CAD
  - 3D modeling
- much much more..

Digital images

Gray scale image a (raster graphics) is stored as a matrix with intensity representing each pixel.A modern computer monitor got around 2660x1600 pixels, or picture elements. Bit depth describes how many bits that are used to represent the intensity. "True color" is 24 bit, 8 bit per color channel. A colored image is represented by three color images Red Green and Blue, some times an additional channel **alpha** for storing opacity.

High dynamic range HDR images

8 bits per color channel often sufficient. In many computer graphics applications a higher dynamic range is needed. It is therefore common to use more bits per color channel, e.g. 16 or 32 bit floating point values:

- When images are manipulated, to avoid artifacts.
- When using image data in the purpose of measuring light information from a scene to realistically integrate synthetic objects

Image based lightning with high dynamic range images of real world environment.

useful to create a realistic illuminations.

To capturing a the great contrast between sky and the ground, 8 bit depth is not usually enough.

Computer graphics history

University of Utah prominent sections that did a lot of work in this are during 70s. Many methods invented by people from this sections, Sutherland, Blinn, Phong, Guoraud, Catmull (Turing awards winner), Newell and others. During the 80s we got Graphical user interfaces, Mac, Amiga and graphics in movies.

- Utah teapot
- Stanford bunny

In the 90s fully animated CG movies and special effects. 00s Rise of the GPU's, start to be able to program and do custom things on GPUs. A move toward physically based rendering, start to use more correct models of illumination thanks to better computational resources. 10s saw fusion with Image processing and computer vision.. Film rendering to **path tracing**. Real-time **ray tracing**. 20s machine learning?? this would mean new principles. Neuro .... read more of own interest

Quick look at what we will learn

- Transformation
  - Affine transformations in homogeneous coordinates
  - Orthographic and perspective projections for cameras.
- Shading
  - Going from geometry to what color to put on a surface. A more general term than illumination

  - Gouraud shading (computed per-vertex)
  - Phong cont...
- Illumination
  - The Phong reflection model
  - Blinn-Phong
- Texure mapping
  - Texture mapping
  - Bump mapping
  - Environment mapping
  - Geometry
  - Normal
- The programmable **Graphics pipeline**

Pipeline

- Input are the objects and their vertices
- The output are the pixels on the screen
- Performs:
  - transformations
  - clipping and assembly
  - shading, texturing and illumination

Conclusion for this lecture

Study the slides but also the other recourse. Prepare for the practicals (assignment and project) and start to play around with graphics programming.

Lecture 2: Graphics programming tuesday 17 jan 10:15

2 Graphics programming

Typically deals with how to define 3D scene, with virtual camera and how to create a 2D projection of the 3D scene

Real time vs offline rendering

Real time used in games, scene must be updated 30-60 fps here is speed > image quality, but today hardware can do pretty much both. **offline** rendering is used for animated movies and visual effects and is not meant to be used interactively. In this field rendering of a single frame can take several hours. Image quality > speed in this case. Movie production uses clusters to calculate the renders in parallel.

How do we draw

Version 1, traditionally used mainly in offline rendering

Version 1

```
For every pixel on screen
        Query all objects to be drawn,
        to determine the correct color for that
           pixel "Ray tracing"
end
```

Version 2, most common method for real time rendering

Version 2

```
For every object
        Draw the object to the screen,
        by setting the correct colors for the
           affected pixels
end
```

Representation

Polygonal meshes are typically used for representing 3D models, using collections of vertices, edges and faces. Faces can be arbitrary polygons but we typically split them in triangles so the implementation gets simpler.

Vertex data

Each vertex in the polygonal mesh has one or several attributes such as:

- Position
- Color
- Normal vector
- texture coordinate

The vertex data is typically loaded on the CPU and uploaded memory that is accessible from the GPU via **buffer objects**.

Transforms

Transforms are used to manipulate positions orientation, size and shape of object in the 2D/3D scene. Transforms are also used to define a virtual camera and go from one coordinate system to another. Theses transformations are often represented as 3 by 3 or 4 by 4 matrices. Examples of some basic transforms:

- Translation
- Rotation
- Scaling

Coordinate systems

The OpenGL pipeline has 6 different coordinate systems

- Object
- World
- Eye (or camera)
- Clip
- Normalized devices
- Window (or screen)

Surface normal

A normal vector is a perpendicular vector to the surface that points outwards from the surface. It describes a local orientation of a surface at some vertex or face. Super important to lighting. Useful to know if the surface is pointing towards me or not

Can visualize it with RGB vector for example.

Shaders

In real time rendering is shader a small program that is compiled on the CPU and executed on the GPU. Common use is to apply transformations on vertices and compute and compute the level of light and colors of the fragments/pixel candidates. Using different shader or shader input can drastically change the visual appearance of a 3D object.

The programmable graphics pipeline

create illustrator image

OpenGL

OpenGL is a cross-platform low level API for rendering of 2D and 3D graphics. It is state-based and maintain a currents state of things (what shader is being used etc). The first version was launched in 1992. Used in many applications such as: games, simulations, scientific visualizations, CAD, mobile applications, VR etc. OpenGL only handles rendering and not any input or windowing. It is callable form many programming languages such as: C, C++, Python, Java, C#, JS, Rust and more. It comes in many variations:

- OpenGL for desktop
- OpenGL ES for embedded systems
- WebGl for web browsing 3D graphics.

Alternatives to OpenGL are: Metal (MacOS), Vulkan and DirectX (Microsoft).

OpenGL functions are executed in the host CPU application and responsible for:

- Creating and initializing buffers, shaders, textures etc.
- Upload data to GPU accessible memory.
- Configure the render state.
- Submit drew calls.
- Clear and swapping buffers.

The CPU and GPU

We here present a couple of very simplified "algorithms" for drawing a object:

"Immediate mode" rendering

```
initialize_system();
for (each object to be drawn){

        o = generate_object();
        draw_object(o);
}
cleanup();
```

"Retained mode" rendering

```
initialize_system();
for (each object to be drawn){

        o = generate_object();
        store_object(o);
}

draw_all_objects();
cleanup();
```

"Retained mode" rendering

```
initialize_system();
for (each object to be drawn){

        o = generate_object();
        store_object(o);
}
send_object_to_GPU();
//Can be a bottleneck, if we want to draw the
   same object multiple times
// it is beneficial to only send them once to the
    GPU

draw_all_objects();
cleanup();
```

Utility libraries

Since OpenGL is only used for rendering, we need to use a variety of so called utility libraries in our applications. Here are the ones we will be using in the labs:

- GLFW create and manage window.
- GLEW extension loader.
- GLM mathematics library.
- ImGui GUI.

GLFW gives us an interface between the windowing system and graphics system and allows us to create a window, executes the rendering loop and setup a mouse and keyboard interaction. **GLEW** is a cross-platform C/C++ library loading OpenGL extensions. GLEW will search for and pull in all the OpenGL extensions that we need/supported, **GLM** is mathematics library for graphical programming and provides vector and matrix datatypes, transforms, quaternions, noise functions and much, much more. It is used in the host/CPU for C++ applications only. **ImGui** is a GUI library that is useful for tweaking rendering parameters.

Vertex buffer objects (VBOs)

VBO's are used for uploading arrays of vertex data/attributes to the GPU memory. Before we submit a draw call we must bind the VBOs. We can split our VBOs into groups of each vertex attribute or interleave several vertex attributes in one single VBO (this us usually more efficient).

Vertex array objects (VAOs)

A VAO stores references to one or several VBOs along with their states and configurations. At drawing we only have to bind the VAO, it simplifies the code so we don't have to bind and configure the VBOs separately at each draw call. Recent OpenGL core profiles require the use of VAOs

glDraqArrays and glDrawElements

These are draw commands for rendering graphics primitives: lines, points and triangles from array data stored in VBOs or VAOs.

Example:

```
void drawTriangle(GLuint program, GLuint vao)
{
  glUseProgram(program);

  glBindVertexArray(vao);
  glDrawArrays(GL_TRIANGLES, 0, 3);
  glBindVertexArray(0);
  glUseProgram(0);
}
```

OpenGL primitives

The following primitives are the basic building blocks in OpenGL applications:

- Point sprites: GL_POINTS
- Lines: GL_LINES, GL_LINE_STRIP, GL_LINE_LOOP
- Triangles: GL_TRIANGLE, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN

GLSL OpenGL shading language

GLSL is a high-level, cross-platform shading language for real-time rendering. It is based on the C-programming language and uses similar naming conventions,

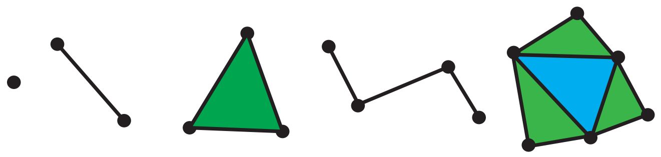

Figure 1: Example of caption

data-types and control structures. Following data-types and functions are built in:

- vector and matrix types
- various math and utility functions for graphics programming, dot, cross, max, normalize etc.
- texture lookup functions

Shader types

OpenGL support six different types of shaders. Vertex and fragment shaders are covered in this course.

- **Vertex**
- **Fragment**
- Geometry
- Tesselation control
- Tesselation evaluation
- Compute

The vertex and fragment shader

An example of the task the vertex shader have is to apply color and transformation on the input vertices and pass varying data the the fragment shader.

The fragment shader takes **uniforms** and interpolated data from the vertex shader and rasterize as input. It then computes the final color of fragments called pixel candidates by evaluating lightning equations.

The GLSL source code is typically stored in plain ASCII text files with the suffixes *.vert for vertex shaders and *.frag for fragment shaders. When the program starts the host application loads the GLSL source files into strings and then compile them into shaders that can be executed on the GPU.

Workflow - Creating, compiling and linking

The main steps in Creating, compiling and linking GLSL shaders are listed here:

1. Read vertex and framgent shaders source files into strings
2. Create vertex shader object from the vertex shader source string

3. Create fragment shader object from the fragment shader string
4. Compile the program
5. Link the compiled program and check for errors
6. De-attach and delete the shader object

Example: GLSL vertex shader

```
#version 330 // specifies the GLSL version
in vec4 a_position; // input vertex position
void main() {
  // just sets the output vertex position
  // to the input vertex position
  gl_Position = a_position;
}
50
GLSL fragment shader
(triangle.frag)
#version 330
out vec4 frag_color; // output fragment color
void main() {
  // sets the output fragment color to white
  frag_color = vec4(1.0, 1.0, 1.0, 1.0);
}
```

The variables types that are used for communicating with shaders are of three types:

- Attribute
- Varying
- Uniform

Attribute variables can be accessed via the **in** qualifier. **Varying variables** provides an interface for passing colors, normals, texture coordinates and other data between the vertex shader and the fragment shader. Varying data is be default linearly interpolated over the geometric primitive. The vertex shader uses the **out** qualifier to pass varying data the fragment shader. The fragment shader accesses the data via the **in** qualifier and writes an output via the **out** qualifier. **Uniform variables** are used for data that should be constant for all vertices and fragments. Examples of such data are: transforms, material properties (color, opacity, etc), light sources, texture samplers, time and flags for enabling/disabling parts of the shading. **Uniform** variables can be accessed in both the vertex and fragment shader via the **uniform** qualifier.

Transforms in shader based OpenGL

Typically is the transform constructed in the host C++ program using GLM and passed to the vertex shader as uniform variables. On the GPU side, the vertex shader applies the transforms on the incoming vertex data.

3 Transformations

This section covers following:

- Obejct representation in computer graphics
- Some review of linear algebra
- Linear transformations such as: rotation and scaling with matrices
- Translation and homogeneous coordinates
- A lot of transforms can be expressed with matrix multiplications

Computer graphics pipeline image here!

Object representations for computer graphics

In computer graphics objects is described by its surface, this means that they are hollow. These objects are specified by a sets of 3D points, so called vertices. An object can be approximated by a multiple convex polygons. We can transform the object by transforming these vertices. We can approximate objects with small polygons, but if each polygon is smaller or in same size of a pixel there is no visual difference. Modern graphics hardware have the capability to render a lot of polygons fast.

Other object representations

Objects can also be expressed as a solid with Voxels = 3D pixels. An object can be described implicitly with coordinates and functions: $F(x, y, z) = C$.

Math: linear algebra review

Scalars are real or complex numbers, real number most often used. Points are locations in space and don't have any size or shape while vectors are directions in space with a magnitude but don't have any position. Review of vector operations:

- Addition and subtraction
- Multiplication by scalar
- Magnitude, Euclidian norm $\mathbf{a} = (a_x, a_y), ||\mathbf{a}|| = \sqrt{(a_x, a_y)} = \sqrt{\mathbf{a} \cdot \mathbf{a}}$
- Normalizing $\hat{\mathbf{a}} = \frac{\mathbf{a}}{||\mathbf{a}||}$ look out for division by zero!!!

Matrix multiplication:

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} ax & by & cz \\ dx & ey & fz \\ gx & hy & iz \end{bmatrix} \tag{1}$$

Linear combinations of vectors

Convex combinations, that are coefficients are all positive and add up up to 1. For example linear interpolation: $\mathbf{p}(t) = (t)\mathbf{a} + (1 - t)\mathbf{b}$, where $\mathbf{a}$ and $\mathbf{b}$ are points and t: $0 \le t \le 1$.

Dot product: $\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \ldots + a_n b_n$. Notice that vectors are orthogonal when the dot product is equal to zero and that vectors are less than 90 degrees apart if the dot product is positive. The dot product can be used to calculate the perpendicular **projection** of a vector onto another vector: $\mathbf{c} = (\mathbf{a} \cdot \mathbf{b})\mathbf{c}$. It can also be used for the **reflection** of a vector over the normal vector $\hat{\mathbf{n}} : \ b = -a + 2(\mathbf{a} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}}$

The **cross product** with $u = (u_x, u_y, u_z), v = (v_x, v_y, v_z)$:

$$u \times v = \begin{vmatrix} \mathbf{x} & \mathbf{y} & \mathbf{z} \\ u_x & u_y & u_z \\ v_x & v_y & v_z \end{vmatrix} = (u_y v_z - u_z v_y)\mathbf{x} - (u_x v_z - u_z v_x)\mathbf{y} + (u_x v_y - u_y v_x)\mathbf{z} \quad (2)$$

The cross product $u \times v$ is perpendicular to both $\mathbf{u}$ and $\mathbf{v}$ and the orientation of it follows the right hand rule. The norm : $||\mathbf{u} \times \mathbf{v}|| = ||\mathbf{u}||||\mathbf{v}|| \sin\theta$

Transformations

In order to create and move objects we need to be able to transform the objects in different ways. Transformations are divided into many classes:

- Translation (move the object around)
- Rotation
- Scaling
- Shear
- Mirroring/Flipping
- ...

Translation is simply adding a constant to all points. $x' = x + \mathrm{d}x$ and $y' = y + \mathrm{d}y$. **Scaling** an object by making it either smaller or bigger by a constant scaling factor. Here we multiply each point of the object with the scaling factor: $x' = s_x x$ and $y' = s_y y$. **Rotation** (about the origin) is easist described with the rotation matrix:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \quad (3)$$

The process goes: translate the rotation centre to the origin, rotate and then translate back. **Translation** looks different in matrix form, here we need to introduce an extra dimension.

$$\begin{pmatrix} x' \\ y' \\ W \end{pmatrix} = \begin{pmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \quad (4)$$

Note, if W = 0 then the point is not affected by translations. Using this extra dimension for the other translations also is called: Using homogenous coordninates:

- Translation

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \tag{5}$$

- Rotation

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \tag{6}$$

- Scaling

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \tag{7}$$

The order of the matrices is important:

$$P' = T^{-1}(S(R(T(P)))) = (T^{-1}SRT)P \tag{8}$$

When moving in to 3D are translation and scaling basically the same as in 2D. Rotation becomes a bit more comlicated: Rx,Ry and Rz. Obersve also that RxRy $\neq$ RyRx.

4 3D viewing and projection

This lecture will cover rotation around arbritary axis, the "view" coordinate system, change of coordinate system, change of frame (change of coordinate system + translation), how to postion a camera and projections (orthogonal and perspective).

Rotation around an arbitary axis

For rotation around an arbritary axis $\mathbf{v}$, we need to find a matrix $\mathbf{M}$ that aligns (1,0,0) with $\mathbf{v}$, then apply $\mathbf{M}^{-1}$, apply rotation and $\mathbf{M}$ to the object. Here $\mathbf{M}$ is the change of coordinate system. But how do we find $\mathbf{M}$?

Change of coordinate system

The goal of change of coordinate system is to express a point with two different sets of basis vectors:

$$\mathbf{P} = x\mathbf{e}_1 + y\mathbf{e}_2 = x'\mathbf{f}_1 + y'\mathbf{f}_2 \tag{9}$$

If we assume to know the basis vectors of the new coordinate system $\mathbf{f}$, in terms of the old coordinate system $\mathbf{e}$.

$$\begin{aligned} \mathbf{f}_1 &= a\mathbf{e}_1 + b\mathbf{e}_2 \\ \mathbf{f}_2 &= c\mathbf{e}_1 + d\mathbf{e}_2 \end{aligned} \tag{10}$$

Applying this gives us:

$$\mathbf{P} = x\mathbf{e}_1 + y\mathbf{e}_2 = x'\mathbf{f}_1 + y'\mathbf{f}_2 = x'(a\mathbf{e}_1 + b\mathbf{e}_2) + y'(c\mathbf{e}_1 + d\mathbf{e}_2) \tag{11}$$

Coordinates in $\mathbf{f}$ transferred to $\mathbf{e}$:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a & c \\ b & d \end{bmatrix} \begin{bmatrix} x' \\ y' \end{bmatrix} = \mathbf{M} \begin{bmatrix} x' \\ y' \end{bmatrix} \tag{12}$$

If we want coordinates in $\mathbf{e}$ transferred to $\mathbf{f}$:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & c \\ b & d \end{bmatrix}^{-1} \begin{bmatrix} x \\ y \end{bmatrix} = \mathbf{M}^{-1} \begin{bmatrix} x \\ y \end{bmatrix} \tag{13}$$

$\mathbf{M}$ is a pure rotation matrix if and only if the basis are Ortho-Normal (ON)-bases. In that case is $\mathbf{M}$ orthogonal and $\mathbf{M}^{-1} = \mathbf{M}^T$. An orthogonal matrix is a matrix where the rows and columns are mutually orthogonal unit-length vectors ?. Some properties of two orthogonal matrices are: the product is always orthogonal, always invertible and the inverse of an orthogonal matrix is equal to the **transpose** of the matrix.

How do we construct this ON basis if we only have one vector $\mathbf{v}$

$$\begin{aligned} v_1 &= \frac{\mathbf{v}}{|\mathbf{v}|} \\ v_2 &= \frac{v_1 \times v'}{|v_1 \times v'|} \\ v_3 &= v_1 \times v_2 \end{aligned} \tag{14}$$

How do we find a vector v' that is not parallel to v_1? We could just pick a random vector and check if the norm of the cross product is non zero. The problem with that is the risk of numerical error if the cross product is close to 0. A better way is to pick two orthogonal vectors $\mathbf{u}_1$ (1,0,0) and $\mathbf{u}_2$ (0,1,0). If $|\mathbf{u}_1 \times \mathbf{v}_1| > |\mathbf{u}_2 \times \mathbf{v}_1|$, set $\mathbf{v}' = \mathbf{u}_1$, otherwise $\mathbf{v}' = \mathbf{u}_2$.

We now have rotation about an arbitrary axis $\mathbf{v}$:

- Construct an **ON**-basis where $\mathbf{v}_1$=$\mathbf{v}$ is the first basis vector,
- Construct a corresponding "change of coordinates" matrix $\mathbf{M}$, which align (1,0,0) with $\mathbf{v}$
- Apply $\mathbf{M}^{-1}$ to object (transfering coordinates to a coordinate system where $\mathbf{v}$ is aligned with (1,0,0))
- Rotate around (1,0,0)
- Apply $\mathbf{M}$ (transfers coordinates back to the original coordinate system)

Some coordinate systems

Here follows some examples of coordinate systems in the pipeline:

$$p' = V_p * P * C * V_t * M * p \tag{15}$$

The **view transform** v$_t$ puts the observer at the origin and align x and y axes with the ones of the screen. Uses a right hand coordinate system so the z-axis is pointing backwards. This simplifies the clipping, light perspective and HSR (?). The view transformation is a so called change of frame, a change of origin + a change of coordinate system. This can be split into a translation and a change of coordinate system :

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \mathbf{M}^{-1}\mathbf{T}(-P_0) \begin{bmatrix} x \\ y \end{bmatrix} \tag{16}$$

glm::lookAt creates a viewing matrix that is derived from an eye point indicating the center of the scene and an UP vector. This build the 4x4 matrix for you but what is described above is what going on behind the scenes.

The **projection** p. Two types of projections: **Perspective** and **Orthographic** projection. In the orthographic projection is the "center of projection" at infinity and the projected points are along the direction of projection DOP. This projection is easy to implement since we only need to set z = 0.

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{17}$$

Perspective projection is a linear equation:

$$\begin{aligned} &(1) \quad \mathbf{p}' = t\mathbf{p} \\ &(2) \quad d = tp_z \end{aligned} \tag{18}$$

if we solve for (2) for t, we get:

$$\mathbf{p}' = \frac{d}{p_z}\mathbf{p} \tag{19}$$

Here we use W to fit the perspective transformation into a matrix multiplication:

$$\begin{aligned} x' &= x\frac{d}{z} \\ y' &= y\frac{d}{z} \\ z' &= z\frac{d}{z} \end{aligned} \tag{20}$$

$$\begin{bmatrix} x \\ y \\ z \\ \frac{z}{d} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{d} & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{21}$$

We divide by the homogeneous coordinate W to obtain the final projection, not that this is not a linear transformation.

glm::perspective(fovy, aspect, near, far) can create the projection matrix.

5 Shading and illumination

How can we make an object appear with shading and reflections? This lecture will cover: Phong illumination model, illumination, light sources, shading (Phong and Gaurad) and Mach bands.

Phong illumination model

The illumination can be described with the following function:

$$I = \frac{1}{a + bd + cd^2}(K_a L_a + K_d L_d \max(\mathbf{N} \cdot \mathbf{L}, 0) + K_s L_s (\mathbf{R} \cdot \mathbf{V})^{\alpha}) \tag{22}$$

Which translates into the sum: Ambient + Diffuse + Specular = Phong Reflection.

Illumination

This term is most often used to describe the process by which the amount of light reaching a surface is determined.

- Colour
- Light sources
- The Phong illumination model
- Shading algorithms
- Normal Computations
- Tangent vectors

An object has flat polygons but can appear smooth thanks to the process of called: **shading** (Phong/Gouraud). We can compare this shading to how we sketch something (drawing).

Shading

Shading is the process of setting the colour values for each of the pixels in the triangles (polygons). The colour depends on the light source (colour and intensity) and material (colour and reflectance properties).

Ambient light

Since the light model is local we do not take into account any bouncing light from the surroundings (which raytracing does), only the direct light from the light source. So the ambient light is approximated using only a constant instead. In other words, the shadowed areas have some light from reflection in the real world that is now approximated with our ambient light constant.

Point lights

The light originates from one single points and is spread equally in all directions to its surroundings. Examples of such light sources are : light bulbs and candles. In this model the light source/point have zero size, so it is only an approximation. We could think of the stars as such light sources but not the sun in the real world (In terms of size).

Parallel light

In the case of parallel light or directional light are the rays parallel, which is the case when the light source is infinitely far away. We can make this approximation, that the light is parallel from the sun (infinity far away)

Spot lights

These light sources have direction and so they also have a **limited width** which results in a spot of light on the "target".

$$C(\phi) = \max(-\mathbf{R} \cdot \mathbf{L}, 0)^P \tag{23}$$

C makes a smooth spot and P changes the size of the spot.

Distance

Light intensity changes over distance (decreases). If we have a distance d, the intensity can be assumed/approximated to be proportional to:

$$\frac{1}{a + bd + cd^2} \quad d = ||\mathbf{d}|| \tag{24}$$

This is not physically correct!

The Phong illumination model

Proposed by Bui Tuong Phong in 1975. A 3D object can be shaded using 3 types of light added together: Ambient, diffuse and specular. They can be computed separately and then added together.

Gauraud

Henri Gouraud hade earlier (1971) proposed that a 3D object can be shaded using the law of cosines (**Lambert's law**). From this comes the name lambertian surfaces. This type of surface is totally matte and doesn't reflect any light, only diffuses it. It scatters light in all directions equally so that no mirror reflection can be seen. Examples of "quite" matte surfaces are:

- Plaster walls
- Skin
- Paper

- Wood

Specular and semi-specular surfaces

The optimal specular surface is a mirror that reflects ray of lights, incoming angle = outgoing angle. Some materials are semi-specular meaning that the mirror reflection is a bit blurry, some of the light is diffused, not bouncing in perfect angle from the object. These kind of materials was the goal to model with the Phong illumination model.

Laws of cosine and trigonometry

Lambert's law of cosine says that when the light is spread over a larger area the intensity decreases (the maximum is in the normal direction).

- A/Hypotenuse = $\cos(\theta)$
- The angle between **L** and **N** is equal to θ
- The intensity is inverse proportional to the area it is spread out over
- By definition $\cos(\theta)$ equals $\mathbf{N} \cdot \mathbf{L}$
  - Hence is the intensity proportional to $\cos(\theta)$

Specular light

In the specular light there are two vectors involved: **R** which is the reflection of **L** around **N** and **V** which is the direction of the viewer (sometimes denoted **E**). The shininess α is a constant that changes the size of the highlight. An example of the use of shininess is that: the smaller the highlight is, the more the shiny the surface appear to be.

Compute the reflection vector

The halfway vector

The halfway vector, denoted **H** is halfway between **L** and **V**. The vector halfway between **L** and **R** is always **N**, the definition!. Hence is the halfway vector **H** close to **N** when **R** is close to **V**. This can be used to calculate the speculare light.

$$\mathbf{H} = \frac{\mathbf{L} + \mathbf{V}}{||\mathbf{L} + \mathbf{V}||} \tag{25}$$

Shading again

OpenGL supported two types shading traditionally: **flat** and **Gouraud/smooth** shading. Flat shading is done per face normals and Gouraud per vertex normals. Both of them uses the Phong illumination model!

Gouraud and Phong shading

Both shading models, Phong and Gouraud typically apply the Phong illumination model. The difference between them lies in where and where the illumination model is applied. The **normals** are typically known at the vertices of the polygons, these are sued to apply the Phong illumination model to the **vertices**.This results in a colour at each **vertex**. After that is done there is two ways to proceed:

BUT first we need to calculate these normals. We can compute them using the cross product of the points given:

$$\mathbf{N} = \frac{(\mathbf{P}_2 - \mathbf{P}_0) \times (\mathbf{P}_1 - \mathbf{P}_0)}{||(\mathbf{P}_2 - \mathbf{P}_0) \times (\mathbf{P}_1 - \mathbf{P}_0)||} \tag{26}$$

Gouraud also introduced a method but there are many possible approaches. **Weighted average** is one of the most used ways to compute vertex normals by using the average of the normals of faces which are adjacent to the vertex.

$$\mathbf{n} = \sum_{i=1}^{n} \mathbf{n}_i ||\mathbf{e}_i|| ||\mathbf{e}_{i+1}|| \sin(\theta_i) = \sum_{i=1}^{n} \mathbf{n}_i ||\mathbf{e}_i \times \mathbf{e}_{i+1}|| = \sum_{i=1}^{n} \mathbf{e}_i \times \mathbf{e}_{i+1} \tag{27}$$

Could be compared to the dot product in shading.

Now when we have the normals we can continue with the first method: **Gouraud shading** which will take the colours at the vertices, which we got from the illumination and interpolate these colours across the edge of the polygon and across the **scan lines**, typically is a bi-linear interpolation used.

The second method: **Phong shading** takes the normals at the vertices and interpolate these across the edges of the polygon and across the scan lines. Then is the illumination model applied to each pixel on the scan line, using that normal. This is more accurate way of shading since the illumination model is applied to each pint on that polygon, instead of interpolating the colours at the vertices, as in the previous method. With few polygons its possible that the **Gouraud** model will miss highlight when they end up somewhere between two vertices.

Mach bands

Mach bands is an illusion that consists of light or dark stripes that are perceived next to the boundary between two region that have different lightness gradients.

6 Shading with shaders

This lectures topics are: shading in modern OpenGL, per-vertex shading, per-fragment shading, the Lambertian reflectance model, Blinn-phong shading and some more advanced lightning techniques. Starting of with a bit of recap. The host CPU application loads/creates vertex data, shaders, texture and more, uploads them to the GPU accessible memory and submit draw calls. It's the shaders that do all the visual magic. Vertices and fragments are processed in parallel on the GPU.

Shading in modern OpenGL

The lightning computations are performed on the GPU by GLSL shaders. On the CPU are transforms, material properties and light sources defined and passed on to the shaders as **uniform** variables.

Lightnings vs shading

Lightning models describe the interaction between light sources and materials while shading is the process of computing the color of a pixel.

Vectors

For computing the lightning contribution a point **p** on a surface we will need:

- **N**, the normal vector at the point **p**
- **V** or **E**, the vector pointing to the viewer
- **L**, a vector pointing to the light source from point **p**

All these vectors are typically defined in the so called - view space. **Surface normals** are vectors that the describes the surface orientation at some vertex or face, can be uploaded to the GPU memory along with vertex position. **Material properties** like diffuse color and specular color can be passed as vectors as uniform variables (or stored in textures). **Light sources**: directional, positional, spotlight and area lights are also passed to the shaders as uniform variables.

Per-vertex shading

The lightning is computed in the vertex shader and interpolated over the triangle. The fragment shader only receives the interpolated result in this case and set it as fragment color. The **advantage** of this is that it's cheap to compute. The **disadvantage** is that it produces not good looking specular highlights for objects with a small amount of polygons.

Per-fragment shading

Here is lightning computed in the fragment shader, using interpolated normal, view and light vectors from the vertex shader as input. The **Advantage** is the resulting specular highlights will look better even if the number of polygons are small. It can be combined with techniques such as texture mapping to produce detailed and realistic surfaces. The **disadvantage** is more expensive to compute compared to per-vertex shading.

Gamma correction

Since LCD monitors have a non-linear response curve we need to correct the pixel values.

$$I_{corrected} = I^{\frac{1}{2.2}} \tag{28}$$

Anisotropic shading

Some materials doesnt scatter light evenly in all directions, examples of such materials are: hair, brushed steel, cloth and wood. The **Ward** anisotropic shading model uses two parameters: **ax** and **ay**, to control the the specular reflections.

Wrap shading

This method wraps the diffuse lightning towards the camera. This is to simulate subsurface scattering in the material. Material that behave like this are: wax, skin and marble. Can be implemented in the Lambertian diffuse term by adding wrap parameters to control the amount of wrap.

The Fresnel effect

Surfaces become more reflective at the grazing view angles (90 - incident angle = grazing angle), shallower angle, more reflection. Materials to be used on: water, glossy surfaces, wet asphalt etc. In GLSL can the Schlicks approximation be used, (computational cheap and commonly used).

Gloss and roughness

This is a common parameter used in physically-based materials. And since they are linear they are more intuitive to adjust than the specular power. The gloss factor = $1.0-$ roughnessFactor.

7 Rasterization and clipping

During rasterization are colour set in a scan line fashion.

Line drawing

Will cover some of 4 methods for line drawing:

- Digital Differential Analyser **DDA** (float)
- Implicit lines
- Bresenhamn (integer)
  - Steps on pixel centers using integer operation
  - 8 connected
- Parametric lines

Differential analyser DDA

A line in 2D is defined as the equation: y = kx + m, where x and y are variables, in this case screen coordinates. The line starts at m=(x_0, y_0) and ends at (x_1, y_1) and the slope is k = $\frac{\Delta y}{\Delta X}$. The algorithm starts at (x_0, y_0) increases x by one and y by k. This is repeated until $x = x_1$. Problem with vertical lines.

Bresenham

Developed the method in 1965. It is integer arithmetic (fast to compute, important at the time). It starts with f(x_0, y_0) = 0, the next pixel? choose (x+1,y) or (x+1,y+1), check the sign of f(x+1,y+1/2), depending on the sign set pixel above or below.

Polygon filling

Orientation of the polygon changes how we fill the triangle. If angle is greater then 0 we need to split the triangle in two smaller triangles.

$$\omega = (x_1 y_2 - x_2 y_1) \tag{29}$$

Line drawing is not as easy as one might think at first glance. The polygon filling is even harder. And then there is also the interpolation of colors or normals. Now day the GPU takes care of it! It also takes care of aliasing and fragments as well as prospectively correct interpolation

Hidden surface removal

In this chapter we take a look at clipping, hidden surface elimination and culling. **Clipping** is the process of clipping everything outside the view frustum. There are different types of actions associated with this process:

- Accept

- Reject
- Clip

With **normalization** clipping is done in a cube, that is easier than in the COP triangle. This is done with perspective division so we can use **parallel projection**. Scale the coordinates in the range ?.

Clipping in 2D

The clipping is usuaslly done in 2D where all polygons behind the camera are discarded, project on the clipping plane and clip polygons on the projection plane, in 2D. Another way is to clip in the frame buffer (called scissoring). Most of the methods in 2D can be extended to be used in 3D.

Algorithms

Here are some well known clipping algorithms listed:

- Cohen-Sutherland
- Liang-Barsky
- Sutherland-Hodgeman
- Weiler-Atherton
- Cyrus-Beck

We here take a look at Cohen-Sutherland in 2D for a line and then discuss the extensions to polygon clipping in 3D.

Cohen-Sutherland, we start with dividing the space into 9 regions and assign codes to them depending on their positions. The outcode o_1 = outcode(x_1, y_1) = (b_0, b_1, b_2, b_3) is assigned by:

$$\begin{gathered}
b_0 = \begin{cases} 1, & \text{if} y > y_{\text{max}}, \\ 0, & \text{otherwise} \end{cases} \\
b_1 = \begin{cases} 1, & \text{if} y < y_{\text{min}} \\ 0, & \text{otherwise} \end{cases} \\
b_2 = \begin{cases} 1, & \text{if} x > x_{\text{max}} \\ 0, & \text{otherwise} \end{cases} \\
b_3 = \begin{cases} 1, & \text{if} x < x_{\text{min}} \\ 0, & \text{otherwise} \end{cases}
\end{gathered} \tag{30}$$

Decisions are then made depending on the outcode:

- $o_1 = o_2 = 0$, both endpoints inside clipping window
- o_1 != 0 $o_2 = 0$, one endpoint inside, the other outside, line segment most be shortened
- o_1 & o_2 != 0, both outside, trivial reject

- o_1 & $o_2 = 0$, outside, but outside different edges, must investigate

Parametric lines

We can compute the intersections with the border of the clipping using the **two point formula**.

$$y = y_1 + \frac{y_2 - y_1}{x_2 - x_1}(x_{\max} - x_1) \tag{31}$$

Similar equations are obtained for the other borders. If we have no intersection with the view port, the parameters are out of range.

Hybrid approach

Using a 3D Cohen-Sutherland for trivial Reject and trivial accept, then project onto viewport and do final clipping in 2D (or using scissoring instead?)

Liang-Barsky

Uses the parametric line. Compute the angle α for each border in a clockwise order. Inside:

$$1 > \alpha_4 > \alpha_3 > \alpha_2 > \alpha_1 > 0 \tag{32}$$

Change of order will occur when outside. Similar equations can be derived for all possible cases. The clipping is done using these computed α's. In 3D, just add one dimension in the parametric line.

Sutherland-Hodgeman

This method is a pipeline clipper, going top, bottom, righ and left. Computing the intersections using the two-point forumula.

Summary of clipping

The previous explained approaches can be used for clipping polygons, but not that a triangle can have more vertices after clipping.

- Cohen-Sutherland - **Divide space and compute outcodes**
- Liang-Barsky - **Parametric line**
- Sutherland-Hodgeman - **Pipeline**

Hidden surface removal

Problem with many names:

- Hidden surface elimination
- Hidden surface determination

- Occlusion culling
- Visible surface determination

Defining the problem: The 3D world is projected onto a 2D screen. Which one of the polygons will be visible if they partly occupy the same pixels in the framebuffer? Well obviously the one that is closest.

Painters algorithm

This algorithm does like a painter, start with the background, then objects closer and closer. We can sort the primitives by Z, but how? the center of the polygon ?, then render from back to front. The problem here is that we dont have a constant Z for the primitives. Making it even harder there can be polygons that intersect. This **cannot** be handled by the painters algorithm (unless we clip polygons against each other).

The Z-buffer algorithm

The Z-buffer is very easy to implement but not perfect, suffers from precissions problem, manifests in flickering of texture for example. It is implemented on the GPU. Renders primitives in arbitrary order and no sorting is needed. The Z-buffer happens in the rasterization part of the pipeline.

The Z-buffer or depth buffer has the same resolution as the frame buffer, initiliazed to some value, range 0.0-1.0?. The algorithm: Record the depth value, z into the z-buffer while writing a pixel on the scanline, but, only writye the pixel if the z-value is less than previously recorded. The depth buffer can be used for other things like: compositing effect like: Fog, atmospheric scattering, etc. One problem with this method is that the precision depends on the range of the far and near clipping planes. The longer it is, the worse precision. This result in that algorithm cannot determine if the polygon is behind or in front of the already stored polygon. Setting the clipping planes most be done carefully. Because of the perspective projection this results in better precision in the front. Another problem is that some pixels will be set more than once.

In a game engine would z-buffer probably be used in combination with some techniques to speed up rendering, such as: discarding polygons that are easy to detected as hidden. The use of bounding volumes. Some sort of sorting like painter algorithm.

Portal culling

If we have a world divided up into cells and portals. Using frustrum culling, what can we see from position in the first cell looking through the first portal? If we cannot see anything in cell, don't render antything in that cell.

Summary of hidden surfaces

The z-buffer technique is more reliable than the painters algorithm. Speed can be improved by using sorting and bounding volumes. Clipping is also useful for portal culling, dividing the world into cells we can or cannot see. In games and visualization hybrid approaches are used.

Backface culling

This is an easy way to discard up to 50% of the polygons. Doesn't work for not closed objects. Doesn't work for so called impostors either, that are polygons textured on both sides.

Parallel and perspective projection

Parallel: By checking if the sign of the normal we can determine if it is pointing away from the camera, if its pointing in the other direction we can discard it. **Perspective** is a different story: computing the cosine between the normal and the projector which is the dot product, if the result is positive, discard it!

OpenGL

OpenGL perform backface culling after clipping. Use the signed area in device coordinate space: the polygon on the screen will have a negative area if it is backface, therefore there is a need for some consistent ordering of the vertices, like clockwise or counter clockwise.

8 Texture mapping

Texture mapping allows us to add more detail without adding more geometry. In this lecture will the following techniques be presented:

- Texture mapping
- Displacement mapping
- Bump and normal mapping
- Environment mapping
- Procedural textures
- Billboarding

Textures

In most real-time rendering is textures of image-based 2D arrays of data used. The textures can be created from photos or hand drawn images. There are also textures in 1D and 3D used in some applications. The elements of these textures are called texels, can hold many types of information such as: Colors, normals, opacity, intensity gloss, height and ambient occlusion for mentioning some.

Texture coordinates

The range of texture coordinates are 0 to 1 and are commonly denoted s,t and p. s is used for 1D textures, (s,t) for 2D and (s,t,p) for 3D textures. In some applications can (u,v,w) occur.

Assigning texture coordinates to models

For assigning 2D textures to 3D models must each vertex be assigned a texture coordinate. This is possible to do manually for simple models like planes, spheres and cubes. For more complex models are the texture coordinates usually assigned by a semi-automatic process of **unwrapping** the mesh on a 2D grid in a 3D modelling software. One other possibility is to use a **projector function** to generate the coordinates automatically. Mapping a 2D map of the earth onto a sphere is done using longitude/latitude as texture coordinates.

Loading texture images from file

There is no function for loading textures provided by OpenGL. This is done by one of many third-party libraries available, such as: stb-image, LodePNG and FreeImage.

Outside the coordinate range

What happens if we would go outside the range of the texture? There are different alternative provided by OpenGL:

- GL_CLAMP_TO_EDGE

- GL_CLAMP_TO_BORDER
- GL_REPEAT
- GL_MIRRORED_REPEAT

This alternatives determines how the image should be wrapped when outside the range of 0-1. They have different advantages and choosing depends on the use case.

Texture filtering

When the texture coordinate doesn't correspond to the center of the texture element (texel) there are different options to choose from. There might be many texels in the same pixel which is called **minification** or many pixels representing one texel, **magnification**. In OpenGL there are some options how to handle these **filter operations**:

- GL_NEAREST
- GL_LINEAR
- GL_NEAREST_MIMAP,GL_NEAREST_MIPMAP_LINEAR ...

One could also implement their own filtering operations in the fragment shader. Aliasing in something the be aware of in minification, here doesn't the choice of filter operations matter.

Mipmapping

We need a method to suppress the aliasing problem from minification. One way to do it is by using **mipmapping**. In mipmapping is the original texture filtered down repeatedly into smaller images to create a chain of mipmaps. During the rendering can then OpenGL select higher level mipmaps for distant object and lower for closer. It is convenient to have textures which are in the dimensions of a power of two for this case. Mipmaps can be created automatically with the function:

- void glGenerateMipmap(GLenum, target);

The resulting texture is only 33 larger than the original texture. This technique improves both image quality and rendering performance.

Displacement mapping

By using a 2D **height map** texture for displacing the vertices in the normal direction we can create a structured surface. The mesh has in this case to be subdivided into smaller triangles to incorporate the height map. The main drawback of this is of course that it's expensive to store and render the extra geometry.

Bump mapping

An alternative way is to use **Bump mapping** where a 2D height map texture is used to perturb (interfere?) the surface normals of the rendered object. This technique is more efficient than displacement mapping since there is no extra geometry involved. The eye is tricked to believe the surface is bumpy. What gives it away is the contour and shadows that is unchanged (unlike in displacement mapping).

Normal mapping

Similar to bump mapping but here it uses a 2D RGB texture containing the normal vectors, we call these kind of maps of the normal vector in a texture: **normal map**. The RGB colors represent the displacement direction. The normal vectors are used to perturb the the surface normals of the object we are rendering.

Bump vs Normal mapping

Comparing the two methods:

Bump	**Normal**
Gray-value map	RGB map
Smaller on disk	Larger on disk
Normals computed on the fly	Normals already computed
Somewhat slower rendering	Somewhat faster rendering

Table 1: example

Tangent space

For normal and bump mapping we need a coordinate system that is local for each vertex to displace the normal in that coordinate system. We call this space **tangent space**. It is defined by three orthogonal unit vectors:

- The surface normal $\mathbf{n}$
- The binormal $\mathbf{b}$
- and the tangent $\mathbf{t}$

We can either pre-compute these vectors from the texture coordinate or compute them directly in the fragment shader.

Environment mapping

Is an **image-based lightning** IBL technique used to approximate global illumination effect such as reflection and refraction. Here is the incoming light from the environment stored in textures and then mapped onto the reflecting or refracting object.

Cube environment mapping

A cube map is a type of texture that stores the incoming light from the environment in a cube with six sides. For cube mapping we find where a perfect reflection would hit the cube map. This is done by calculating a reflection vector **R** from the surface normal **N** and the incident vector **I**. The incident vector is a vector that points from the camera to the reflecting point at the surface.

$$\mathbf{R} = \mathbf{I} - (\mathbf{N} \cdot \mathbf{I})\mathbf{N} \tag{33}$$

Using this reflection vector as a texture coordinate to fetch the incoming light from the cube coordinate.

Procedural textures

Procedural textures are textures that can be generate **on-the-fly** on the GPU by evaluating a function that describes a pattern of interest. Unlike image-based textures these doesn't need any storage since they are generated on the fly. They can also be rendered at arbitrary resolution thanks to this. Example of such patterns are chessboard pattern and different randomly generated, smoke like textures. They are easy to use for creating seamless or animated patterns.

Perlin noise

A method to produce "**structured chaos**". Have been widely used for rendering natural looking object with noisy or fractal patterns such as terrain, fur, vegetation, clouds, water, smoke, fire and so on.

Skyboxes

Are used for rendering backgrounds in a 3D scene. The idea is to place the viewer/camera inside a large cube and use cube mapping for projecting 2D background images on the cube's faces. This creates the illusion of a 3D surrounding.

Alpha mapping and billboarding

Trees and grass can be represented as set of **billboards** instead of solid surfaces. By using a set of billboards, quads with semi-transparent textures we can create the illusion of a 3D object.

Imposter rendering

A view aligned billboard is called an **impostor**. By doing this can thousands or even millions billboards be rendered at very little cost. Very useful for molecule visualizations for an example.

Particle systems

Particle systems are used for modelling realistic visual effects suchs as explosion, smoke, fire, water and dust. The idea is to define a set of initial attributes such as: position, color, opacity size etc for all particles. In each frame we update the particle simulation on the CPU/GPU and render the particles as semi-transparent texture quads.

9 Global illumination

So far in this course we have used local illumination models such as Phong and environment mapping which are simplified models of illumination. In global illumination there will be bounces and reflections of light that was not present in previous methods. Some thing will come for "free" with global illumination:

- Environment mapping - multiple inter-reflections
- Shadows
- Colour bleeding
- Refraction

But all this comes at the high computational cost.

The main techniques in global illuminations are:

- Raytracing
  - Works well for specular and translucent surfaces
- Radiosity
  - Works well for matte surfaces
- Photon mapping
  - Works for both translucent and matte surface (not covered)

The rendering equation

The goal of global illumination methods is to compute both the direct and indirect light. The theoretical light can be computed with the **rendering equation**. The methods above tries to approximate this equation:

$$L_0(\mathbf{x}, \omega, \lambda, t) = L_e(\mathbf{x}, \omega, \lambda, t) + \int_\Omega f_r(\mathbf{x}, \omega', \omega, \lambda, t) + L_i(\mathbf{x}, \omega', \lambda, t)(-\omega' \cdot \mathbf{n})\, d\omega' \tag{34}$$

Where L_e are a light source or fluorescent ?,L_i incoming light, f_r is the surface behavior (BRDF) and $(-\omega' \cdot \mathbf{n})$ is the Lambert.

BRDF

The Bidirectional Reflection Distribution Function describes how the incoming light interacts with the surface of the object: absorption, transmission and reflectance. Described by a "surface" that is spanned by the vectors of the incoming light. The physically based BRDF should be reciprocal, conserve the energy and be measured for any physical material. The Phong method doesn't conserve energy and is there for **not** physically based!

Raytracing

The basic idea is to cast rays through our viewing window against the target and let it interact with it, getting the attenuation of the color of the light ray eventually to output in the pixel the ray was cast through. This process mean a lot of work...

Specular vs diffuse

Tracing rays in the specular direction if the material is specular is no problem. If the material is matte we must shoot rays in ALL directions which is **not** possible. This is the reason why raytracing is used mainly for specular and translucent surfaces.

Super sampling

A ray that is shot through the same pixel but in different positions inside the pixel, it can give very different results. To solve this we shoot many rays in each pixel by varying it slightly in how it shots through the pixel. This is called super sampling and can be done stochastic, the more the better. This reduce the jagged edges that can be seen in raytracing otherwise.

Bounding volumes

To speed up the ray traversal one can use bounding volume techniques. This is to reduce the time it takes to check if a ray hits any of the triangles in a scene. By checking what bounding volume it hit first we can save computational time. By placing for example a box over the polygon object we are left with first checking 6 sides instead of thousands of polygons for a hit. The cases we are left with are:

- Ray hit both the cube and the object
- Ray misses the cube
- Ray hits the cube but misses the object

If we hit the box we can further create subboxes so we perhaps only have a hundred of polygons to check, while some boxes are empty and simply discarded. The math we need is for computing planes. We have a point in the plane and a vector for that plane, so we have all that we need:

$$\mathbf{N} \cdot (\mathbf{P} - \mathbf{P}_0) = 0 \tag{35}$$

We then check the sign of the plane equation to see if we are inside or outside the plane. Another way to this is to project on the axis and compare min and max for all axis.

Intersections can be computed by regarding the line as a parametric one. This is a linear interpolation with

$$p(\alpha) = (1 - \alpha)p_1 + \alpha p_2 \quad 0 \geq \alpha \geq 1 \tag{36}$$

If we write the line and plane equation in matrix form, n is the normal of the plane and p_0 is point on the plane. We then need to solve the following equation:

$$\begin{aligned} p(\alpha) &= (1-\alpha)p_1 0 \alpha p_2 \\ n \cdot (p(\alpha) - p_0) &= 0 \end{aligned} \tag{37}$$

Solving this result in:

$$\alpha = \frac{n \cdot (p_0 - p_1)}{n \cdot (p_2 - p_1)} \tag{38}$$

When we know if a line hits a plane we can continue with computing where it hits. We need to determine if it hits inside or outside the polygon. One way to check this is to check to cross product with the edges counter clockwise and see if the sign of the resulting normal. The signs will be different if the point is outside.

Bounding spheres

In this case there is only one center and a radius to be checked. The downside is that not all objects are suitable for sphere, lot of empty space. Could be partially solved with hierarchy of spheres. To test if inside the sphere we apply the point-sphere test. Compare the point with the center and radius:

$$||\mathbf{p} - \mathbf{c}||^2 > r^2 \tag{39}$$

The ray sphere intersection can be calculated with this quadratic equation:

$$(\mathbf{d} \cdot \mathbf{d})t^2 + 2(\mathbf{o} - \mathbf{c}) \cdot \mathbf{d}t + (\mathbf{o} - \mathbf{c}) - r^2 = 0 \tag{40}$$

It is quadratic since it enters and exits in two points.

Space partitioning

A hierarchy of bounding volumes, the idea is to shoot more rays where we have a lot of details and where there is light. Some techniques for this are:

- Quadtrees - 2D each node has 4 children
- Octrees - 3D each node have 8 children
- K-D - Like a binary search tree
- BSP - Binary space partition tree, for complicated object,

Radiosity

This is an application of a finite element method to solve the rendering equation. It comes from the field of **heat transfer**. Example : the "Cornell box". This can actually be solved analytically depending on the triangulation of the scene.

The method computes form factors and solves the resulting radiosity matrix. It is a slow approach.

The form factor is defined as the fraction of energy leaving one surface. This is a purely geometric relationship, independent of the viewpoint or the surface attributes.

Progressive refinement

The first step is for the light sources shoot energy. Then add patches that gather energy. For the next iteration all patches shoots energy, and than they gather energy, we let this continue until equilibrium. This is generally a faster method. We can start by letting all surfaces have some energy.

10 Curves and surfaces

Until this point we have worked with flat entities such as lines and polygons. They fit well with the graphics hardware and are mathematically simple to work with. But since the world is not composed of only flat entities we need also to be able to work with curves and curved surfaces. We may only need them at the application level, in implementation we render then approximately with flat primitives.

Modelling with curves

From data points we approximate a curve and interpolate between the data points. So what is a good representation? There are multiple ways to represent curves and lines and we want our representation to be:

- Stable
- Smooth
- Easy to evaluate

Do we need to interpolate or is it possible just to come close to data?

Explicit representation

The most familiar form of a curve in 2D is the equation:

$$y = f(x) \tag{41}$$

The problem with this expression is that it cannot express vertical lines or circles. For an extension i 3D we have:

$$\begin{aligned} y = f(x), z = g(x) \\ z = f(x, y) \end{aligned} \tag{42}$$

Where the second form defines a surface.

Implicit representation

We can describe a two dimensional curve implicitly:

$$g(x, y) = 0 \tag{43}$$

This is much more robust:

$$\begin{aligned} \text{Lines } ax + by + c = 0 \\ \text{Circles } x^2 + y^2 - r^2 = 0 \end{aligned} \tag{44}$$

In 3D, g(x,y,z) defines a surface, and by intersecting two surfaces we can get a curve. But in general: we cannot find y for a given x

Parametric curves

Here we separate equation for each spatial variable:

$$\begin{aligned} x &= x(u) \\ y &= y(u) \\ z &= z(u) \\ p(u) &= \begin{bmatrix} x(u) & y(u) & z(u) \end{bmatrix}^T \end{aligned} \tag{45}$$

Parametric lines

We can normalize u to be over the interval $\begin{bmatrix}0, 1\end{bmatrix}$. A line connecting two points p_0 and p_1 : $\mathbf{p}(u) = (1-u)\mathbf{p}_0 + u\mathbf{p}_1$. A ray from $\mathbf{p}_0$ in the direction $\mathbf{d}$ gives us: $\mathbf{p}(u) = \mathbf{p}_0 + u\mathbf{d}$

Parametric surfaces

Surfaces require 2 parameters. x=(u,v)

$$\begin{aligned} x &= x(u, v) \\ y &= y(u, v) \\ z &= z(u, v) \\ \mathbf{p}(u, v) &= \begin{bmatrix} x(u, v) & y(u, v) & z(u, v) \end{bmatrix} \end{aligned} \tag{46}$$

Normals

We can differentiate with respect to the parameters u and v to obtain the normal at a point **p**

$$\frac{\partial \mathbf{p}(u, v)}{\partial u} = \begin{bmatrix} \partial x(u, v)/\partial u \\ \partial y(u, v)/\partial u \\ \partial z(u, v)/\partial u \end{bmatrix} \quad \frac{\partial \mathbf{p}(u, v)}{\partial v} = \begin{bmatrix} \partial x(u, v)/\partial v \\ \partial y(u, v)/\partial v \\ \partial z(u, v)/\partial v \end{bmatrix} \tag{47}$$

$$\mathbf{n} = \frac{\partial \mathbf{p}(u, v)}{\partial u} \times \frac{\partial \mathbf{p}(u, v)}{\partial v} \tag{48}$$

Curve segments

After we have normalizing u, each curve is written : $\mathbf{p} = \begin{bmatrix} x(u) & y(u) & z(u) \end{bmatrix}^T, 0 \leq u \leq 1$. In classic numerical methods we can design a single global curve. In computer graphics and in CAD, it is often better to design **small** connected curve **segments**.

Selecting functions

what we want is functions that can approximate and interpolate our data, it should be easy to evaluate, easy to differentiate and the functions must be smooth.

Polynomials

We start off with polynomials, these are easy to evaluate, they are continuous and differentiable everywhere. What we must worry about are how the continuity at the joints and the continuity of the derivatives here.

Cubic parametric polynomials

Polynomials of degree three gives a balance and is most often used. We got four coefficients to determine for each of x,y and z. Must seek for four independent conditions for the various values of u resulting in 4 equations and 4 unknowns for each of x,y and z. The conditions are a mixture of the requirements for continuity at the joints and fitting to the data.

$$\sum_{n=0}^{3} c_n u^n = c_0 + c_1 u + c_2 u^2 + c_3 u^3, \quad p(u) = \mathbf{u}^T c = c^T \mathbf{u} \tag{49}$$

Interpolating curve

Given four data points, we want to determine a cubic **p**(u) which passes through them, meaning we need to find our coefficients c_0, c_1, c_2, c_3