Chapter 1

OpenGL and computer graphics

Slide OpenGL:

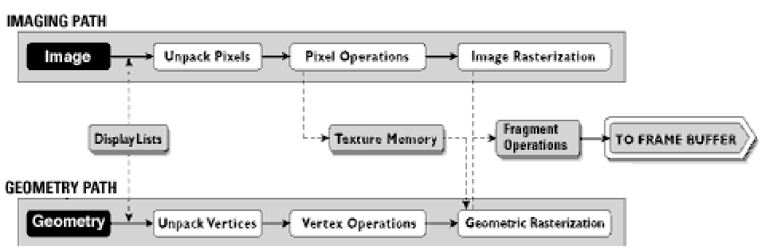
"OpenGL is the premier environment for developing portable, interactive 2D and 3D graphics applications. Since its introduction in 1992, OpenGL has become the industry's most widely used and supported 2D and 3D graphics application programming interface (API), bringing thousands of applications to a wide variety of computer platforms. OpenGL fosters innovation and speeds application development by incorporating a broad set of rendering, texture mapping, special effects, and other powerful visualization functions. Developers can leverage the power of OpenGL across all popular desktop and workstation platforms, ensuring wide application deployment."

`www.opengl.org`

Slide OpenGL: What can it do?:

- Imaging part: works on pixels, bitmaps
- Geometry part: works on vertices, polygons
- uses a rendering pipeline that starts from data and ends with a display device.

Slide OpenGL rendering pipeline:

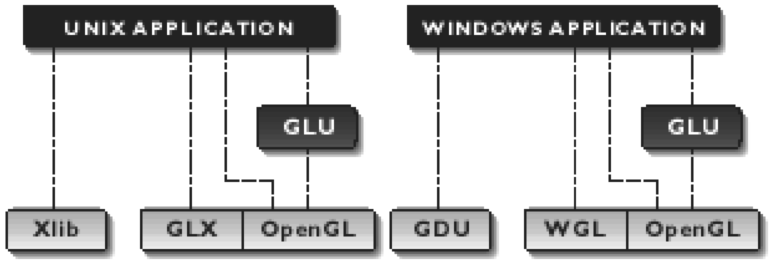


Slide OpenGL: More info:

- Application Program Interface based on C-style function calls
- industry standard: one of several (Java3D, DirectX are others)
- stable, reliable and portable

- scalable: low-end PC to supercomputer
- well documented and easy to use

Slide OpenGL on Windows and Unix:



- GLU: OpenGL-Extension for complex polygons, curves etc.

Slide The structure of an OpenGL application:

```
int main(int argc, char** argv)
{
        glutInit(&argc, argv);
        glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
        glutInitWindowSize(640,480);
        glutInitWindowPosition(100, 150);
        glutCreateWindow("my␣first␣attempt");
        glutDisplayFunc(myDisplay);
        myInit();
        glutMainLoop();
        return 0;
}
```

Slide Other Callback Functions:

```
...
    glutDisplayFunc(myDisplay);
    glutReshapeFunc(myReshape);
    glutMouseFunc(myMouse);
    glutKeyboardFunc(myKeyboard);
...
```

Slide Draw three points:

```
void myDisplay(void)
{
        glClear(GL_COLOR_BUFFER_BIT);
        glBegin(GL_POINTS);
```

```
                glVertex2i(100, 50);
                glVertex2i(100, 130);
                glVertex2i(150, 130);
        glEnd();
        glFlush();
}
```

Slide OpenGL Functions:
`glVertex2i()`

- `gl` is the prefix of all OpenGL function names
- `Vertex` is a function name
- `2i` describes the arguments: two integers

Slide OpenGL Datatypes:

GLenum,GLboolean,GLbitfield unsigned datatypes

GLvoid pseudo datatype for pointers and return values

GLbyte,GLshort,GLint 1,2,4-byte signed

GLubyte,GLushort,GLuint 1,2,4-byte unsigned

GLsizei 4-byte signed size datatype

Slide OpenGL Datatypes:

GLfloat single precision float

GLclampf single precision float in [0,1]

GLdouble double precision float

GLclampd double precision float in [0,1]

Slide Drawing Dots:

```
        glBegin(GL_POINTS);
                glVertex2i(100, 50);
                glVertex2i(100, 130);
                glVertex2i(150, 130);
        glEnd();
```

Slide Drawing a line:

```
glBegin(GL_LINES);
        glVertex2i(100, 50);
        glVertex2i(100, 130);
glEnd();
```

Slide Drawing two lines:

```
glBegin(GL_LINES);
        glVertex2i(10, 20);
        glVertex2i(40, 20);
        glVertex2i(20, 10);
        glVertex2i(20, 40);
glEnd();
```

Slide Drawing a polyline:

```
glBegin(GL_LINE_STRIP);
        glVertex2i(10, 20);
        glVertex2i(40, 20);
        glVertex2i(20, 10);
        glVertex2i(20, 40);
glEnd();
```

Slide Drawing a polygon:

```
glBegin(GL_LINE_LOOP);
        glVertex2i(10, 20);
        glVertex2i(40, 20);
        glVertex2i(20, 10);
        glVertex2i(20, 40);
glEnd();
```

Slide Drawing an aligned rectangle:

```
glRecti(x1,y1,x2,y2);
```

Slide What are those numbers?:

- There is no predefined way of interpreting the coordinates
- OpenGL can work with different coordinate systems

- For OpenGL, we have to define a coordinate system to be used

---

Slide Colors and a Coordinate System:

```
 void myInit(void)
 {
    glClearColor(1.0,1.0,1.0,0.0);
    glColor3f(0.0f, 0.0f, 0.0f);
    glPointSize(4.0);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0.0, 640.0, 0.0, 480.0);
}
```

Slide Algorithmic Drawing:

```
void Sierpinski(void){
    GLintPoint T[3]= {{10,10},{300,30},{200, 300}};
    int index = random(3);
    GLintPoint point = T[index];
    drawDot(point.x, point.y);
    for(int i = 0; i < 4000; i++) {
         index = random(3);
         point.x = (point.x + T[index].x) / 2;
         point.y = (point.y + T[index].y) / 2;
         drawDot(point.x,point.y);
    }
    glFlush();
}
```

Chapter 2

CG Basics

Slide Lecture 4:

- Coordinate Systems, Viewports, World Windows
- Clipping
- Relative Drawing
- Parameterized Curves
- Double Buffering for Animation

Slide Coordinate System:

- For now, we have used a simple coordinate system:
  $x : 0 \ldots \text{ScreenWidth} - 1, y = 0 \ldots \text{ScreenHeight} - 1$
- In case ScreenWidth or ScreenHeight change, glut can inform us via the `glutReshapeFunc(myReshape);`
- We can manually apply a *coordinate transformation* in order to display arbitrary coordinate systems.
- Or we can have OpenGL do this for us

Slide Some terms:

- The space in which objects are described uses *world coordinates*.
- The part of this space that we want to display is called *world window*.
- The window that we see on the screen is our *viewport*.
- In order to know where to draw something, we need the *world-to-viewport transformation*
- Note that these terms can be used both for 2D and for 3D.

Slide A simple example:

$$\begin{aligned} sx &= Ax + C \\ sy &= By + D \\ A &= \frac{V.r - V.l}{W.r - W.l} \\ C &= V.l - AW.l \\ B &= \frac{V.t - V.b}{W.t - W.b} \\ D &= V.b - bW.b \end{aligned}$$

Slide In OpenGL:

```
 void setWindow(float left, float right,
                float bottom, float top)
 {
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(left, right, bottom,top);
}
void setViewport(int left, int right,
                 int bottom, int top)
{
  glViewport(left,bottom,right-left,top-bottom);
}
```

Slide Clipping:

- What happens to parts of the “world” that are outside of the world window?
  Answer: They are not drawn.
- How to identify the parts of the world that are to be drawn?
- Clipping Lines: identifying the segment of a line to be drawn
- Input: the endpoints of a line and a world window
- Output: the new endpoints of the line (if anything is to be drawn)

Slide Clipping:

- First step: Testing for trivial accept or reject
- Cohen Sutherland Clipping Algorithm
- For each point do four tests, compute 4 bit word:
  1. Is P to the left of the world window?
  2. Is P above the top of the world window?
  3. Is P to the right of the world window?
  4. Is P below the bottom of the world window?

Slide Cohen Sutherland:

- Compute tests for both points of the line
- Trivial Accept: all tests false, all bits 0
- Trivial Reject: the words for both points have 1s in the same position
- Deal with the rest: neither trivial accept nor reject

Slide The rest:

- Identify which point is outside and to which side of the window
- Find the point where the line touches the world window border
- Move the outer point to the border of the window
- repeat all until trivial accept or reject

Slide CLIPSEGMENT$(p1, p2, W)$**:**

```
while (TRUE) do
  if (trivial accept) then
    RETURN 1
  end if
  if (trivial reject) then
    RETURN 0
  end if
  if (p1 is outside) then
    if (p1 is to the left) then
      chop against the left edge of W
    else
      if (p1 is to the right) then
        chop against the right edge of W
      else
        if ( . . . ) then
          . . .
        end if
      end if
    end if
  end if
end while
```

Slide Relative drawing:

- It is often convenient to draw figures relative to a current pen position
- Idea: maintain the current position (CP) a static global variable
- use two functions MOVEREL and LINEREL to move/draw relative to CP
- implementation is obvious. (or can be found in the book on page 105)

Slide Application of relative drawing:

- Turtle graphics: originally from the logo programming language
  - logo has been invented at MIT to teach children how to program. try google for more info
- Simple primitives: TURNTO (absolute angle) TURN (relative angle) FORWARD (distance, isVisible)

- Implementation obvious: maintain additional current direction (CD) in a static global variable, use simple (sin, cos) trigonometry functions for FORWARD.

Slide Application of relative drawing: n-gons:

- The vertices of an n-gon lie on a circle
- divide the circle into n equal parts
- connect the endpoints of the parts on the circle with lines
- using relative drawing, this is very easy to implement
- by connecting every endpoint to every other endpoint, a rosette can be drawn

---

Slide relative hexagon:

```
for (i=0;i<6;i++)
{
  forward(L,1);
  turn(60);
}
```

Slide Circles and Arcs:

- Circles can be approximated with n-gons (with a high n)
- Arcs are partially drawn circles, instead of dividing the circle, divide the arc

---

Slide Representing curves:

- Two principle ways of describing a curve: implicitly and parametrically
- Implicitly: Give a function F so that $F(x, y) = 0$ for all points of the curve
- Example: $F(x, y) = (y - A_y)(B_x - A_x) - (x - A_x)(B_y - A_y)$ (a line)
- Example: $F(x, y) = x^2 + y^2 - R^2$ (a circle)

Slide Implicit form of curves:

- The implicit form is good for testing if a point is on a curve.
- For some cases, we can use the implicit form to define an "inside" and an "outside" of a curve: $F(x, y) < 0 \rightarrow$ inside, $F(x, y) > 0 \rightarrow$ outside
- some curves are *single valued* in x: $F(x, y) = y - g(x)$ or in y:$F(x, y) = x - h(y)$
- some curves are neiter, e.g. the circle needs two functions $y = \sqrt{R^2 - x^2}$ and $y = -\sqrt{R^2 - x^2}$

Slide Parametric form of curves:

- The parametric form of a curve suggests the movement of a point through time.
- Example: $x(t) = A_x + (B_x - A_x)t$,$y(t) = A_y + (B_y - A_y)t$,$t \in [0, 1]$

- Example: $x(t) = W\cos(t)$, $y(t) = H\sin(t)$,$t \in [0, 2\pi]$
- In order to find an implicit form from a parametric form, we can use the two $x(t)$ and $y(t)$ equations to eliminate t and find a relationship that holds true for all t.
- For the Ellipse: $\left(\frac{x}{W}\right)^2 + \left(\frac{y}{H}\right)^2 = 1$

Slide Drawing parametric curves:

- In order to draw a parametric curve, we have to approximate it.
- In order to do that, we chose some values of t and sample the functions x and y at t_i.
- One option is to approximate the function in between with line segments.

```
glBegin(GL_LINES);
for (i=0;i<n;i++)
    glVertex2f(x(t[i]),y(t[i]));
glEnd();
```

Slide Superellipses:

- A *superellipse* is defined by the implicit form $\left(\frac{x}{W}\right)^n + \left(\frac{y}{H}\right)^n = 1$
- A *supercircle* is a superellipse with $W = H$.
- $x(t) = W\cos(t)|\cos(t)^{2/n-1}|$
- $y(t) = H\sin(t)|\sin(t)^{2/n-1}|$

Slide Polar coordinate shapes:

- Polar coordinates can be used to draw parametric curves.
- The curve is represented by a distance to the center point r and an angle θ.
- $x(t) = r(t)\cos(\theta(t))$,$y(t) = r(t)\sin(\theta(t))$ (general form)
- $x(\theta) = f(\theta)\cos(\theta)$,$y(t) = f(\theta)\sin(\theta)$ (simple form)
- Cardioid $f(\theta) = K(1 + \cos(\theta))$
- Rose Curves $f(\theta) = K\cos(n\theta)$
- Archimedian Spiral $f(\theta) = K\theta$
- Conic sections $f(\theta) = \frac{1}{1 \pm e\cos(\theta)}$
- Logarithmic Spiral $f(\theta) = Ke^{a\theta}$

Slide 3D parametric curves:

- We can also specify 3d curves using three functions $x(t), y(t), z(t)$

- Helix: $x(t) = \cos(t), y(t) = \sin(t), z(t) = bt$
- Toroidal spiral:
  - $x(t) = (a\sin(ct) + b)\cos(t)$
  - $y(t) = (a\sin(ct) + b)\sin(t)$
  - $z(t) = a\cos(ct)$

Slide Animation w. double buffering:

- When we do a fast animation, the image starts to flicker.
- This results from the time it takes to draw the lines.
- We can avoid this via double-buffering
- in OpenGL, double buffering is simple:
- `glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGB);`
- `glutSwapBuffers();`

Slide Lecture 5:

- Vectors
- Lines and Planes in 3D space
- affine representation
- the dot product and the cross product
- homogenous representations
- intersection and clipping

Slide Vectors:

- We all remember what vectors are, right?
- The difference of two points is a vector
- The sum of a point and a vector is a point
- A linear combination $a\vec{v} + b\vec{w}$ is a vector
- Let's write $w = a_1\vec{v}_1 + a_2\vec{v}_2 + \cdots + a_n\vec{v}_n$
- If $a_1 + a_2 + \cdots + a_n = 1$ this is called an affine combination
- if additionally $a_i \geq 0$ for $i = 1 \ldots n$, this is a convex combination
- To find the length of a vector, we can use Pythagoras: $|\vec{w}| = \sqrt{w_1^2 + w_2^2 + \cdots + W_n^2}$

Slide Vectors:

- When we know the length, we can normalize the vector, i.e. bring it to unit length: $\hat{a} = \vec{a}/|\vec{a}|$. We can call such a unit vector a *direction*.
- The dot product of two vectors is $\vec{a} \cdot \vec{b} = \sum_{i=1}^{n} \vec{v}_i \vec{w}_i$ has the well-known properties
  - $\vec{a} \cdot \vec{b} = \vec{b} \cdot \vec{a}$ (Symmetry)
  - $(\vec{a} + \vec{c}) \cdot b = \vec{a} \cdot \vec{b} + \vec{c} \cdot \vec{b}$ (Linearity)
  - $(s\vec{a}) \cdot \vec{b} = s(\vec{a} \cdot \vec{b})$ (Homogeneity)
  - $|\vec{b}|^2 = \vec{b} \cdot \vec{b}$
- We can play the usual algebraic games with vectors (simplification of equations)

Slide Angles between vectors:

- We can use the dot product to find the angle between two vectors: $\vec{a} \cdot \vec{b} = |\vec{a}||\vec{b}| \cos(\theta)$. If the dot product of two (non-zero-length) vectors is 0 then they are *perpendicular* or *orthogonal* or *normal* to eachother.
- In 2D, we can find a perpendicular vector by exchanging the two components and negate one of them: If $\vec{a} = (a_x, a_y)$ then $\vec{b} = (-a_y, a_x)$ and we call this the *counterclockwise perpendicluar* vector of $\vec{a}$ or short $\vec{a}^{\perp}$

Slide The 2D "Perp" Vector:

- The "prep" vector is useful for projections (see book, page 157)
- The distance from a point C to the line through A in direction $\vec{v}$ is $|\vec{v}^{\perp} \cdot (C - A)|/|\vec{v}|$.
- Projections are used to simulate reflections

Slide The cross product:

- Everybody remembers $\vec{a} \times \vec{b}$
- One trick to write the cross product: Let $\vec{i}, \vec{j}, \vec{k}$ be the 3D standard unit vectors. Then the cross product of $\vec{a} \times \vec{b}$ can be written as the *determinant* of a matrix:

$$\vec{a} \times \vec{b} = \begin{vmatrix} \vec{i} & \vec{j} & \vec{k} \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix}$$

- and we have the usual algebraic properties: antisymmetry, linearity, homogeneity...

Slide Coordinate Systems and Coordinate Frames:

- A coordinate system can be defined by three mutually perpendicular unit vectors.
- If we put these unit vectors into a specific point ϑ called origin, we call this a coordinate frame.
- In a coordinate frame, a point can be represented as $P = p_1\vec{a} + p_2\vec{b} + p_3\vec{c} + \vartheta$.

- This leads to a distinction between points and vectors by using a fourth coefficient in the so-called homogenous representation of points and vectors.

Slide Homogenous Representation:

- A vector in a coordinate frame:

$$\vec{v} \quad = \quad (\vec{a}, \vec{b}, \vec{c}, \vartheta) \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ 0 \end{pmatrix}$$

Slide Homogenous Representation:

- A point in a coordinate frame:

$$P \quad = \quad (\vec{a}, \vec{b}, \vec{c}, \vartheta) \begin{pmatrix} P_1 \\ P_2 \\ P_3 \\ 1 \end{pmatrix}$$

Slide Homogenous coordinates:

- The difference of two points is a vector
- The sum of a point and a vector is a point
- Two vectors can be added
- A vector can be scaled
- Any linear combination of vectors is a vector
- An affine combination of two points is a point. (An affine combination is a linear combination where the coefficients add up to 1.)
- A linear interpolation $P = (a(1-t) + Bt$ is a point.
- This fact can be used to calculate a "tween" of two points.

Slide Representing lines and planes:

- A line can be represented by its endpoints B and C
- It can also be represented parametrically with a point and a vector $L(t) = C + \vec{b}t$.
- A line can also be represented in *point normal form* $\vec{n} \cdot (R - C)$
- For $\vec{n}$ we can use $\vec{b}^{\perp}$ with $\vec{b} = B - C$
- A plane can be represented by three points
- It can also be represented parametrically by a point and two nonparallel vectors: $P(s,t) = C + \vec{a}s + \vec{b}t$

- It can also be represented in a point normal form with a point in the plane and a normal vector. For any point R in the plane $n \cdot (R - B) = 0$.
- A part of the plane restricted by the length of two vectors is called a *planar patch*.

Slide intersections:

- Every line segment has a *parent line*.
- We can first find the intersection of the parent lines
- and then see if the intersection point is in both line segments
- In order to intersect a plane with a line, we describe the line parametrically and the plane in the point normal form. Solving this equation gives us a "hit time" t that can be put into the parametric representation of the line to identify the $hitpoint$.

Slide polygon intersections:

- In convex polygons, the problem is rather easy: we can test all the bounding lines/surfaces.
- In order to know which side of a line/plane is "outside", we represent them in a point normal form.
- We have to find exactly two "hit times" t_{in} and t_{out}.
- The right t_{in} will be the maximal "hit time" before the ray enters the polgon.
- The right t_{out} will be the minimal "hit time" after the ray exits the polgon.
- This approach can be used to clip against convex polygons. This is called the Cyrus-Beck-Clipping Algorithm.

Slide Lecture 6:

- Transformations
- in 2D
- in 3D
- in OpenGL

Slide Transformations:

- Transformations are an easy way to reuse shapes
- A transformation can also be used to present different views of the same object
- Transformations are used in animations.

Slide Transformations in OpenGL:

- When we're calling a `glVertex()` function, OpenGL automatically applies some transformations. One we already know is the world-window-to-viewport transformation.

- There are two principle ways do see transformations:
  - *object transformations* are applied to the coordinates of each point of an object, the coordinate system is unchanged
  - *coordinate transformations* defines a new coordinate system in terms of the old coordinate system and represents all points of the object in the new coordinate system.
- A transformation is a function that mapps a point P to a point Q, Q is called the image of P.

Slide 2d affine transformations:

- A subset of transformations that uses transformation functions that are linear in the coordinates of the original point are the affine transformations.
- We can write them as a class of linear functions:

$$\begin{pmatrix} Q_x \\ Q_y \\ 1 \end{pmatrix} = \begin{pmatrix} m_{11}P_x + m_{12}P_y + m_{13} \\ m_{21}P_x + m_{22}P_y + m_{23} \\ 1 \end{pmatrix}$$

Slide 2d affine transformations:

- or we can just use matrix multiplication

$$\begin{pmatrix} Q_x \\ Q_y \\ 1 \end{pmatrix} = \begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ 1 \end{pmatrix}$$

- or we can also transform vectors with the same matrix

$$\begin{pmatrix} W_x \\ W_y \\ 0 \end{pmatrix} = \begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} V_x \\ V_y \\ 0 \end{pmatrix}$$

Slide standard transformations:

- Translation

$$\begin{pmatrix} Q_x \\ Q_y \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & m_{13} \\ 0 & 1 & m_{23} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ 1 \end{pmatrix}$$

- scaling (and reflection for $S_{\{x,y\}} < 0$)

$$\begin{pmatrix} W_x \\ W_y \\ 1 \end{pmatrix} = \begin{pmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} V_x \\ V_y \\ 1 \end{pmatrix}$$

Slide standard transformations:

- Rotation (positive θ is CCW rotation)

$$\begin{pmatrix} Q_x \\ Q_y \\ 1 \end{pmatrix} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ 1 \end{pmatrix}$$

- shearing

$$\begin{pmatrix} Q_x \\ Q_y \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & h & 0 \\ g & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ 1 \end{pmatrix}$$

Slide Inverse transformations:

- inverse Rotation (positive θ is CW rotation)

$$\begin{pmatrix} Q_x \\ Q_y \\ 1 \end{pmatrix} = \begin{pmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ 1 \end{pmatrix}$$

- inverse Scaling

$$\begin{pmatrix} Q_x \\ Q_y \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{1}{S_x} & 0 & 0 \\ 0 & \frac{1}{S_y} & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ 1 \end{pmatrix}$$

Slide Inverse transformations:

- inverse shearing

$$\begin{pmatrix} Q_x \\ Q_y \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & -h & 0 \\ -g & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ 1 \end{pmatrix}$$

- inverse translation

$$\begin{pmatrix} Q_x \\ Q_y \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & -m_{13} \\ 0 & 1 & -m_{23} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ 1 \end{pmatrix}$$

Slide Inverse transformations:

- In general (provided that M is nonsingular)

$$P = M^{-1}Q$$

- But as M is quite simple:

$$\begin{aligned} \det M &= m_{11}m_{22} - m_{12}m_{21} \\ M^{-1} &= \frac{1}{\det M} \begin{pmatrix} m_{22} & -m_{12} \\ -m_{21} & m_{11} \end{pmatrix} \end{aligned}$$

Slide composing affine transformations:

- As affine transformations are simple matrix multiplications, we can combine several operations to a single matrix.
- In a matrix multiplication of transformations, the sequence of translations can be read from right to left.
- We can also take this combined matrix and reconstruct the four basic operations $M =$(translation)(shear)(scaling)(rotation) (this is for 2D only)

Slide Some more facts:

- Affine transformations preserve affine combinations of points
- Affine transformations preserve lines and planes
- Affine transformations preserve parallelism of lines and planes
- The column vectors of an affine transformation reveal the effect of the transformation on the coordinate system.
- An affine transformation has an interesting effect on the area of an object: $\frac{\text{area after transformation}}{\text{area before transformation}} = |\det M|$

Slide The same game in 3D...:

- The general form of an affine 3D transformation

$$\begin{pmatrix} Q_x \\ Q_y \\ Q_z \\ 1 \end{pmatrix} = \begin{pmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ P_z \\ 1 \end{pmatrix}$$

Slide Translation...:

- As expected:

$$\begin{pmatrix} Q_x \\ Q_y \\ Q_z \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & m_{14} \\ 0 & 1 & 0 & m_{24} \\ 0 & 0 & 1 & m_{34} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ P_z \\ 1 \end{pmatrix}$$

Slide Scaling in 3D...:

- Again:

$$\begin{pmatrix} Q_x \\ Q_y \\ Q_z \\ 1 \end{pmatrix} = \begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ P_z \\ 1 \end{pmatrix}$$

Slide Shearing...:

- in one direction

$$\begin{pmatrix} Q_x \\ Q_y \\ Q_z \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ f & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ P_z \\ 1 \end{pmatrix}$$

Slide Rotations 3D...:

- x-roll, y-roll and z-roll
- x-roll:

$$\begin{pmatrix} Q_x \\ Q_y \\ Q_z \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & c & -s & 0 \\ 1 & s & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ P_z \\ 1 \end{pmatrix}$$

Slide Rotations 3D...:

- y-roll:

$$\begin{pmatrix} Q_x \\ Q_y \\ Q_z \\ 1 \end{pmatrix} = \begin{pmatrix} c & 0 & s & 0 \\ 0 & 1 & 0 & 0 \\ -s & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ P_z \\ 1 \end{pmatrix}$$

Slide Rotations 3D...:

- z-roll:

$$\begin{pmatrix} Q_x \\ Q_y \\ Q_z \\ 1 \end{pmatrix} = \begin{pmatrix} c & -s & 0 & 0 \\ s & c & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ P_z \\ 1 \end{pmatrix}$$

Slide Some facts about Rotations 3D:

- 3D affine transformations can be composed as in 2D
- 3D rotation matrices do not commute (unlike 2D).
- Question: how to rotate around an arbitrary axis?
- Every 3D affine transformation can be decomposed into (translation)(scaling)(rotation)(shear_1)(shear_2).
- A 3D affine transformation has an effect on the volume of an object: $\frac{\text{volume after transformation}}{\text{volume before transformation}} = |\det M|$

Slide point vs coordinate system transformations:

- If we have an affine transformation M, we can use it to transform a coordinate frame F_1 into a coordinate frame F_2.
- A point $P = (P_x, P_y, 1)^T$ represented in F_2 can be represented in F_1 as MP
- $F_1 \rightarrow^{M_1} F_2 \rightarrow^{M_2} \rightarrow F_3$ then P in F_3 is $M_1 M_2 P$ in F_1.
- To apply the sequence of transformations M_1, M_2, M_3 to a point P, calculate $Q = M_3 M_2 M_1 P$. An additional transformation must be *premultiplied*.
- To apply the sequence of transformations M_1, M_2, M_3 to a coordinate system, calculate $M = M_1 M_2 M_3$. A point P in the transformed coordinate system has the coordinates MP in the original coordinate system. An additional transformation must be *postmultiplied*.

Slide And now in OpenGL...:

- Of course we can do everything by hand: build a point and vector datatype, implement matrix multiplication, apply transformations and call `glVertex` in the end.
- In order to avoid this, OpenGL maintains a *current transformation* that is applied to every `glVertex` command. This is independent of the window-to-viewport translation that is happening as well.
- The current transformation is maintained in the *modelview matrix*.

Slide And now in OpenGL...:

- It is initialized by calling `glLoadIdentity`
- The modelview matrix can be altered by `glScaled()`,`glRotated` and `glTranslated`.
- These functions can alter any matrix that OpenGL is using. Therefore, we need to tell OpenGL which matrix to modify: `glMatrixMode(GL_MODELVIEW)`.

Slide The 2D transformations:

- Scaling in 2d:

```
glMatrixMode(GL_MODELVIEW);
glScaled(sx,sy,1.0);
```

- Translation in 2d:

```
glMatrixMode(GL_MODELVIEW);
glTranslated(dx,dy,0);
```

- Rotation in 2d:

```
glMatrixMode(GL_MODELVIEW);
glRotated(angle,0.0,0.0,1.0);
```

Slide A stack of CTs:

- Often, we need to "go back" to a previous CT. Therefore, OpenGL maintains a "stack" of CTs (and of any matrix if we want to).
- We can push the current CT on the stack, saving it for later use: `glPushMatrix()`. This pushes the current CT matrix and makes a copy that we will modify now
- We can get the top matrix back: `glPopMatrix()`.

Slide 3D! (finally):

- For our 2D cases, we have been using a very simple parallel projection that basically ignores the perspective effect of the z-component.
- the view volume forms a rectangular parallelepiped that is formed by the border of the window and the *near plane* and the *far plane*.
- everything in the view volume is parallel-projected to the window and displayed in the viewport. Everything else is clipped off.
- We continue to use the parallel projection, but make use of the z component to display 3D objects.

Slide 3D Pipeline:

- The 3d Pipeline uses three matrix transformations to display objects
  - The modelview matrix
  - The projection matrix
  - The viewport matrix
- The modelview matrix can be seen as a composition of two matrices: a model matrix and a view matrix.

Slide in OpenGL:

- Set up the projection matrix and the viewing volume:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(left,right,bottom,top,near,far);
```

- Aiming the camera. Put it at eye, look at look and upwards is up.

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(eye_x,eye_y,eye_z,
    look_x,look_y,look_z,up_x,up_y,up_z);
```

Slide Basic shapes in OpenGL:

- A wireframe cube:

```
glutWireCube(GLdouble size);
```

- A wireframe sphere:

```
glutWireSphere(GLdouble radius,
    GLint nSlices,GLint nStacks);
```

- A wireframe torus:

```
glutWireTorus(GLdouble inRad, GLdouble outRad,
    GLint nSlices,GLint nStacks);
```

Slide And the most famous one...:

- The Teapot

```
glutWireTeapot(GLdouble size);
```

Slide The five Platonic solids:

- Tetrahedron: `glutWireTetrahedron()`
- Octahedron: `glutWireOctahedron()`
- Dodecahedron: `glutWireDodecahedron()`
- Icosahedron: `glutWireIcosahedron()`
- Missing one?

Slide Moving things around:

- All objects are drawn at the origin.
- To move things around, use the following approach:

```
glMatrixMode(GL_MODELVIEW);
glPushMatrix();
glTranslated(0.5,0.5,0.5);
glutWireCube(1.0);
glPopMatrix();
```

Slide Lecture 7:

- Wrapup of the lab session
- How was it again with those coordinates?
- representing hierarchic object structures
- perspective

Slide Again: And now in OpenGL...:

- Of course we can do everything by hand: build a point and vector datatype, implement matrix multiplication, apply transformations and call `glVertex` in the end.
- In order to avoid this, OpenGL maintains a *current transformation* that is applied to every `glVertex` command. This is independent of the window-to-viewport translation that is happening as well.
- The current transformation is maintained in the *modelview matrix*.

Slide Again: And now in OpenGL...:

- It is initialized by calling `glLoadIdentity`
- The modelview matrix can be altered by `glScaled()`,`glRotated` and `glTranslated`.
- These functions can alter any matrix that OpenGL is using. Therefore, we need to tell OpenGL which matrix to modify: `glMatrixMode(GL_MODELVIEW)`.

Slide Again: A stack of CTs:

- Often, we need to "go back" to a previous CT. Therefore, OpenGL maintains a "stack" of CTs (and of any matrix if we want to).
- We can push the current CT on the stack, saving it for later use: `glPushMatrix()`. This pushes the current CT matrix and makes a copy that we will modify now
- We can get the top matrix back: `glPopMatrix()`.

Slide Again: 3D:

- For our 2D cases, we have been using a very simple parallel projection that basically ignores the perspective effect of the z-component.
- the view volume forms a rectangular parallelepiped that is formed by the border of the window and the *near plane* and the *far plane*.

- everything in the view volume is parallel-projected to the window and displayed in the viewport. Everything else is clipped off.
- We continue to use the parallel projection, but make use of the z component to display 3D objects.

Slide Again: 3D Pipeline:

- The 3d Pipeline uses three matrix transformations to display objects
  - The modelview matrix
  - The projection matrix
  - The viewport matrix
- The modelview matrix can be seen as a composition of two matrices: a model matrix and a view matrix.

Slide Again: in OpenGL:

- Set up the projection matrix and the viewing volume:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(left,right,bottom,top,near,far);
```

- Aiming the camera. Put it at eye, look at look and upwards is up.

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(eye_x,eye_y,eye_z,
    look_x,look_y,look_z,up_x,up_y,up_z);
```

Slide Basic shapes in OpenGL:

- A wireframe cube:

```
glutWireCube(GLdouble size);
```

- A wireframe sphere:

```
glutWireSphere(GLdouble radius,
    GLint nSlices,GLint nStacks);
```

- A wireframe torus:

```
glutWireTorus(GLdouble inRad, GLdouble outRad,
    GLint nSlices,GLint nStacks);
```

Slide And the most famous one...:

- The Teapot

```
glutWireTeapot(GLdouble size);
```

Slide The five Platonic solids:

- Tetrahedron: `glutWireTetrahedron()`
- Octahedron: `glutWireOctahedron()`
- Dodecahedron: `glutWireDodecahedron()`
- Icosahedron: `glutWireIcosahedron()`
- Missing one?

Slide Moving things around:

- All objects are drawn at the origin.
- To move things around, use the following approach:

```
glMatrixMode(GL_MODELVIEW);
glPushMatrix();
glTranslated(0.5,0.5,0.5);
glutWireCube(1.0);
glPopMatrix();
```

Slide Rotating things:

- To rotate things, use the following approach:

```
glMatrixMode(GL_MODELVIEW);
glPushMatrix();
glRotatef(angle,0.0,1.0,0.0);
glutWireTeapot(1.0);
glPopMatrix();
```

Slide Hierarchical Modeling:

- If we try to model an everyday object (like a house), we do not want to move all its components separately.
- Instead we want to make sure that if we move the house, the roof of the house move together with the walls.
- The CT stack gives us a simple way to implement this.

Slide Global motion:

- The easiest case of hierarchical modeling is global motion.
- To implement it, we apply a number of transforms before we start drawing objects.

```
  glMatrixMode(GL_MODELVIEW);
  glPushMatrix();
  glTranslated(x,y,z);
  glRotatef(turnit,0.0,1.0,0.0);
  drawMyScene();
  glPopMatrix();
```

Slide Local motion:

- To implement local motion, apply an extra transformation before the object is drawn

```
drawmyteapot(){
  glMatrixMode(GL_MODELVIEW);
  glPushMatrix();
  glRotatef(spinit,0.0,0.0,1.0);
  glutWireTeapot(1.0);
  glPopMatrix();
}
```

Slide Perspective:

- Our current parallel projection is quite poor in giving us a “real” view of things.
- That is because it is “ignoring” the z component which leads to ambiguities.

Slide Perspective:

from http://www.leinroden.de/

Slide Perspective in OpenGL:

- Set up the projection matrix and the viewing volume:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(viewAngle,aspectRatio,N,F);
```

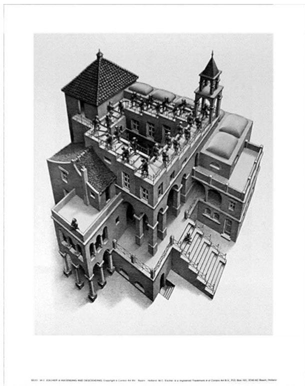
- Aiming the camera. Put it at eye, look at look and upwards is up. (no change here)

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(eye_x,eye_y,eye_z,
    look_x,look_y,look_z,up_x,up_y,up_z);
```

Slide Perspective:

- The point perspective in OpenGL resolves some ambiguities
- but it cannot solve all ambiguities

Slide Perspective:



from http://www.worldofescher.com

Slide Lecture 8:

- Solid Modeling
- Polygonal Meshes
- Shading

Slide Solid Modeling:

- We can model a solid object as a collection of polygonal faces.
- Each face can be specified as a number of vertices and a normal vector (to define the inside and the outside)
- For clipping and shading, it is useful to associate a normal vector with every vertex. Multiple vertices can be associated with the same normal vector and a vertex can be associated with multiple normal vectors.
- To represent and object, we could store all vertices for all polygons together with a normal vector for every vertex. That would be highly redundant.

Slide Storing polygonal meshes:

- Instead, we can use three lists:
  - the vertex list
    It contains all distinct vertices
  - the normal list
    It contains all distinct normal vectors
  - the face list
    It only contains lists of indices of the two other lists

Slide The basic barn:

vertex	x	y	z
0	0	0	0
1	1	0	0
2	1	1	0
3	0.5	1.5	0
4	0	1	0
5	0	0	1
6	1	0	1
7	1	1	1
8	0.5	1.5	1
9	0	1	1

normal	n_x	n_y	n_z
0	-1	0	0
1	-0.707	0.707	0
2	0.707	0.707	0
3	1	0	0
4	0	-1	0
5	0	0	1
6	0	0	-1

Slide The basic barn:

face	vertices	normals
0	0,5,9,4	0,0,0,0
1	3,4,9,8	1,1,1,1
2	2,3,8,7	2,2,2,2
3	1,2,7,6	3,3,3,3
4	0,1,6,5	4,4,4,4
5	5,6,7,8,9	5,5,5,5,5
6	0,4,3,2,1	6,6,6,6,6

Slide Finding the normal vectors:

- We can compute the normal of a face using three vectors and the cross product $m = (V_1 - V_2) \times (V_3 - V_2)$ and normalize it to unit length.
- Two problems arrise:
  - What if $(V_1 - V_2)$ and $(V_3 - V_2)$ are almost parallel?
  - What to do with faces that are defined through more than three vertices?
- Instead, we can use Newell's method:
  - $m_x = \sum_{i=0}^{N-1} (y_i - y_{next(i)})(z_i + z_{next(i)})$

- $m_y = \sum_{i=0}^{N-1}(z_i - z_{next(i)})(x_i + x_{next(i)})$
- $m_z = \sum_{i=0}^{N-1}(x_i - x_{next(i)})(y_i + y_{next(i)})$

Slide Properties of polygonal meshes:

- Solidity (if the faces enclose a positive and finite amount of space)
- Connectedness (if there is a path between every two vertices along the polygon edges)
- Simplicity (if the object is solid and has no "holes")
- Planarity (if every face is planar, i.e. every vertex of a polygon lies in a plane)
- Convexity (if a line connecting any two points in the object lies completely within the object)
- A Polyhedron is a connected mesh of simple planar polygons that encloses a finite amount of space

---

Slide Properties of polyhedrons:

- Every edge is shared by exactly two faces
- at least three edges meet at each vertex
- faces do not interpenetrate: they either touch at a common edge or not at all.
- Euler's formula for simple polyhedrons: $V + F - E = 2$ (E:Edges, F: Faces, V: Vertices)
- For non-simple polyhedrons: $V + F - E = 2 + H - 2G$ (G: holes in the polyhedron, H: holes in faces)

Slide Lecture 9:

- Shading
  - Toy physics and shading models
  - diffuse reflection
  - specular reflections
  - and everything in OpenGL

Slide Shading:

- Displaying Wireframe models is easy from a computational viewpoint
- But it creates lots of ambiguities that even perspective projection cannot remove
- If we model objects as solids, we would like them to look "normal". One way to produce such a normal view is to simulate the physical processes that influence their appearance (Ray Tracing). This is computationally very expensive.

- We need a cheaper way that gives us some realism but is easy to compute. This is shading.

Slide Types of shading:

- Remove hidden lines in wireframe models
- Flat Shading
- Smooth Shading
- Adding specular light
- Adding shadows
- Adding texture

Slide Toy-Physics for CG:

- There are two types of light sources: ambient light and point light sources.
- If all incident light is absorbed by a body, it only radiates with the so-called blackbody radiation that is only dependent of its temperature. We're dealing with cold bodys here, so blackbody radiation is ignored.
- Diffiuse Scattering occurs if light penetrates the surface of a body and is then re-radiated uniformily in all directions. Scattered lights interact strongly with the surface, so it is usually colored.
- Specular reflections occur in metal- or plastic-like surfaces. These are mirrorlike and highly directional.
- A typical surface displays a combination of both effects.

Slide Important vector tools for shading:

- The normal vector $\vec{m}$ to the surface P.
- The vector $\vec{v}$ from P to the viewer's eye.
- The vector $\vec{s}$ from P to the light source.
- The cosine of two vectors is the normalized dot-product.
- $\frac{\vec{a}\cdot\vec{b}}{|\vec{a}||\vec{b}|}$

Slide Calculating the diffuse component I_d:

- Diffuse scattering is uniform, so forget v (unless we do not see the surface, $v \cdot m < 0$)
- It depends on s vs. m.
- Lambert's Law: A surface receives the illumination from a light source that is proportional to the cosine of the angle between the normal of the surface and the direction to the light source.

- $I_d = I_s \rho_d \frac{\vec{s} \cdot \vec{m}}{|\vec{s}||\vec{m}|}$
- I_d is the intensity of the light source, ρ_d is the diffuse reflection coefficient.
- We do not want negative intensities, so we set negative values of the cosine term to zero.

Slide Specular reflection:

- The specular reflection component is I_d.
- specular reflection is not uniform, so it should depend on $\vec{s}$,$\vec{m}$ and $\vec{v}$.
- Several models have been developed for modeling specular reflection, the one OpenGL uses is the model by Phong (1975, Communications of the ACM 18: Illumination for Computer Generated Images)
- Phong: The light reflected in the direct mirror direction is the strongest. Light reflected in other directions is proportional to the f th power of the cosine to the mirror direction.

Slide Specular reflection (2):

- The mirror direction r can be found like this:
  $\vec{r} = -\vec{s} + 2\frac{(\vec{s} \cdot \vec{m})}{|\vec{m}|^2}\vec{m}$
- $I_{sp} = I_s \rho_s \left(\frac{\vec{r}}{|\vec{r}|} \cdot \frac{\vec{v}}{|\vec{v}|} \right)^f$
- Again, I_d is the intensity of the light source, ρ_{sp} is the specular reflection coefficient. f is determined experimentally and lies between 1 and 200.
- Finding $\vec{r}$ is computationally expensive.

Slide Avoid finding $\vec{r}$:

- Instead of finding the correct $\vec{r}$, compute the *halfway vector* between $\vec{s}$ and $\vec{v}$: $\vec{h} = \vec{s} + \vec{v}$.
- $\vec{h}$ gives the direction in which the brightest light is to be expected if all vectors are in the same plane.
- $I_{sp} = I_s \rho_s \left(\frac{\vec{h}}{|\vec{h}|} \cdot \frac{\vec{m}}{|\vec{m}|} \right)^f$
- The falloff of the cosine function is now a different one. But this can be compensated by chosing a different f.
- Of course all these models are not very realistic, but easy to compute.

Slide Ambient Light:

- Ambient light is a uniform background light that exists everywhere in the scene. It models the light that is usually reflected from surfaces.
- Its source has an intensity I_a. Every surface has an ambient reflection coefficient ρ_a (often equal to ρ_d).

- All light contributions combined: $I = I_a\rho_a + I_d\rho_d \times \text{lambert} + I_{sp}\rho_s \times \text{phong}^f$

Slide Color Light:

- It's easy to extend this model to colored light: Simply treat the three color components separately:
- $I_r = I_{ar}\rho_{ar} + I_{dr}\rho_{dr} \times \text{lambert} + I_{spr}\rho_{sr} \times \text{phong}^f$
  $I_g = I_{ag}\rho_{ag} + I_{dg}\rho_{dg} \times \text{lambert} + I_{spg}\rho_{sg} \times \text{phong}^f$
  $I_b = I_{ab}\rho_{ab} + I_{db}\rho_{db} \times \text{lambert} + I_{spb}\rho_{sb} \times \text{phong}^f$

Slide In OpenGL:

- Creating a light source:

```
GLfloat myLightPosition[]={3.0,6.0,5.0,1.0};
glLightfv(GL_LIGHT0,GL_POSITION,
     myLightPosition);
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
```

- OpenGL handles up to 8 light sources `LIGHT0` to `LIGHT7`.
- Giving a vector instead of a position creates a light source of infinite distance. This type of light source is called *directional* instead of *positional*.

Slide Colored Light:

- Creating a light source:

```
GLfloat amb0[]={0.2,0.4,0.6,1.0};
GLfloat diff0[]={0.8,0.9,0.5,1.0};
GLfloat spec0[]={1.0,0.8,1.0,1.0};
glLightfv(GL_LIGHT0,GL_AMBIENT,amb0);
glLightfv(GL_LIGHT0,GL_DIFFUSE,diff0);
glLightfv(GL_LIGHT0,GL_SPECULAR,spec0);
```

- Colors are specified in the RGBA model. A stands for *alpha*. For the moment, we set alpha to 1.0.

Slide Spot Lights:

- By default, OpenGL uses point light sources.

- Creating a spot light source:

```
glLightf(GL_LIGHT0,GL_SPOT_CUTOFF,45.0);
glLightfv(GL_LIGHT0,GL_SPOT_EXPONENT,4.0);
GLfloat dir[]={2.0,1.0,-4.0};
glLightfv(GL_LIGHT0,GL_SPOT_DIRECTION,dir);
```

Slide Other light properties:

- Light attenuation:

```
glLightf(GL_LIGHT0,
      GL_CONSTANT_ATTENUATION,2.0);
glLightf(GL_LIGHT0,
      GL_LINEAR_ATTENUATION,0.2);
glLightf(GL_LIGHT0,
      GL_QUADRATIC_ATTENUATION,0.1);
```

- Ambient Light:

```
GLfloat amb[]={0.2,0.3,0.1,1.0};
glLightModelfv(
      GL_LIGHT_MODEL_AMBIENT, amb);
```

Slide Other light properties:

- Recompute $\vec{v}$ for every point

```
glLightModeli(
      GL_LIGHT_MODEL_LOCAL_VIEWER,
      GL_TRUE);
```

- Faces are two-sided:

```
glLightModeli(
      GL_LIGHT_MODEL_TWO_SIDE,
      GL_TRUE);
```

Slide Material properties:

- Set the diffuse component for a surface:

```
GLfloat myDiffuse[]={0.8,0.2,0.0,1.0};
glMaterialfv(GL_FRONT,GL_DIFFUSE,myDiffuse);
```

- The first parameter choses the face: `GL_FRONT, GL_BACK, GL_FRONT_AND_BACK`
- The second parameter choses the coefficients: `GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR,GL_AMBIENT_AND_DIFFUSE`

Slide Lab Session tomorrow:

- Set up a scene
- Define some materials
- Set up some lights
- Play around

Slide Lecture 10:

- Smooth objects
  - Representation
  - Generic Shapes
- Flat vs. Smooth Shading
- Perspective and (pseudo) Depth

Slide Smooth Objects:

- Remember the n-gon?

```
for (i=0;i<N;i++)
{
  forward(L,1);
  turn(360/N);
}
```

Slide Mesh approximations:

- Smooth objects can be approximated with fine meshes.
- For shading, we want to preserve the information that these objects are actually smooth so that we can shade them “round”.
- The basic approach: Use a parametric representation of the object and “polygonalize” it. (also called “tesselation”)

Slide Representing Surfaces:

- Lecture 4: Representing Curves
  - Two principle ways of describing a curve: implicitly and parametrically
  - Implicitly: Give a function F so that $F(x, y) = 0$ for all points of the curve
  - The parametric form of a curve suggests the movement of a point through time.
- Lecture 5: Representing a planar patch: $P(s, t) = C + \vec{a}s + \vec{b}t$, $s, t \in [0, 1]$

Slide Representing surfaces:

- Parametric form: $P(u, v) = (X(u, v), Y(u, v), Z(u, v))$
- Keeping v fixed and let u vary: v-*contour*
- Keeping u fixed and let u vary: u-*contour*
- Implicit form: $F(x, y, z) = 0)$
- F is also called the *inside-outside-function*: $F < 0$:inside, $F = 0$ on the surface, $F > 0$ outside.

Slide Normal vectors of parametric surfaces:

- $\vec{p}(u, v)$ is the vector from the origin of the surface to $P(u, v)$.
- $\vec{n}(u_0, v_0)$ is the normal vector in surface point $P(u_0, v_0)$.

$$\vec{n}(u_0, v_0) = \left(\frac{\partial \vec{p}}{\partial u} \times \frac{\partial \vec{p}}{\delta v} \right) \Big|_{u=u_0, v=v_0}$$

Slide Normal vectors of parametric surfaces:

- As $p(u, v) = X(u, v)\vec{i} + Y(u, v)\vec{j} + Z(u, v)\vec{k}$:

$$\frac{\partial \vec{p}(u, v)}{\partial u} = \left(\frac{\partial X(u, v)}{\partial u}, \frac{\partial Y(u, v)}{\partial u}, \frac{\partial Z(u, v)}{\partial u} \right)$$

Slide Normal vectors of implicit surfaces:

- We can use the gradient ∇F of the surface as the normal vector:

$$\begin{aligned} \vec{n}(x_0, y_0, z_0) &= \nabla F|_{x=x_0, y=y_0, z=z_0} \\ &= \left(\frac{\partial F}{\partial x}, \frac{\partial F}{\partial y}, \frac{\partial F}{\partial z} \right) \Big|_{x=x_0, y=y_0, z=z_0} \end{aligned}$$

Slide Affine Transformations:

- We can apply affine transformation to the homogenous form of the representations: if $\tilde{P}(u,v) = (X(u,v), Y(u,v), Z(u,v), 1)^T$, then $M\tilde{P}(u,v))$ is the parametric representation under the transformation M.
- We can apply a transformation to the implicit form $F(\tilde{P})$: $F'(\tilde{P}) = F(M^{-1}\tilde{P})$
- The normal vector of the transformed surface is $M^{-T}\vec{n}(u,v)$

Slide Some generic shapes:

- Sphere:
  - $F(x,y,z) = x^2 + y^2 + z^2 - 1$
  - $P(u,v) = (\cos(v)\cos(u), \cos(v)\sin(u), \sin(v))$
  - u-contours are called *meridians*, v-contours are called *parallels*
- Tapered Cylinder:
  - $F(x,y,z) = x^2 + y^2 - (1 + (s-1)z)^2$ for $0 < z < 1$
  - $P(u,v) = ((1 + (s-1)v)\cos(u), (1 + (s-1)v)\sin(u), v)$
  - $s = 1$: Cylinder, $s = 0$: Cone

Slide Shading:

- Flat shading: Compute the color for each face, fill the entire face with the color
- Flat shading is OK if light sources are far away
- Flat shading espechially looks bad on approximated smooth objects.
- in OpenGL:`glShadeModel(GL_FLAT);`

Slide Smooth Shading:

- Gouraud Shading: Compute a different color for every pixel.
- For each scanline at y_s compute $color_{left}$ by linear interpolation between the color of the top and bottom of the left edge.
- Compute $color_{right}$ the same way.
- Then fill the scanline by linear interpolation between $color_{left}$ and $colorright$.
- in OpenGL:`glShadeModel(GL_SMOOTH);`

Slide Better Smooth Shading:

- Phong Shading: Compute a different normal vector for every pixel.
- Instead of interpolating the colors, interpolate the normal vectors
- in OpenGL: not implemented

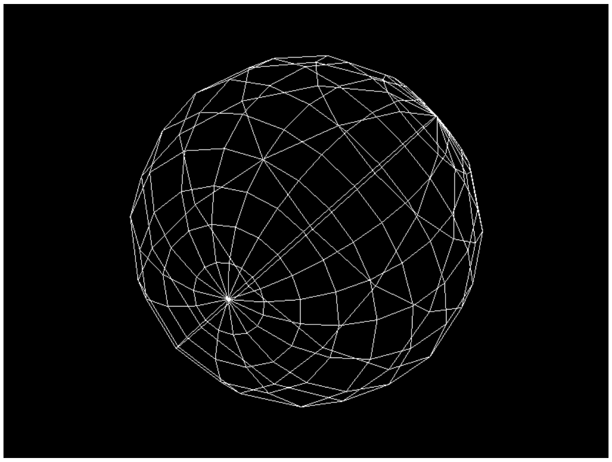
Slide Removing hidden surfaces:

- Depth Buffer: Stores a value for every pixel
- During shading: For each pixel compute a pseudodepth.
- Only draw the pixel if its pseudodepth is lower, and update the pseudodepth if the pixel is drawn.
- Again, compute the correct pseudodepth for the endpoints of the scanline and use interpolation in between.

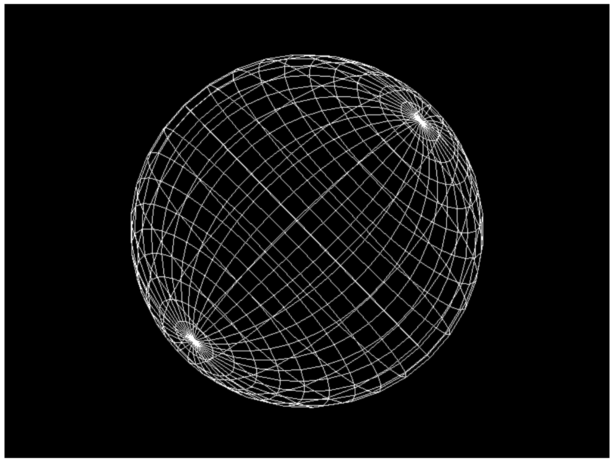
Slide Lecture 11:

- Smooth objects demo
- Flat vs. Smooth Shading demo
- Perspective and (pseudo) Depth

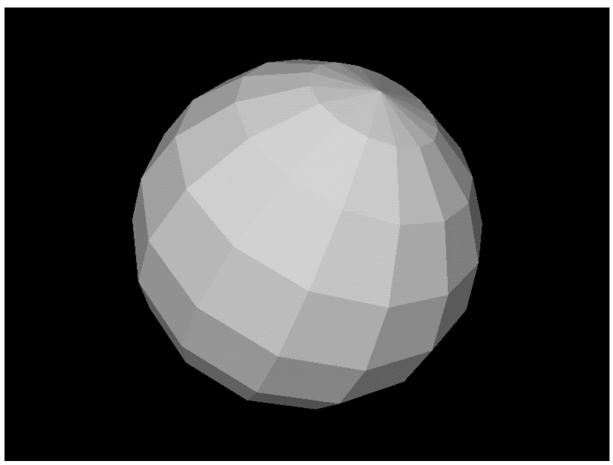
Slide Insert Demos Here:



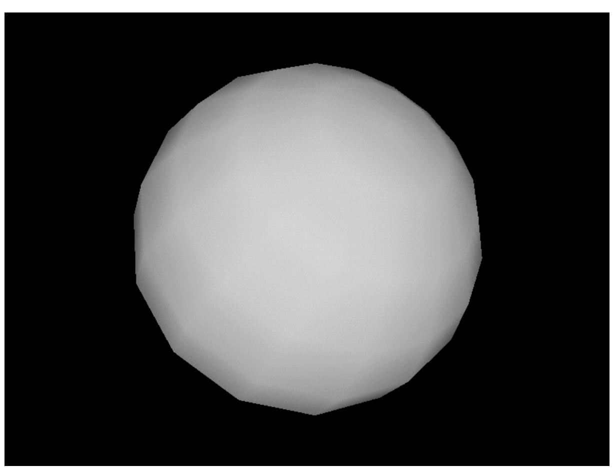
Slide Insert Demos Here:



Slide Insert Demos Here:



Slide Insert Demos Here:



Slide Removing hidden surfaces:

- Depth Buffer: Stores a value for every pixel
- During shading: For each pixel compute a pseudodepth.
- Only draw the pixel if its pseudodepth is lower, and update the pseudodepth if the pixel is drawn.
- Again, compute the correct pseudodepth for the endpoints of the scanline and use interpolation in between.

Slide What is pseudodepth?:

- A perspective projection projects a 3D point to a 2D point
- The parallel projection is the most simple one. It removes the z-Component.
- A better perspective projection is the following:

$$(x^*, y^*) = \left(N\frac{P_x}{-P_z}, N\frac{P_y}{-P_z}\right)$$

N is the distance from the eye to the near plane.

Slide What is pseudodepth?:

- Pseudodepth should be lower if a point is in front of another point.
- Unfortunately, the projection removes this information.
- We could use P_z directly.
- But it's more convenient to set the pseudodepth to a fixed interval, i.e. $-1 \ldots 1$.
- And it's convenient to use the same denominator $-P_z$.

Slide What is pseudodepth?:

- So we can use:

$$(x^*, y^*, z^*) = \left(N\frac{P_x}{-P_z}, N\frac{P_y}{-P_z}, \frac{aP_z + b}{-P_z}\right)$$

for the right a and b.

Slide Pseudodepth in a projection matrix:

- This projection matrix computes the pseudodepth and the perspective projection at the same time:

$$P = \begin{pmatrix} N & 0 & 0 & 0 \\ 0 & N & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Slide Lecture 12:

- Pixmaps
- Colors
- Texture

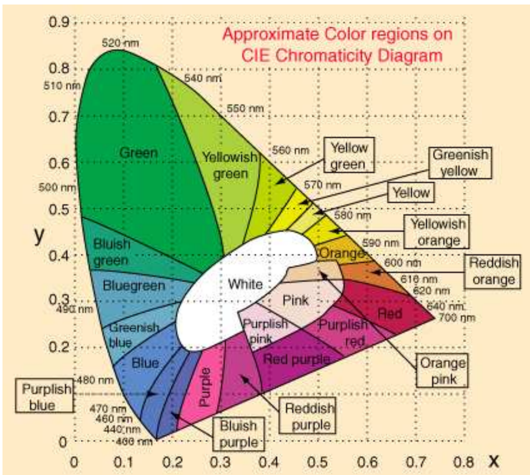
Slide Pixmaps:

- From Lecture 2: A Pixel is a point sample and a pixmap (or pixel map or "bitmap") is created by sampling an original discrete points. In order to restore an image from pixels, we have to apply areconstruction filter.
- Reconstruction filters are e.g. Box, Linear, Cubic, Gaussian...
- OpenGL is another method to create these point samples: for every pixel in the viewport window, OpenGL determines its color value.

Slide Pixmaps:

- Internally, OpenGL stores these pixmaps in *buffers*.
- The call to `glutInitDisplayMode()` allocates the basic draw buffer(s).

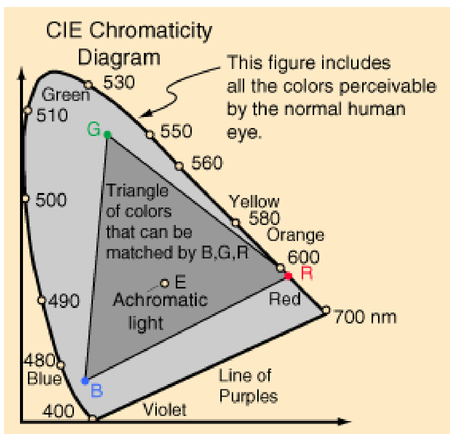
Slide CIE Cromaticity Diagram:



Slide Colors:

- Visible light is a contiuum, so there is no "natural" way to represent color
- RGB color model
  - Inspired by human perception
  - three spectral components: red, green, blue
  - binary representation of the component values, different standards
  - example: 16-bit RGB (565): one short, 5 bits for red and blue, 6 bits for green.

Slide RGB in CIE Cromaticity Diagram:



Slide Colors:

- Y/Cr/Cb
  - based on the CIE Cromaticity Diagram
  - used for TV applications: compatible with old B/W TV standards
  - Y: greyscale component, Cr: red-green-component, Cb: blue-green-component
  - possibility to reduce bandwith for color "signal"

Slide Colors:

- HSI model
  - hue: color (i.e. dominant wavelength), saturation: ratio between white and color, intensity: ratio between black and color
  - good for computer vision applications

Slide Colors:

- CYM(K) model
  - subtractive color model: white light is filtered, spectral components are removed.
  - C: cyan (removes red) Y: yellow (removes blue) M: magenta (removes green)
  - K: coal (i.e. black) removes everything.
  - often used in print production

Slide Colors:

- Conversion between different color models (and output devices) often leads to different colors. In order to get the "right" color, the devices have to be color-corrected. This is the task of a color management system.

Slide Never The Same Color:

- In pixmaps, colors are represented using binary values. This leads to problems:
  - quantization errors: when using few bits per pixel
  - minimum and maximum values: clamping
- But other things go wrong too.
- Display devices react nonlinearily: A intensity value of128 is less than half as bright than 255.

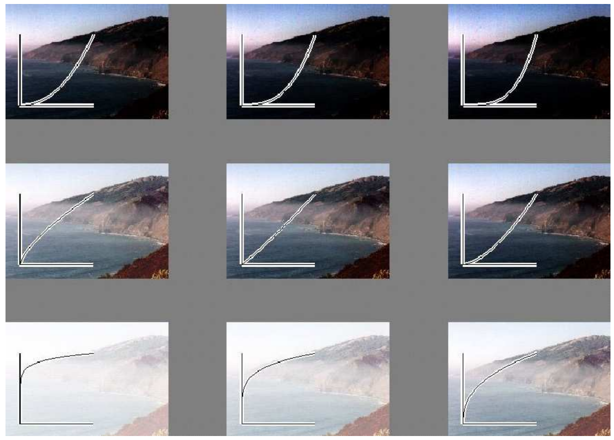
Slide Gamma correction:

- The intensity of the display devices is roughly a power function:

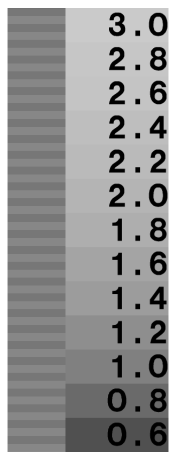
$$i_D \approx \left(\frac{i}{255}\right)^\gamma$$

- γ is usually in the range of $1.7 \ldots 2.5$.

Slide Different gamma values:



Slide What's the gamma?:



(from `http://www.graphics.cornell.edu/ westin/gamma/gamma.html`

Slide What's the A in RGBA?:

- OpenGL represents pixmaps internally using 4 values per pixel, RGB and A.
- The A stands for α, i.e. Alpha and indicates the transparent regions of a pixmap.
- α is a measure of opacity, $(1 - \alpha)$ is transparency

  $\alpha = 1$ Pixel is fully opaque

  $\alpha = 0$ Pixel is fully transparent

  $0 < \alpha < 1$ Pixel is semi transparent

Slide Compositing:

- The alpha values of a pixmap are called the alpha matte of the pixmap

- The process of merging two images with alpha mattes is called compositing or alpha blending.
- Given two pixels F (foreground) and B (background) and α for the foreground pixel.
- $B_{new} = (1-\alpha)B_{old} + \alpha F$
- $B_{new} = B_{old} + \alpha(F - B_{old})$
- OpenGL uses this in its blending functions.

Slide Associated Color:

- Treating alpha and colors separately gives strange effects when filtering or interpolating

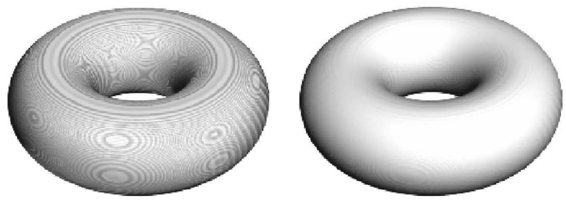


Figure 8. Left: Artifacts of separate interpolation of colors and opacity. Right: Improved using opacity-weighted color interpolation.

- But storing the pixels already premultiplied with their opcaity removes the effect. This is called associated color or opacity-weighted color.

Slide Associated Color Compositing:

- Associated color: $\tilde{F} = \alpha F$
- Compositing with associated color: $\tilde{B}_{new} = (1-\alpha)\tilde{B}_{old} + \tilde{F}$
- and computing the new alpha: $\beta_{new} = (1-\alpha)\beta_{old} + \alpha$
- β is the α of the background pixel.

Slide Gamma Correction ?:

- Do you gamma-correct alpha ? (Does alpha need a gamma correction?)
- Do you alpha-blend gamma? (Does an alpha blending change gamma ?)
- Alpha is never gamma-corrected. Gamma-correction only applies to the “real” colors.

Slide Textures:

- Textures are pixmaps that are applied to faces.
- They can be “displayed” in all the different surface coefficients of the object, i.e. intensity or reflection coefficients.
- Texture pixmaps can either be stored beforehand or created by the program (procedural textures).

Slide Textures:

- OpenGL needs to know which part of the texture belongs to which part of the face. Therefore, the vertices of the object are both specified in 3D worldspace and in texture coordinates. When rendering, OpenGL uses interpolated texture coordinates to find the “right” part of the texture.

Slide Object and Texture Space:

- A texture is a pixmap. It has a simple 2d coordinate system.
- A surface of an object has coordinates in 3d space.
- Question: how to find the right 2d coordinates for a pixel in 3d space. (This is yet another projection.)

Slide Object and Texture Space:

- OpenGL knows several texture generation modes:
  - `GL_OBJECT_LINEAR`: Texture coordinates are linear combinations of the vertex coordinates.
  - `GL_EYE_LINEAR`: Texture coordinates are computed relative to the eye coordinates.
  - `GL_SPHERE_MAP`

Slide A Sphere Map:

www.debevec.org