

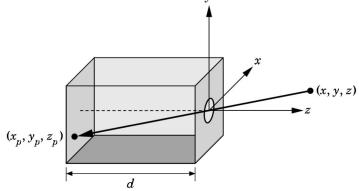
CS3241

INTRO

Image formation

Elements Objects | Viewer | Light sources | Materials

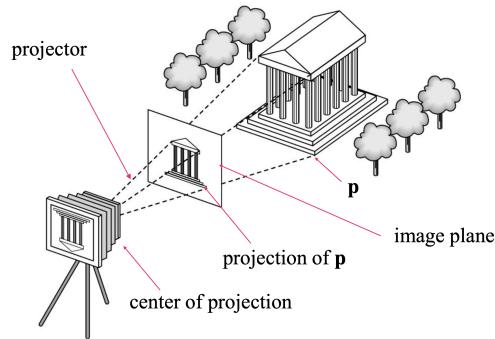
Pinhole camera



Use trigonometry to find **projection** of 3D point at (x, y, z)

$$x_p = -dx/z \quad y_p = -dy/z \quad z_p = -d$$

Synthetic camera model



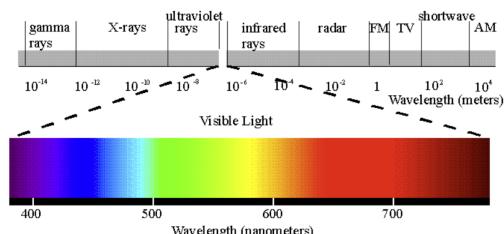
Luminance and color

Monochromatic images Grayscale values stored

Color images Hue | Saturation | Lightness

Light

- Visible spectrum with wavelength in range 350-750 nanometers
- Long wavelengths are red, short wavelengths are blue



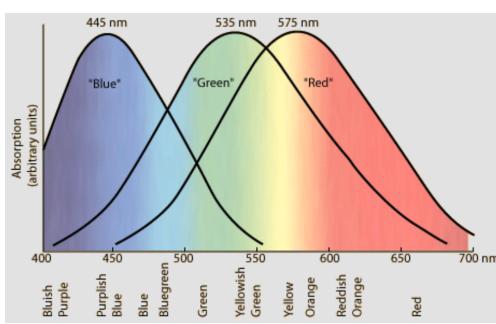
Three-color theory

Human visual system has two types of light sensors

- Rods: good for monochromatic, night vision
- Cones: sensitive to color

Cones

- Three types of cones, sensitive to different wavelengths
- Use RGB to excite the cones
- Humans are more sensitive to green, then red, then blue. If there are only 8 bits total for RGB, allocate 3 : 3 : 2 for R:G:B

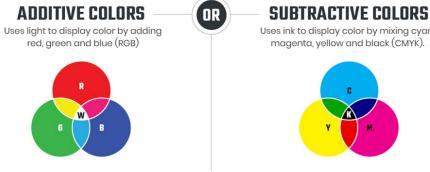


Additive and subtractive color

Additive color Add amounts of RGB

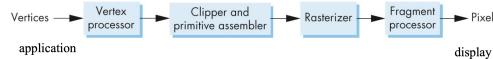
Subtractive color

- Filter white light with cyan, magenta, and yellow filters
- There are 2 ways to think about subtractive color:
 - What does it filter? Cyan = -Red; Magenta = -Green; Yellow = -Blue
 - What color is remaining? Refer to image below



Polygon rasterization

- Image is generated by processing a mesh of polygons
- Can be modelled as a pipeline



ELEMENTARY OPENGL

Function format

`glVertex3f(x, y, z)`

- gl means it belongs to GL library
- Vertex is the function name
- 3 is the number of dimensions
- f means the parameters are floats

Coordinate systems

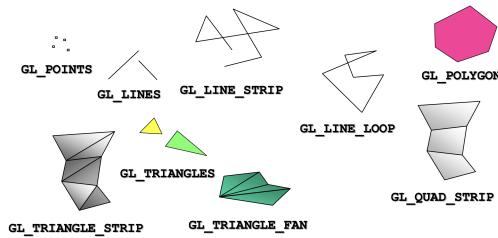
- Units in `glVertex` are determined by application; called object coords
- Camera is positioned in world coords
- Viewing specifications are specified in camera (eye) coords
- Internally, OpenGL converts vertices to camera (eye) coords, then to window coords

Camera

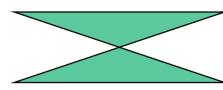
Defaults:

- Placed at origin in world space
- Faces negative z direction
- Viewing volume is a box, centered at origin, with side length 2

OpenGL primitives



- Polygons are displayed correctly if they are
 - Simple: edges do not cross
 - Convex: all points on line segment between any two points in the polygon or on the boundary are also in the polygon
 - Flat: all vertices are in the same plane
- If some condition is not true, OpenGL will still produce output, but it may not be what is desired
- Triangles satisfy all conditions



non-simple polygon

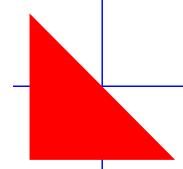


non-convex polygon

Usage (right-angled triangle)

- Use `glBegin` and `glEnd` to mark the start/end of declaring vertices for a polygon
- Try to write in the display callback, so it is run once per display callback

```
glBegin(GL_TRIANGLES);
  glColor3f(1, 0, 0);
  glVertex3f(1, -1, 0);
  glVertex3f(-1, 1, 0);
  glVertex3f(-1, -1, 0);
glEnd();
```



Number of entities formed Assume pass in N vertices.

Primitive	Result
GL_POINTS	N points
GL_LINES	$\frac{N}{2}$ lines
GL_LINE_STRIP	$N - 1$ lines
GL_LINE_LOOP	N lines
GL_TRIANGLES	$\frac{N}{3}$ triangles
GL_TRIANGLE_STRIP	$N - 2$ triangles
GL_TRIANGLE_FAN	$N - 2$ triangles
GL_QUADS	$\frac{N}{4}$ quads
GL_QUAD_STRIP	$\frac{N}{2} - 1$ quads
GL_POLYGON	1 polygon

Color

- Each color component/channel stored separately in frame buffer
- Usually 8 bits per component
- `glColor3f` expects range from 0.0 to 1.0
- `glColor3ub` expects range from 0 to 255
- Is part of OpenGL state, will be used until changed

Smooth color

- Default is smooth shading
 - Interpolates vertex colors across visible polygons
 - `glShadeModel(GL_SMOOTH)`
- Alternative is flat shading
 - Color of first vertex determines fill color
 - `glShadeModel(GL_FLAT)`

Viewport

- Specifies the drawing area within the window
- `glViewport(x, y, width, height)`
- `x, y` specify lower left corner of viewport rectangle
- Can have multiple, can overlap
- Use `glClear(GL_COLOR_BUFFER_BIT)` to clear entire window

INPUT AND INTERACTION

Input

- Devices contain a trigger that sends a signal to the OS
- When triggered,
 - Return info (**measure**) to the system
 - Generate event, put into event queue
 - The event queue starts after `glutMainLoop()` is called in the `main()` function
- e.g. Window (resize, expose, minimize) | Mouse (click, motion) | Keyboard (press, release) | Idle (if no event)

Callbacks

A callback is defined for each event

- `glutDisplayFunc` (required for GLUT programs)
 - `void glutDisplayFunc(void (*func)(void))`
- `glutMouseFunc`
 - `void glutMouseFunc(void (*func)(int button, int state, int x, int y))`
 - `button` is either left, middle, or right
 - `state` is either down or up
- `glutReshapeFunc` (window resizing)
 - `void glutReshapeFunc(void (*func)(int width, int height))`
- `glutKeyboardFunc`
 - `void glutKeyboardFunc(void (*func)(unsigned char key, int x, int y))`
 - Returns ASCII code of depressed key and mouse location
- `glutIdleFunc` (useful for animations)
 - `void glutIdleFunc(void (*func)(void))`
- `glutMotionFunc` (mouse moving while button is pressed)
 - `void glutMotionFunc(void (*func)(int x, int y))`
- `glutPassiveMotionFunc` (mouse moving without button press)
 - `void glutPassiveMotionFunc(void (*func)(int x, int y))`

Display callback

Called when

- Window is first opened | reshaped | uncovered | restored from minimized state
- User program wants to change the display

Posting redisplays

- Many events may invoke display callback function
- Hence, display callback may run multiple times in a single event loop pass
- Use `glutPostRedisplay()` instead to set a flag that tells GLUT to execute the display call back function at the end of a single event loop

Reshape callback

- Good place to put viewing functions, since it is invoked when window is first opened
- Redisplay is posted automatically at the end of execution of the callback

Double buffering

Issue

- Drawing of info in frame buffer is decoupled from displaying the frame buffer
- We may see partially rendered frames on the display, that includes info from different frames (known as flickering)

Fix

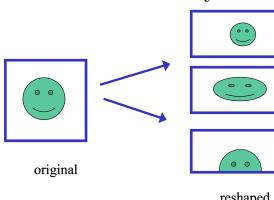
- Maintain two buffers. Front buffer is displayed, while back buffer is written to
- Swap buffers at end of display callback (after writing is complete)
- Turn on: `glutInitDisplayMode(... | GLUT_DOUBLE)`
- Swap buffers: `glutSwapBuffers()`

Positioning

- OpenGL has origin at bottom left corner
- But for window system, mouse callback, motion callback, origin at top left corner
- Remember to invert `y`-coord when necessary
 $y_{opengl} = h - 1 - y_{win}$

Reshaping a window

There are a few ways to redraw the scene.



Obtaining OpenGL state variables

Use `void glGetIntegerv(GLenum pname, GLint *params)`

- `pname` is the state name
- `params` is used to return the state variables

Special keys

- Special keys have constants defined in `glut.h`:
 - Function key 1: `GLUT_KEY_F1`
 - Up arrow key: `GLUT_KEY_UP`
- Check if modifier keys are depressed
 - `if(glutGetModifiers() == GLUT_ACTIVE_CTRL)`
 - Can emulate three-button mouse with one- or two-button mice

GEOMETRY

Elements

Scalar Fully described using magnitude

Vector Has direction and magnitude

Vector space Operations:

- Scalar-vector multiplication
- Vector-vector addition

Point Location in space

- Point-point subtraction yields vector

Affine space Point + vector space. Operations:

- Scalar-vector multiplication
- Vector-vector addition
- Point-vector addition
- Scalar-scalar operations
- For a point P , define $1 \cdot P = P$, $0 \cdot P = \vec{0}$

Products

Dot product

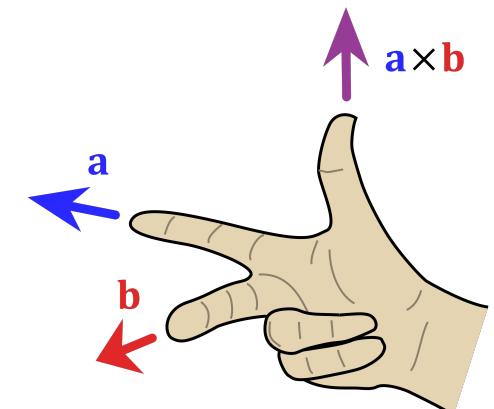
$$A \cdot B = |A||B| \cos \theta$$

where θ is the angle between A and B .

Cross product (3D vectors)

$$A \times B = |A||B|(\sin \theta) \vec{n}$$

where \vec{n} is a unit vector perpendicular to A and B , given by the right hand rule:



Alternatively,

$$\begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \times \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} a_2 b_3 - a_3 b_2 \\ -(a_1 b_3 - a_3 b_1) \\ a_1 b_2 - a_2 b_1 \end{pmatrix}$$

Lines

For specific P_0, d , a line is the set of all points of the form

$$P(\alpha) = P_0 + \alpha d$$

Representations

- Parametric form (using a point and a direction vector)
$$P(\alpha) = P_0 + \alpha d$$
- Parametric form (using 2 points)
$$P(\alpha) = \alpha P_0 + (1 - \alpha) P_1$$
- Explicit: $y = mx + h$
- Implicit: $ax + by + c = 0$

Ray

- Using the 1 point parametric definition of a line, a ray is a line with $\alpha \geq 0$.

$$P(\alpha) = P_0 + \alpha d$$

Line segment

- A line segment is bounded by 2 distinct points on a line.
 - Using the 2 point parametric definition of a line, the line segment between P_0 and P_1 is a line with $0 \leq \alpha \leq 1$
- $$P(\alpha) = \alpha P_0 + (1 - \alpha)P_1$$

Planes

Defined either by

- A point and two vectors, or
- $$\text{Plane}(\alpha, \beta) = R + \alpha \vec{u} + \beta \vec{v}$$

- Three points

$$\text{Plane}(\alpha, \beta) = R + \alpha(Q - R) + \beta(P - Q)$$

- A point and a normal to the plane

$$(P - R) \cdot \vec{n} = 0$$

where P is any point on the plane, and $\vec{n} = \vec{u} \times \vec{v}$.

Representation

Coordinate system (vector spaces)

- A basis $\{v_1, v_2, \dots, v_n\}$ is a set of linearly independent vectors
- A vector can be expressed as $v = \alpha_1 v_1 + \alpha_2 v_2 + \dots + \alpha_n v_n$
- The list of scalars $\{\alpha_1, \alpha_2, \dots, \alpha_n\}$ is the representation of v with respect to the given basis
- Can write the representation as a row or column vector

$$\vec{a} = (\alpha_1 \quad \alpha_2 \quad \dots \quad \alpha_n)^T = \begin{pmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_n \end{pmatrix}$$

Frames (affine spaces)

- Coordinate system is only able to specify vectors in 3D space
 - Need a reference point to represent a point
 - Frame is determined by tuple (P_0, v_1, v_2, v_3)
 - Vector is still written as
- $$v = \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3$$
- $$= (v_1 \quad v_2 \quad v_3 \quad P_0) (\alpha_1 \quad \alpha_2 \quad \alpha_3 \quad 0)^T$$
- Point can now be written as
- $$P = P_0 + \beta_1 v_1 + \beta_2 v_2 + \beta_3 v_3$$
- $$= (v_1 \quad v_2 \quad v_3 \quad P_0) (\beta_1 \quad \beta_2 \quad \beta_3 \quad 1)^T$$

Homogeneous coordinates

Given a coordinate vector $(x \quad y \quad z \quad w)^T$,

- If $d = 0$ it is a vector
- Else it represents the point $(\frac{x}{w}, \frac{y}{w}, \frac{z}{w})$.

Usefulness

- Standardized coordinate system that can represent both vectors and points simultaneously
- All standard transformation (rotation, translation, scaling) can be implemented with 4 by 4 matrix multiplications
- Allows perspective projection (and perspective division) using matrix multiplication

Planar test for polygon in 3D space

Method 1:

- Suppose polygon is defined by vertices v_1, v_2, \dots, v_n
- Assume three consecutive vertices are not collinear

- Find normal vectors of triangle v_1, v_2, v_3 , triangle v_3, v_4, v_5 , and so on until triangle v_{n-2}, v_{n-1}, v_n

- Polygon is planar iff all the normal vectors are parallel

Method 2:

- Suppose polygon is defined by vertices v_1, v_2, \dots, v_n
- Assume three consecutive vertices are not collinear
- Form equation of plane using v_1, v_2, v_3
- Substitute in the rest of the points. Polygon is planar iff all the points agree with the equation

Convex test for polygon in xy plane

- Suppose polygon is defined by vertices v_1, v_2, \dots, v_n
- Assume three consecutive vertices are not collinear
- Compute normal vectors of triangle v_1, v_2, v_3 , triangle v_2, v_3, v_4 , and so on until triangle v_n, v_1, v_2
- Polygon is convex iff all normal vectors are in the same direction (not just parallel)

TRANSFORMATIONS

Affine transformations

- Preserves lines, so we can just transform endpoints of line segments and let implementation draw line segments
- Rigid body transformations: rotation, translation
- Scaling, shearing

Translation

- Move/translate/displace point to new location
- Displacement determined by vector d

Coordinate representation

Define

$$p = (x \quad y \quad z \quad 1)^T$$

$$p' = (x' \quad y' \quad z' \quad 1)^T$$

$$d = (d_x \quad d_y \quad d_z \quad 0)^T$$

Here, $p' = p + d$, so

$$x' = x + d_x$$

$$y' = y + d_y$$

$$z' = z + d_z$$

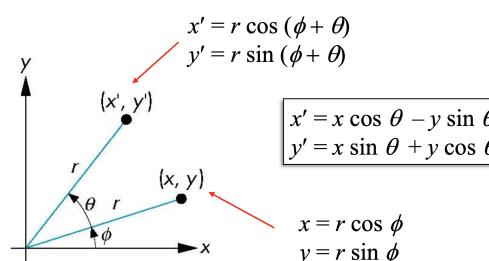
Matrix representation

We can also get $p' = Tp$ with the following matrix T :

$$T = T(d_x, d_y, d_z) = \begin{pmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotation (2D / about z -axis)

Consider anti-clockwise rotation about origin by θ degrees.



Note that this is equivalent to rotation about z -axis, since z is kept unchanged.

Coordinate representation

$$x' = x \cos \theta - y \sin \theta$$

$$y' = x \sin \theta + y \cos \theta$$

$$z' = z$$

Matrix representation

We have $p' = R_z(\theta)p$, where

$$R_z(\theta) = \begin{pmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotation about x or y -axis

Similarly, for rotation about x or y -axis, x or y is kept unchanged. Again, θ is measured anti-clockwise.

$$R_x(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$R_y(\theta) = \begin{pmatrix} \cos \theta & 0 & \sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Note that rotations do not commute.

Scaling

Coordinate representation

$$x' = s_x x$$

$$y' = s_y y$$

$$z' = s_z z$$

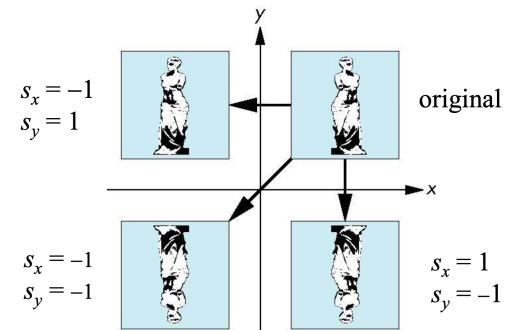
Matrix representation

We have $p' = S(s_x, s_y, s_z)p$, where

$$S(s_x, s_y, s_z) = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Reflection

Corresponds to negative scale factors



Concatenation

- We can form arbitrary affine transformation matrices by taking the product of the rotation, translation, scaling matrices

- $p' = DCBA(p)$ applies transformation A first, then B, C, D

- Note:

$$(ABC)^T = C^T B^T A^T$$

Inverses

Use geometric observations to get inverses:

- Translation:

$$T^{-1}(d_x, d_y, d_z) = T(-d_x, -d_y, -d_z)$$

- Rotation: $R^{-1}(\theta) = R(-\theta) = R^T$

- Scaling:

$$S^{-1}(s_x, s_y, s_z) = S \left(\frac{1}{s_x}, \frac{1}{s_y}, \frac{1}{s_z} \right)$$

- Concatenation:

$$(ABC)^{-1} = C^{-1}B^{-1}A^{-1}$$

General rotation in 3D

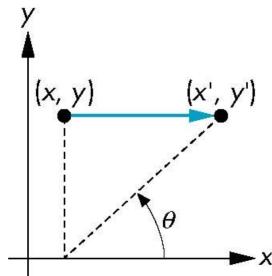
- Translate so that rotation axis passes through origin
- Rotate about the x -axis, so that the rotation axis lies in the $x - z$ plane
- Rotate about the y -axis, so that the rotation axis lies along the z -axis
- Perform the desired rotation by θ , about the z -axis
- Apply inverse of step 3
- Apply inverse of step 2
- Apply inverse of step 1

Instance transformation

- Draw object centered at origin, oriented with the axis, and at a standard size
- Scale, rotate, then translate

Shearing

Consider the following shear along the x -axis.



Coordinate representation

$$\begin{aligned} x' &= x + y \cot \theta \\ y' &= y \\ z' &= z \end{aligned}$$

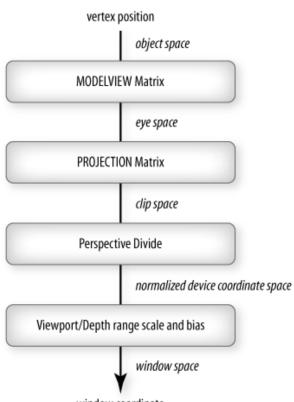
Matrix representation

We have $p' = H(\theta)p$, where

$$H(\theta) = \begin{pmatrix} 1 & \cot \theta & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

OpenGL transformations

- Model-View: `glMatrixMode(GL_MODELVIEW)`
- Projection: `glMatrixMode(GL_PROJECTION)`



Current Transformation Matrix (CTM)

- Part of OpenGL state
- Can load/post-multiply matrices
- Since only post-multiply allowed, the last operation specified is actually the first transformation applied to vertices
 - When drawing a vertex, you pre-multiply the CTM to the point
 - Hence the last operation is first applied
- OpenGL stores has a different matrix for Model-View and Projection, which is loaded using `glMatrixMode`

Model-View matrix

- (Modelling transformation) Transforms objects into the world space
- (View transformation) Positions the camera, aligning camera and world coordinate frames
- Projection matrix is used to defined view volume

Operations

- Load identity matrix: `glLoadIdentity()`
- Rotation: `glRotatef(theta, vx, vy, vz)`
 - `theta` is in degrees
 - `(vx, vy, vz)` is axis of rotation
 - Convert radian to degrees by multiplying by $\frac{180}{\pi}$
- Translation: `glTranslatef(dx, dy, dz)`
- Scale: `glScalef(sx, sy, sz)`
- Each has `(f)loat` and `(d)ouble` format

Arbitrary matrices

- Load arbitrary matrix: `glLoadMatrixf(m)`
- Post-multiply by arbitrary matrix: `glMultMatrixf(m)`

Matrix stacks

- Might want to save transformation matrices for later use
- OpenGL maintains a stack for each matrix mode
- `glPushMatrix()` pushes a copy of the top of the stack onto it
- `glPopMatrix()` pops the stack

CAMERA & VIEWING

Planar geometric projections

- Project onto a plane
- Preserves lines, so we can just transform endpoints of line segments and let implementation draw line segments
- May not preserve angles
- Non-planar projection surfaces needed for applications such as map construction

Examples Perspective projection, parallel projection

Projectors are lines that either

- Converge at center of projection
- Are parallel

Orthographic projection

- Special case of parallel projection, where projectors are orthogonal to projection surface
- In the default orthographic view, points are projected along the z -axis, onto the plane $z = 0$

Perspective projection

- Projectors converge at center of projection
- Objects further from viewer are projected smaller (diminution)
- Equal distances along a line are not projected into equal distances (non-uniform foreshortening)
- Angles preserved only in planes which are parallel to the projection plane

Projection matrix

- Basically selects a lens
- Sets view volume/clipping volume, defining the region of the 3D world that appears in the viewport

Spaces

- Refer to slides.
- Note that world space is after the model transformation, but since OpenGL combines the model and view transformations into a single matrix, this space is not explicitly in the pipeline

Model transformation

- Prior to this, objects are typically at the origin, possibly at unit size
- This transformation uses rotation, scaling, translation to put the object in its intended position in the world

gluLookAt() / View transformation

Usage

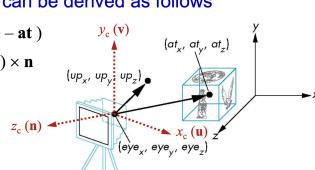
- `gluLookAt(eyeX, eyeY, eyeZ, atX, atY, atZ, upX, upY, upZ)`
- `(eyeX, eyeY, eyeZ)` specifies the position of the camera
- `(atX, atY, atZ)` specifies the point that the camera should look at
- `up-vector` is the vector from the eye position to `(upX, upY, upZ)`.
- Note that camera should face negative z direction

up-vector

- Lies anywhere on the original $y - z$ plane, but should not lie on z -axis (view direction)
- Does not need to be perpendicular to view direction (negative z direction)
- Cross product of view direction and up-vector gives unique x -axis
- Cross product of x -axis and view direction gives unique y -axis
- Cross product of x -axis and y -axis gives unique z -axis

The vectors u, v, n can be derived as follows

- $n = \text{normalize}(eye - at)$
- $u = \text{normalize}(up) \times n$
- $v = n \times u$



Concept

- Positions camera at required location and orientation wrt world frame
- Internally generates transformation matrix that can be used to express all points in world frame wrt camera frame (i.e. view transformation)

$$M_{view} = RT$$

$$= \begin{pmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Translation T moves camera to origin
- Rotation R aligns camera frame axes to world frame axes
- u, v, n defined in picture above
- e is the eye position

Transformation of normal vectors

Let a plane equation be given by $n \cdot v = 0$. Let M_t be the upper-left 3 by 3 submatrix of the model-view transformation matrix M .

$$n \cdot v = 0$$

$$n^T v = 0$$

$$n^T M_t^{-1} M_t v = 0$$

But notice that $M_t v$ produces a vector in eye coordinates. Now we have

$$n^T M_t^{-1} \cdot v_{eye} = 0$$

which is the equation of a plane, which was transformed by M_t . Hence, the new normal is $n^T M_t^{-1}$. This is a row vector pre-multiplied to a matrix, which gives a row vector. To get a column vector, transpose this, getting $(M_t^{-1})^T n$.

Result

- Normal vectors are pre-multiplied by $(M_t^{-1})^T$, where M_t is the upper-left 3 by 3 submatrix of M .
- If M_t is a rotation, then $M_n = M_t$, since $(M_t)^{-1} = (M_t)^T$.

Projection matrix

Projections

- Specifies view volume for projection wrt camera frame
- 3D region of scene to appear in the rendered image

Matrix

- Maps points in view volume to canonical view volume
- Canonical view volume is the 2 by 2 by 2 cube bounded by the planes $x = \pm 1, y = \pm 1, z = \pm 1$, also called Normalized Device Coordinates
- Preserves depth order (in z -coordinate)
- Preserves lines
- This intermediate canonical view volume is then mapped to the viewport

Orthographic projection

Example code:

```
// Set the matrix mode
glMatrixMode(GL_PROJECTION);
// Reset the matrix and apply default
// view volume
glLoadIdentity();
glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
```

glOrtho

- `glOrtho(left, right, bottom, top, near, far)`
- `left, right, bottom, top` specify coords of clipping planes
- `near` and `far` specify displacements from $z = 0$ to near/far clipping planes in the -ve z -direction. Since the camera faces the -ve z direction, a -ve number means that the plane is behind the camera.
- `gluOrtho2D(left, right, bottom, top)` is equivalent to `glOrtho(left, right, bottom, top, -1, 1)`

Transformation matrix

$$\begin{pmatrix} 2 & 0 & 0 & -(right + left) \\ right - left & 2 & 0 & -(right - left) \\ 0 & top - btm & 0 & -(top + btm) \\ 0 & 0 & -2 & -(top - btm) \\ 0 & 0 & far - near & -(far + near) \\ 0 & 0 & 0 & far - near \end{pmatrix}$$

$$= S \left(\frac{2}{right - left}, \frac{2}{top - btm}, \frac{2}{near - far} \right).$$

$$T \left(\frac{-(right + left)}{2}, \frac{-(top + btm)}{2}, \frac{far + near}{2} \right)$$

- Note that $z = -near$ is mapped to $z = -1$, and $z = -far$ to $z = 1$

Viewport transformation

- Canonical view volume is mapped to the viewport (from NDC to window coords). Consider viewport of size (w, h) and bottom left corner (x_{vp}, y_{vp}) .

$$\frac{x_{NDC} - (-1)}{2} = \frac{x_{win} - x_{vp}}{w} \Rightarrow x_{win} = x_{vp} + \frac{w(x_{NDC} + 1)}{2}$$

$$\frac{y_{NDC} - (-1)}{2} = \frac{y_{win} - y_{vp}}{h} \Rightarrow y_{win} = y_{vp} + \frac{h(y_{NDC} + 1)}{2}$$

$$z_{win} = \frac{z_{NDC} + 1}{2}$$

- By default, $0 \leq z_{win} \leq 1$, for z -buffer hidden surface removal

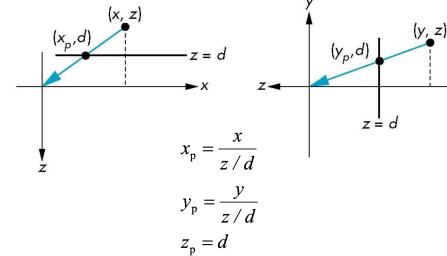
Perspective projection

Example

- Center of projection at origin
- Projection plane is $z = d$ where $d < 0$

Coordinates

Consider top and side views



Matrix We have $p = Mq$, where

$$p = \begin{pmatrix} x \\ y \\ z \\ z/d \end{pmatrix} \quad M = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{pmatrix} \quad q = \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Perspective division

- Notice that in the transformed point p , $z/d \neq 1$, so we divide by z/d to return from homogeneous coordinates
- This is called perspective division

$$p = \begin{pmatrix} x \\ y \\ z \\ z/d \end{pmatrix} \xrightarrow{\text{perspective division}} p' = \begin{pmatrix} \frac{x}{z/d} \\ \frac{y}{z/d} \\ d \\ 1 \end{pmatrix}$$

glFrustum()

- `glFrustum(left, right, bottom, top, near, far)`
 - `left, right, bottom, top` specify clipping planes for the `near` plane
- Maps the view frustum to canonical view volume, which will then be mapped to viewport
- Matrix

$$\begin{pmatrix} 2 \cdot near & 0 & right + left & 0 \\ right - left & 2 \cdot near & right - left & 0 \\ 0 & top - btm & top + btm & 0 \\ 0 & 0 & top - btm & 0 \\ 0 & 0 & -(far + near) & -2 \cdot far \cdot near \\ 0 & 0 & far - near & far - near \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

gluPerspective()

- `glFrustum` allows for off-center, non-symmetric view volume
- But we often want a symmetric view volume. We can use `gluPerspective(fovy, aspect, near, far)` instead
 - `fovy` specifies the fov angle in degrees in the *y*-direction
 - `aspect` is $\frac{width}{height}$
 - `near` and `far` specify distances to near/far planes

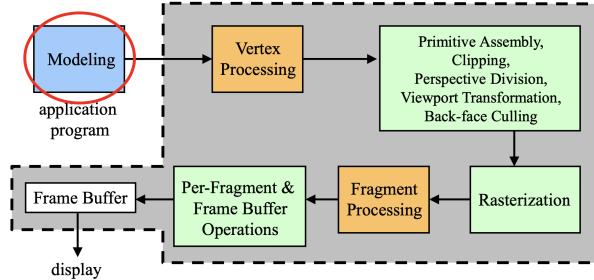
- Let $f = \cot\left(\frac{fovy}{2}\right)$. The matrix is

$$\begin{pmatrix} \frac{f}{aspect} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \frac{far + near}{near - far} & \frac{2 \cdot far \cdot near}{near - far} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

OPENGL RENDERING PIPELINE

- To render a primitive using OpenGL, the primitive goes through the following main stages

- Turning each primitive into pixels / fragments



Modelling

- Performed in application
- Provides a set of vertices that specifies geometric objects
- May perform scene processing to reduce amount of geometric data passed to rendering pipeline
 - View-frustum culling: do not pass objects that are outside the frustum
 - Occlusion culling: do not pass objects that are completely blocked

Vertex processing

- In this stage, model-view transformation is applied (now in camera-/eye space), affecting vertices and vertex normals
- Assigns color to vertices using lighting computation
- Compute texture coordinates at vertices
- Projection matrix is applied (now in clip space)

Primitive assembly

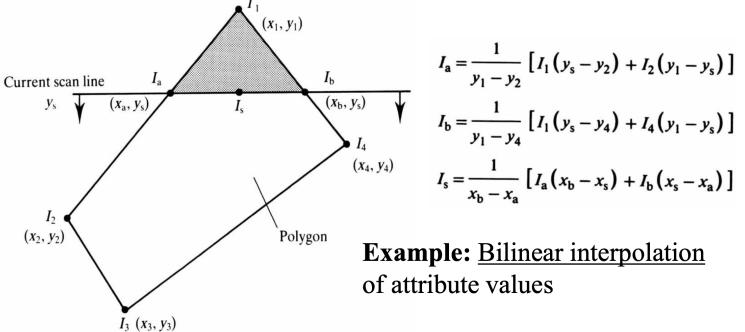
- Primitive assembly
 - Vertex data was processed separately in previous vertex processing stage
 - Vertex data is collected into complete primitives
 - Necessary for clipping, rasterization, back face culling
- Clipping
- Perspective division (to NDC space)
- Viewport transformation (to window space, include depth-range scaling)
 - For rasterization to color specific pixels in the window
- Back-face culling

Rasterization

- If geometric primitive not clipped out, appropriate pixels in frame buffer must be assigned colors
- Rasterizer produces a set of fragments for each primitive
 - Fragments are potential pixels, which have a pixel location, color, depth attributes
- Vertex attributes are interpolated over the primitive
- In bilinear interpolation, take weighted average of endpoints on a line segment

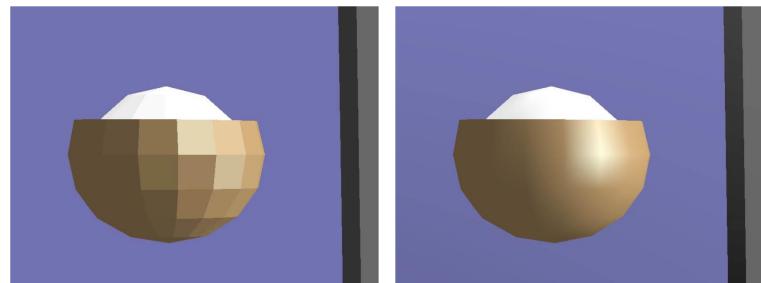
- Attribute values at fragments are computed by interpolating attribute values assigned to vertices

- Interpolation is performed in window space (2D)



Example: Bilinear interpolation of attribute values

- Color interpolation



Flat Shading
No interpolation of vertex colors

Smooth (Gouraud) Shading
Interpolation of vertex colors

- Can also interpolate z -value (depth), or texture coordinates

Fragment processing

- Each generated fragment is processed to determine the color of the corresponding pixel in the frame buffer
- Fragment color can be modified by texture mapping
 - Texture access (use the texture directly)
 - Texture application (combine texture with fragment color)

Per-fragment operations

Z-buffer hidden surface removal Discard fragment if it is blocked (occluded) by corresponding pixel in frame buffer

Blending Blend fragment with corresponding pixel in frame buffer

CLIPPING

- Easy for line segments and polygons
- Hard for curves and text - convert to lines and polygons first

Clipping 2D line segments

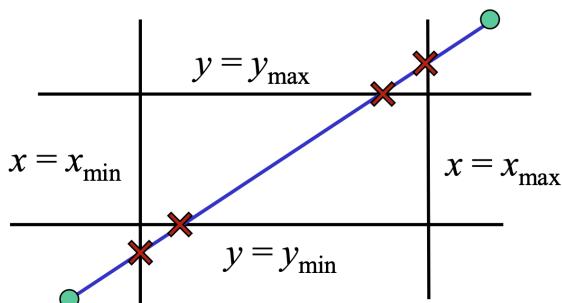
Brute force

- Compute intersections with all sides of clipping window
- Inefficient: one division per intersection
- Alternatives: Cohen-Sutherland, Liang-Barsky line clipping algorithms

Cohen-Sutherland

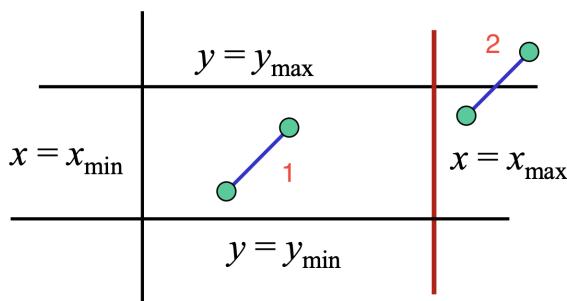
Split window into distinct regions

Extend sides of clipping window to get 9 distinct regions

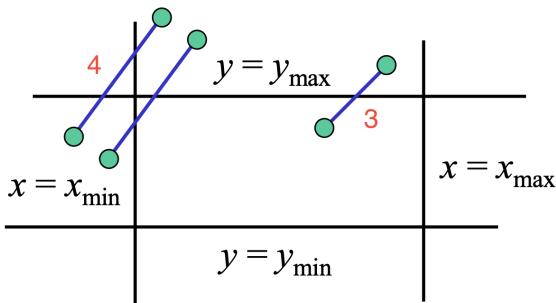


4 cases

- 1. Both endpoints inside all four lines - Accept
- 2. Both endpoints outside same line - Reject



- 3. One endpoint inside all four lines, one outside - Perform intersection
- 4. Both outside \Rightarrow may have some part inside. Do one intersection, reducing to Case 2, Case 3, or even Case 4 again.



Outcodes

- Define outcodes for easier computation
 - Outcodes divide space into 9 regions
 - Computation of outcode requires at most 4 subtractions

■ For each endpoint, define an outcode

$$b_0 b_1 b_2 b_3$$

- $b_0 = 1$ if $y > y_{\max}$, 0 otherwise
- $b_1 = 1$ if $y < y_{\min}$, 0 otherwise
- $b_2 = 1$ if $x > x_{\max}$, 0 otherwise
- $b_3 = 1$ if $x < x_{\min}$, 0 otherwise

1001	1000	1010	$y = y_{\max}$
0001	0000	0010	$y = y_{\min}$
0101	0100	0110	
$x = x_{\min}$	$x = x_{\max}$		

Consider line segment AB

- Case 1: outcodes are both 0
 - Accepted
- Case 2: outcode A AND outcode B (bitwise) is not 0
 - Both outcodes have 1 in same bit location (i.e. both outside the same line)
 - Rejected
- Case 3: outcode A is 0, outcode B not 0

- Compute intersection, by using the location of 1 in outcode B

- If outcode B has two 1s, then do two intersections

- Case 4: Neither 0, logical AND yields 0

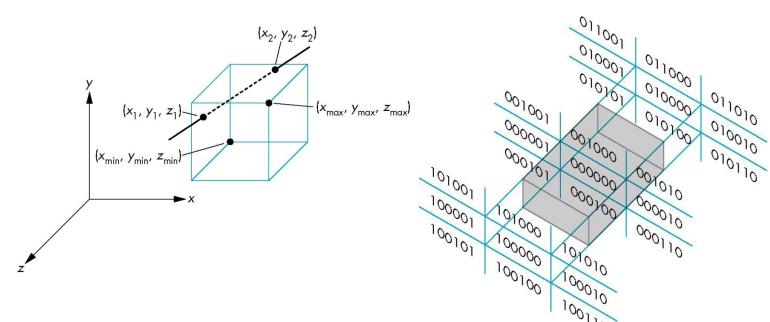
- Shorten line segment by intersecting with one side of window
- Compute outcode of new line segment
- Repeat algorithm. Yields case 2, 3 or 4

Efficiency

- Clipping window is usually small, relative to the entire database of line segments
- Most line segments are outside one or more sides of the window, and can be eliminated early
- Inefficient if case 4 occurs often

Application to 3D

- Use 6 bit outcodes
- Clip line segment against planes



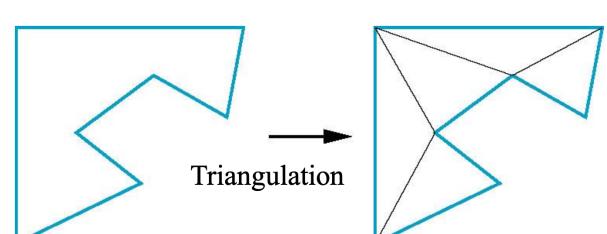
Misc

Clipping in clip space

- A vertex is in clip space after multiplication by projection matrix, and before perspective division
- Has coordinates (x, y, z, w)
- If vertex is in canonical view volume, then
 - $-1 \leq x/w \leq 1 \Rightarrow -w \leq x \leq w$
 - $-1 \leq y/w \leq 1 \Rightarrow -w \leq y \leq w$
 - $-1 \leq z/w \leq 1 \Rightarrow -w \leq z \leq w$
- The 6 inequalities are used to decide whether a vertex should be clipped away, in clip space

Polygon clipping

- If convex, clipping yields at most one other polygon
- If concave, use tessellation to split the polygon into a set of smaller convex polygons
 - Usually use triangles
 - Also makes rasterization easier

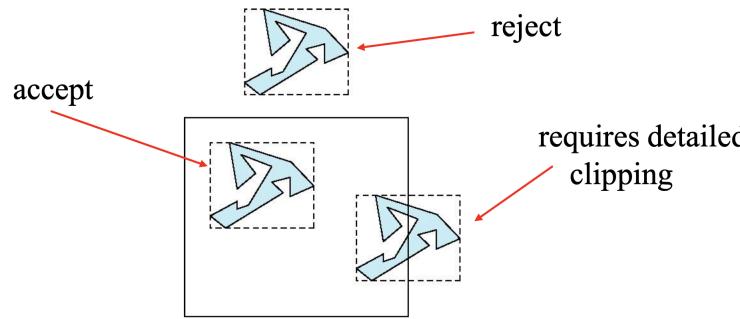


Pipeline clipping of line segments

- Clipping against each side of window is independent of other sides
- Can use four independent clippers in pipeline

Simple early acceptance & rejection

- Use an axis-aligned bounding box (AABB) instead of clipping a complex polygon
- AABB: smallest axis-aligned rectangle that encloses the polygon
- Simple to compute: max and min of x and y



RASTERIZATION

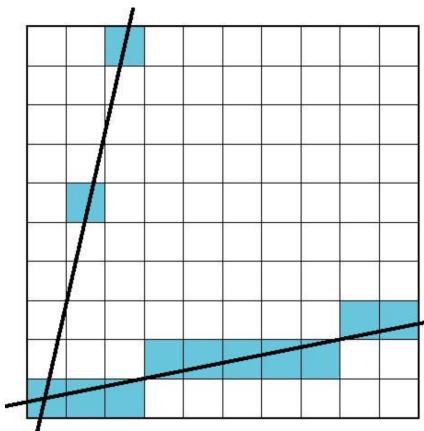
Digital Differential Analyzer - DDA

- For every x , plot pixel at closest y

```
for (x = x0, y = y0; x <= xe; x++) {
    write_pixel(x, round(y), line_color);
    y += m;
}
```

Steep lines

- Problem with steep lines, so use only when $0 \leq |m| \leq 1$.
- For $|m| > 1$, swap roles of x and y , i.e. for every y , plot pixel at closest x



Bresenham's Algorithm

- DDA requires one floating point addition per step
- Bresenham's algorithm can avoid floating point operations
- Consider only $0 \leq m \leq 1$, the rest can be derived by symmetry
 - Next pixel is either to the right, or right and top
 - Turns into binary decision problem

$$y = m(x_k + 1) + b$$

$$\begin{aligned} d_{\text{lower}} &= y - y_k \\ d_{\text{upper}} &= (y_k + 1) - y \end{aligned}$$

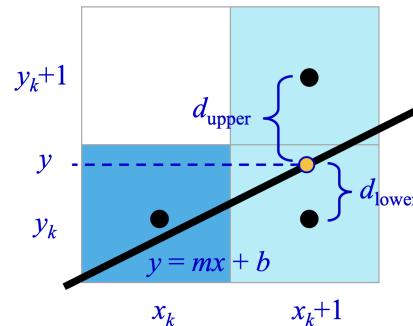
$$\begin{aligned} p_k &= \Delta x (d_{\text{lower}} - d_{\text{upper}}) \\ &= 2x_k \Delta y - 2y_k \Delta x + c \end{aligned}$$

where $c = 2\Delta y + \Delta x (2b - 1)$ is a integer constant.

p_k is an integer.

Recall that $\Delta x > 0$, therefore

- if $p_k < 0$, plot lower pixel
- if $p_k > 0$, plot upper pixel



- We can incrementally compute p_{k+1} from p_k

$$p_{k+1} = 2x_{k+1}\Delta y - 2y_{k+1}\Delta x + c$$

$$\begin{aligned} p_{k+1} - p_k &= 2(x_{k+1} - x_k)\Delta y - 2(y_{k+1} - y_k)\Delta x \\ \Rightarrow p_{k+1} &= p_k + 2\Delta y - 2(y_{k+1} - y_k)\Delta x \end{aligned}$$

Therefore,

- if $p_k < 0$, $p_{k+1} = p_k + 2\Delta y$
- if $p_k > 0$, $p_{k+1} = p_k + 2\Delta y - 2\Delta x$

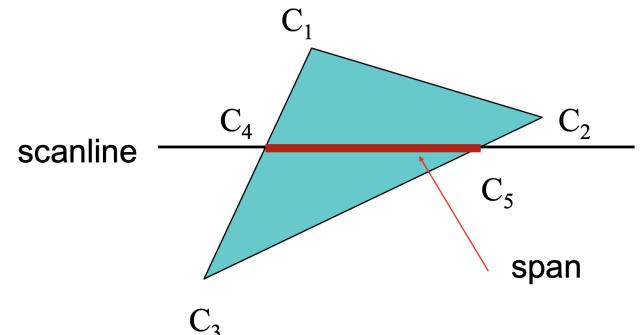
where $p_0 = 2\Delta y - \Delta x$

- For each x_k , we need only an integer addition and a test

- Made into a single instruction on many graphics chips

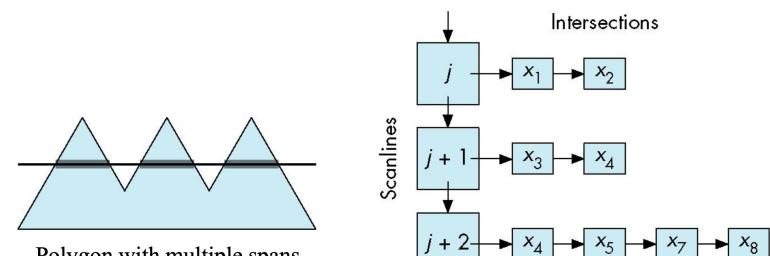
Scan-line fill

- Applies to convex polygons (usually triangles)
 - Non-convex polygons assumed tessellated
- Shades (colors) have been computed for vertices
- Combine with z -buffer algorithm
 - Vertices also have z -values (depth-values)
- March across horizontal scan lines, interpolating shades and z -values
- $C_1 C_2 C_3$ specified by `glColor` or by vertex shading (lighting computation)
- C_4 determined by interpolating between C_1 and C_3
- C_5 determined by interpolating between C_2 and C_3
- Interpolate between C_4 and C_5 along span



- Can also fill by maintaining a data structure of all intersections of polygons with scanlines

- Sort by scan line
- Fill each span



Flood-fill

- Fill can be done recursively if we know a seed point located inside
- But first scan-convert edges using an edge color

HIDDEN-SURFACE REMOVAL

- In 3D graphics, some object should appear in front of another
- Without hidden-surface removal, objects "on top" are the ones that are drawn last
- Hidden-surface removal fixes this problem using the z -buffer algorithm

Painter's algorithm

- Assumes we have a valid back-to-front ordering
- Render polygons in back-to-front order so that polygons behind others are simply painted over

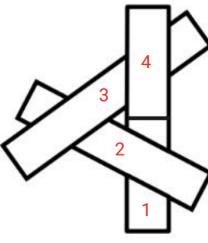
Depth sorting

- Can be done in $O(n \log n)$ time
- Ran before executing painter's algorithm
- Cannot handle cyclic overlap of polygons
 - Solution: split original polygons

Not always possible to be able to sort a set of polygons in a back-to-front order

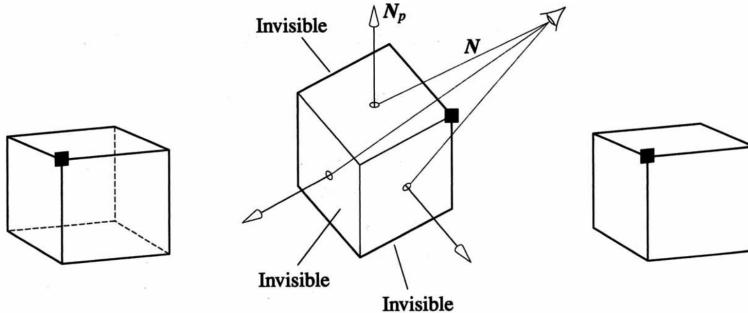
E.g. cyclic occlusion

Solution: Split original polygons



Back-face culling

- Eliminate polygon if it is back-facing and invisible (e.g. back face of a 3d building)
- It is back facing if $N_p \cdot N < 0$



- In OpenGL, simply enable culling
 - Polygon vertices must be provided in counter-clockwise order
 - May not work correctly for non-convex polygon

Image space approach

- For each projector (nm projectors for a n by m frame buffer), find the closest polygon among k polygons
- $O(nmk)$
- e.g. Ray tracing, z-buffer

z-buffer algorithm

- Use a z-buffer or depth buffer to store depth of closest object at each pixel so far
 - Even for double buffering, there is only ONE depth buffer, for the currently drawn frame
- As we render each polygon, compare depth of each fragment, to depth in z-buffer
- If less than, update, placing fragment in color buffer and its depth in z-buffer
- If not, then discard

```
// In main()
glutInitDisplayMode(... | GLUT_DEPTH);
// In init()
glEnable(GL_DEPTH_TEST);
// Cleared in display callback
glClear(... | GL_DEPTH_BUFFER_BIT);
```

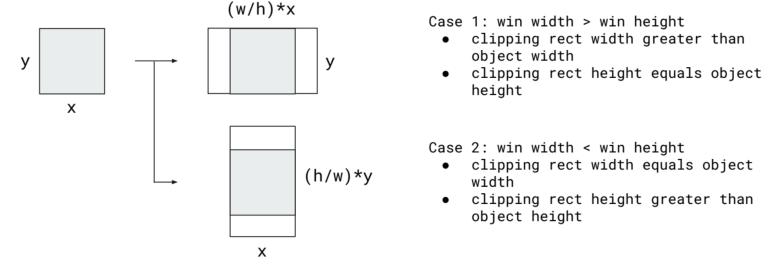
TUTORIALS

T2 Q7

Question

In many old CRT monitors, the pixels are not square. Let's assume the pixel width-to-height aspect ratio is 4:3. Suppose in the camera coordinate frame, there is a disc in the $z = 0$ plane, centered at $(100, 200, 0)$, and has a radius of 10. You want to draw the entire disc as big as possible inside the window, and it should appear circular and not oval. (A) If the window size is 600×300 (width \times height), how would you set up the viewport and the orthographic projection using OpenGL? (B) What if the window size is now 300×600 ? (C) What if the window size is now 300×320 ?

Answer



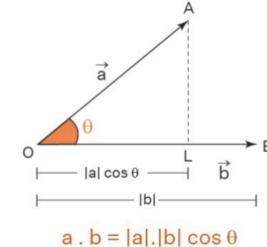
```
double pixelAspectRatio = 4.0 / 3.0;
int winWidth = 600; // or 300 for (B) and (C)
int winHeight = 300; // or 600 for (B) and 320 for (C)
double apparentWinHeight = winHeight / pixelAspectRatio;
double center[2] = {100.0, 200.0};
double radius = 10.0;

glViewport(0, 0, winWidth, winHeight);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();

if (winWidth >= apparentWinHeight)
    gluOrtho2D(
        center[0] - radius * (winWidth / apparentWinHeight),
        center[0] + radius * (winWidth / apparentWinHeight),
        center[1] - radius, center[1] + radius);
else
    gluOrtho2D(
        center[0] - radius, center[0] + radius,
        center[1] - radius * (apparentWinHeight / winWidth),
        center[1] + radius * (apparentWinHeight / winWidth));
```

T3 Q1c

Projection of a onto b



$$\text{Projection of Vector } a \text{ on Vector } b = \frac{a \cdot b}{|b|}$$

Perpendicular distance of point Q to a plane

Let plane be defined as $ax + by + cz + d = 0$. The perpendicular distance from Q to the plane is given by

$$\frac{|ax_q + by_q + cz_q + d|}{\sqrt{a^2 + b^2 + c^2}}$$

Proof:

What is the perpendicular distance of the point Q from the plane $ax + by + cz + d = 0$?

$$QP = (x_q - x_p, y_q - y_p, z_q - z_p)$$

$$n = (a, b, c)$$

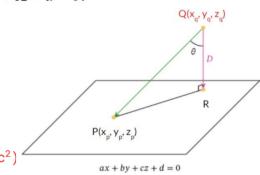
$$|QR| = |(x_q - x_p, y_q - y_p, z_q - z_p) \cdot (a, b, c)| / \sqrt{a^2 + b^2 + c^2}$$

$$= |a(x_q - x_p) + b(y_q - y_p) + c(z_q - z_p)| / \sqrt{a^2 + b^2 + c^2}$$

$$= |(ax_q + by_q + cz_q) - (ax_p + by_p + cz_p)| / \sqrt{a^2 + b^2 + c^2}$$

For any point in the plane (x, y, z) , $ax + by + cz + d = 0$
Thus, $ax_p + by_p + cz_p + d = 0 \Rightarrow d = -(ax_p + by_p + cz_p)$

$$\text{Therefore, the distance is: } |QR| = \frac{|ax_q + by_q + cz_q + d|}{\sqrt{a^2 + b^2 + c^2}}$$



T3 Q4

Given the following 4×4 transformation matrix \mathbf{M} , where r_{ij} are the elements of a rotation matrix, s_1, s_2, s_3 are non-zero scale factors in the x, y, z directions, and t_1, t_2, t_3 are the translations in the x, y, z directions. Find the *inverse matrix* \mathbf{M}^{-1} . Write your answer as a single matrix.

$$\mathbf{M} = \begin{bmatrix} s_1 r_{11} & s_1 r_{12} & s_1 r_{13} & t_1 \\ s_2 r_{21} & s_2 r_{22} & s_2 r_{23} & t_2 \\ s_3 r_{31} & s_3 r_{32} & s_3 r_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{M} = \mathbf{T} \cdot \mathbf{S} \cdot \mathbf{R}$$

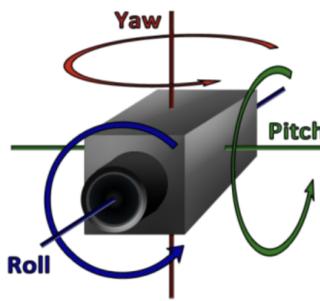
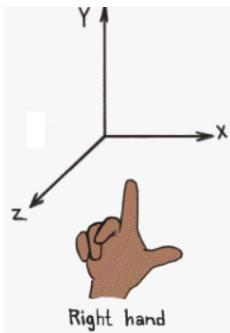
$$\mathbf{R} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{S} = \begin{bmatrix} s_1 & 0 & 0 & 0 \\ 0 & s_2 & 0 & 0 \\ 0 & 0 & s_3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & t_1 \\ 0 & 1 & 0 & t_2 \\ 0 & 0 & 1 & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Clearly T is applied last, since you can reverse the translation and get a clean matrix
- Then, S is applied after R because s_1 is distributed along the row

T4 Q3 - right handed coordinate system

Coordinate system

- By convention we use a right-handed coordinate system
- The right thumb points along the z axis in the positive direction and the curl of the fingers represents a motion from the first or x axis to the second or y axis
- Alternatively, thumb points at x , pointer points at y , middle finger points at z (left image below)



Pitch: rotation about x -axis | Yaw: rotation about y -axis | Roll: rotation about z -axis

Rotation

When rotating around a vector, let thumb be direction of vector. Use grip rule, direction that fingers wrap in is the direction of rotation

Cross product

Refer to Geometry section.

T4 Q5 - camera to NDC

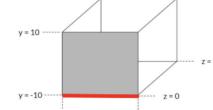
- To convert from camera coordinates to NDC, notice that the ratio of the vertex along each axis stays the same after the transformation
- For the z -axis, the near plane maps to -1 , while the far plane maps to 1 .

x-axis example

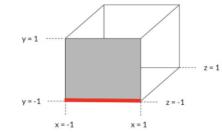
Question 5

A vertex, whose **camera coordinates** are $(4, 6, -6)$

Consider the **x** axis

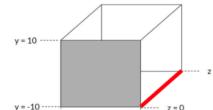


$$\frac{4 - (-10)}{10 - (-10)} = \frac{x' - (-1)}{1 - (-1)}$$



z-axis example

Consider the **z** axis



$$\frac{-6 - 0}{-8 - 0} = \frac{z' - (-1)}{1 - (-1)}$$

