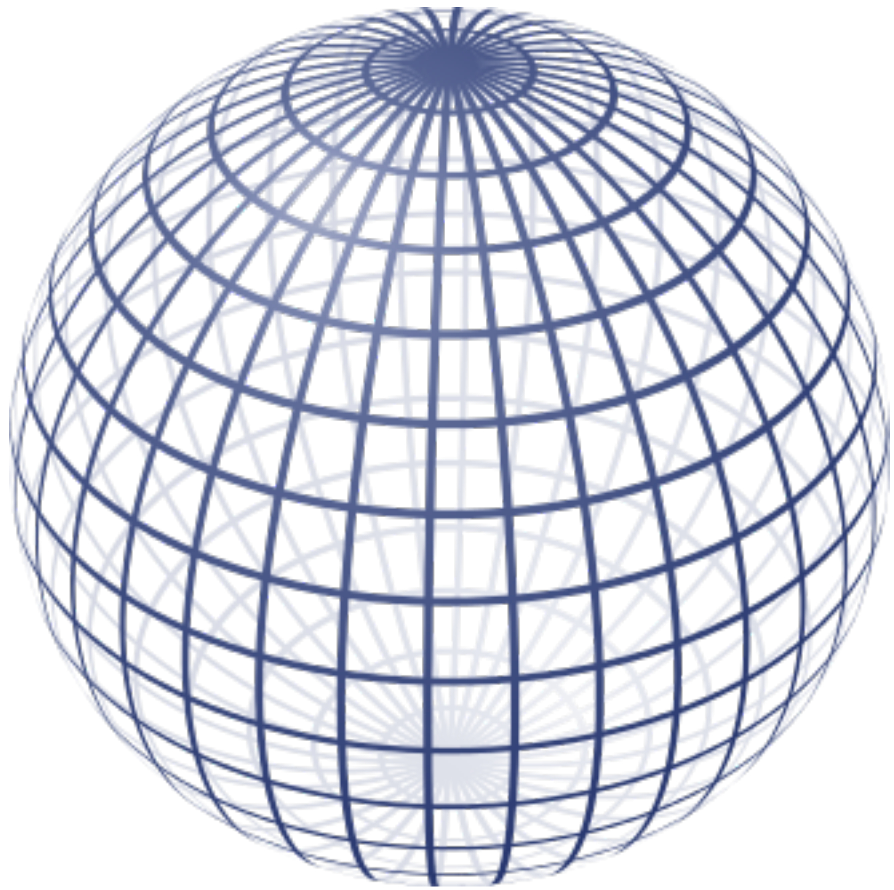
CSE 5542 - Real Time Rendering
Week 10

Spheres



GLUT

void glutSolidSphere(GLdouble radius, GLint slices, GLint stacks);

void glutWireSphere(GLdouble radius, GLint slices, GLint stacks);

Direct Method

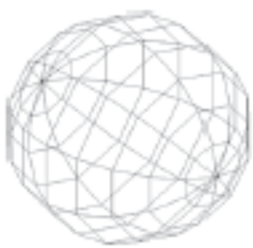


FIGURE 2.15 Sphere approximation with quadrilaterals.

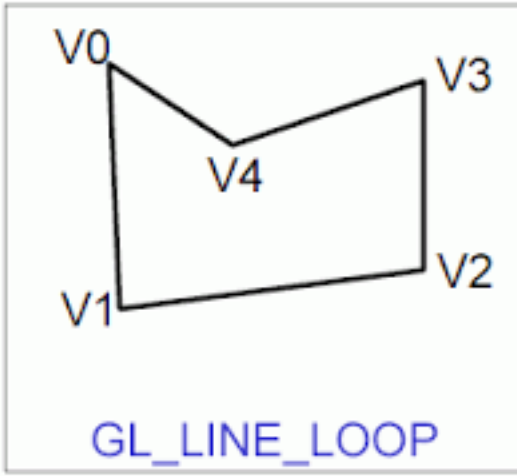
```
const float DegreesToRadians = M_PI / 180.0; // M_PI = 3.14159...

point3 quad_data[342];   // 8 rows of 18 quads

int k = 0;
for(float phi = -80.0; phi <= 80.0; phi += 20.0)
{
    float phir   = phi*DegreesToRadians;
    float phir20 = (phi + 20.0)*DegreesToRadians;

    for(float theta = -180.0; theta <= 180.0; theta += 20.0)
    {
        float thetar = theta*DegreesToRadians;
        quad_data[k] = point3(sin(thetar)*cos(phir),
                              cos(thetar)*cos(phir), sin(phir));
        k++;
        quad_data[k] = point3(sin(thetar)*cos(phir20),
                              cos(thetar)*cos(phir20), sin(phir20));
        k++;
    }
}
```

$$x(\theta, \phi) = \sin\theta \cos\phi,$$

$$y(\theta, \phi) = \cos\theta \cos\phi,$$

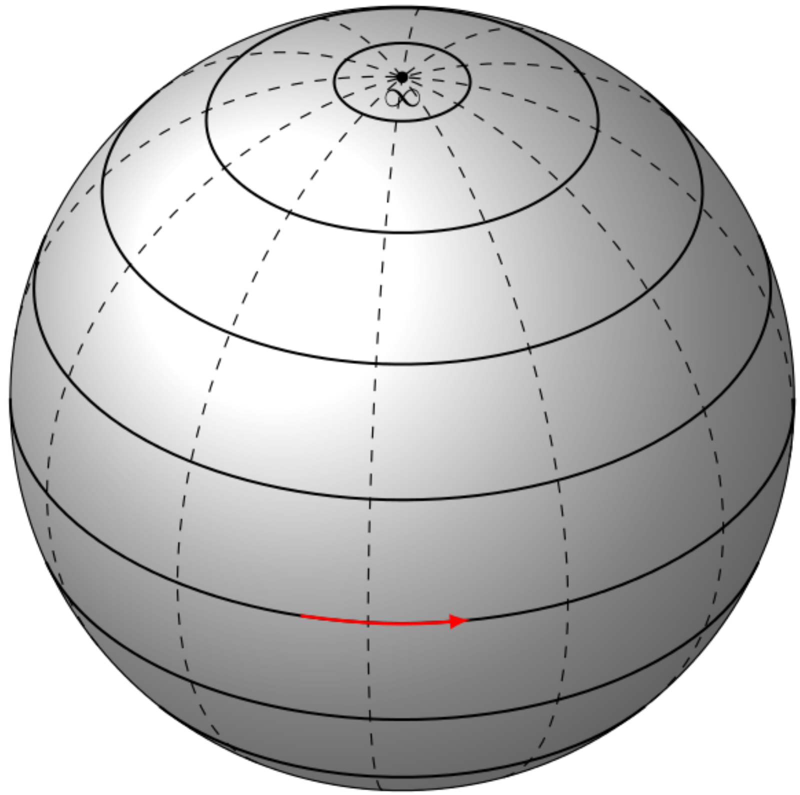
$$z(\theta, \phi) = \sin\phi.$$

GL_LINE_LOOP

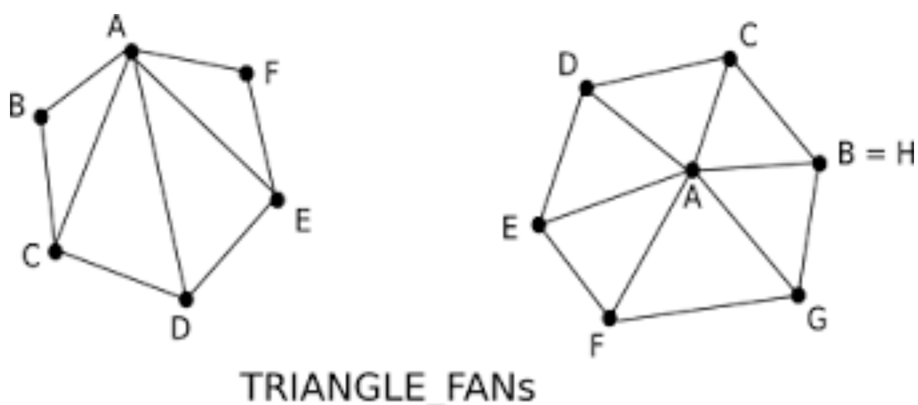


glDrawArrays(GL_LINE_LOOP,...)

Problems - @ poles



Use GL_TRIANGLE FAN



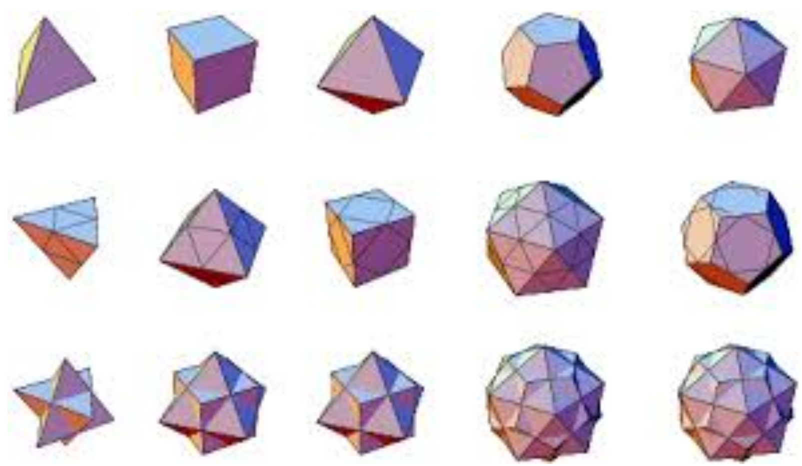

```
const float DegreesToRadians = M_PI / 180.0; // M_PI = 3.14159...

int k = 0;
point3 strip_data[40];

strip_data[k] = point3(0.0, 0.0, 1.0);
k++;

float sin80 = sin(80.0*DegreesToRadians);
float cos80 = cos(80.0*DegreesToRadians);


for(float theta = -180.0; theta <= 180.0; theta += 20.0)
{
    float thetar = theta*DegreesToRadians;
    strip_data[k] = point3(sin(thetar)*cos80,
                           cos(thetar)*cos80, sin80);
    k++;
}

strip_data[k] = point3(0.0, 0.0, -1.0);
k++;

for(float theta = -180.0; theta <= 180.0; theta += 20.0)
{
    float thetar = theta;
    strip_data[k] = point3(sin(thetar)*cos80,
                           cos(thetar)*cos80, sin80);
    k++;
}


glDrawArrays(GL_TRIANGLE_FAN, ....)
```

Method II

YAS (Yet Another Solution)

http://www.andrewnoske.com/wiki/Generating_a_sphere_as_a_3D_mesh

Platonic Solids



Procedure

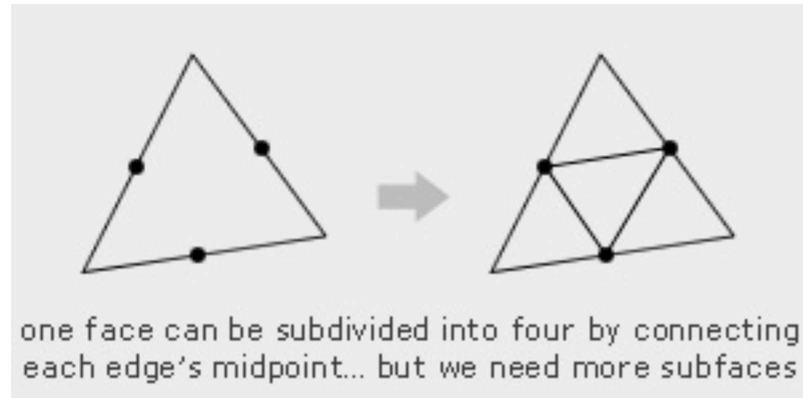
Create Platonic Solid –

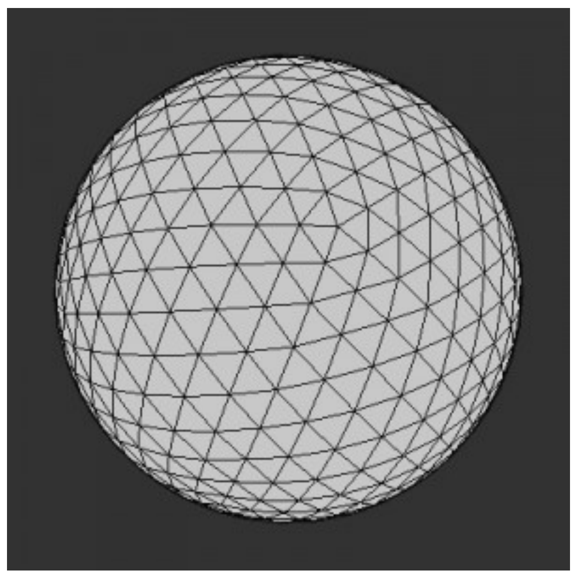
http://www.csee.umbc.edu/~squire/reference/polyhedra.shtml#icosahedron

Subdivide each face –

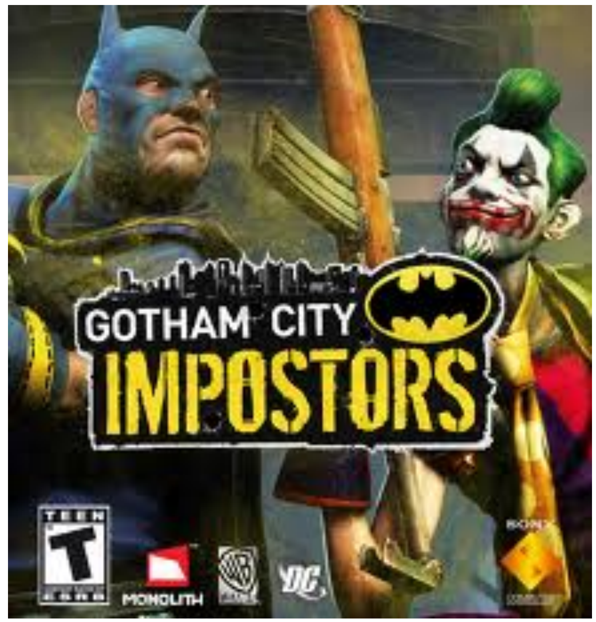
http://donhavey.com/blog/tutorials/tutorial-3-the-icosahedron-sphere/

Think Sierpinski-like





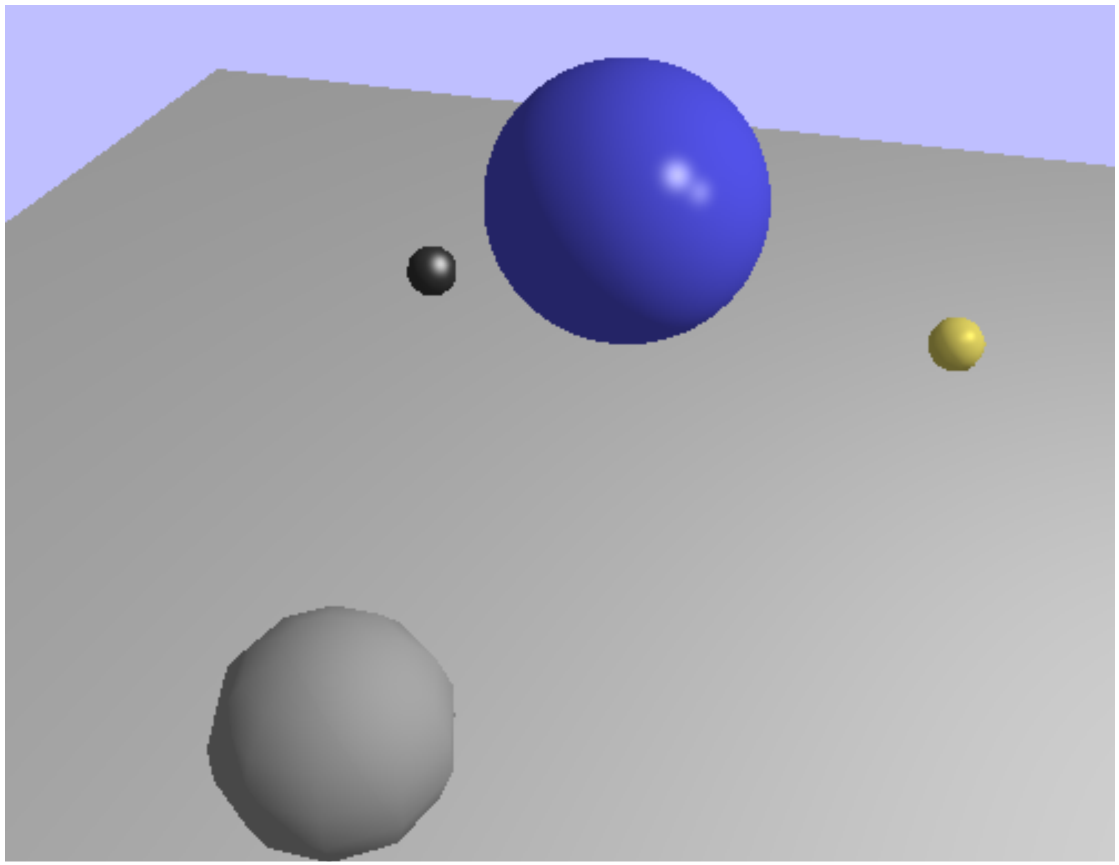
Method III


GOTHAM CITY
IMPOSTORS
TEEN
T
MONOLITH

Impostor Spheres

http://www.arcsynthesis.org/gltut/Illumination/Tutorial%2013.html

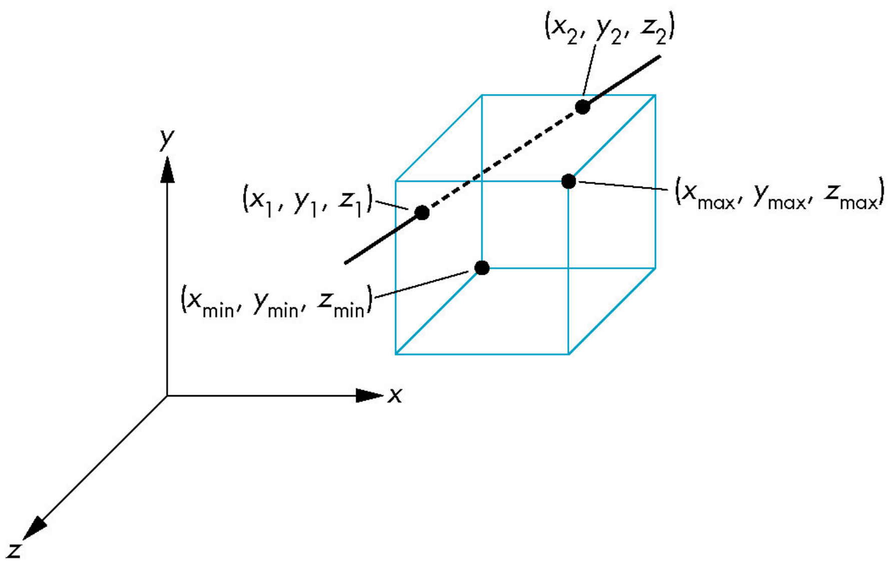
Impostors

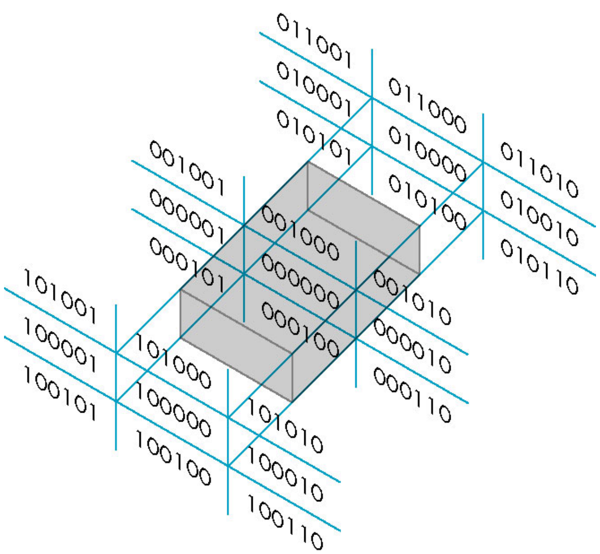


Clipping and Scan Conversion

Cohen Sutherland in 3D

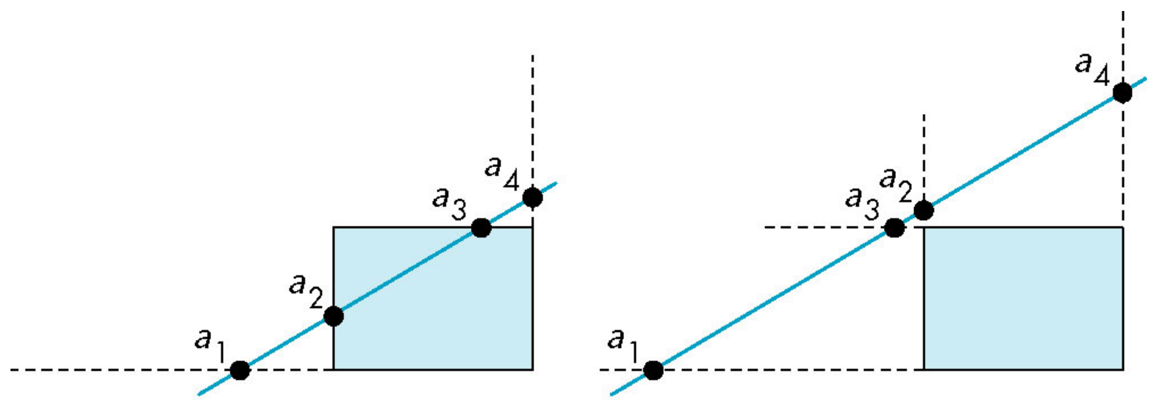
- Use 6-bit outcodes
- When needed, clip line segment against planes





Liang-Barsky Clipping

- In (a): $a_4 > a_3 > a_2 > a_1$
  - Intersect right, top, left, bottom: shorten
- In (b): $a_4 > a_2 > a_3 > a_1$
  - Intersect right, left, top, bottom: reject



(a) (b)

Polygon Clipping

- Not as simple as line segment clipping
  - Clipping a line segment yields at most one line segment
  - Clipping a polygon can yield multiple polygons

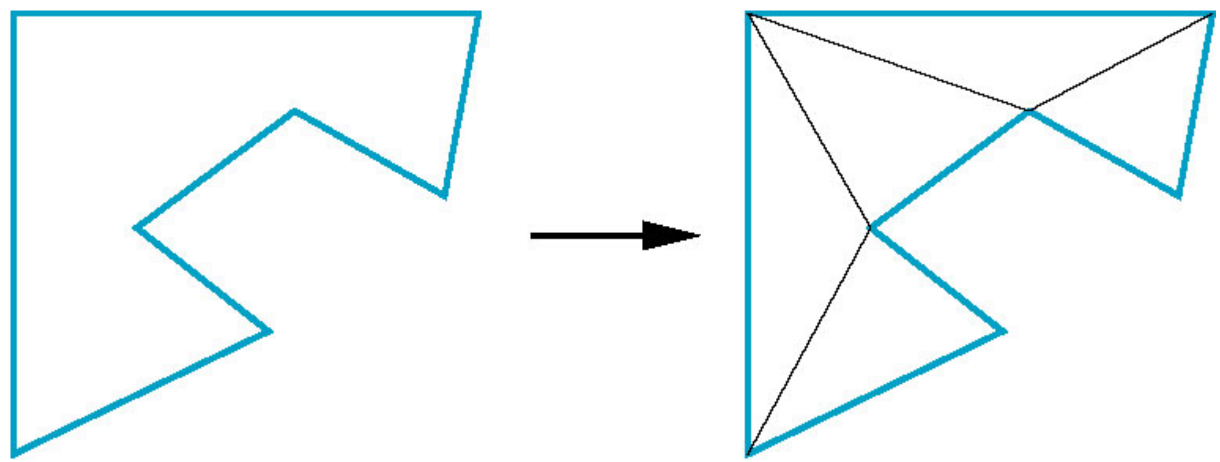
- Convex polygon is cool ☺

THE OHIO STATE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Fixes

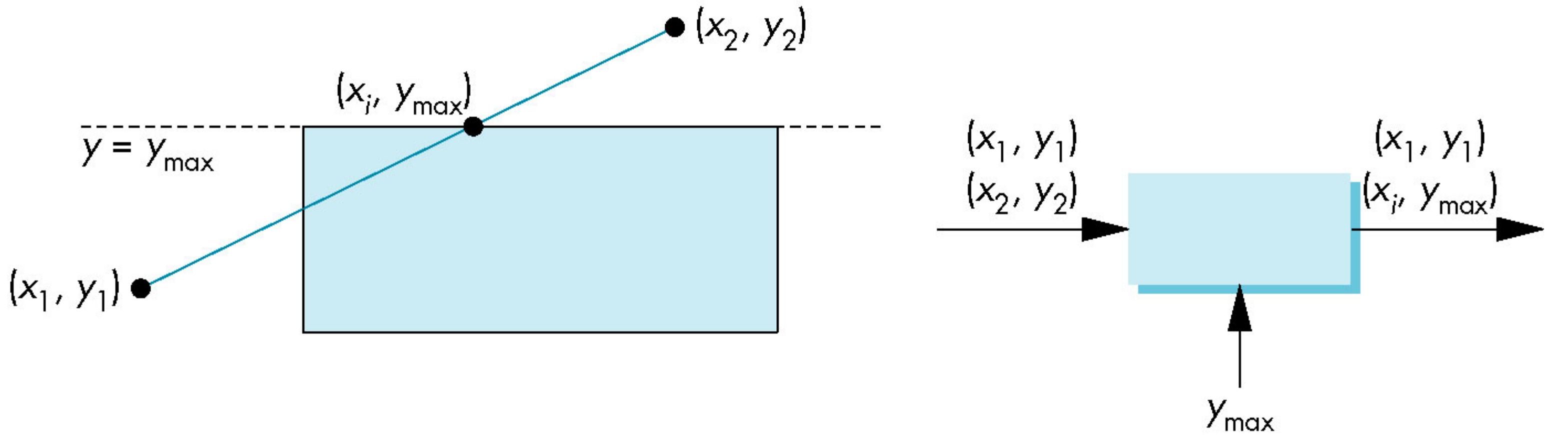
Tessellation and Convexity

Replace nonconvex (*concave*) polygons with triangular polygons (a *tessellation*)



Clipping as a Black Box

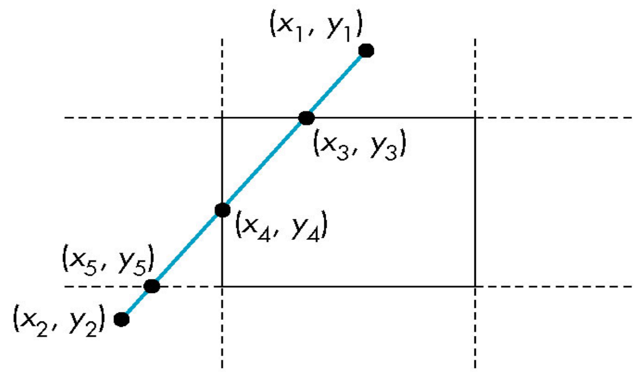
Line segment clipping - takes in two vertices and produces either no vertices or vertices of a clipped segment

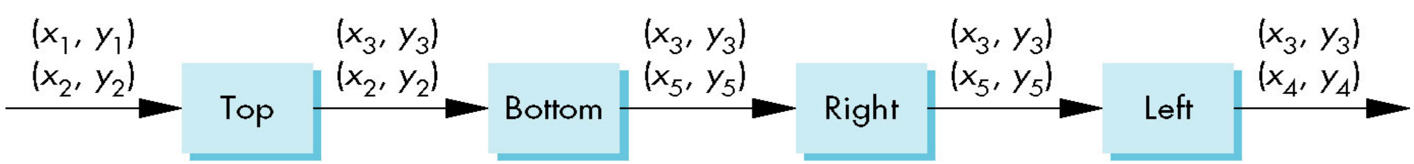


Pipeline Clipping - Line Segments

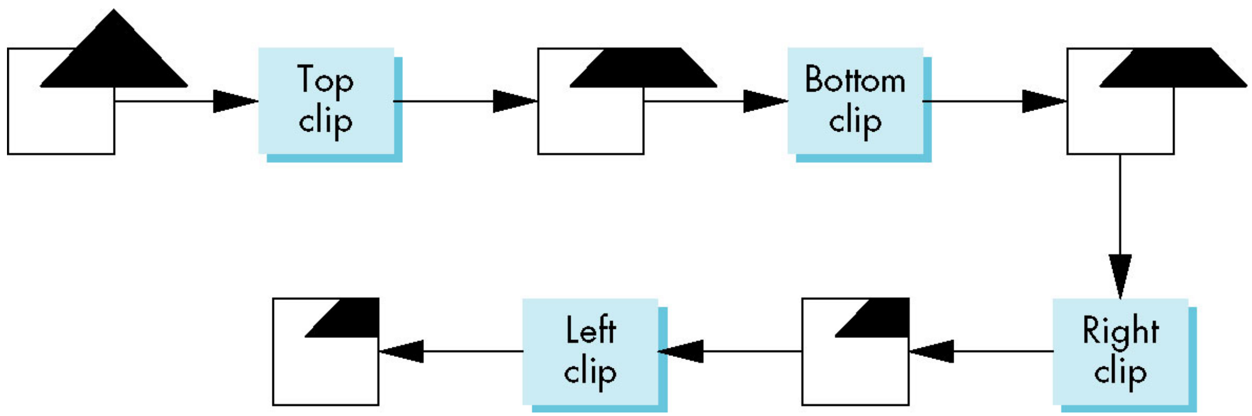
Clipping side of window is independent of other sides

– Can use four independent clippers in a pipeline





Pipeline Clipping of Polygons

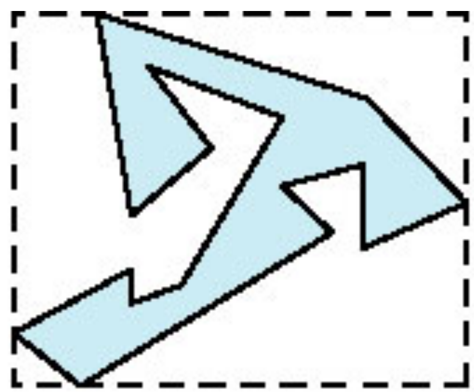


- Three dimensions: add front and back clippers
- Small increase in latency

Bounding Boxes

Ue an *axis-aligned bounding box* or *extent*

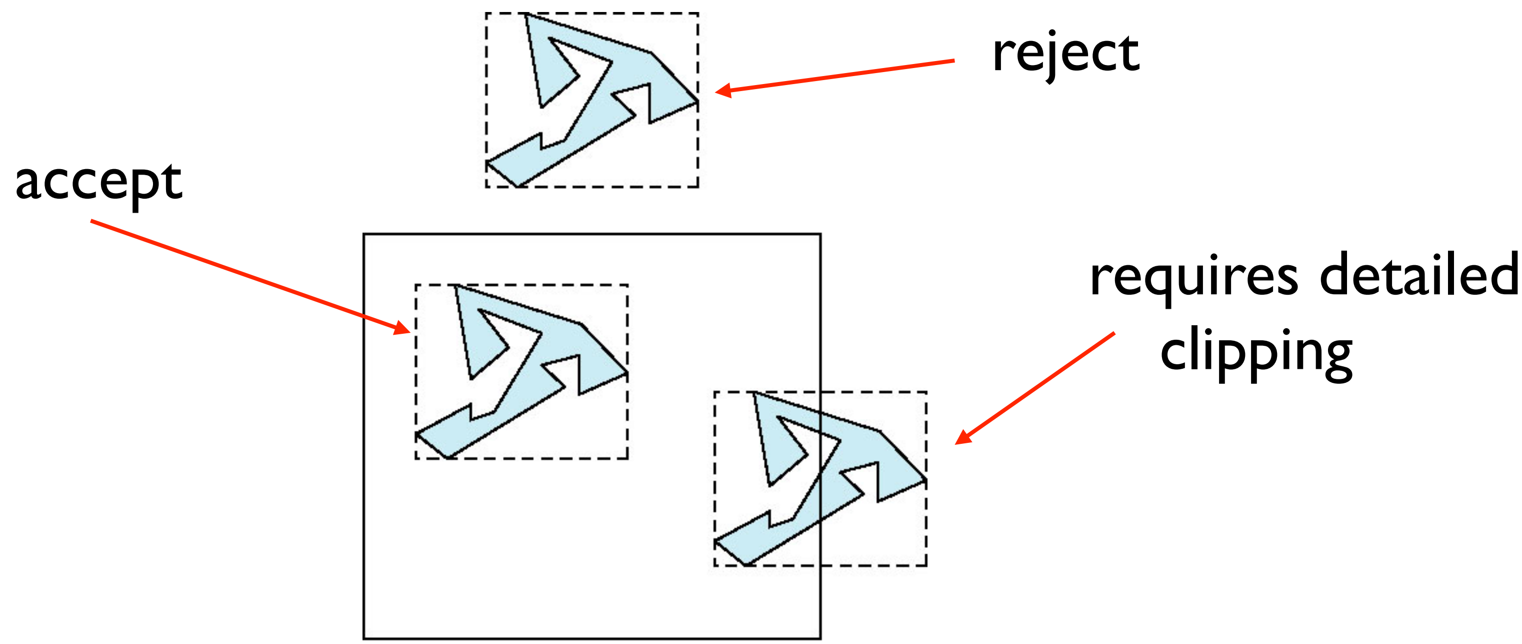
- Smallest rectangle aligned with axes that encloses the polygon
- Simple to compute: max and min of x and y



THE OHIO STATE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Bounding boxes

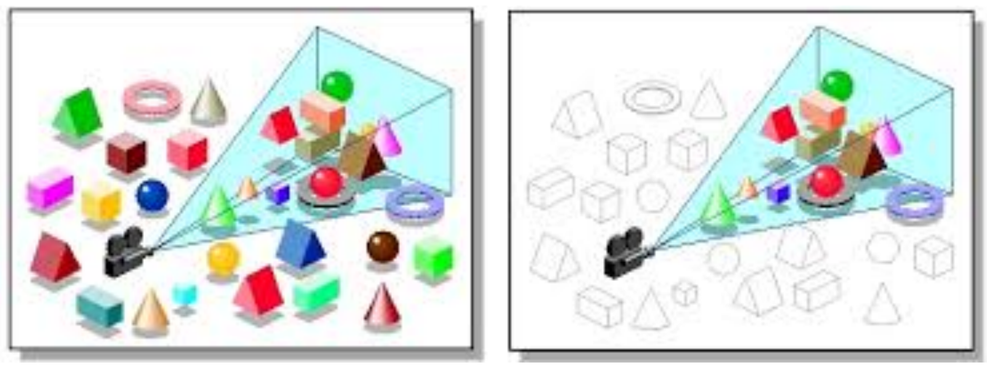
Can usually determine accept/reject based only on bounding box

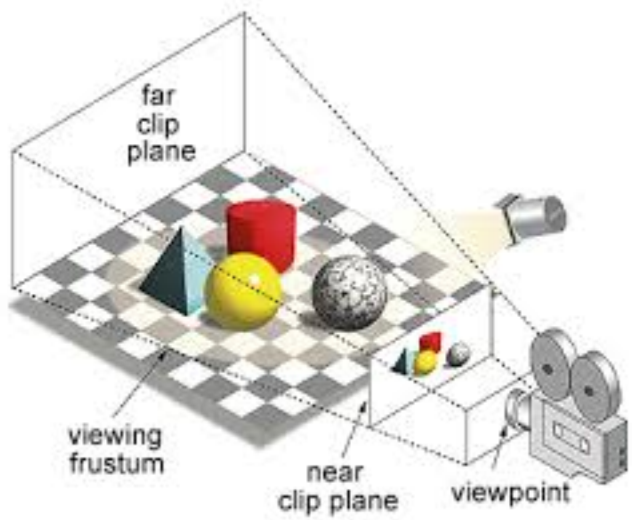


Clipping vs. Visibility

- Clipping similar to hidden-surface removal
- Remove objects that are not visible to the camera
- Use visibility or occlusion testing early in the process to eliminate as many polygons as possible before going through the entire pipeline

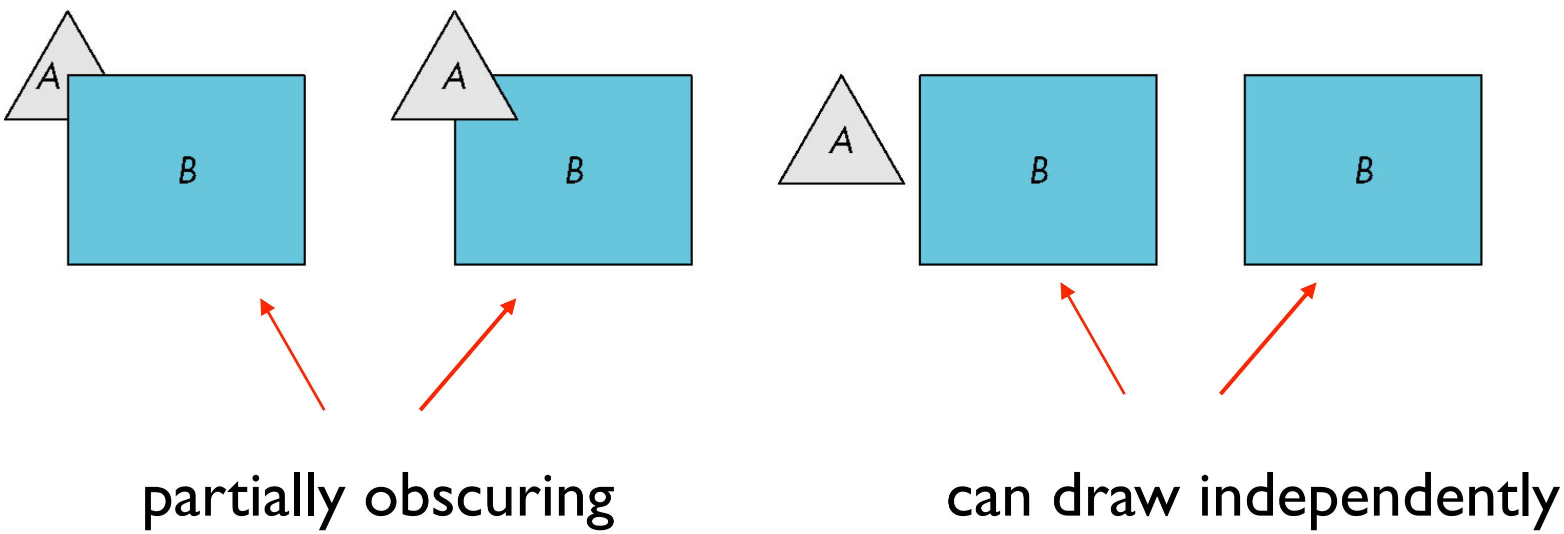
Clipping





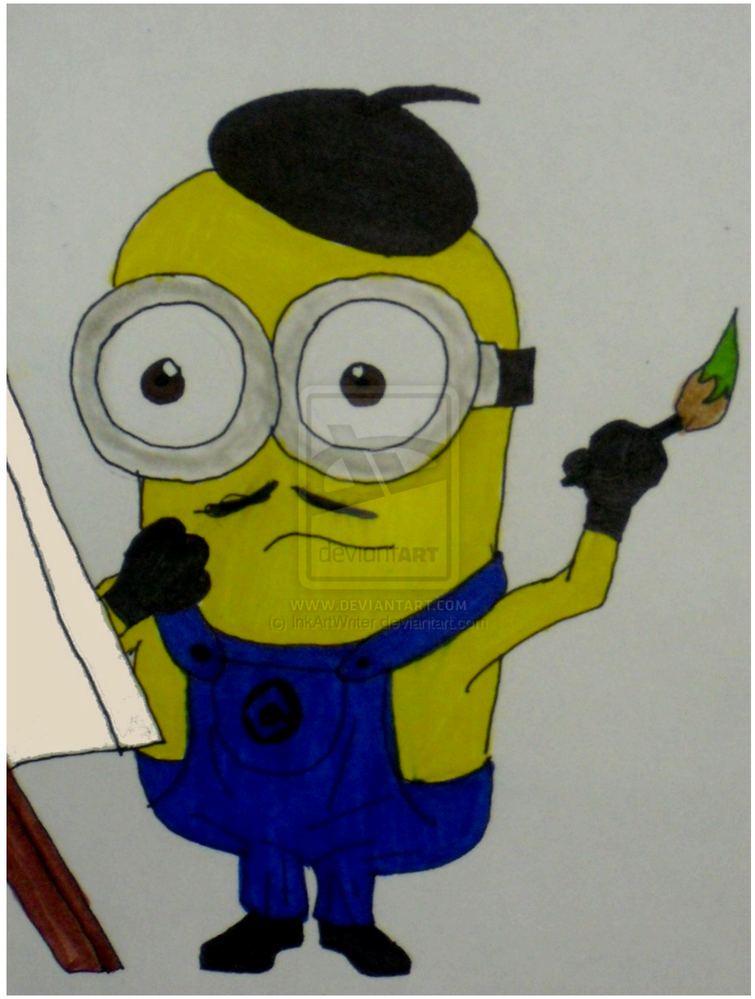
Hidden Surface Removal

Object-space approach: use pairwise testing between polygons (objects)



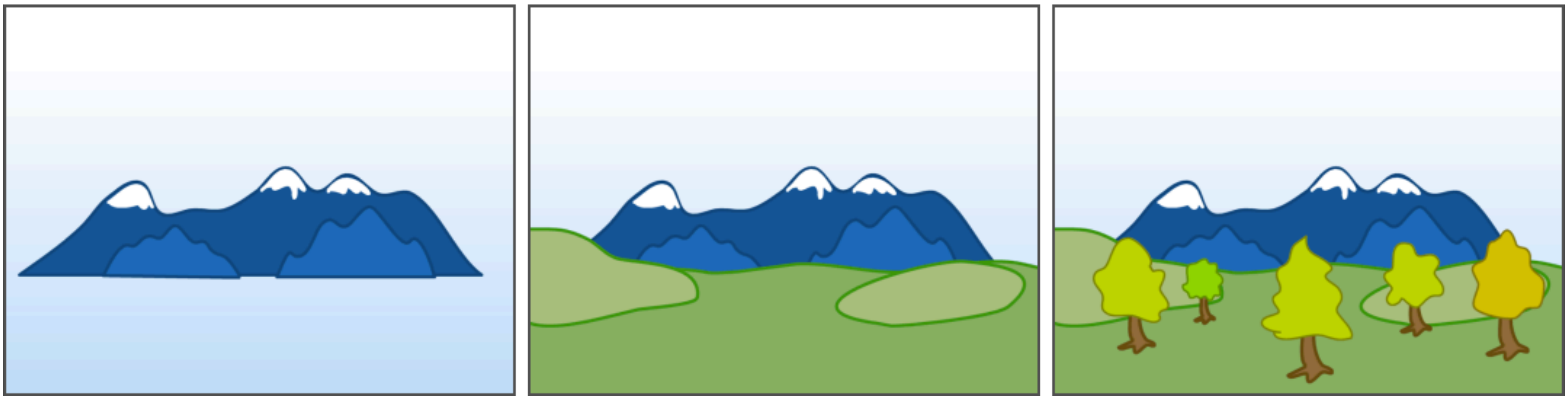
Worst case complexity $O(n^2)$ for n polygons

Better Still



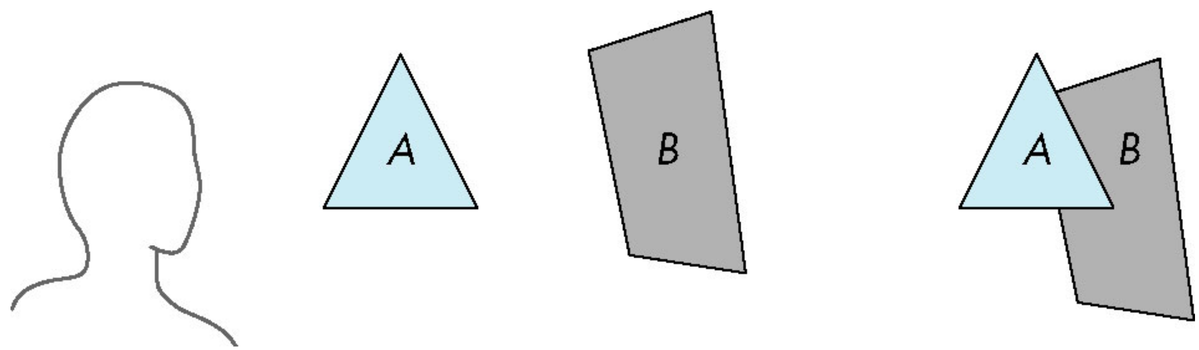
DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Better Still



Painter's Algorithm

Render polygons a back to front order so that polygons behind others are simply painted over



B behind A as seen by viewer

Fill B then A

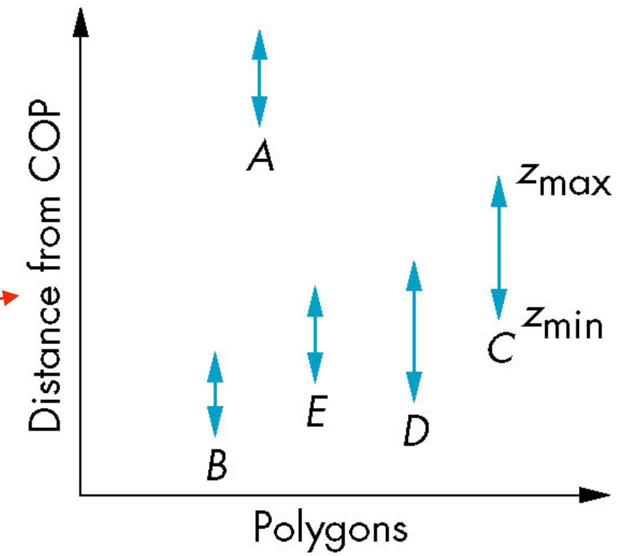
Depth Sort

Requires ordering of polygons first

- O(n log n) calculation for ordering
- Not all polygons front or behind all other polygons

Order polygons and deal with easy cases first, harder later

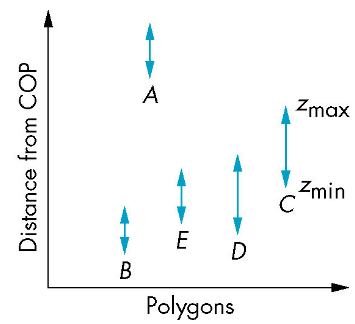
Polygons sorted by distance from COP



Easy Cases

A lies behind all other polygons

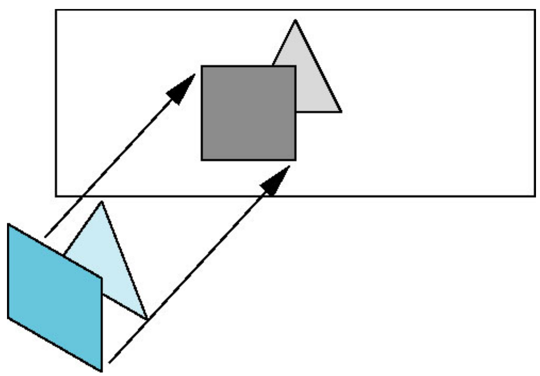
– Can render



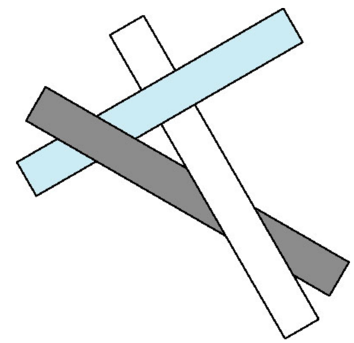
Polygons overlap in z but not in either x or y

– Can render independently

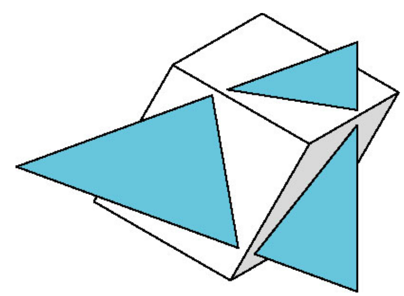
Hard Cases



Overlap in all directions but can one is fully on one side of the other



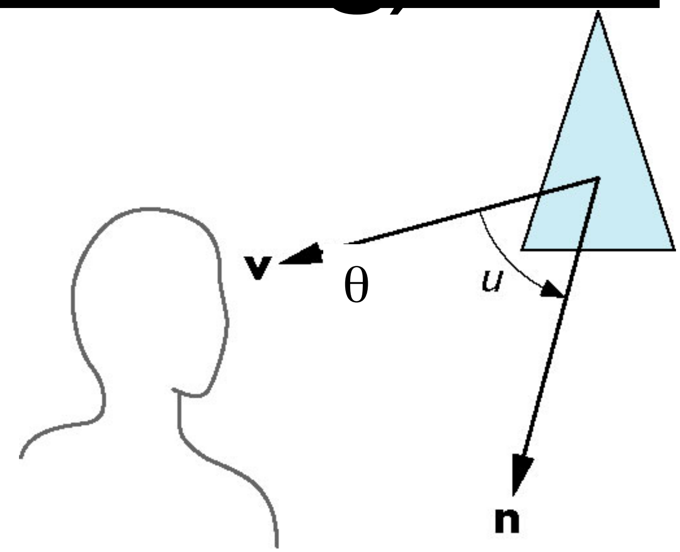
cyclic overlap



penetration

Back-Face Removal (Culling)

face is visible iff $90 \geq \theta \geq -90$
equivalently $\cos \theta \geq 0$
or $\mathbf{v} \cdot \mathbf{n} \geq 0$



- plane of face has form $ax + by + cz + d = 0$
- After normalization $\mathbf{n} = (0\ 0\ 1\ 0)^T$

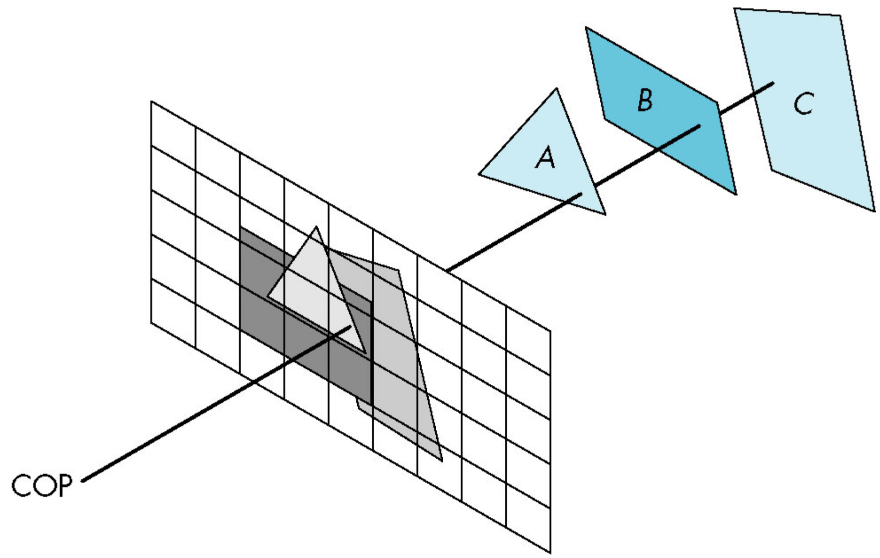
+ Need only test the sign of c

- Will not work correctly if we have nonconvex objects

THE OHIO STATE UNIVERSITY DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Image Space Approach

- Look at each ray (nm for an n x m frame buffer)
- Find closest of k polygons
- Complexity O(nmk)
- Ray tracing
- z-buffer

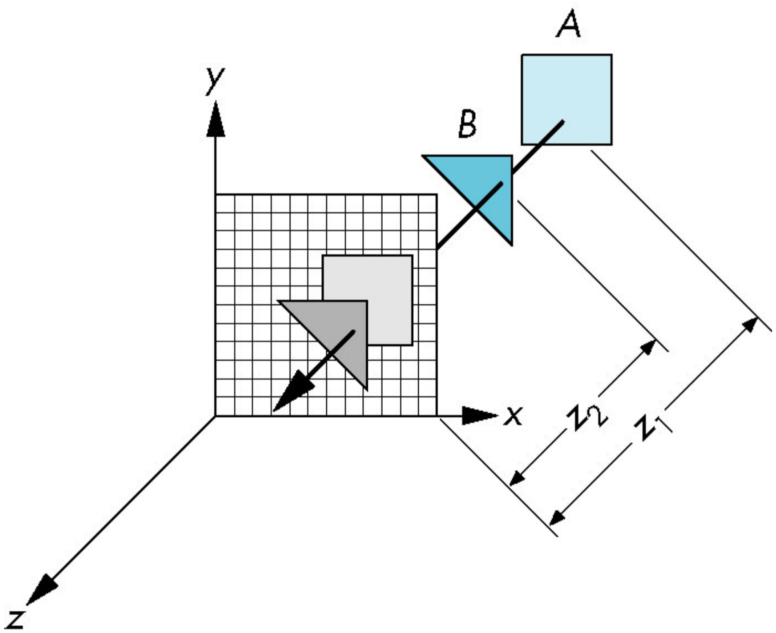


DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

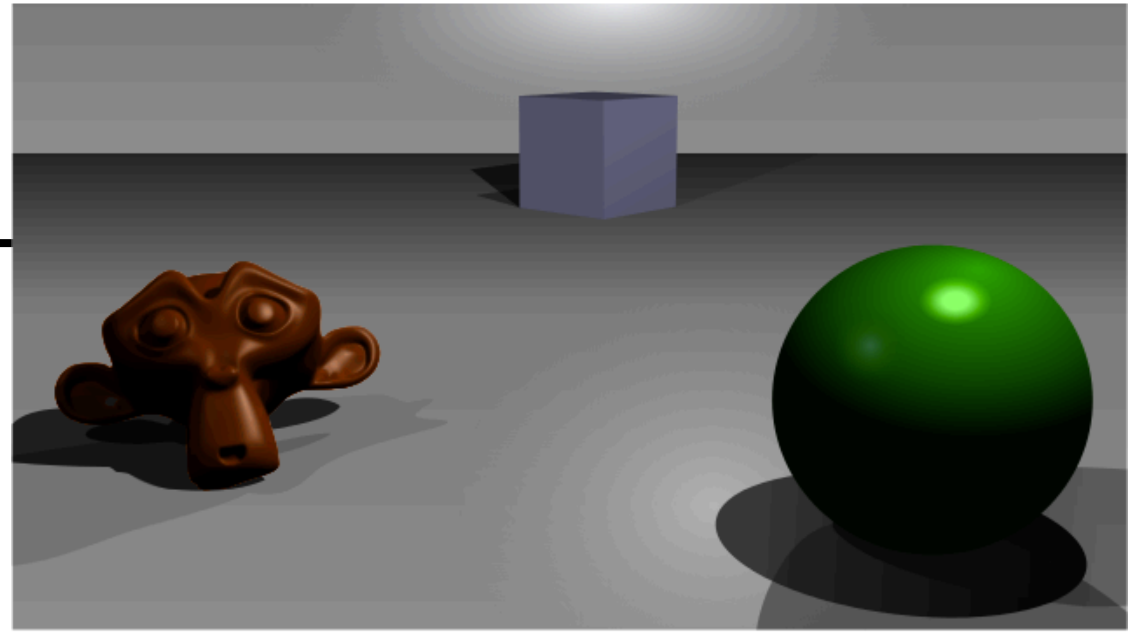
z-Buffer Algorithm

- Use a buffer called z or depth buffer to store depth of closest object at each pixel found so far
- As we render each polygon, compare the depth of each pixel to depth in z buffer
- If less, place shade of pixel in color buffer and update z buffer

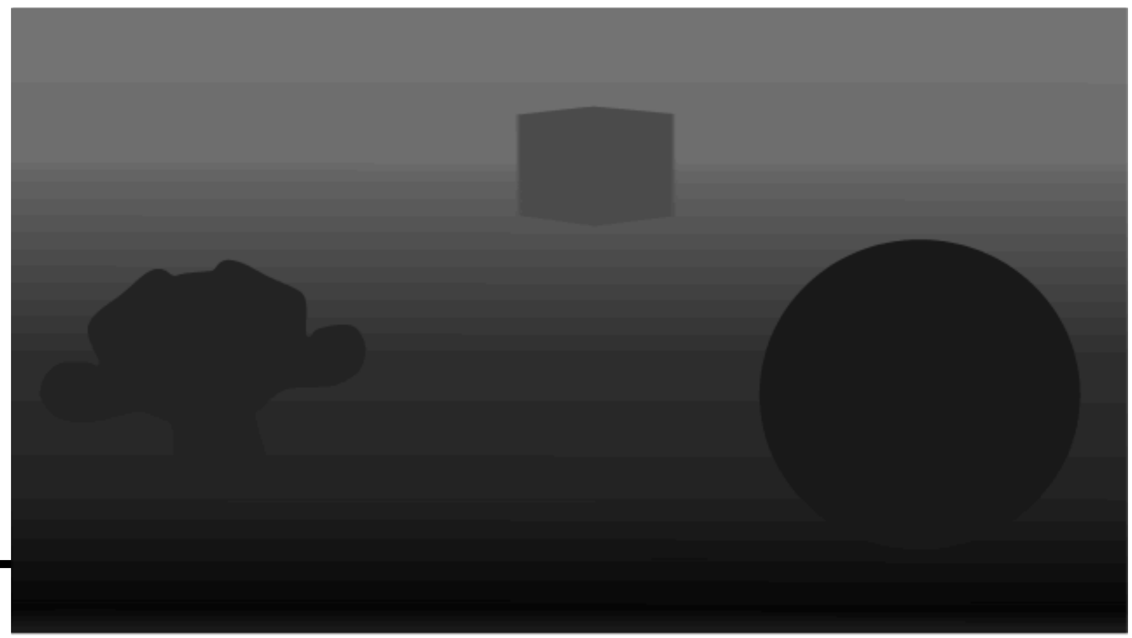
z-Buffer




```
for(each polygon P in the polygon list)
      do{
          for(each pixel(x,y) that intersects P)
          do{
               Calculate z-depth of P at (x,y)
               If (z-depth < z-buffer[x,y])
               then{
                      z-buffer[x,y]=z-depth;
                      COLOR(x,y)=Intensity of P at(x,y);
                   }
               #If-programming-for alpha compositing:
               Else if (COLOR(x,y).opacity < 100%)
               then{
                      COLOR(x,y)=Superimpose
COLOR(x,y) in front of Intensity of P at(x,y);
                   }
               #Endif-programming-for
             }
         }
  display COLOR array.
```



A simple three-dimensional scene



Z-buffer representation

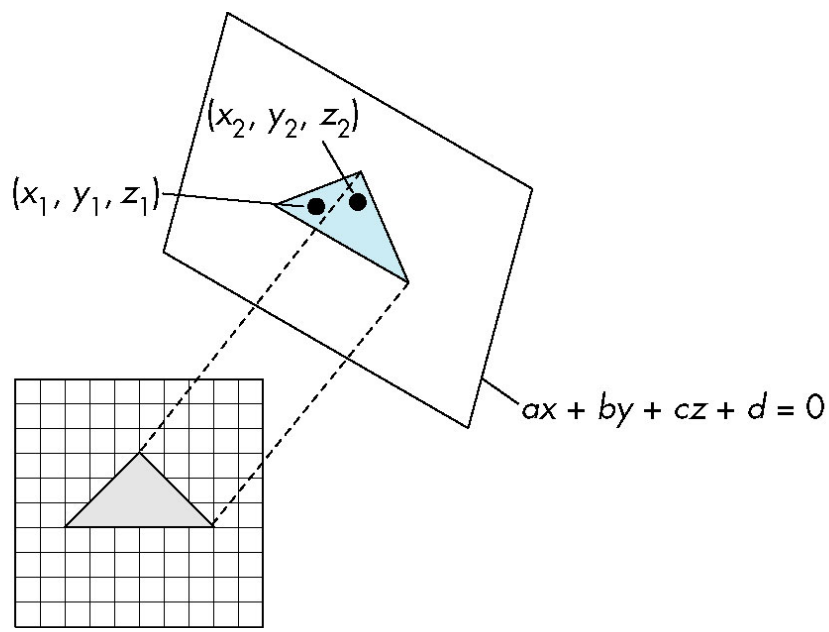
Efficiency - Scanline

As we move across a scan line, the depth changes satisfy $a\Delta x+b\Delta y+c\Delta z=0$

Along scan line

$\Delta y = 0$

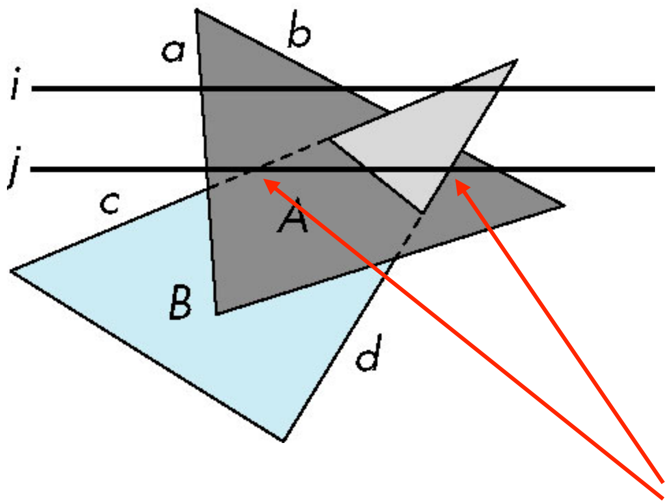
$$\Delta z = -\frac{a}{c}\Delta x$$



In screen space $\Delta x = 1$

Scan-Line Algorithm

Combine shading and hsr through scan line algorithm



scan line i: no need for depth information, can only be in no or one polygon

scan line j: need depth information only when in more than one polygon

Implementation

Need a data structure to store

- Flag for each polygon (inside/outside)
- Incremental structure for scan lines that stores which edges are encountered
- Parameters for planes

THE OHIO STATE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Rasterization

- Rasterization (scan conversion)
  - Determine which pixels that are inside primitive specified by a set of vertices
  - Produces a set of fragments
  - Fragments have a location (pixel location) and other attributes such color and texture coordinates that are determined by interpolating values at vertices
- Pixel colors determined later using color, texture, and other vertex properties

Diversion

Rendering Spheres

Spheres - Application

A.7 PER-FRAGMENT LIGHTING OF SPHERE MODEL

A.7.1 Application Code

```
// fragment shading of sphere model

#include "Angel.h"

const int NumTimesToSubdivide = 5;
const int NumTriangles        = 4096;
// (4 faces)^(NumTimesToSubdivide + 1)
const int NumVertices         = 3 * NumTriangles;

typedef Angel::vec4 point4;
typedef Angel::vec4 color4;

point4 points[NumVertices];
vec3   normals[NumVertices];

// Model-view and projection matrices uniform location
```

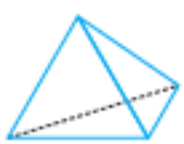


FIGURE 5.32 Tetrahedron.

FIGURE 5.33 Subdivision of a triangle by (a) bisecting angles, (b) computing the centroid, and (c) bisecting sides.

```
GLuint  ModelView, Projection;

//----------------------------------------------------------------------

int Index = 0;

void
triangle( const point4& a, const point4& b, const point4& c )
{
    vec3  normal = normalize( cross(b - a, c - b) );

    normals[Index] = normal;  points[Index] = a;  Index++;
    normals[Index] = normal;  points[Index] = b;  Index++;
    normals[Index] = normal;  points[Index] = c;  Index++;
}

//----------------------------------------------------------------------

point4
unit( const point4& p )
{
    float len = p.x*p.x + p.y*p.y + p.z*p.z;

    point4 t;
    if ( len > DivideByZeroTolerance ) {
        t = p / sqrt(len);
        t.w = 1.0;
    }

    return t;
}
```

Sphere-Definition

```
void
divide_triangle( const point4& a, const point4& b,
      const point4& c, int count )
{
    if ( count > 0 ) {
        point4 v1 = unit( a + b );
        point4 v2 = unit( a + c );
        point4 v3 = unit( b + c );
        divide_triangle(  a, v1, v2, count - 1 );
        divide_triangle(  c, v2, v3, count - 1 );
        divide_triangle(  b, v3, v1, count - 1 );
        divide_triangle( v1, v3, v2, count - 1 );
    }
    else {
        triangle( a, b, c );
    }
}
```

```
void
tetrahedron( int count )
{
    point4 v[4] = {
        vec4( 0.0, 0.0, 1.0, 1.0 ),
        vec4( 0.0, 0.942809, -0.333333, 1.0 ),
        vec4( -0.816497, -0.471405, -0.333333, 1.0 ),
        vec4( 0.816497, -0.471405, -0.333333, 1.0 )
    };

    divide_triangle( v[0], v[1], v[2], count );
    divide_triangle( v[3], v[2], v[1], count );
    divide_triangle( v[0], v[3], v[1], count );
    divide_triangle( v[0], v[2], v[3], count );
}
```

FIGURE 5.34 Sphere approximations using subdivision.

Sphere-Lighting


```
// OpenGL initialization
void
init( void )
{
    // Subdivide a tetrahedron into a sphere
    tetrahedron( NumTimesToSubdivide );

    // Create a vertex array object
    GLuint vao;
    glGenVertexArrays( 1, &vao );
    glBindVertexArray( vao );

    // Create and initialize a buffer object
    GLuint buffer;
    glGenBuffers( 1, &buffer );
    glBindBuffer( GL_ARRAY_BUFFER, buffer );
    glBufferData( GL_ARRAY_BUFFER, sizeof(points) + sizeof(normals),
        NULL, GL_STATIC_DRAW );
    glBufferSubData( GL_ARRAY_BUFFER, 0, sizeof(points), points );
    glBufferSubData( GL_ARRAY_BUFFER, sizeof(points),
           sizeof(normals), normals );

    // Load shaders and use the resulting shader program
    GLuint program = InitShader( "vshader56.glsl", "fshader56.glsl" );
    glUseProgram( program );

    // set up vertex arrays
    GLuint vPosition = glGetAttribLocation( program, "vPosition" );
    glEnableVertexAttribArray( vPosition );
    glVertexAttribPointer( vPosition, 4, GL_FLOAT, GL_FALSE, 0,
            BUFFER_OFFSET(0) );
```

VBOs & VAOs

```
// set up vertex arrays
GLuint vPosition = glGetAttribLocation( program, "vPosition" );
glEnableVertexAttribArray( vPosition );
glVertexAttribPointer( vPosition, 4, GL_FLOAT, GL_FALSE, 0,
        BUFFER_OFFSET(0) );
```

```
GLuint vNormal = glGetAttribLocation( program, "vNormal" );
glEnableVertexAttribArray( vNormal );
glVertexAttribPointer( vNormal, 3, GL_FLOAT, GL_FALSE, 0,
        BUFFER_OFFSET(sizeof(points)) );
```

Material Properties

```
GLuint vNormal = glGetAttribLocation( program, "vNormal" );
glEnableVertexAttribArray( vNormal );
glVertexAttribPointer( vNormal, 3, GL_FLOAT, GL_FALSE, 0,
        BUFFER_OFFSET(sizeof(points)) );

// Initialize shader lighting parameters
point4 light_position( 0.0, 0.0, 2.0, 0.0 );
color4 light_ambient( 0.2, 0.2, 0.2, 1.0 );
color4 light_diffuse( 1.0, 1.0, 1.0, 1.0 );
color4 light_specular( 1.0, 1.0, 1.0, 1.0 );

color4 material_ambient( 1.0, 0.0, 1.0, 1.0 );
color4 material_diffuse( 1.0, 0.8, 0.0, 1.0 );
color4 material_specular( 1.0, 0.0, 1.0, 1.0 );
float  material_shininess = 5.0;

color4 ambient_product = light_ambient * material_ambient;
color4 diffuse_product = light_diffuse * material_diffuse;
color4 specular_product = light_specular * material_specular;

glUniform4fv( glGetUniformLocation(program, "AmbientProduct"),
    1, ambient_product );
glUniform4fv( glGetUniformLocation(program, "DiffuseProduct"),
    1, diffuse_product );
glUniform4fv( glGetUniformLocation(program, "SpecularProduct"),
    1, specular_product );

glUniform4fv( glGetUniformLocation(program, "LightPosition"),
    1, light_position );

glUniform1f( glGetUniformLocation(program, "Shininess"),
   material_shininess );

// Retrieve transformation uniform variable locations
ModelView = glGetUniformLocation( program, "ModelView" );
Projection = glGetUniformLocation( program, "Projection" );

glEnable( GL_DEPTH_TEST );

glClearColor( 1.0, 1.0, 1.0, 1.0 ); // white background
}
```

The Usual

```
void
display( void )
{
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

    point4 at( 0.0, 0.0, 0.0, 1.0 );


    point4 eye( 0.0, 0.0, 2.0, 1.0 );
    vec4   up( 0.0, 1.0, 0.0, 0.0 );

    mat4 model_view = LookAt( eye, at, up );
    glUniformMatrix4fv( ModelView, 16, GL_TRUE, model_view );

    glDrawArrays( GL_TRIANGLES, 0, NumVertices );
    glutSwapBuffers( void );
}
```

```
void
keyboard( unsigned char key, int x, int y )
{
    switch( key ) {
        case 033: // Escape Key
        case 'q': case 'Q':
            exit( EXIT_SUCCESS );
            break;
    }
}

//----------------------------------------------------------------------

void
reshape( int width, int height )
{
    glViewport( 0, 0, width, height );

    GLfloat left = -2.0, right = 2.0;
    GLfloat top = 2.0, bottom = -2.0;
    GLfloat zNear = -20.0, zFar = 20.0;

    GLfloat aspect = GLfloat(width)/height;

    if ( aspect > 1.0 ) {
        left *= aspect;
        right *= aspect;
    }
    else {
        top /= aspect;
        bottom /= aspect;
    }

    mat4 projection = Ortho( left, right, bottom, top, zNear, zFar );
    glUniformMatrix4fv( Projection, 1, GL_TRUE, projection );
}

//----------------------------------------------------------------------
```

Finally

```
int
main( int argc, char **argv )
{

    glutInit( &argc, argv );
    glutInitDisplayMode( GLUT_RGBA | GLUT_DEPTH );
    glutInitWindowSize( 512, 512 );
    glutCreateWindow( "Sphere" );

    glewInit( void );

    init( void );

    glutDisplayFunc( display );
    glutReshapeFunc( reshape );
    glutKeyboardFunc( keyboard );

    glutMainLoop( void );
    return 0;
}
```

But …

Vertex Shader – Object Space

A.7.2 Vertex Shader

```
#version 150

in   vec4 vPosition;
in   vec3 vNormal;

// output values that will be interpolated per-fragment
out  vec3 fN;
out  vec3 fE;
out  vec3 fL;

uniform mat4 ModelView;
uniform vec4 LightPosition;
uniform mat4 Projection;

void main()
{
    fN = vNormal;
    fE = vPosition.xyz;
    fL = LightPosition.xyz;

    if( LightPosition.w != 0.0 ) {
        fL = LightPosition.xyz - vPosition.xyz;
    }

    gl_Position = Projection*ModelView*vPosition;
}
```

Fragment Shader

A.7.3 Fragment Shader

```
#version 150

// per-fragment interpolated values from the vertex shader
in  vec3 fN;
in  vec3 fL;
in  vec3 fE;

out vec4 fColor;

uniform vec4 AmbientProduct, DiffuseProduct, SpecularProduct;
uniform mat4 ModelView;
uniform vec4 LightPosition;
uniform float Shininess;

void main()
{
    // Normalize the input lighting vectors
    vec3 N = normalize(fN);
    vec3 E = normalize(fE);
    vec3 L = normalize(fL);

    vec3 H = normalize( L + E );

    vec4 ambient = AmbientProduct;

    float Kd = max(dot(L, N), 0.0);
    vec4 diffuse = Kd*DiffuseProduct;

    float Ks = pow(max(dot(N, H), 0.0), Shininess);
    vec4 specular = Ks*SpecularProduct;

    // discard the specular highlight if the light's behind the vertex
    if( dot(L, N) < 0.0 ) {
        specular = vec4(0.0, 0.0, 0.0, 1.0);
    }

    fColor = ambient + diffuse + specular;
    fColor.a = 1.0;
}
```

Yet Another Way

Vertex Lighting Shaders I

```
// vertex shader
in vec4 vPosition;
in vec3 vNormal;
out vec4 color;  //vertex shade

// light and material properties
uniform vec4 AmbientProduct, DiffuseProduct, SpecularProduct;
uniform mat4 ModelView;
uniform mat4 Projection;
uniform vec4 LightPosition;
uniform float Shininess;
```

Vertex Lighting Shaders II

```
void main()
{
   // Transform vertex  position into eye coordinates
   vec3 pos = (ModelView * vPosition).xyz;

   vec3 L = normalize( LightPosition.xyz - pos );
   vec3 E = normalize( -pos );
   vec3 H = normalize( L + E );

   // Transform vertex normal into eye coordinates
   vec3 N = normalize( ModelView*vec4(vNormal, 0.0) ).xyz;
```

Vertex Lighting Shaders II

```
void main()
{
    // Transform vertex  position into eye coordinates
    vec3 pos = (ModelView * vPosition).xyz;

    vec3 L = normalize( LightPosition.xyz - pos );
    vec3 E = normalize( -pos );
    vec3 H = normalize( L + E );

    // Transform vertex normal into eye coordinates
    vec3 N = normalize( ModelView*vec4(vNormal, 0.0) ).xyz;
```


THE OHIO STATE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Vertex Lighting Shaders III

```
// Compute terms in the illumination equation
    vec4 ambient = AmbientProduct;

    float Kd = max( dot(L, N), 0.0 );
    vec4  diffuse = Kd*DiffuseProduct;
    float Ks = pow( max(dot(N, H), 0.0), Shininess );
    vec4  specular = Ks * SpecularProduct;
    if( dot(L, N) < 0.0 )  specular = vec4(0.0, 0.0, 0.0, 1.0);
    gl_Position = Projection * ModelView * vPosition;

    color = ambient + diffuse + specular;
    color.a = 1.0;
}
```

Vertex Lighting Shaders IV

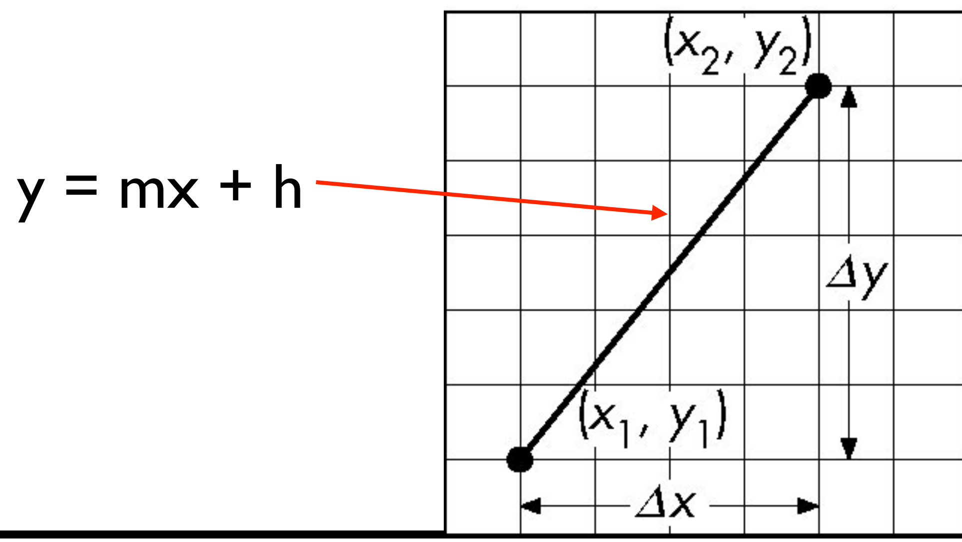
```
// fragment shader

in vec4 color;

void main()
{
   gl_FragColor = color;
}
```

Scan-Line Rasterization

ScanConversion -Line Segments

- Start with line segment in window coordinates with integer values for endpoints
- Assume implementation has a **write_pixel** function

$$m = \frac{\Delta y}{\Delta x}$$



DDA Algorithm

- Digital Differential Analyzer
  - Line y=mx+ h satisfies differential equation

    $dy/dx = m = Dy/Dx = y_2\text{-}y_1/x_2\text{-}x_1$

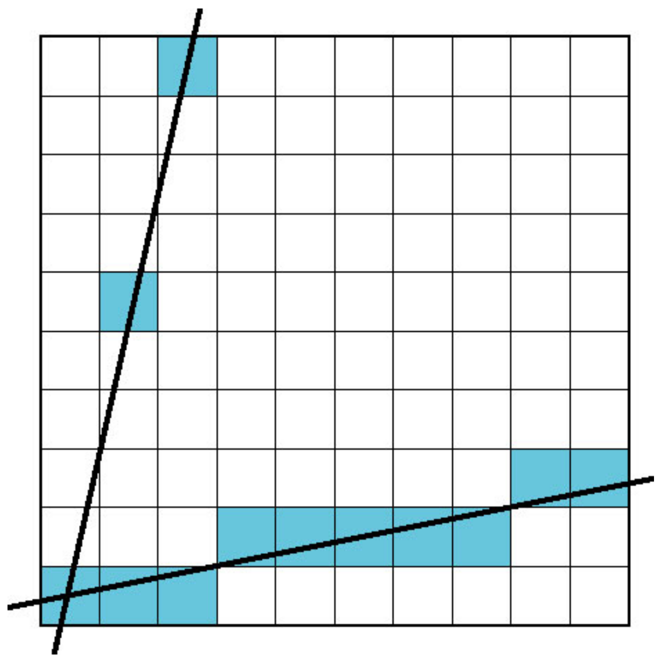
- Along scan line Dx = 1

```
For(x=x1; x<=x2,ix++) {
   y+=m;
 display (x, round(y), line_color)
}
```

Problem

DDA = for each x plot pixel at closest y

– Problems for steep lines

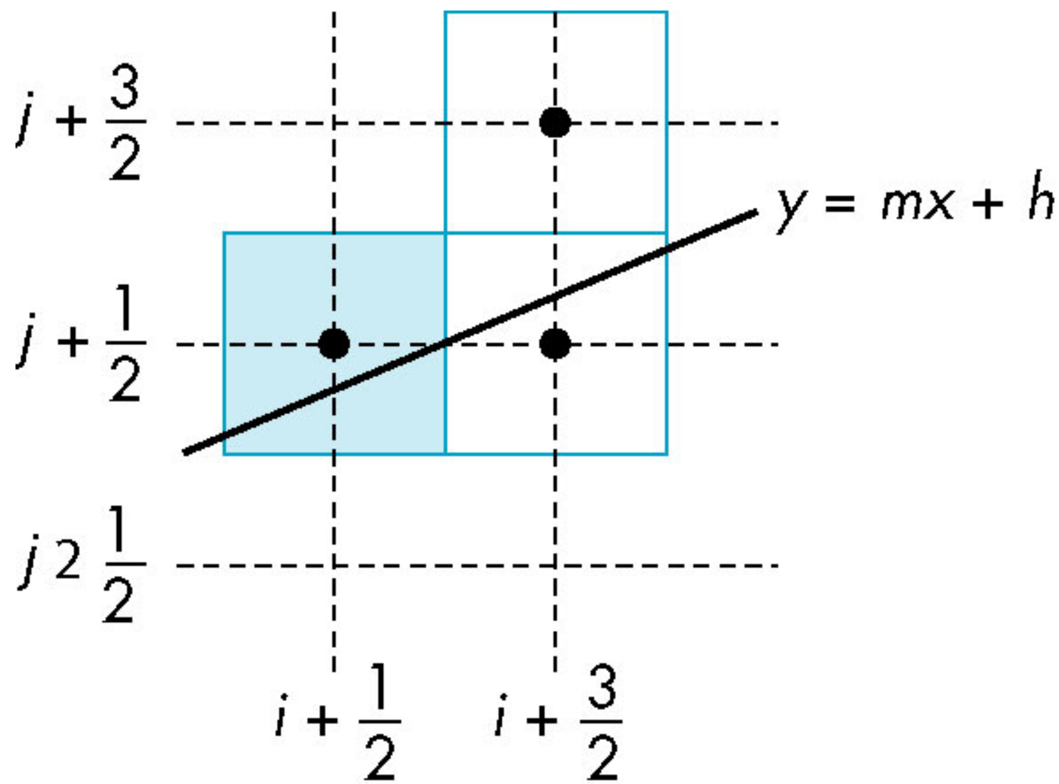


Bresenham's Algorithm

- DDA requires one floating point addition per step
- Eliminate computations through Bresenham's algorithm
- Consider only $1 \geq m \geq 0$
  - Other cases by symmetry
- Assume pixel centers are at half integers

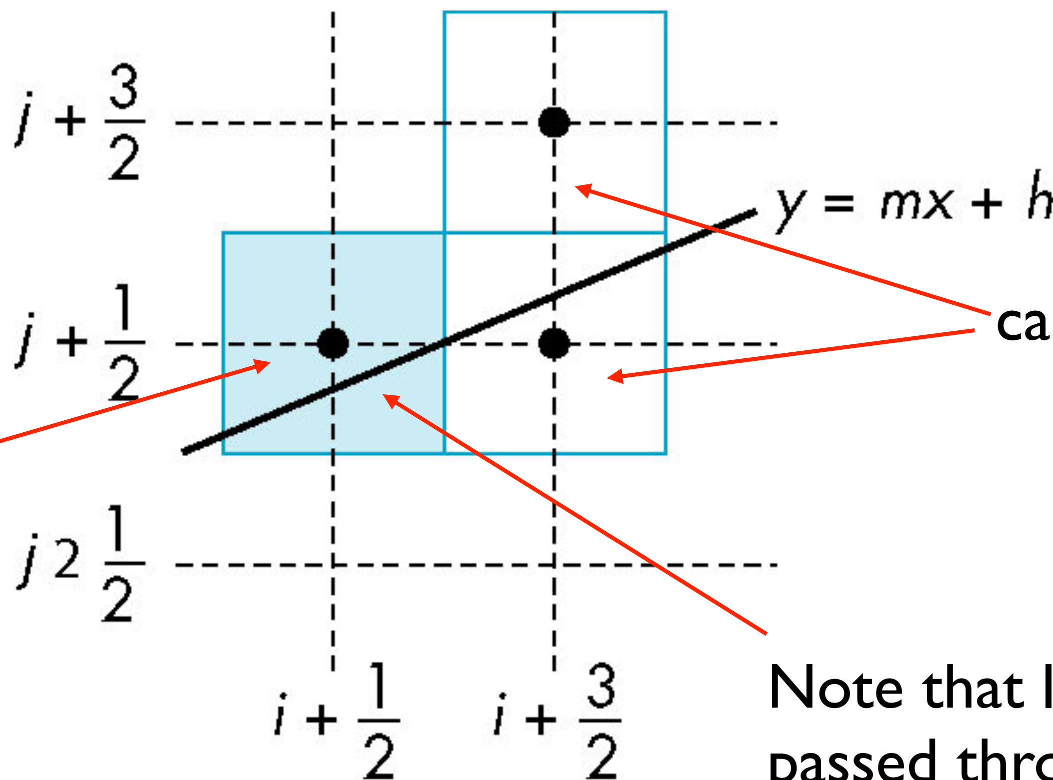
Main Premise

If we start at a pixel that has been written, there are only two candidates for the next pixel to be written into the frame buffer



Candidate Pixels

1 ≥ m ≥ 0



candidates

last pixel

Note that line could have passed through any part of this pixel

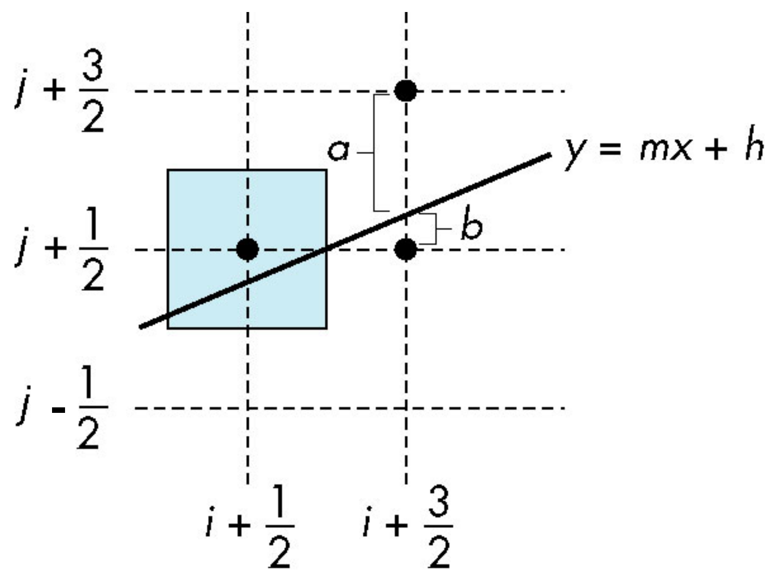
Decision Variable

$d = \Delta x(b-a)$

d is an integer

d > 0 use upper pixel

d < 0 use lower pixel



Incremental Form

Inspect d_k at $x = k$

$$d_{k+1} = d_k - 2Dy, \quad \text{if } d_k < 0$$
$$d_{k+1} = d_k - 2(Dy - Dx), \quad \text{otherwise}$$

For each x, we need do only an integer addition and test

Single instruction on graphics chips

Polygon Scan Conversion

- Scan Conversion = Fill
- How to tell inside from outside
  - Convex easy
  - Nonsimple difficult
  - Odd even test
    - Count edge crossings

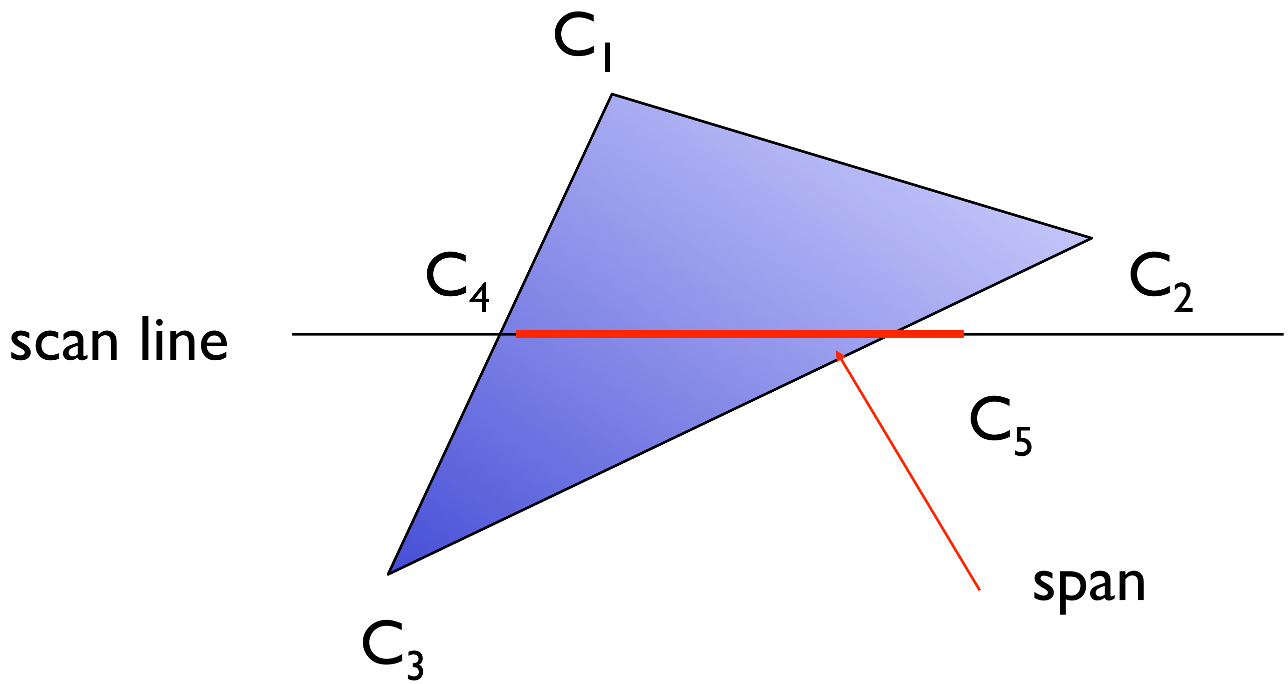
Filling in the Frame Buffer

Fill at end of pipeline

- Convex Polygons only
- Nonconvex polygons assumed to have been tessellated
- Shades (colors) have been computed for vertices (Gouraud shading)
- Combine with z-buffer algorithm
  - March across scan lines interpolating shades
  - Incremental work small

Using Interpolation

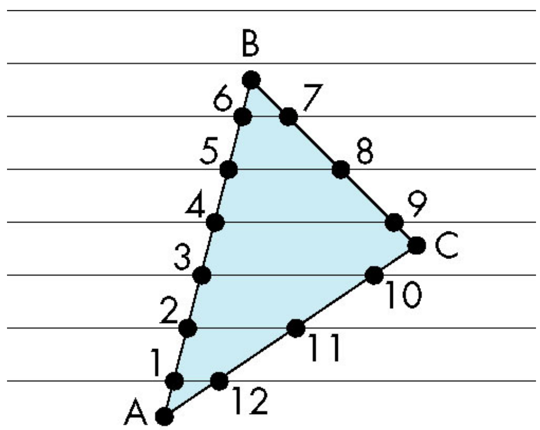
C_1 C_2 C_3 specified by **glColor** or by vertex shading
C_4 determined by interpolating between C_1 and C_2
C_5 determined by interpolating between C_2 and C_3
interpolate between C_4 and C_5 along span



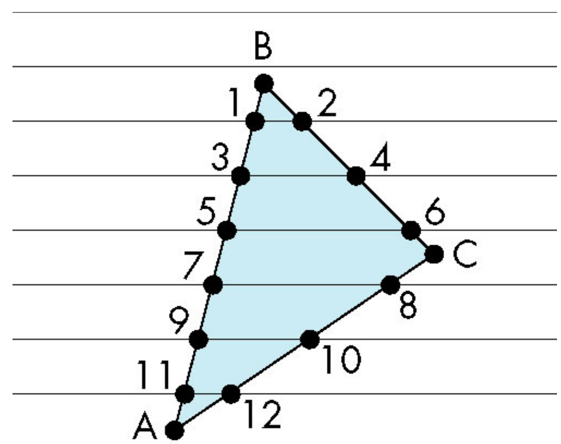
Scan Line Fill

Can also fill by maintaining a data structure of all intersections of polygons with scan lines

- Sort by scan line
- Fill each span

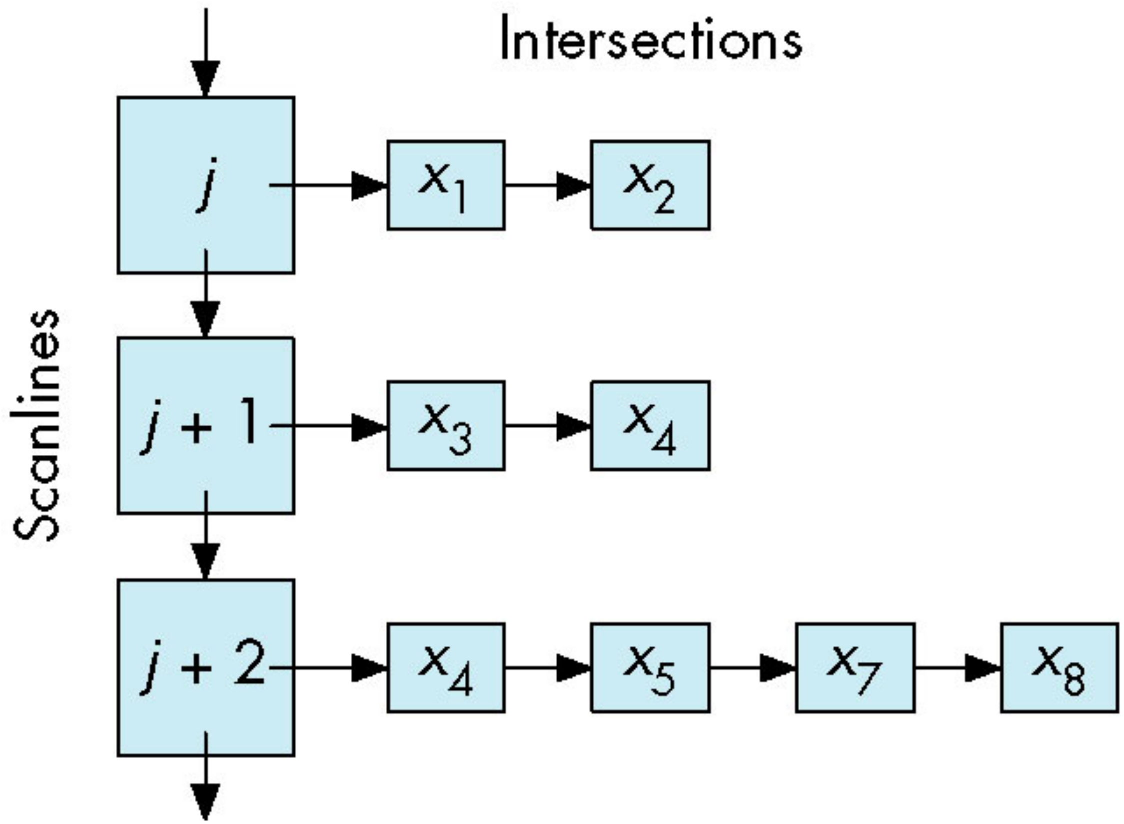


vertex order generated by vertex list



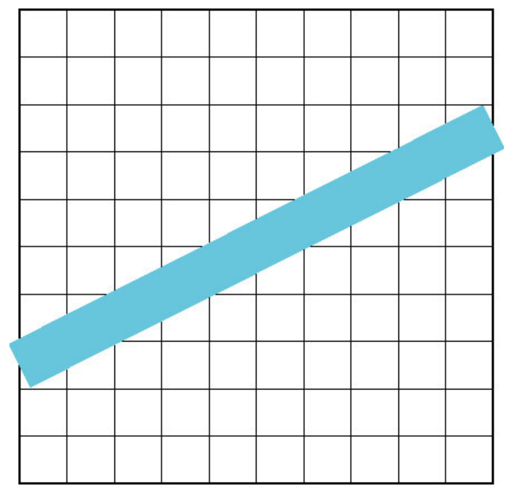
desired order

Data Structure



Aliasing

- Ideal rasterized line should be 1 pixel wide

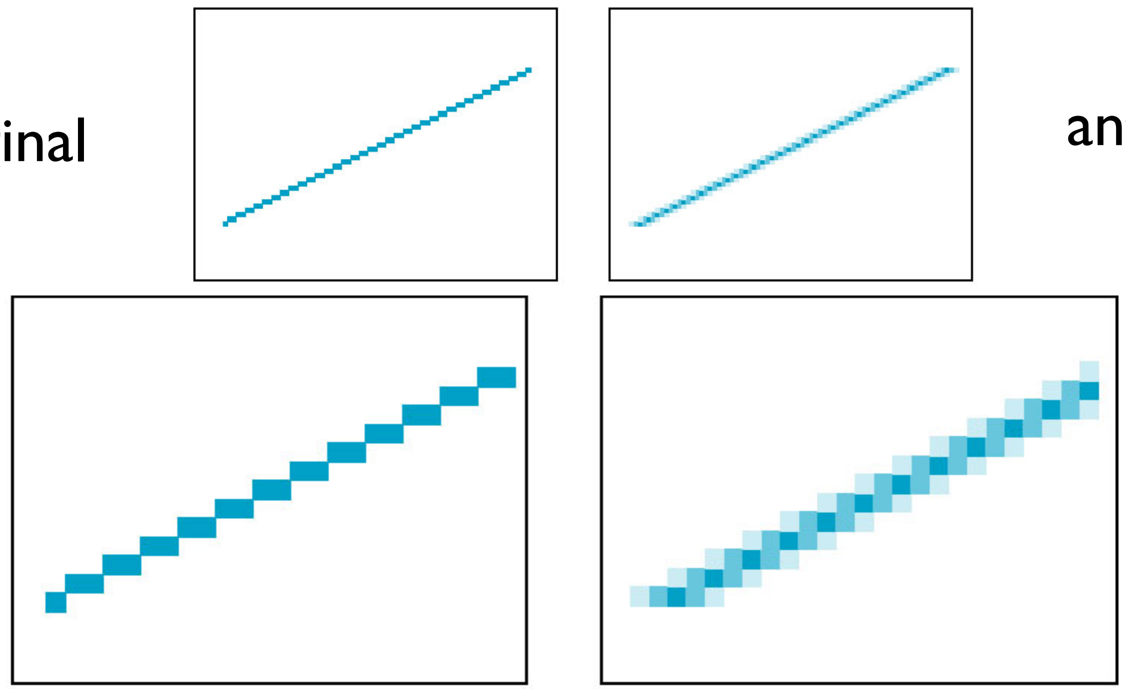


- Choosing best y for each x (or visa versa) produces aliased raster lines

Antialiasing by Area Averaging

- Color multiple pixels for each x depending on coverage by ideal line

original

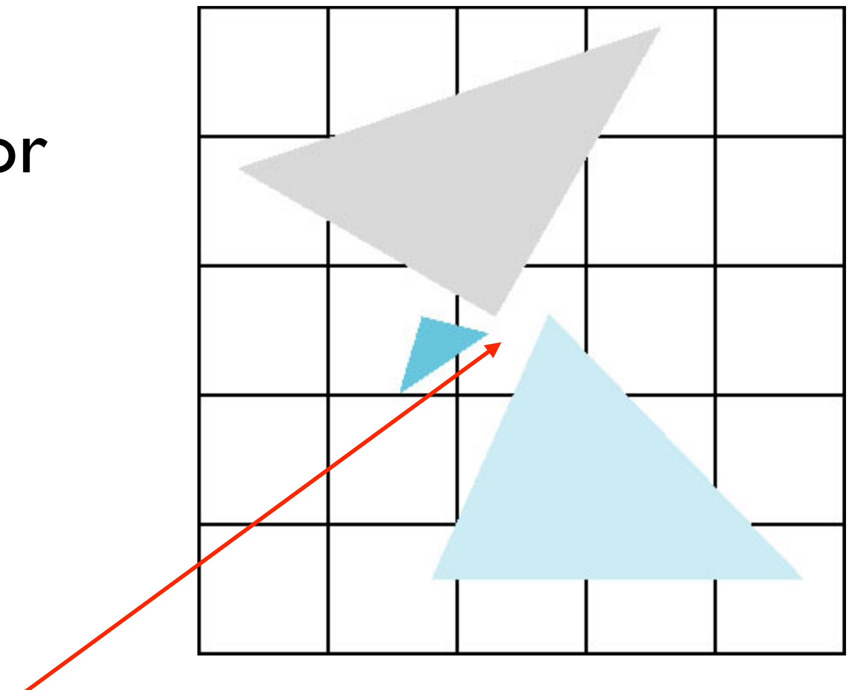


antialiased

magnified

Polygon Aliasing

- Aliasing problems can be serious for polygons
  - Jaggedness of edges
  - Small polygons neglected
  - Need compositing so color of one polygon does not totally determine color of pixel



All three polygons should contribute to color

Hierarchical Modeling

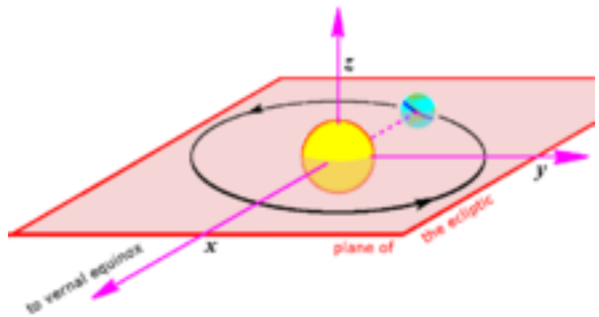
Cars, Robots, Solar System

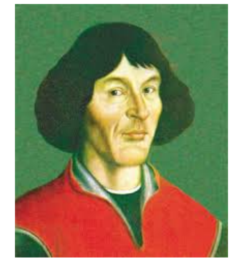
The Terminator

Our Goal ☺


Pluto
Neptune
Uranus
Saturn
Jupiter
Mars
Moon
Earth
Venus
Mercury

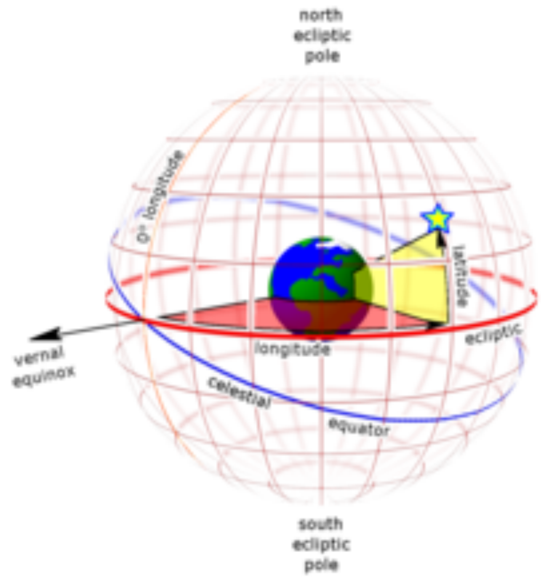
Heliocentric Coordinates



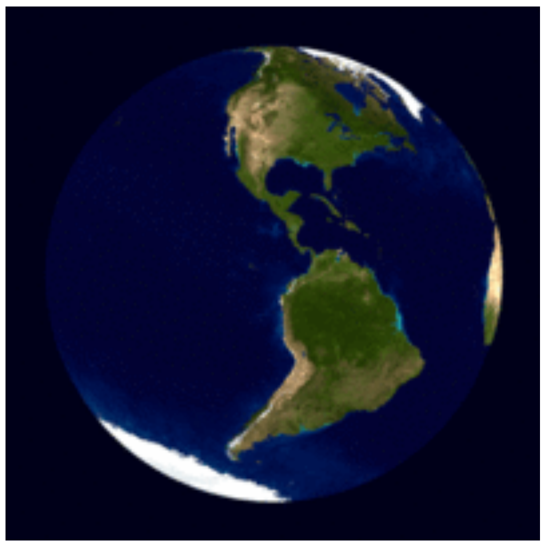


Heliocentric ecliptic coordinates. The origin is the center of the Sun. The fundamental plane is the plane of the ecliptic. The primary direction (the x axis) is the vernal equinox. A right-handed convention specifies a y axis 90° to the east in the fundamental plane; the z axis points toward the north ecliptic pole. The reference frame is relatively stationary, aligned with the vernal equinox.

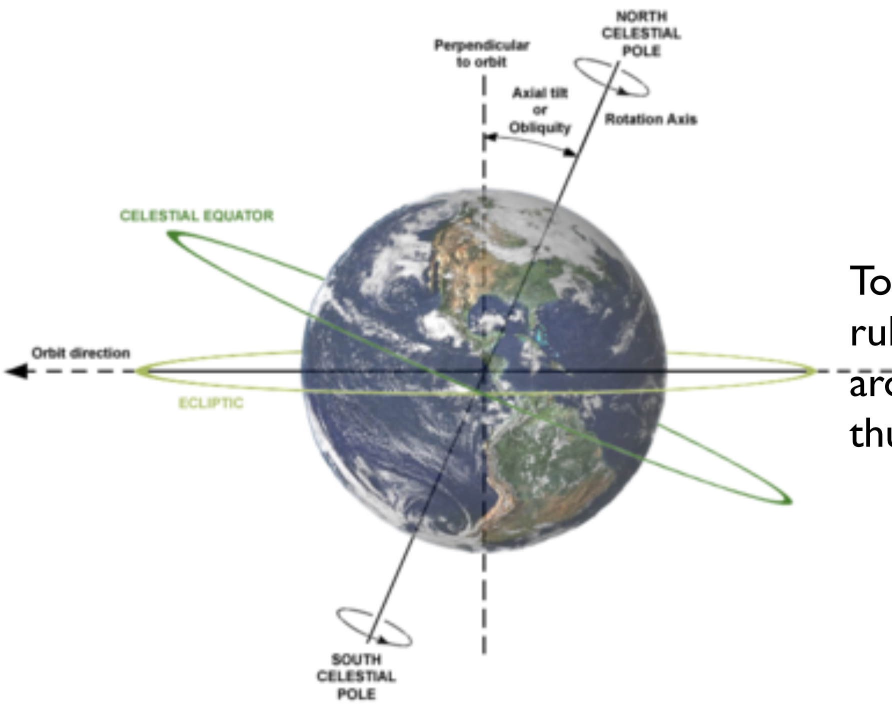
Inclinations



Axial Tilt

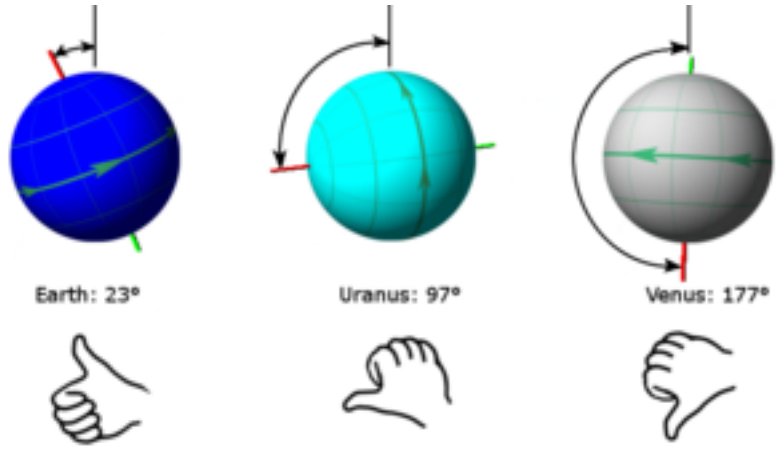


W. Pedia says



To understand axial tilt, we employ the right-hand rule. When the fingers of the right hand are curled around in the direction of the planet's rotation, the thumb points in the direction of the north pole.

Axial Tilts of Planets



The axial tilt of three planets: Earth, Uranus, and Venus. Here, a vertical line (black) is drawn perpendicular to the plane of each planet's orbit. The angle between this line and the planet's north pole (red) is the tilt. The surrounding arrows (green) show the direction of the planet's rotation.

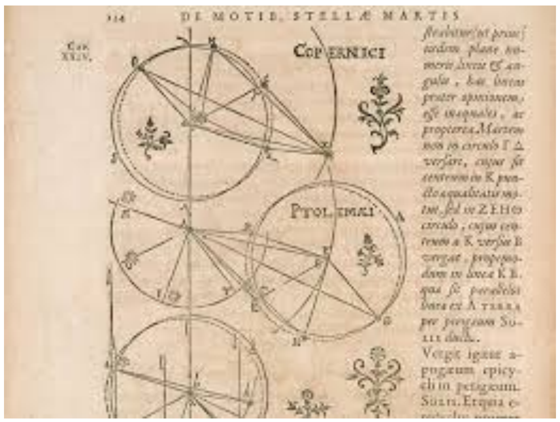
Ecliptic Coordinate System

$$\begin{bmatrix} x_{equatorial} \\ y_{equatorial} \\ z_{equatorial} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\epsilon & -\sin\epsilon \\ 0 & \sin\epsilon & \cos\epsilon \end{bmatrix} \cdot \begin{bmatrix} x_{ecliptic} \\ y_{ecliptic} \\ z_{ecliptic} \end{bmatrix}$$

$$\begin{bmatrix} x_{ecliptic} \\ y_{ecliptic} \\ z_{ecliptic} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\epsilon & \sin\epsilon \\ 0 & -\sin\epsilon & \cos\epsilon \end{bmatrix} \cdot \begin{bmatrix} x_{equatorial} \\ y_{equatorial} \\ z_{equatorial} \end{bmatrix}$$

where \epsilon is the obliquity of the ecliptic.

Roots

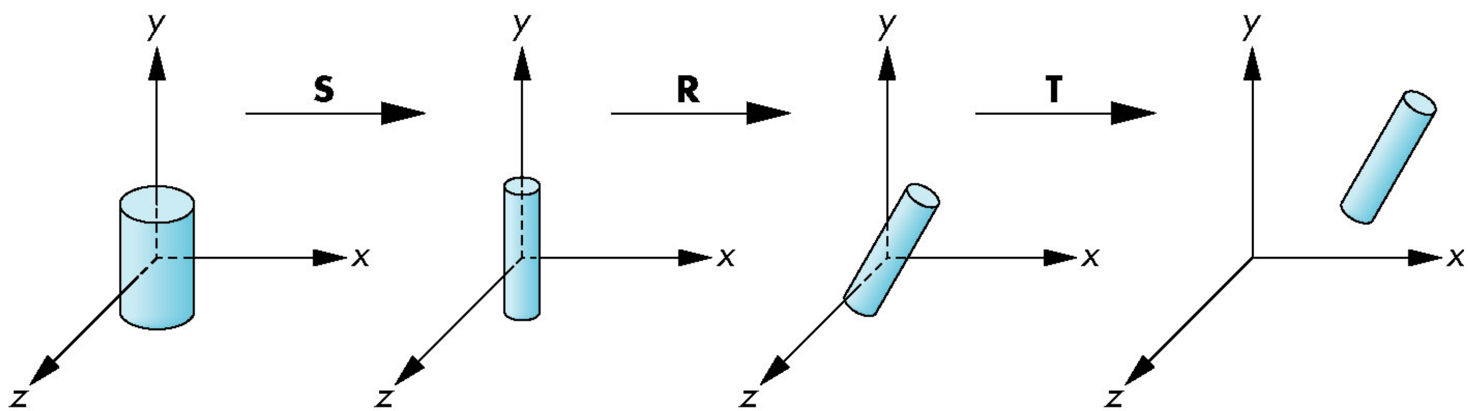

DE MOTIB. STELLÆ MARTIS
COPERNICI
PTOLEMAI


MERCURY
VENUS
EARTH
MARS
JUPITER
SATURN

Back 2 Earth ☺

Instance Transformation

- Start with prototype object
- Each appearance of object in model is *instance*
  - Must scale, orient, position
  - Defines instance transformation



THE OHIO STATE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

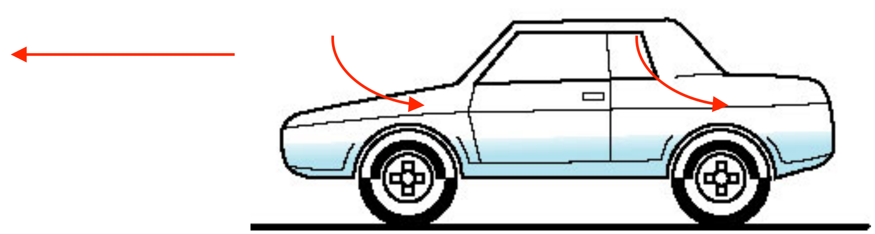
Symbol-Instance Table

Symbol	Scale	Rotate	Translate
1	s_x, s_y, s_z	$\theta_x, \theta_y, \theta_z$	d_x, d_y, d_z
2			
3			
1			
1			
.			
.			

THE OHIO STATE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Relationships

- Car
  - Chassis + 4 identical wheels
  - Two symbols
- Rate of forward motion function of rotational speed of wheels



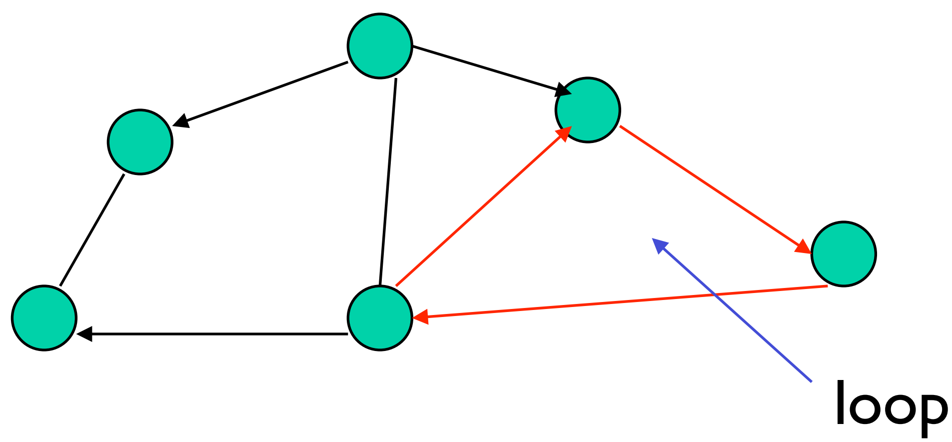
THE OHIO STATE UNIVERSITY DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Move The Car

```
car(speed)
{
    chassis()
    wheel(right_front);
    wheel(left_front);
    wheel(right_rear);
    wheel(left_rear);
}
```

Graphs – Composition of Car

- Set of *nodes* and *edges (links)*
- Edge connects a pair of nodes
  - Directed or undirected
- *Cycle*: directed path that is a loop

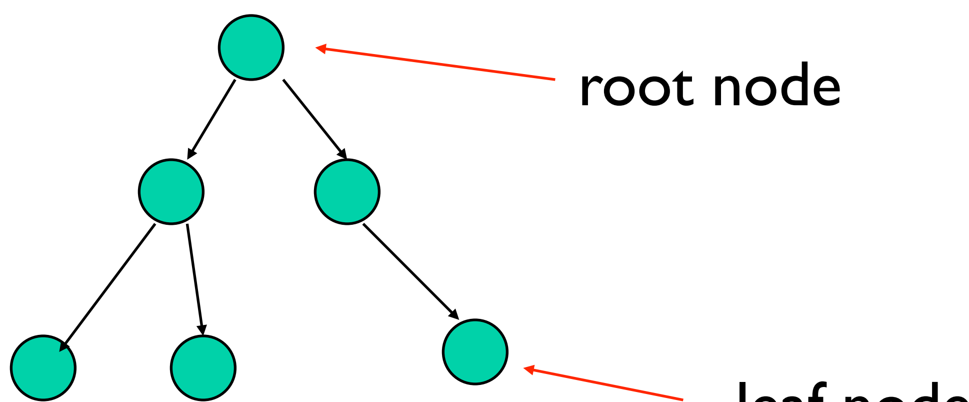


THE OHIO STATE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

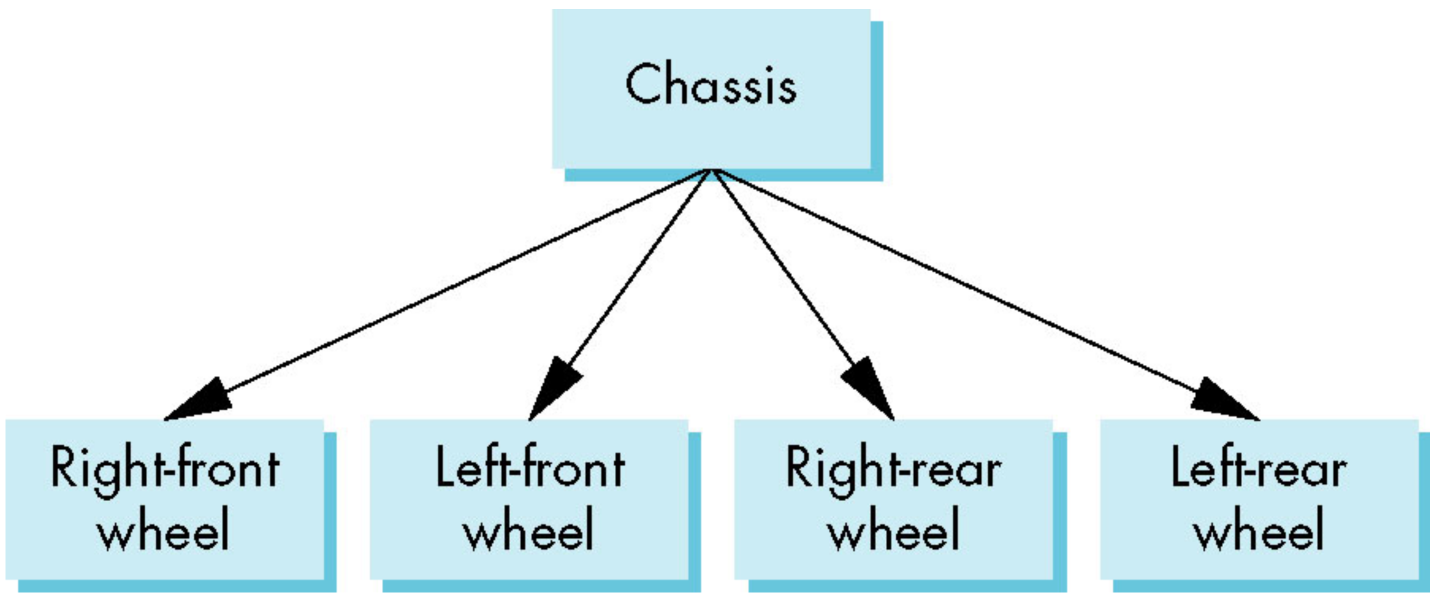
Tree – Composition of Car

Graph in which each node (except the root) has exactly one parent node

- May have multiple children
- Leaf or terminal node: no children



Tree Model of Car

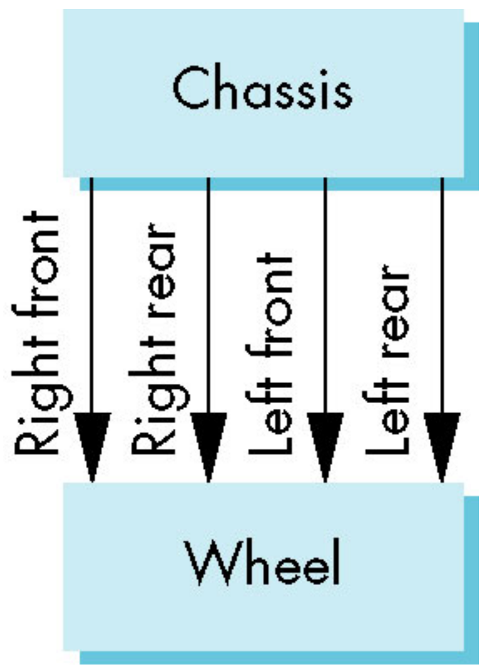


THE OHIO STATE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

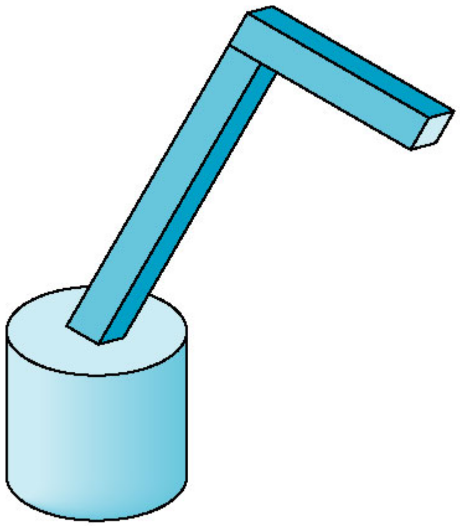
DAG Model

All the wheels are identical

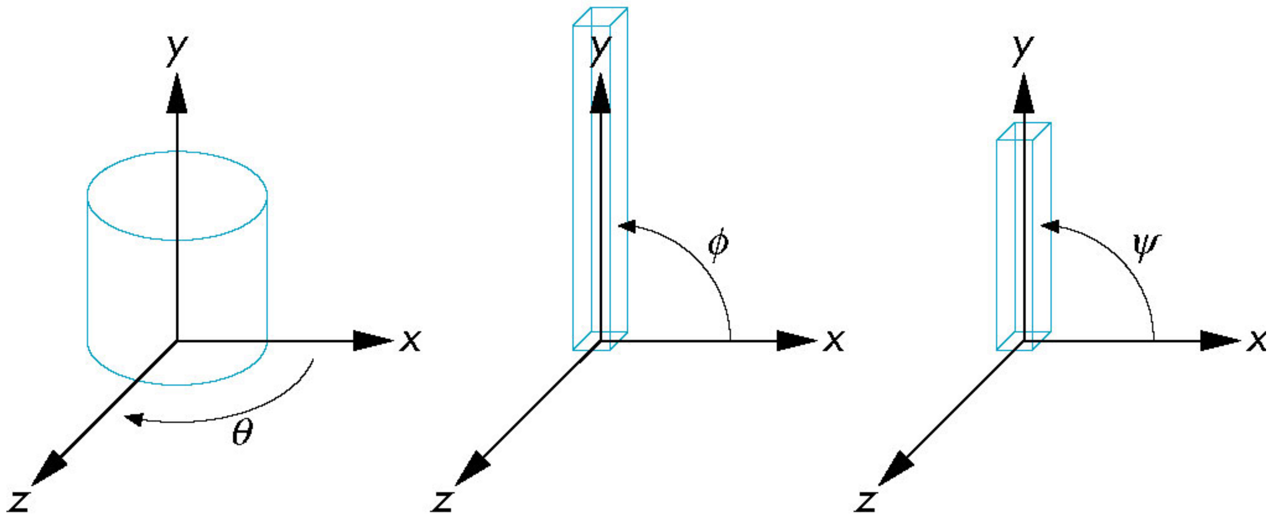
Not much different than dealing with a tree



Robot Arm



robot arm

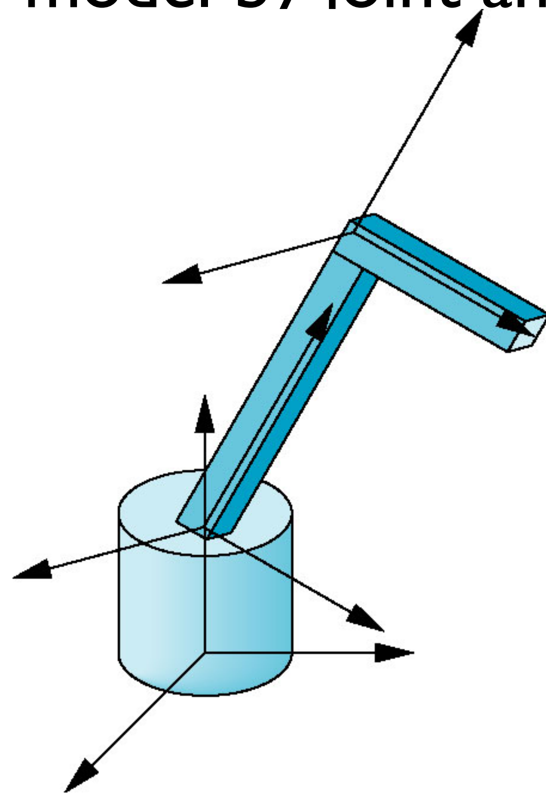


parts in their own coodinate systems

Articulated Models

– Parts connected at joints

– Specify state of model by joint angles



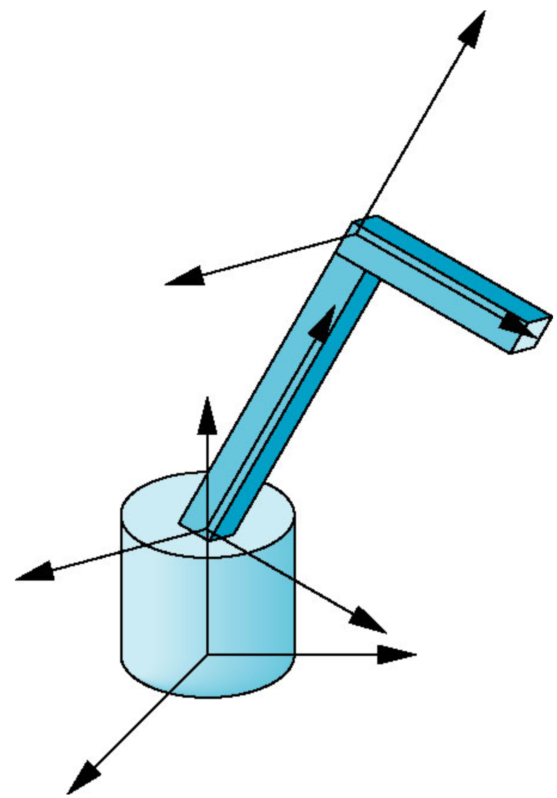
THE OHIO STATE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Relationships - Composition

- Base
- Lower Arm
- Upper Arm

Base

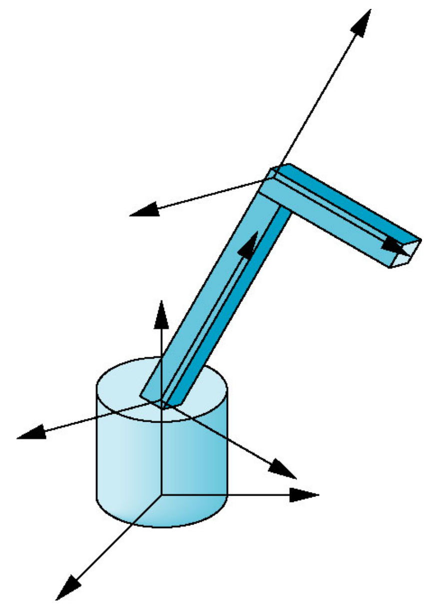
- Single angle determines position
- Is cylinder



Lower Arm

Attached to base

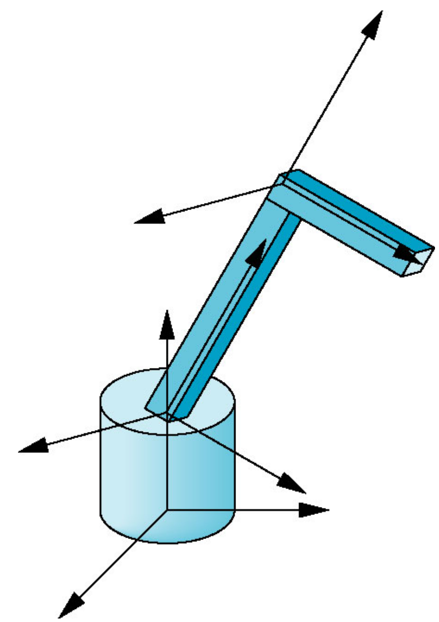
- Position depends on rotation of base
- Also translate relative to base, rotate about connecting joint
- Is cube



Upper Arm

Upper arm attached to lower arm

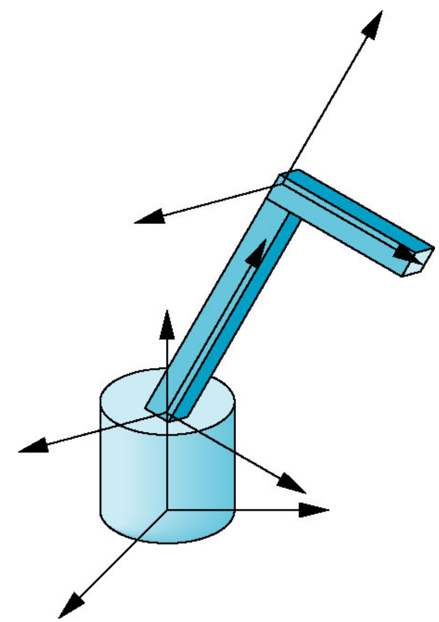
- Its position depends on both base and lower arm
- Translate relative to lower arm and rotate about joint connecting to lower arm



Upper Arm

Upper arm attached to lower arm

- Its position depends on both base and lower arm
- Translate relative to lower arm and rotate about joint connecting to lower arm



Do the same …

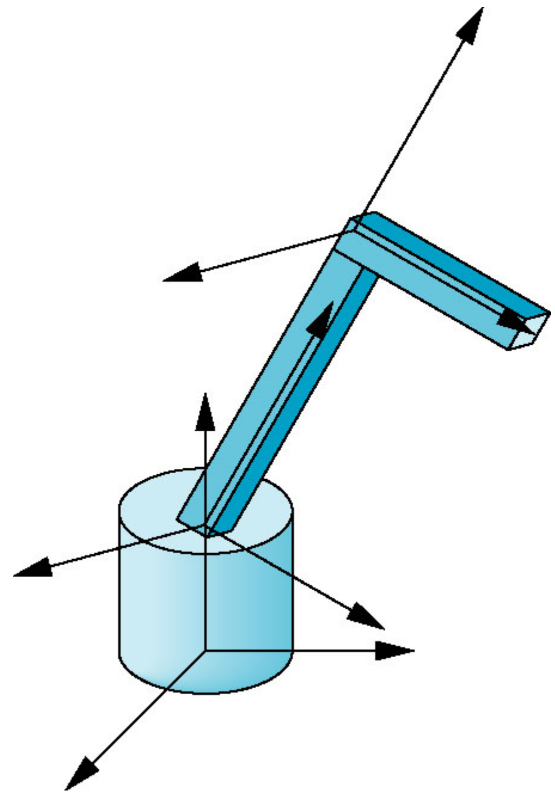

Pluto
Neptune
Uranus
Saturn
Jupiter
Mars
Moon
Earth
Venus
Mercury

Required Matrices

Base

Rotation of base: $\mathbf{R}_b$

– Apply $\mathbf{M} = \mathbf{R}_b$ to base

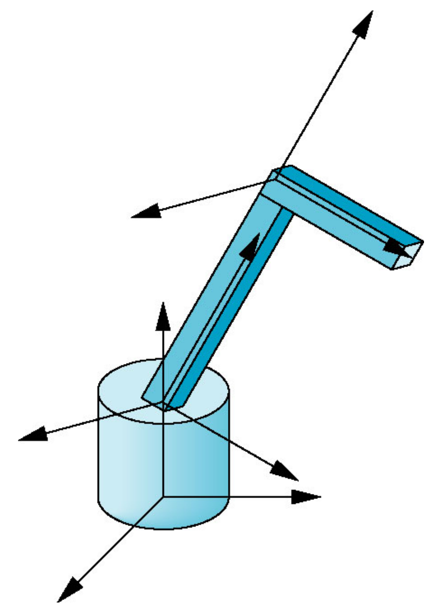


Lower Arm

Translate lower arm relative to base: $\mathbf{T}_{lu}$

Rotate lower arm around joint: $\mathbf{R}_{lu}$

- Apply $\mathbf{M} = \mathbf{R}_b \mathbf{T}_{lu} \mathbf{R}_{lu}$ to lower arm

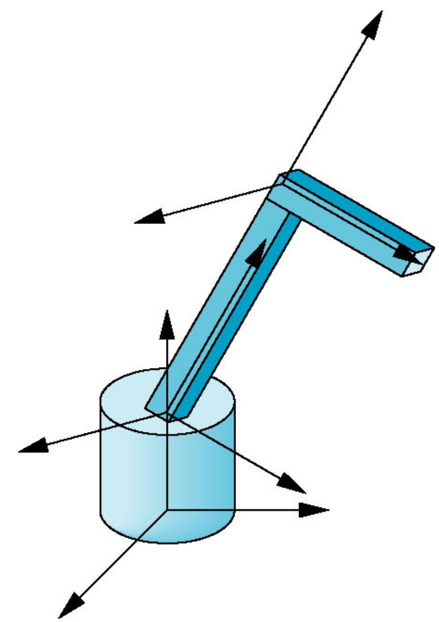


Upper Arm

Translate upper arm <u>relative</u> to upper arm: $\mathbf{T}_{uu}$

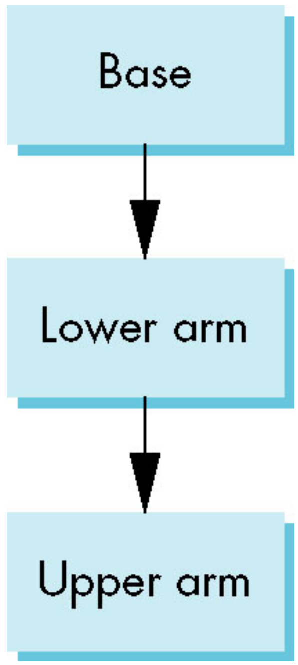
Rotate upper arm around joint: $\mathbf{R}_{uu}$

- Apply $\mathbf{M} = \mathbf{R}_b \mathbf{T}_{lu} \mathbf{R}_{lu} \mathbf{T}_{uu} \mathbf{R}_{uu}$ to upper arm



Simple Robot

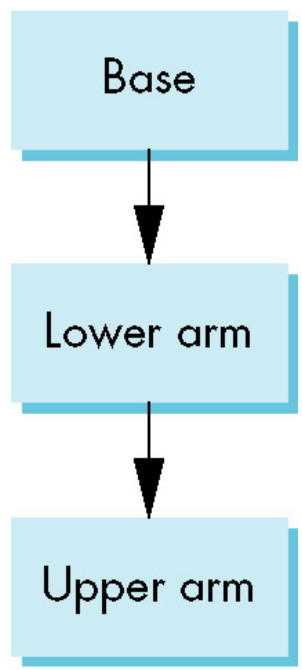
```
mat4 ctm;
robot_arm()
{
    ctm = RotateY(theta);
    base();
    ctm *= Translate(0.0, h1, 0.0);
    ctm *= RotateZ(phi);
    lower_arm();
    ctm *= Translate(0.0, h2, 0.0);
    ctm *= RotateZ(psi);
    upper_arm();
}
```



Tree Model of Robot

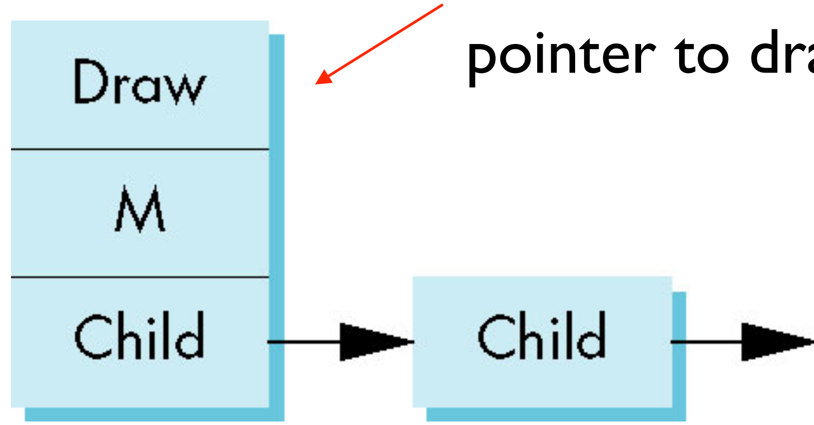
Code shows relationships between parts of model

– Can change shape/texture w/o altering relationships



Possible Node Structure

Code for drawing part or pointer to drawing function



linked list of pointers to children

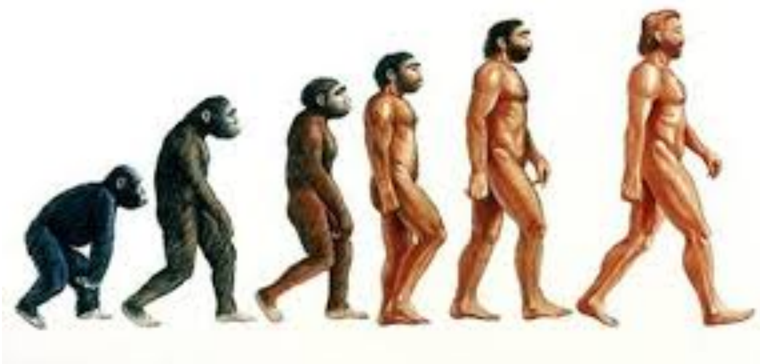
matrix relating node to parent

THE OHIO STATE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Do the same …


Pluto
Neptune
Uranus
Saturn
Jupiter
Mars
Moon
Earth
Venus
Mercury

Generalizations

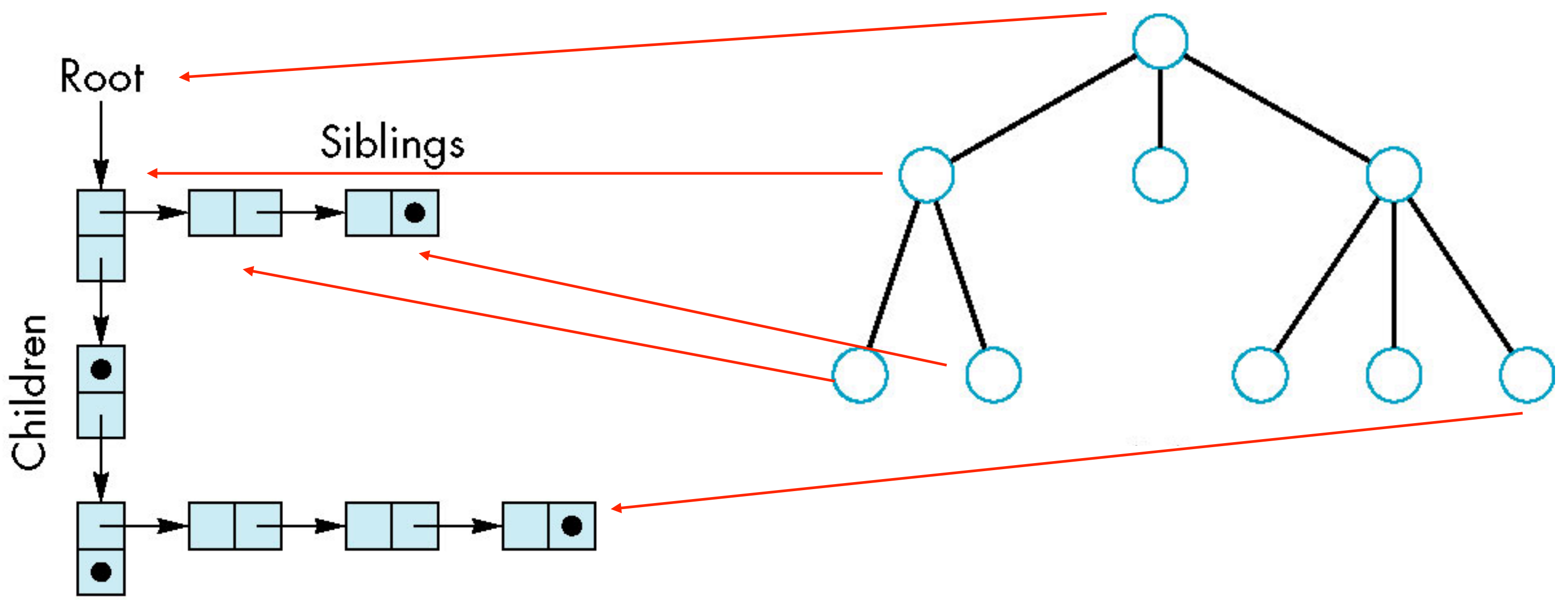


Generalizations

- Need to deal with multiple children
  - How do we represent a more general tree?
  - How do we traverse such a data structure?
- Animation
  - How to use dynamically?
  - Can we create and delete nodes during execution?

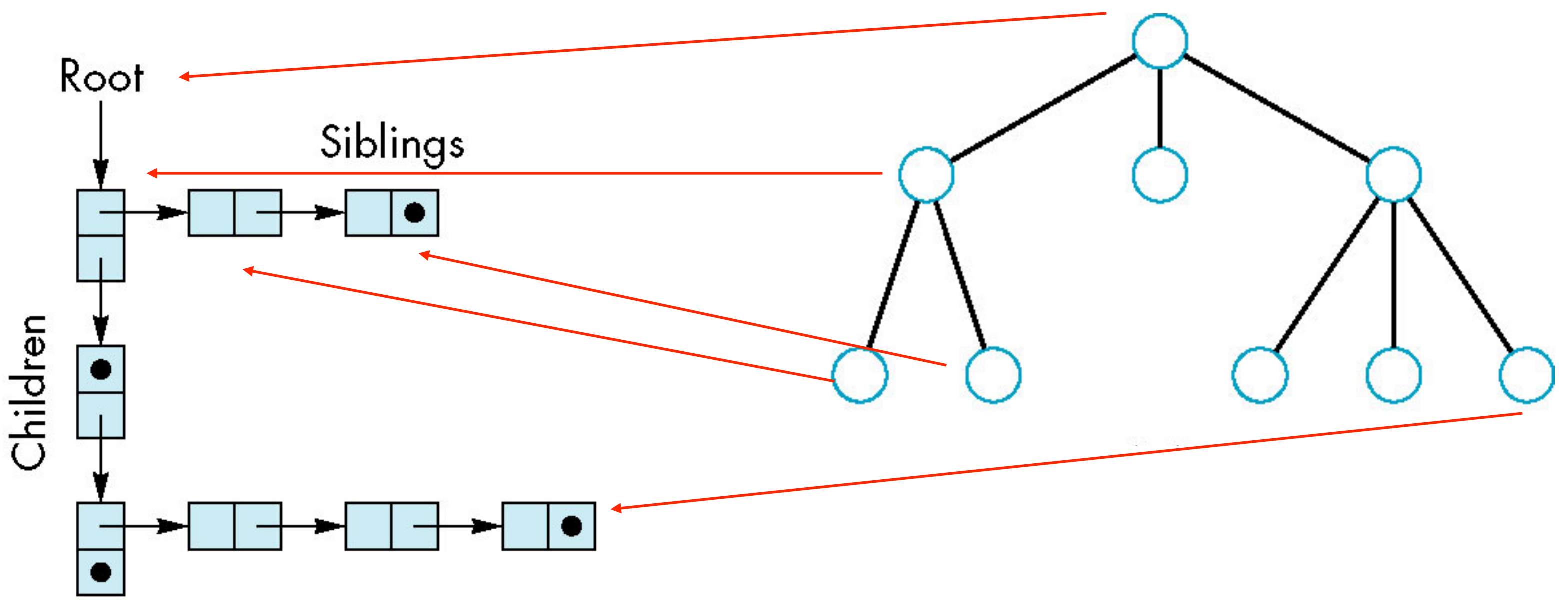
THE OHIO STATE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Breadth-First Tree

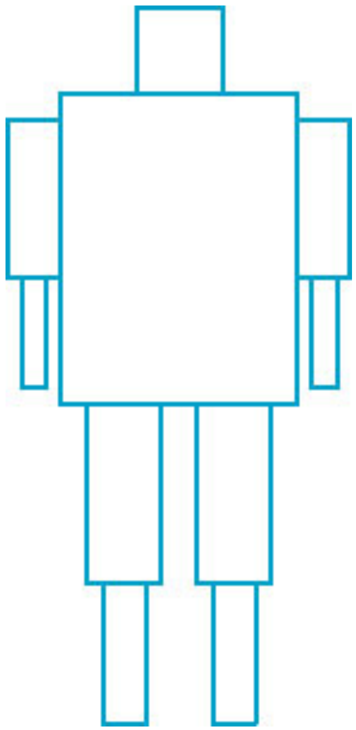


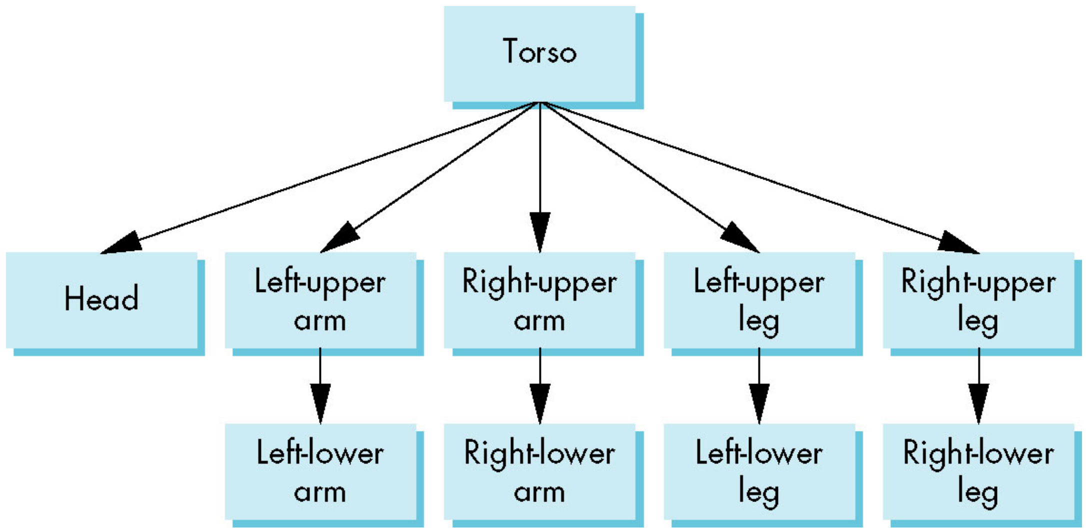
THE OHIO STATE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Solar System ?



Humanoid Figure



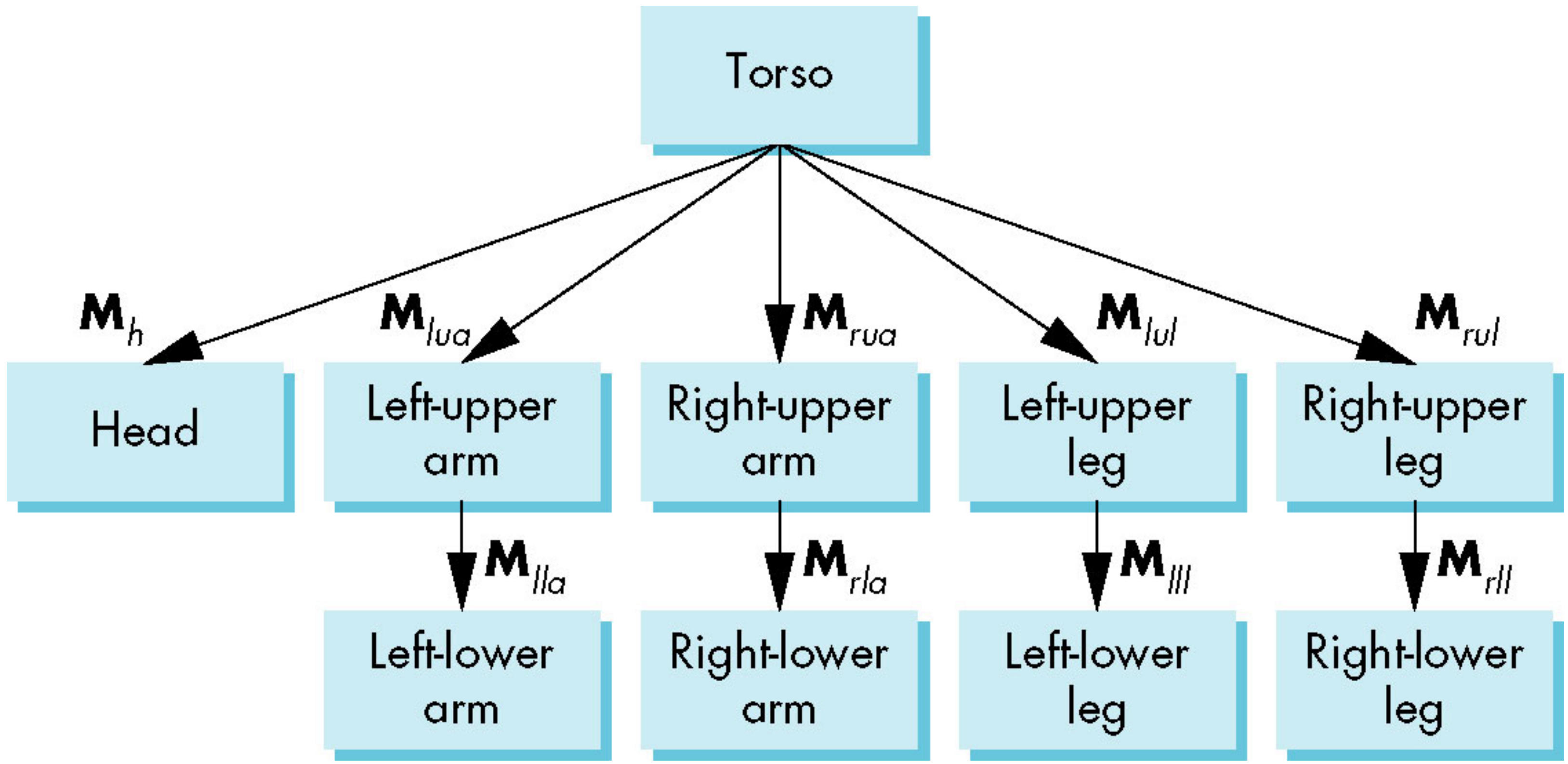


Building the Model

- Implementation using quadrics: ellipsoids and cylinders
- Access parts through functions
  - torso()
  - left_upper_arm()
- Matrices describe position of node with respect to parent
  - $\mathbf{M}_{lla}$ positions leftlowerleg with respect to leftupperarm

THE OHIO STATE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Matrices Tree



Display and Traversal

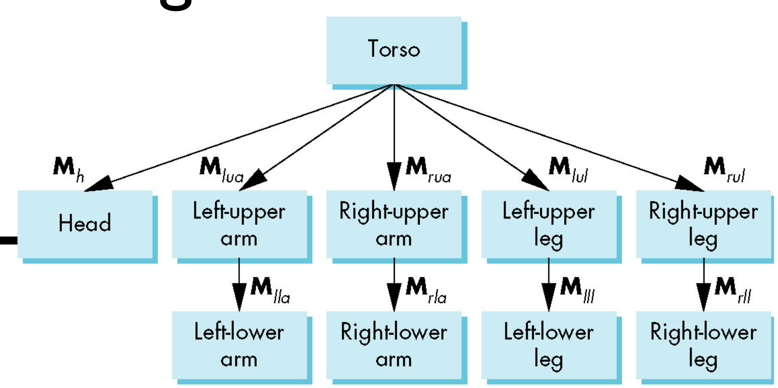
- The position determined by 11 joint angles (two for the head and one for each other part)
- Display of the tree requires a *graph traversal*
  - Visit each node once
  - Display function at each node pertaining to part
  - Applying correct transformation matrix for position and orientation

THE OHIO STATE UNIVERSITY DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Transformation Matrices

10 relevant matrices

- **M** positions and orients entire figure through the torso which is the root node
- $\mathbf{M}_h$ positions head with respect to torso
- $\mathbf{M}_{lua}$, $\mathbf{M}_{rua}$, $\mathbf{M}_{lul}$, $\mathbf{M}_{rul}$ position arms and legs with respect to torso
- $\mathbf{M}_{lla}$, $\mathbf{M}_{rla}$, $\mathbf{M}_{lll}$, $\mathbf{M}_{rll}$ position lower parts of limbs with respect to corresponding upper limbs



Stack-based Traversal

- Set model-view matrix to **M** and draw torso
- Set model-view matrix to $\mathbf{MM}_h$ and draw head
- For left-upper arm need $\mathbf{MM}_{lua}$ and so on

- No need recomputing $\mathbf{Mm}_{lua}$
  - Use the matrix stack to store **M** and other matrices in tree traversal

THE OHIO STATE UNIVERSITY DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Old Style GL Code

```
figure() {
  PushMatrix()
  torso();
  Rotate (…);
  head();
  PopMatrix();
  PushMatrix();
  Translate(…);
  Rotate(…);
  left_upper_arm();
  PopMatrix();
  PushMatrix();
```

save present model-view matrix

update model-view matrix for head

recover original model-view matrix

save it again

update model-view matrix for left upper arm

recover and save original model-view matrix again

rest of code

Tree Data Structure

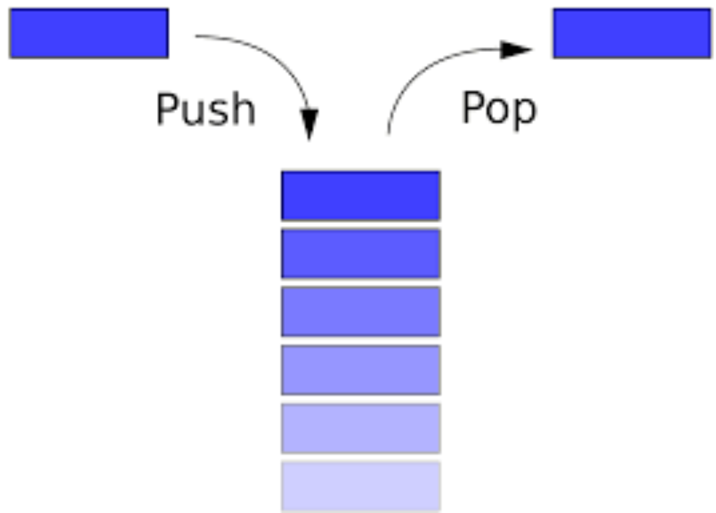
- Represent tree and algorithm to traverse tree
- We will use a *left-child right sibling* structure
  - Uses linked lists
  - Each node in data structure is two pointers
  - Left: next node
  - Right: linked list of children

THE OHIO STATE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

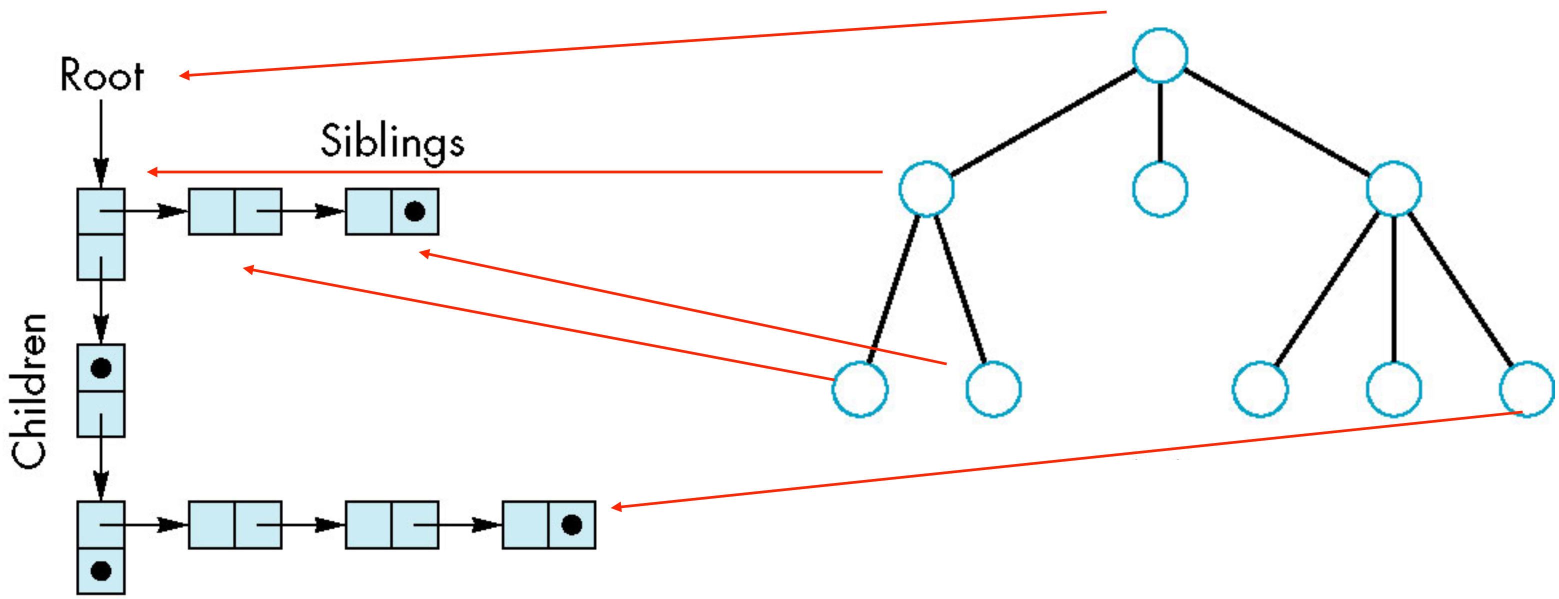
In GLSL

In GLSL

Still Use



Left-Child Right-Sibling Tree



Tree node Structure

At each node

- Pointer to sibling
- Pointer to child
- Pointer to a function that draws the object represented by the node
- Homogeneous coordinate matrix to multiply on the right of the current model-view matrix
  - Represents changes going from parent to node
  - In OpenGL this matrix is a 1D array storing matrix by columns

C

```
typedef struct treenode
{
   mat4 m;
   void (*f)();
   struct treenode *sibling;
   struct treenode *child;
} treenode;
```

torso and head nodes

```
treenode torso_node, head_node, lua_node, … ;
torso_node.m = RotateY(theta[0]);
torso_node.f = torso;
torso_node.sibling = NULL;
torso_node.child =  &head_node;

 head_node.m = translate(0.0, TORSO_HEIGHT
    +0.5*HEAD_HEIGHT, 0.0)*RotateX(theta[1])*RotateY(theta[2]);
head_node.f = head;
head_node.sibling = &lua_node;
head_node.child = NULL;
```


THE OHIO STATE UNIVERSITY DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

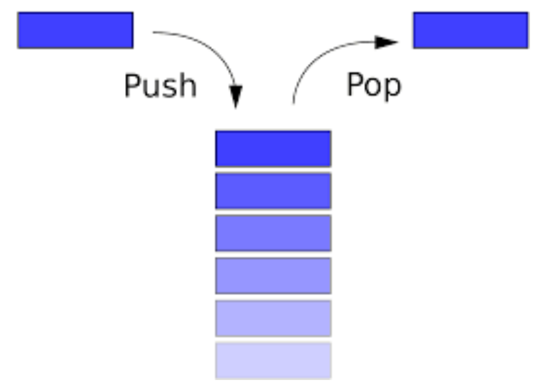
Notes

- Position determined by 11 joint angles in theta[11]
- Animate by changing angles and redisplaying
- Form required matrices using Rotate and Translate

THE OHIO STATE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Preorder Traversal

```
void traverse(treenode* root)
{
   if(root==NULL) return;
   mvstack.push(model_view);
   model_view = model_view*root->m;
   root->f();
   if(root->child!=NULL) traverse(root->child);
   model_view = mvstack.pop();
   if(root->sibling!=NULL) traverse(root->sibling);
}
```



THE OHIO STATE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Notes

- Save model-view matrix before multiplying it by node matrix
  - Updated matrix applies to children but not to siblings
- Traversal applies to any left-child right-sibling tree
  - Particular tree encoded in definition of individual nodes
- Order of traversal matters given state changes in the functions

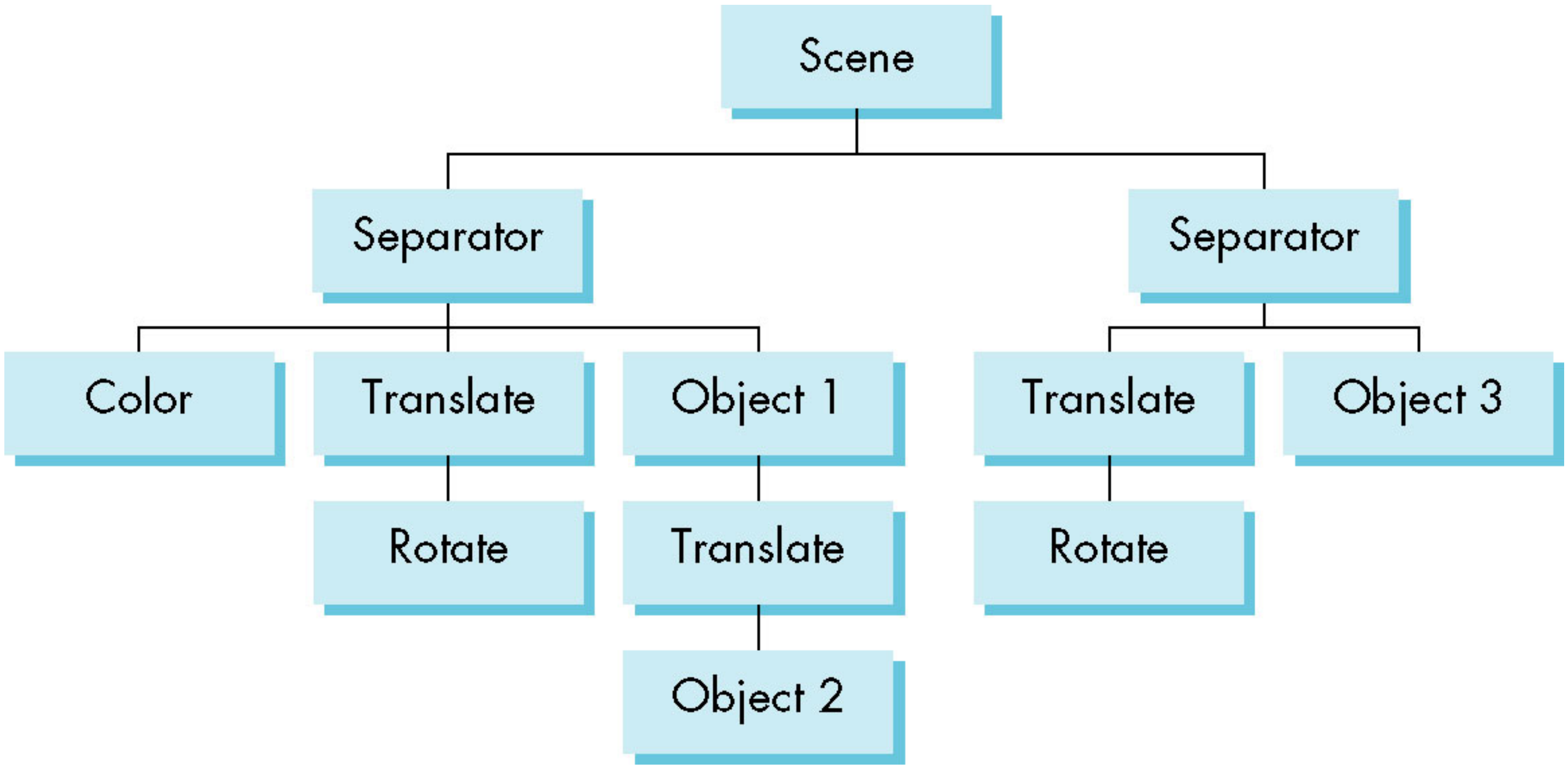
Dynamic Trees

Use pointers, the structure can be dynamic

```
typedef treenode *tree_ptr;
tree_ptr torso_ptr;
torso_ptr = malloc(sizeof(treenode));
```

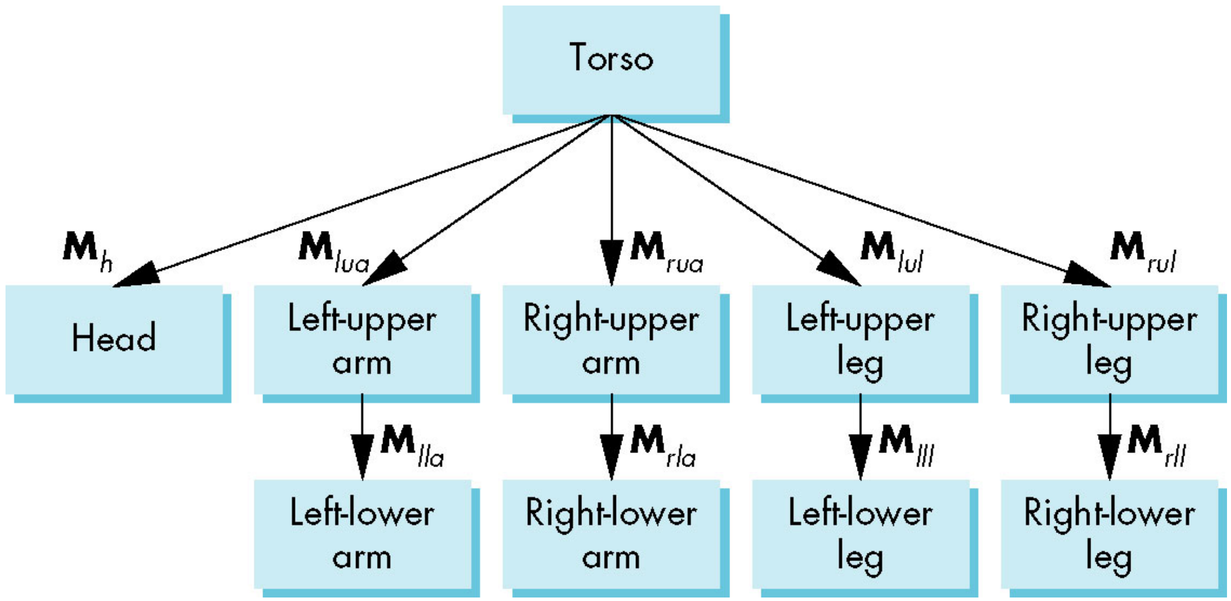
Definition of nodes and traversal are essentially the same as before but we can add and delete nodes during execution

The Real Thing



THE OHIO STATE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

As Opposed


Torso
$\mathbf{M}_h$
$\mathbf{M}_{lua}$
$\mathbf{M}_{rua}$
$\mathbf{M}_{lul}$
$\mathbf{M}_{rul}$
Head
Left-upper arm
Right-upper arm
Left-upper leg
Right-upper leg
$\mathbf{M}_{lla}$
$\mathbf{M}_{rla}$
$\mathbf{M}_{lll}$
$\mathbf{M}_{rll}$
Left-lower arm
Right-lower arm
Left-lower leg
Right-lower leg