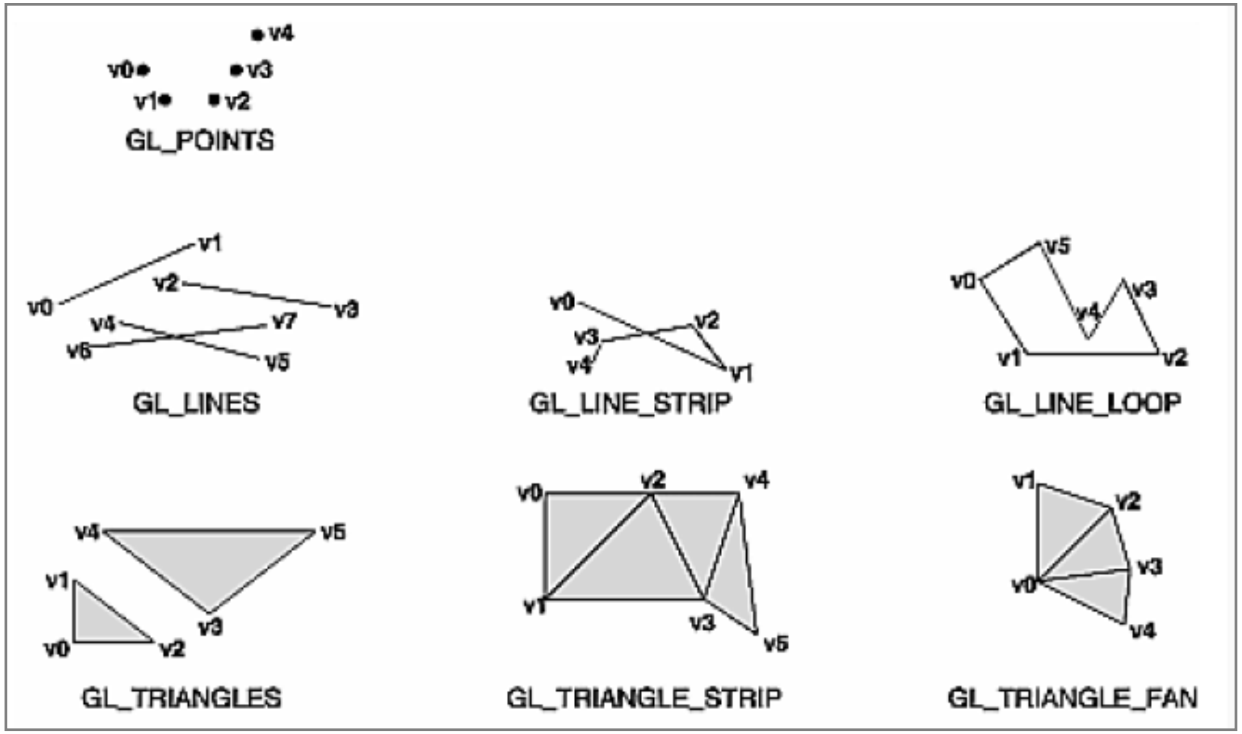
Lecture 3

Graphics Primitives & Hierarchical Modeling

Lecture outline:

1. Graphics Primitives: Points, Lines, and Triangles
2. Data structure: vertex list and index list
3. Hierarchical structure
4. View-world or Modelview transformations
5. Basic scenegraph concept

Graphics Primitives:

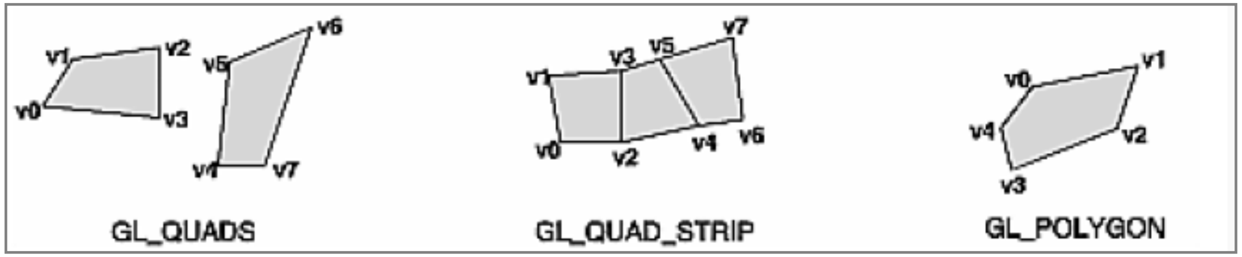


It’s all about

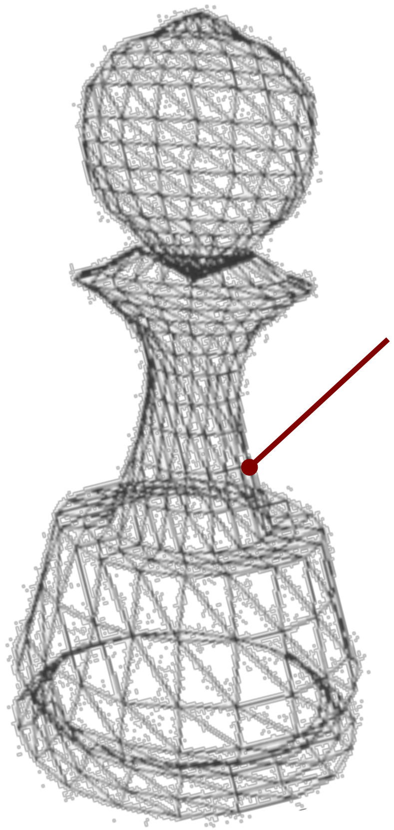
coordinates

and ***connectivity***

Deprecated from OpenGL 3+



Graphics Primitives:



A 3D Mesh

(shown as a wireframe)

Break $\longrightarrow$ down

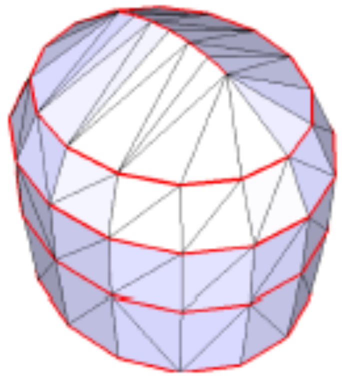
tessellation / triangulation

Points
OR Lines
OR Triangles (other)

* Why triangles? We will see this later in the scan conversion lecture

Graphics Primitives:

Example: four tri. strips



Question: which input type is the most

efficient one for rendering polygons?

- Answer: *Triangle strip (preferred)* or *Triangle fan*

- Because more polygons with fewer input vertices and no tessellation needed

- Some software to break down a 3D mesh into triangle strips for efficient rendering, e.g. tri stripper, NVtristrip (NVidia), or OpenGL optimizer (SGI)

- See http://www.plunk.org/~grantham/public/meshifier/oldmesh.html
  http://www.nvidia.com/object/nvtristrip_library.html

Data structure

(1) Vertex set (storing one array with vertex coordinates)

e.g. float coords[N][3];

Vertex List

Vertex #	Coordinates
1	x1,y,1z1
2	x2,y2,z2
3	x3,y3,z3
4	x4,y4,z4
5	x5,y5,z5
6	x6,y6,z6
7	x7,y7,z7
8	x8,y8,z8

Triangle number 1

Triangle number 2

Good: sequential memory access

Bad: it is very likely to have duplicated vertices in the list

Data structure

(2) Indexed Face Set (one more array for indices)

Vertex List

Vertex #	Coordinates
1	x1,y,1z1
2	x2,y2,z2
3	x3,y3,z3
4	x4,y4,z4
5	x5,y5,z5
6	x6,y6,z6
7	x7,y7,z7
8	x8,y8,z8

⋮

Face List

a mesh	
Polygon #	Vertex List
1	V1,V2,V3
2	V1,V4,V8
3	V5,V6,V2
4	V2,V6,V7
5	V4,V3,V8
6	V5,V8,V7

⋮

e.g. float coords[N][3];

int tri_index[N][3];

Need delimitors (-1) or an additional field for Vertex count if number of vertices per polygon is flexible

Bad: random memory access (may have cache miss)

Good: reuse vertices and keeps a compact Vertex list

Data structure

Note:

1. Triangle strip can be implemented on either data structure
2. Some high level APIs like VRML, Inventor, etc. have these data structure built-in and we can input the two lists directly
3. The Vertex ID of the first vertex in the vertex list may starts from 0 or 1, depending on which API / file format: e.g. VRML starts from 0 and obj (3D file format) starts from 1

Check: http://paulbourke.net/dataformats/

Data structure

Example #1: Indexed Line Set in VRML:

```
geometry IndexedLineSet {
        coord Coordinate {
           point [
                    0.00  0.00  0.00,   1.00  0.00  0.00,
                    0.92  0.38  0.00,   0.71  0.71  0.00,
                    0.38  0.92  0.00,   0.00  1.00  0.00,
                   -0.38  0.92  0.00,  -0.71  0.71  0.00,
                   -0.92  0.38  0.00,  -1.00  0.00  0.00,
                   -0.92 -0.38  0.00,  -0.71 -0.71  0.00,
                   -0.38 -0.92  0.00,   0.00 -1.00  0.00,
                    0.38 -0.92  0.00,   0.71 -0.71  0.00,
                    0.92 -0.38  0.00,
           ]
        }
        coordIndex [
           0,  1, -1,   0,  2, -1,
           0,  3, -1,   0,  4, -1,
           0,  5, -1,   0,  6, -1,
           0,  7, -1,   0,  8, -1,
           0,  9, -1,   0, 10, -1,
           0, 11, -1,   0, 12, -1,
           0, 13, -1,   0, 14, -1,
           0, 15, -1,   0, 16, -1
        ]
}
```

Vertex list

Index list

From http://www.sdsc.edu/~nadeau/Talks/NASA_EOSDIS/syntax10.htm

Data structure

Example #2: a very simple quad in *obj* file format (common and simple)

```
mttlib quad.mtl
usemtl quad

# Vertices
v -1.0 -1.0 0.0
v  1.0 -1.0 0.0
v  1.0  1.0 0.0
v -1.0  1.0 0.0

# Normals
vn 0.0 0.0 1.0

# Texture Coordinates
vt 0.000000 0.000000
vt 1.000000 0.000000
vt 1.000000 1.000000
vt 0.000000 1.000000

# Faces (Vertex/Texture/Normal)
g 1
f 1/1/1 2/2/1 3/3/1 4/4/1
```

Vertex list

for lighting and texture

(you will learn in future lectures)

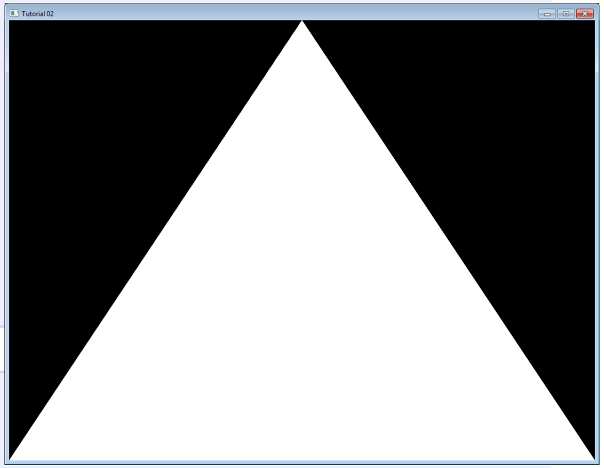
a group

Index list

Data structure

Example #3: Vertex Array Object in OpenGL 3+

```
// An array of 3 vectors which represents 3 vertices
static const GLfloat g_vertex_buffer_data[] = {
   -1.0f, -1.0f, 0.0f,
   1.0f, -1.0f, 0.0f,
   0.0f,  1.0f, 0.0f,
};
```

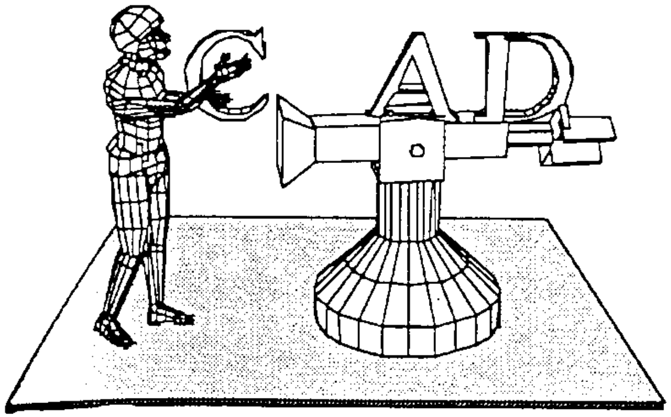


```
// This will identify our vertex buffer
GLuint vertexbuffer;
// Generate 1 buffer, put the resulting identifier in vertexbuffer
glGenBuffers(1, &vertexbuffer);
// The following commands will talk about our 'vertexbuffer' buffer
glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
// Give our vertices to OpenGL.
glBufferData(GL_ARRAY_BUFFER, sizeof(g_vertex_buffer_data), g_vertex_buffer_data, GL_STATIC_DRAW);
```

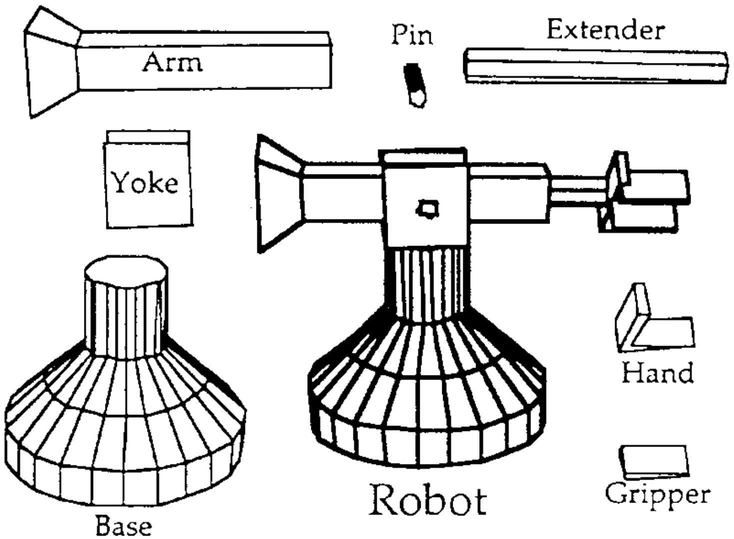
Note: you may also use indexed mode to construct geometry

http://www.opengl-tutorial.org/beginners-tutorials/tutorial-2-the-first-triangle/

Hierarchical Model



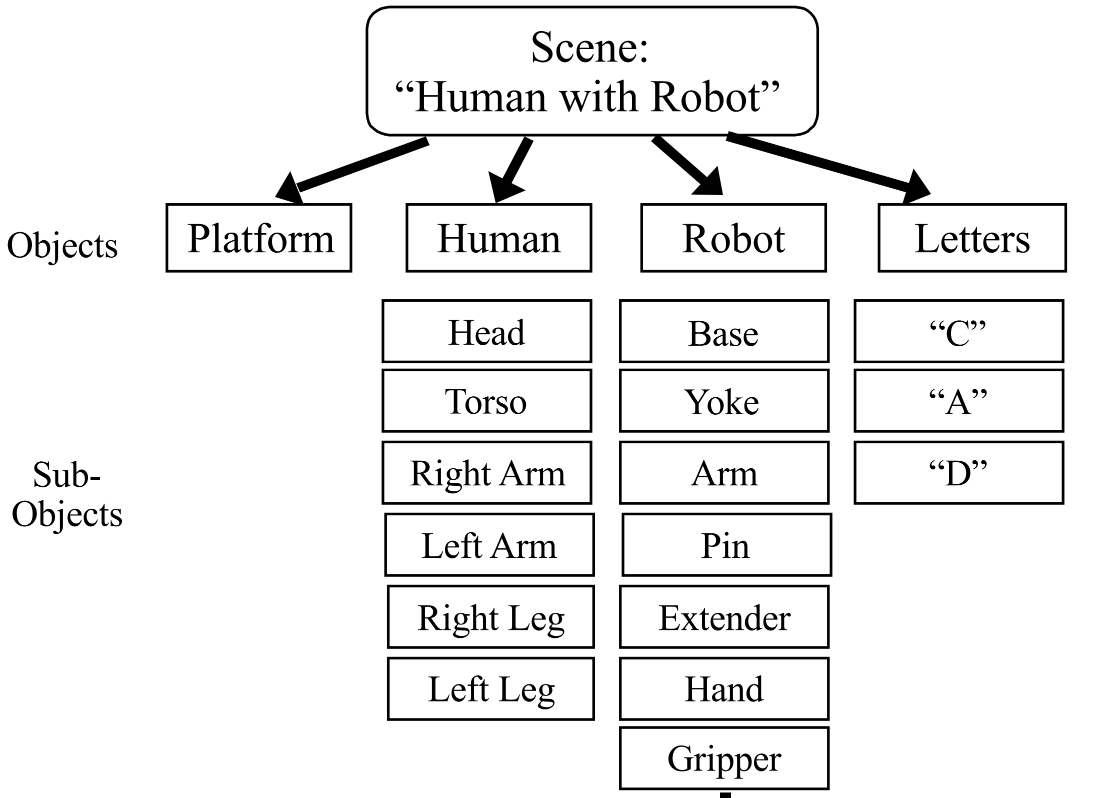
Human with Robot scene based on polyhedra. Note how the whole scene is composed of rectangles, trapezoids, or, in the case of the 3D letters, rectangles and n-sided polygons



Exploded view of hierarchical structure of robot. The robot main object (bold) is constructed by assembling graphical primitive subobjects which are easily generated by CAD systems

Hierarchical Model

Hierarchical structure of a polyhedral scene. Note that each subobject will have its own polygon list and associated vertex list. Also, subobjects such as right arm will have its own subobjects such as upper arm, lower arm, and hand. The hand may, in turn, have subobjects such as a fingers, and so on.



Data Structure

Vertex List

Vertex #	Coordinates
1	x1,y,1z1
2	x2,y2,z2
3	x3,y3,z3
4	x4,y4,z4
5	x5,y5,z5
6	x6,y6,z6
7	x7,y7,z7
8	x8,y8,z8

Polygon List

Gripper	
Polygon #	Vertex List
1	V1,V2,V3,V4
2	V1,V4,V8,V5
3	V5,V6,V2,V1
4	V2,V6,V7,V3
5	V4,V3,V7,V8
6	V5,V8,V7,V6

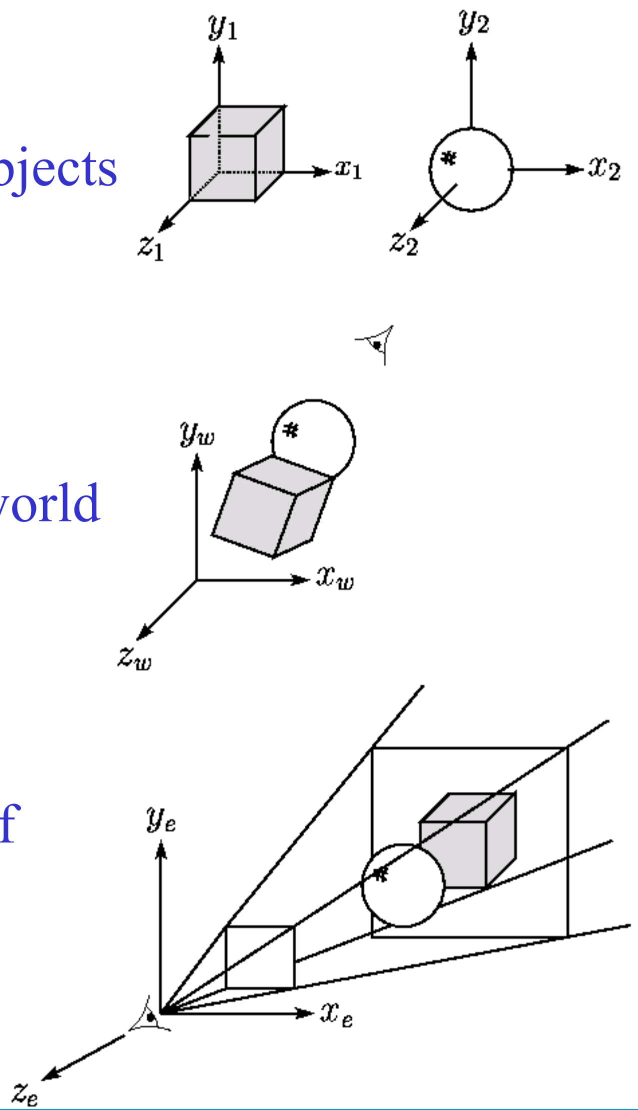
General Viewworld Transformation (2 Matrices)

Different objects

Common world

In front of

the Eye



Model space
(Object space)

↓ $M_{obj2world}$

World space
(Object space)

↓ $M_{world2eye}$

Eye space
(View space)

↓

……

Given object coordinate $P_{obj} = (x,y,z)^T$

World coord. $= M_{obj2world} \times P_{obj}$

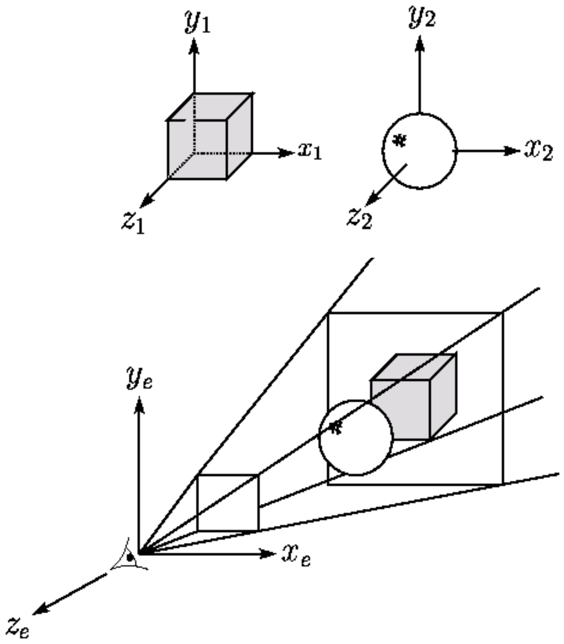
Eye coord. $=$ $M_{world2eye} \times M_{obj2world} \times P_{obj}$

OpenGL Modelview Transformation (1 Matrix)

$M_{obj2world}$ and $M_{world2eye}$ are merged (just matrix composition) into one matrix, called the modelview (or viewworld) matrix "$M_{modelview}$"

An object

In front of the Eye



Model space
(Object space)

↓ $M_{modelview}$

Eye space
(View space)

↓

……

Given object coordinate $P_{obj} = (x,y,z)^T$

Eye coord. = $M_{modelview}$ x P_{obj}

Note: OpenGL puts the eye-point at the origin looking towards negative z-axis

OpenGL Modelview Transformation (1 Matrix)

About the modelview matrix "$M_{modelview}$"

Important Note:

1. OpenGL puts the eye-point at the origin looking towards negative z-axis
2. OpenGL is a state machine: it has a internal memory storage (4 x 4 floating point numbers) for the modelview matrix
3. When calling translate / rotate / …, the kernel will first construct a matrix for the T/R/S and right multiply it with its internal modelview matrix

OpenGL Modelview Transformation (1 Matrix)

Illustration:

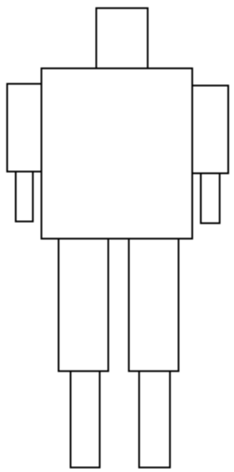
	Memory in Graphics hardware	OGL Commands	Kernel Operations	Comment
Step 0	$M_{modelview}$			Initial value: an identity
Step 1		translate	Construct M_{tran}	Construct a matrix for T
Step 2			$M_{modelview} x M_{tran}$	Right multiply M_{tran} on $M_{modelview}$
Step 3	$M_{modelview} x M_{tran}$			Store the result

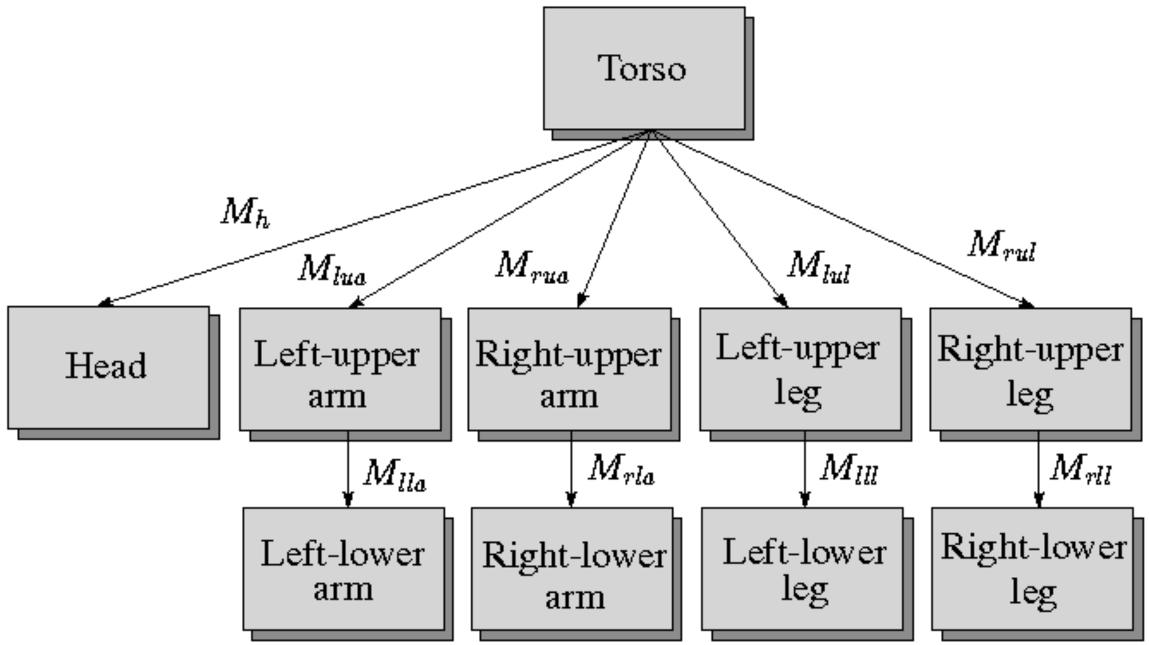
Note:

1. OGL always uses right-multiplication whereas DirectX is flexible (?)

OpenGL Modelview Transformation

A more complex example: Human figure





Human figure

Q: What's the most sensible way to traverse this tree?

Basic Scenegraph Concept

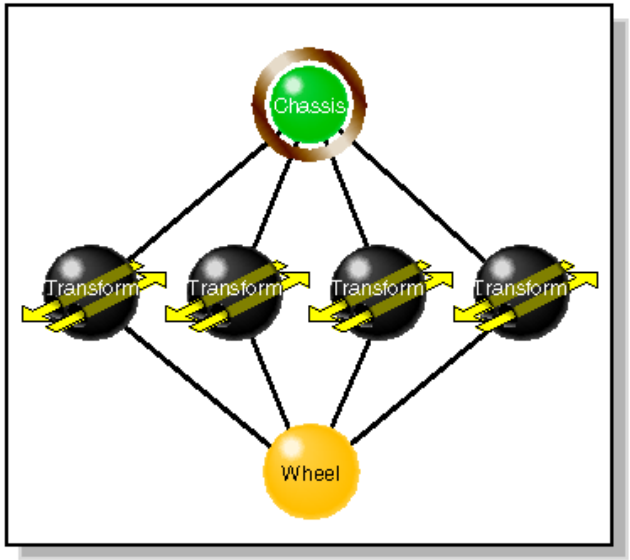
- Organize the whole model hierarchy (the geometry as well as the lighting, material, camera, etc.) as a tree structure
- Examples (API/Language): VRML, OpenSG (open scenegraph), etc.

For example:

Two most general classifications of node functionality are:

Group nodes - associate nodes into hierarchies.

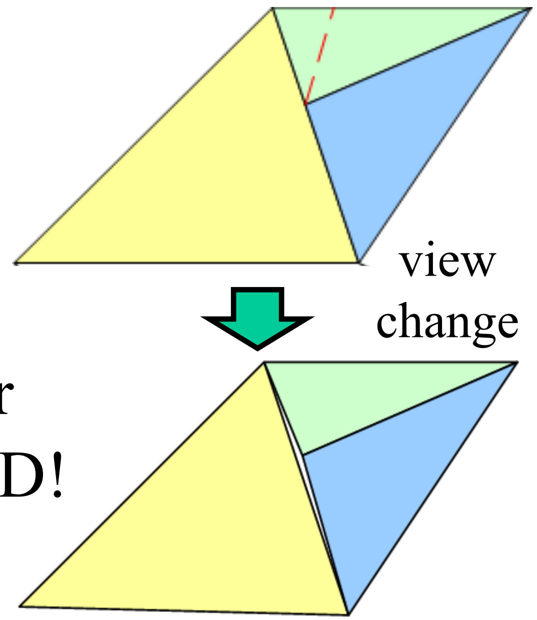
Leaf nodes - contain all the descriptive data of objects in the virtual world used to render them.



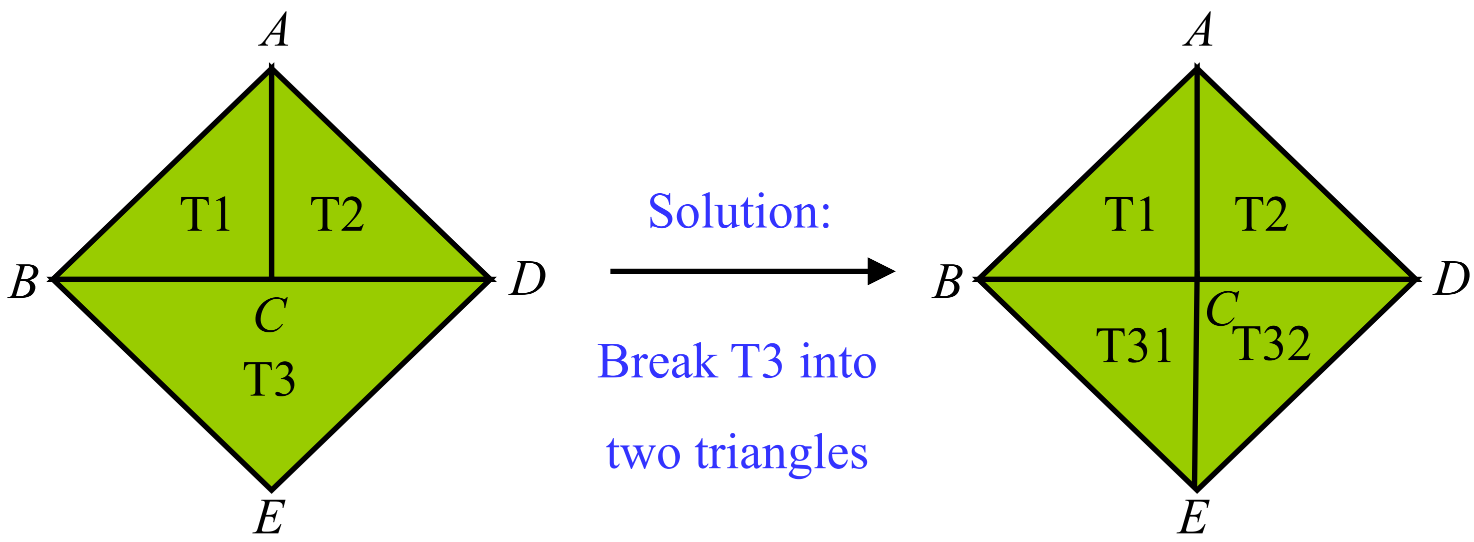
* Reuse the wheel geometry

https://en.wikipedia.org/wiki/Scene_graph

Avoid Modeling Glitches

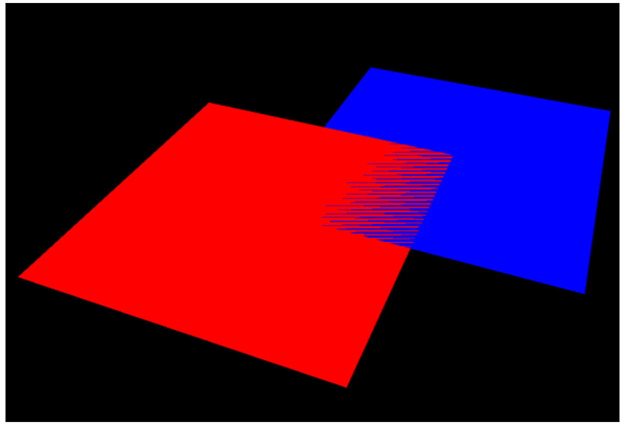


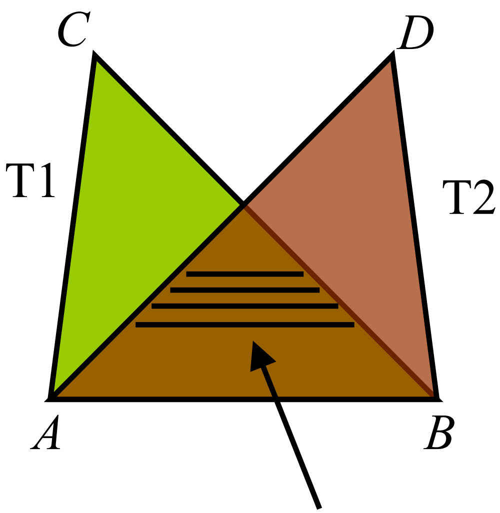
1. Avoid T-join in your models
   - E.g., triangles T1(ABC), T2(ACD), and T3(BED)
   - Edge BD of T3 may not exactly touch point C after transformations, so you may see a hole at edge ABD!
   - Reason: computation with float may not be exact



Avoid Modeling Glitches

2. Avoid overlapping polygons in your models
   - E.g., overlap triangles T1(ABC) and T2(ABD)
   - If colors of T1 and T2 are different, you may see flipping colors in the overlap (z fighting) area due to the numeric computation.

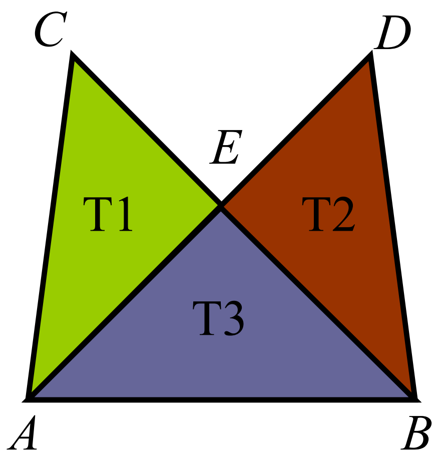




Colors flips between R and G

Solution:

One more triangle for the overlapping area



Another trick:

Turn off Depth test when you draw the next Triangle

(See hidden surface removal lecture)