Introduction to Graphics

Lecture 5:

Review & Introduction to OpenGL

Surgical
Robotics &
Imaging

Imperial College London

Lecture Overview

- Review:
  - Projections
  - Transformations
- Introduction to OpenGL
- Software Tools:
  - *JPot* – OpenGL Tutorial
  - JOGL – Java bindings for OpenGL

Projections

- n-dimensional vector space $\rightarrow$ m-dimensional vector space (m < n)
- 3D $\rightarrow$ 2D: select projection surface
  define projectors
  pass projectors through each vertex
  display intersections with projection surface
- Parallel projections - parallel projectors
  defined by direction of projectors
- Perspective projections – projectors pass through one single point
  defined by centre of projection

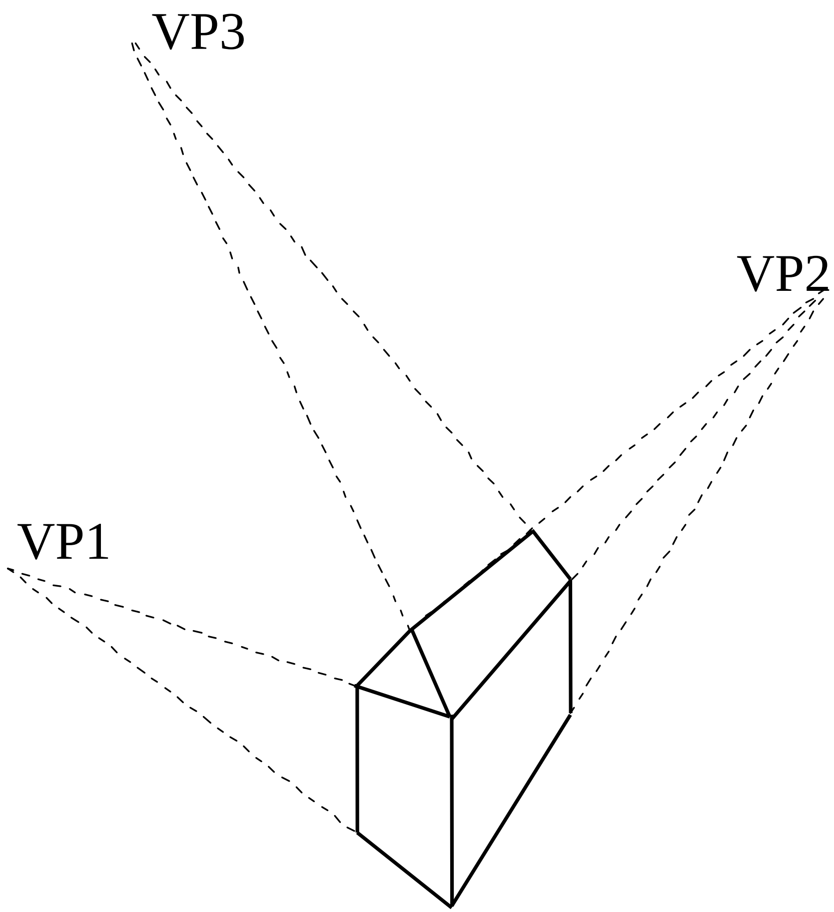
Perspective Projection

$Px = \mu_p Vx$ and $Py = \mu_p Vy$

$\mu_p = f/Vz$ - Foreshortening factor

- The further an object is (large Vz), the smaller μ_p and the smaller the object will appear
- Orientation of original image preserved if centre of projection behind f

- Lines that are parallel in 3D are **NOT** necessarily parallel in the 2D projection
- Images of parallel lines which are parallel to projection surface **WILL** remain parallel
- Others will meet at *vanishing points* (perspective projection of a point at infinity)

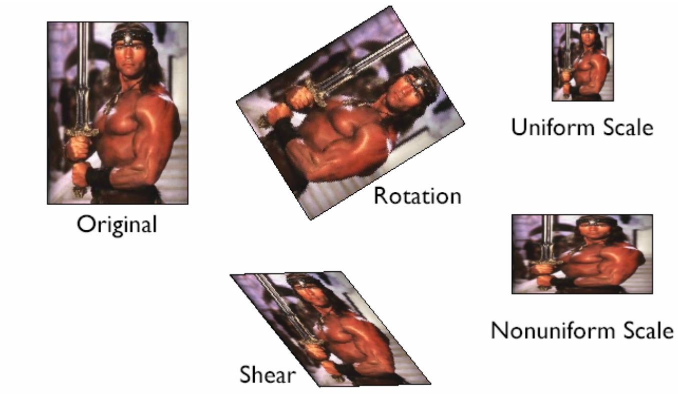

VP3
VP2
VP1

Transformations

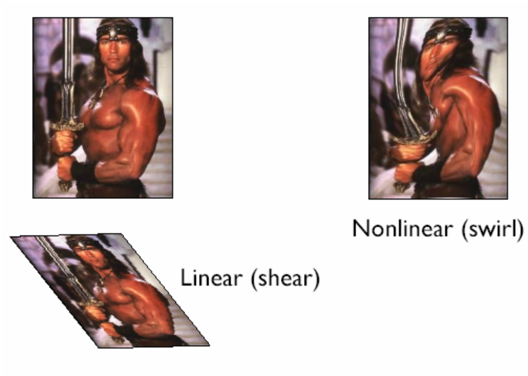
An operation that changes one configuration into another.

A geometric transformation maps positions that define the object to other positions.

Linear transformation means the transformation is defined by a linear function... which is what matrices are good for



Images from *Conan The Destroyer*, 1984



Transformation Composition

Complex transformations can be created by composing individual transformations together

Matrix multiplication is non-commutative $\Rightarrow$ order is vital!

$$A*B \neq B*A$$

Some special cases work, but they are EXCEPTIONS

Matrices are associative

$$(A*B)*C = A*(B*C)$$

What commutes?

Two **translations** commute $T_1 * T_2 = T_2 * T_1$

Two **scales** commute $S_1 * S_2 = S_2 * S_1$

Two **rotations** *sometimes* commute. In 2D rotations do commute, while in 3D most pairs of rotations do not commute.

Rotations and **translations** do *not* commute $R * T \neq T * R$

Translations and **scales** do *not* commute $S * T \neq T * S$

Scales and **rotations** commute only in the *special case* when scaling by the same amount in all directions.

In general the two operations do not commute.

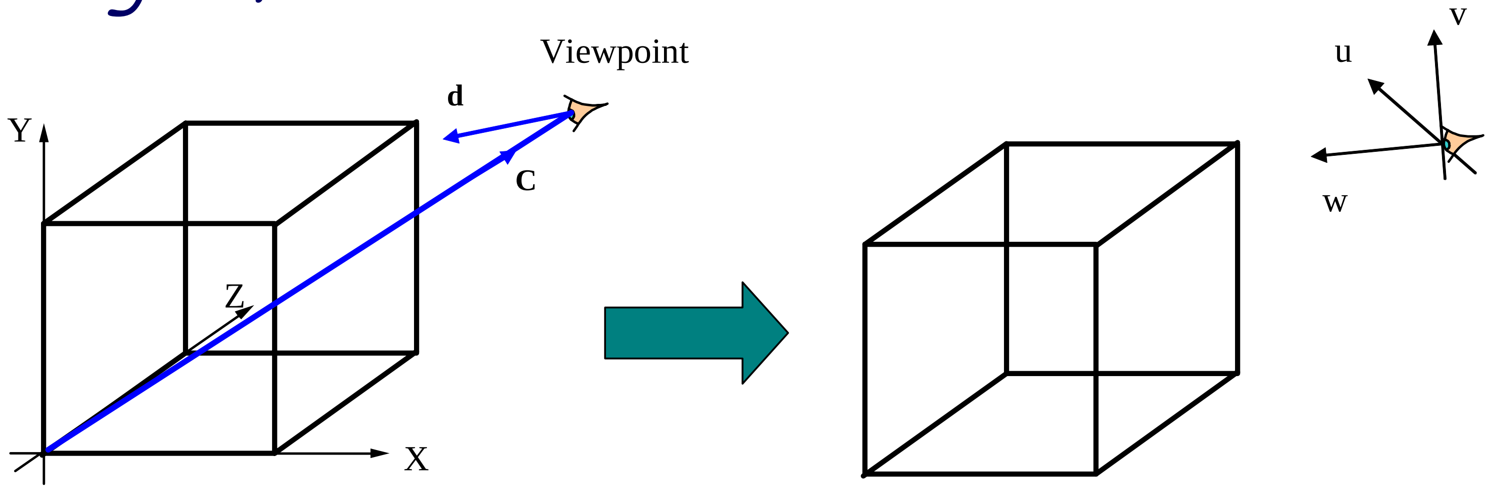
Applications

Transformations are used for a variety of purposes, e.g:

- Changing viewpoint / Change of axes
- Special effects
- Image Registration
- Dealing with hierarchical structures

Application Examples

Change of Axes



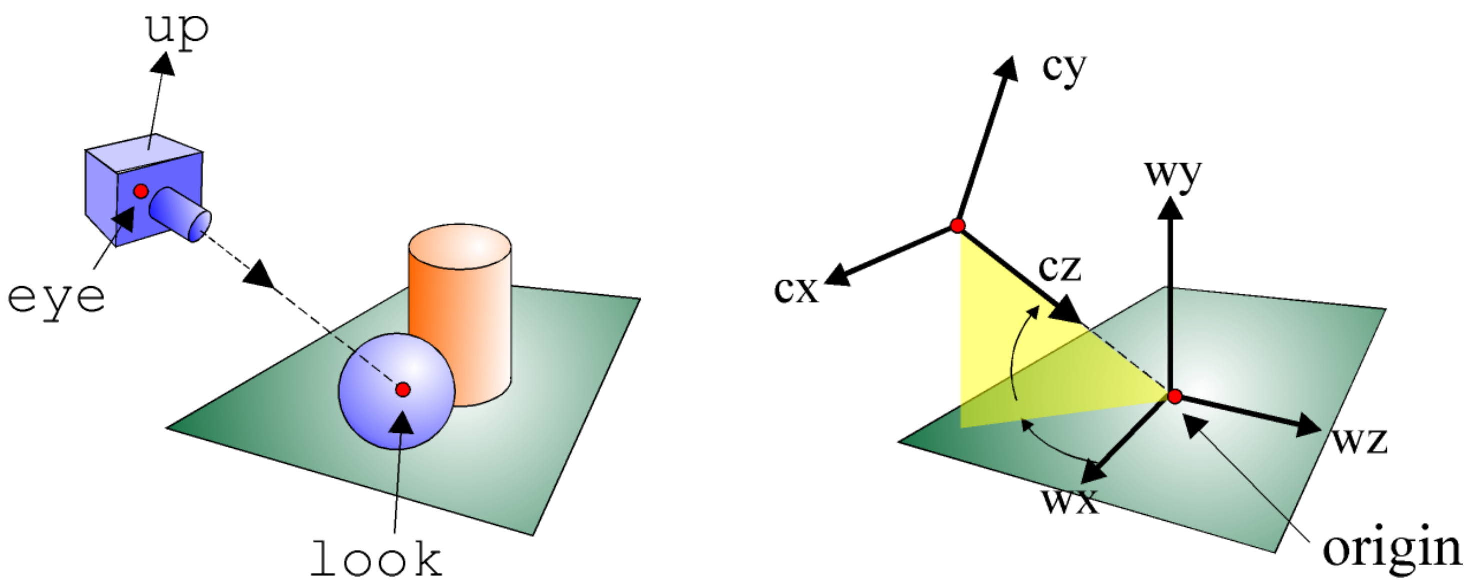
Coordinate System for definition

Coordinate System for viewing

Given [u,v,w] and C, find the transformation matrix that moves the scene to that coordinate system.

Why? What for?

Changing Viewpoint




PYROS PICTURES

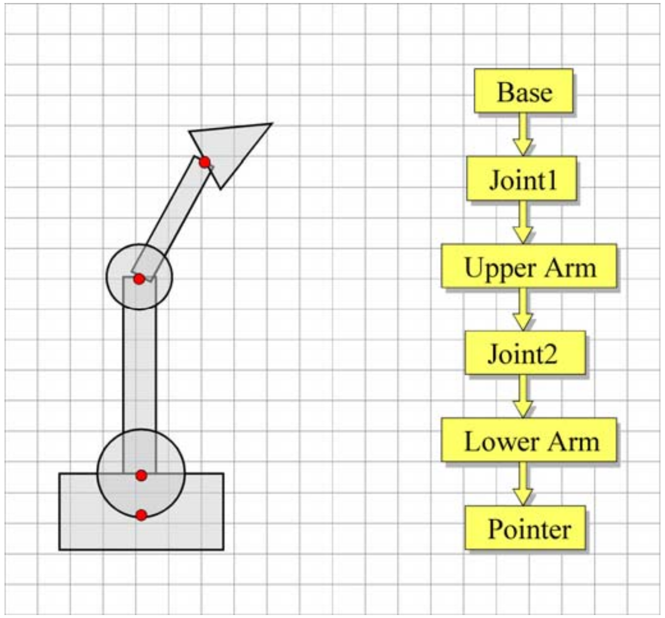
Hierarchical Transformations

For geometries with an implicit *hierarchy* we wish to associate local frames with sub-objects in the assembly.

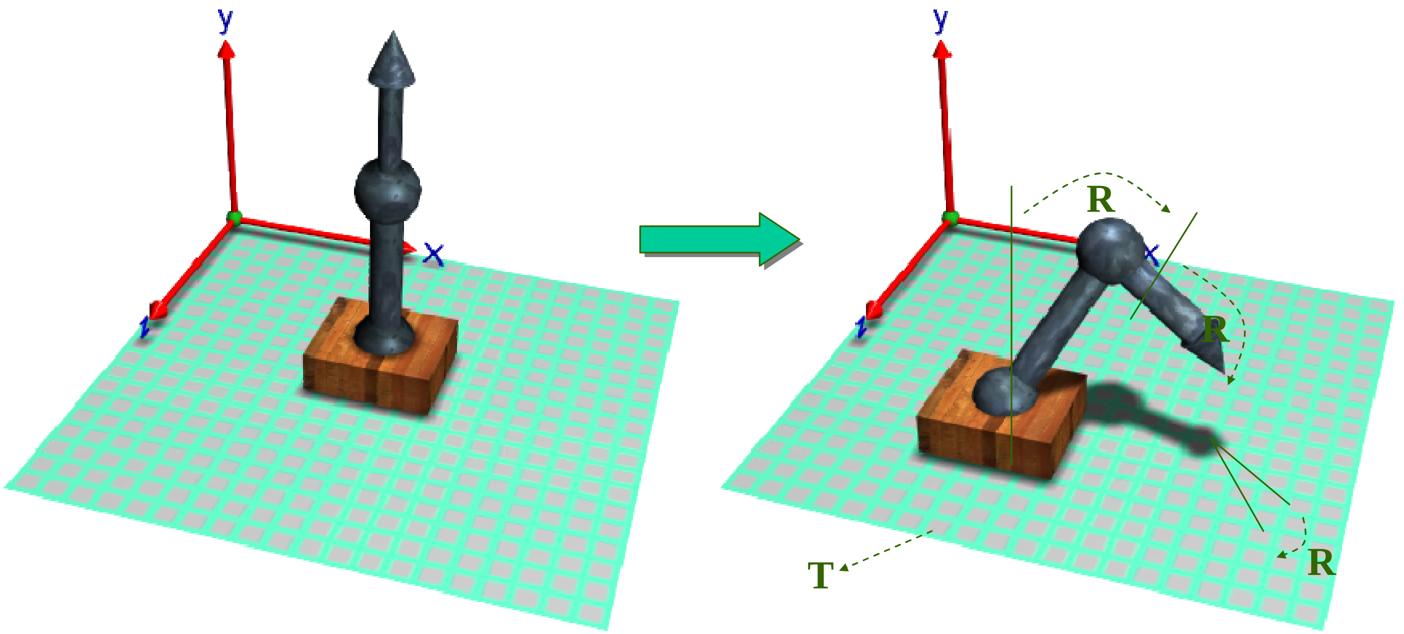
Parent-child frames are related via a transformation.

Transformation linkage is described by a *tree*:

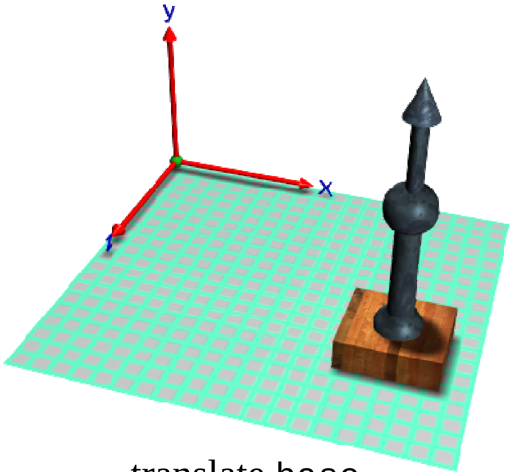
Each node has its own *local co-ordinate system*.



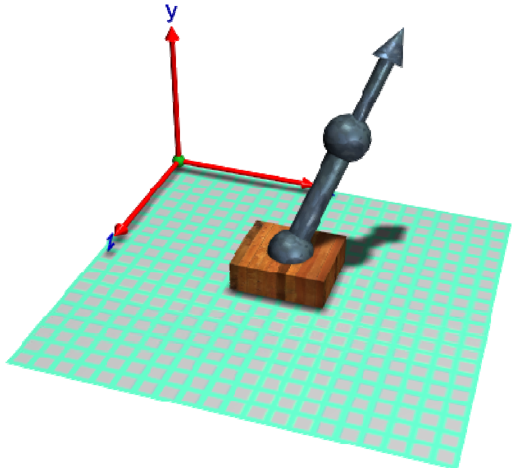
Hierarchical Transformations



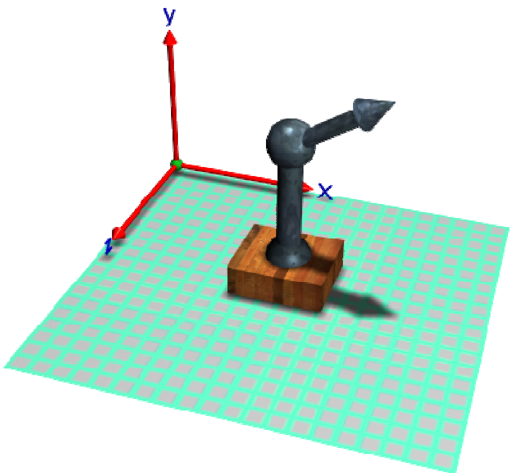
Hierarchical transformation allow independent control over sub-parts of an assembly



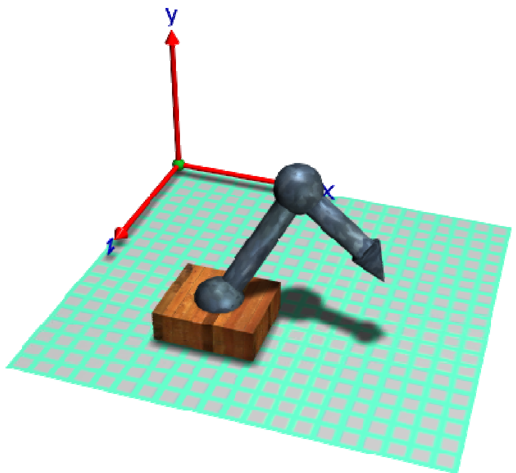
translate `base`



rotate `joint1`



rotate `joint2`



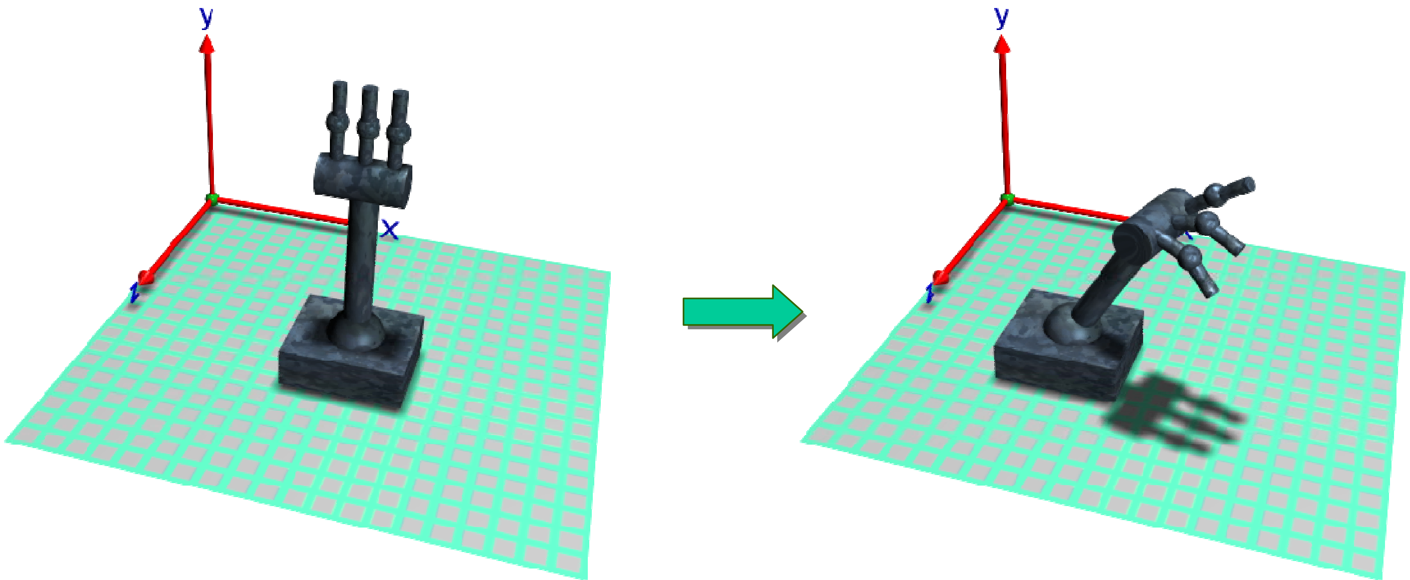
Complex hierarchical transformation

Hierarchical Transformations

The previous example had simple *one-to-one* parent-child linkages.

In general there may be many *child frames* derived from a single parent frame.

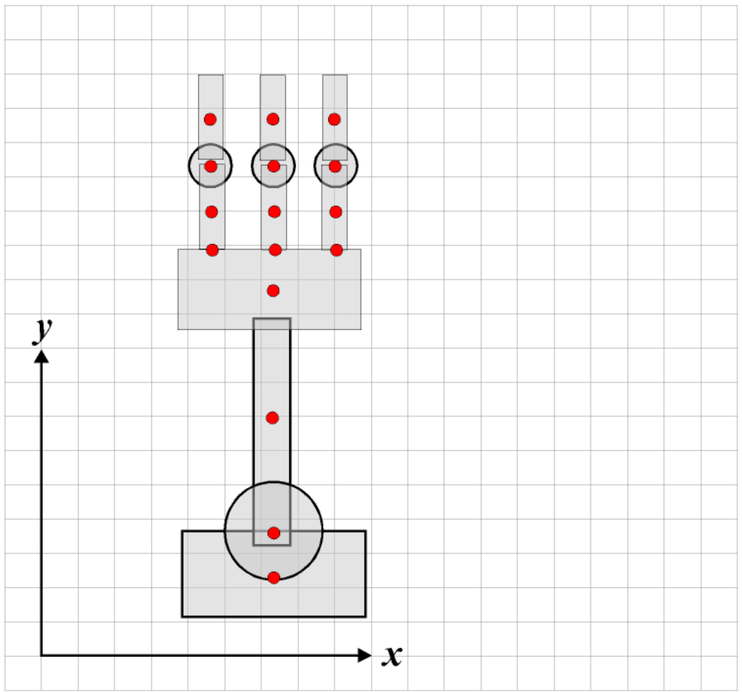
Hierarchical Transformations

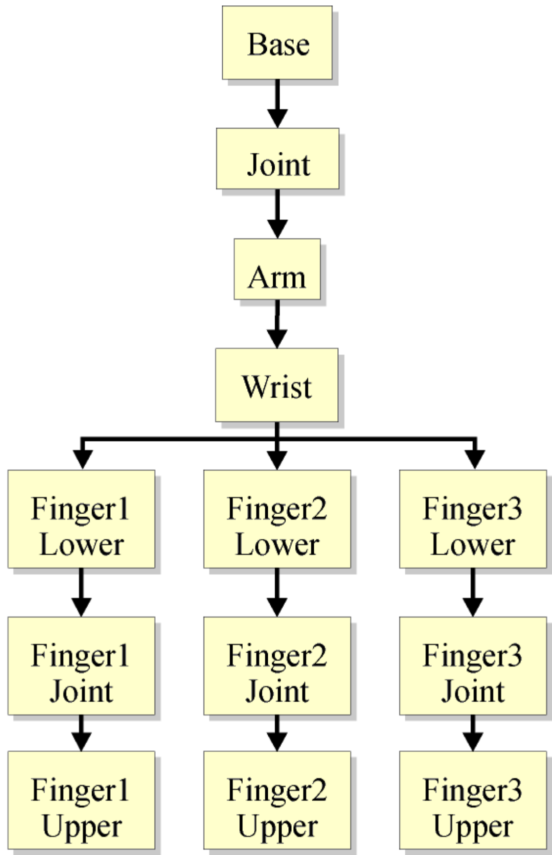


Each finger is a child of the parent (wrist)

⇒ independent control over the orientation of the fingers relative to the wrist

Hierarchical Transformations





Introduction to OpenGL

- What is OpenGL?

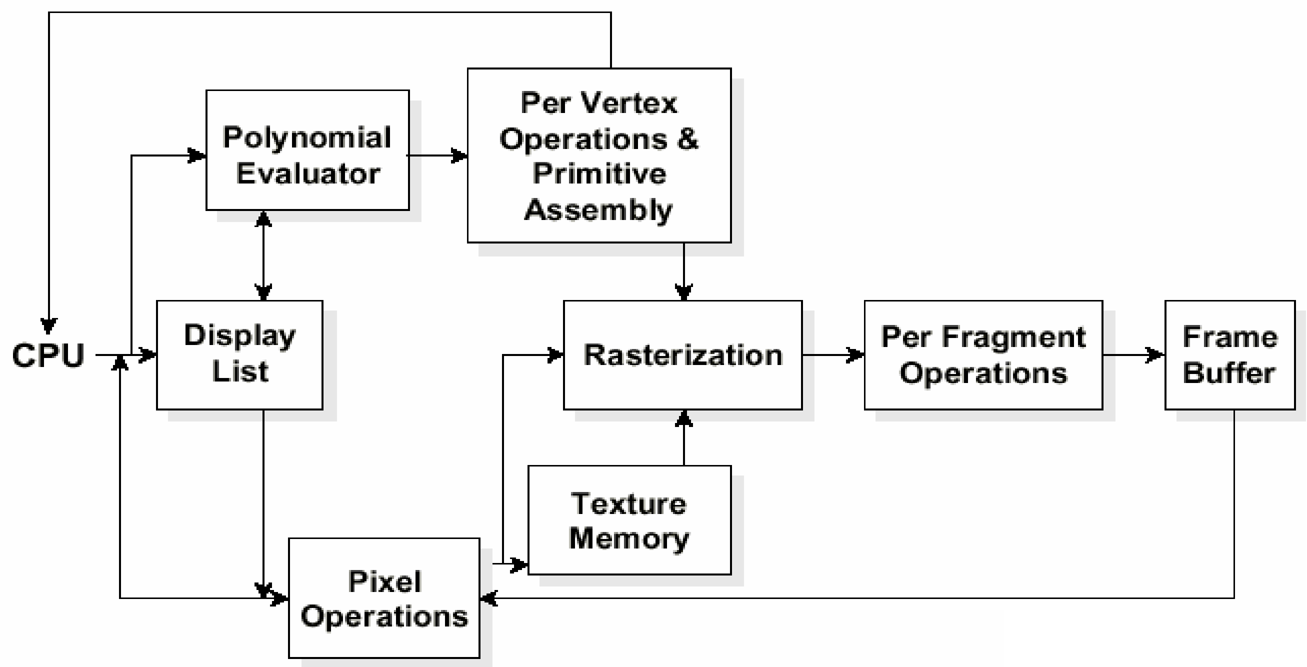
  Graphics rendering API

  ✓high-quality color images composed of geometric and image primitives

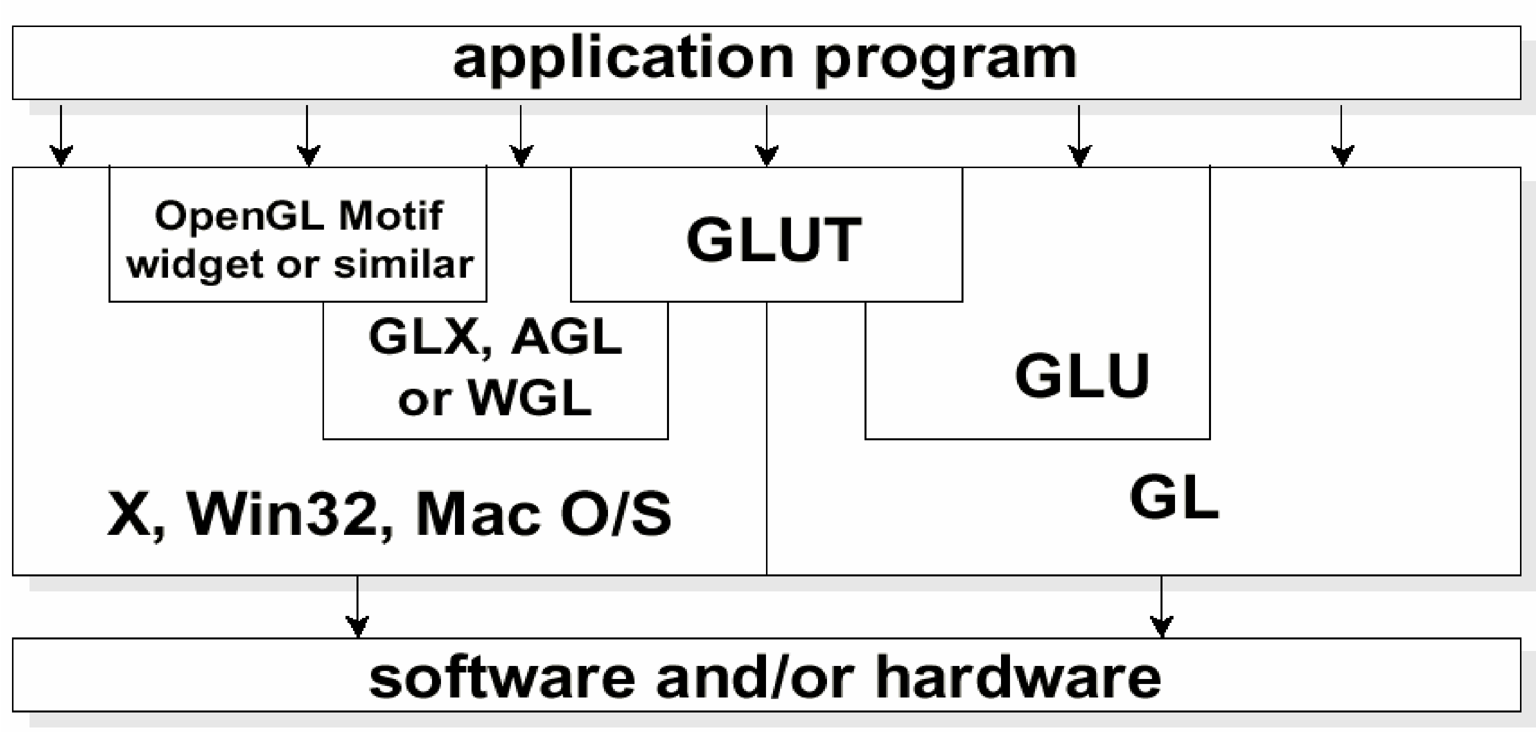
  ✓window system independent

  ✓operating system independent

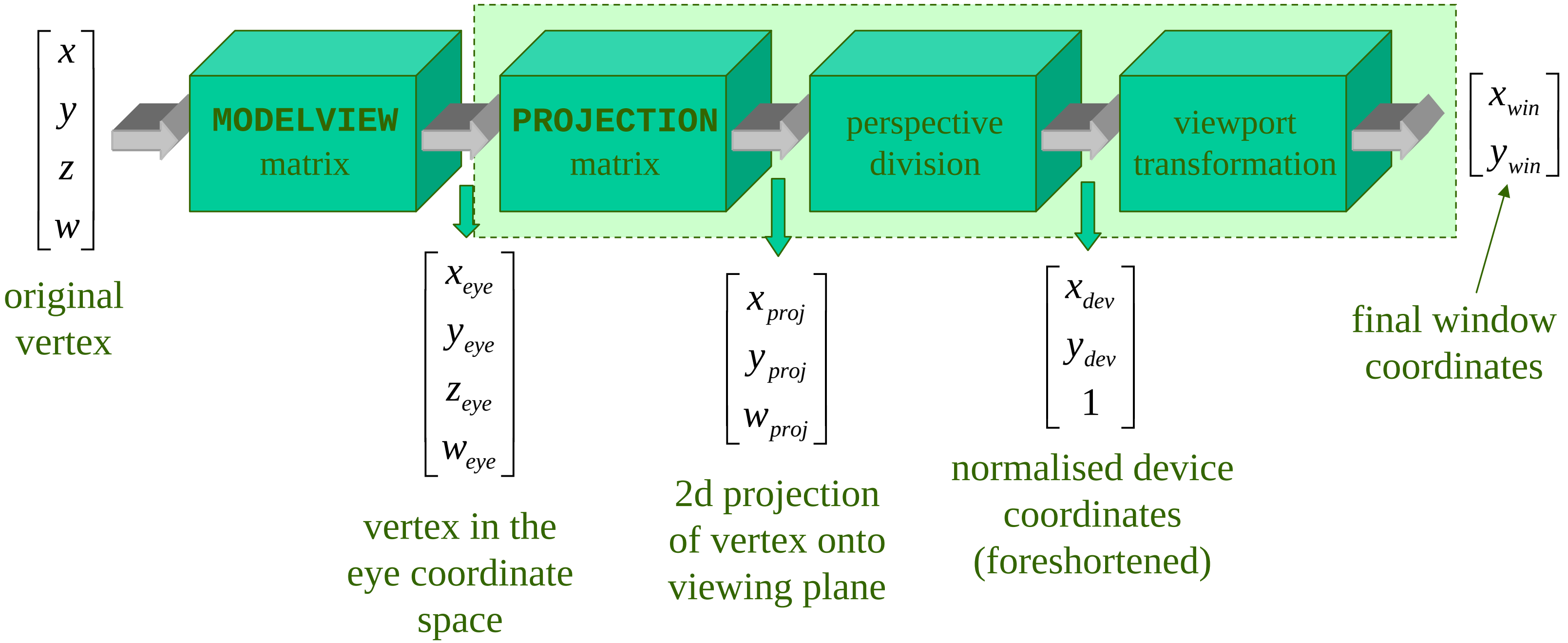
OpenGL Architecture



OpenGL and Related APIs



OpenGL® Geometry Pipeline



The Camera System

To create a view of a scene we need:

- a description of the scene geometry
- a camera or view definition

Default OpenGL camera is located at the origin looking down the ***-z*** axis.

The camera definition allows *projection* of the 3D scene geometry onto a 2D surface for display.

This projection can take a number of forms:

- *orthographic* (parallel lines preserved)
- *perspective* (foreshortening): *1-point*, *2-point* or *3-point*
- *skewed orthographic*

Camera Types

Before generating an image we must choose our viewer:

The ***pinhole camera model*** is most widely used:

- infinite *depth of field* (everything is in focus)

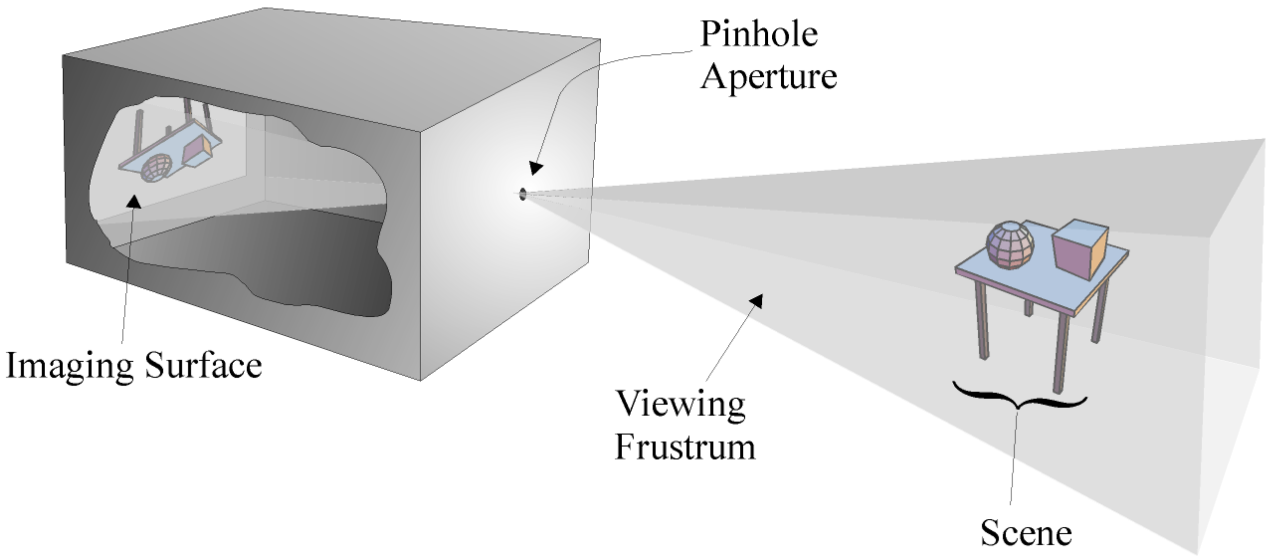
Advanced rendering systems model the camera

- *double gauss lens* as used in many professional cameras
- model depth of field and non-linear optics (including *lens flare*)

Photorealistic rendering systems often employ a physical model of the eye for rendering images

- model the eyes response to varying *brightness* and *colour* levels
- model the internal optics of the eye itself (*diffraction* by lens fibres etc.)

Pinhole Camera Model

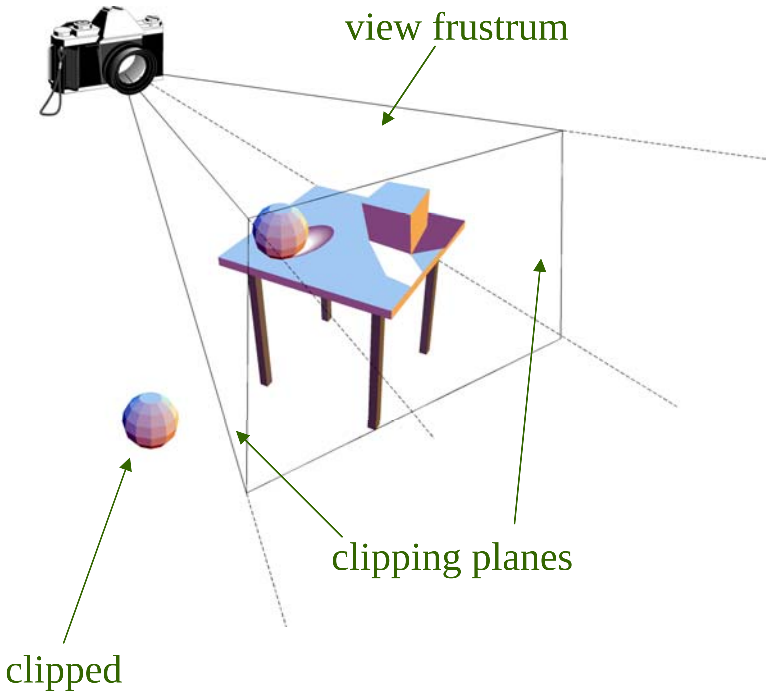


Viewing System

We are only concerned with the *geometry* of viewing at this stage.

The camera's position and orientation define a *view-volume* or *view-frustrum*.

- objects completely or partially within this volume are potentially visible on the viewport.
- objects fully outside this volume cannot be seen ⇒ *clipped*



Camera Models

Each vertex in our model must be projected onto the 2D *camera viewport* plane in order to be display on the screen.

The *CTM* is employed to determine the location of each vertex in the camera coordinate system:

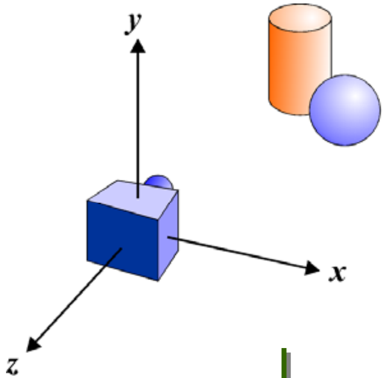
$$\vec{x}' = \mathbf{M}_{CTM}\,\vec{x}$$

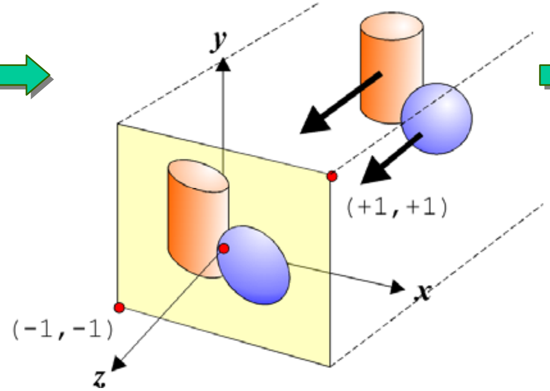
We then employ a projection matrix defined by `GL_PROJECTION` to map this to a 2D viewport coordinate.

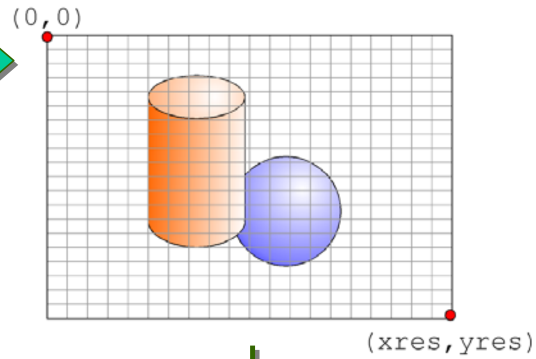
Finally, this 2D coordinate is mapped to device coordinates using the viewport definition (given by `glViewport()`).

Camera Modeling in OpenGL ®

camera coordinate system → viewport coordinate system → device/screen coordinate system








```
glMatrixMode(GL_MODELVIEW)
...
```

```
glViewport(0,0,xres,yres)
```

```
glMatrixMode(GL_PROJECTION)
...
```

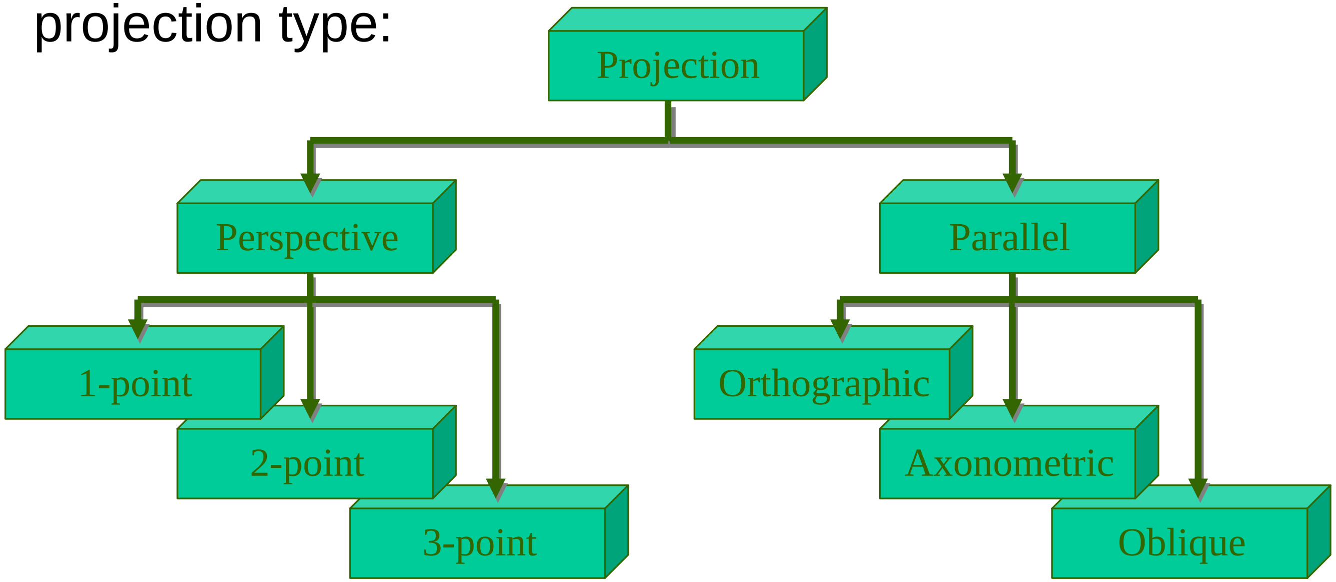
3D → 2D Projection

Type of projection depends on a number of factors:

location and *orientation* of the viewing plane (*viewport*)

direction of projection (described by a vector)

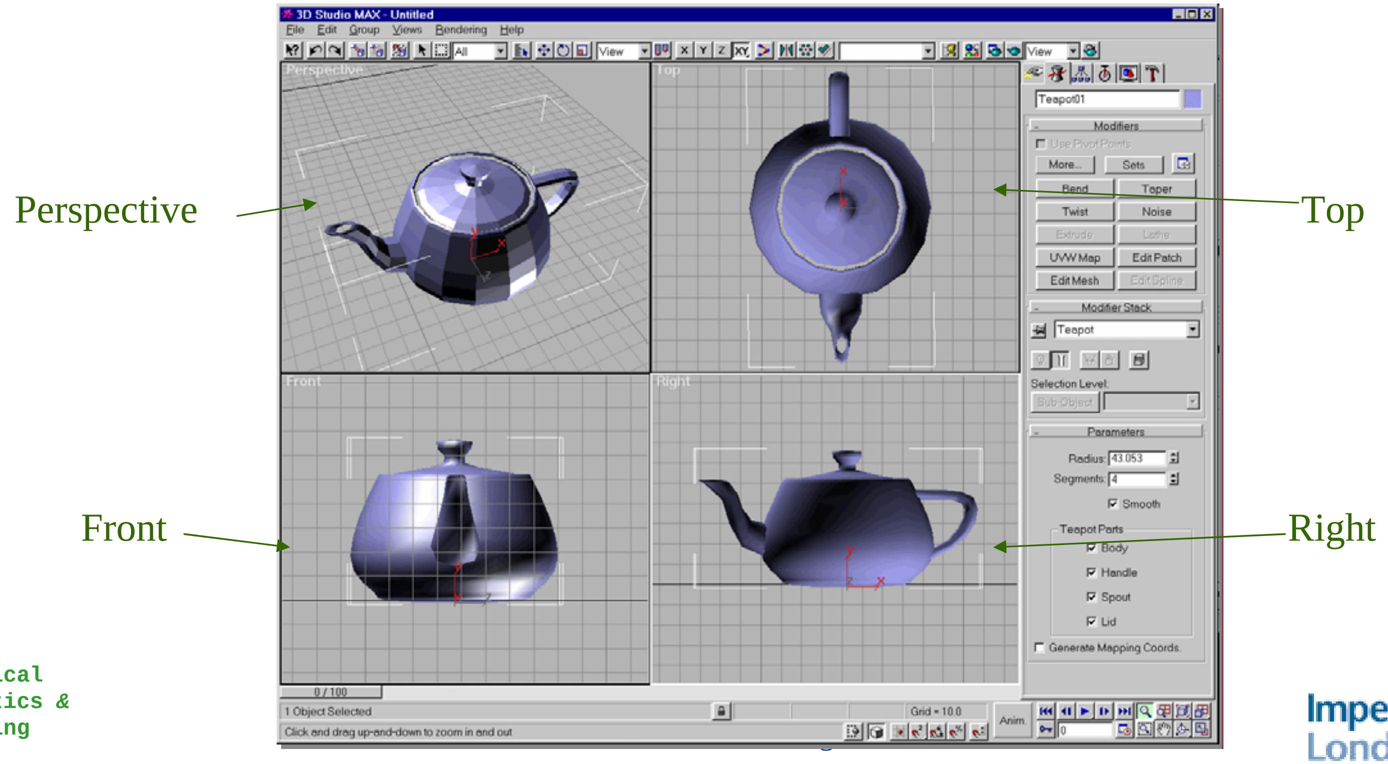
projection type:



Multiple Projections

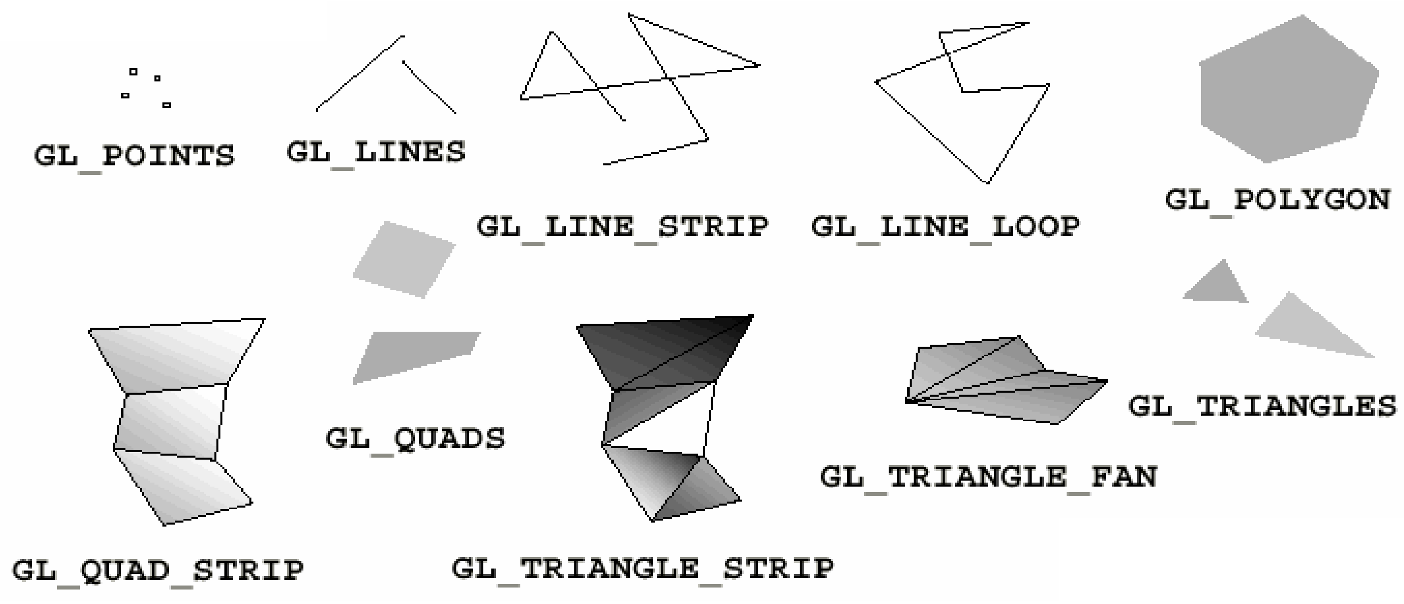
It is often useful to have *multiple projections* available at any given time

usually: plan (top) view, front & left or right elevation (side) view



OpenGL Geometric Primitives

All geometric primitives are specified by vertices



Specifying Geometric Primitives

Primitives are specified using

```
glBegin( primType );
glEnd();
```

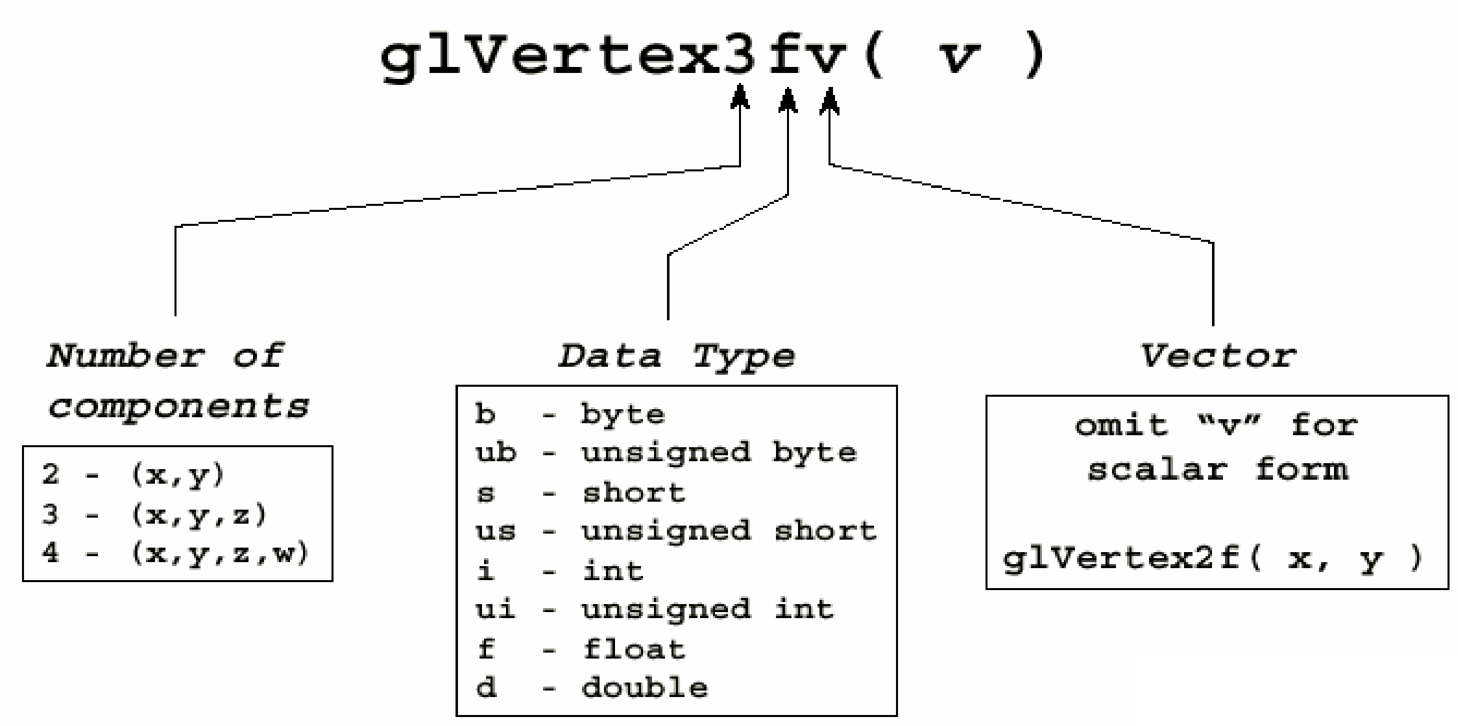
primType determines how vertices are combined

```
GLfloat red, greed, blue;
Glfloat coords[3];
glBegin( primType );
    for (i =0;i <nVerts; ++i ) {
       glColor3f( red, green, blue );
       glVertex3fv( coords );
    }
glEnd();
```

Simple Example

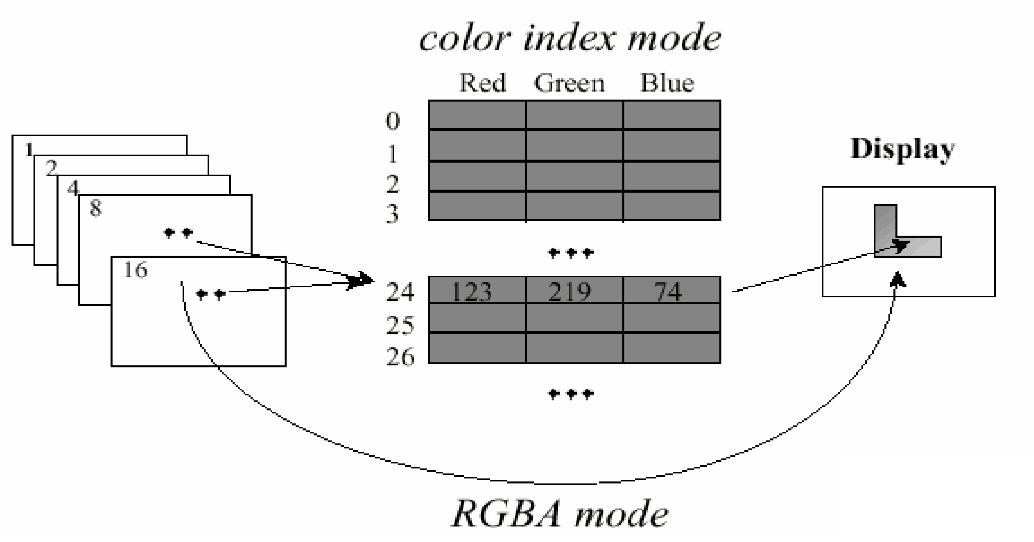
```
void drawRhombus( GLfloat color[] )
{
    glBegin( GL_QUADS );
        glColor3fv( color );
        glVertex2f( 0.0, 0.0 );
        glVertex2f( 1.0, 0.0 );
        glVertex2f( 1.5, 1.118 );
        glVertex2f( 0.5, 1.118 );
    glEnd();
}
```

OpenGL Command Formats



OpenGL Color Model

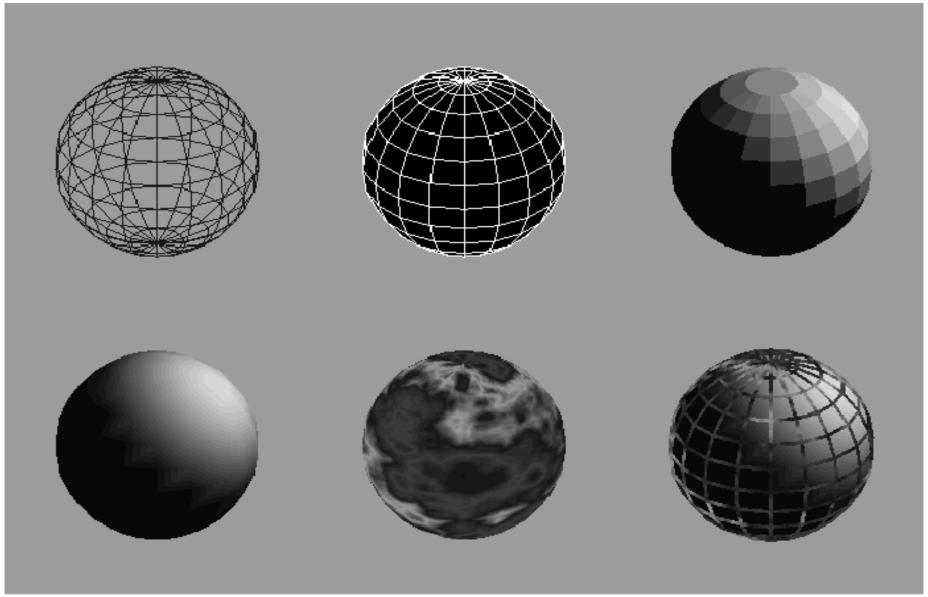
Both RGBA (true color) and Color Index



Controlling Rendering

Appearance

From Wireframe to Texture mapped



OpenGL State:

- rendering styles
- shading
- lighting
- texture mapping

–glColor*() / glIndex*()

–glNormal*()

–glTexCoord*()

Software Tools - JPot

- Java-based interactive OpenGL tutor

  http://www.cs.uwm.edu/~grafix2/

- DoC Linux workstations:

  Run "jpot-install" before using jpot for the first time
  Run JPot as usual:

  java –cp <JPot directory> JPot

- For Windows: Download and follow **ALL** instructions **CAREFULLY**

  (Requires JRE; glut32.dll and trigger.exe in C:\Windows\system32)

Software Tools – JOGL

- The JOGL project hosts the development version of the Java Bindings for OpenGL.
- Designed to provide hardware-supported 3D graphics to applications written in Java.
- JOGL provides full access to the APIs in the OpenGL 2.0 specification as well as nearly all vendor extensions.
- Already installed in DoC Linux Workstations.
- Download from https://jogl.dev.java.net/