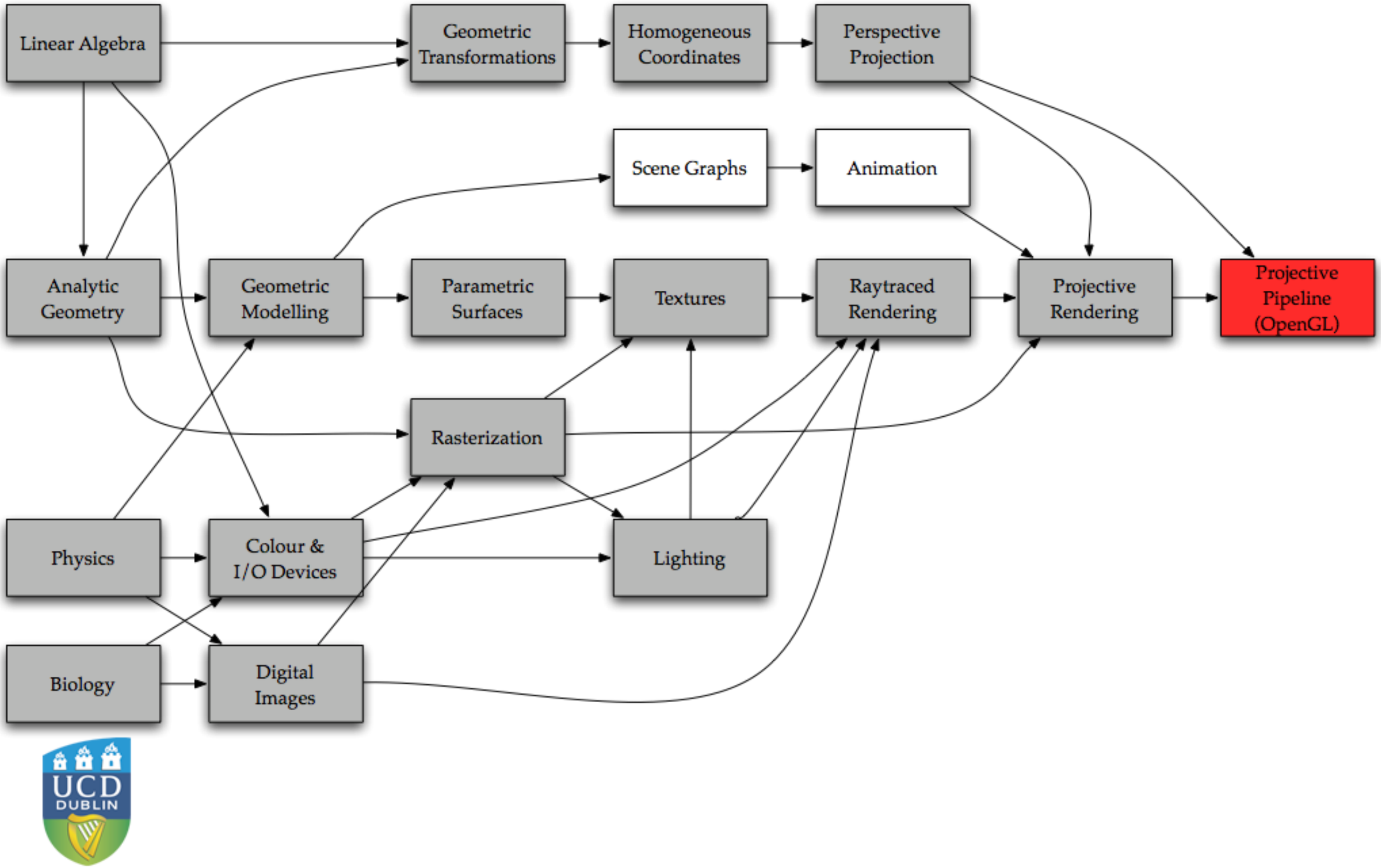
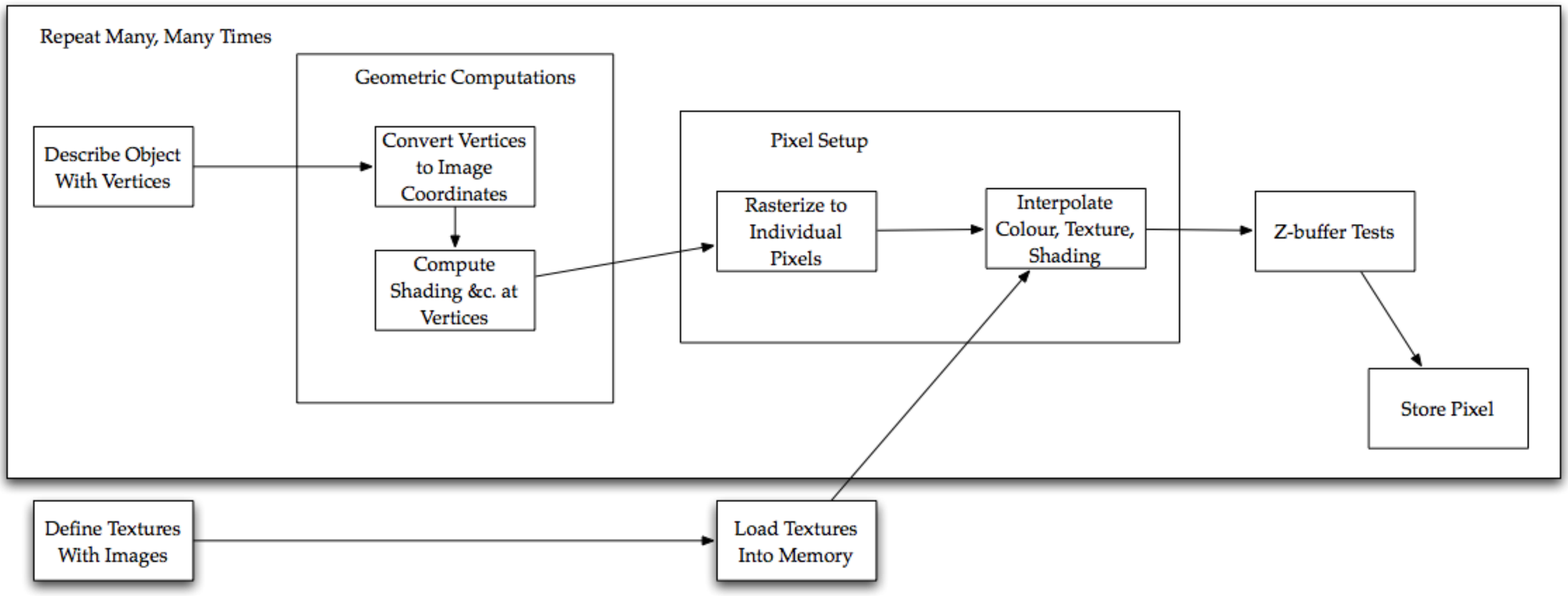
Open GL &
Projective Rendering

Where we Are



UCD DUBLIN

Projective Pipeline



I: Setup

- Define *model, view, projection* matrices
- Define *viewport* transformation
- Define *lights* & lighting properties
- Define *shading & lighting* model
- Load *textures* as images
- Set *texture* properties

II: Define Objects

- Describe object with *primitives*
  - points, lines, triangles
  - made up of *vertices*
- Define *colour*, *material properties*, *textures*
  - for each vertex

III: Geometric Operations

- *Transform* vertices to image coordinates
  - apply *model, view, projection* matrices
  - perform *perspective* division
    - apply *viewport* transformation
- Also transform *normal* vectors
- *Clip* primitives to view volume

IV: Rasterization

- Rasterize primitives to *fragments*
  - i.e. pixels which may or may not render
- Use barycentric coordinates to interpolate
  - colour
  - material properties
  - texture coordinates

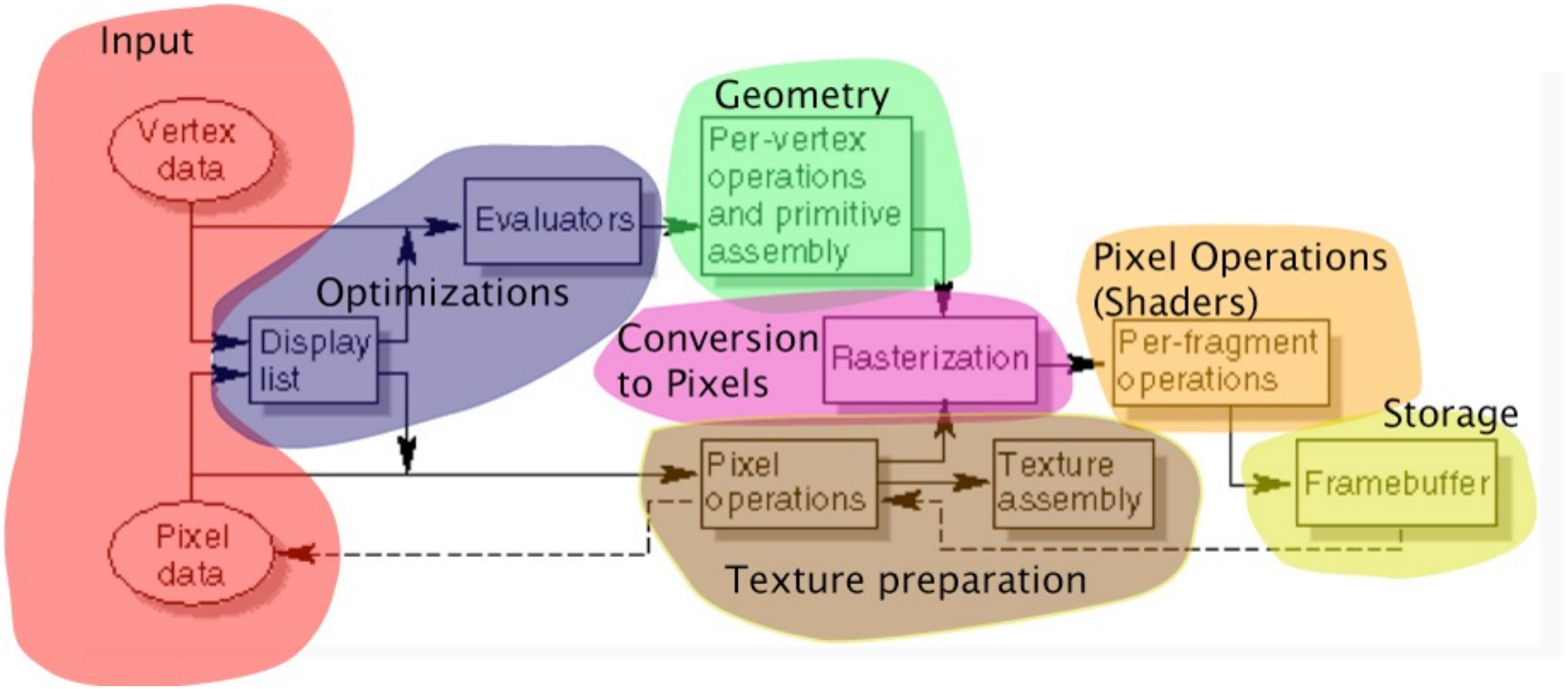
V: Pixel Operations

- For each *fragment*
  - find *Texel* colour
    - with bilinear or NN interpolation
  - *combine* texture & shading
    - *replace* or *modulate*
- perform *fragment* tests
  - Stencil (aids shadows) , depth buffer (is one object in front of another, &c.

VI: Framebuffer

- Write pixel to framebuffer
- Keep two framebuffers (at least)
  - for *double-buffering*
- Hardware transfers framebuffer to screen
  - or to an image
  - or to a texture

OpenGL 2 Pipeline



- From the OpenGL Red Book

Input

- We will:
  - create & model surfaces using triangles
    - feed the triangles into the pipeline
      - with surface properties
        - colour, texture, material, &c.

Vertex Operations

- Per-vertex operations are mostly geometric:
  - transformations are applied here
    - rotation, projection, &c.
  - texture coordinates are calculated
  - lighting is computed

Primitive Assembly

- This includes:
  - *clipping* objects at boundaries
  - *culling* objects that are invisible
  - *perspective division* for foreshortening

Texture Preparation

- Loading textures into memory (or VRAM)
  - from main memory
  - from disk
  - from framebuffer
  - and modifying them if needed

Rasterization

- Converts triangles to *pixels*
  - also known as *scan conversion*
- Generates multiple pixels per triangles
  - this allows parallelization
- Assigns colour, depth, &c. to each pixel
- OpenGL calls pixels *fragments* here

Pixel Operations

- For each pixel, video card computes:
  - texture effects
  - fog, scissor, alpha, stencil, & z-buffer
  - blending, dithering, masking, logic
  - programmable shaders go here!

An Example

```
for (float i = 0.0; i < nSegments; i += 1.0)
  { /* a loop around circumference of a tube */
  float angle = PI * i * 2.0 / nSegments ;
  float nextAngle = PI * (i + 1.0) * 2.0 / nSegments;

  /* compute sin & cosine */
  float x1 = sin(angle), y1 = cos(angle);
  float x2 = sin(angle), y2 = cos(angle);

  /* draw top (green) triangle */
glNormal3f(x1, y1, 0.0); glVertex3f(x1, y1, 0.0);
glNormal3f(x2, y2, 0.0); glVertex3f(x2, y2, 1.0);
glNormal3f(x1, y1, 0.0); glVertex3f(x1, y1, 1.0);

/* draw bottom (red) triangle */
glNormal3f(x1, y1, 0.0); glVertex3f(x1, y1, 0.0);
glNormal3f(x2, y2, 0.0); glVertex3f(x2, y2, 0.0);
glNormal3f(x2, y2, 0.0); glVertex3f(x2, y2, 1.0);

} /* a loop around circumference of a tube */
```

OpenGL Conventions

- All functions start with gl: e.g. glEnd()
- Constants start with GL_: e.g. GL_LINES
- Suffix describes parameters, e.g. in glVertex3fv(...), suffix 3fv gives:
  - dimension: 1, 2, 3, or 4:
  - data type: (b)yte, (i)nt, (f)loat, (d)ouble
  - whether data is an array: (v)ector

OpenGL State

- Once a property is set, it hangs around
  - e.g. transformation matrices
    - also colour, texture, &c., &c.
- This is technically referred to as *state*
  - often a source of problems

Input (Vertices)

- OpenGL describes objects with vertices:
  - use glBegin() to start
  - use glEnd() to stop
- OpenGL does **NOT** balance pairs
  - so be careful

glBegin()

- glBegin(GLenum mode);
- mode can be:
  - GL_POINTS
  - GL_LINES
  - GL_TRIANGLES
  - among others

glBegin() & glEnd()

- Vertices between these are grouped:
  - if GL_POINTS, each vertex is a point
  - if GL_LINES, each pair is a line
  - if GL_TRIANGLES, each trio is a triangle
- Extra vertices are discarded

Restrictions

- Only the following can be used between glBegin() and glEnd():
  - glVertex(), glNormal()
  - glColor(), glMaterial()
  - glTexCoord()
  - & some others (not relevant yet)

Vertex Properties

- Vertices can be assigned:
  - Normal Vectors - glNormal()
  - Colours - glColor() (if lighting off)
  - Materials - glMaterial() (if lighting on)
  - Textures - glTexCoord()
  - Position - glVertex() (always comes *last)*

or Surface Properties

- These are actually *surface* properties
  - *normal* is perpendicular to surface
    - ***NOT*** to the triangle!
- OpenGL will interpolate properties
  - pick values between those of vertices
  - computed during rasterization

OpenGL Lighting

- Turn on / off lighting
  - glEnable(GL_LIGHTING)
  - glDisable(GL_LIGHTING)
- Turn on a specific light
  - glEnable(GL_LIGHT0)
  - glDisable(GL_LIGHT0)

Light Parameters

- Lights have:
  - position, colour, and other properties
- Specify with glLightf()
  - `glLightf(GL_LIGHT0,GL_AMBIENT,redColour)`
  - assigns a red ambient colour to #0

Light Parameters

- colour: GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR
- position: GL_POSITION
- spotlights: GL_SPOT_DIRECTION, GL_SPOT_EXPONENT, &c.

Shading Model

- Can use flat or smooth (Gouraud)
  - glShadeModel(GL_FLAT)
  - glShadeModel(GL_SMOOTH)
  - can’t use Phong shading

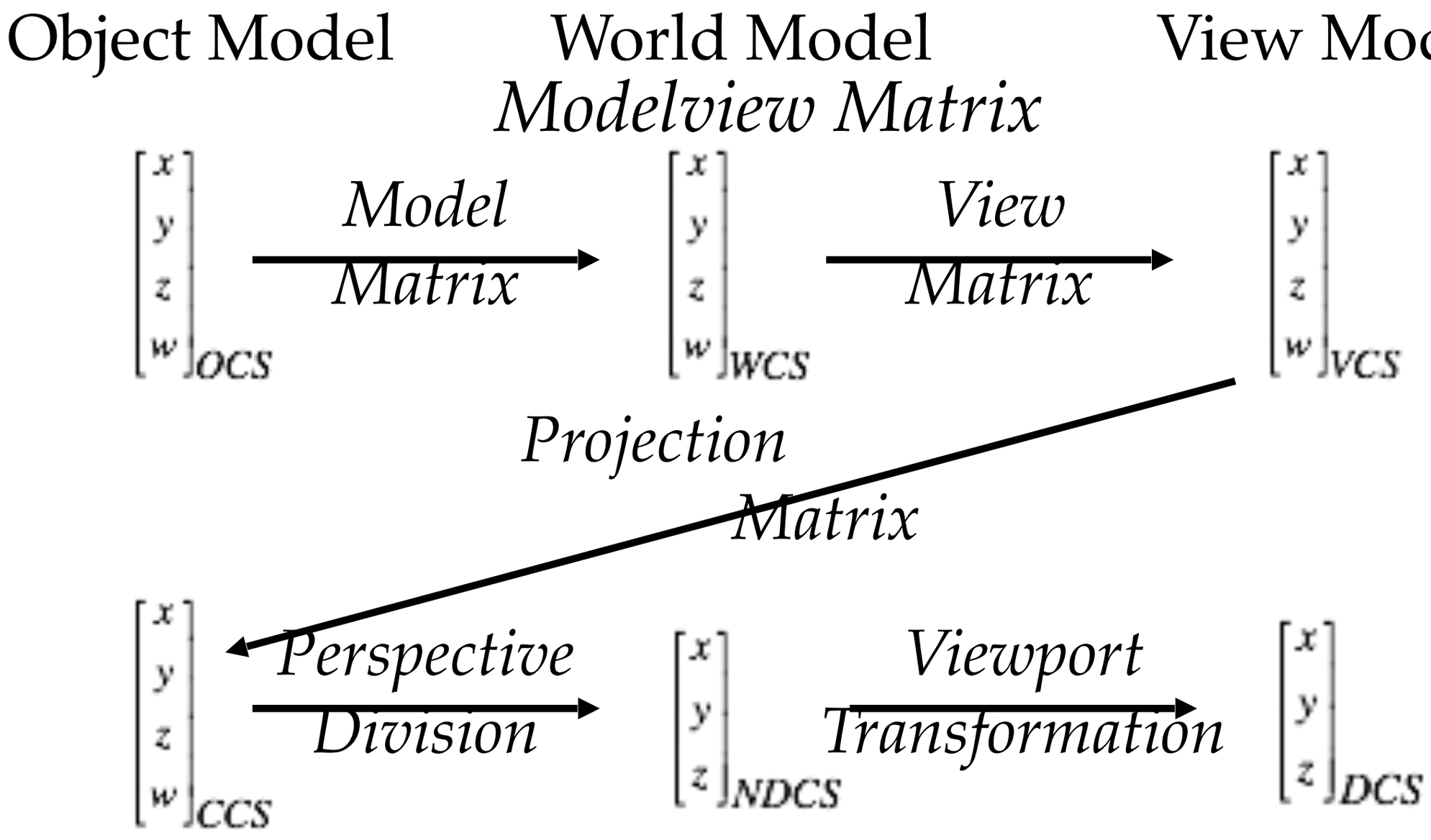
Lighting Model

- glLightModel(GL_LIGHT_MODEL_TWO_SIDE,1)
- switches on two-sided lighting

Transformations

- OpenGL uses two principal matrices
  - a *modelview* matrix
  - a *projection* matrix
- To specify which one is being changed:
  - glMatrixMode(GL_MODELVIEW);
  - glMatrixMode(GL_PROJECTION);

Matrices


Object Model
World Model
View Model
Modelview Matrix
Model Matrix
View Matrix
Projection Matrix
Perspective Division
Viewport Transformation
Clipping
Image Plane
Screen

Modelview Matrix

- Combines *model* (M) and *view* (V) matrices
  - view transformation happens last
  - so always specify it first
  - but view matrix changes less anyway
  - if needed, make V changes in proj. mat.

Matrix Manipulation

- glLoadIdentity(): sets matrix back to I
  - good idea always to start with this
- glLoadMatrix(): sets a specific matrix
  - stored in *column-major* order
- glMultMatrix(): *multiply* current matrix
  - applies additional transformation

Column-Major Order

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ 3 & 2 & 1 & 0 \\ 4 & 3 & 2 & 1 \end{bmatrix}$$

- OpenGL stores this matrix as:
  - glInt *M = {1, 2, 3, 4, 0, 1, 2, 3, 0, 0, 1, 2, 0, 0, 0, 1};

Transformations

- glTranslate() is easy
- so is glScale()
- glRotate(angle, x, y, z) rotates by angle
  - CCW around vector (x, y, z)
  - looking in from (x, y, z) to origin
- there is no glShear()

View Coordinates

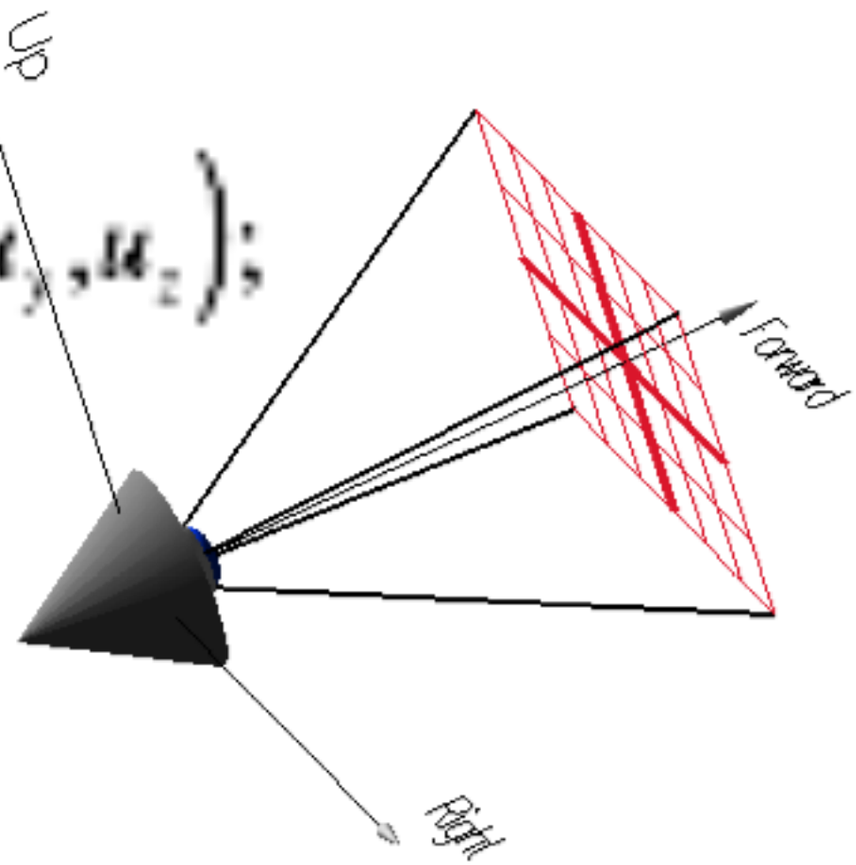
- We can specify the view point with
- import org.lwjgl.util.glu.GLU;
- GLU.gluLookAt(eyex, eyey, eyez, centerx, centery, centerz, upx, upy, upz);
- gluLookAt(Ex,Ey,Ez,Cx,Cy,Cz,Ux,Uy,Uz)

gluLookAt$(e_x, e_y, e_z, c_x, c_y, c_z, u_x, u_y, u_z)$;

which puts the eye at (e_x, e_y, e_z),

facing toward (c_x, c_y, c_z),

with *up* vector (u_x, u_y, u_z)



Projection Matrix

- Switch with glMatrixMode(GL_PROJECTION);
- Then define projection with:
  - glFrustum(*left, right, bottom, top, zNear, zFar*);
  - gluPerspective(*fovy, aspect, zNear, zFar*);
  - glOrtho(*left, right, bottom, top, zNear, zFar*);
- All set matrix for eye at origin looking at (0,0, -1)
- often convenient to apply transforms first

glFrustum()

- near plane of frustum is given by:
  - (*left, bottom, -zNear*)
  - (*right, top, -zNear*)
- far plane is at distance *zFar*
  - with left, right, &c. scaled appropriately
- Anything outside frustum will be *clipped*

gluPerspective()

- *fovy* is the field of view in y-direction
  - in degrees
- *aspect* is the aspect ratio:
  - ratio between x- and y- fields of view
- *zNear* and *zFar* are same as glFrustum()

Viewport

- Specifies mapping from *NDCS* to *DCS*
  - *NDCS* usually ends up in range -1 .. 1
  - *DCS* is in terms of pixel location
    - use glViewport(*0, 0, width, height*)
      - *width* is pixel width of window
      - *height* is pixel height of window