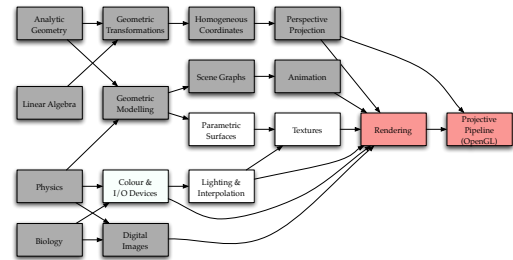


# Rendering and the OpenGL Pipeline

COMP 3003  
Autumn 2005

## Where We Are



## Topics Today

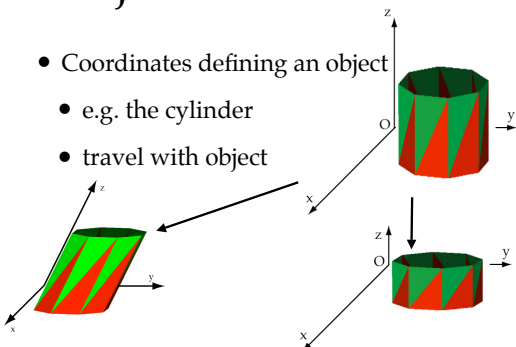
- Coordinate Systems
- Rendering Models
- OpenGL

## Coordinate Systems

- OCS is the *Object Coordinate System*
- WCS is the *World Coordinate System*
- VCS is the *View Coordinate System*
- CCS is the *Clipping Coordinate System*
- NDCS is the *Normalized DCS*
- DCS is the *Device Coordinate System*

## Object Coordinates

- Coordinates defining an object
  - e.g. the cylinder
  - travel with object



## World Coordinates

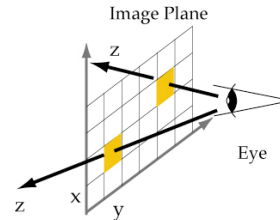
- Arbitrary coordinate system
- Where is the origin?
  - The Earth?
  - The Sun?
  - Greenwich meridian?
- Used to keep track of other systems

## View Coordinates

- Tilt your head 90 degrees right
- “Up” (y) is now to the right
- “Backward” (z) hasn’t changed
- We can use cross-product to get x
- Each camera / eye has coordinates

## Clipping Coordinates

- coordinates in perspective projection
- z is distance from the eye



## Clipping Coords

- Projection to CCS models the eye’s lens
- In CCS, z is the distance from the eye
  - *not always in same direction*
  - i.e. Cartesian coordinates fail
  - homogeneous coordinates work

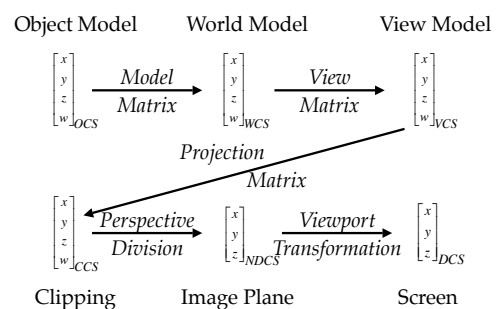
## Normalized DCS

- *Normalized Device Coordinate System*
- Big mouthful, but simple idea
- Divide CCS through by w
  - converts homog. coords to Cartesian

## Device Coordinates

- Finally, we convert into *device* coords
  - (x, y): position on image plane
  - z: distance in front of image plane
- I.e. pixel position & object depth

## Why so many?

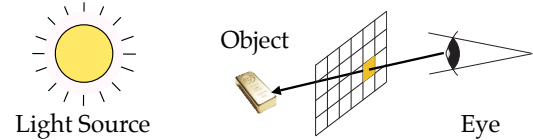


## Rendering Models

- There are three major forms of rendering:
  - ray-tracing
  - radiosity
  - projective rendering
- OpenGL uses projective rendering for speed

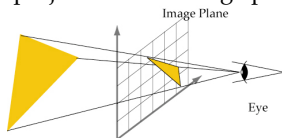
## Raytracing

- For each pixel
  - Start at eye
  - Trace a ray through image plane
  - Compute colour of object it hits



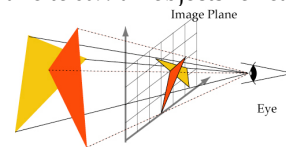
## Projective Rendering

- Ray-tracing computes one pixel at a time
- Instead, we compute multiple pixels
  - For each triangle, compute it's image
  - i.e. project it to the image plane



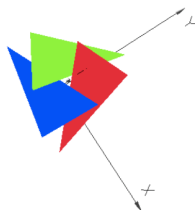
## Painter's Algorithm

- If we draw objects from back to front
  - the back objects will be *occluded*
    - i.e. we will see only the front object
- We have to *sort* all objects for each image



## Worst Case

- Three triangles
  - Red overlaps Green
  - Green overlaps Blue
  - Blue overlaps Red
- Painter's Algorithm fails!



## Solutions

- Compute intersections & redraw (slow)
- Cheat (fast)
  - When drawing a pixel, store z value
  - Check previous z value first
    - only draw pixel if new z is larger
- This is called z-buffering

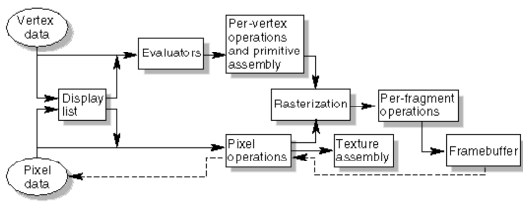
## Why Triangles?

- Triangles are
  - the *simplest* 3-D object (aka *simplices*)
  - always flat (makes computation easier)
  - defined by three vertices
  - cheap to process (i.e. fast)

## Origin of OpenGL

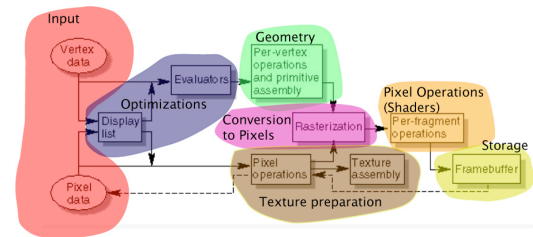
- IrisGL on Silicon Graphics Irix OS
- SGI opened it up for other OS'es
- Now the *de facto* standard in graphics
  - projectively rendered triangles
  - hardware-accelerated (pipelines)
  - texture-oriented

## Projective Pipeline



- From the Red Book

## Breakdown



## Input

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

## Optimization

- We will talk about:
  - display lists
  - evaluators
  - splines
  - LATER!
  - For now, ignore them

## Vertex Operations

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

## 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
- We'll worry about this later

## Rasterization

- Converts triangles, &c. 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!

## 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:
  - Position - glVertex()
  - Normal Vectors - glNormal()
  - Colours - glColor() (if lighting off)
  - Materials - glMaterial() (if lighting on)
  - Textures - glTexCoord()

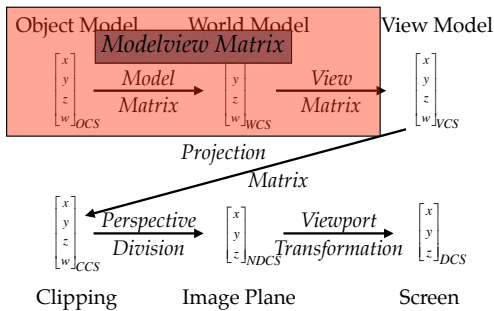
## or Surface Properties

- 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

## 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 (again)



## 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

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

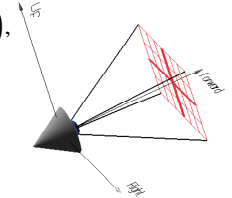
- OpenGL stores this matrix as:
  - `glInt *M = {1, 2, 3, 4, 0, 1, 2, 3, 0, 0, 1, 2, 0, 0, 0, 1};`

## 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 `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