the gamedesigninitiative at cornell university

Lecture 10

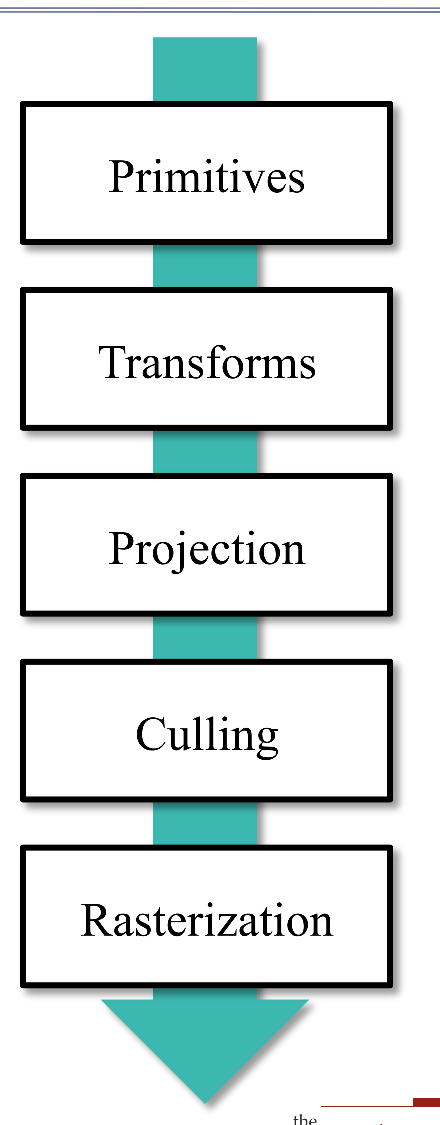
OpenGL Basics

Scope of Today's Talk

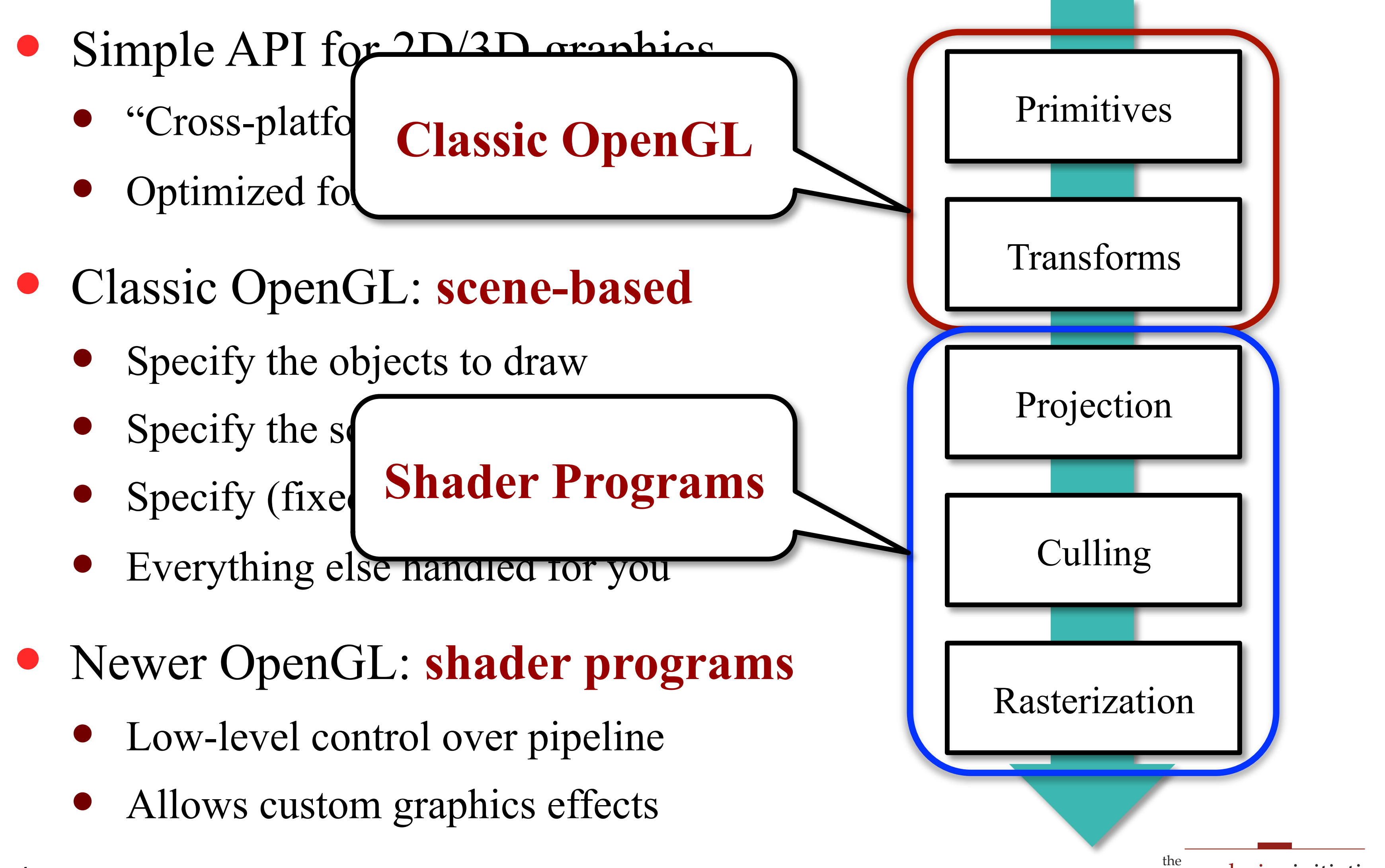
- Introduce the basics of OpenGL
  - Some material is review from computer graphics
  - But cover a lot less detail (only what we need)
- Why should you learn OpenGL?
  - Most 2D game engines abstract it away
  - But might want to add effects (e.g. particles)
  - Might want tighter control over memory use
  - Might want tighter control over player perspective

What is OpenGL?

- Simple API for 2D/3D graphics
  - "Cross-platform" 3D solution
  - Optimized for graphics cards
- Classic OpenGL: **scene-based**
  - Specify the objects to draw
  - Specify the scene perspective
  - Specify (fixed) scene lighting
  - Everything else handled for you
- Newer OpenGL: **shader programs**
  - Low-level control over pipeline
  - Allows custom graphics effects



What is OpenGL?



History of OpenGL

- (1992) Open Source alternative to **Iris GL**
  - Used in the Silicon Graphics workstations
  - Leading 3D platform in the early 90s
- Accompanied by a architectural review board
  - Oversees changes in the language specification
  - **Founders**: SGI, Intel, IBM, DEC and *Microsoft*
- Changes in the language have been historically slow
  - OpenGL 2.0 (2004)
  - OpenGL 3.0 (2008)

OpenGL: A Cross-Platform Solution?

- OpenGL is available on just about *any* device
  - Android and iOS have it built-in
  - OS X and most Linux distros have it built-in
  - Windows supported via a 3rd party (GLEW, GLFW)
  - Standard Java support via JOGL, a 3rd party library
- But understanding versions is complicated
  - Mobile platforms only support **OpenGL ES**
  - Linux and Windows can support **OpenGL 4.x**
  - OS X only currently supports **OpenGL 3.2**
  - JOGL supports whatever the **platform** supports (OUCH)

What are the Differences?

- **OpenGL 2.x**: Equivalent to DirectX 9.x
  - Supported on *all* non-mobile platforms
  - Focus of all *classic* Open GL documentation
- **OpenGL 3.x**: Equivalent to DirectX 10.x
  - Made shader programs the *primary focus*
  - Standardized hardware optimization
- **OpenGL ES**: OpenGL 3.x stripped down
  - Removed deprecated features of OpenGL 2.x
  - Only vertex and fragment (e.g. not geometry) shaders
- **OpenGL 4.x**: Equivalent (?) to DirectX 11.x

What are the Differences?

- **OpenGL 2.x**: Equ X 9.x
  - Supported on *all* r s
  - Focus of all *classi* ntation

Good Place to Learn

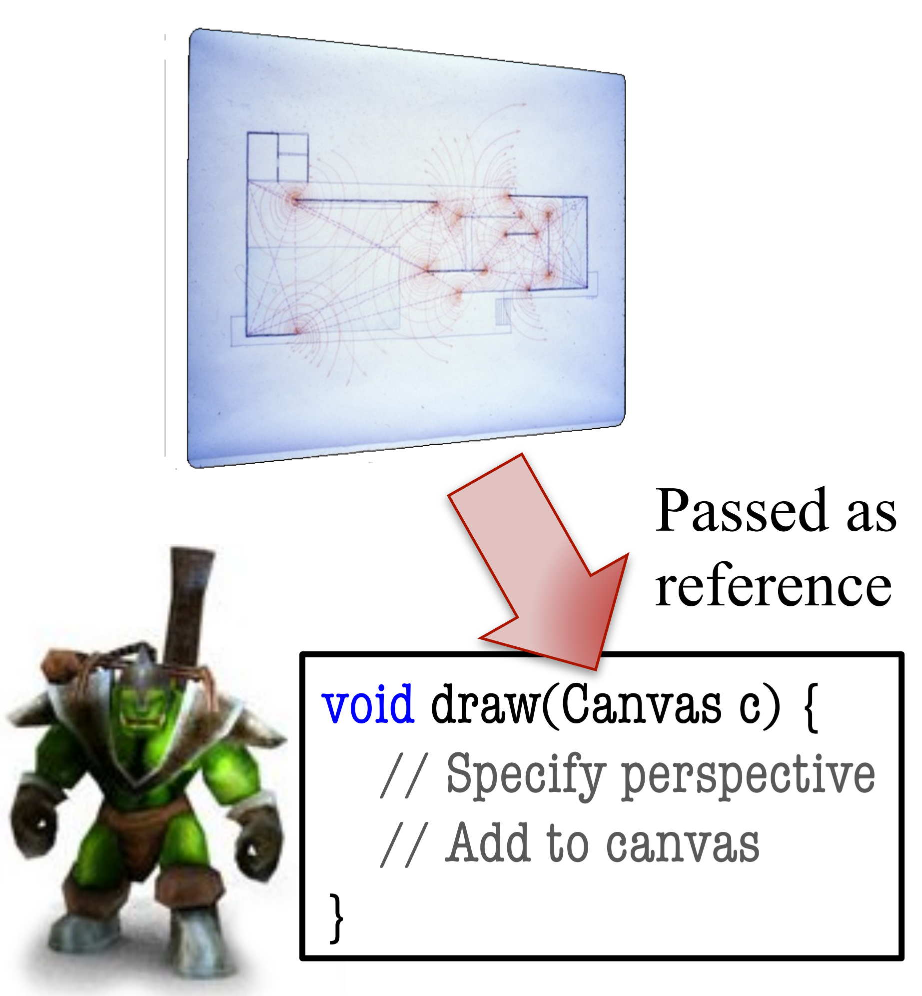
- **OpenGL 3.x**: Equivalent to DirectX 10.x
  - Made shader programs the *primary focus*
  - Standardized hardware optimization
- **OpenGL ES**: Ope d down
  - Removed depreca GL 2.x
  - Only vertex and f ometry) shaders

Needed for Mobile Devices

- **OpenGL 4.x**: Equivalent (?) to DirectX 11.x

Getting Started with OpenGL

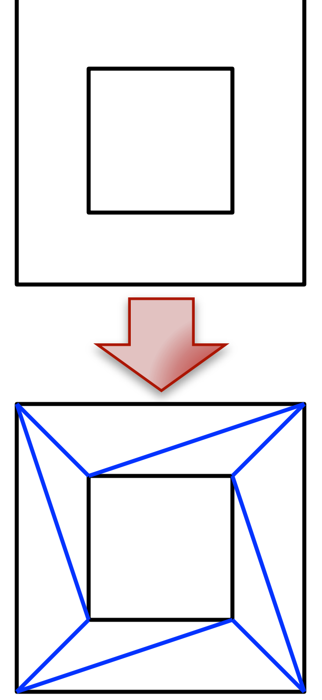
- Start with a *GL canvas*
  - OO abstraction of display
  - Draw to it via methods
  - ... or pass it into functions
- Depends on your *platform*
  - **Classic Canvas**: GLUT
  - **iOS**: EAGLContext
  - **Android**: GLSurfaceView
  - **JOGL**: GLCanvas
- Will use JOGL for demos
  - Quick to understand



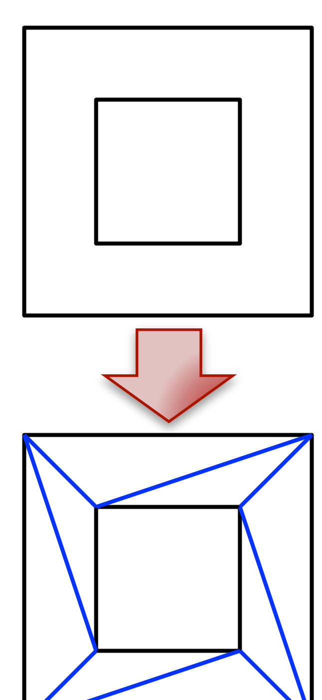

```
void draw(Canvas c) {
    // Specify perspective
    // Add to canvas
}
```

Classic OpenGL APIs

- **GL**: Lowest level API
  - Supports basic scene manipulation
  - Entry point for shader programs
- **GLU**: Utility and helper functions
  - Breaking complex shapes into triangles
  - Convert any perspective into its matrix
- **GLUT**: Window and interface management
  - Historically the source of your GL canvas
  - Made obsolete on most modern platforms



Classic OpenGL APIs



- **GL**: Lowest level API
  - Supports basic scene manipulation
  - Entry point for shader programs
- **GLU**: Utility and helper functions
  - Breaking complex shapes into triangles
  - Convert any perspective into its matrix
- **GLUT**: Win and interfa anagement
  - Historically the s GL canvas
  - Made obsole most mode tforms

Types of OpenGL Functions

- **Setting Functions**
  - Enable/disable functionality
  - Control OpenGL state
  - **Example**: alpha, transforms

- glEnable(capability);
- glDisable(capability);
- glLightfv(light, pName, pValue);
- glTranslate(x, y, z);

- **Data Handling Functions**
  - Create persistent structures
  - Involves memory allocation
  - **Example**: Texture loading

- glVertexPointer(...);
- glGenTextures(size, names);
- glDeleteTextures(size, names);
- glTexImage2D(target, level,...);

- **Rendering Functions**
  - Draw and texture primitives
  - **Example**: triangles, quads

- glBegin()/glEnd()
- glVertex3f(x,y,z);
- glDrawElements(...);

OpenGL Conventions

- Many functions have multiple forms:
  - `glVertex2f`, `glVertex3i`, `glVertex4dv`, etc.
- Number indicates number of arguments
- Letters indicate type
  - f: float, d: double
  - i: integer, ub: unsigned byte
- 'v' (if present) indicates a pointer argument
  - `glVertex3f(point.x, point.y, point.z)`
  - `glVertex3fv(point) // Array or struct!`

Understanding the Pipeline

Description	Stage
Specify the scene contents (geometry, material, lighting)	Primitives
Specify the scene perspective (using matrix transforms)	Transforms
Project scene onto 2D space	Projection
Remove unneeded objects	Culling
Convert objects into pixels	Rasterization

Understanding the Pipeline

Specify the scene contents (geometry, material, lighting)	Primitives
Specify the scene perspective (using matrix transforms)	Transforms
Project scene onto 2D space	Projection
Remove unneeded objects	Culling
Convert objects into pixels	Rasterization

Today

the gamedesigninitiative at cornell university

JOGL: GLEventListener

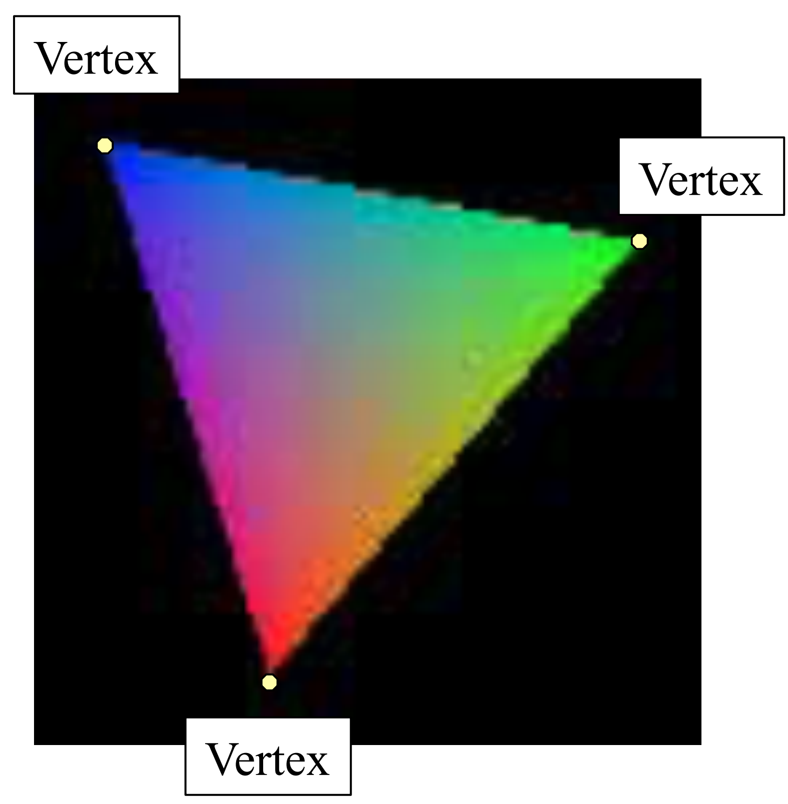
```
public void init(GLAutoDrawable drawable) {
    // Initialize the default settings and allocate any needed memory buffers
}

public void reshape(GLAutoDrawable drawable, int x, int y, int w, int h) {
    // Reset the initialization values if the window size changes
}

public void display(GLAutoDrawable drawable) {
   // Arrange and render scene for the current frame
}

public void dispose(GLAutoDrawable drawable) {
   // Release any memory allocated for Open GL
   // Example: Memory buffers, shader programs
}
```

Primary Method

Example Java Source:
OrthoTriangleFrame.java

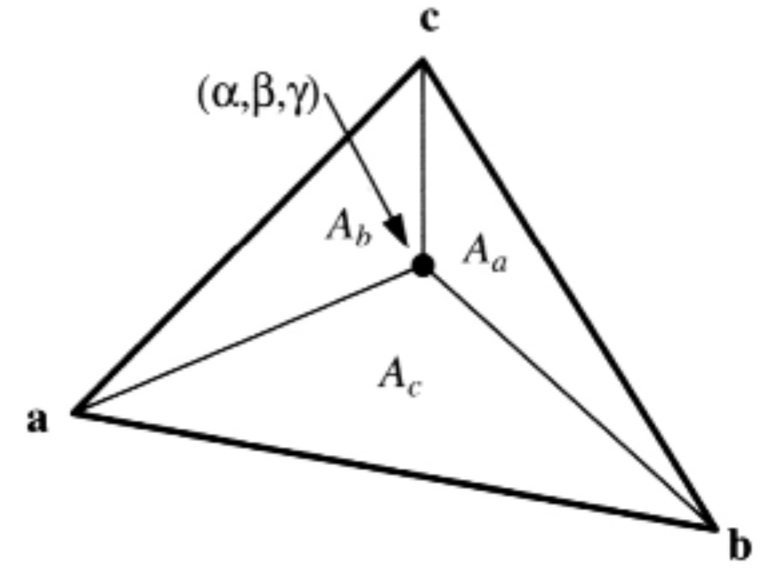
Basic Rasterization

- Specify properties at **vertices**
  - (3D) location of vertex
  - (Base) color of the vertex
  - Texture information (later)
- **Interpolate** internal pixels
  - Pixel associated w/ triangle
  - Attached to three vertices
  - Average colors at vertices
  - Weighted by vertex distance
- Want fine-tuned control?
  - Must use *shader programs*



Barycentric Coordinates

- *Internal* triangle *coordinates*
  - **Algebraic interpretation**:
    - $\mathbf{p} = \alpha\,\mathbf{a} + \beta\,\mathbf{b} + \gamma\,\mathbf{c}$
    - $1 = \alpha + \beta + \gamma$
  - **Geometric interpretation**:
    - α is area of A_a
    - β is area of A_b
    - γ is area of A_c



- Gives the *averaging weights*
  - $\mathbf{p}_C = \alpha\,\mathbf{a}_C + \beta\,\mathbf{b}_C + \gamma\,\mathbf{c}_C$
  - Similar trick for textures

Classic OpenGL Drawing in JOGL

```
// Draw a triangle
gl.glBegin(GL2.GL_TRIANGLES);

// top central vertex
gl.glColor3f(1.0f,0.0f,0.0f);
gl.glVertex3f(width/2.0f, height/4.0f, 0.0f);

// bottom right vertex
gl.glColor3f(0.0f,1.0f,0.0f);
gl.glVertex3f(3.0f*width/4.0f, 3.0f*height/4.0f, 0.0f);

// bottom left vertex
gl.glColor3f(0.0f,0.0f,1.0f);
gl.glVertex3f(width/4.0f, 3.0f*height/4.0f, 0.0f);

// Finish drawing the triangle
gl.glEnd();
```

Current "Pen" Color

Vertex Position
"Draws" the vertex

the gamedesigninitiative at cornell university

OpenGL: glBegin()…glEnd()

```
// Classic OpenGL geometry is passed between glBegin()/glEnd()
glBegin(GL_TRIANGLES);
for (int i=0; i<ntris; i++) {
    glColor3f(tri[i].r0,tri[i].g0,tri[i].b0);            // Color of vertex
    glNormal3f(tri[i].nx0,tri[i].ny0,tri[i].nz0);    // Normal of vertex
    glVertex3f(tri[i].x0,tri[i].y0,tri[i].z0);          // Position of vertex
    …
    glColor3f(tri[i].r2,tri[i].g2,tri[i].b2);
    glNormal3f(tri[i].nx2,tri[i].ny2,tri[i].nz2);
    glVertex3f(tri[i].x2,tri[i].y2,tri[i].z2);
}
glEnd();   // Sends all the vertices/normals to the OpenGL library
```

glBegin Specifies How to Use Vertices

1-Dimensional

- **GL_POINTS** (Draws *n* points)
  - Each vertex an individual point
- **GL_LINES** (Draws *n*/2 lines)
  - Each alternating pair is a line
- **GL_LINE_STRIP** (*n*-1 lines)
  - Connected line segments
- **GL_LINE_LOOP** (n lines)
  - As line strip, but closes loop

2-Dimensional

- **GL_TRIANGLE** (n/3 triangles)
  - Each alternating triple a triangle
- **GL_TRIANGLE_FAN** (n-2 tris)
  - Triangles radiating about a point
- **GL_TRIANGLE_STRIP** (n-2)
  - Like a line strip, for triangles
- **GL_QUADS** (Draws n/4 quads)
  - Ideal for drawing sprites

glBegin Specifies How to Use Vertices

1-Dimensional

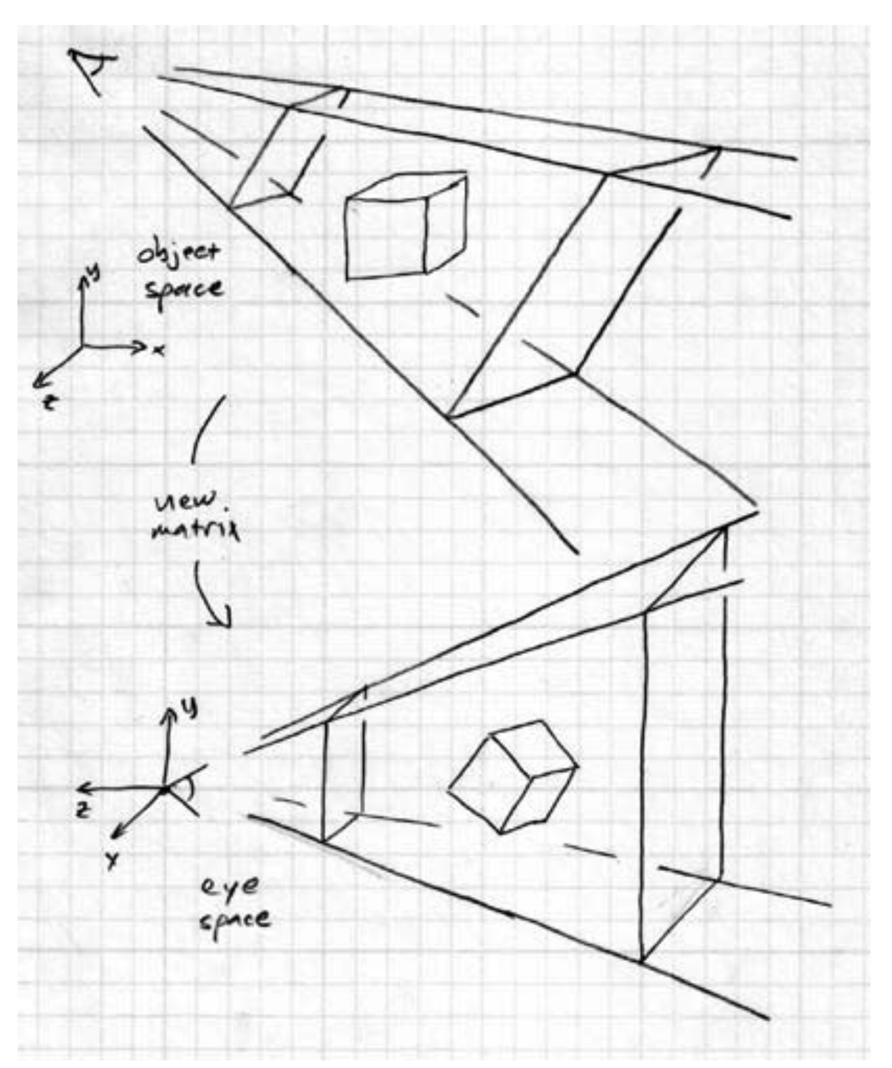
- **GL_POINTS** (Draws *n* points)
  - Each vertex an individual point
- **GL_LINES** (Draws *n*/2 lines)
  - Each alternating pair is a line
- **GL_LINE_STRIP** (*n*-1 lines)
  - Connected line segments
- **GL_LINE_LOOP** (n lines)
  - As line strip, but closes loop

2-Dimensional

- **GL_TRIANGLE** (n/3 triangles)
  - Each alternating triple a triangle
- **GL_TRIANGLE_FAN** (n-2 tris)
  - Triangles radiating about a point
- **GL_TRIANGLE_STRIP** (n-2)
  - Like a line strip, for triangles
- **GL_QU**… …ads)

Perspective and Coordinate Systems

- **Object Space**:
  - Standard coordinates
  - Where we place objects
  - Part of scene specification
- **Eye Space**:
  - With the camera at origin
  - Display = view frustrum
  - Necessary for rendering
  - Often defined in init()
- *Change of basis* problem
  - Done with matrices



The View Frustrum

z=-f

y

z=-n

1

z

d

-1

Near

Far

Field of View

y'

z'=-1

z'=1

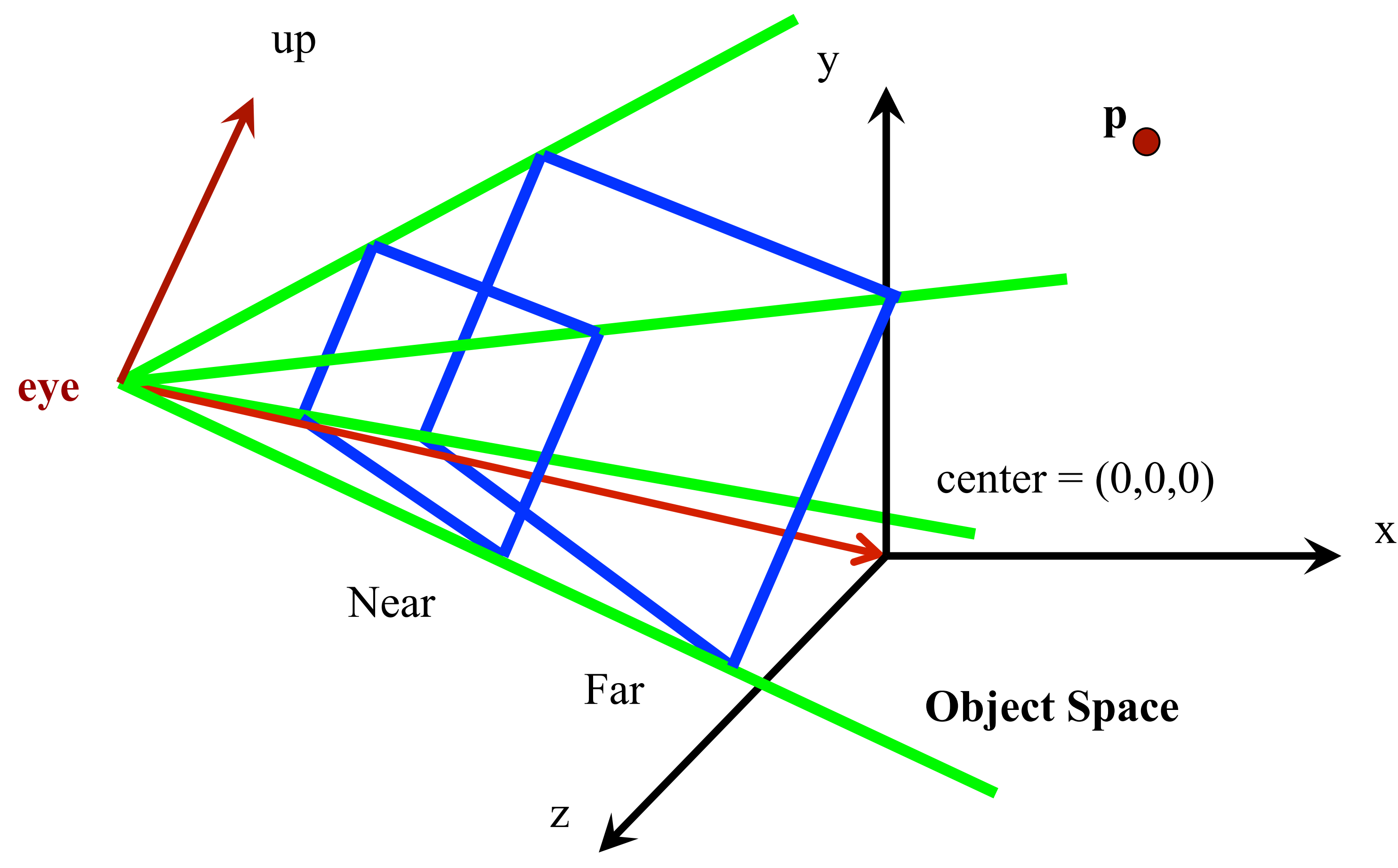
z'

How do we specify this in OpenGL?

Two Things to Specify

- **MODELVIEW Matrix**: Location of camera (eye)
  - Where is the *origin* of eye space?
  - Which *direction* is it pointing?
  - What is the *orientation* of the camera?
- **PROJECTION Matrix**: Projection properties
  - What is *depth of field* (e.g. near and far)?
  - What is *field of view* (extent of x and y directions)?
- Control current matrix with the **matrix mode**
  - **Example**: `glMatrixMode(GL_PROJECTION);`

Specifying the Projection



Defining the Projection

OpenGL Simplifies This Process

Eye Space Projection

- You specify the features
  - OpenGL makes matrices
- Requieres two functions
  - `gluPerspective(fov,asp,far,near);`
  - `gluLookAt(eye,target,up);`

Orthographic Projection

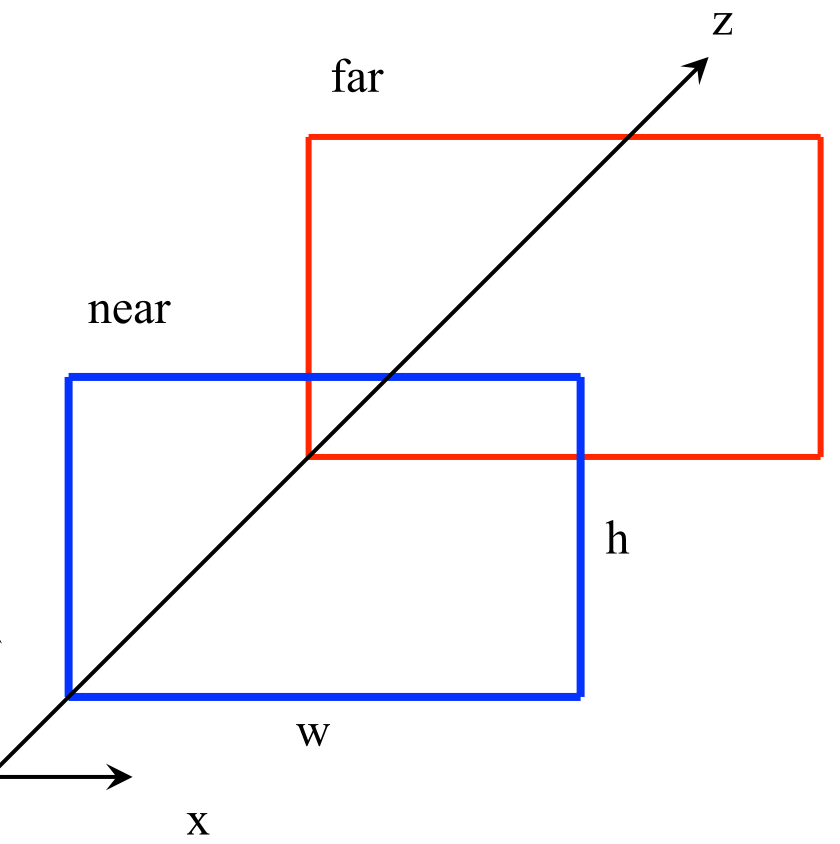
- Fits view to x-y plane
  - Moves eye so display fits
  - Natural fit for 2D games
- Requires one function
  - `glOrtho(x, w, y, h, near, far);`

Calling functions computes appropriate matrix and loads it into OpenGL

Using gluPerspective

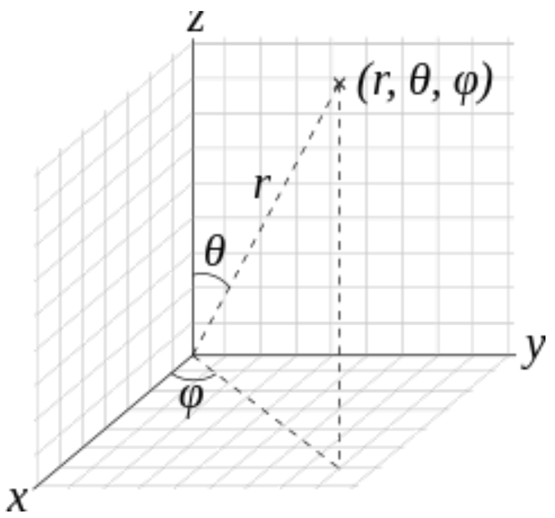
```
// Specify the active mode
glMatrixMode(GL_PROJECTION);

// Clear the current matrix values
glLoadIdentity();

gluPerspective(fovy, aspect, near, far);
```

aspect = w/h



Example: CameraTriangleFrame

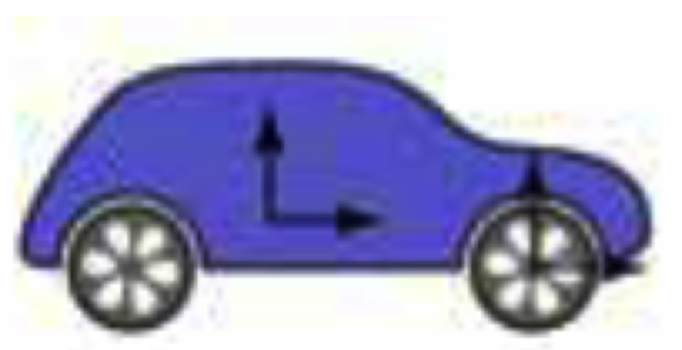
- Shows off (movable camera)
  - Scene origin fixed
  - Eye distance fixed
  - Only eye & up can change
- Eye moves along a sphere
  - Sliders use spherical coords
  - Converts to cartesian coords
- Classic camera control
  - Mouse controls θ and ϕ
  - Scroll to control distance

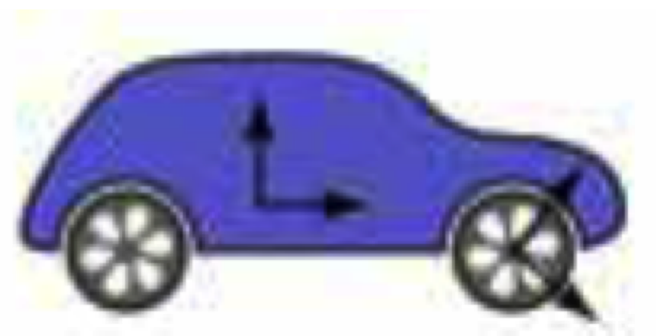


- Conversion formula
  - $x = r \sin\theta \cos\phi$
  - $y = r \sin\theta \sin\phi$
  - $z = r \cos\theta$
- Also need to convert **up**

Classic OpenGL Stacks

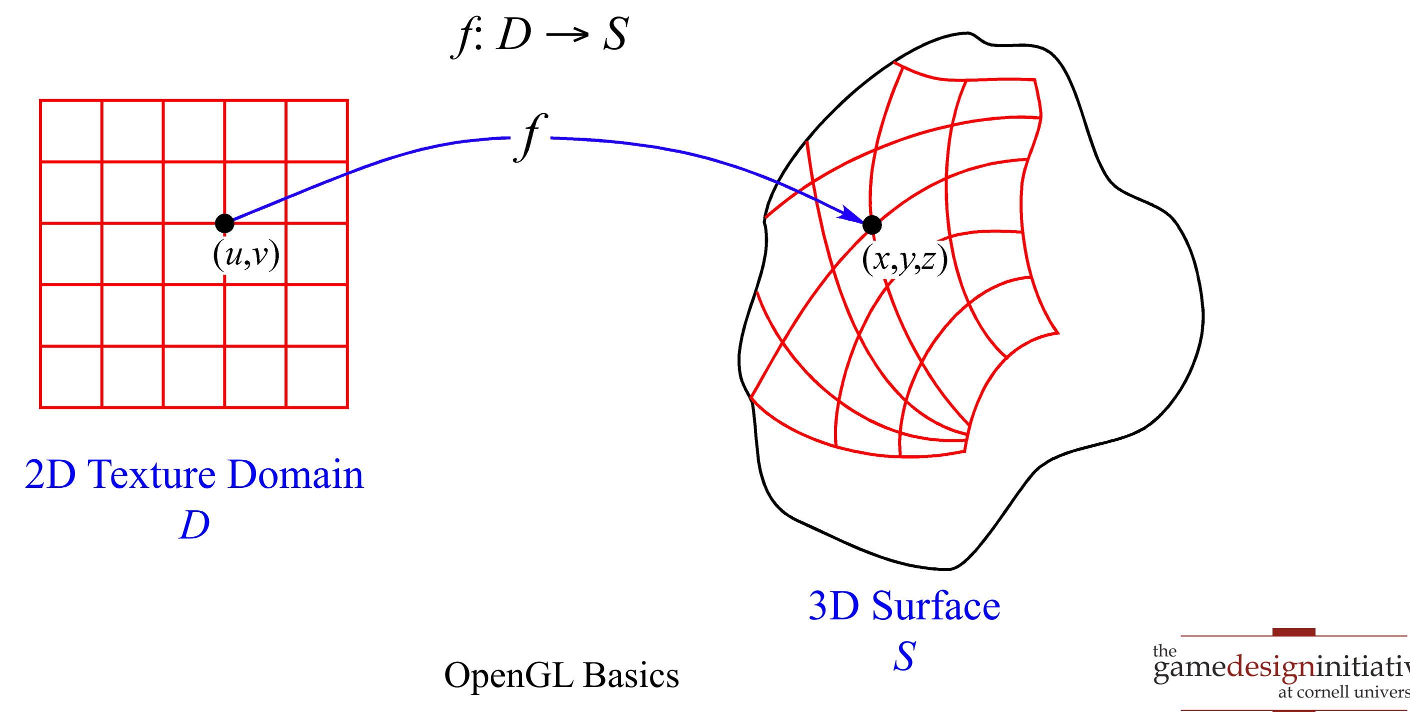
- May want many coord spaces
  - Each sprite has its own basis
  - May animate each separately
- Manage context with the stack
  - Create a new context with push
    - Copies current matrices
    - Modify w/ transforms
    - **Example**: rotate, translate, ...
  - Restore old context with pop
- Applies to any type of settings
  - **Example**: texture, blending





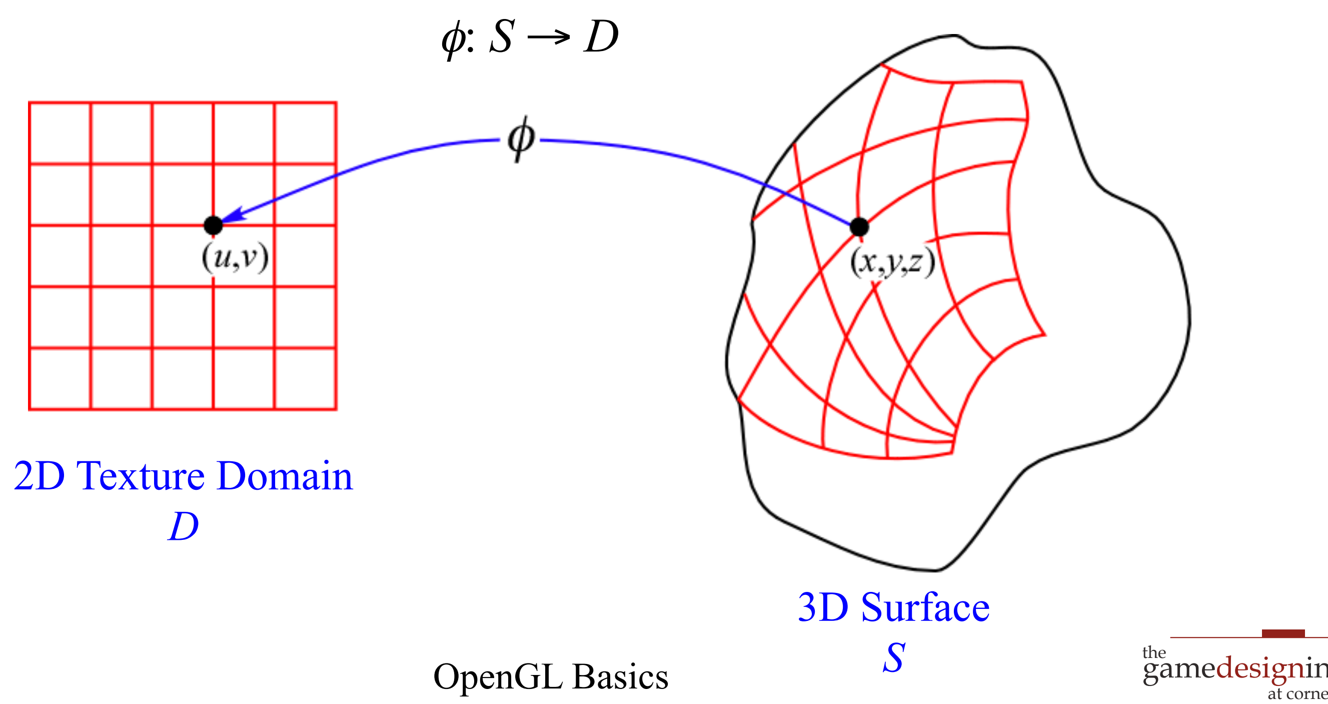
Mapping Textures to Surfaces

- Challenge is "putting the image on the surface"
- Need function *f* to map texture pixels to surface
  - Color that vertex with color at texture pixel



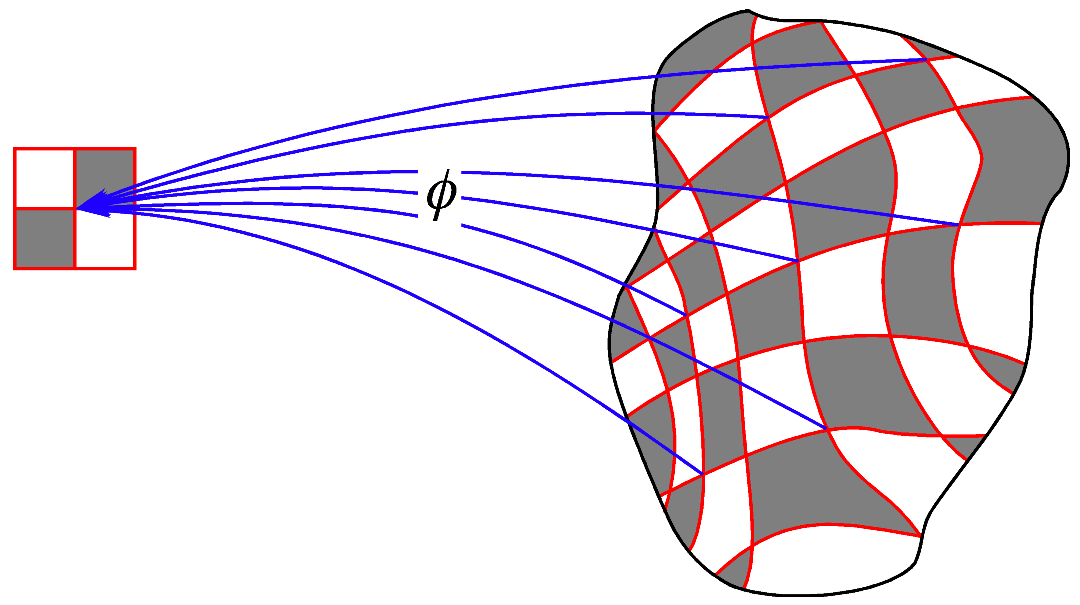
Texture Coordinate Functions

- Hard for programmer to specify function f
- Specify texture coordinate function: ϕ: $S \rightarrow \mathbb{R}^2$
  - For a vertex at **p**, we get texture at $f(\mathbf{p})$
  - Function f is inverse of this function



Texture Coordinate Functions

- Mapping from S to D can be many-to-one
  - Every surface point gets only one color assigned
  - But many surface points can map to same texture point
- **Example**: Tessellated textures



Textures and Triangles

(0,0) (1,0)

(0,1) (1,1)

Interpolates

Specify Both!

(1,0)
(4,0)

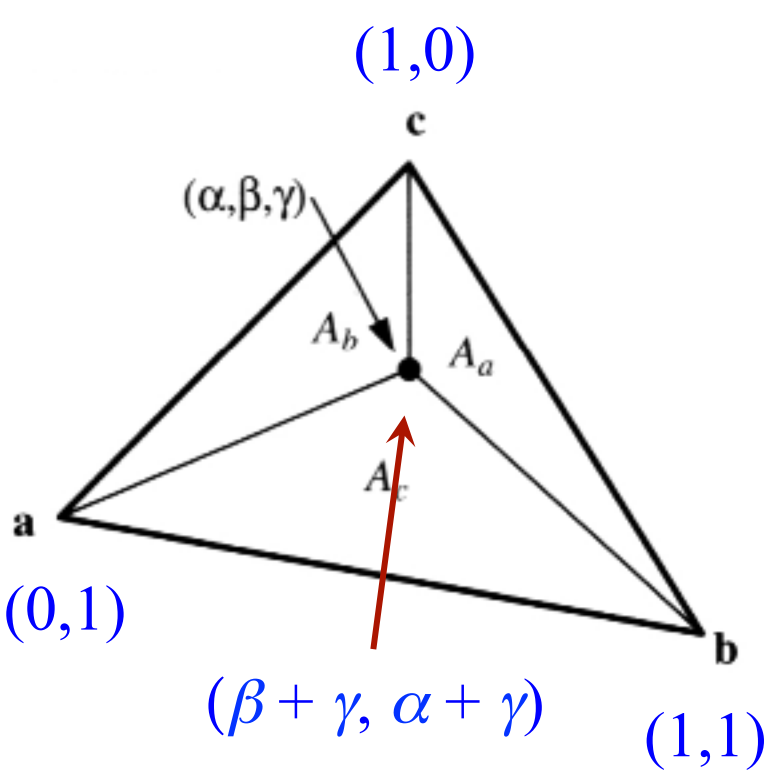
(0,4) (3,4)
(0,1) (1,1)

Texture Coordinates
(even if not square)

Triangle Coordinates

Texture Coordinate Interpolation

- Similar to pixel coloring
  - Texture coord for each vertex
  - Barycentric coords for insides
- Average coords, not colors
  - $\mathbf{p}_u = \alpha\,\mathbf{a}_u + \beta\,\mathbf{b}_{\underline{u}} + \gamma\,\mathbf{c}_u$
  - $\mathbf{p}_v = \alpha\,\mathbf{a}_v + \beta\,\mathbf{b}_{\underline{v}} + \gamma\,\mathbf{c}_v$
- What if not an exact match?
  - Define *rescaling filter*
  - Define *clamp behavior*
  - **Example**: `GLTexture.java`



Textures and More Complex Shapes

(0,0) (1,0)

(0,1) (1,1)

Interpolates

(0,0) (1,0)

(2,0) (4,0)

(0,4) (3,4)

(0,1) (1,1)

Texture Coordinates
(even if not square)

Triangle Coordinates
(more than one triangle)

Classic Texture Mapping in JOGL

```
gl.glBegin(GL2.GL_QUADS);

// top left vertex
gl.glTexCoord2f(0.0f, 1.0f);
gl.glVertex3f(dx, 3.0f*dy, 0.0f);

// bottom left vertex
gl.glTexCoord2f(0.0f, 0.0f);
gl.glVertex3f(dx, dy, 0.0f);

// bottom right vertex
gl.glTexCoord2f(1.0f, 0.0f);
gl.glVertex3f(3.0f*dx, dy, 0.0f);

// top right vertex
gl.glTexCoord2f(1.0f, 1.0f);
gl.glVertex3f(3.0f*dx, 3.0f*dy, 0.0f);

gl.glEnd();
```

Not in OpenGL ES

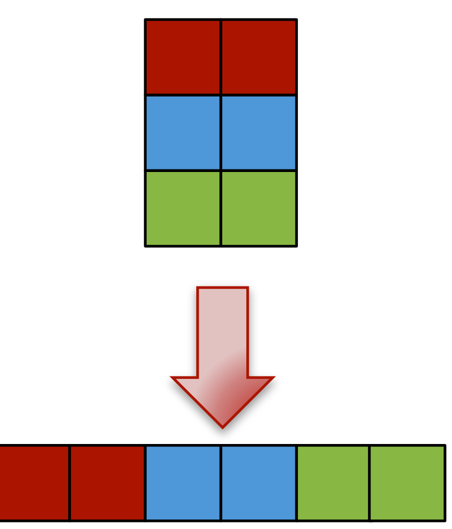
Current Texture Coord

Vertex Position
"Draws" the vertex

What happens when we combine this process with **colors**?

Some Words on Texture Loading

- Read file into a (1-dimensional) integer array
  - 4 bytes in int to store RGBA encoding
  - Uses row major ordering
- Also need to generate texture
  - Create with `glGenTexture` function
  - Loads texture into graphics card memory
  - Returns integer (e.g. pointer) to id texture
- Must be disposed, even in Java!
  - **Example**: `GLTextureLoader.java`, `GLTexture.java`



Next Time

- Loss of **immediate mode**
  - No more glBegin()/glEnd()
  - Have to load all into memory
  - Instruct it to draw contents
- Writing **shader programs**
  - Low-level control over pipeline
  - Allows custom graphics effects
  - Replaces OpenGL stack