ES3 Lecture 6

Mobile Graphics: OpenGLES

What is OpenGLES?

- OpenGLES is a standard API for accelerated 2D + 3D graphics
    - Implemented on many platforms
    - Standard maintained by Khronos group *http://www.khronos.org/opengles/spec/*

- Cut-down version of OpenGL standard
    - Much simplified version
    - Implements modern features from OpenGL
        - Basically restricts data types (to GL_FLOAT)
        - Color models are always RGBA
        - Only allows vertex buffer objects -- no immediate mode drawing
- Allows drawing of geometric primitives with coloring, lighting and texturing
    - Hardware does the rendering
    - Primitives are points, lines or triangles
- Rendering with OpenGLES is timeconsuming to implement
    - but is standard, and offers best performance and access to features

What we'll cover

- OpenGL conventions
- Setting up the OpenGL state
- Simple orthogonal views
- Creating vertex arrays
- Drawing colored lines, points and triangles
- Transforming geometry
- Loading textures
- Using simple textures for 2D sprites

What we'll not cover!

- OpenGLES 2.0 functionality (shaders)
- 3D projection
- Depth buffering, depth testing, clipping
- Loading and working with 3D models
- Lighting and materials
- Stencil buffers, framebuffer objects, scissoring, fog
- Multi-texturing
- Mip-mapping

- **Anything in very much detail!**

Anatomy of OpenGLES

- OpenGLES (and OpenGL) are **state machines**
- OpenGL code is a series of **state changes** sent to an implicit **context**
- Changes are made immediately!

- **Example**

English	OpenGLES
Enable lighting	glEnable(GL_LIGHTING);
Draw an object	glDrawArray(...)
Move left 2 units	glTranslatef(2,0,0)
Draw another object	glDrawArray(...)
Disable lighting	glDisable(GL_LIGHTING);
Set a color	glColor4f(0.5, 0.5, 1, 1)
Draw a final object	glDrawArray(...)

OpenGLES state

- There are a huge number of states that can be set
    - Look up the API docs for more info

- Lots of enable/disables (lighting, blending, fog, texturing)
    - **glEnable(GL_BLEND**)
- Current color
    - **glColor3f(1,0,1**)
- Current modelview matrix / projection matrix
    - **glLoadIdentity**()
- Blending modes
    - **glBlendFunc(GL_SRC_ALPHA**, **GL_ONE**)

- Note that you don't get an object and start modifying it
    - You just execute calls which affect a hidden implicit context

OpenGLES

- OpenGL and OpenGLES are C API's
  - wrappers exist for many other languages too

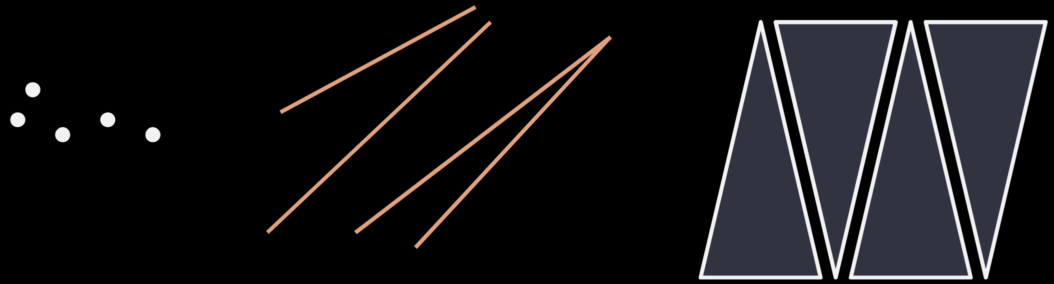
- No objects or object orientation
  - On the iPhone, for example, no use of Objective-C features in the API

- All OpenGLES functions begin **gl**
- All OpenGLES constants begin GL_

```
//Note use of gl* function name and GL_ constants
//OpenGL/OpenGLES constants are often very longwinded

glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

Drawing something

- OpenGLES can only draw three things:
  - points
  - lines
  - triangles



- Points and lines can just be **colored**
  - points can also have limited texturing (point sprites)

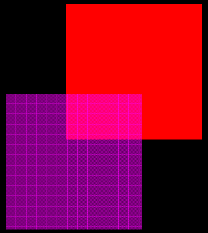
- Triangles can be **colored**, can be **lit** and have **textures** mapped on

- There are various ways in which the geometry can be simplified (e.g. lines and triangles often share vertices)

OpenGLES colors

- OpenGLES colors are always specified as RGB triples or RGBA triples
  - The "A" is alpha (transparency)

- Values range from 0.0 -- 1.0



- **glColor3f(1.0, 0.0, 0.0**) sets the current color to pure red, for example

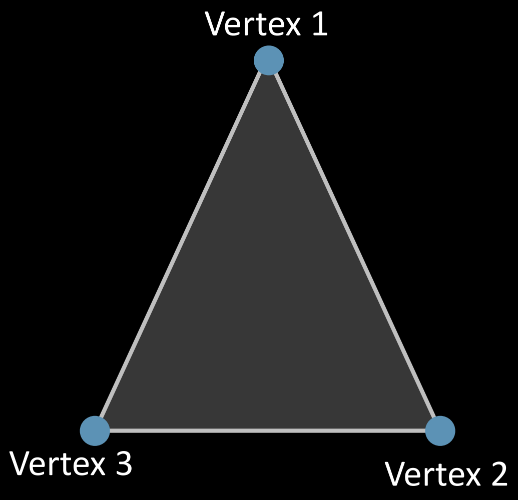
- **glColor4f(1.0, 0.0, 1.0, 0.5)** is semi transparent pink

- The default color is black!
  - Remember to set it, or you will never see anything

A Vertex

- In OpenGL, all primitives are constructed from **vertices**
- A **vertex** is a point on the primitive

- A vertex has:
  - A position (in 2 or 3 dimensional space) **(mandatory)**
  - A color (RGB or RGBA) (optional)
  - A normal (optional)
    - Defines the way light reflects at that point
  - A texture co-ordinate (optional)
    - Defines which part of a 2D texture is linked to that point



Vertex buffers

- OpenGLES requires that you store the vertices (points) making up primitives in advance
    - These arrays of vertices are known as vertex buffers

- Many geometric primitives can be drawn from a single buffer (e.g. hundreds of triangles in a single array)
    - Need only a single function call to push the data to the GPU
    - Commonly, one "model" (a game character, for example) will be stored in one buffer

- These arrays are just flat arrays of C floats
    - Must be 2 or 3 floats per vertex, depending on whether vertices are 3D
    - We'll always use 3D vertices for simplicty

```
// represents (1,5,7)
GLfloat vertices[3] = {1.0, 5.0, 7.0};

// represents (1,5,7), (10,10,10)
// Note that the structure is not represented in the array
GLfloat other_vertices[3] = {1.0, 5.0, 7.0, 10.0, 10.0, 10.0};
```

Colors in vertex buffers

- The minimum data required to render an primitive is the position of its vertices
- **Each vertex** can also be colored
    - **Note: not just each primitive!**

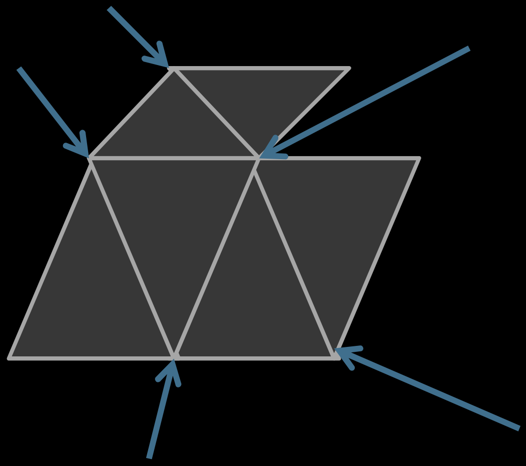
- Color arrays are just the same as vertex arrays
    - flat C arrays of floats
    - must have 4 components (either RGB or RGBA)
    - only linked to vertex positions by same ordering!

- You don't have to use color buffers
    - can just specify a drawing color which will appy to all primitives drawn until the next color is specified
    - but needed if you want per-primitive or per-vertex coloring

```
// Represents one vertex color (red, with one half transparency)
GLfloat colorBuffer[4] = {1.0, 0.0, 0.0, 0.5};
```

Indexed triangles

- A mesh of triangles usually share lots of vertices

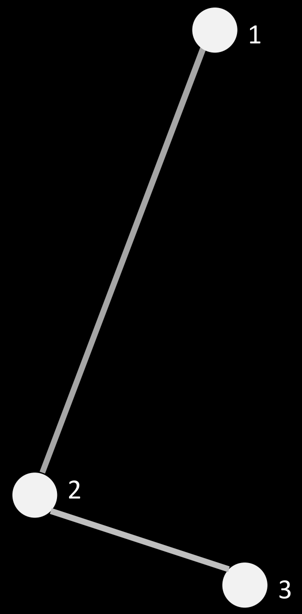


- OpenGLES uses indexed drawing to take advantage of this redundancy
    - You provide a list of vertices
    - Then for each triangle, list just the 3 indices of these vertices needed
    - Index list always has 3*(number of triangles) elements

- An index is an 8-bit or 16-bit integer
    - much smaller than a fully specified <x,y,z> floating point triple

Indexed lines

- The same applies to lines
  - Lines quite often (not as much as triangles) share points
- So you specify a list of vertices
  - and then a pair of vertex *indices* for each line
- Note that vertices specify position (at a minimum)
  - the can also specify color
    - and texture co-ordinates, and normals...



Drawing a line

- In OpenGLES there are four basic steps in drawing
  1. Create a simple C array for the vertex data
     1. create arrays for the indices if needed
  2. Enable the vertex arrays
  3. Set the current array pointer to your array from 1
  4. Tell OpenGLES to render

```
// Create a position array
GLfloat vertices = {0,50,0,  320,50,0};
//Create an index array
GLubyte indices = {0, 1};

// Enable the position vertex data
glEnableClientState(GL_VERTEX_ARRAYS);

// Set the pointer
// First parameter is number of elements in one position
// Here, 3 for XYZ
glVertexPointer(3, GL_FLOAT, 0, &(vertices[0]));

glDrawElements(GL_LINES, 2, GL_UNSIGNED_BYTE, &(indices[0]));
```

Drawing lots of lines (slow)

- We could draw lots of lines like this

```
// Set up all the data here...
// Assume vertices is a list of vertices giving line pairs

// Draw the lines
for(int i=0;i<nLines;i++)
{
          // move forward two vertices for each line
          glVertexPointer(3, GL_FLOAT, 0, &(vertices[0+2*i]));
          glDrawElements(GL_LINES, 2, GL_UNSIGNED_BYTE, &(indices[0]));
}
```

Drawing lots of lines (fast!)

- Every call to **glDrawArrays/glDrawElements** actually copies data to the GPU
    - glVertexPointer etc. doesn't actually *do* anything
        - it just tell OpenGLES where to copy from when the draw command comes

- It's much more efficient to make one call to **glDrawElements**

```
// Set up all the data here...
// Assume vertices is a list of vertices giving line pairs
// NOW: indices must have 2*nLines elements
// specifying the start and end indices of each line in vertices

glVertexPointer(3, GL_FLOAT, 0, &(vertices[0]));
glDrawElements(GL_LINES, 2*nLines, GL_UNSIGNED_BYTE, &(indices[0]));
```

Drawing a triangle

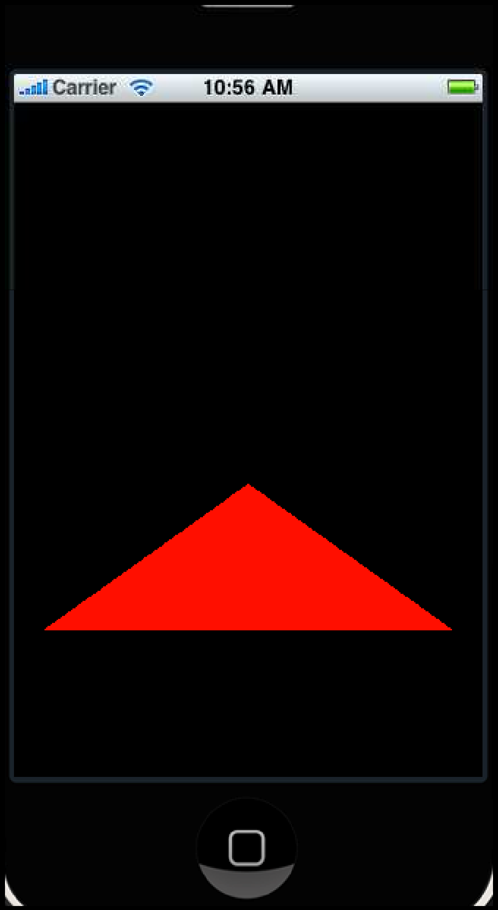
- The minimum to draw an *indexed* triangle:
  - Specify the vertices (as an array of floats)
  - Specify the indices (as as an array of chars or shorts -- you choose which!)
  - Specify a color
  - Tell OpenGLES where the vertices are
  - Enable vertex arrays
  - Request that the triangle(s) be drawn

```
// Three vertices * 3 components = 9
// Z is always zero because we are drawing in 2D
GLfloat triangle[9] = {200,100,0,  160,200,0,  300,100,0};
GLubyte triangleIndices = {0, 1, 2};

glColor4f(1,0,0,1); // Red, no transparency
// 3 components per vertex
glVertexPointer(3, GL_FLOAT, 0, &(triangle[0]));
// Enable vertex array drawing
glEnableClientState(GL_VERTEX_ARRAY);

// Draw three indices worth
glDrawElements(GL_TRIANGLES, 3, GL_UNSIGNED_BYTE, &(triangleIndices[0]));
```

Solid coloring the triangle

- Color of the triangle can be specified per vertex as well
  - This code will do the same as the previous

```
// Three vertices * 3 components = 9
// Z is always zero because we are drawing in 2D
GLfloat triangle[9] = {200,100,0,  160,200,0,  300,100,0};
// RGBA, RGBA ...
GLfloat colors[9] = {1.0,1.0,0.0,1.0,  1.0,1.0,0.0,1.0,
1.0,1.0,0.0,1.0};
GLubyte triangleIndices = {0, 1, 2};

glVertexPointer(3, GL_FLOAT, 0, &(triangle[0]));
glColorPointer(4, GL_FLOAT, 0, &(colors[0]));
// Enable vertex array drawing
glEnableClientState(GL_VERTEX_ARRAY);
// Enable color array
glEnableClientState(GL_COLOR_ARRAY);

// Draw three indices worth
glDrawElements(GL_TRIANGLES, 3, GL_UNSIGNED_BYTE,
               &(triangleIndices[0]));
```



Smooth coloring the triangle

- If each vertex color is different, OpenGLES automatically **interpolates** between the colors

```
GLfloat triangle[9] = {200,100,0,  160,200,0,  300,100,0};
// RGBA, RGBA ...
// Now yellow, blue, red at the vertices
GLfloat colors[9] = {1.0,1.0,0.0,1.0,  0.0,0.0,1.0,1.0,
                     1.0,0.0,0.0,1.0};
GLubyte triangleIndices = {0, 1, 2};

glVertexPointer(3, GL_FLOAT, 0, &(triangle[0]));
// Note first parameter is 4 because we are
// using 4-component colors
glColorPointer(4, GL_FLOAT, 0, &(colors[0]));
// Enable vertex array drawing
glEnableClientState(GL_VERTEX_ARRAY);
// Enable color array
glEnableClientState(GL_COLOR_ARRAY);

// Draw three indices worth
glDrawElements(GL_TRIANGLES, 3, GL_UNSIGNED_BYTE,
               &(triangleIndices[0]));
```

Projections, view

- OpenGLES translates from world space to screen co-ordinates
  - you draw in (an arbitrary) co-ordinate system
  - map to screen co-ordinates via a series of **matrices**

- **ModelView** matrix transforms local coordinates to global coordinates (e.g. represents camera location)

- **Projection** matrix transforms these coordinates to normalized 2D coordinates
  - in perspective, this involves a perspective divide
  - makes far away points closer together

- Viewport transforms normalized 2D coordinates
  - Just scales coordinates to fit pixel draw window

- **We will only cover the simple orthographic 2D (straight on) display**

Matrices

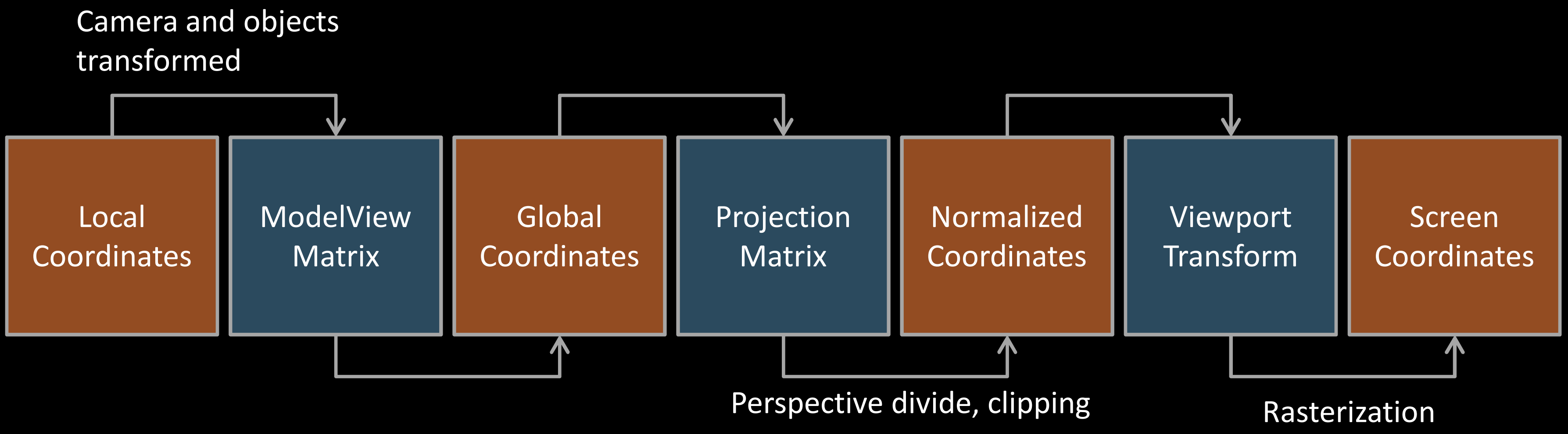
- Matrices *just represent transforms compactly*

- A single 4x4 OpenGL matrix can represent any combination of:
  - 3D translations (movement)
  - 3D rotations
  - 3D scaling (including non-uniform)
  - 3D shearing (very rare!)

- **Matrices can be composed by multiplication!**
  - i.e. the product of two matrices results in the composition of the transforms
  - (rotation1 * scale1) creates a matrix which rotates by rotation1 and then scales by scale1

- OpenGL provides useful functions for rotating, translating and scaling which implicitly create matrices for you
  - You will not have to work with matrices explictly

View transform process



Camera and objects transformed

Local Coordinates

ModelView Matrix

Global Coordinates

Projection Matrix

Normalized Coordinates

Viewport Transform

Screen Coordinates

Perspective divide, clipping

Rasterization

Normalized coordinates

- OpenGL always considers the screen to extend from (-1,-1) to (1,1)
  - **Note well: OpenGL's y coordinate starts at the bottom of the screen!**
  - **This is not the way conventional graphics systems work**

- **In an orthogonal view, the projection matrix just rescales coordinates**
  - **(0,0) ,(screen_width, screen_height) --> (-1, -1), (1,1)**

- Normally you will set a projection matrix and a viewport **once**

- **ModelView matrix is constantly changed to lay out objects in the world**

Common OpenGLES structure

- Initialise (once)
  - Set viewport
  - Set projection
  - Set drawing states (lighting enabled, fog enabled...)

- Every frame
  - Clear the screen
  - Reset modelview matrix
  - Set the camera position (if camera moves)
  - For each object:
    - store the modelview matrix
    - transform to the objects location/scale/rotation
    - draw the object
    - restore the modelview

Clearing the screen

- To clear the screen
  - set the clear color
  - clear the color buffer

```
glClearColor(0,0,0,1); // Clear to black
glClear(GL_COLOR_BUFFER_BIT); // clear the color buffer
```

- There are other buffers you can clear (depth buffer in particular)
  - but for 2D drawing, only the color buffer is likely to be important

Setting up a viewport

- The viewport is specified in *pixel space*
  - It specifies a region of pixels to draw into
  - OpenGLES code never needs to know about actual onscreen pixel sizes
  - **glViewport** is how mapping from normalized coordinates to pixels is done
- Usually it is just set to the entire device size

```
glViewport(0,0,screen_width,screen_height);
```

- But you can specify other regions, for example for split screen displays
  - set left hand viewport, draw, set right hand viewport, draw
  - your draw functions are completely unchanged!

```
// Left hand side
glViewport(0,0,screen_width/2,screen_height);
doDrawSomeStuff();

// Right hand side
glViewport(screen_width/2,0,screen_width,height);
doDrawSomeOtherStuff()
```

Setting the basic OpenGL state

- At a minimum an OpenGLES initialisation routine must set a viewport
- A projection of some kind is usually set
  - Perspective for 3D, orthographic for 2D

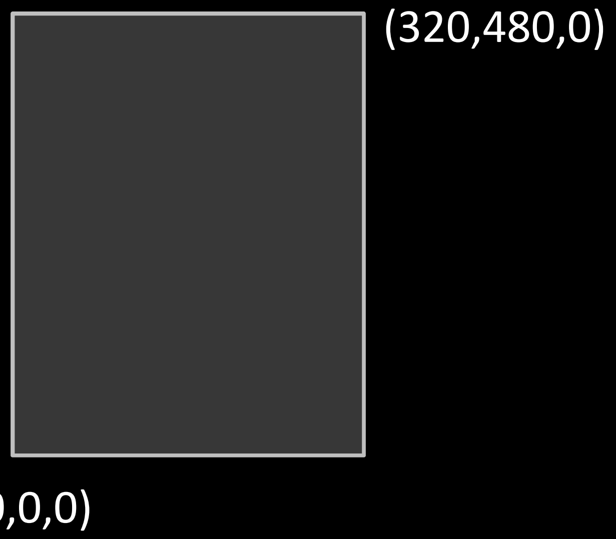
- The various enable/disables for features used are given

- Specific implementations may require other setup
  - creating color buffers, binding them etc...
  - Usually there will be boilerplate for you
    - (e.g. XCode generates all the boilerplate to get a simple drawing going)

Camera and Projection

- We will only use a simple orthographic projection

- This emulates a 2D display with a coordinate system from (0,0) to (screen_width, screen_height)
  - Camera is effectively "straight on" to the screen



Orthographic Projection

- To set the orthographic projection
  - set the matrix mode to GL_PROJECTION
  - clear the projection matrix
  - use glOrthof to set the extent of the view
  - set the matrix mode back to GL_MODELVIEW

```
// set the matrix mode to work with projection matrices
glMatrixMode(GL_PROJECTION);
glLoadIdentity(); // clear the matrix

// arguments are left, right, bottom, top, near z and far z
glOrthof(0, 320, 0, 480, -1, 1);

// go back to working with the modelview matrix
glMatrixMode(GL_MODELVIEW);
```

Modelview matrix

- We can easily transform things in OpenGLES by changing the modelview matrix
  - we do **not** change each of the vertices of the object!

- **glTranslatef(x,y,z)** moves everything by x,y,z
- **glRotatef(angle**, **xaxis**, **yaxis**, **zaxis**) rotates by angle *about* xaxis, yaxis, zaxis
  - 3D rotations are tricky!
  - **glRotatef(angle, 0, 0, 1)** does 2D rotation for our purposes
- **glScalef(x,y,z)** scales everything by x,y,z

- **These transforms apply to everything drawn after that point**

- **Transforms are order dependent**
  - scale then translate is different than translate then scale!
  - transforms are applied in reverse order to the way they are written

Transforms

- If you want to make a unit sized square 32 units across and move it 10 units left

```
glTranslatef(-10, 0, 0);
glScalef(32, 32, 1);
```

- If you do this:

```
glScalef(32, 32, 1);
glTranslatef(-10, 0, 0);
```

- You will make it 32 units across and move it *320* units left!
  - likely nothing will appear at all!

- Never scale any axis by zero!
  - the results might be very strange
  - don't do **glScalef(32,32,0)**, even if you're not using the z component

Transforms

- Each transform actually multiplies the current matrix (usually modelview) by a matrix for the transform

- **glLoadIdentity()** loads the identity matrix into the current matrix
  - **i.e. resets it completely**

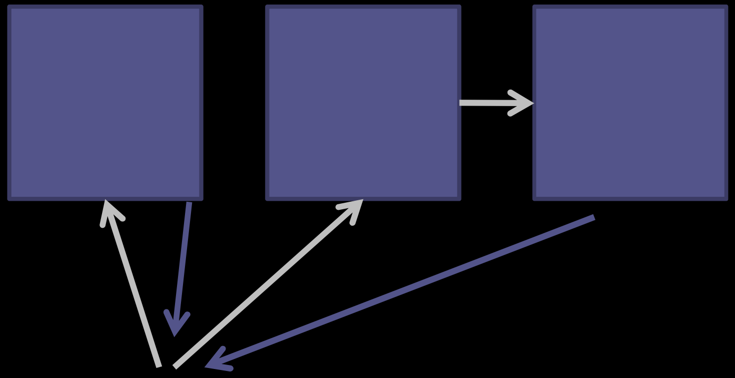
- **Summary:**
  - To move or transform something in OpenGLES, multiply the modelview matrix by a transform, *then* draw your object
  - Do not manually transform vertices!
  - Change the "camera" position by setting the modelview matrix *before* drawing anything

Pushing and popping matrices

- It is very common to want to transform one object to a location, then another to another position and so on
    - But when we apply **glTranslatef** etc., the modelview matrix is changed from then on

- OpenGLES provides a matrix stack
    - the state of the matrix can be preserved and restored
    - **glPushMatrix** stores the current transform
    - **glPopMatrix** restores it

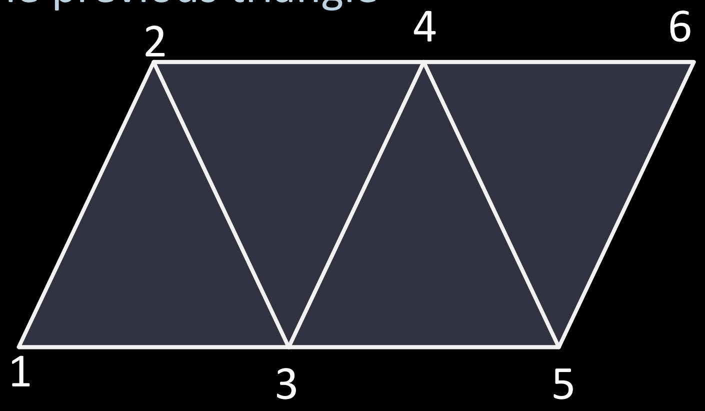
- This means you can draw objects relative to each other in a hierarchical manner



- Push, draw on object, pop
    - [**push**, draw an motorcycle, [**push**, draw a wheel, **pop]**, [**push**, draw a wheel, **pop]**, **pop]**

Drawing a strip of triangles

- OpenGLES very often uses **triangle strips**
  - triangles which all share an edge with the previous triangle
  - each new triangle only needs 1 vertex!
  - this is very efficient

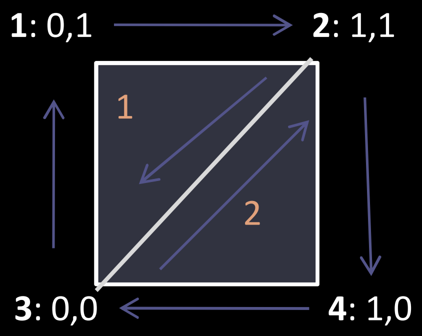


- Note that now you don't always need indices
  - The vertices are already ordered
  - You can still use indexed drawing if you want to order the array differently

  - There are also **triangle fans**, where each triangle shares a common point and an edge with the previous triangle

Drawing a square with strips

- Squares are made up of two triangles
  - One common edge
  - Triangle strips mean need to only specify 4 vertices
    - instead of redundant 6 for naive triangles
- Squares are commonly used for drawing flat images



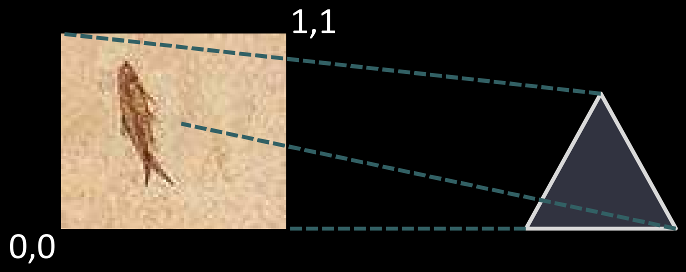
```
// Set up the arrays
GLfloat vertices =      {0,1,0,   1,1,0,  0,0,0  1,0,0};

// Enable the array pointers
glEnableClientState(GL_VERTEX_ARRAY);

// Set the array pointers
glVertexPointer(3, GL_FLOAT, 0, vertices);

// Draw the strip
glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
```

Texturing Basics

- OpenGLES supports texturing
  - An image is stretched across triangles so as to simulate a texture

- To use textures you need
  - an image representing the texture
  - a way of mapping the texture to the primitives

- **Texture coordinates** tell OpenGLES how to map a 2D image onto triangles
  - **Texture coordinates always go from (0,0) to (1,1)**
  - Each vertex of a primitive can specify a texture coordinate



Using textures

- In OpenGLES textures are part of the hidden context like everything else

- You manipulate them using a **name**
  - In OpenGLES a **name** is just an integer which uniquely identifies an object

- When a new texture is created, first generate a new name
  - This does not allocate any space or load anything -- it just generates an ID!

```
int newTexture;
glGenTextures(1, &newTexture);
```

- When modifying or using the texture, you must **bind** it
  - this makes it the "current" active texture

```
glBindTexture(GL_TEXTURE_2D, newTexture);
```

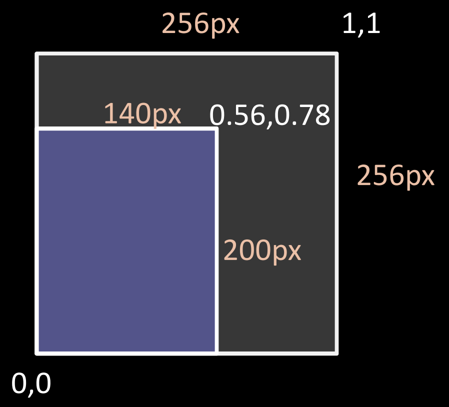
- All future drawing operations or texture modifiers will work on this texture

Using Textures

- Textures are only used if texturing is enabled
  - Otherwise primitives will be drawn in solid colors
  - Must set this before executing a draw command

```
glEnable(GL_TEXTURE_2D); // enable texturing
```

- **OpenGLES textures must have sizes which are powers of 2**
  - e.g. 64x64 or 512x256
  - Do not have to be *equal* powers of 2
  - Maximum size is often 1024x1024

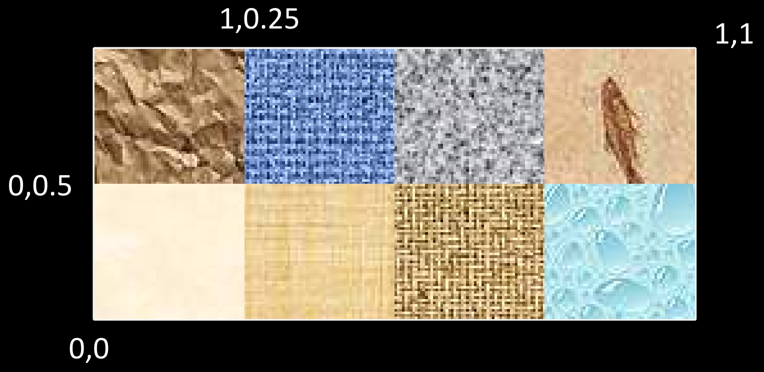


- If you want to use a texture smaller than this, you just create a slightly larger texture with a blank border
  - Then use coordinates which map to a the subsection where your texture is

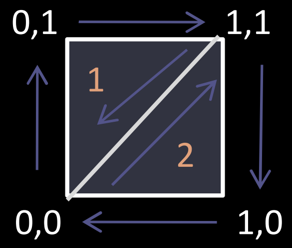
Texture Atlas

- If many textures must be drawn, it is very inefficient to load a large number of separate textures
  - You would have to draw one primitive, bind a new texture, draw another primitive etc.

- A texture atlas is just a number of textures on a grid on a texture
  - Select texture just be setting coordinates
  - Map different parts of a model to different textures



Drawing a textured square



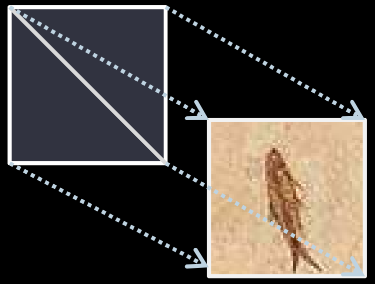
- OpenGLES doesn't support drawing squares or quads
  - two triangles will do though
  - triangle strips make this easy

- Specify vertice positions and texture coordinates in same order

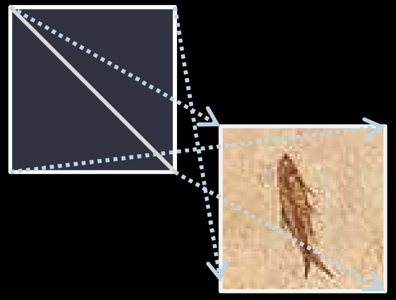
```
// Set up the arrays
GLfloat textureCoords = {0,1,     1,1,    0,0,   1,0};
GLfloat vertices =      {0,1,0,   1,1,0,  0,0,0  1,0,0};
// Bind the texture
glBindTexture{GL_TEXTURE_2D, textureName);
glEnable(GL_TEXTURE_2D);
// Enable the array pointers
glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_TEXTURE_COORD_ARRAY);
// Set the array pointers
glVertexPointer(3, GL_FLOAT, 0, vertices);
glTexCoordPointer(2, GL_FLOAT, 0, textureCoords);
// Draw the strip
glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
```

Vertex ordering

- Note: ordering of vertices is important!
    - texture will be twisted if you specify it twisted

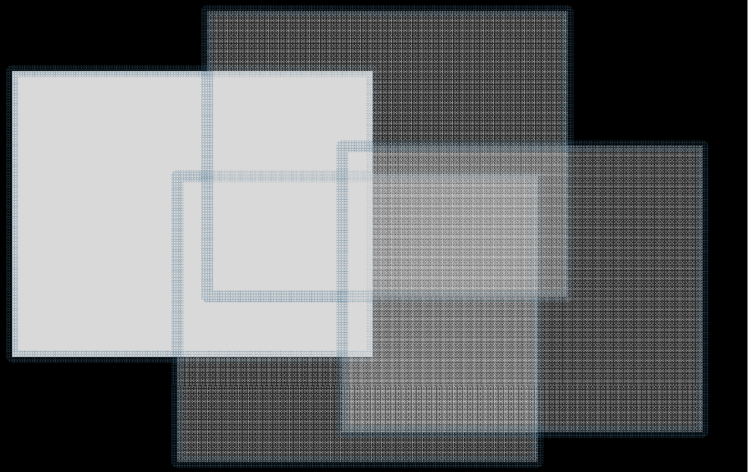


Correct!



Wrong!

The alpha channel



- OpenGL natively supports blending
  - i.e. transparency
- You must enable it

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

  - the blending function allows lots of effects
  - the default one above just gives standard transparency
- Every color has a fourth component for transparency
  - **glColor4f(r,g,b,a**) sets a 4 component color
- Textures can have **alpha channels**
  - just stores the transparency along with the rgb components
  - e.g. for putting masks around sprites

Fading and tinting textures

- Note that textures and colors are automatically combined by OpenGLES
  - If the current color is white, the texture is rendered as is
  - If the current color is non-white, the color is multiplied by the texture
    - by default, at least
- Textures can therefore be tinted (or made transparent) at runtime just by changing the color
  - If the color is pure red, only the red component of the texture will be shown

```
glColor4f(1,1,1,1); // solid white
//... draw something
// will appear as normal

glColor4f(1,1,1,0.5); // white, alpha = 0.5 (half transparent)
//... draw something
// will appear semi-transparent

glColor4f(0.5,1,0.5,1); // solid light green
//... draw something
// will appear with a green tint
```

GLES 2.0: end of the pipeline

- In OpenGLES 2.0 there is no more fixed pipeline
- No transformation commands
    - glTranslatef, glRotatef, glPushMatrix...

- No lighting commands
    - glMaterialfv, glLightModel

- Everything is written in shaders
    - Small fragments of code run on GPU
    - Vertex shaders transform vertices (e.g. applying rotations, translations or distortions)
    - Pixel shaders describe how polygons are rasterized (how to color a pixel given geometry)

- Very flexible -- procedural texturing, new lighting models, special effects
    - But more work
        - Just getting an image on the screen takes a lot more code
        - You have to compute all matrices etc. yourself and pass them to the shaders!

GLES 2.0: end of the pipeline

- New devices will move to OpenGLES 2.0
    - iPhone 3.0+ uses 2.0
    - Provides compatibiltiy with 1.1 by emulating fixed pipeline

- Try creating an empty OpenGLES project in XCode
    - Look at the ES2Renderer.m for an example of GLES 2.0 code