ES3 Lab 3

OpenGLES

This lab

- Creating an OpenGLES project
- Drawing simple triangles
- Drawing a quad
- Loading textures
- Drawing many OpenGLES objects

Outline of steps

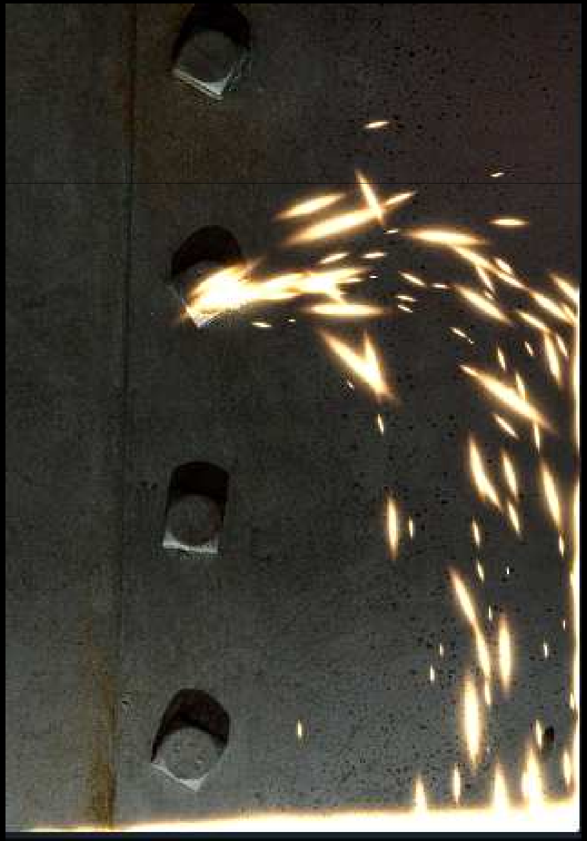
- Create a blank project
- Make it OpenGLES 1.1
- Strip out the default rendering code

- Draw a simple triangle
- Load a texture
- Draw it as a background

- Create a simple "particle"
- Make it move in random direction
- Make many particles spawn from finger location
- Give particles a limited lifetime
- Add gravity, wall deflection

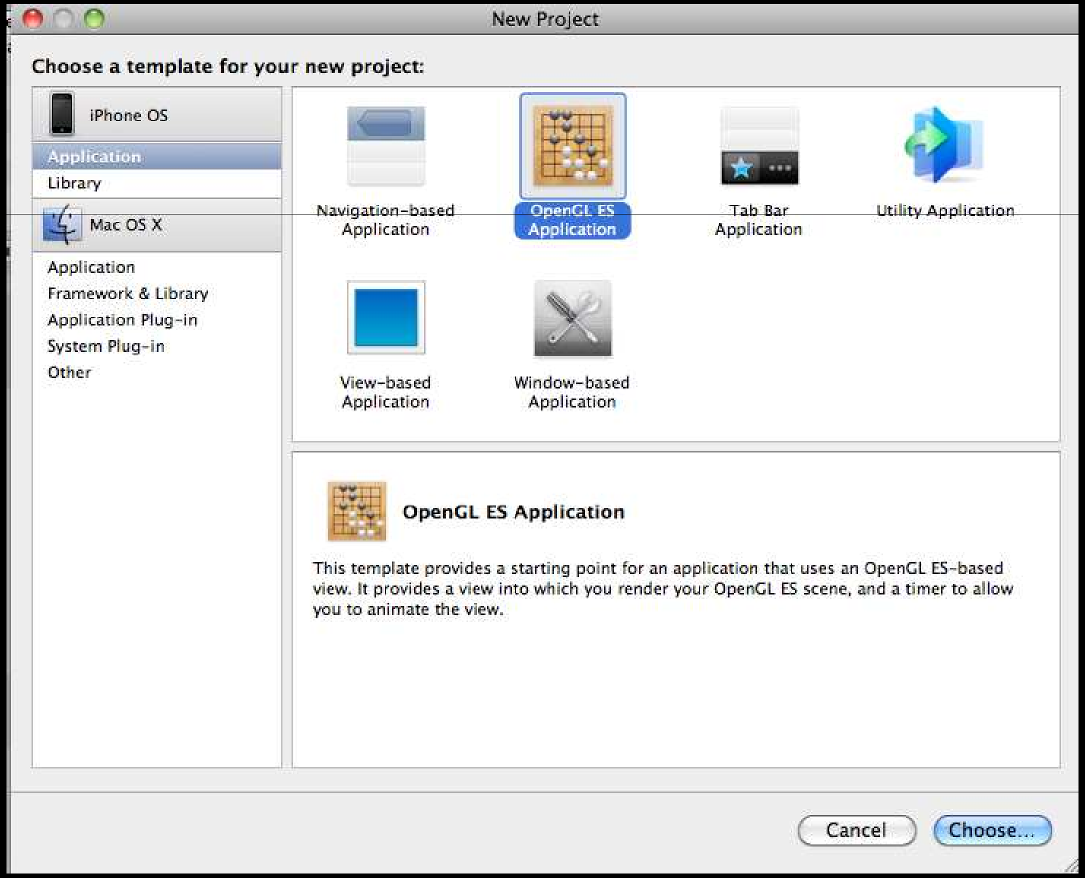
Result

- A particle system which shoots "sparks" from the finger



Creating an OpenGLES project

- Create a new OpenGLES based application in XCode



OpenGL ES Application

This template provides a starting point for an application that uses an OpenGL ES-based view. It provides a view into which you render your OpenGL ES scene, and a timer to allow you to animate the view.

Project structure

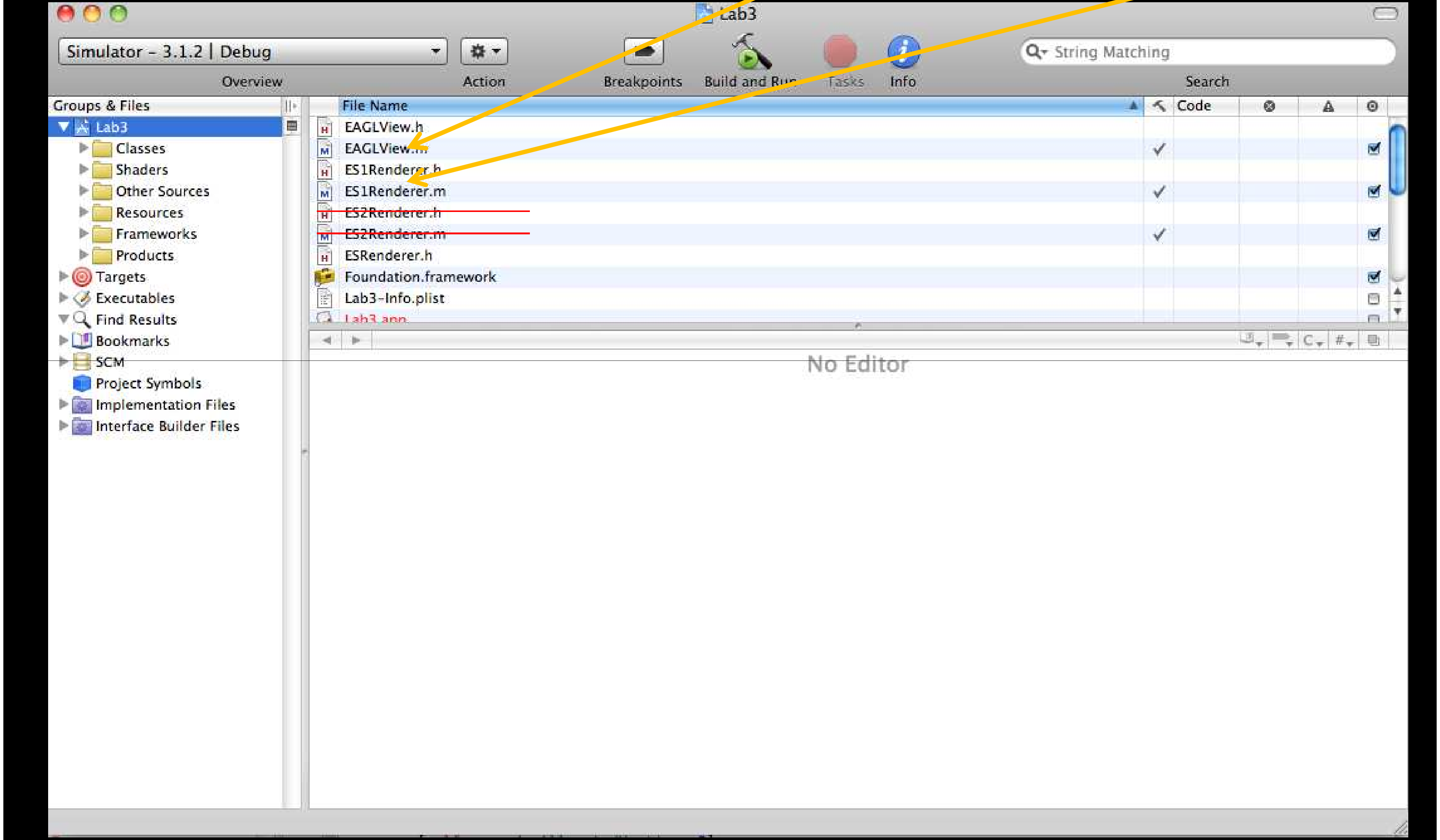
- Note the structure of the project
  - **EAGLView.m** defines a subclass of a control which provides an OpenGLES context
  - i.e. can draw **OpenGLES** in a control

- **ES1Renderer** actually defines a skeleton block of code for OpenGLES 1.1
  - initialise and draw callback

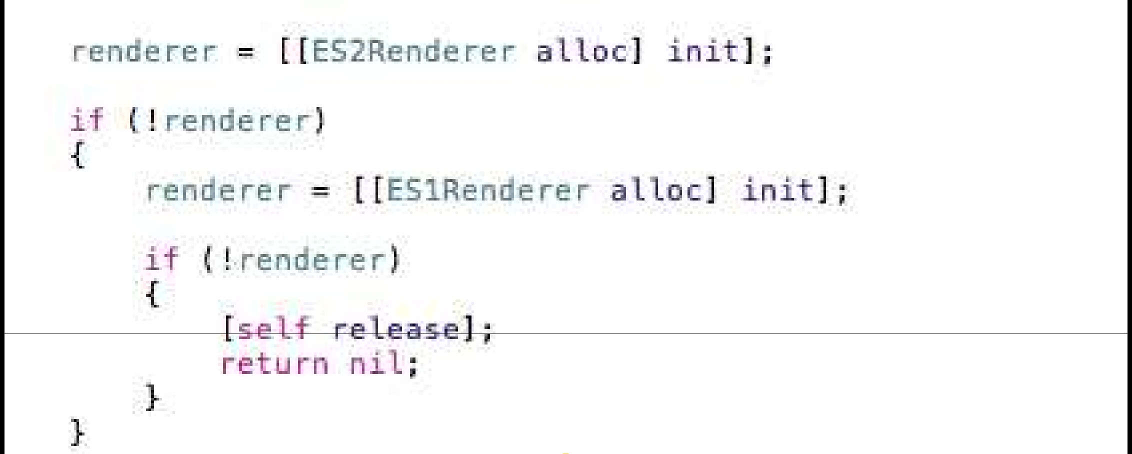
- **ES2Renderer** defines a OpenGLES 2.0 skeleton
  - **We don't want this**
  - In EAGLView.m change the code so that only **ES1Renderer** is used
  - Delete **ES2Renderer** from the project!
  - You will need to change **ESRenderer** as well to eliminate references to **ES2Renderer.h**
- **Build it and check that the default bouncing square appears**

View subclass

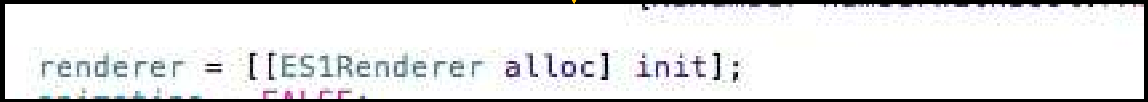
OpenGLES Code



File Name		Code			
EAGLView.h					
EAGLView.m	✓				☑
ES1Renderer.h					
ES1Renderer.m	✓				☑
~~ES2Renderer.h~~					
~~ES2Renderer.m~~	✓				☑
ESRenderer.h					
Foundation.framework					☑
Lab3-Info.plist					□
Lab3.app					□



```
renderer = [[ES2Renderer alloc] init];

if (!renderer)
{
    renderer = [[ES1Renderer alloc] init];

    if (!renderer)
    {
        [self release];
        return nil;
    }
}
```



```
renderer = [[ES1Renderer alloc] init];
```

Project Structure

- Note: **EAGLView** is the control
  - it receives events and is instantiated in the app delegate
    - added to the main **UIWindow**
  - if you wanted to manipulate touch events, you would override **touchesBegan** etc. in **EAGLView**
  - the default project creates a fullscreen control

- **ES1Renderer** is where OpenGLES drawing commands go
  - **EAGLView** will call **render** in **ES1Renderer** when the control needs redrawn
  - This will be called regularly (e.g. at 60FPS)

- The initial **ES1Renderer** has a lot of setup and other stuff in it
  - The key place for rendering is **render**

Cleaning up render

- Remove everything that's currently in **render** and replace it with the following blank skeleton

```
- (void) render
{
     [EAGLContext setCurrentContext:context];
     glBindFramebufferOES(GL_FRAMEBUFFER_OES, defaultFramebuffer);

     // Render stuff will go here!

     glBindRenderbufferOES(GL_RENDERBUFFER_OES, colorRenderbuffer);
     [context presentRenderbuffer:GL_RENDERBUFFER_OES];
}
```

Clearing the screen

- We need to clear the screen

- Set the clear color using **glClearColor(r,g,b,a)**
- Clear using **glClear(GL_COLOR_BUFFER_BIT)**
  - Insert these Immediately after **glBindFramebufferOES(...)**
  - clearing should happen before anything else

- Choose an interesting color for the clear color

- **Build, check that the screen goes to the color you set!**

Setting the projection

- We need to set the *projection matrix*

- We will use an orthographic perspective which emulates pixel coordinates
- Set the matrix mode to **GL_PROJECTION** and then reset it with **glLoadIdentity**()

```
glClear(GL_COLOR_BUFFER_BIT);

glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
```

- Remember to set the matrix mode back to model view and call **glLoadIdentity()**!

Adding an orthographic projection

- After **glMatrixMode(GL_PROJECTION**), **glLoadIdentity**() add a call to **glOrthof**
  - **This sets the projection matrix to orthographic**

- Note: it must go after the **glLoadIdentity()**, and before the matrix mode is set back to **GL_MODELVIEW**!

- The parameters are the left, right, bottom and top extents, and the z range
  - **The z range is effectively unimportant**
  - **We will always draw at z=0**

- Left should be 0, right should be **backingWidth** (size of the screen)
- Bottom should be 0, top should be **backingHeight**
- zNear, zFar should be -1, 1
  - this includes the region at z=0 where we will draw

Drawing a triangle

- Add a triangle (using indexed drawing)
  - Create an array (of GLfloat) for the vertex positions (must have 9 elements!)
    - positions are in screen coordinates
  - Create an array of (GLubyte) for the vertex indices

```
GLfloat trianglePositions[9] = {... // 9 floats x1,y1,z1,x2,y2,z2,x3,y3,z3
// z should be zero for all
GLubyte triangleIndices[3] = {0, 1, 2}; // use first three vertices
```

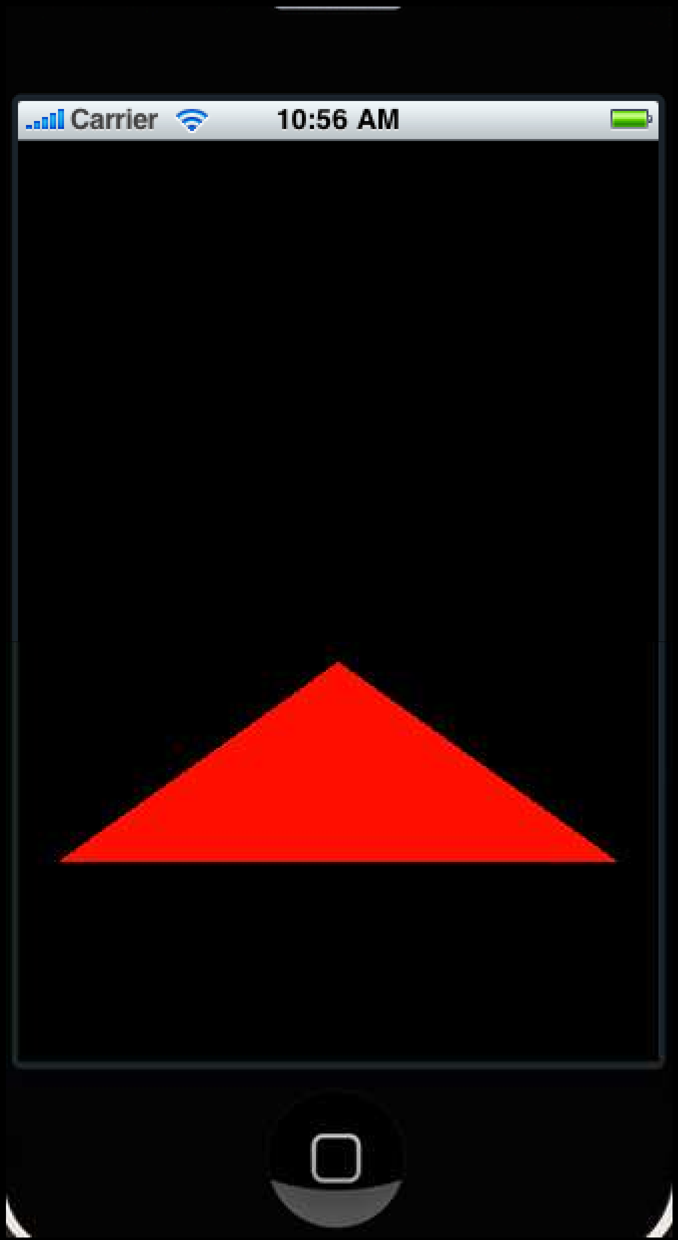
  - Set the color using **glColor4f**

```
glColor4f(1,0,0,1); // red (choose your own color!)
```

  - Enable vertex arrays
  - Set the vertex pointer
  - Call **glDrawElements**

```
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, 0, &(trianglePositions[0]));
glDrawElements(GL_TRIANGLES, 3, GL_UNSIGNED_BYTE, &(triangleIndices[0]));
```

  - **Build, check a triangle actually appears!**


Carrier
10:56 AM

Loading a texture

- In the lab zip file, there is are **Utils.m** and **Utils.h**
  - Add these to your project
  - Import **Utils.h** in **EAGLView.h** and **ES1Renderer.h**

- **Utils** provides an PNG image loading function called **loadTexture()**
  - Have a look at this function
  - It has a lot of boilerplate, but it basically just loads an image and converts it to a plain array of RGBA floats and passes this to OpenGL

- It takes a string for the filename (minus the extension!) and returns a texture **name**
  - This is just an integer

- The other two parameters write the width and height into the passed pointer
- **Note: you must add the CoreGraphics framework to the frameworks to make this code compile!**

Loading the background image

- Add **background.png** to the project

- Add a member variable for the background image to **ES1Renderer** (of type **GLuint)**
  - in **init** load the texture:

```
int w,h; // we don't use these, but we need to pass something
backgroundTexture = loadTexture(@"background", &w, &h);
```

- Note: **background.png** is 512x512
  - This is because OpenGLES textures must have widths and heights which are powers of 2
  - The image is actually 320x480 with a border around it

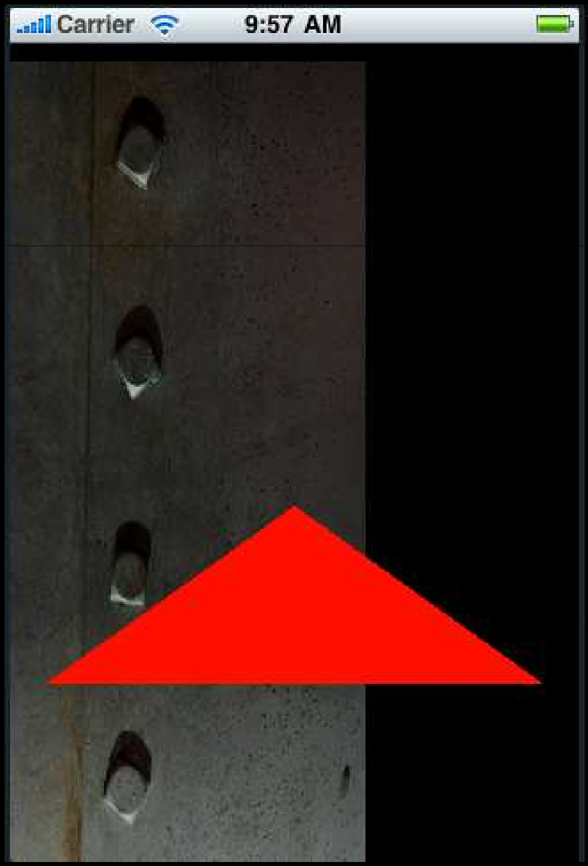
Drawing the background

- Add a **drawBackground** method to ES1Renderer
  - Call it from **render**, before the triangle drawing
- Here we need to draw a textured quad

```
glEnable(GL_TEXTURE_2D);              // Enable texturing
glEnableClientState(GL_VERTEX_ARRAY); // Enable the right arrays
glDisableClientState(GL_NORMAL_ARRAY);
glDisableClientState(GL_COLOR_ARRAY);
glEnableClientState(GL_TEXTURE_COORD_ARRAY);

GLfloat texCoords[] = {0, 1,   1, 1,     0, 0,      1, 0};
GLfloat vertices = {0, 1, 0,   1, 1, 0,  0, 0, 0,   1, 0, 0};

glPushMatrix();                       // Store modelview matrix

glScalef(320, 480, 1);                // map (0,0),(1,1) to (0,0),(320,480)
glColor4f(1,1,1,1);                   // white color
glBindTexture(GL_TEXTURE_2D, backgroundTexture); // set the current texture
glVertexPointer(3, GL_FLOAT, 0, vertices);       // set the pointers
glTexCoordPointer(2, GL_FLOAT, 0, texCoords);
glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);

glPopMatrix();                                // Restore modelview matrix
glDisable(GL_TEXTURE_2D);                     // Important: disable texturing again!
```

Test it!

- Build this, run it.
- It should look like the following:

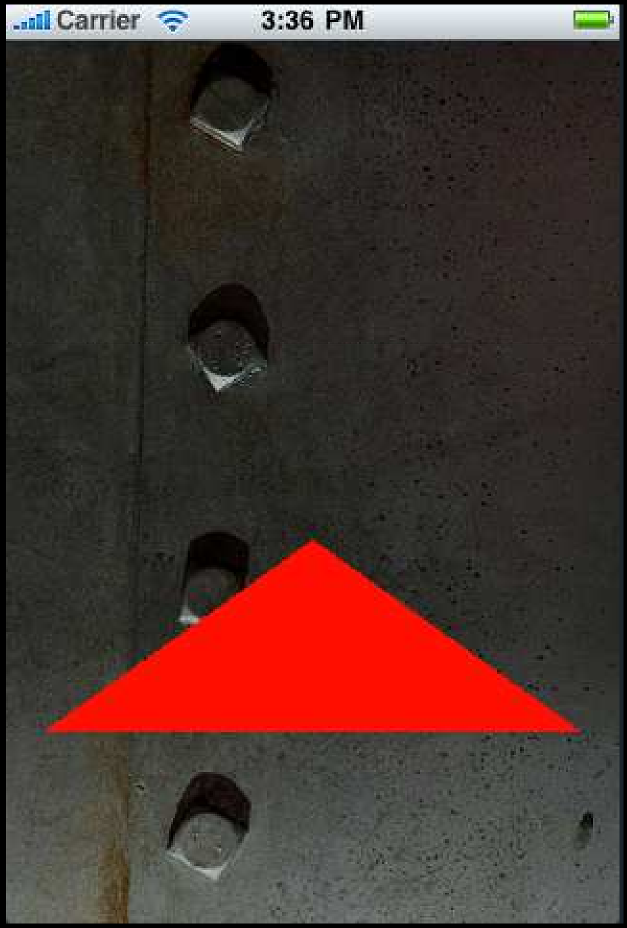


The size is wrong

- This doesn't look right
  - we mapped the *whole* 512x512 texture to the screen
  - *including the border!*
  - everything is very stretched out

- To fix this, set the texture coordinates to only cover the region we are interested in
  - Hint: work out what fraction 320/512 is (and 480/512) and use that in the texture coordinates (not the vertex positions!)

- **Check that it now looks correct**



Carrier
3:36 PM

Particle System

- A particle system just simulates very simple physics on a bunch of points
  - Commonly used in games for effects like fire, smoke, fog, plasma etc.

- Each particle has (at least) a position, a velocity and a lifetime

  - At each redraw, the position of each particle is updated according to the particle physics
    - can be as simple as just move by the current velocity

  - Each particle is drawn on the screen at its current position

  - Particles are randomly generated and are removed after a certain time
    - i.e. they have a lifetime and then "die"

Mr. Sparky

- To create the spark effect, we will use a simple *particle system*
  - Each particle will be a textured quad

- We maintain a list of these particles
  - update their movement every frame
  - draw a textured quad at their new position

- In **EAGLView** add a mutable array instance variable to hold the list of particles

- Create a class **Particle** to represent a particle
  - i.e. representing the position and state of a particle

- Just create an subclass of **NSObject** called **Particle**
  - It needs an x and y position (floats) at a minimum
  - Remember to add properties for the x and y position

Particle class

- Add an array variable to **ES1Renderer** to hold a reference to the particle array in **EAGLView**
  - Add a property for it!

- In **EAGLView** initialise the array to be empty in the **init** method
  - Then set the particle array in the **ES1Renderer** to this array

    ```
    self.particleList = [NSMutableArray arrayWithCapacity:500];
    [self.renderer setParticleList:self.particleList];
    ```

  - Note that you have to call **setParticleList** explicitly

Drawing the sprite

- Add a texture name to **ES1Renderer**, as you did for the background
- This time, load **spark.png** (also in the lab zip file)
- Add a method **drawParticle** to ES1Renderer
  - taking one argument, an instance of the Particle class
- In **render** (in ES1Renderer), iterate through the particle list and call **drawRender** on each particle

Drawing the sprite

- in **drawParticle**
  - Draw a textured quad, exactly as in the background drawing
  - Bind the spark texture instead
  - Add a translate to the position given by the particle class
    - use **glTranslatef(x,y,0)**
    - translate *before* scaling
  - Instead of scaling to 320x480, scale to 32x32 instead
  - Remember: Push the matrix, transform, draw, pop the matrix
    - exactly as in the background drawing example
- In the **EAGLView init** method, create a new instance of **Particle**
  - put it into the array
  - set its x and y to something like 160, 240 (middle of the screen)
- **Build, run, check that the particle appears!**

Making it move

- Now the single sprite is visible we can move it

- To move a sprite, translate it by a different amount each frame
  - Give the sprite a velocity
    - Add **dx** and **dy** as variables to the **Particle** class
    - Add a method **update**

- In **update** just do x+=dx, y+=dy

- In **EAGLView**, add an **updateParticles** method and call it from the drawNow function
  - Iterate through the particles and call **update** on each

- When you create the particle object, remember to set **dx** and **dy** to sensible values
  - choose small values like 0.1 to start
- The particle should move!

Particle lifetimes + replacement

- Particles shouldn't last forever

- Add a **lifetime** variable to the particle class
  - Make it start at some maximum age (e.g. 40) when it it initialised
  - Decrement it by one in every **update** cycle

- Now, in **EAGLView's updateParticles** method, look through all particles and check for any with lifetime<=0
  - Remove these from the list of particles
    - Note: to do this, place all the expired particles in a "kill list"
    - Then iterate through the kill list and remove all those particles from the main list
    - If you try and remove things directly while iterating through the particle list, you will cause an error

Color by lifetime

- Particles should fade out as they get "older"

- Color particles by their age
  - brightness = currentAge/maximumAge
  - ==1 when particles are generated
  - ==0 when particles are about to be removed

- Set brightness by setting the color before drawing the particle
  - use **glColor4f** and set the brightness using the alpha component
  - other components should be 1

- Now the particle should fade out and disappear after a while

Drawing lots of particles

- One particle isn't very exciting
  - In **updateParticles** randomly add new particles

```
// r will be 0-4 random value
int r = arc4random() % 5;

for(int i=0;i<r;i++)
{
  // create new instance of particle
  // add it to particleList
}
```

- **Utils.m** has a function **generateGaussian**
  - this generates a normal random number centered around zero

- Use this to set the velocity of the particle (dx and dy)
  - Add a constant on to the random value so the particles "spray off" in a definite direction

- Check this works!

Setting the blend mode

- We want the particles to "add together"
  - OpenGLES supports this natively

- Before drawing the particles
  - enable blending
    - **glEnable(GL_BLEND)**
  - set the additive blending mode
    - **glBlendFunc(GL_SRC_ALPHA, GL_ONE)**
    - this is equivalent to newColor = alpha*sourceColor + oldColor

Distributing them around the finger

- We want the particles to spray out from the finger
  - Add member variables to **EAGLView** which indicate whether the finger is down or not, and its current position
- Add **touchesBegan**, **touchesEnded** and **touchesMoved** methods to **EAGLView**
  - These will receive the touch events
  - When the finger goes down or moves, record the position
- In **updateParticles** only add new particles if the finger is down
- Make the initial position of the particles the finger location
  - The y coordinate will be wrong!
    - Correct value is 480-y (OpenGL coordinates are upside with respect to device coordinates)

Orienting particles

- The particles are all facing horizontally
  - This doesn't look right
  - They should point along the direction they are moving

- In **update**, compute the angle of the particle
  - store it in an instance variable
  - NB: **angle = atan2(dy,dx)** (**atan** gives result in radians!)

- In the **renderParticle** method, rotate by the particle's **angle** (after the translation)
  - Remember to convert from radians to degrees!
  - Rotate around the z axis (0,0,1)
  - Now the particles should line up with the direction they are going

- **Build it, and check that it works**

Gravity! :)

- Gravity is easy to simulate
  - Just a constant on to **dy** in each **update call**
    - NB: increment **dy**, not **y**!
  - -2 works well
- The particles will now fall down
  - However, they will speed up unrealistically
  - Real sparks have (lots of) air resistance
- Simulate air resistance by multiplying **dx** and **dy** by a constant < 1.0 on every **update**
  - a scaling of 0.7 -> 0.99 will give good results
  - e.g. **dx** = **dx** * 0.9

The walls

- One final touch: the particles should not fall "off" the edge of the screen

- In **update**, test if the y coordinate is < 0
  - if it is, set the y coordinate to 0, and set they y velocity (**dy**) to 0
  - i.e. stop it at the edge

- Now the particles should pool up nicely at the bottom of the screen!

- Also, add a **scale** variable to Particle to introduce size variability
  - Scale the particle by this value when rendering it (using **glScalef**)
  - Set the scale value to some random value when particles are created
    - say between 0.2 and 1.5
    - note: to generate a random number from 0.0--1 .0use **arc4random()/(double)ARC4RANDOM_MAX**

- **That's it, we're done!**

Bonus points

- If you have a real iPhone / iPod you could make the gravity in the particle system depend on the real angle of the device
  - look up **UIAccelerometer**
  - the angle of the device is given by atan2(accelerationY, accelerationX)

- You could make the particle system multi-touch
  - for example, have the particles shoot from one finger to another
  - or have them orbit around the second finger
  - You just need to extend the code in **touchesBegan/touchesEnded/touchesMoved**

- You could also add some sound
  - a loopable "sparking" sound is in the lab zip file...
  - look up **AVAudioPlayer**
    - **it has playback/stop functionality with looping support**