All about OpenGL ES 2.x – (part 2/3)

Very welcome back, my friends!!!
Now we are aware about the basic concepts of 3D world and OpenGL. Now it'is time to start the fun! Let's go deep into code and see some result on our screens. Here I'll show you how to construct an OpenGL application, using the best practices.

If you lost the first one, you can check the parts of this serie below.
This serie is composed by 3 parts:

- Part 1 – Basic concepts of 3D world and OpenGL (Beginners)
- Part 2 – OpenGL ES 2.0 in deeply (Intermediate)
- Part 3 – Jedi skills in OpenGL ES 2.0 and 2D graphics (Advanced)

Before start, I want to say something: is Thank You!
The first tutorial of this serie became much much bigger than I could imagine!
When I saw the news at the home of http://www.opengl.org website, I was speechless, stunned! That was really amazing!!!
So I want to say again, Thank you so much!

Here is a little list of contents to orient your reading:

LIST OF CONTENTS TO THIS TUTORIAL

At a glance

Remembering the first part of this serie, we've seen:

1. OpenGL's logic is composed by just 3 simple concepts: Primitives, Buffers and Rasterize.
2. OpenGL works with fixed or programmable pipeline.
3. Programmable pipeline is synonymous of Shaders: Vertex Shader and Fragment Shader.

Here I'll show code more based in C and Objective-C. In some parts I'll talk specifically about iPhone and iOS. But in general the code will be generic to any language or platform. As OpenGL ES is the most concise API of OpenGL, I'll focus on it. If you're using OpenGL or WebGL you could use all the codes and concepts here.

The code in this tutorial is just to illustrate the functions and concepts, not real code. In the link bellow you can get a Xcode project which uses all the concepts and code of this tutorial. I made the principal class (CubeExample.mm) using Objective-C++ just to make clearly to everybody how the OpenGL ES 2.0 works, even those which don't use Objective-C. This training project was made for iOS, more specifically targeted for iPhone.

Download now

Xcode project files to iOS 4.2 or later.

172kb

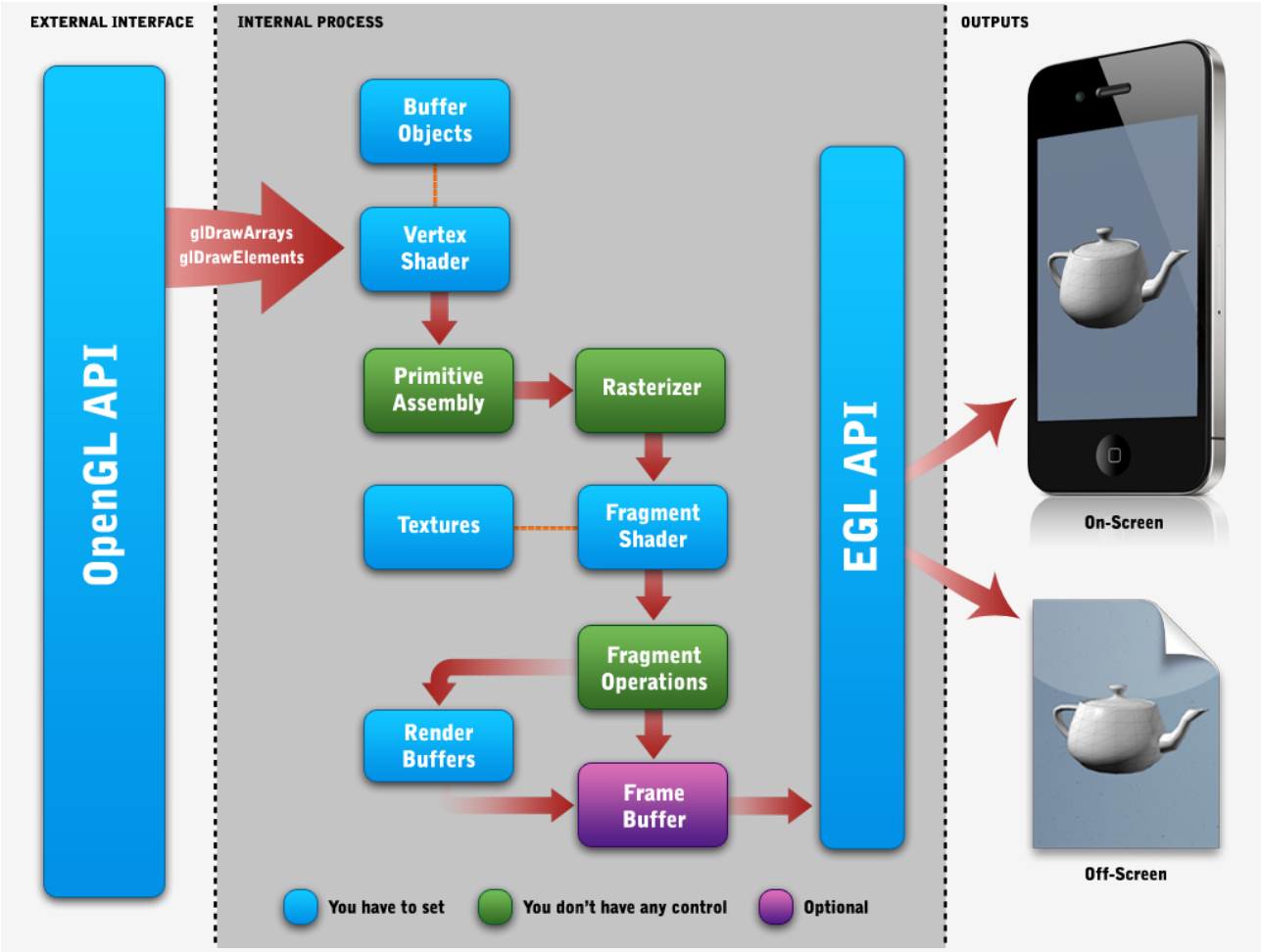
Here I'll use OpenGL functions following the syntax: gl + FunctionName. Almost all implementation of OpenGL use the prefix "gl" or "gl.". But if your programming language don't use this, just ignore this prefix in the following lines.

Another important thing to say before we start is about OpenGL data types. As OpenGL is multiplatform and depends of the vendors implementation, many data type could change from one programming language to another. For example, a float in C++ could represent 32 bits exactly, but in JavaScript a float could be only 16 bits. To avoid these kind of conflict, OpenGL always works with it's own data

types. The OpenGL's data type has the prefix "GL", like **GLfloat** or **GLint**. Here is a full list of OpenGL's data type:

OPENGL'S DATA TYPE	SAME AS C	DESCRIPTION
GLboolean (1 bits)	unsigned char	0 to 1
GLbyte (8 bits)	char	-128 to 127
GLubyte (8 bits)	unsigned char	0 to 255
GLchar (8 bits)	char	-128 to 127
GLshort (16 bits)	short	-32,768 to 32,767
GLushort (16 bits)	unsigned short	0 to 65,353
GLint (32 bits)	int	-2,147,483,648 to 2,147,483,647
GLuint (32 bits)	unsigned int	0 to 4,294,967,295
GLfixed (32 bits)	int	-2,147,483,648 to 2,147,483,647
GLsizei (32 bits)	int	-2,147,483,648 to 2,147,483,647
GLenum (32 bits)	unsigned int	0 to 4,294,967,295
GLdouble (64 bits)	double	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
GLbitfield (32 bits)	unsigned int	0 to 4,294,967,295
GLfloat (32 bits)	float	-2,147,483,648 to 2,147,483,647
GLclampx (32 bits)	int	Integer clamped to the range 0 to 1
GLclampf (32 bits)	float	Floating-point clamped to the range 0 to 1
GLclampd (64 bits)	double	Double clamped to the range 0 to 1
GLintptr	int	pointer *
GLsizeiptr	int	pointer *
GLvoid	void	Can represent any data type

A very important information about Data Types is that OpenGL ES does NOT support 64 bits data types, because embedded systems usually need performance and several devices don't support 64 bits processors. By using the OpenGL data types, you can easily and safely move your OpenGL application from C++ to JavaScript with less changes, for example.
One last thing to introduce is the graphics pipeline. We'll use and talk about the Programmable Pipeline a lot, here is a visual illustration:



The OpenGL programmable pipeline.

We'll talk deeply about each step in that diagram. The only thing I want to say now is about the "Frame Buffer" in the image above. The Frame Buffer is marked as optional because you have the choice of don't use it directly, but internally the OpenGL's core always will work with a Frame Buffer and a Color Render Buffer at least.

Did you notice the EGL API in the image above?

This is a very very important step to our OpenGL's application. Before start this tutorial we need to know at least the basic concept and setup about EGL API. But EGL is a dense subject and I can't place it here. So I've created an article to explain that. You can check it here: EGL and EAGL APIs. I really recommend you read that before to continue with this tutorial.

If you've read or already know about EGL, let's move following the order of the first part and start talking about the Primitives.

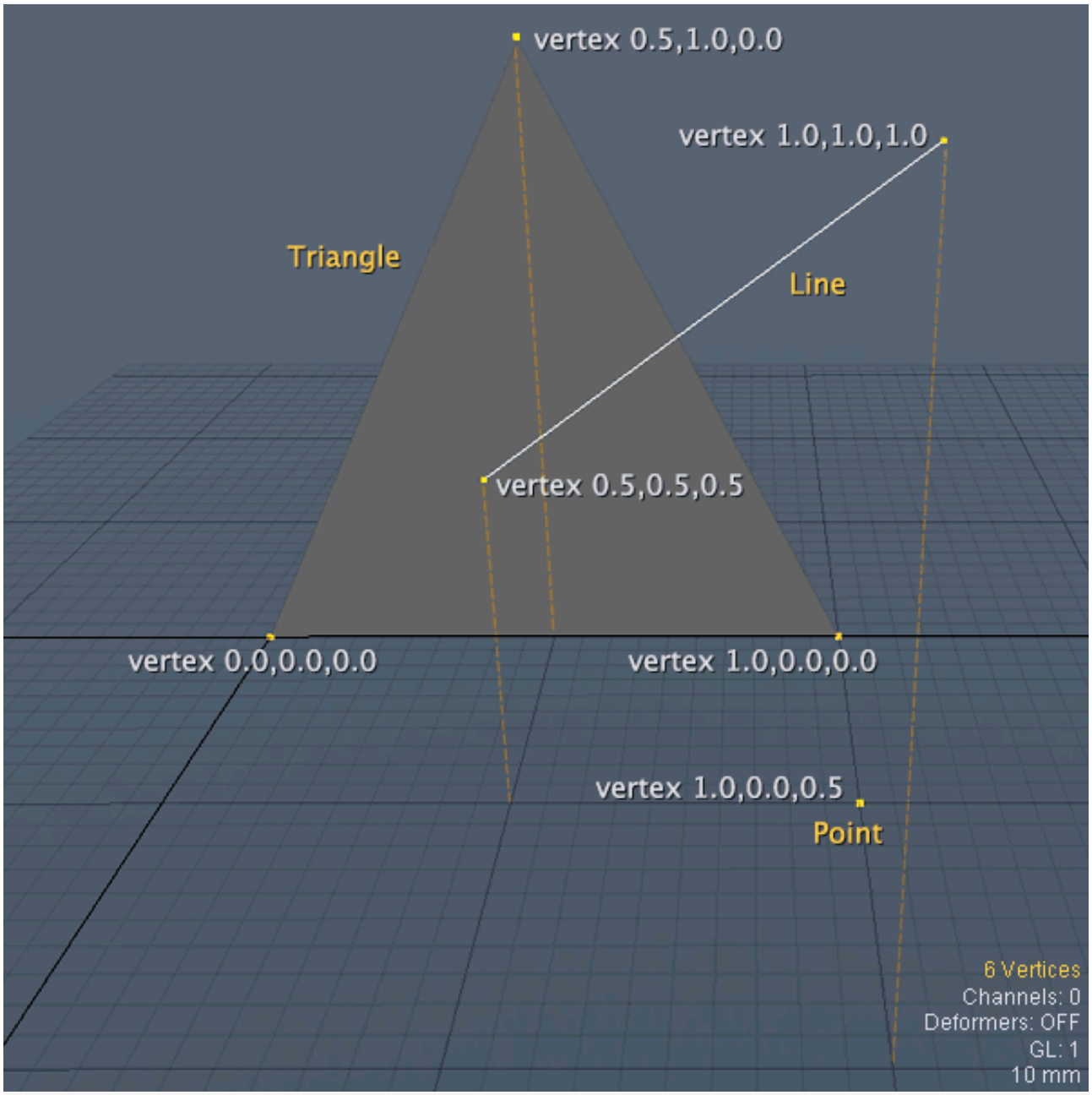
Primitives

top

Do you remember from the first part, when I said that Primitives are Points, Lines and Triangles?
All of them use one or more points in space to be constructed, also called vertex. A vertex has 3 informations, X position, Y position and Z position. A 3D point is constructed by one vertex, a 3D line is composed by two vertices and a triangle is formed by three vertices. As OpenGL always wants to boost the performance, all the informations should be a single dimensional array, more specifically an array of float values. Like this:

```
1   GLfloat point3D = {1.0,0.0,0.5};
2   GLfloat line3D = {0.5,0.5,0.5,1.0,1.0,1.0};
3   GLfloat triangle3D = {0.0,0.0,0.0,0.5,1.0,0.0,1.0,0.0,0.0};
```

As you can see, the array of floats to OpenGL is in sequence without distinction between the vertices, OpenGL will automatically understand the first value as the X value, the second as Y value and the third as the Z value. OpenGL will loop this interpretation at every sequence of 3 values. All that you need is inform to OpenGL if you want to construct a point, a line or a triangle. An advanced information is which you can customize this order if you want and the OpenGL could work with a fourth value, but this is a subject to advanced topics. For now assume that the order always will be X,Y,Z.
The coordinates above will construct something like this:



The three OpenGL's primitives

In this image, the dashed orange lines is just an indication to you see more clearly where the vertices are related to the floor. Until here seems very simple! But now a question comes up: “OK, so how could I transform my 3D models from 3DS Max or Maya into an OpenGL’s array?”

When I was learning OpenGL, I thought that could exist some 3D file formats we could import directly into OpenGL. “After all, the OpenGL is the most popular Graphics Library and is used by almost all 3D softwares! I’m sure it has some methods to import 3D files directly!”

Well, I was wrong! Bad news.

I’ve learned and need to say you: remember that OpenGL is focused on the most important and hard part of the 3D world construction. So it should not be

responsible by fickle things, like 3D file formats. Exist so many 3D file formats, .obj, .3ds, .max, .ma, .fbx, .dae, .lxo… is too much to the OpenGL and the Khronos worrying about.
But the Collada format is from Khronos, right? So would I expect that, one day, OpenGL will be able to import Collada files directly? No! Don't do this. Accept this immutable truth, OpenGL does not deal with 3D files!
OK, so what we need to do to import 3D models from 3D softwares into our OpenGL's application? Well my friend, unfortunately I need to say you: you will need a 3D engine or a third party API. There's no easy way to do that.
If you choose a 3D engine, like PowerVR, SIO2, Oolong, UDK, Ogre and many others, you'll be stuck inside their APIs and their implementation of OpenGL. If you choose a third party API just to load a 3D file, you will need to integrate the third party class to your own implementation of OpenGL.
Another choice is to search a plugin to your 3D software to export your objects as a .h file. The .h is just a header file containing your 3D objects in the OpenGL array format. Unfortunately, until today I just saw 2 plugins to do this: One to Blender made with Phyton and another made with Pearl and both was horribles. I never seen plugins to Maya, 3DS Max, Cinema 4D, LightWave, XSI, ZBrush or Modo.
I wanna give you another opportunity, buddy. Something called NinevehGL!
I'll not talk about it here, but it's my new 3D engine to OpenGL ES 2.x made with pure Objective-C. I offer you the entire engine or just the parse API to some file formats as .obj and .dae. Whatever you prefer. You can check the NinevehGL's website here:
http://nineveh.gl (not released yet, coming soon)
What is the advantage of NinevehGL? Is to KEEP IT SIMPLE! The others 3D engines is too big and unnecessary expensive. NinevehGL is free!
OK, let's move deeply into primitives.

Meshes and Lines Optimization

top

A 3D point has only one way to be draw by OpenGL, but a line and a triangle have three different ways: normal, strip and loop for the lines and normal, strip and fan for the triangles. Depending on the drawing mode, you can boost your render performance and safe memory in your application. But at the right time we'll discuss this, later on this tutorial.

For now, all that we need to know is that the most complex 3D mesh you could imagine will be made with a bunch of triangles. We call these triangles of “faces”. So let’s create a 3D cube using an array of vertices.

```
1   // Array of vertices to a cube.
2   GLfloat cube3D[] =
3   {
4       0.50,-0.50,-0.50,   // vertex 1
5       0.50,-0.50,0.50,    // vertex 2
6       -0.50,-0.50,0.50,   // vertex 3
7       -0.50,-0.50,-0.50,  // vertex 4
8       0.50,0.50,-0.50,    // vertex 5
9       -0.50,0.50,-0.50,   // vertex 6
10      0.50,0.50,0.50,     // vertex 7
11      -0.50,0.50,0.50     // vertex 8
12  }
```

The precision of the float numbers really doesn’t matter to OpenGL, but it can safe a lot of memory and size into your files (precision of 2 is 0.00 precision of 5 is 0.00000). So I always prefer to use low precision, 2 is very good!
I don’t want to make you confused too soon, but has something you have to know. Normally meshes have three great informations: verticex, texture coordinates and normals. A good practice is to create one single array containing all these informations. This is called **Array of Structures**. A short example of it could be:

```
1   // Array of vertices to a cube.
2   GLfloat cube3D[] =
3   {
4       0.50,-0.50,-0.50,   // vertex 1
5       0.00,0.33,          // texture coordinate 1
6       1.00,0.00,0.00      // normal 1
7       0.50,-0.50,0.50,    // vertex 2
8       0.33,0.66,          // texture coordinate 2
9       0.00,1.00,0.00      // normal 2
10      ...
11  }
```

You can use this construction technique for any kind of information you want to use as a per-vertex data. A question arises: Well, but at this way all my data must be of only one data type, GLfloat for example? Yes. But I'll show you later in this tutorial that this is not a problem, because to where your data goes, just accept floating-point values, so everything will be GLfloats. But don't worry with this now, in the right time you will understand.
OK, now we have a 3D mesh, so let's start to configure our 3D application and store this mesh into an OpenGL's buffer.

Buffers

top

Do you remember from the first part when I said that OpenGL is a state machine working like a Port Crane? Now let's refine a little that illustration. OpenGL is like a Port Crane with severeal arms and hooks. So it can hold many containers at the same time.

OpenGL is like a Port Crane with few arms and hooks.

Basically, there are four "arms": texture arm (which is a double arm), buffer object arm (which is a double arm), render buffer arm and frame buffer arm. Each arm can hold only one container at a time. This is very important, so I'll repeat this: **Each arm can hold only ONE CONTAINER AT A TIME!** The texture and buffer object arms are double arms because can hold two different kinds of texture and buffer objects, respectively, but also only ONE KIND OF CONTAINER AT A TIME! We need is instruct the OpenGL's crane to take a container from the port, we can do this by informing the name/id of the container.

Backing to the code, the command to instruct OpenGL to "take a container" is: **glBind***. So every time you see a glBindSomething you know, that is an instruction to OpenGL "take a container". Exist only one exception to this rule, but we'll discuss that later on. Great, before start binding something into OpenGL we

need to create that thing. We use the **glGen*** function to generate a “container” name/id.

Frame Buffers

top

A frame buffer is a temporary storage to our render output. Once our render is in a frame buffer we can choose present it into device’s screen or save it as an image file or either use the output as a snapshot.
This is the pair of functions related to frame buffers:

FRAMEBUFFER CREATION

GLvoid glGenFramebuffers (GLsizei n, GLuint* framebuffers)

- **n**: The number representing how many frame buffers's names/ids will be generated at once.
- **framebuffers**: A pointer to a variable to store the generated names/ids. If more than one name/id was generated, this pointer will point to the start of an array.

GLvoid glBindFramebuffer (GLenum target, GLuint framebuffer)

- **target**: The target always will be GL_FRAMEBUFFER, this is just an internal convention for OpenGL.
- **framebuffers**: The name/id of the frame buffer to be bound.

In deeply the creation process of an OpenGL Object will be done automatically by the core when we bind that object at the first time. But this process doesn't generates a name/id to us. So is advisable always use **glGen*** to create buffer names/ids instead create your very own names/ids. Seems confused?
OK, let's go to our first lines of code and you'll understand more clearly:

```
1    GLuint frameBuffer;
2
3    // Creates a name/id to our frameBuffer.
4    glGenFramebuffers(1, &frameBuffer);
5
6    // The real Frame Buffer Object will be created here,
7    // at the first time we bind an unused name/id.
8    glBindFramebuffer(GL_FRAMEBUFFER, frameBuffer);
9
10   // We can suppress the glGenFrambuffers.
11   // But in this case we'll need to manage the names/ids by ourselves.
12   // In this case, instead the above code, we could write something like:
13   //
14   // GLint frameBuffer = 1;
15   // glBindFramebuffer(GL_FRAMEBUFFER, frameBuffer);
```

The above code creates an instance of Gluint data type called frameBuffer. Then we inform the memory location of frameBuffer variable to the glGenFramebuffers and instruct this function to generate only 1 name/id (Yes, we can generate multiple names/ids at once). So finally we bind that generated frameBuffer to OpenGL's core.

Render Buffers

top

A render buffer is a temporary storage for images coming from an OpenGL's render. This is the pair of functions related to render buffers:

RENDERBUFFER CREATION

GLvoid glGenRenderbuffers (GLsizei n, GLuint* renderbuffers)

- **n**: The number representing how many render buffers's names/ids will be generated at once.
- **renderbuffers**: A pointer to a variable to store the generated names/ids. If more than one name/id was generated, this pointer will point to the start of an array.

GLvoid glBindRenderbuffer (GLenum target, GLuint renderbuffer)

- **target**: The target always will be GL_RENDERBUFFER, this is just an internal convention for OpenGL.
- **renderbuffer**: The render buffer name/id to be bound.

OK, now, before we proceed, do you remember from the first part when I said that render buffer is a temporary storage and could be of 3 types? So we need to specify the kind of render buffer and some properties of that temporary image. We set the properties to a render buffer by using this function:

RENDERBUFFER PROPERTIES

GLvoid glRenderbufferStorage (GLenum target, GLenum internalformat, GLsizei width, GLsizei height)

- **target**: The target always will be GL_RENDERBUFFER, this is just an internal convention for OpenGL.
- **internalformat**: This specifies what kind of render buffer we want and what color format this temporary image will use. This parameter can be:
    - **GL_RGBA4**, **GL_RGB5_A1** or **GL_RGB56** to a render buffer with final colors;
    - **GL_DEPTH_COMPONENT16** to a render buffer with Z depth;
    - **GL_STENCIL_INDEX** or **GL_STENCIL_INDEX8** to a render buffer with stencil informations.
- **width**: The final width of a render buffer.
- **height**: The final height of a render buffer.

You could ask, "but I'll set these properties for which render buffer? How OpenGL will know for which render buffer is these properties?" Well, is here that the great OpenGL's state machine comes up! The properties will be set to the last render buffer bound! Very simple.
Look at how we can set the 3 render buffers kind:

```
1  GLuint colorRenderbuffer, depthRenderbuffer, stencilRenderbuffer;
2  GLint sw = 320, sh = 480; // Screen width and height, respectively.
3
4  // Generates the name/id, creates and configures the Color Render Buffer.
5  glGenRenderbuffers(1, &colorRenderbuffer);
6  glBindRenderbuffer(GL_RENDERBUFFER, colorRenderbuffer);
7  glRenderbufferStorage(GL_RENDERBUFFER, GL_RGBA4, sw, sh);
8
9  // Generates the name/id, creates and configures the Depth Render Buffer.
10 glGenRenderbuffers(1, &depthRenderbuffer);
11 glBindRenderbuffer(GL_RENDERBUFFER, depthRenderbuffer);
12 glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT16, sw, sh);
13
14 // Generates the name/id, creates and configures the Stencil Render
   Buffer.
15 glGenRenderbuffers(1, &stencilRenderbuffer);
16 glBindRenderbuffer(GL_RENDERBUFFER, stencilRenderbuffer);
17 glRenderbufferStorage(GL_RENDERBUFFER, GL_STENCIL_INDEX8, sw, sh);
```

OK, but in our cube application we don’t need stencil buffer, so let’s optimize the above code:

```
1    GLuint *renderbuffers;
2    GLint sw = 320, sh = 480; // Screen width and height, respectively.
3
4    // Let's create multiple names/ids at once.
5    // To do this we declared our variable as a pointer *renderbuffers.
6    glGenRenderbuffers(2, renderbuffers);
7
8    // The index 0 will be our color render buffer.
9    glBindRenderbuffer(GL_RENDERBUFFER, renderbuffers[0]);
10   glRenderbufferStorage(GL_RENDERBUFFER, GL_RGBA4, sw, sh);
11
12   // The index 1 will be our depth render buffer.
13   glBindRenderbuffer(GL_RENDERBUFFER, renderbuffers[1]);
14   glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT16, sw, sh);
```

At this point I need to make a digression.
This step is a little bit different if you are in Cocoa Framework. The Apple doesn't allow to us put the OpenGL render directly onto device's screen, we need to place the output into a color render buffer and ask to the EAGL (The EGL's implementation by Apple) to present the buffer on the device's screen. As the color render buffer in this case is always mandatory, to set their properties we need to call a different method from the EAGLContext called**renderbufferStorage:fromDrawable:** and inform a CAEAGLLayer which we want to render onto. Seems confused? So is time to you make a digression in your reading and go to this article: Apple's EAGL.
In that article I explain what is the EAGL and how to use it.
Once knowing about EAGL, you use the following code to set the color render buffer's properties, instead glRenderbufferStorage:

RENDERBUFFER PROPERTIES IN CASE OF COCOA FRAMEWORK

- (BOOL) renderbufferStorage:(NSUInteger)target fromDrawable:(id)drawable

- **target**: The target always will be GL_RENDERBUFFER, this is just an internal convention for OpenGL.
- **fromDrawable**: Your custom instance of CAEAGLLayer.

```
1  // Suppose you previously set EAGLContext *_context
2  // as I showed in my EAGL article.
3
4  GLuint colorBuffer;
5
6  glGenRenderbuffers(1, & colorBuffer);
7  glBindRenderbuffer(GL_RENDERBUFFER, colorBuffer);
8  [_context renderbufferStorage:GL_RENDERBUFFER fromDrawable:myCAEAGLLayer];
```

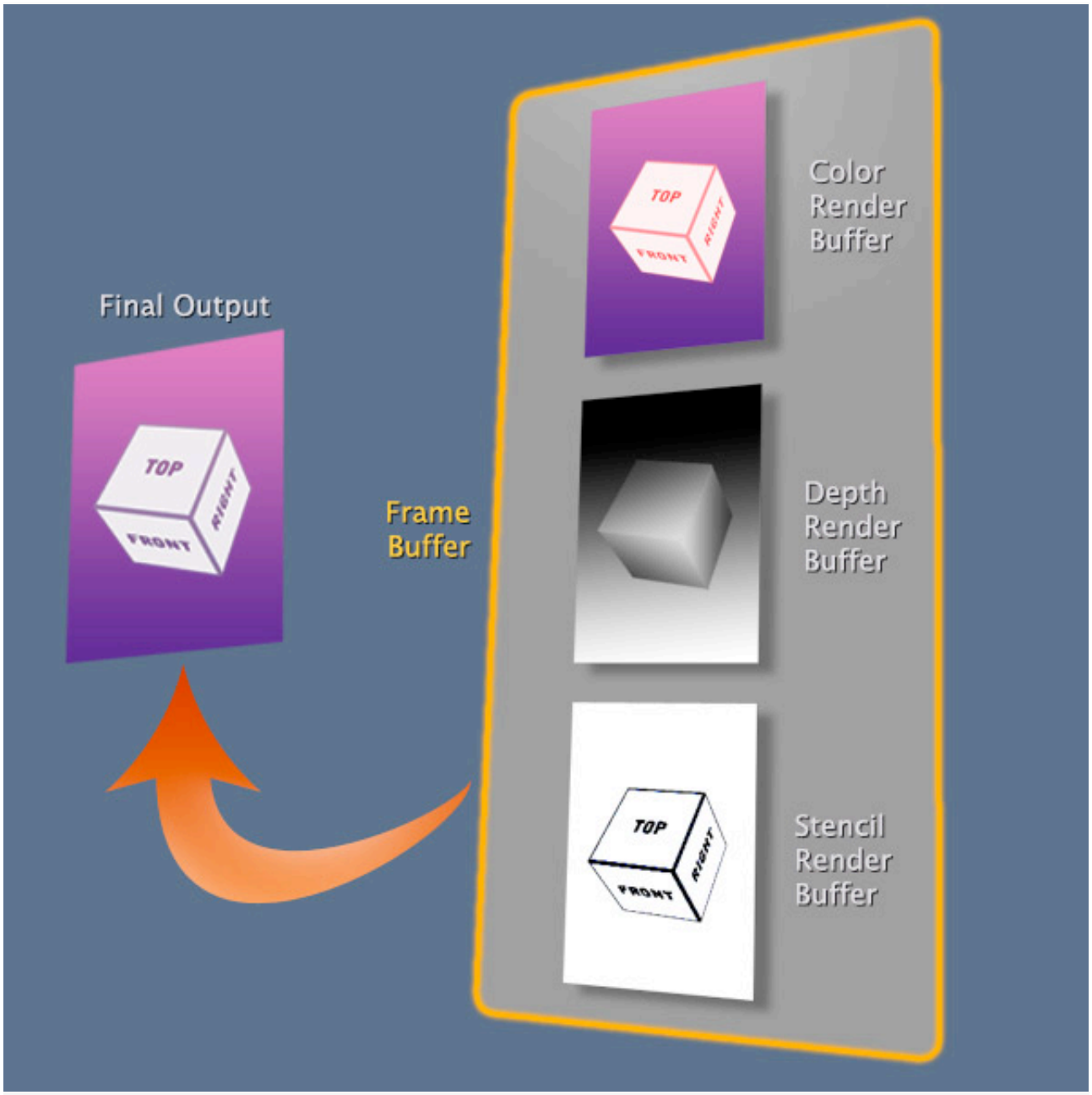
When you call renderbufferStorage:fromDrawable: informing a CAEAGLLayer, the EAGLContext will take all relevant properties from the layer and will properly set the bound color render buffer.
Now is time to place our render buffers inside our previously created frame buffer. Each frame buffer can contain ONLY ONE render buffer of each type. So we can't have a frame buffer with 2 color render buffers, for example. To attach a render buffer into a frame buffer, we use this function:

ATTACH RENDERBUFFERS TO A FRAMEBUFFER

GLvoid glFramebufferRenderbuffer (GLenum target, GLenum attachment, GLenum renderbuffertarget, GLuint renderbuffer)

- **target**: The target always will be GL_FRAMEBUFFER, this is just an internal convention for OpenGL.
- **attachment**: This specifies which kind of render buffer we want to attach inside a frame buffer, this parameter can be:
  - **GL_COLOR_ATTACHMENT0**: To attach a color render buffer;
  - **GL_DEPTH_ATTACHMENT**: To attach a depth render buffer;
  - **GL_STENCIL_ATTACHMENT**: To attach a stencil render buffer.
- **renderbuffertarget**: The renderbuffertarget always will be GL_RENDERBUFFER, this is just an internal convention for OpenGL.
- **renderbuffer**: The name/id of the render buffer we want to attach.

The same question comes up: “How OpenGL will know for which frame buffer attach these render buffers?” Using the state machine! The last frame buffer bound will receive these attachments.
OK, before move on, let’s talk about the combination of Frame Buffer and Render Buffer. This is how they looks like:



Relationship between Frame Buffer and Render Buffers.

Internally OpenGL's always works with a frame buffer. This is called window-system-provided frame buffer and the frame buffer name/id 0 is reserved to it. The frame buffers which we control are know as application-created frame buffers. The depth and stencil render buffers are optionals. But the color buffer is always enabled and as OpenGL's core always uses a color render buffer too, the render buffer name/id 0 is reserved to it. To optimize all the optional states, OpenGL gives to us an way to turn on and turn off some states (understanding as state every optional OpenGL's feature). To do this, we use these function:

TURNING ON/OFF THE OPENGL' STATES

GLvoid glEnable(GLenum capability)

- **capability**: The feature to be turned on. The values can be:
  - **GL_TEXTURE_2D**
  - **GL_CULL_FACE**
  - **GL_BLEND**
  - **GL_DITHER**
  - **GL_STENCIL_TEST**
  - **GL_DEPTH_TEST**
  - **GL_SCISSOR_TEST**
  - **GL_POLYGON_OFFSET_FILL**
  - **GL_SAMPLE_ALPHA_TO_COVERAGE**
  - **GL_SAMPLE_COVERAGE**

GLvoid glDisable(GLenum capability)

- **capability**: The feature to be turned off. The values can be the same as **glEnable**.

Once we turned on/off a feature, this instruction will affect the entire OpenGL machine. Some people prefer to turn on a feature just for a while to use it and then turn off, but this is not advisable. It's expensive. The best way is turn on once and turn off once. Or if you really need, minimize the turn on/off in your application. So, back to the depth and stencil buffer, if you need in your application to use one of them or both, try enable what you need once. As in our cube's example we just need a depth buffer, we could write:

```
1  // Doesn't matter if this is before or after
2  // we create the depth render buffer.
3  // The important thing is enable it before try
4  // to render something which needs to use it.
5  glEnable(GL_DEPTH_TEST);
```

Later I'll talk deeply about what the depth and stencil tests make and their relations with fragment shaders.

Buffer Objects

top

The buffer objects are optimized storage for our primitive's arrays. Has two kind of buffer objects, the first is that we store the array of vertices, because it the buffer objects is also know as Vertex Buffer Object (VBO). After you've created the buffer object you can destruct the original data, because the Buffer Object (BO) made a copy from it. We are used to call it VBO, but this kind of buffer object could hold on any kind of array, like array of normals or an array of texture coordinates or even an array of structures. To adjust the name to fit the right idea, some people also call this kind of buffer object as Array Buffer Object (ABO).

The other kind of buffer object is the Index Buffer Object (IBO). Do you remember the array of indices from the first part? (click here to remember). So the IBO is to store that kind of array. Usually the data type of array of indices is GLubyte or GLushort. Some devices have support up to GLuint, but this is like an extension, almost a plugin which vendors have to implement. The majority just support the default behavior (GLubyte or GLushort). So my advice is, always limit your array of indices to GLushort.

OK, now to create these buffers the process is very similar to frame buffer and render buffer. First you create one or more names/ids, later you bind one buffer object, and then you define the properties and data into it.

BUFFER OBJECTS CREATION

GLvoid glGenBuffers(GLsizei n, GLuint* buffers)

- **n**: The number representing how many buffer objects's names/ids will be generated at once.
- **buffers**: A pointer to a variable to store the generated names/ids. If more than one name/id was generated, this pointer will point to the start of an array.

GLvoid glBindBuffer(GLenum target, GLuint buffer)

- **target**: The target will define what kind of buffer object will be, VBO or IBO. The values can be:
  - **GL_ARRAY_BUFFER**: This will set a VBO (or ABO, whatever).
  - **GL_ELEMENT_ARRAY_BUFFER**: This will set an IBO.
- **buffer**: The name/id of the frame buffer to be bound.

Now is time to refine that illustration about the Port Crane's hooks. The BufferObject Hook is in reality a double hook. Because it can hold two buffer

objects, one of each
type: **GL_ARRAY_BUFFER** and**GL_ELEMENT_ARRAY_BUFFER**.
OK, once you have bound a buffer object is time to define its properties, or was better to say define its content. As the “BufferObject Hook” is a double one and you can have two buffer objects bound at same time, you need to instruct the OpenGL about the kind of buffer object you want to set the properties for.

BUFFER OBJECTS PROPERTIES

GLvoid glBufferData(GLenum target, GLsizeiptr size, const GLvoid* data, GLenum usage)

- **target**: Indicates for what kind of buffer you want to set the properties for. This param can be**GL_ARRAY_BUFFER** or **GL_ELEMENT_ARRAY_BUFFER**.
- **size**: The size of the buffer in the basic units (bytes).
- **data**: A pointer to the data.
- **usage**: The usage kind. This is like a tip to help the OpenGL to optimize the data. This can be of three kinds:
    - **GL_STATIC_DRAW**: This denotes an immutable data. You set it once and use the buffer often.
    - **GL_DYNAMIC_DRAW**: This denotes a mutable data. You set it once and update its content several times using it often.
    - **GL_STREAM_DRAW**: This denotes a temporary data. For who of you which is familiar with Objective-C, this is like an autorelease. You set it once and use few times. Later the OpenGL automatically will clean and destroy this buffer.

GLvoid glBufferSubData(GLenum target, GLintptr offset, GLsizeiptr size, const GLvoid* data)

- **target**: Indicates for what kind of buffer you want to set the properties for. This param can be**GL_ARRAY_BUFFER** or **GL_ELEMENT_ARRAY_BUFFER**.
- **offset**: This number represent the offset which you will start to make changes into the previously defined buffer object. This number is given is basic units (bytes).
- **size**: This number represent the size of the changes into the previously defined buffer object. This number is given is basic units (bytes).
- **data**: A pointer to the data.

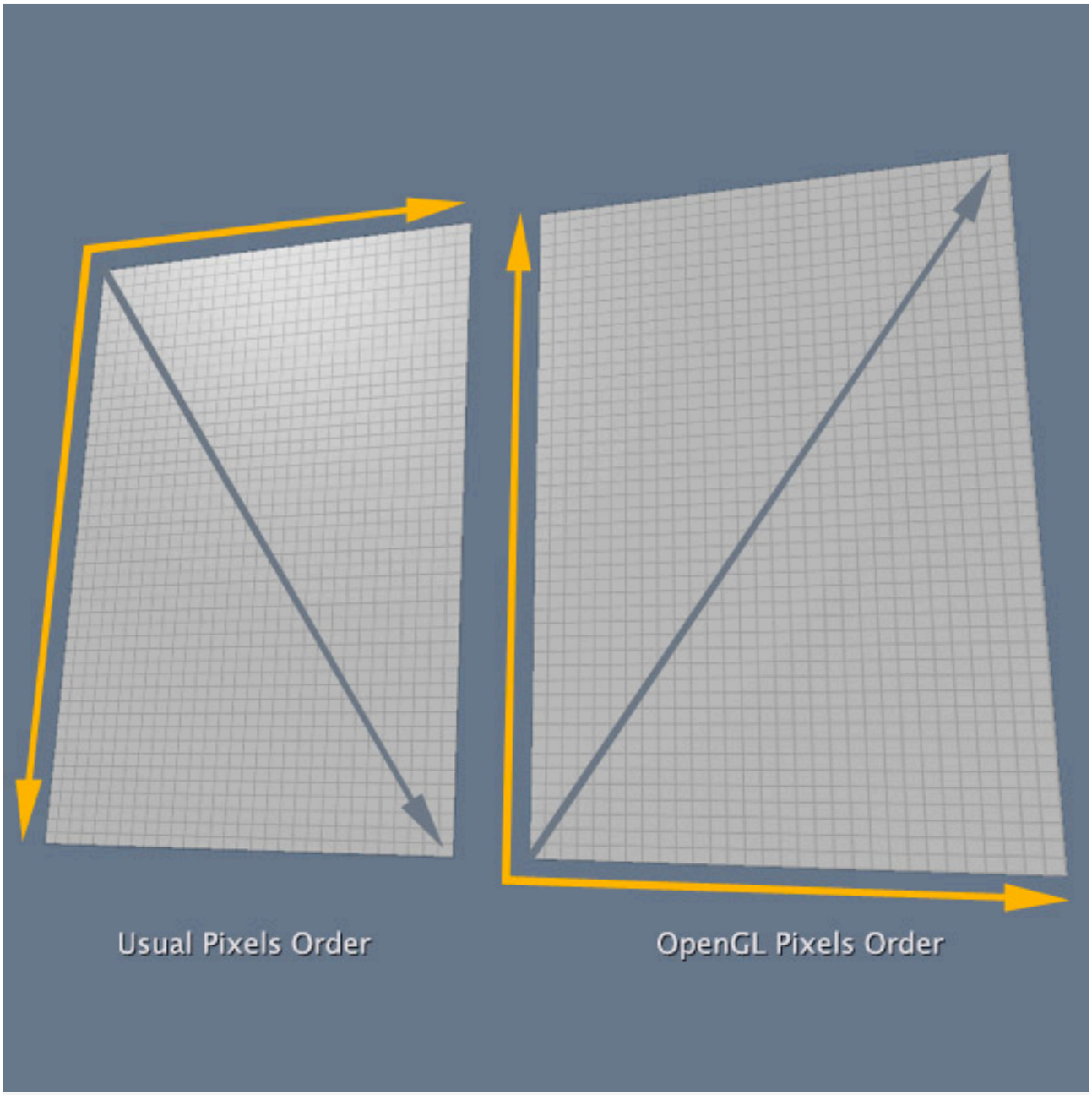
Now let’s understand what these functions make. The first one, glBufferData, you use this function to set the content for your buffer object and its properties. If you choose the usage type of GL_DYNAMIC_DRAW, it means you want to update that

buffer object later and to do this you need to use the second one, glBufferSubData.
When you use the glBufferSubData the size of your buffer object was previously defined, so you can't change it. But to optimize the updates, you can choose just a little portion of the whole buffer object to be updated.
Personally, I don't like to use GL_DYNAMIC_DRAW, if you stop to think about it you will see that doesn't exist in the 3D world an effect of behavior which only can be done changing the original vertex data, normal data or texture coordinate data. By using the shaders you can change almost everything related to those data. Using GL_DYNAMIC_DRAW certainly will be much more expansive than using a shaders's approach. So, my advice here is: Avoid to use GL_DYNAMIC_DRAW as much as possible! Always prefer think in a way to achieve the same behavior using the Shaders features.
Once the Buffer Object was properly created and configured, is very very simple to use it. All we need to do is bind the desired Buffer Objects. Remember we can bind only one kind of Buffer Object at a time. While the Buffer Objects stay bound, all the drawing commands we make will use them. After the usage is a good idea unbind them.
Now let's move to the textures.

Textures

top

Oh man, textures is very large topic in OpenGL. To don't increase the size of this tutorial more than it actually is, let's see the basic about the texture here. The advanced topics I'll let to the third part of this tutorial or an exclusive article.
The first thing I need to say you is about the Power of Two (POT). OpenGL ONLY accept POT textures. What that means? That means all the textures must to have the width and height a power of two value. Like 2, 4, 8, 16, 32, 64, 128, 256, 512 or 1024 pixels. To a texture, 1024 is a bigger size and normally indicate the maximum possible size of a texture. So all texture which will be used in OpenGL must to have dimensions like: 64 x 128 or 256 x 32 or 512 x 512, for example. You can't use 200 x 200 or 256 x 100. This is a rule to optimize the internal OpenGL processing in the GPU.
Another important thing to know about textures in OpenGL is the read pixel order. Usually image file formats store the pixel information starting at the upper right corner and moves through line by line to the lower left corner. File format like JPG, PNG, BMP, GIF, TIFF and others use this pixel order. But in the OpenGL this order is flipped upside down. The textures in OpenGL reads the pixels starting from the lower left corner and goes to the upper right corner.



OpenGL reads the pixels from lower left corner to the upper right corner.

So, to solve this little issue, we usually make a vertical flip on our images data before upload it to the OpenGL's core. If your programming language let you re-scale the images, this is equivalent to re-scale the height in -100%.

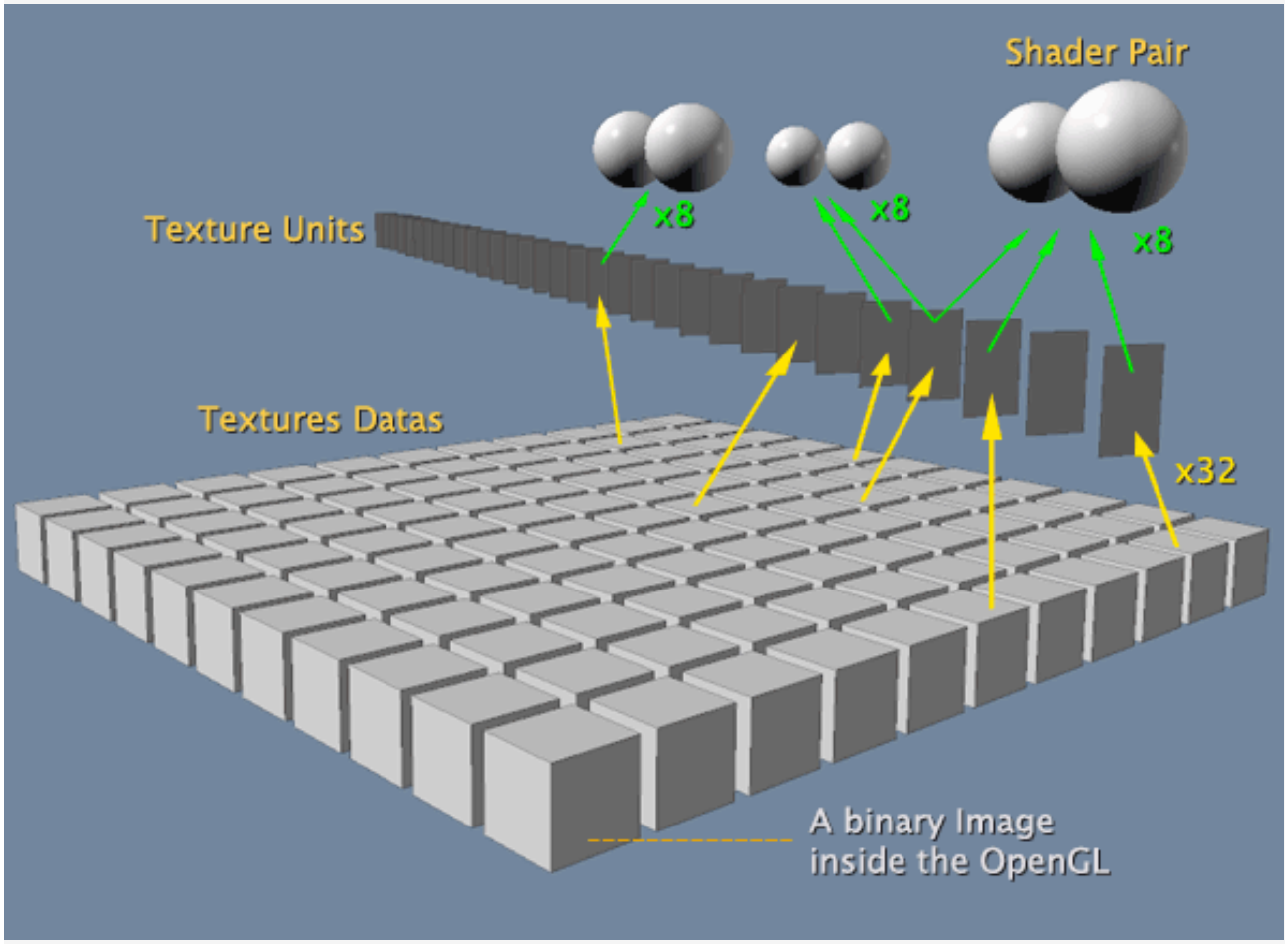
Image data must be flipped vertically to right fit into OpenGL.

Now, shortly about the logic, the textures in OpenGL works at this way: You have an image file, so you must to extract the binary color informations from it, the hexadecimal value. You could extract the alpha information too, OpenGL supports RGB and RGBA format. In this case you'll need to extract the hexadecimal + alpha value from your image. Store everything into an array of pixels.

With this array of pixel (also called texels, because will be used in a texture) you can construct an OpenGL's texture object. OpenGL will copy your array and store it in an optimized format to use in the GPU and in the frame buffer, if needed.

Now is the complex part, some people has criticized OpenGL so much by this approach. Personally I think this could be better too, but is what we have today. OpenGL has something called "Texture Units", by default any OpenGL

implementation by vendors must supports up to 32 Texture Units. These Units represent a temporary link between the stored array of pixels and the actual render processing. You'll use the Texture Units inside the shaders, more specifically inside the fragment shaders. By default each shader can use up to 8 textures, some vendors's implementation support up to 16 textures per shader. Further, OpenGL has a limit to the pair of shader, though each shader could use up to 8 texture units, the pair of shader (vertex and fragment) are limited to use up to 8 texture units together. Confused? Look, if you are using the texture units in only one shader you are able to use up to 8. But if you are using texture units in both shader (different textures units), you can't use more than 8 texture units combined.
Well, OpenGL could hold on up to 32 Texture Units, which we'll use inside the shaders, but the shader just support up to 8, this doesn't make sense, right? Well, the point is that you can set up to 32 Texture Units and use it throughout many shaders. But if you need a 33th Texture Unit you'll need reuse a slot from the firsts 32.
Very confused! I know…
Let's see if an visual explanation can clarify the point:



You can define up to 32 Texture Units, but just up to 8 textures per shader.

As you saw in that image, one Texture Unit can be used many times by multiple shaders pairs. This approach is really confused, but let's understand it by the Khronos Eyes: "Shader are really great!", a Khronos developer said to the other, "They are processed very fast by the GPU. Right! But the textures… hmmm.. textures data still on the CPU, they are bigger and heavy informations! Hmmm.. So we need a fast way to let the shaders get access the textures, like a bridge, or something temporary. Hmmm… We could create an unit of the texture that could be processed directly in the GPU, just as the shaders. We could limit the number of current texture units running on the GPU. A cache in the GPU, is fast, is better! Right! To make the setup, the user bind a texture data to a texture unit and instruct his shaders to use that unit! Seems simple! Let's use this approach."
Normally the texture units are used in the fragment shader, but the vertex shader can also performs look up into a texture. This is not common but could be useful in some situations.
Well, the OpenGL texture units approach could be better, of course, but as I said, is what we have for now! OK, again the code is very similar to the others above: You Generate a texture object, Bind this texture and set its properties. Here are the functions:

TEXTURE CREATION

GLvoid glGenTextures(GLsizei n, GLuint* textures)

- **n**: The number representing how many textures' names/ids will be generated at once.
- **textures**: A pointer to a variable to store the generated names/ids. If more than one name/id was generated, this pointer will point to the start of an array.

GLvoid glBindTexture(GLenum target, GLuint texture)

- **target**: The target will define what kind of texture will be, a 2D texture or a 3D texture. The values can be:
  - **GL_TEXTURE_2D**: This will set a 2D texture.
  - **GL_TEXTURE_CUBE_MAP**: This will set a 3D texture.
- **texture**: The name/id of the texture to be bound.

Is so weirdest a "3D texture"? The first time I heard "3D texture" I thought:"WTF!". Well, because of this weirding, the OpenGL calls a 3D texture of a Cube Map. Sounds better! Anyway, the point is that represent a cube with one 2D texture in each face, so the 3D texture or cube map represents a collection of six 2D textures. And how we can fetch the the texels? With a 3D vector placed in the center of the cube. This subject need much more attention, so I'll skip the 3D

textures here and will let this discussion to the third part of this tutorial. Let's focus on 2D texture. Using only the GL_TEXTURE_2D.
So, after we've created a 2D texture we need to set its properties. The Khronos group calls the OpenGL's core as "server", so when we define a texture data they say this is an "upload". To upload the texture data and set some properties, we use:

TEXTURE PROPERTIES

GLvoid glTexImage2D (GLenum target, GLint level, GLint internalformat, GLsizei width, GLsizei height, GLint border, GLenum format, GLenum type, const GLvoid* pixels)

- **target**: To a 2D texture this always will be GL_TEXTURE_2D.
- **level**: This parameter represents the mip map level. The base level is 0, for now let's use only 0.
- **internalformat**: This represents the color format of the pixels. This parameter can be:
  - **GL_RGBA**: For RGB + Alpha.
  - **GL_RGB**: For RGB only.
  - **GL_LUMINANCE_ALPHA**: For Red + Alpha only. In this case the red channel will represent the luminosity.
  - **GL_LUMINANCE**: For Red only. In this case the red channel will represent the luminosity.
  - **GL_ALPHA**: For Alpha only.
- **width**: The width of the image in pixels.
- **height**: The height of the image in pixels.
- **border**: This parameter is ignored in OpenGL ES. Always use the value 0. This is just an internal constant to conserve the compatibly with the desktop versions.
- **format**: The format must to have the same value of **internalformat**. Again, this is just an internal OpenGL convention.
- **type**: This represent the data format of the pixels. This parameter can be:
  - **GL_UNSIGNED_BYTE**: This format represent 4 Bytes per pixel, so you can use 8 bits for red, 8 bits for green, 8 bits for blue and 8 bits for alpha channels, for example. This definition is used with all color formats.
  - **GL_UNSIGNED_SHORT_4_4_4_4**: This format represents 2 bytes per pixel, so you can use 4 bits for red, 4 bits for green, 4 bits for blue and 4 bits for alpha channels, for example. This definition is used with RGBA only.
  - **GL_UNSIGNED_SHORT_5_5_5_1**: This format represents 2 bytes per pixel, so you can use 5 bits for red, 5 bits for green, 5 bits for blue and 1 bit for alpha channels, for example. This definition is used with RGBA only.
  - **GL_UNSIGNED_SHORT_5_6_5**: This format represents 2 bytes per pixel, so you can use 5 bits for red, 6 bits for green and 5 bits for blue, for example. This definition is used with RGB only.
- **pixels**: The pointer to your array of pixels.

Uolll! A lot of parameters!
OK, but is not hard to understand. First of all, same behavior of the others “Hooks”, a call to glTexImage2D will set the properties for the last texture bound. About the mip map, it is another OpenGL feature to optimize the render time. In few words, what it does is progressively create smaller copies of the original texture until an insignificant copy of 1×1 pixel. Later, during the rasterize process, OpenGL can choose the original or the one of the copies to use depending on the final size of the 3D object in relation to the view. For now, don’t worry with this feature, probably I’ll create an article only to talk about the texture with OpenGL. After the mip map, we set the color format, the size of the image, the format of our data and finally our pixel data. The 2 bytes per pixel optimized data formats is the best way to optimize your texture, use it always you can. Remember which the color you use in the OpenGL can’t exceed the color range and format of your device and EGL context.
OK, now we know how to construct a basic texture and how it works inside OpenGL. So now let’s move to the Rasterize.

Rasterize

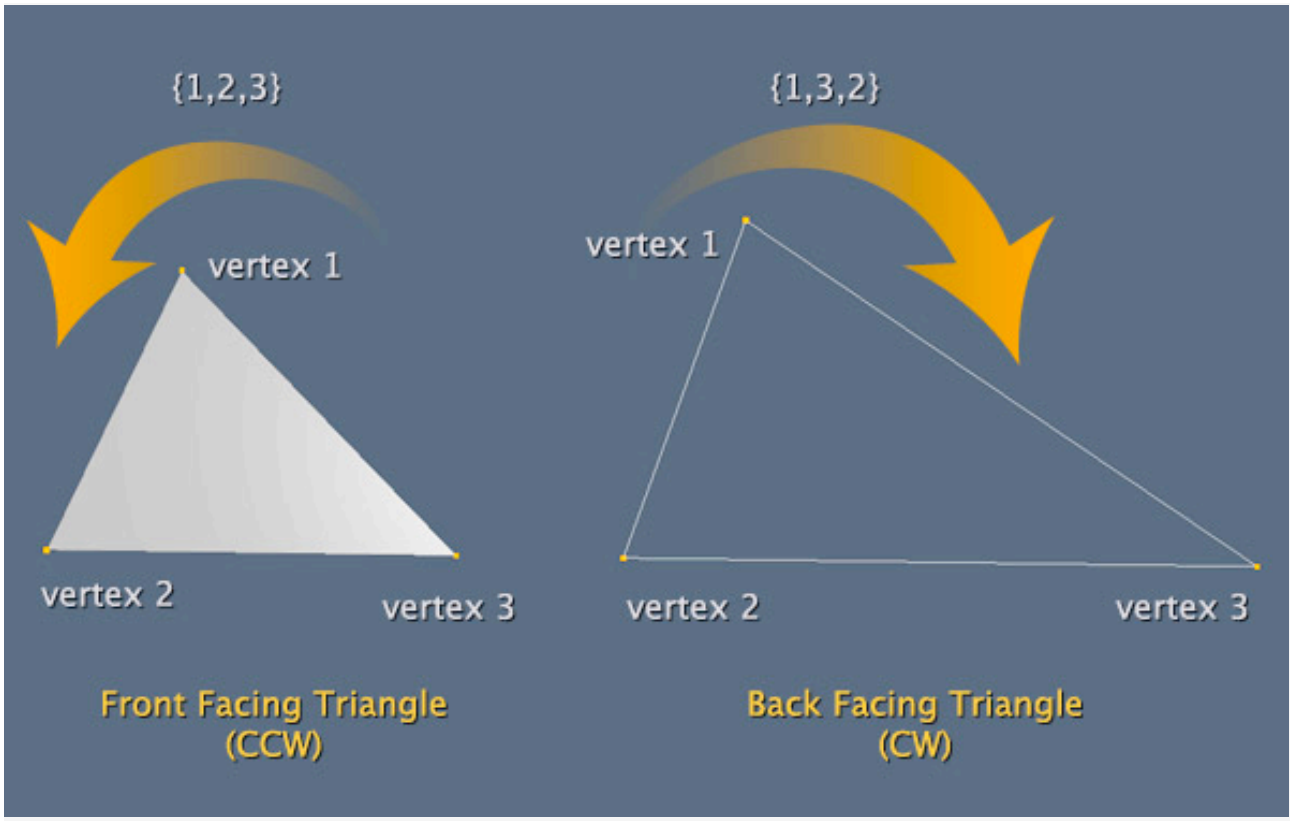
top

The Rasterize in the strict sense is only the process which the OpenGL takes a 3D object and convert its bunch of maths into a 2D image. Later, each fragment of this visible area will be processed by the fragment shader.
Looking at that Programmable Pipeline illustration at the beginning of this tutorial, you can see the Rasterize is just a small step through the graphics pipeline. So why it is so important? I like to say which everything that comes after the Rasterize step is also a Rasterize process, because all that is done later on is also to construct a final 2D image from a 3D object. OK, anyway.
The fact is the Rasterize is the process of creating an image from a 3D object. The Rasterize will occurs to each 3D object in the scene and will update the frame buffer. You can do interferences by many ways in the Rasterize process.

Face Culling

top

Now is time to talk about the Culling, Front Facing and Back Facing. OpenGL works with methods to find and discard the not visible faces. Imagine a simple plane in your 3D application. Let's say you want this plane be visible just by one side (because it represent a wall or a floor, whatever). By default OpenGL will render the both sides of that plane. To solve this issue you can use the culling. Based on the order of vertices OpenGL can determine which is the front and the back face of your mesh (more precisely, it calculates the front and back face of each triangle) and using the culling you can instruct OpenGL to ignore one of these sides (or even both). Look at this picture:



If the culling was enabled, the default behavior will treat clock wise order as a back face.

This feature called culling is completely flexible, you have at least three ways to do the same thing. That picture show only one way, but the most important is understand how it works. In the picture's case, a triangle is composed by vertex 1, vertex 2 and vertex 3. The triangle at the left is constructed using the order of {1,2,3} and that one at the right is formed by the order {1,3,2}. By default the culling will treat triangles formed in the Counter ClockWise as a Front Face and it will not be culled. Following this same behavior, in the right side of the image, the triangle formed in ClockWise will be treated as Back Face and it will be culled (ignored in rasterization process).

To use this feature you need to use **glEnable** function using the parameter **GL_CULL_FACE** and doing this the default behavior will be the explained above. But if you want to customize it, you can use these functions:

CULL FACE PROPERTIES

GLvoid glCullFace(GLenum mode)

- **mode**: Indicates which face will be culled. This parameter can be:
  - **GL_BACK**: This will ignore the back faces. This is the default behavior.
  - **GL_FRONT**: This will ignore the front faces.
  - **GL_FRONT_AND_BACK**: This will ignore both front and back faces (don't ask me why someone will want to exclude the both sides, even knowing which this will produce no render. I'm still trying to figure out the reason of this silly setup until today).

GLvoid glFrontFace(GLenum mode)

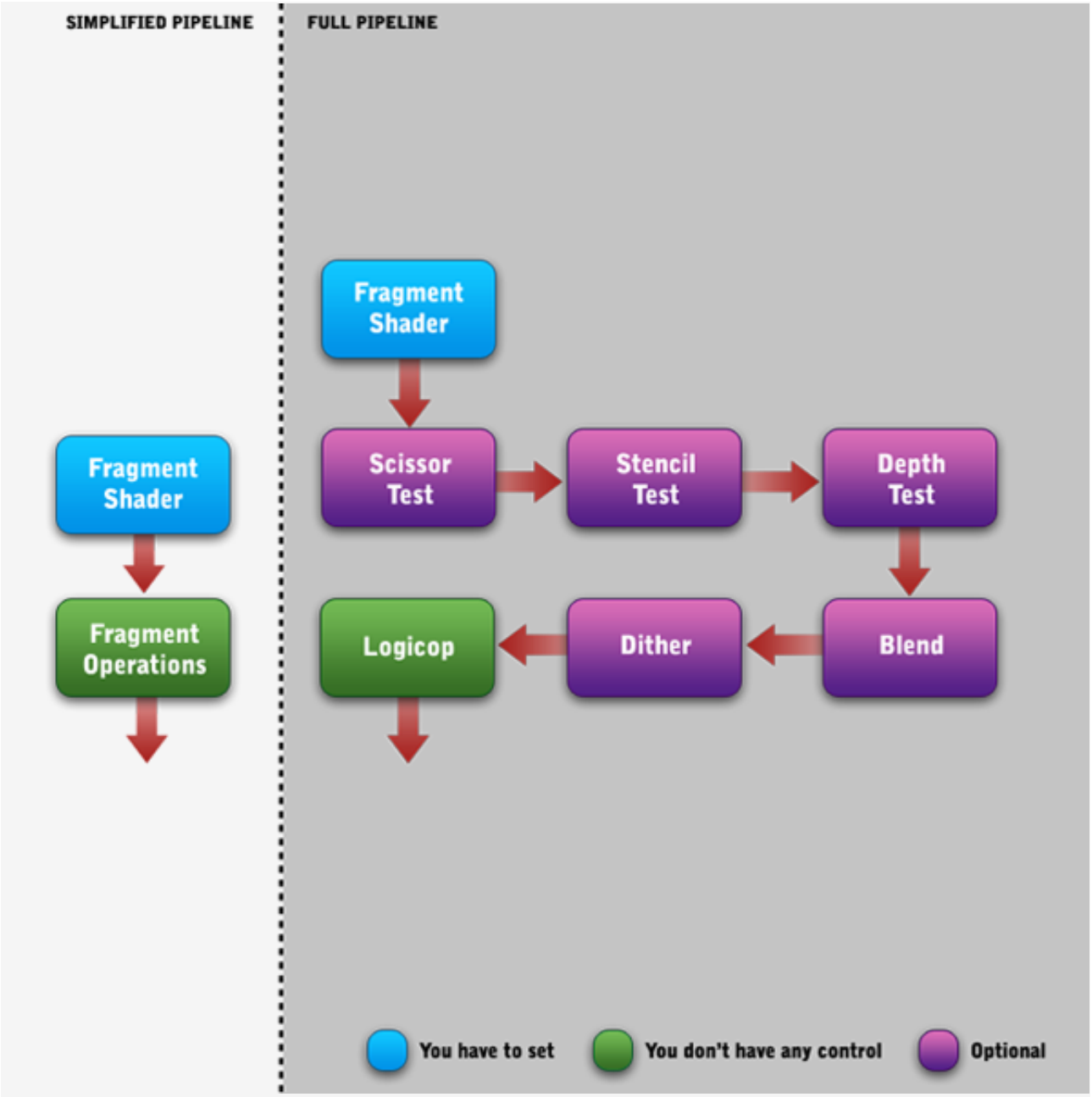
- **mode**: Indicates how the OpenGL will define the front face (and obviously also the back face). This parameter can be:
  - **GL_CCW**: This will instruct OpenGL to treat triangles formed in counter clock wise as a Front Face. This is the default behavior.
  - **GL_CW**: This will instruct OpenGL to treat triangles formed in clock wise as a Front Face.

As you can imagine, if you set the **glCullFace(GL_FRONT)** and **glFrontFace (GL_CW)** you will achieve the same behavior as default. Another way to change the default behavior is by changing the order which your 3D objects are constructed, but of course this is much more laborious, because you need to change your array of indices.
The culling is the first thing to happens in the Rasterize step, so this can determine if a Fragment Shader (the next step) will be processed or not.

Per-Fragment Operations

top

Now let's refine a little our programmable pipeline diagram at the top of this tutorial, more specifically what happens in the process of post fragment shader.



Refined Per-Fragment Operations into the the programmable pipeline.

Between the Fragment Shader and the Scissor Test exist one little omitted step. Something called “Pixel Ownership Test”. This is an internal step. It will decide the ownership of a pixel between the OpenGL internal Frame Buffer and the current EGL’s context. This is an insignificant step to us. You can’t use it to anything, I just told this to you know what happens internally. To us, developers, this step is completely ignored.

As you saw, the only step which you don’t have access is the Logicop. The Logicop is an internal process which includes things like clamp values to 0.0 – 1.0 range, process the final color to the frame buffer after all per-fragment operations, additional Multisample and other kind of internal things. You don’t need to worry about that. We need to focus on purple boxes.

The purple boxes indicate processes which is disabled by default, you need to enable each of them using the**glEnable** function, if you want to use them, of course. You can look again at the **glEnable** parameters, though just to make this point clear, in short words the purple boxes at this image represent the following parameters and means:

- **Scissor Test**: GL_SCISSOR_TEST – This can crop the image, so every fragment outside the scissor area will be ignored.
- **Stencil Test**: GL_STENCIL_TEST – Works like a mask, the mask is defined by a black-white image where the white pixels represent the visible area. So every fragment placed on the black area will be ignored. This requires a stencil render buffer to works.
- **Depth Test**: GL_DEPTH_TEST – This test compares the Z depth of the current 3D object against the others Z depths previously rendered. The fragment with a depth higher than another will be ignored (that means, more distant from the viewer). This will be done using a grey scale image. This requires a depth render buffer to works.
- **Blend**: GL_BLEND – This step can blend the new fragment with the existing fragment into the color buffer.
- **Dither**: GL_DITHER – This is a little OpenGL's trick. In the systems wich color available to the frame buffer is limited, this step can optimize the color usage to appears to have more colors than has in real. The Dither has no configuration, you just choose to use it or not.

To each of them, OpenGL gives few functions to setup the process like **glScissor**, **glBlendColor** or **glStencilFunc**. There are more than 10 functions and I'll not talk about they here, maybe in another article. The important thing to understand here is the process. I told you about the default behavior, like the black and white in stencil buffer, but by using those functions you can customize the processing, like change the black and white behavior on the stencil buffer.
Look again at the programmable pipeline at the top. Each time you render a 3D object, that entire pipeline will occur from the **glDraw*** until the frame buffer, but does not enter in EGL API. Imagine a complex scene, a game scene, like a Counter Strike scene. You could render tens, maybe hundreds 3D object to create only one single static image. When you render the first 3D object, the frame buffer will begin to be filled. If the subsequents 3D objects had their fragments ignored by one or more of the Fragment Operations, then the ignored fragment will not be placed in the frame buffer, but remember that this action will not exclude the fragments which are already in the frame buffer. The final Counter Strike scene is a single 2D image resulting for many shaders, lights, effects and 3D objects. So

every 3D object will have its vertex shader processed, maybe also its fragment shader, but this doesn't means that its resulting image will be really visible.
Well, now you understand why I said the rasterization process include more than only one single step in the diagram. Rasterize is everything between the vertex shader and the frame buffer steps.
Now let's move to the most important section, the shaders!

Shaders

top

Here we are! The greatest invention of 3D world!
If you've read the first part of this serie of tutorials and read all this part until here, I think you have now a good idea of what the shaders are and what they do. Just to refresh our memories, let's remember a little:

- Shaders use the GLSL or GLSL ES, a compact version of the first one.
- Shaders always work in pairs, a Vertex Shader (VSH) and a Fragment Shader (FSH).
- That pair of shader will be processed every time you submit a render command, like **glDrawArrays** or**glDrawElements**.
- VSH will be processed per-vertex, if your 3D object has 8 vertices, so the vertex shader will be processed 8 times. The VSH is responsible by determine the final position of a vertex.
- FSH will be processed in each visible fragment of your objects, remember that FSH is processed before the "Fragment Operations" in the graphics pipeline, so the OpenGL doesn't knows yet what object is in front of others, I mean, even the fragments behind the others will be processed. The FSH is responsible by define the final color of a fragment.
- VSH and FSH must be compiled separately and linked together within a Program Object. You can reuse a compiled shader into multiple Program Objects, but can link only one kind of shader (VSH and FSH) at each Program Object.

Shader and Program Creation

top

OK, first let's talk about the process of creating a shader object, put some source code in it and compile it. As any other OpenGL's object, we first create a nane/id to it and then set its properties. In comparison to the other OpenGL objects, the additional process here is the compiling. Remember that the shaders will be processed by the GPU and to optimize the process the OpenGL compiles your source code into a binary format. Optionally, if you have a previously compiled shader in a binary file you could load it directly instead to load the source and compile. But for now, let's focus on the compiling process.
These are the functions related to shader creation process:

SHADER OBJECT CREATION

GLuint glCreateShader(GLenum type)

- **type**: Indicates what kind of shader will be created. This parameter can be:
  - **GL_VERTEX_SHADER**: To create a Vertex Shader.
  - **GL_FRAGMENT_SHADER**: To create a Fragment Shader.

GLvoid glShaderSource(GLuint shader, GLsizei count, const GLchar** string, const GLint* length)

- **shader**: The shader name/id generated by the **glCreateShader** function.
- **count**: Indicates how many sources you are passing at once. If you are uploading only one shader source, this parameter must be 1.
- **string**: The source of your shader(s). This parameter is a double pointer because you can pass an array of C strings, where each element represent a source. The pointed array should has the same length as the **count** parameter above.
- **length**: A pointer to an array which each element represent the number of chars into each C string of the above parameter. This array must has the same number of elements as specified in the **count** parameter above. This parameter can also be NULL. In this case, each element in the **string** parameter above must be null-terminated.

GLvoid glCompileShader(GLuint shader)

- **shader**: The shader name/id generated by the **glCreateShader** function.

As you saw, this step is easy. You create a shader name/id, make the upload of the source code to it and then compile it. If you upload a source code into a shader which had another source into it, the old source will be completely replaced. Once the shader has been compiled, you can't change the source code anymore using the**glShaderSource**.
Each Shader object has a GLboolean status to indicate if it is compiled or not. This status will be set to TRUE if the shader was compiled with no errors. This status is good for you use in debug mode of your application to check if the shaders are being compiled correctly. Jointly with this check, is a good idea you query the info log which is provided. The functions are **glGetShaderiv** to retrieve the status and **glGetShaderInfoLog** to retrieve the status message. I'll not place the functions and parameters here, but I'll show this shortly in a code example.
Is important tell you the OpenGL names/ids reserved to the shaders are one single list. For example, if you generate a VSH which has the name/id 1 this number will

never be used again, if you now create a FSH, the new name/id will probably be 2 and so on. Never a VSH will has the same name/id of a FSH, and vice versa.
Once you have a pair of shaders correctly compiled is time to create a Program Object to place both shaders into it. The process to create a program object is similar to the shader process. First you create a Program Object, then you upload something (in this case, you place the compiled shaders into it) and finally you compile the program (in this case we don't use the word "compile", we use "link"). The Program will be linked to what? The Program will link the shaders pair togheter and be link itself to the OpenGL's core. This process is very important, because is into it that many verifications on your shaders occur. Just as the shaders, the Programs also has a link status and a link info log which you can use to check the errors. Once a Program was linked with success, you can be sure: your shaders will work correctly. Here are the functions to Program Object:

PROGRAM OBJECT CREATION

GLuint glCreateProgram(void)

- This function requires no parameter. This is because only exist one kind of Program Object, unlike the shaders. Plus, instead to take the memory location to one variable, this function will return the name/id directly, this different behavior is because you can't create more than one Program Object at once, so you don't need to inform a pointer.

GLvoid glAttachShader(GLuint program, GLuint shader)

- **program**: The program name/id generated by the **glCreateProgram** function.
- **shader**: The shader name/id generated by the **glCreateShader** function.

GLvoid glLinkProgram(GLuint program)

- **program**: The program name/id generated by the **glCreateProgram** function.

In the **glAttachShader** you don't have any parameter to identify if the shader is a Vertex or Fragment one. You remember the shaders names/ids are one single list, right?. The OpenGL will atumatically identify the type of the shaders based on their unique names/ids. So the important part is you call the **glAttachShader** twice, one to VSH and other to FSH. If you attach two VSH or two FSH, the program will not be properly linked, also if you attach more than two shaders, the program will fail in linking.
You could create many programs, but how the OpenGL will know which program to use when you call a **glDraw***? The OpenGL's Crane doesn't has an arm and a hook to programs objects, right? So how the OpenGL will know? Well, the

programs are our exception. OpenGL doesn't has a bind function to it, but works with programs at the same way as a hook. When you want to use a program you call this function:

PROGRAM OBJECT USAGE

GLvoid glUseProgram(GLuint program)

- **program**: The program name/id generated by the **glCreateProgram** function.

After calling the function above, every subsequent call to **glDraw*** functions will use the program which is currently in use. As any other **glBind*** function, the name/id 0 is reserved to OpenGL and if you call **glUseProgram(0)** this will unbind any current program.
Now is time to code, any OpenGL application which you create will have a code like this:

```
1   GLuint _program;
2
3   GLuint createShader(GLenum type, const char **source)
4   {
5       GLuint name;
6
7       // Creates a Shader Object and returns its name/id.
8       name = glCreateShader(type);
9
10      // Uploads the source to the Shader Object.
11      glShaderSource(name, 1, &source, NULL);
12
13      // Compiles the Shader Object.
14      glCompileShader(name);
15
16      // If you are running in debug mode, query for info log.
17      // DEBUG is a pre-processing Macro defined to the compiler.
18      // Some languages could not has a similar to it.
19  #if defined(DEBUG)
20
21      GLint logLength;
22
23      // Instead use GL_INFO_LOG_LENGTH we could use COMPILE_STATUS.
24      // I prefer to take the info log length, because it'll be 0 if the
25      // shader was successful compiled. If we use COMPILE_STATUS
26      // we will need to take info log length in case of a fail anyway.
27      glGetShaderiv(name, GL_INFO_LOG_LENGTH, &logLength);
28
29      if (logLength > 0)
30      {
31          // Allocates the necessary memory to retrieve the message.
32          GLchar *log = (GLchar *)malloc(logLength);
```

Here I've made a minimal elaboration to make it more reusable, separating the functions which creates OpenGL objects. For example, instead to rewrite the code of shader creation, we can simply call the function **createShader**and inform the kind of shader we want and the source from it. The same with programs. Of course if you are using an OOP language you could elaborate it much more, creating separated classes for Program Objects and Shader Objects, for example.
This is the basic about the shader and program creation, but we have much more to see. Let's move to the Shader Language (SL). I'll treat specifically the GLSL ES, the compact version of OpenGL Shader Language for Embedded Systems.

Shader Language

top

The shader language is very similar to C standard. The variables declarations and function syntax are the same, the if-then-else and loops has the same syntax too, the SL even accept preprocessor Macros, like #if, #ifdef, #define and others. The shader language was made to be as fast as possible. So be careful about the usage of loops and conditions, they are very expansive. Remember the shaders will be processed by the GPU and the floating-point calculations are optimized. To explore this great improvement, SL has exclusive data types to work with 3D world:

SL'S DATA TYPE	SAME AS C	DESCRIPTION
void	void	Can represent any data type
float	float	The range depends on the precision.
bool	unsigned char	0 to 1
int	char/short/int	The range depends on the precision.
vec2	-	Array of 2 float. {x, y}, {r, g}, {s, t}
vec3	-	Array of 3 float. {x, y, z}, {r, g, b}, {s, t, r}
vec4	-	Array of 4 float. {x, y, z, w}, {r, g, b, a}, {s, t, r, q}
bvec2	-	Array of 2 bool. {x, y}, {r, g}, {s, t}
bvec3	-	Array of 3 bool. {x, y, z}, {r, g, b}, {s, t, r}
bvec4	-	Array of 4 bool. {x, y, z, w}, {r, g, b, a}, {s, t, r, q}
ivec2	-	Array of 2 int. {x, y}, {r, g}, {s, t}
ivec3	-	Array of 3 int. {x, y, z}, {r, g, b}, {s, t, r}
ivec4	-	Array of 4 int. {x, y, z, w}, {r, g, b, a}, {s, t, r, q}
mat2	-	Array of 4 float. Represent a matrix of 2×2.
mat3	-	Array of 9 float. Represent a matrix of 3×3.
mat3	-	Array of 16 float. Represent a matrix of 4×4.
sampler2D	-	Special type to access a 2D texture
samplerCube	-	Special type to access a Cube texture (3D texture)

All the vector data types (**vec***, **bvec*** and **ivec***) can have their elements accessed by either using "." syntax or array subscripting syntax "[x]". In the above table you saw the sequences **{x, y, z, w}, {r, g, b, a}, {s, t, r, q}**. They are the accessors for the vector elements. For example, .xyz could represent the first three elements of a vec4, but you can't use .xyz in a vec2, because .xyz in a vec2 is out of bounds, for a vec2 just .xy could be used. You also can change the order to achieve your results, for example .yzx of a vec4 means you are querying the second, third and first elements, respectively. The reason for three different sequences is because a **vec** data type can be used to represent vectors (x,y,z,w), colors (r,g,b,a) or even texture coordinates (s,t,r,q). The important thing is that you can't mix these sets, for example you can't use .xrt. The following example can help:

```
1  vec4 myVec4 = vec4(0.0, 1.0, 2.0, 3.0);
2  vec3 myVec3;
3  vec2 myVec2;
4
5  myVec3 = myVec4.xyz;        // myVec3 = {0.0, 1.0, 2.0};
6  myVec3 = myVec4.zzx;        // myVec3 = {2.0, 2.0, 0.0};
7  myVec2 = myVec4.bg;         // myVec2 = {2.0, 1.0};
8  myVec4.xw = myVec2;         // myVec4 = {2.0, 1.0, 2.0, 1.0};
9  myVec4[1] = 5.0;            // myVec4 = {2.0, 5.0, 2.0, 1.0};
```

Is very simple.
Now, about the conversions, you need to take care with some things. The SL uses something called Precision Qualifiers to define the range of minimum and maximum values to a data type.
Precision Qualifiers are little instructions which you can use in front of any variable declaration. As any data range, this depends on the hardware capacity. So, the following table is about the minimum range necessary to SL. Some vendors can increase these ranges:

PRECISION	FLOATING POINT RANGE	INTEGER RANGE
lowp	-2.0 to 2.0	-256 to 256
mediump	-16,384.0 to 16,384.0	-1,024 to 1,024
highp	-4,611,686,018,427,387,904.0 to 4,611,686,018,427,387,904.0	-65,536 to 65,536

Instead declare a qualifier at each variable you can also define global qualifiers by using the keyword **precision**. The Precision Qualifiers can help when you need to convert between data types, this should be avoided, but if you really need, use the

Precision Qualifiers to help you. For example, to convert a float to an int you should use a mediump float and a lowp int, if you try to convert a lowp float (range -2.0 to 2.0) to a lowp int all result you will have is between -2 and 2 integers. And to convert you must use a build-in function to the desired data type. The following code can help:

```
1   precision mediump float;
2   precision lowp int;
3
4   vec4 myVec4 = vec4(0.0, 1.0, 2.0, 3.0);
5   ivec3 myIvec3;
6   mediump ivec2 myIvec2;
7
8   // This will fail. Because the data types are not compatible.
9   //myIvec3 = myVec4.zyx;
10
11  myIvec3 = ivec3(myVec4.zyx);    // This is OK.
12  myIvec2.x = myIvec3.y;          // This is OK.
13
14  myIvec2.y = 1024;
15
16  // This is OK too, but the myIvec3.x will assume its maximum value.
17  // Instead 1024, it will be 256, because the precisions are not
18  // equivalent here.
19  myIvec3.x = myIvec2.y;
```

One of the great advantages and performance gain of working directly in the GPU is the operations with the floating-point. You can do multiplications or other operation with the floating-point very easily. Matrices types, vectors types and float type are fully compatibles, respecting their dimensions, of course. You could make complex calculations, like matrices multiplications, in a single line, just like these:

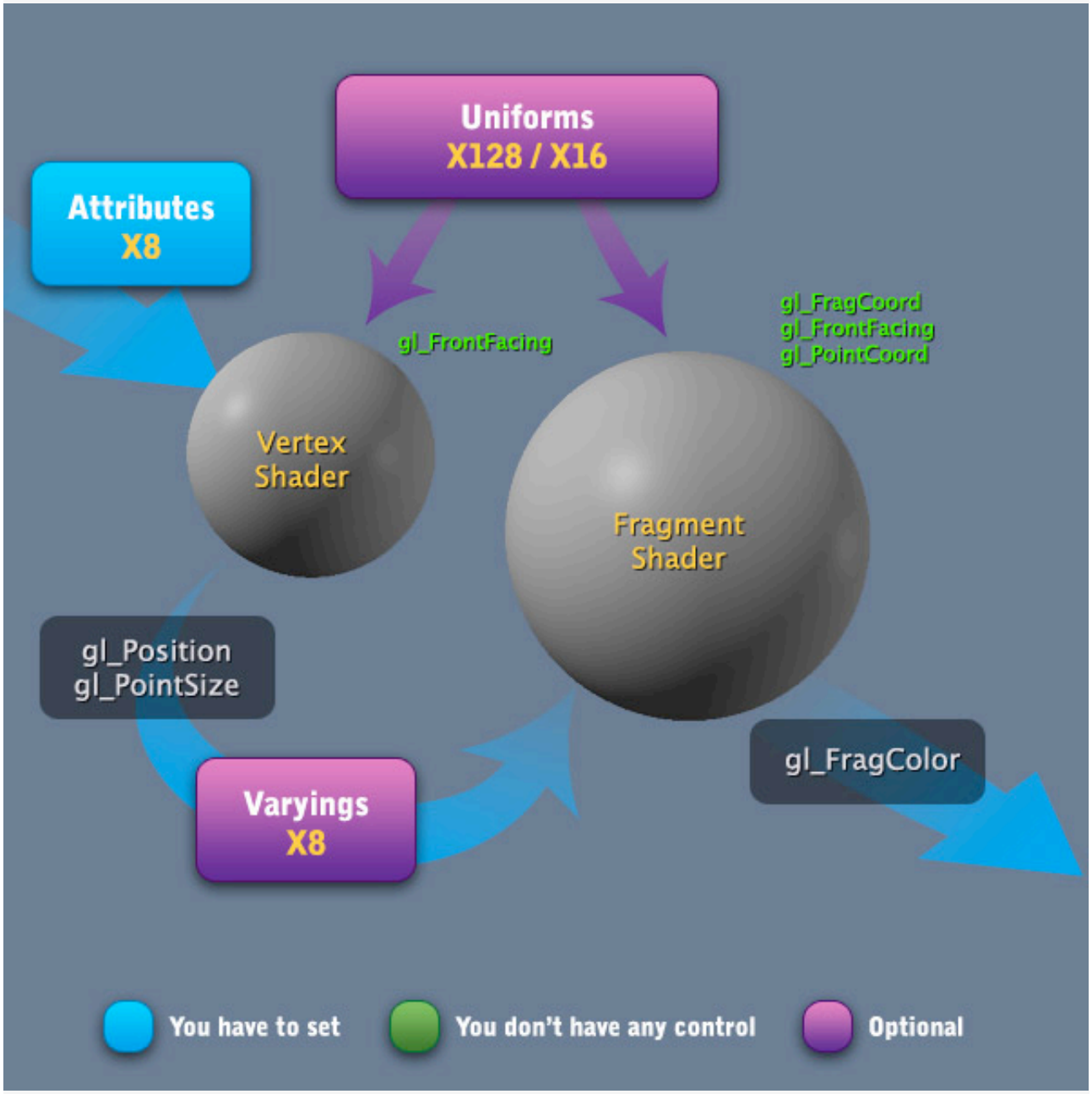
```
1
    mat4 myMat4;
2
    mat3 myMat3;
3
    vec4 myVec4 = vec4(0.0, 1.0, 2.0, 3.0);
4
    vec3 myVec3 = vec3(-1.0, -2.0, -3.0);
5
    float myFloat = 2.0;
6
7
    // A mat4 has 16 elements, could be constructed by 4 vec4.
8
    myMat4 = mat4(myVec4,myVec4,myVec4,myVec4);
9
10  // A float will multiply each vector value.
11  myVec4 = myFloat * myVec4;
12
13  // A mat4 multiplying a vec4 will result in a vec4.
14  myVec4 = myMat4 * myVec4;
15
16  // Using the accessor, we can multiply two vector of different orders.
17  myVec4.xyz = myVec3 * myVec4.xyz;
18
19  // A mat3 produced by a mat4 will take the first 9 elements.
20  myMat3 = mat3(myMat4);
21
22  // A mat3 multiplying a vec3 will result in a vec3.
23  myVec3 = myMat3 * myVec3;
```

You can also use array of any data type and even can construct structs, just like in C. The SL defines which every shader must have one function **void main()**. The shader execution will start by this function, just like C. Any shader which doesn't has this function will not be compiled. A function in SL works exactly as in C. Just remember that SL is an inline language, I mean, if you've wrote a function before call it, is OK, otherwise the call will fail. So if you have more functions in your shader, remember which the **void main()** must be the last to be written.
Now is time to go deeply and understand what exactly the vertex and fragment shaders make.

Vertex and Fragment Structures

top

First of all, let's take a look into the Shaders Pipeline and then I'll introduce you the Attributes, Uniforms, Varyings and Built-In Functions.



The shaders pipeline.

Your VSH should always have one or more Attributes, because the Attributes is used to construct the vertices of your 3D object, only the attributes can be defined per-vertex. To define the final vertex position you'll use the built-in variable gl_Position. If you are drawing a 3D point primitive you could also set the gl_PointSize. Later on, you'll set the gl_FragColor built-in variable in FSH.
The Attributes, Uniforms and Varyings construct the bridge between the GPU processing and your application in CPU. Before you make a render

(call **glDraw*** functions), you'll probably set some values to the Attributes in VSH. These values can be constant in all vertices or can be different at each vertex. By default, any implementation of OpenGL's programmable pipeline must supports at least 8 Attributes.
You can't set any variable directly to the FSH, what you need to do is set a Varying output into the VSH and prepare your FSH to receive that variable. This step is optional, as you saw in the image, but in reality is very uncommon construct a FSH which doesn't receive any Varying. By default, any implementation of OpenGL's programmable pipeline must to support at least 8 Varyings.
Another way to comunicate with the shader is by using the Uniforms, but as the name suggest, the Uniforms are constants throughout all the shaders processing (all vertices and all fragments). A very common usage to uniforms is the samplers, you remember sampler data types, right? They are used to hold our Texture Units. You remember the Texture Units too, right? Just to make this point clear, samplers data types should be like int data types, but is a special kind reserved to work with textures. Just it. The minimum supported Uniforms is different from each shader type. The VSH supports at least 128 Uniforms, but the FSH supports at least 16 Uniforms.
Now, about the Built-In Variables, OpenGL defines few variables which is mandatory to us at each shader. The VSH must define the final vertex position, this is done through the variable **gl_Position**, if current drawing primitive is a 3D point is a good idea to set the **gl_PointSize** too. The **gl_PointSize** will instruct the FSH about how many fragments each point will affect, or in simple words, the size in the screen of a 3D point. This is very useful to make particle effects, like fire. In the VSH has built-in read-only variable, like the **gl_FrontFacing**. This variable is of bool data type, it instructs if the current vertex is front facing or not.
In the FSH the built-in output variable is **gl_FragColor**. For compatibility with the OpenGL's desktop versions, the**gl_FragData** can also be used. **gl_FragData** is an array which is related to the drawable buffers, but as OpenGL ES has only one internal drawable buffer, this variable must always be used as **gl_FragData[0]**. My advice here is to forget it and focus on **gl_FragColor**.
About the read-only built-in variables, the FSH has three of them: **gl_FrontFacing**, **gl_FragCoord** and**gl_PointCoord**. The **gl_FrontFacing** is equal in the VSH, is a bool which indicates if the current fragment is front facing or not. The **gl_FragCoord** is vec4 data type which indicates the fragment coordinate relative to the window (window here means the actual OpenGL's view.
The **gl_PointCoord** is used when you are rendering 3D points. In cases when you specify **gl_PointSize** you can use the **gl_PointCoord** to retrieve the texture

coordinate to the current fragment. For example, a point size is always square and given in pixels, so a size of 16 represent a point formed by 4 x 4 pixels.
The **gl_PointCoord** is in range of 0.0 – 1.0, exactly like a texture coordinate information.
The most important in the built-in output variables is the final values. So you could change the value of **gl_Position**several times in a VSH, the final position will be the final value. The same is true for **gl_FragColor**.
The following table shows the built-in variables and their data types:

BUILT-IN VARIABLE	PRECISION	DATA TYPE
VERTEX SHADER BUILT-IN VARIABLES		
gl_Position	highp	vec4
gl_FrontFacing	-	bool
gl_PointSize	mediump	float
FRAGMENT SHADER BUILT-IN VARIABLES		
gl_FragColor	mediump	vec4
gl_FrontFacing	-	bool
gl_FragCoord	mediump	vec4
gl_PointCoord	mediump	vec2

Is time to construct a real shader. The following code constructs a Vertex and a Fragment Shader which uses two texture maps. Let's start with the VSH.

```
1   precision mediump float;
2   precision lowp int;
3
4   uniform mat4        u_mvpMatrix;
5
6   attribute vec4      a_vertex;
7   attribute vec2      a_texture;
8
9   varying vec2        v_texture;
10
11  void main()
12  {
13      // Pass the texture coordinate attribute to a varying.
14      v_texture = a_texture;
15
16      // Here we set the final position to this vertex.
17      gl_Position = u_mvpMatrix * a_vertex;
18  }
```

And now the corresponding FSH:

```
precision mediump float;
precision lowp int;

uniform sampler2D   u_maps[2];

varying vec2        v_texture;

void main()
{
    // Here we set the diffuse color to the fragment.
    gl_FragColor = texture2D(u_maps[0], v_texture);

    // Now we use the second texture to create an ambient color.
    // Ambient color doesn't affect the alpha channel and changes
    // less than half the natural color of the fragment.
    gl_FragColor.rgb += texture2D(u_maps[1], v_texture).rgb * .4;
}
```

Great, now is time to back to OpenGL API and prepare our Attributes and Uniforms, remember that we don't have directly control to the Varyings, so we must to set an Attribute to be send to a Varying during the VSH execution.

Setting the Attributes and Uniforms

top

To identify any variable inside the shaders, the Program Object defines locations to its variables (location is same as index). Once you know the final location to an Attribute or Uniform you can use that location to set its value.
To setup an Uniform, OpenGL gives to us only one way: after the linking, we retrieve a location to the desired Uniform based on its name inside the shaders. To setup the Attributes, OpenGL gives to us two ways: we could retrieve the location after the program be linked or could define the location before the program be linked. I'll show you the both ways anyway, but set the locations before the linking process is useless, you'll understand why. So let's start with this useless method.

Do you remember that exception to the rule of glBindSomething = “take a container”? OK, here is it. To set an attribute location before the program be linked we use a function which starts with glBindSomething, but in reality the OpenGL’s Port Crane doesn’t take any container at this time. Here the “bind” word is related with the process inside the Program Object, the process of make a connection between an attribute name and a location inside the program. So, the function is:

SETTING ATTRIBUTE LOCATION BEFORE THE LINKAGE

GLvoid glBindAttribLocation(GLuint program, GLuint index, const GLchar* name)

- **program**: The program name/id generated by the **glCreateProgram** function.
- **index**: The location we want to set.
- **name**: The name of attribute inside the vertex shader.

The above method must be called after you create the Program Object, but before you link it. This is the first reason because I discourage doing this. Is a middle step in the Program Object creation. Obviously you can choose the best way to your application. I prefer the next one.
Now let’s see how to get the locations to Attributes and Uniforms after the linking process. Whatever way you choose, you must to hold the location to each shader variable in your application. Because you will need these locations to set its values later on. Here is the functions to use after the linking:

GETTING ATTRIBUTE AND UNIFORM LOCATION

GLint glGetAttribLocation(GLuint program, const GLchar* name)

- **program**: The program name/id generated by the **glCreateProgram** function.
- **name**: The attribute’s name inside the vertex shader.

GLint glGetUniformLocation(GLuint program, const GLchar* name)

- **program**: The program name/id generated by the **glCreateProgram** function.
- **name**: The uniform’s name inside the shaders.

Once we have the locations to our attributes and uniforms, we can do use these locations to set the values we want. OpenGL gives to us 28 different function to set the values of our attributes and uniforms. Those functions are separated in

groups which let you define constant values (uniforms or attributes) or dynamic values (attributes only). To use dynamic attributes you need enable them for a while. You could ask what is the difference between the uniforms, which are always constants, and the constants attributes. Well, the answer is: Good question! Just like the culling GL_FRONT_AND_BACK, this is one thing which I can't understand why the OpenGL continue using it. There are no real difference on the performance of uniforms and constant attributes, or on the memory size and these kind of impacts. So my big advice here is: let the attributes to dynamic values only! If you have a constant value, use the uniforms!
Plus, have two things which make the uniforms the best choice to constant values: Uniforms can be used 128 times in the vertex shader but the attributes just 8 times and the other reason is because attributes can't be arrays. I'll explain this fact later on. For now, although by default the OpenGL does use the attributes as constants, they was not made for this purpose, they was made to be dynamic.
Anyway, I'll show how to set the dynamic attributes, uniforms and even the useless constant attributes. Uniforms can be used with any of the data types or even be a structure or an array of any of those. Here are the functions to set the uniforms values:

DEFINING THE UNIFORMS VALUES

GLvoid glUniform{1234}{if}(GLint location, T value[N])

- **location**: The uniform location retrieved by the **glGetUniformLocation** function.
- **value[N]**: The value you want to set based on the last letter of the function name, *i* = GLint, *f* = GLfloat. You must repeat this parameter N times, according to the number specified in the function name {1234}.

GLvoid glUniform{1234}{if}v(GLint location, GLsizei count, const T* value)

- **location**: The uniform location retrieved by the **glGetUniformLocation** function.
- **count**: The length of the array which you are setting. This will be 1 if you want to set only a single uniform. Values greater than 1 means you want to set values to an array.
- **value**: A pointer to the data you want to set. If you are setting vector uniforms (vec3, for example), each set of 3 values will represent one vec3 in the shaders. The data type of the values must match with the letter in the function name, *i* = GLint, *f* = GLfloat.

GLvoid glUniformMatrix{234}fv(GLint location, GLsizei count, GLboolean transpose, const GLfloat* value)

- **location**: The uniform location retrieved by the **glGetUniformLocation** function.
- **count**: The number of matrices which you are setting. This will be 1 if you want to set only one mat {234} into the shader. Values greater than 1 means you want to set values to an array of matrices, arrays are defined as mat{234}["*count*"] in the shaders.
- **transpose**: This parameter must be GL_FALSE, it is an internal convention just to compatibility with desktop version.
- **value**: A pointer to your data.

Many question, I know… Let me explain it step by step.
The above table shows exactly 19 OpenGL functions. The notation {1234} means you have to write one of these number in the function name, followed by {if} which means you have to choose one of those letters to write the function name and the final “v” of “fv” means you have to write one of both anyway. The [N] in the parameter means you have to repeat that parameter according to the number {1234} in the function name. Here are the complete list of 19 functions:

- **glUniform1i(GLint location, GLint x)**
- **glUniform1f(GLint location, GLfloat x)**
- **glUniform2i(GLint location, GLint x, GLint y)**
- **glUniform2f(GLint location, GLfloat x, GLfloat y)**
- **glUniform3i(GLint location, GLint x, GLint y, GLint z)**
- **glUniform3f(GLint location, GLfloat x, GLfloat y, GLfloat z)**
- **glUniform4i(GLint location, GLint x, GLint y, GLint z, GLint w)**
- **glUniform4f(GLint location, GLfloat x, GLfloat y, GLfloat z, GLfloat w)**
- **glUniform1iv(GLint location, GLsizei count, const GLint* v)**
- **glUniform1fv(GLint location, GLsizei count, const GLfloat* v)**
- **glUniform2iv(GLint location, GLsizei count, const GLint* v)**
- **glUniform2fv(GLint location, GLsizei count, const GLfloat* v)**
- **glUniform3iv(GLint location, GLsizei count, const GLint* v)**
- **glUniform3fv(GLint location, GLsizei count, const GLfloat* v)**
- **glUniform4iv(GLint location, GLsizei count, const GLint* v)**
- **glUniform4fv(GLint location, GLsizei count, const GLfloat* v)**
- **glUniformMatrix2fv(GLint location, GLsizei count, GLboolean transpose, const GLfloat* value)**
- **glUniformMatrix3fv(GLint location, GLsizei count, GLboolean transpose, const GLfloat* value)**

- **glUniformMatrix4fv(GLint location, GLsizei count, GLboolean transpose, const GLfloat* value)**

Uol!!!
By this perspective, can seems so many functions to learn all! But trust me, is not! I prefer look at that table. If I want to set a single uniform which is not of matrix data type I use **glUniform{1234}{if}**according to what I want, 1 = **float/bool/int**, 2 = **vec2/bvec2/ivec2**, 3 = **vec3/bvec3/ivec3** and 4 =**vec4/bvec4/ivec4**. Very simple! If I want to set an array I just place a "v" (of vector) at the end of my last reasoning, so I'll use **glUniform{1234}{if}v**. And finally if what I want is set a matrix data type, being an array or not, I surely will use **glUniformMatrix{234}fv** according to what I want, 2 = **mat2**, 3 = **mat3** and 4 =**mat4**. To define an array you need to remember that the count of your array must be informed to one of above functions by the parameter **count**. Seems more simple now, right?
This is all about how to set an uniform into the shaders. Remember two important things: the same uniform can be used by both shaders, to do this just declare it into both. And the second thing is the most important, the uniforms will be set to the currently program object in use. So you MUST to start using a program before set the uniforms and attributes values to it. To use a program object you remember, right? Just call **glUseProgram** informing the desired name/id.
Now let's see how to set up the values to attributes. Attributes can be used only with the data types **float**, **vec2**,**vec3**, **vec4**, **mat2**, **mat3**, and **mat4**. Attributes cannot be declared as arrays or structures. Following are the functions to define the attributes values.

DEFINING THE ATTRIBUTES VALUES

GLvoid glVertexAttrib{1234}f(GLuint index, GLfloat value [N])

- **index**: The attribute's location retrieved by the **glGetUniformLocation** function or defined with**glBindAttribLocation**.
- **value[N]**: The value you want to set. You must repeat this parameter N times, according to the number specified in the function name {1234}.

GLvoid glVertexAttrib{1234}fv(GLuint index, const GLfloat* values)

- **index**: The attribute's location retrieved by the **glGetUniformLocation** function or defined with**glBindAttribLocation**.
- **values**: A pointer to an array containing the values you want to set up. Only the necessary elements in the array will be used, for example, in case of setting a vec3 if you inform an array of 4 elements, only the first three elements will be used. If the shader need to make automatically fills, it will use the identity of vec4 (x = 0, y = 0, z = 0, z = 1), for example, in case you setting a vec3 if you inform an array of 2 elements, the third elements will be filled with value 0. To matrices, the auto fill will use the matrix identity.

GLvoid glVertexAttribPointer(GLuint index, GLint size, GLenum type, GLboolean normalized, GLsizei stride, const GLvoid* ptr)

- **index**: The attribute's location retrieved by the **glGetUniformLocation** function or defined with**glBindAttribLocation**.
- **size**: This is the size of each element. Here the values can be:
  - **1**: to set up float in shader.
  - **2**: to set up vec2 in shader.
  - **3**: to set up vec3 in shader.
  - **4**: to set up vec4 in shader.
- **type**: Specify the OpenGL data type used in the informed array. Valid values are:
  - **GL_BYTE**
  - **GL_UNSIGNED_BYTE**
  - **GL_SHORT**
  - **GL_UNSIGNED_SHORT**
  - **GL_FIXED**
  - **GL_FLOAT**
- **normalized**: If set to true (GL_TRUE) this will normalize the non-floating point data type. The normalize process will place the converted float number in the range 0.0 – 1.0. If this is set to false (GL_FALSE) the non-floating point data type will be converted directly to floating points.
- **stride**: It means the interval of elements in the informed array. If this is 0, then the array elements will be used sequentially. If this value is greater than 0, the elements in the array will be used respecting this stride. This values must be in the basic machine units (bytes).
- **ptr**: The pointer to an array containing your data.

The above table has the same rules as the uniforms notations. This last table have 9 functions described, which 8 are to set constant values and only one function to set dynamic values. The function to set dynamic values is**glVertexAttribPointer**. Here are the complete list of functions:

- **glVertexAttrib1f(GLuint index, GLfloat x)**
- **glVertexAttrib2f(GLuint index, GLfloat x, GLfloat y)**
- **glVertexAttrib3f(GLuint index, GLfloat x, GLfloat y, GLfloat z)**
- **glVertexAttrib4f(GLuint index, GLfloat x, GLfloat y, GLfloat z, GLfloat w)**
- **glVertexAttrib1fv(GLuint index, const GLfloat* values)**
- **glVertexAttrib2fv(GLuint index, const GLfloat* values)**
- **glVertexAttrib3fv(GLuint index, const GLfloat* values)**
- **glVertexAttrib4fv(GLuint index, const GLfloat* values)**
- **glVertexAttribPointer(GLuint index, GLint size, GLenum type, GLboolean normalized, GLsizei stride, const GLvoid* ptr)**

The annoying thing here is the constant value is the default behavior to the shaders, if you want to use dynamic values to attributes you will need to temporarily enable this feature. Dynamic values will be set as per-vertex. You must to use the following functions to enable and disable the dynamic values behavior:

VARIABLE VALUES FEATURE

GLvoid glEnableVertexAttribArray(GLuint index)

- **index**: The attribute's location retrieved by the **glGetUniformLocation** function or defined with**glBindAttribLocation**.

GLvoid glDisableVertexAttribArray(GLuint index)

- **index**: The attribute's location retrieved by the **glGetUniformLocation** function or defined with**glBindAttribLocation**.

So, before using **glVertexAttribPointer** to define per-vertex values to the attributes, you must enable the location of the desired attribute to accept dynamic values by using the **glEnableVertexAttribArray**.
To the pair of VSH and FSH shown early, we could use the following code to setup their values:

```
1
2   // Assume which _program defined early in another code example.
3
4   GLuint mvpLoc, mapsLoc, vertexLoc, textureLoc;
5
6   // Gets the locations to uniforms.
7   mvpLoc = glGetUniformLocation(_program, "u_mvpMatrix");
8   mapsLoc = glGetUniformLocation(_program, "u_maps");
9
10  // Gets the locations to attributes.
11  vertexLoc = glGetAttribLocation(_program, "a_vertex");
12  textureLoc = glGetAttribLocation(_program, "a_texture");
13
14  // ...
15  // Later, in the render time...
16  // ...
17
18  // Sets the ModelViewProjection Matrix.
19  // Assume which "matrix" variable is an array with
20  // 16 elements defined, matrix[16].
21  glUniformMatrix4fv(mvpLoc, 1, GL_FALSE, matrix);
22
23  // Assume which _texture1 and _texture2 are two texture names/ids.
24  glBindTexture(GL_TEXTURE_2D, _texture1);
25  glActiveTexture(GL_TEXTURE0);
26  glBindTexture(GL_TEXTURE_2D, _texture2);
27  glActiveTexture(GL_TEXTURE1);
28
29  // The {0,1} correspond to the activated textures units.
30  int textureUnits[2] = {0,1};
31
32  // Binds the texture units to an uniform.
    glUniform1iv(mapsLoc, 2, &textureUnits);
```

I had enabled and disabled the dynamic values to attributes just to show you how to do. As I said before, enable and disable features in OpenGL are expansive tasks, so you could want enable the dynamic values to the attributes once, maybe in time you get its location, for example. I prefer enable them once.

Using the Buffer Objects

top

To use the buffer objects is very simple! All that you need is bind the buffer objects again. Do you remember the buffer object hook is a double one? So you can bind a **GL_ARRAY_BUFFER** and a**GL_ELEMENT_ARRAY_BUFFER** at the same time. Then you call the **glDraw*** informing the starting index of the buffer object which you want to initiate. You'll need to inform the start index instead of an array data, so the start number must be a pointer to void. The start index must be in the basic machine units (bytes).
To the above code of attributes and uniforms, you could make something like this:

```
1
    GLuint arrayBuffer, indicesBuffer;
2
3
    // Generates the name/ids to the buffers
4
    glGenBuffers(1, &arrayBuffer);
5
    glGenBuffers(1, &indicesBuffer);
6
7
    // Assume we are using the best practice to store all informations about
8
    // the object into a single array: vertices and texture coordinates.
9
    // So we would have an array of {x,y,z,s,t,  x,y,z,s,t,  ...}
10
    // This will be our "arrayBuffer" variable.
11
    // To the "indicesBuffer" variable we use a
12
    // simple array {1,2,3,  1,3,4,  ...}
13
14
    // ...
15
    // Proceed with the retrieving attributes and uniforms locations.
16
    // ...
17
18
    // ...
19
    // Later, in the render time...
20
    // ...
21
22
    // ...
23
    // Uniforms definitions
24
    // ...
25
26
    glBindBuffer(GL_ARRAY_BUFFER, arrayBuffer);
27
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indicesBuffer);
28
29
    int fsize = sizeof(float);
30
    GLsizei str = 5 * fsize;
31
    void * void0 = (void *) 0;
32
    void * void3 = (void *) 3 * fsize;
```

If you are using an OOP language you could create elegant structures with the concepts of buffer objects and attributes/uniforms.
OK, those are the basic concepts and instructions about the shaders and program objects. Now let’s go to the last part (finally)! Let’s see how to conclude the render using the EGL API.

Rendering

top

I'll show the basic kind of render, a render to the device's screen. As you noticed before in this serie of tutorials, you could render to an off-screen surfaces like a frame buffer or a texture and then save it to a file, or create an image in the device's screen, whatever you want.

Pre-Render

top

I like to think in the rendering as two steps. The first is the Pre-Render, in this step you need to clean any vestige from the last render. This is important because exists conservation in the frame buffers. You remember what is a frame buffer, right? A colletion of images from render buffers. So when you make a complete render, the images in render buffer still alive even after the final image have been presented to its desired surface. What the Pre-Render step does is just clean up all the render buffers. Unless you want, for some reason, reuse the previous image into the render buffers.
Make the clean up in the frame buffer is very simple. This is the function you will use:

CLEARING THE RENDER BUFFERS

GLvoid glClear(GLbitfield mask)

- **mask**: The mask represent the buffers you want to clean. This parameter can be:
  - **GL_COLOR_BUFFER_BIT**: To clean the Color Render Buffer.
  - **GL_DEPTH_BUFFER_BIT**: To clean the Depth Render Buffer.
  - **GL_STENCIL_BUFFER_BIT**: To clean the Stencil Render Buffer.

As now you know well, every instruction related to one of the Port Crane Hooks will affect the last object bound. So before call the above function, make sure you have bound the desired frame buffer. You can clean many buffers at once, as the mask parameter is bit informations, you can use the bitwise operator OR "|". Something like this:

```
1   glBindFramebuffer(GL_FRAMEBUFFER, _frameBuffer);
2   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

OpenGL also gives to us others functions to make the clean up. But the above function is pretty good for any cases.

The Pre-Render step should be called before any **glDraw*** calls. Once the render buffer is clean, is time to draw your 3D objects. The next step is the drawing phase, but it is not one of the two render steps I told you before, it is just the drawing.

Drawing

top

I've showed it several times before in this tutorial, but now is time to drill deep inside it. The triggers to drawing in OpenGL is composed by two functions:

CLEARING THE RENDER BUFFERS

GLvoid glDrawArrays(GLenum mode, GLint first, GLsizei count)

- **mode**: This parameter specify which primitive will be rendered and how its structure is organized. This parameter can be:
  - **GL_POINTS**: Draw points. Points are composed by single sequences of 3 values (x,y,z).
  - **GL_LINES**: Draw Lines. Lines are composed by two sequences of 3 values (x,y,z / x,y,z).
  - **GL_LINE_STRIP**: Draw Lines forming a strip. Lines are composed by two sequences of 3 values (x,y,z / x,y,z).
  - **GL_LINE_LOOP**: Draw Lines closing a loop of them. Lines are composed by two sequences of 3 values (x,y,z / x,y,z).
  - **GL_TRIANGLES**: Draw Triangles. Triangles are composed by three sequences of 3 values (x,y,z / x,y,z / x,y,z).
  - **GL_TRIANGLE_STRIP**: Draw Triangles forming a strip. Triangles are composed by three sequences of 3 values (x,y,z / x,y,z / x,y,z).
  - **GL_TRIANGLE_FAN**: Draw Triangles forming a fan. Triangles are composed by three sequences of 3 values (x,y,z / x,y,z / x,y,z).
- **first**: Specifies the starting index in the enabled vertex arrays.
- **count**: Represents number of vertices to be draw. This is very important, it represents the number of vertices elements, not the number of the elements in the array of vertices, take care to don't confuse both. For example, if you are drawing a single triangle this should be 3, because a triangle is formed by 3 vertices. But if you are drawing a square (composed by two triangles) this should be 6, because is formed by two sequences of 3 vertices, a total of 6 elements, and so on.

GLvoid glDrawElements(GLenum mode, GLsizei count, GLenum type, const GLvoid* indices)

- **mode**: This parameter specify which primitive will be rendered and how its structure is organized. This parameter can be:
  - **GL_POINTS**: Draw points. Points are composed by single sequences of 3 values (x,y,z).
  - **GL_LINES**: Draw Lines. Lines are composed by two sequences of 3 values (x,y,z / x,y,z).
  - **GL_LINE_STRIP**: Draw Lines forming a strip. Lines are composed by two sequences of 3 values (x,y,z / x,y,z).
  - **GL_LINE_LOOP**: Draw Lines closing a loop of them. Lines are composed by two sequences of 3 values (x,y,z / x,y,z).
  - **GL_TRIANGLES**: Draw Triangles. Triangles are composed by three sequences of 3 values (x,y,z / x,y,z / x,y,z).
  - **GL_TRIANGLE_STRIP**: Draw Triangles forming a strip. Triangles are composed by three sequences of 3 values (x,y,z / x,y,z / x,y,z).
  - **GL_TRIANGLE_FAN**: Draw Triangles forming a fan. Triangles are composed by three sequences of 3 values (x,y,z / x,y,z / x,y,z).
- **count**: Represents number of vertices to be draw. This is very important, it represents the number of vertices elements, not the number of the elements in the array of vertices, take care to don't confuse both. For example, if you are drawing a single triangle this should be 3, because a triangle is formed by 3 vertices. But if you are drawing a square (composed by two triangles) this should be 6, because is formed by two sequences of 3 vertices, a total of 6 elements, and so on.
- **type**: Represent the OpenGL's data type which is used in the array of indices. This parameter can be:
  - **GL_UNSIGNED_BYTE**: To indicate a GLubyte.
  - **GL_UNSIGNED_SHORT**: To indicate a GLushort.

Many questions, I know. First let me introduce how these functions work. One of the most important things in the programmable pipeline is defined here, the number of times which the VSH will be executed! This is done by the parameter *count*. So if you specify 128 to it, the currently program in use will process its VSH 128 times. Of course, the GPU will optimize this process so much as possible, but in general words, your VSH will be processed 128 times to process all your defined attributes and uniforms. And why I said to take care about the difference between number of vertices elements and number of the elements in the array of vertices? Is simple, you could have an array of vertices with 200 elements, but for some reason you want to construct just a triangle at this time, so *count*will be 3, instead 200. This is even more usefull if you are using an array of

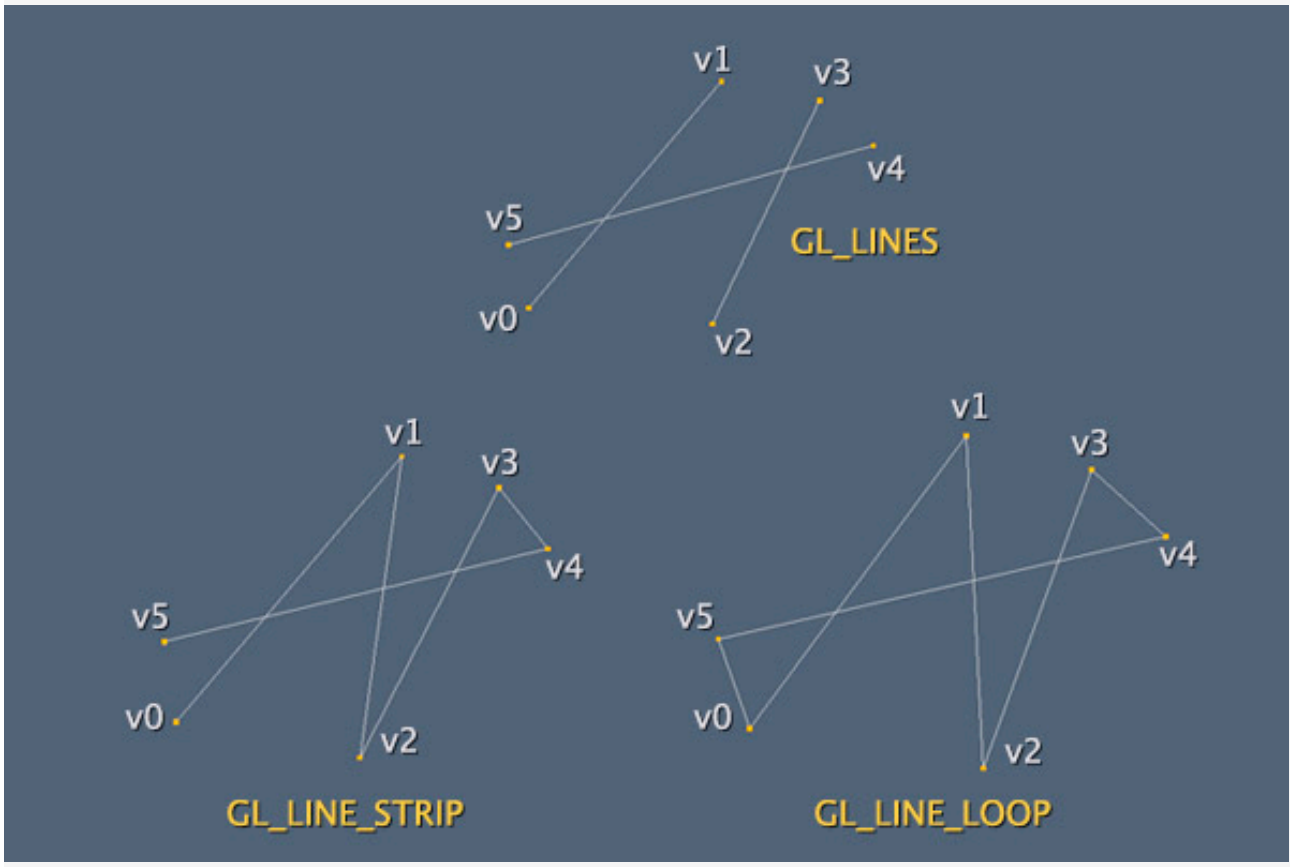
indices. You could have 8 elements in the array of vertices, but the array of indices specifies 24 elements, in this case the parameter *count* will be 24. In general words, is the number of vertices elements you want to draw.

If you are using the **glDrawArrays** the *first* parameter works like an initial stride to your per-vertex attributes. So if you set it to 2, for example, the values in the vertex shader will start by the index 2 in the array you specified in**glVertexAttribPointer**, instead start by 0 as default.

If you are using the **glDrawElements** the *first* parameter will work like an initial stride to the array of indices, and not directly to your per-vertex values.

The *type* identifies the data type, in really it's an optimization hint. If your array of indices has less than 255 elements is a very good idea to use **GL_UNSIGNED_BYTE**. Some implementations of OpenGL also support a third data type: **GL_UNSIGNED_INT**, but this is not very common.

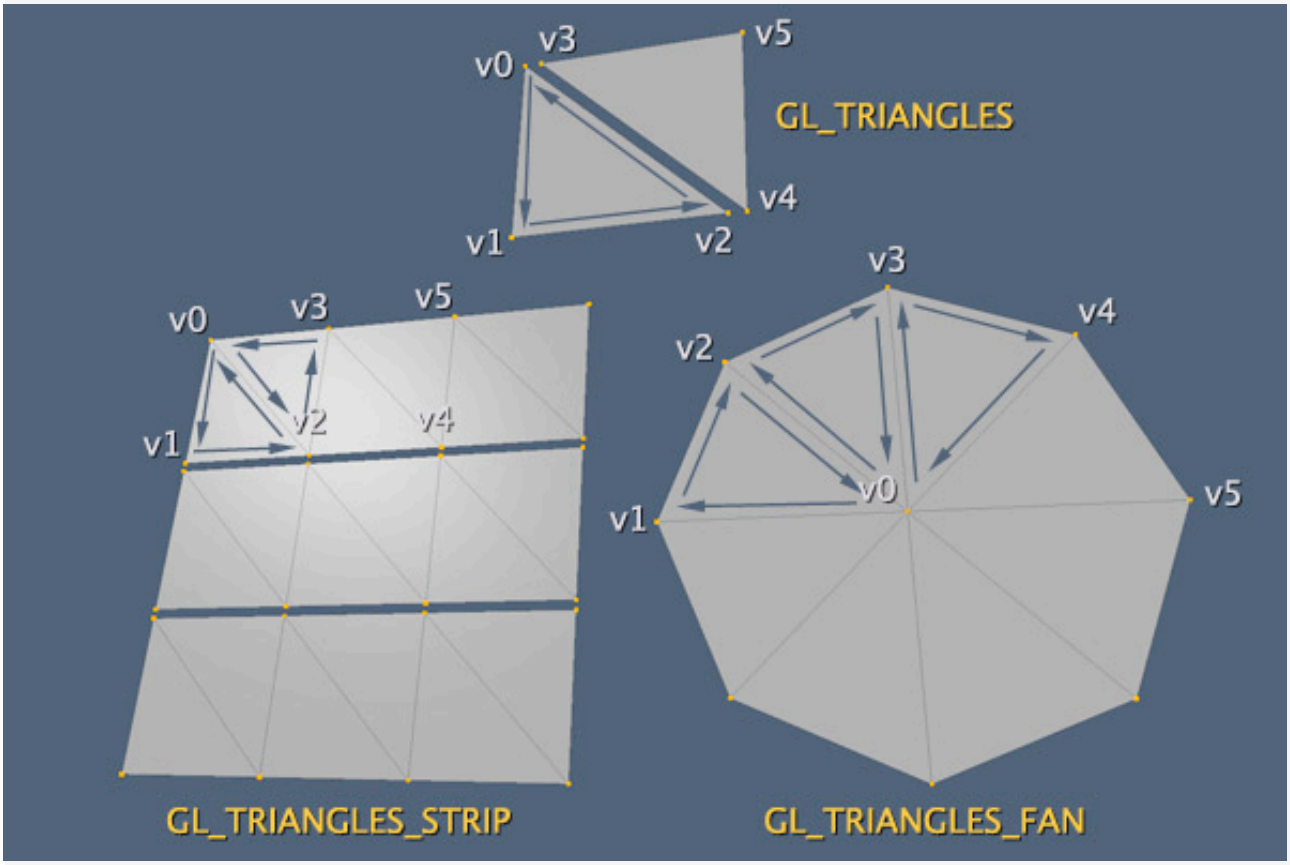
OK, now let's talk about the construction modes, defined in the **mode** parameter. It's a hint to use in the construction of your meshes. But the modes are not so useful to all kind of meshes. The following images could help you to understand:



Line construction modes.

The image above shows what happens when we draw using one of the line drawing modes. All the drawing in the image above was made with the sequence of {v0,v1,v2,v3,v4,v5} as the array of vertices, supposing which each vertex has an unique values to x,y,z coordinates. As I told before, the unique mode which is

compatible with any kind of draw is the GL_LINES, the others modes is for optimization in some specific situations. Optimize? Yes, look, using the GL_LINES the number of drawn lines was 3, using GL_LINE_STRIP the number of drawn lines was 5 and with GL_LINE_LOOP was 6, always using the same array of vertices and the same number of VSH loops.
Now the drawing modes to triangles are similar, look:



Triangle construction mode.

The image above shows what happens when we draw using one of the triangles drawing modes. All the indicated drawing in the image was made with the sequence of {v0,v1,v2,v3,v4,v5} as the array of vertices, supposing which each vertex has an unique values to x,y,z coordinates. Here again, the same thing, just the basic **GL_TRIANGLES**is useful to any kind of mesh, the others modes is for optimization in some specific situations. Using the**GL_TRIANGLES_STRIP** we need to reuse the last formed line, in the above example we must to draw using an array of index like {0,1,2, 0,2,3, 3,2,4,…}. Using the **GL_TRIANGLES_FAN** we must to always return to the first vertex, in the image we could use {0,1,2, 0,2,3, 0,3,4, …} as our array of indices.
My advice here is to use **GL_TRIANGLES** and **GL_LINES** as much as possible. The optimization gain of **STRIP**,**LOOP** and **FAN** could be achieved by optimizing the OpenGL draw in other areas, with other techniques, like reducing the number of polygons into your meshes or optimizing your shaders processing.

Render

top

This last step is just to present the final result of the frame buffer to the screen, even if you are not explicitely using a frame buffer. I've explained this on my EGL article. If you lost it, click here to egl or if you are using Objective-C click here to eagl.

So I will not repeat the same content here. But I want to remember for who of you which are working on EAGL API, before to call the **presentRenderbuffer:GL_RENDERBUFFER**, you must to bind the color render buffer and, obviously, you need to bind the frame buffer too. This is because the render buffer stay "inside" the frame buffer, you remember it, right? The final code will be something like this:

```
1  - (void) makeRender
2  {
3      glBindFramebuffer(_framebuffer);
4      glBindRenderbuffer(_colorRenderbuffer);
5      [_context presentRenderbuffer:GL_RENDERBUFFER];
6  }
```

For who of you which are using the EGL API, the process is just swap the internal buffers, using the **glSwapBufers**function. At my EGL article I explain that too.

OK guys! This is the basic about the render. OpenGL also provides something called Multisample, is a special kind of render to produce anti-aliased images. But this is an advanced discussion. I'll let it to the next part.

This tutorial is already long enough, so let's go to the conclusion and our final revision.

Conclusion

top

Here we are! I don't know how many hours you spent reading this, but I have to admit, is a big tiring reading. So I want thank you! Thank you for reading. Now, as we are used, let's remember from everything:

- First we saw the OpenGL's data types and the programmable pipeline.
- Your meshes (primitives) data must be an array of informations. It should be optimized to form an Array of Structures.
- The OpenGL works as a Port Crane which has multiple arms with hooks. Four great hooks: Frame Buffer Hook, Render Buffer Hook, Buffer Object Hook and Texture Hook.
- The Frame Buffers holds 3 Render Buffers: Color, Depth and Stencil. They form an image coming from the OpenGL's render.
- Textures must to have specific formats before be uploaded to OpenGL (a specific pixel order and color bytes per pixel). Once an OpenGL's texture was created, you need to activate a Texture Unit which will be processed in the shaders.
- The Rasterize is a big process which includes several tests and per-fragment operations.
- Shaders are used in pairs and must be inside a Program Object. Shaders use their own language called GLSL or GLSL ES.
- You can define dynamic values only in Attributes into VSH (per-vertex values). The Uniforms are always constant and can be used in both shaders kind.
- You need to clean the buffers before start to draw new things to it.
- You will call a **glDraw*** function to each 3D object you want to render.
- The final render step is made using EGL API (or EAGL API, in iOS cases).

My last advice is to you check again the most important points, if you have some doubt, just ask, let a comment bellow and if I could help I'll be glad.

On the Next

All I showed here is an intermediate level about OpenGL. Using these knowledge you can create great 3D applications. I think this is around the half of the OpenGL study. In the next part, an advanced tutorial, I'll talk about everything I skipped here: the 3D textures, multisampling, render to off-screen surfaces, per-fragment operations in deeply and some best practices I've learned developing my 3D engine.

Probably I'll make another two articles before the third part of this serie. One about the textures, the complexity of binary images and compressed formats like PVR textures. And another one about the ModelViewProjection Matrix, Quaternions and matrices operations, is more a mathematical article, for who likes, will appreciate.

Thanks again for reading and see you in the next part!