Graphics pipelines for mobile and embedded devices

Leif Andersen
University of Utah

***Abstract*—Mobile and embedded systems continually require more powerful graphics. The most common API and hardware implementation for these graphics is OpenGL ES. Different versions of OpenGL ES has different types of pipelines that allow the programmer different amounts of flexibility. Various implementations of OpenGL can also be optimized for embedded devices, as well as portability, but at the cost of complexity for the programmer.**

I. Introduction

Mobile and embedded systems are continually requiring more powerful computational and graphics hardware. Cell phones of the past simply needed to display a number on an LCD screen. Today, phones, such as the ones running Google's Android and Apple's iOS operating systems are expected to run a wide variety of computationally, and graphics heavy tasks, such as games [1]. OpenGL is one of two standard graphics Application programming interface (API) used for this task, the other being DirectX. OpenGL ES is a version of OpenGL for embedded systems. Most OpenGL ES code will work with standard OpenGL, however OpenGL ES is missing many standard API calls, as well as commonly used libraries, such as GLU [2].

There are two major versions of OpenGL ES. They are:

1) OpenGL ES 1.x
2) OpenGL ES 2.0

Unlike standard OpenGL, newer versions of OpenGL ES are not backwards compatible with older versions. OpenGL ES has other limitations, the rest of which depend on specifically on the graphics hardware.

OpenGL takes data through several steps before it will be displayed on the screen. First it goes a series of linear transformations, and reconstructs the data, (takes data in an arbitrary subspace into one that OpenGL understands). After that, it rasterizes the data, which turns it from vector data, to bitmap data, such as pixels. From there, it determines the color of the pixel, and puts it through several more buffers. Finally, the data is put into a frame buffer, which can be shown on the screen.

Section II will cover the required mathematics to make the linear transformations, as well as rasterizing and simple texturing. Section III will compare the deprecated fixed function pipeline in OpenGL ES 1.x, to the more modern Programmable Pipeline in OpenGL ES 2.0. Section IV will discuss some of the limitations of graphics on current embedded systems. Finally, section V will cover commonly used texture compression schemes, and how it effects the speed of the rendering.

II. Required Mathematics

There are two major stages to modern graphics pipelines. They are:

1) Determining the geometry of the objects being drawn.
2) Determining the texture, lighting, colors, and other particles of the objects and environment.

A. Geometry

The shape of an object in three space is defined as a polygon mesh, or simply just mesh. This is a list of vertices in three space, with edges and faces connecting them. Standard OpenGL supports many different types of these: triangles, triangle strips, triangle fans, quads, etc. [3]. Each of these objects can be represent as list of vertices, and each vertex can be represented as four integers, equation 1.

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{1}$$

The first three numbers in the vector store the (x, y, z) values of the vector. The last element of the vector is almost always saved as 1. This allows the vector in equation 1 to be manipulated using the equations: 2, translation, 3 scale, and 4 rotation. Where, equation 4 uses c from equation 5 and s from equation 6.

$$\begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \\ z_0 \\ 1 \end{bmatrix} = \begin{bmatrix} x + x_0 \\ y + y_0 \\ z + z_0 \\ 1 \end{bmatrix} \tag{2}$$

$$\begin{bmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \\ z_0 \\ 1 \end{bmatrix} = \begin{bmatrix} x \times x_0 \\ y \times y_0 \\ z \times z_0 \\ 1 \end{bmatrix} \tag{3}$$

$$\begin{bmatrix} x^2(1-c)+c & xy(1-c)-zs & xz(1-c)+ys & 0 \\ yx(1-c)+zs & y^2(1-c)+c & yz(1-c)-ys & 0 \\ zx(1-c)-ys & yz(1-c)+xs & z^2(1-c)+c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4}$$

$$c = cos(angle) \tag{5}$$

$$s = sin(angle) \tag{6}$$

In equation 2, the vector (x_0, y_0, z_0) is translated by the vector (x, y, z). In the equation 3, the vector (x_0, y_0, z_0) is scaled up by a factor of (x, y, z). Finally, in equation 4, the vector (x_0, y_0, z_0) (not shown), is rotated around the axis (x, y, z). To prevent the object from having extraneous translations, it is best to preform the rotation and scale before the translation is completed, assuming the mesh is placed at the origin.

In a modern application, vertices will need to go through several transformations before they can be drawn on the screen. A mesh can go from mesh coordinates, which describes the mesh from some normalized axis, to world coordinates, which places each object relative to each other object, to projection coordinates, which moves the world to be viewable to the camera. There can also be a final translation to clip coordinates, which is what the graphics library and hardware uses to determine if a mesh should be drawn. In OpenGL ES, this is defined as $[-1.0, 1.0]$ in the x, y, and z coordinates [2]. As these matrices are affine transforms, they can be collapsed down to a single matrix which the vector can be put through. Furthermore, because a mesh object is formed of many vertices, every vertex in the mesh can be put through the same matrix. Figure 1 demonstrates this process.

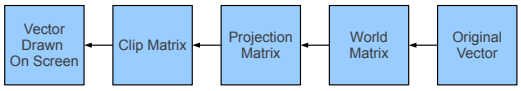

Fig. 1: Standard Vertex Shader Pipe

B. Texturing

The second thing that must be completed is the texturing phase. This is where any lighting and particle effects are done. This phase runs through the process of turning the vertex data determined in the previous section into pixels that can be drawn on the screen. Only texturing that requires extremely accurate ray tracing requires the value of other pixels. As such, every pixel can be done independently, in an individual thread.

A simple algorithm for determining the color of the face could simply keep a collection of the solid color for each face, then determine which face is being drawn, and assign that color to the pixel. A more robust algorithm would store the image on the graphics card, and find the proper location on that image for the given face, and draw that color of the pixel on the screen. An even more elaborate one would involve dimming or brightening the color based on lighting, and even surrounding colors.

III. Pipelines

OpenGL ES provides several different pipelines for producing the graphics shown on screen. A pipeline takes some data, and possibly some code, and turns it into an image shown on screen. The two major types of pipelines are fixed function pipelines and programmable pipelines. Desktop OpenGL 2.0 and higher supports both types of pipelines. However, Khronos decided that having both pipelines would be redundant for mobile platforms. In OpenGL ES 2.0, the fixed function pipeline was removed, and was replaced with a programmable one [2].

Fewer devices are capable of running OpenGL ES 2.0 than OpenGL ES 1.x. If the application in question is being designed for older hardware, than 1.x should be used. However, with newer hardware, OpenGL ES 2.0 can be used to greatly speed up performance by creating a pipeline that provides only the needed functionality. In cases where work is being done for a broad target range, such as the Android platform, currently, over 90% of android devices are capable of running OpenGL ES 2.0 [4]. In addition, the iPhone 3G S, iPod Touch 3, and iPad, as well as all newer mobile devices from Apple, are capable of OpenGL ES 2.0 [5].

A. Fixed Function Pipeline

Figure 2 demonstrates the fixed function pipeline in opengl.

The Fixed function pipeline simply takes vertices into the pipe, loads some matrices, images, and other data into the graphics processor, and outputs a framebuffer of pixels. The simplest way to do this is to run OpenGL in immediate mode using calls to **glBegin()** and **glEnd()**. While this is the easiest way for the programmer to prototype an application quickly, it is significantly slower at run time. This is due to the cross talk between the CPU and the GPU, as the CPU needs to tell the GPU to do anything, as well as constantly load the vertex data onto the GPU. A more efficient solution would be to load the data onto the GPU in a single array (called a vertex buffer object (VBO), and have the GPU preform it's calculations [3].

In OpenGL ES 1.x, the latter way is the only supported method of drawing data. This is because Khronos found that having both modes would be redundant. Furthermore, the only time **glBegin()** and **glEnd()** would be useful is when the CPU needs to do rather unusual computations that the normal graphics pipe does not do very well. These applications are not the primary focus of such embedded devices [6].

B. Programmable Pipeline

Figure 3 is an example of the OpenGL ES 2.0 pipeline.

Unlike the OpenGL ES 1.x pipeline, this pipeline is programmable. Unlike the desktop edition of OpenGL, it must be programmed. The missing portions of the pipeline in figure 3 that were in figure 2 can be duplicated with code that the user types. Unlike the CPU of the mobile device, many programs need to be written to support multiple types of graphics cards. As such, the programs that are loaded onto the graphics cards, called shaders [7], are stored in memory as source code. Each of the programs are then compiled and loaded onto the graphics card, where they are linked into the main pipe [2][4].

The following listing shows an example of a simple vertex shader.

```
uniform mat4 uMVP;
attribute vec4 aPosition;
attribute vec4 aNormal;
```

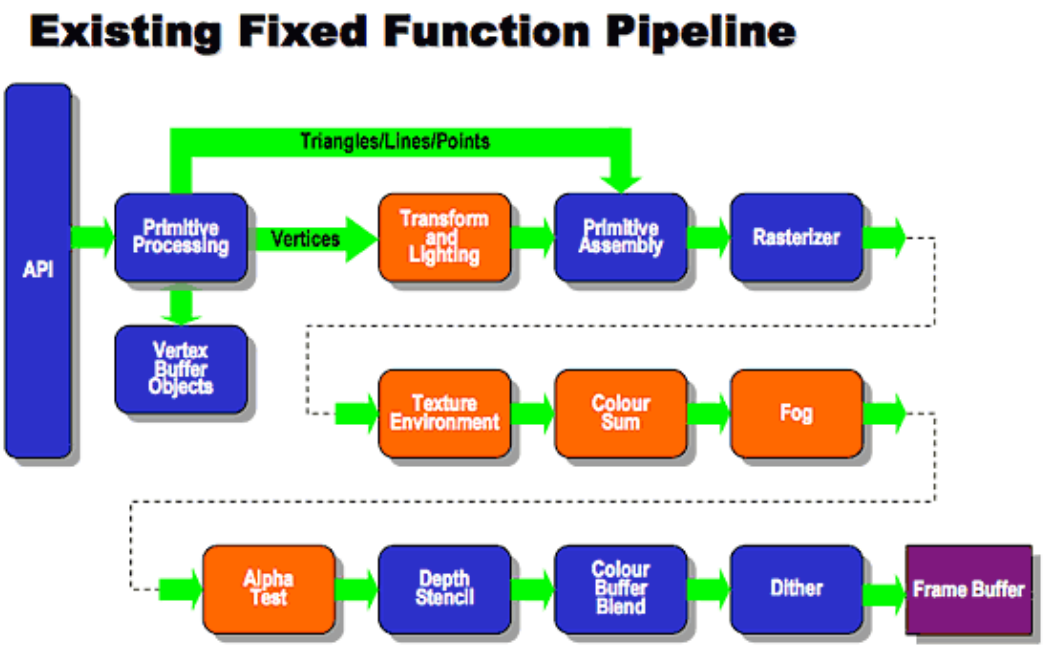

Fig. 2: OpenGLES Fixed Function Pipeline[2]

```
attribute vec2 aTexCoord;
varying vec2 vTexCoord;
varying vec4 vPrimaryColor;
void main() {
  gl_Position = uMVP * aPosition;
  vPrimaryColor = vec4(1.0, 1.0,
                       1.0, 1.0);
  vTexCoord = aTexCoord;
}
```

The variable **gl_Position** is the final output of the vertex on the screen. **uMVP** is the matrix the input vertex is multiplied by to get the final position. The other two variables of type varying are simply passed onto the fragment shader [2].

The following listing shows an example of a simple fragment shader.

```
precision mediump float;
uniform sampler2D sTex;
varying vec2 vTexCoord;
varying vec4 vPrimaryColor;
void main() {
  gl_FragColor = texture2D(sTex,
                 vTexCoord)*vPrimaryColor;
}
```

The variable **gl_FragColor** is used to tell OpenGL what color to make the pixel being drawn. the **texture2D()** function tells OpenGL where in the provided texture given the stored texture mappings to find the image. The multiplication **vPrimaryColor** is simply to state that we want no lighting. A more advanced shader would then multiply the pixel by the proper lighting needed to make an image appear to be three dimensional [2].

More intelligent shader programs can use techniques such as loop unrolling and precision control for a speed up of over eight times the performance of the naive algorithms [8].

As with OpenGL ES 1.x, it is best to load data onto the GPU as an array. **glBegin()** and **glEnd()** are not even supported in openGL ES 2.0. An even faster way to do this would be to load all of the data onto the GPU at the beginning of the program, and store it into Vertex Buffer Objects, or VBOs. This way, all the CPU has to provide is the matrix for the change in perspective [3]. VBOs are not required in OpenGL ES 2.0, however, every major android device to date ships with them. It is still suggested that the program check at run time if VBOs are supported, and if not, use a more traditional method to provide data to the GPU [5].

IV. Limitations of Mobile Devices

A. Java and Native Bindings

Many embedded devices, especially ones running the Android operating system, use Java as the primary programming language. OpenGL was designed with C in mind, and is also possible to use with C++. Java bindings for OpenGL have been created, however they use the Java Native Interface, or JNI, an interface which allows C and C++ code to talk to java code [9]. The problem is that the JNI is slow, and can take up to several milliseconds per call. This slow performance is fine for most applications. However, a game that is trying to run at sixty frames per second cannot afford this wait that long. It is possible to get around this by making few calls to OpenGL. Another possible way is to write the entire rendering code in C++ itself, using the JNI. This is significantly harder to be programmed though, and unless the code must be cross platform, it is easier to simply optimize the rendering engine using the Java bindings [10][11].

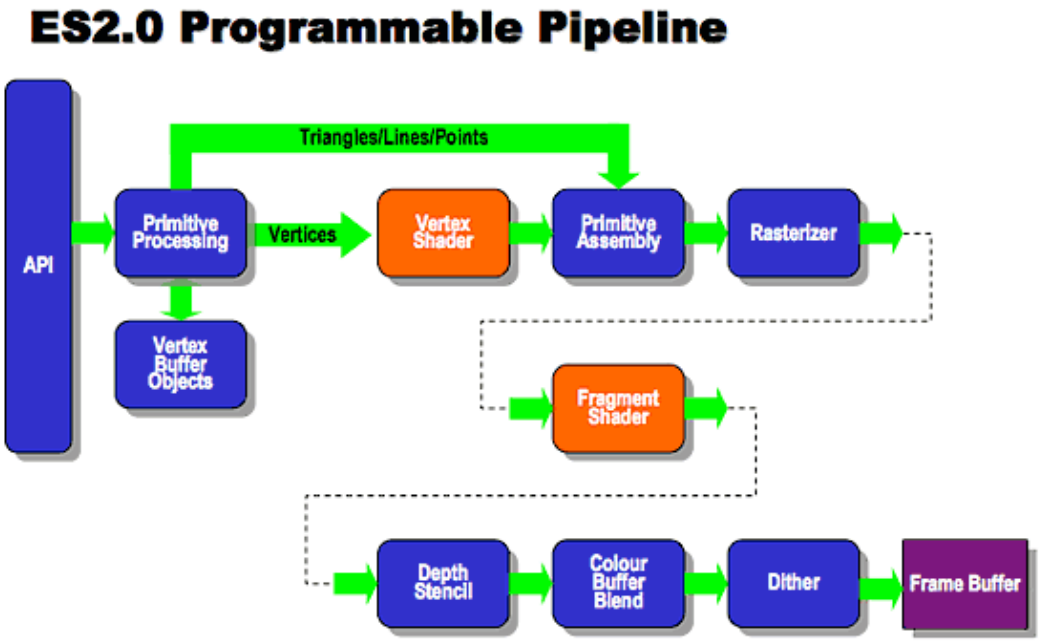

Fig. 3: OpenGLES Programmable Pipeline[2]

B. Texture Size

Storing texture data is a large problem for all but the most simple programs. It is optimal to store as many textures as possible in a single image, and refer to that single image. If this is not possible, then the next solution is to store the data in as few images as possible. Older devices such as the HTC G1 and the could not store images larger than 512x512 pixels. However, newer devices are capable of much larger images. It is currently reasonable to assume that any device made in the past two years is capable of a 2048x2048 resolution image [5].

V. Texture Compression

In desktop graphics, it is not vital that textures get compressed to small sizes. Modern hardware is capable of both storing the larger images, as well as displaying them in real time. However, on mobile devices, there is much less disk space to spare, as well as less processing power to draw all objects. As such, texture compression is needed. There are four main types of texture compression. They are:

1) ETC1
2) PVRTC
3) ATITC
4) S3TC

A. Raw

It is possible to use draw and store raw images onto a phone. Using libpng, it is also even to store them in a more popular format that does save some space. However, the texture must still be decompressed to be placed on the graphics chip. As such, this can, at best, only save storage space. At worst, it will also take up a significant portion of disk space [2].

B. ETC1

ETC1 texture compression is the most common form of texture compression on phones. While Android phones are not required to support it, every major phone does [5]. The Android SDK also provides many tools for making this texture compression fairy simple [12]. The largest drawback to using ETC1 is that it is the slowest of all of the texture compressions. In addition, ETC1 does not support an alpha channel in the texture. As such, any either two textures must be used for alpha, or an alpha channel must be sent separately [2][13].

C. PVRTC/ATITC/S3TC

PVRTC, ATITC, and S3TC are texture compressions that are specific to the creator of the GPUs in the devices they are used. As all iOS devices use PowerVR for their chips, PVRTC is the texture compression best suited for ios [5]. ATITC is for ATI cards, and S3TC is for NVidea cards. These texture compression algorithms do support an alpha channel. To be most efficient with resources, a programmer should use these three textures. Any program that uses this route for texture compression must also support using raw formats in case the hardware cannot use that compression. The program should contain algorithms for each of these texture compression algorithms and then determine at run time which is to be used based on the creator of the GPU [14].

VI. Conclusion

There are two different types of OpenGL ES, a fixed function pipeline, and a programmable one. There will be faster results if the graphics code is made using native code rather than the JNI. However, it is much simpler for the

programmer to use Java code to make the OpenGL calls due to the APIs in the Android SDK. Application performance will be greatly increased by using texture compression. Furthermore, the more powerful texture compressions are specific to the makers of the graphics chips.

REFERENCES

[1] T. Akenine-Moller and J. Strom, "Graphics processing units for handhelds," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 779–789, 2008.

[2] A. Munshi, D. Ginsburg, and D. Shreiner, *The OpenGL ES 2.0 programming guide*. Addison-Wesley Professional, 2009.

[3] J. Neider, T. Davis, and M. Woo, *OpenGL. Programming guide*. Addison-Wesley Reading, MA, 1997.

[4] Google, "Opengl es versions." [Online]. Available: developer.android.com/resources/dashboard/opengl.html, October 2011.

[5] Kishonti, "Glbenchmark." [Online]. Available: glbenchmark.com, October 2011.

[6] S. Hashimi, S. Komatineni, and D. MacLean, *Pro Android 2*. Springer, 2010.

[7] R. Rost and J. Kessenich, *OpenGL shading language*. Addison-Wesley Professional, 2006.

[8] N. Singhal, J. Yoo, H. Choi, and I. Park, "Design and optimization of image processing algorithms on mobile gpu,"

[9] Google, "Android ndk." [Online]. Available: developer.android.com/sdk/ndk/index.html, October 2011.

[10] C. Pruett, "Writing real-time games for android redux." [Online]. Available: developer.android.com/videos/index.html#v=7-62tRHLcHk, October 2011.

[11] I. Valdin, "Graphics optimization for j2me compatible mobile phones," in *Consumer Electronics, 2006. ISCE'06. 2006 IEEE Tenth International Symposium on*, pp. 1–4, IEEE.

[12] Google, "Android development documentation." [Online]. Available: developer.android.com, October 2011.

[13] Google, "Android ndk." [Online]. Available: developer.motorola.com/docstools/library/understanding-texture-compression/, October 2011.

[14] V. Moya, C. Gonzalez, J. Roca, A. Fernandez, and R. Espasa, "Shader performance analysis on a modern gpu architecture," in *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pp. 355–364, IEEE Computer Society, 2005.