COMP3201 – Computer Graphics

Module 3: Realism and Performance

3.6 OpenGL Fundamentals

3.6.1 Graphics Programming with OpenGL

OpenGL is an interface to the graphics hardware. Specifically, OpenGL provides a platform independent manner in which to access the *framebuffer*.

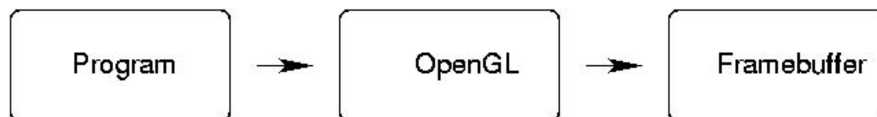


Figure 1: OpenGL provides an interface to the framebuffer

Defn: Framebuffer

Memory on the video card acts as a buffer between the computer processor and the display device. This memory is often called VRAM but should not to be confused with a type of memory also known as VRAM. This buffer is known as the *framebuffer*. The data is read from the framebuffer and converted into an analog signal for the display device by a chip called the RAMDAC (random access memory digital to analogue converter). This analogue signal is then suitable for use by a typical analogue monitor (usually a Cathode Ray Tube).

OpenGL only requires that the video card have a framebuffer. In such a case, OpenGL is implemented in software and the rendering of the 3D scene occurs on the CPU. Modern video cards often have Graphics Processing Units. Thus most modern implementations of OpenGL pass much of the rendering work onto the GPU and hence free up the CPU to spend more time on the actual application. However, due to the proprietary nature of graphics hardware, and a lack of standards, a "cross video card" implementation of OpenGL (e.g. MESA) must be implemented in software and bypass the acceleration offered by the GPUs.

The framebuffer is used by OpenGL to contain several logical buffers. These are the colour, depth, stencil, and accumulation buffers. If double buffering is enabled then there are two logical colour buffers used. If quad buffering is enabled (for stereo viewing) then there are four logical colour buffers used.

OpenGL works on a client-server architecture. The client (the program) issues GL commands to be processed by the server (the OpenGL state machine). There is no constraint for the client and server to be located on the same machine, thus OpenGL is network transparent.

© PM Burrage, GB Ericksson (ACMC)

3.6.2 OpenGL Rendering Pipeline

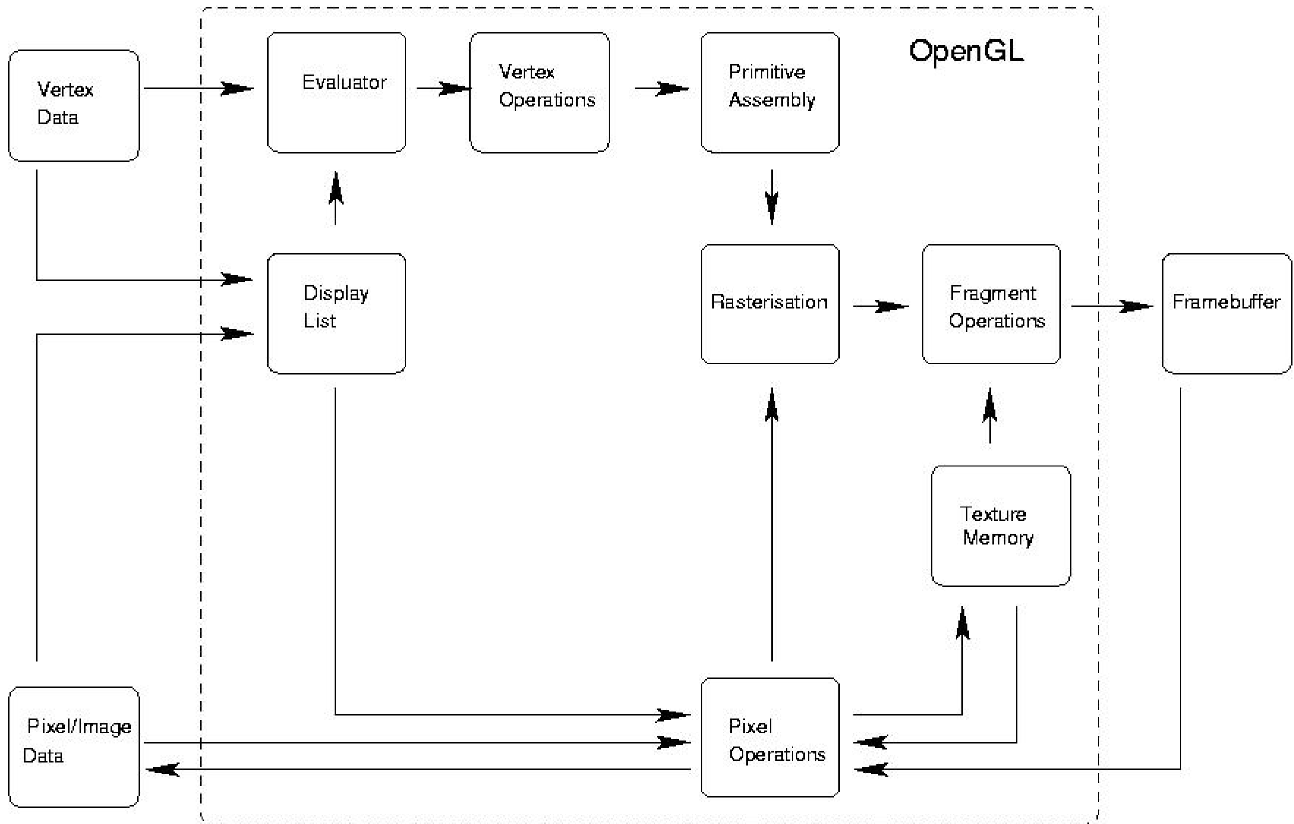


Figure 2 : OpenGL Rendering Pipeline

Figure 2 shows in greater detail the OpenGL section of Figure 1. An implementation of OpenGL does not necessarily process data in the order set out in Figure 2. However, Figure 2 will give you a reliable predictive method for what the OpenGL state machine will do to your data.

The program provides data to OpenGL. There are two types of data that OpenGL can process: pixel (or image) data and vertex data. In this course we have predominantly covered the vertex side of the OpenGL pipeline. Data is sent to the OpenGL state machine using commands like glVertex*(). It should be noted that vertex data provided is in world coordinates. In the following sections each of the steps in the rendering pipeline will be examined.

3.6.2.1 Evaluator and Display List

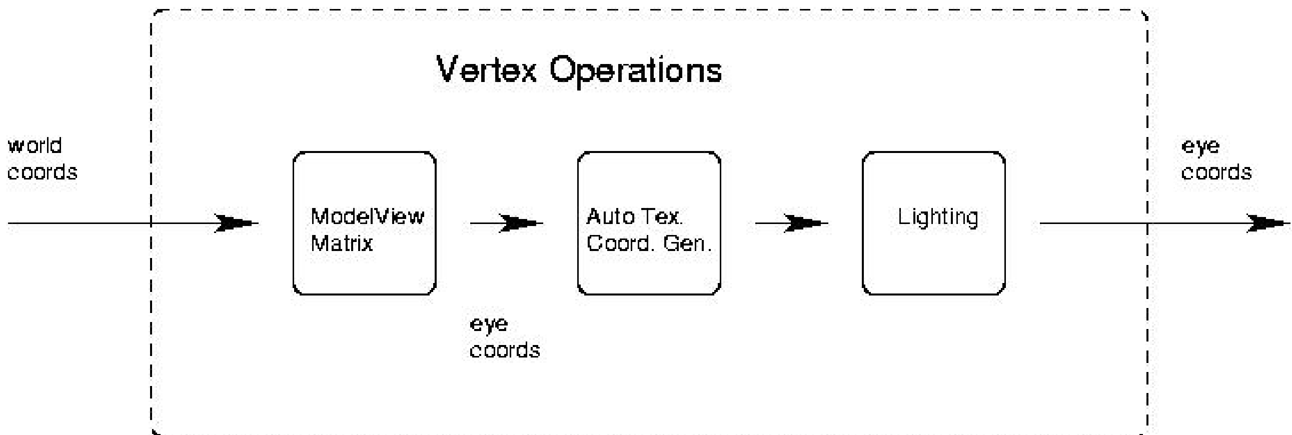
Neither of these stages have been covered in this course. The evaluator stage converts vertex data that was specified in terms of control points for Bezier curves into vertices. The display list stage allows the recording and playing back of OpenGL commands. In some implementations of OpenGL using this facility can result in a performance boost.

3.6.2.2 Pixel Operations and Texture Memory

These stages have only cursorily been covered in this course. In fact all that has been covered so far is the placing of pixel data into texture memory. From Figure 2 it can be seen that it is in fact possible to move data in both directions from the texture memory and the framebuffer. This can be useful for generating textures on the fly, recording frames for movies, amongst other applications. It is also possible to manipulate pixels with operations such as scale, bias, pass through a pixel map, etc.

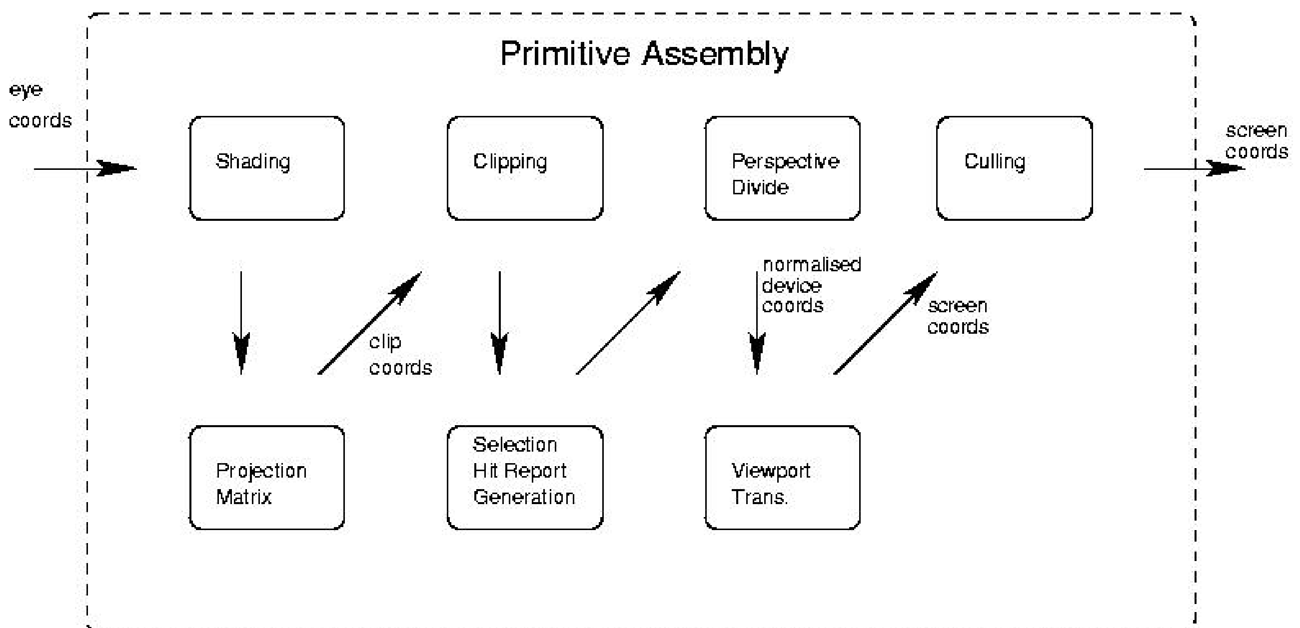
© PM Burrage, GB Ericksson (ACMC)

3.6.2.3 Vertex Operations



The first stage of the vertex operations is to convert the vertex data from world coordinates into eye coordinates. This is done by multiplication with the current modelview matrix. Next (not covered in this course) any automatic texture coordinate generation is done. Finally lighting calculations are performed. Note that the vertex data output from this stage is in eye coordinates.

3.6.2.4 Primitive Assembly



The first stage of the primitive assembly is conditional on whether flat or smooth shading is currently enabled. If the flat shading model has been chosen then the colour of the vertices of the object are set to be the same. Interpolative shading for the smooth shading model is done as part of the rasterisation stage (see Figure 2). The data is then converted from the incoming eye coordinates to clip coordinates by application of the projection matrix.

Clipping is now performed. There are 6 clipping planes specified by boundaries of the canonical viewing volume, and there can also be user defined clipping planes (not covered in this course). If a polygon or line intersects a clipping plane, then only part of the object is clipped and additional

© PM Burrage, GB Ericksson (ACMC)

vertices are created to complete the object along the boundaries of the clipping plane.

If OpenGL is in selection mode (not covered in this course) then a hit report is generated and no further processing occurs.

Homogenisation of the coordinates now occurs. That is, the 4 homogenous coordinates are converted to the usual three spatial coordinates by dividing by the scaling factor w. This stage is also known as Perspective Divide because, except for a perspective style projection, the scaling factor will be unity. This perspective divide stage converts the objects into normalised device coordinates.

The viewport projection now occurs which converts the normalised device coordinates into screen coordinates.

The last step is to do any face culling of polygons. This step is conditional on the culling being enabled. Also the polygon mode is applied at this stage (that is a polygon can be converted to points or lines here).

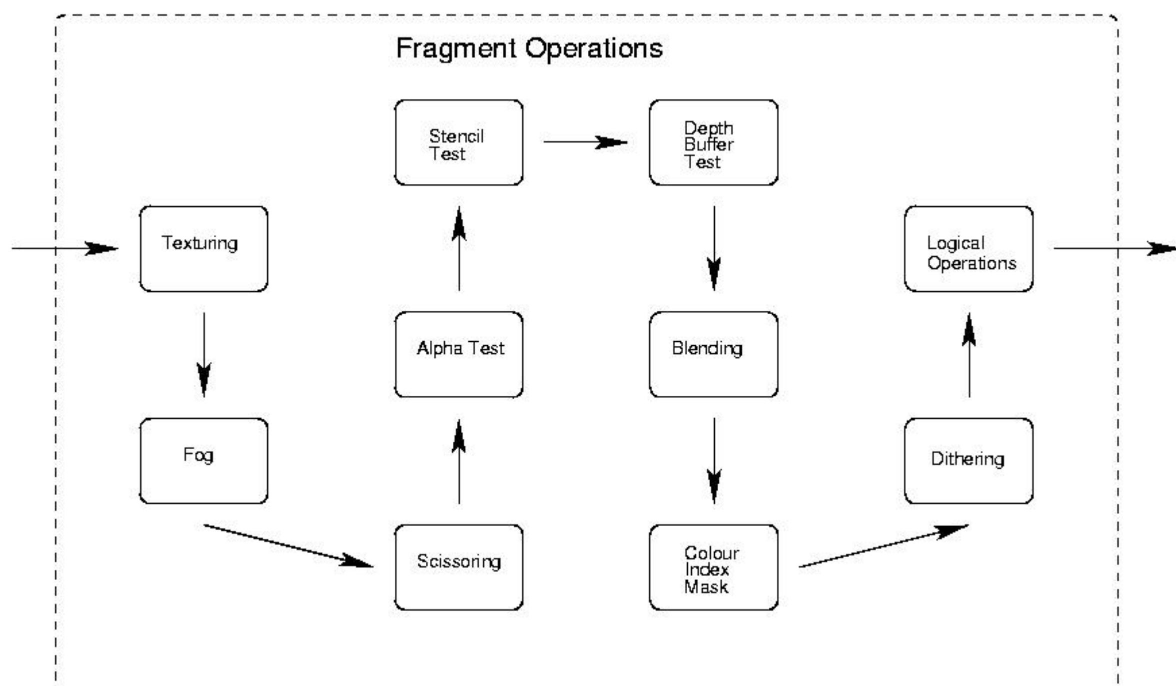
3.6.2.5 Rasterisation

Rasterisation is the process of converting the OpenGL primitives into fragments.

> **Defn: Fragment**
>
> A fragment can be thought of as a proto-pixel. As well as an address in the framebuffer (which corresponds to a pixel on the screen), a fragment also carries colour information, texture coordinates, and a depth value.

The vertex attributes are interpolated to calculate fragment values for the colour, texture coordinates, and depth value. The rasterisation process is affected by any polygon or line stipple (not covered in this course), line width, and point size. If antialiasing is enabled then the fractional coverage by the object for the fragment is calculated. If polygon offset is enabled (not covered in this course) then the depth value is altered by the offset value.

3.6.2.6 Fragment Operations



© PM Burrage, GB Ericksson (ACMC)

Each of the per fragment operations can be enabled or disabled. Some also depend on the existence of an appropriate logical buffer.

The first step in the per fragment operations is the application of texturing. According to the algorithm outlined in Section 3.4 an appropriate texel is chosen from texture memory and combined into the fragment colour. If fog is enabled then the appropriate fog calculation is performed (see Section 3.6.2 for details) and the fragment colour is altered.

Scissoring, the alpha test and the stencil test have not been covered in this course. The scissor test lets a fragment pass if it is within a predefined rectangular portion of the window. If the colour buffer has an alpha component then the alpha test passes or fails a fragment based on its alpha value. If a stencil buffer exists, the stencil test passes or fails a fragment based on a comparison of a reference value with the value for the fragment currently in the stencil buffer.

The depth buffer test compares (using the previously specified depth comparison operation) the depth value of the fragment with that recorded in the depth buffer and passes or fails the fragment. See Section 2.8.3 for more information.

If blending has been enabled and the colour buffer is not in a colour index mode then blending now occurs. Note that blending occurs after many other operations (particularly the depth test) have occurred. This means that many fragments are failed before the blending stage and thus are not blended into the final scene.

Blending is followed by (not covered in this course) either a colour mask or an index mask based on the mode of the colour buffer. If the fragment is being drawn into either the depth buffer or stencil buffer then masking occurs directly following the appropriate tests. The results in this case are written directly into the framebuffer without the dithering or logical operations occurring.

Lastly dithering and logical operations occur. Neither of these operations has been covered in this course. Dithering is a technique to increase the apparent number of colours on memory-limited devices but at the expense of spatial resolution. It is not commonly used now. There are 16 logical operations that can be used to combine the incoming fragment and the colour value in the colour buffer. Typically these operations are used to move data around internally in a window, or between windows, or between a window and processor memory (or vice-versa).

© PM Burrage, GB Ericksson (ACMC)