GPU Shaders

Sandro Spina
Computer Graphics and Simulation Group

Computer Science Department
University of Malta

Graphics Accelerators ...later GPUs

- Graphics accelerators main purpose is that of accelerating the graphics pipeline
- Acceleration has started towards the end of the pipeline, by performing rasterization of a triangle's scanlines. Basically drawing on the screen.
- Successive iterations (improvements) on this hardware worked back up the pipeline.
- The more things done in hardware, the faster the pipeline became. Speed is fundamental in graphics.

Nvidia GeForce256

- The first consumer chip to include hardware vertex processing was Nvidia's GeForce256 graphics card in 1999. (Not too many years ago !!!)
- Nvidia then coined the term Graphics Processing Unit, so that it could differentiate this chip from others that could only accelerate rasterisation.
- This GPU was highly configurable ... Made of a highly complex and configurable fixed function pipeline.
- Over the years hardware vendors improved on this.

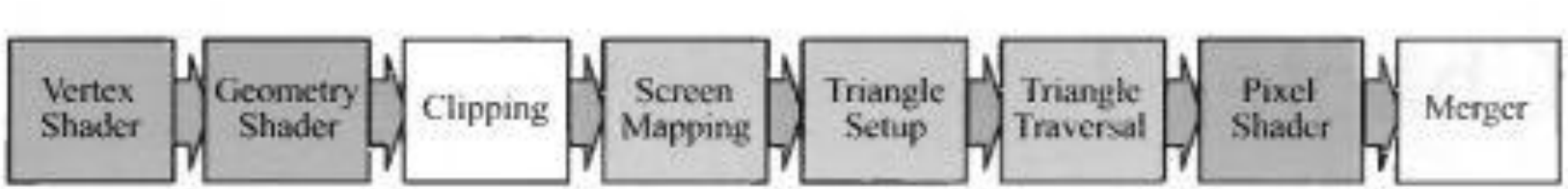
From fixed to programmable ...

- From a highly configurable piece of hardware, the GPU evolved into a highly programmable unit, where developers could implement their own algorithms.
- The programs that can be written for GPU are referred to as shaders.
- We'll be discussing in some detail (and using when we discuss lighting) two types of shaders:
  - Vertex Shader
  - Pixel(Fragment) Shader

Vertex and Pixel Shaders

- A **Vertex Shader** is responsible for the various operations that are performed on the vertices composing our 3D scene.
- A **Pixel Shader** (executed after the vertex shader) processes individual pixels, thus allowing complex shading equations to the evaluated on a per pixel basis.
- DirectX uses HLSL (High Level Shading Language)
- OpenGL uses GLSL (Graphics Library Shading Language)

GPU Pipeline overview (visual)



- Vertex, Geometry and Pixel and completely programmable using shader languages such as GLSL, HLSL, and CG (C for Graphics)
- Geometry shader stage is optional ...
- Clipping and Merger are configurable but not programmable.
- Screen Mapping, Triangle Setup, Triangle Traversal are fixed function.

GPU Pipeline (i)

- Through the pipeline the Vertex Shader is typically used to implement the :
  - Model and View Transform
  - Vertex Shading
  - Projection Transform
- The Pixel Shading perform (obviously) the pixel shading i.e. determining the final colour of each individual pixel.
- The Merger functional stage is in charge of the many buffers (Z, Blend, Stencil, Colour, etc) at the end of the pipeline. Functionality is configurable.

Shader Models

- In a similar way as to how GPUs evolved, shaders (and their languages) have also evolved over time.
- Modern shader stages (Shader Model 4.0, DirectX 10 on Vista) use what is referred to as a common-shader core, which indicates that the various vertex, geometry and pixel shaders share the same programming model.
- This common-shader core is essentially the API that is made available to the programmer.
- NOTE: Throughout this course we shall not discuss in detail the variations between these shader models but we shall be using shaders as required to illustrate for eg. how lighting works.

Shader Programming Models (i)

- Shaders are programmed using a C-like programming language.
- These include GLSL, HLSL and Cg.
- All are compiled to a machine-independent assembly language, the *intermediate language* (IL).
- The assembly is then converted into actual machine code in a separate step by the drivers.
- This is done in order to increase compatibility across different hardware implementations.

Shader Programming Models (ii)

- This assembly language can be seen as defining a virtual machine (similar to the JVM for eg.) which is targeted by the shading language compiler.
- This virtual machine is (similar to) a processor with various types of registers and data sources, programmed with a set of instructions.
- These processors have SIMD capabilities.
- 32-bit single precision floating-point scalars and vectors and the basic data types. Floating point vectors are used to store vertices, normals, matrix rows, colours (rgba), etc. ...

Shader Programming Models (iii)

- A draw call invokes the API to draw a group of primitives, so causing the graphics pipeline to execute. glBegin() ... glEnd()

- Each shader stage has two types of input :
  - Uniform Inputs : with values that remain constant throughout a draw call (can change between calls of course)
  - Varying Inputs : with values are different for each vertex or pixel processed by the shader.

- Different registers are used for these different types of inputs. Uniform types use read-only registers.

- Textures (uniform input) use separate registers.

Virtual Machine Architecture

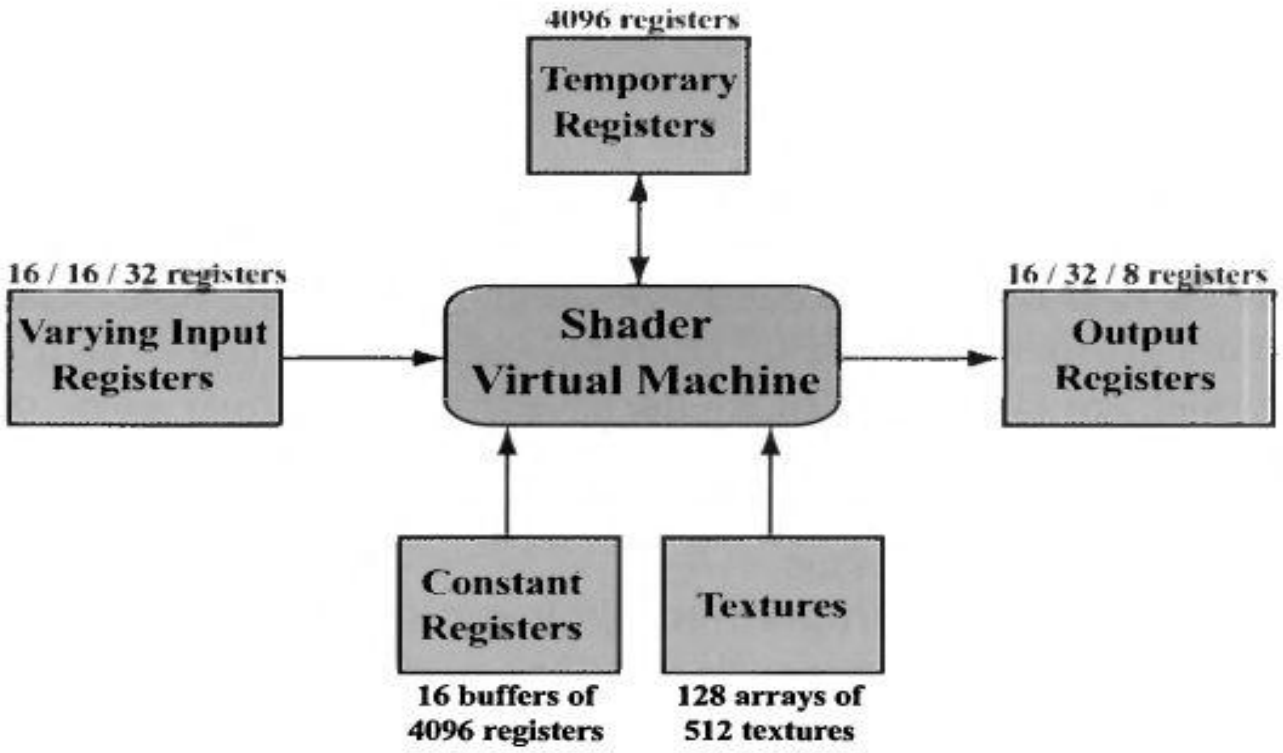
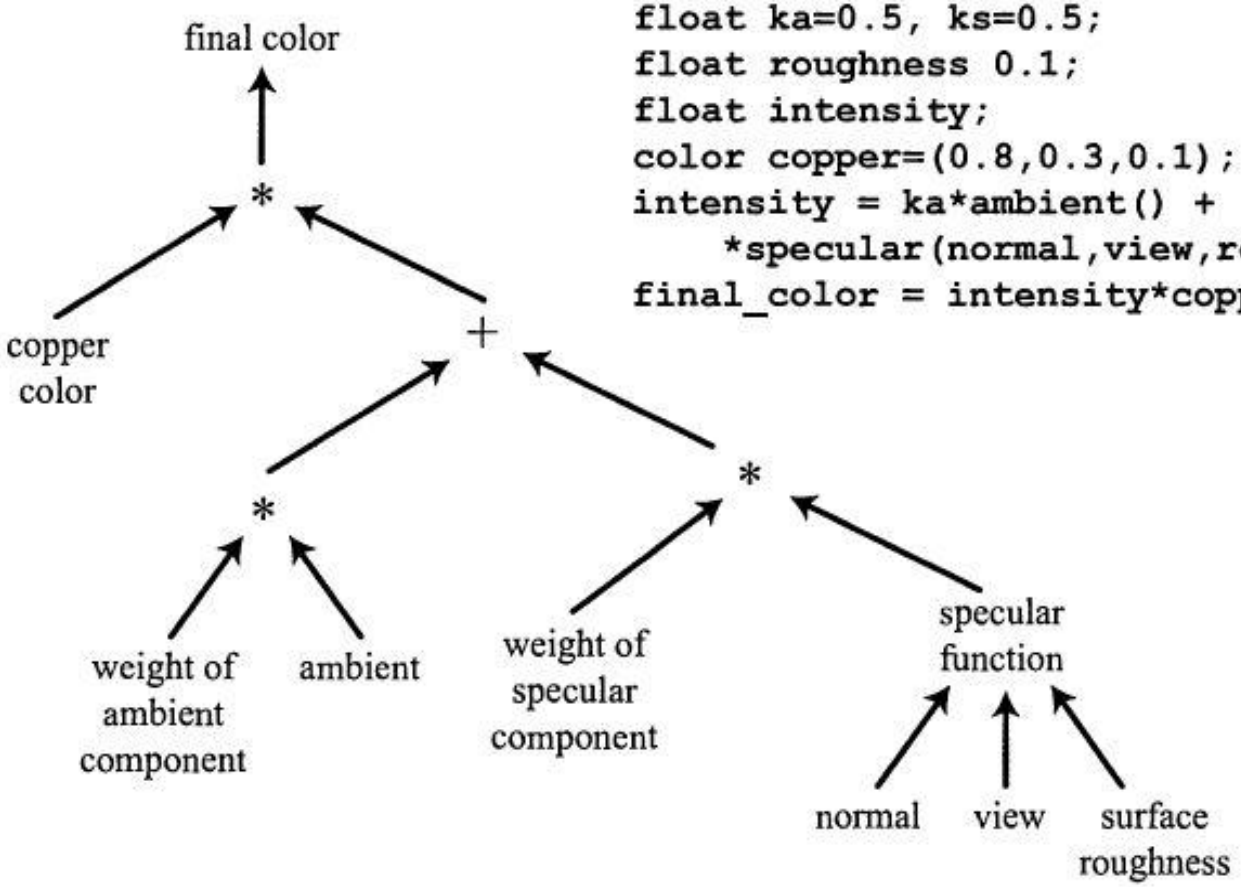


Figure 3.2. Common-shader core virtual machine architecture and register layout, under DirectX 10. The maximum available number is indicated next to each resource. Three numbers separated by slashes refer to the limits for vertex, geometry, and pixel shaders (from left to right).

Shader Programming Model (iv)

- Operations that are common in CG are efficiently executed on GPUs.
- These operations include::
  - Vector Multiplications, Addition, Dot-Product, etc.
  - Reciprocal, Square-roots, sine, cosine, etc.
  - Texture operations.
  - Intrinsic functions such as vector normalisation, reflection, cross product, matrix transpose, determinant, etc
- Shader language also provide traditional flow-control such as that found in general purpose programming languages. Eg. if, case, etc.
- Either Static or Dynamic flow control can be used.

Lighting Calculation example ...




```
float ka=0.5, ks=0.5;
float roughness 0.1;
float intensity;
color copper=(0.8,0.3,0.1);
intensity = ka*ambient() +
    *specular(normal,view,roughness);
final_color = intensity*copper;
```

Figure 3.3. Shade tree for a simple copper shader, and its corresponding shader language program. *(After Cook [194].)*

An example shader for lighting ...

	SM 2.0/2.X	SM 3.0	SM 4.0
Introduced	DX 9.0, 2002	DX 9.0c, 2004	DX 10, 2007
VS Instruction Slots	256	$\geq 512^a$	4096
VS Max. Steps Executed	65536	65536	∞
PS Instruction Slots	$\geq 96^b$	$\geq 512^a$	$\geq 65536^a$
PS Max. Steps Executed	$\geq 96^b$	65536	∞
Temp. Registers	$\geq 12^a$	32	4096
VS Constant Registers	$\geq 256^a$	$\geq 256^a$	14×4096^c
PS Constant Registers	32	224	14×4096^c
Flow Control, Predication	Optional^d	Yes	Yes
VS Textures	None	4^e	128×512^f
PS Textures	16	16	128×512^f
Integer Support	No	No	Yes
VS Input Registers	16	16	16
Interpolator Registers	8^g	10	$16/32^h$
PS Output Registers	4	4	8

The Vertex Shader

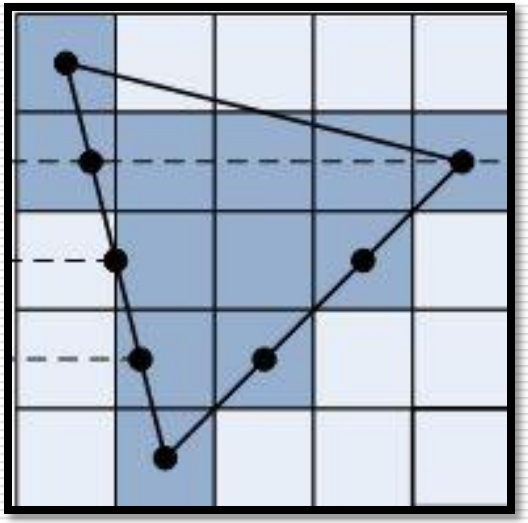
- First shader in the pipeline ...
- The input to this first shader is a triangle mesh represented by a set of vertices and additional information stating which vertices form each triangle.
- This is however the vertex shader (not triangle shader) so the data available is only the vertex. The vertex shader deals exclusively with incoming vertices.
- As a minimum the vertex shader transforms vertices from model space to homogeneous clip space. There performing the model->view->projection transform.

Between Vertex and Pixel(Fragment) Shaders

- Before the output of the vertex shader is passed on to the input of the pixel (fragment) shader a number of (fixed function) operations take place.
- The first operation is clipping, which clips the current primitive against the view volume and in so doing possibly adds or removes vertices.
- After clipping the perspective divide by W occurs, yielding normalised device coordinates, which eventually produce window-space coordinates.
- The rasterisation stage is then responsible for taking these processed vertices (of a triangle primitive for eg.) and turn them into fragments.

Rasterisation

- Check the triangle here ->



- The three vertices defining the primitive determine the fragments (pixels).
- Which has 12 pixels. The fragments fill in the triangle primitive.
- Rasterisation happens for point, line and polygon primitives. In the case of a polygon the algorithms needs to fill in the whole area of the polygon with fragments.

The Pixel(Fragment) Shader

- After the vertex (and possibly the geometry) shader comes the pixel shader. Referred to as fragment shader in OpenGL.

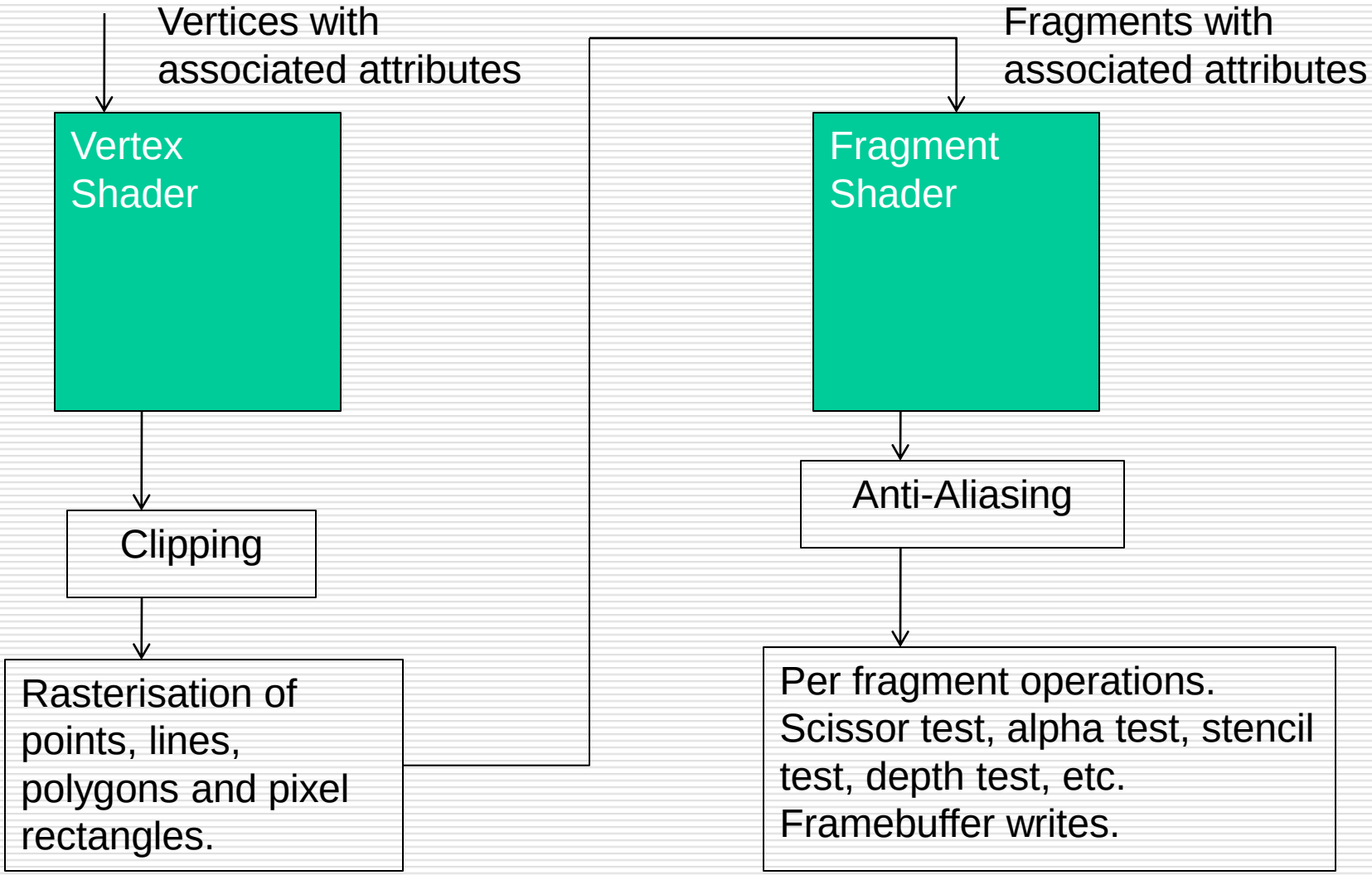
- Prior to entering this stage the primitive is clipped if necessary (against the unit volume) and is set up for rasterisation (as seen in previous slide), i.e. each triangle is traversed and the values at the vertices are interpolated across the triangle's area.

- The pixel shader's role is that of shading all the surfaces of the triangles composing the scene. But it does so one pixel at a time ... The input to the pixel shader is equal to the output of the vertex shader.

The Pixel(Fragment) Shader (ii)

- The input to the pixel shader includes vertex information (such as colour) output from the vertex shader.
- Texture Coordinates per vertex (we still need to check these out) are also available to the fragment shader.
- The pixel shader's limitation is that it can influence only the fragment (pixel) handed it.
- It cannot send its results to neighbouring pixels. This is not such a problem really because each fragment's colour is calculated by interpolating data from the same vertices.

OpenGL Shaders



Managing GLSL Shaders (i)

- Shaders, written on separate files, have to be loaded and then compiled by an OpenGL application. The following steps are required

- Creating a shader object: we first need to create a shader object specifying what type of shader it will be loading. Either GL_VERTEX_SHADER or GL_FRAGMENT_SHADER using the function:
- Gluint myVertexShader = glCreateShader(GL_VERTEX_SHADER);

- 0 is returned if problems are encountered when creating the object.

Managing GLSL Shaders (ii)

- We next specify and load the shader text.
- Note that GLSL accepts shader text rather than precompiled binaries. This is always a good thing as you then compile it with the latest OpenGL drivers for your card.
- The shader text is loaded into the shader object (just created) using the function:
- glShaderSource(myVertexShader, 1, myStringPtr, NULL);
- myStringPtr is a pointer to the shader text loaded from file.

Managing GLSL Shaders (iii)

- Once loaded the shader needs to be compiled.
- Shader text is first parsed making sure there are no errors.
- glCompileShader(myVertexShader);
- glCompileShader(myFragmentShader);
- IMP: you can debug your shader by querying the status of compilation. glGetShaderInfoLog(....) returns a string with any compilation errors.
- GLSL code is compiled when your OpenGL code is executed.

Managing GLSL Shaders (iv)

- Now that we have the compiled shader object we'll need to link it into the same executable (OpenGL + GLSL binaries)
- OpenGL provides Program Objects for this purpose to which compiled shaders can be attached.
- Creating Program Objects involves calling the function:
  - Gluint myProgram = glCreateProgram();
  - glDeleteProgram(myProgram) \\ when done to release memory
- We then attach a shader object to this program object.
  - glAttachShader(myProgram, myVertexShader)
  - glAttachShader(myProgram, myFragmentShader)

Managing GLSL Shaders (v)

- Now that we have the compiled shader object we'll need to link it into the same executable (OpenGL + GLSL binaries)
- OpenGL provides Program Objects for this purpose to which compiler shaders can be attached.
- Creating Program Objects involves calling the function:
  - Gluint myProgram = glCreateProgram();
  - glDeleteProgram(myProgram) \\ when done to release memory
- We then attach a shader object to this program object.
  - glAttachShader(myProgram, myVertexShader)
  - glAttachShader(myProgram, myFragmentShader)

Managing GLSL Shaders (vi)

- Finally before we can use GLSL for rendering we need to link the program object using glLinkProgram(myProgram);
- All the shaders are now within the same executable with the opengl program.
- You can check whether the linking went well by querying the infoLog using:
  - glGetProgramiv(myProgram, GL_LINK_STATUS, &success)
  - glGetProgramInfoLog(myProgram, MAX_INFO_LOG_SIZE, NULL, infoLog)
- Just to make sure everything is fine one can use glValidateProgram(myProgram) and finally issue a call to glUseProgram(myProgram).

Communicating with GLSL (i)

- Suppose we've defined some light sources in our OpenGL code. We clearly need to pass this information over to the shaders.
- Communication is one way ... The shaders cannot talk back to the program. They simply render to some buffer (colour, depth, etc)
- The shader has access to some of the state within OpenGL, therefore by altering this state one can effectively communicate with the shaders.
- This is a bit cumbersome however and not very intuitive really.

Communicating with GLSL (ii)

- GLSL also allows the definition of user defined variables for an OpenGL application to communicate with the shaders.

- For this purpose GLSL provides two types of variable qualifiers :
  - Uniform: A value that remains constant during each shader execution, however it's value is not known at compile time. Not like a const and is initialised out of the shader. A uniform is shared between the vertex and fragment shaders.

  - Attribute: Read-only per-vertex data, available only within vertex shaders. This data comes from the current vertex state or from vertex arrays. Can be either floating-point scalar, vector, or matrix. Not an array or structure.

Variables

- In addition to bool, int, and float-point types (as found in C), GLSL introduces some data types commonly used in CG.
- These include vector types, matrix types, samplers (used to reference image textures).
- All variables and functions must be declared in advance.
- GLSL uses as well some built-in variables (starting with gl_) which allow for interaction with fixed pipeline functionality.

Uniform Variables

- The value of a uniform variable (declared in GLSL code) can only be set for a primitive, therefore modified outside a glBegin/glEnd primitive draw.
- Their value will not change during this call ... hence vertex properties cannot be passed from OpenGL to the shader using these variables.
- Light positions do not change throughout the render of the frame hence can be communicated using these variables.
- glGetUniformLocation(Gluint program, const char *name) is used from the OpenGL code to get the memory address where the variable is stored.
- This memory location is then used in glUniform**n**f(GLint location, Glfloat v0 ... Glfloat v**n**) to assign the variable.

Attribute Variables

- If we are required to set attributes(properties) **per vertex** then attribute variables must be used.
- These variables can only be read in a vertex shader, mostly because they would normally contain data related to the vertex attributes/properties not required by a fragment shader.
- Attribute variables are set in a similar way to uniform variables by getting their memory locations than updating them from the OpenGL program.
- GLint glGetAttribLocation(GLuint program,char *name) with parameters *program* referring to the handle to the program and *name* referring to the name of the attribute variable

Some Fragment Shaders

- We shall be looking at lighting shaders when we cover illumination ... For now we'll have a look at some simple fragment shaders in order to get an idea of how these work.
- GrayScale Fragment Shader
- Sepia Tone Fragment Shader
- Inversion Fragment Shader
- B&W Negative Fragment Shader

Image Processing with Fragment Shaders

- Image processing is one application of fragment shaders which does not depend on vertex shaders.
- Essentially we can apply convolution kernels after drawing a scene to post-process the scene in a number of ways.
- The scene is first draw using a 'normal' fragment shader. The pixels in the frame buffer are then used as a texture, which is then used for image processing.
- We shall check a number of implementation including:
  - Blur Fragment Shader
  - Sharpen Fragment Shader
  - Edge Detection Fragment Shader
  - Erosion and Dilation Fragment Shaders

HLSL Shaders

- HLSL shaders work in a similar way to GLSL ones ...
- Using XNA we can however avoid all the hassle of loading, compiling and linking a shader ... And we can use the content pipeline.
- HLSL Semantics : HLSL has semantics that are used to link the input and output of values from the graphics pipeline. A semantic is a keyword used to bind your variables to the inputs and outputs of a shader, and tells HLSL how the variable is used.
- For eg:
- float4 vertexPos : POSITION
- float4 color : COLOR0;

Shaders, Techniques and Passes

- Unlike GLSL, in HLSL shaders do not have reserved words for their entry points (such as main()).
- In HLSL you would create the function, name it whatever you wanted, and tell the HLSL compiler its name. This is done in the shaders technique.
- A technique is basically an effect that is specified by a set of vertex, geometry and pixel shaders. You can have as many techniques as you want in one file.

- Technique SimpleEffect {

```
    Pass Pass0 {
        //shaders
        VertexShader = compile vs_3_0 VertexShaderMain();
        PixelShader = compile ps_3_0 PixelShaderMain();
    }
}
```