Introduction to Modern OpenGL® in C++

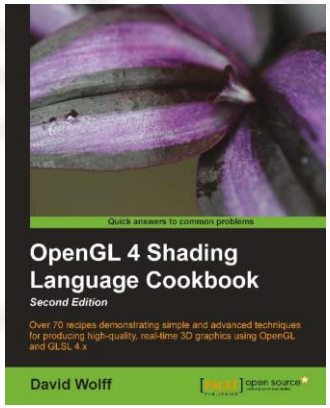
Dr. Muhammad Mobeen Movania
Assistant Professor
Department of Computer Science
DHA Suffa University

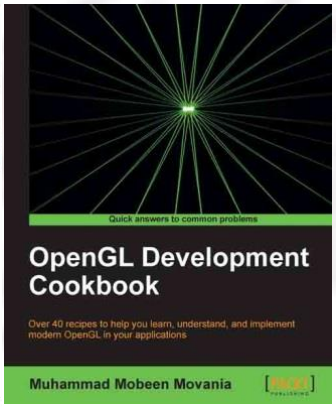
Outline

- About the speaker
- About the tutorial
- Basics
  - What is OpenGL®
  - The OpenGL® Ecosystem and History
  - Introduction to Shaders
  - Compute vs Shader Mode/Available Libraries and Frameworks
  - OpenGL ES and WebGL
  - Coordinate Spaces
- Tutorials…
  - Getting Started with Modern OpenGL
  - Setting up the development environment
  - Setup glew, freeglut, glm, DevIL on Visual studio 2012

About the Speaker

- Received PhD in Advanced Computer Graphics and Visualization from Nanyang Technological University Singapore in 2012
- Post doctoral researcher at Institute for Infocomm Research (I^2R), a division of A-Star, Singapore (~1.5 years)
- Have published in several international conference including a poster at SIGGRAPH 2013
- Wrote a book and have contributed/reviewed recent OpenGL books and courses including
  - WebGL Insights (Expected Aug, 2015) (Contributor/reviewer)
  - OpenGL 4 Shading Language Cookbook (Second Edition) (2014) (Reviewer)
  - Building Android Games with OpenGL ES online course (2014) (Reviewer)
  - OpenGL Development Cookbook (2013) (Sole Author)
  - OpenGL Insights (2012) (Contributor/reviewer)





About the Tutorial

- Outcomes
  - Introductory knowledge of how to get started with Modern OpenGL® in C/C++
- Assumptions
  - You know basic vector math and matrices
  - You know basics of how to program in C/C++
  - Ideally able to create simple console applications in an IDE preferably MS Visual Studio 2012 or higher

Basics

What is OpenGL®

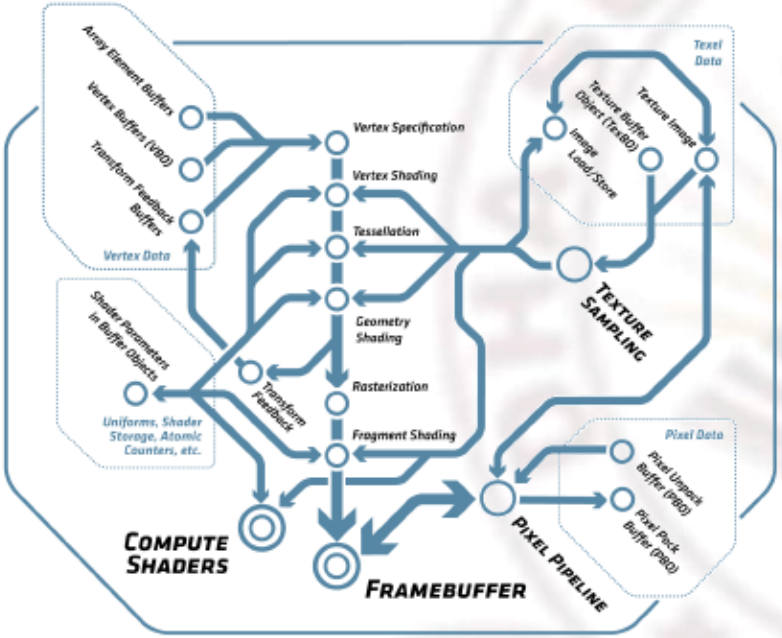
- An open graphics API specification
- It is not open source, rather the specifications are open
- There is no OpenGL® SDK unlike DirectX SDK
  - The OpenGL® functions are provided by the graphics driver given by your vendor
- To get the latest OpenGL® functions, ensure that you have the latest graphics drivers from your hardware vendor

The OpenGL® Ecosystem

OpenGL History

- OpenGL v 1.0 (1992)
  - Fixed function pipeline
- OpenGL v 2.0 (2004)
  - Introduced shading language (GLSL) which allowed programming of the vertex transformation and fragment shading
- OpenGL v 3.0 (2008)
  - Introduced deprecation model
- OpenGL v 3.1 (2009)
  - Removed most deprecated features (no fixed function pipeline support i.e. no transformation matrices support, no material or lighting support etc.)
- OpenGL v 3.2 (2009)
  - Introduced core and compatibility profile context
  - Core profile has no deprecated functionality
  - Compatibility profile retains deprecated functionality
- Current OpenGL v 4.4/4.5 (2014)
- For more information: http://www.opengl.org/wiki/History_of_OpenGL

OpenGL® Timeline



Bringing state-of-the-art functionality to cross-platform graphics

OpenGL 4.5
OpenGL 4.4
OpenGL 4.3
OpenGL 4.2
OpenGL 4.1
OpenGL 3.3/4.0
OpenGL 3.2
OpenGL 3.1
OpenGL 2.0
OpenGL 2.1
OpenGL 3.0

2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014
DirectX 9.0c		DirectX 10.0		DirectX 10.1		DirectX 11		DirectX 11.1		DirectX 11.2

Introduction to shaders

- Special micro programs that are run on the GPU
- Allow programmability of different stages of the modern GPU graphics pipeline
- Legacy OpenGL (less than v 3.0) only support two shader types: vertex and fragment shader
- Modern OpenGL (v3.0 and above) support vertex, geometry and fragment shaders
- Current OpenGL (v 4.0 and above) supports vertex, tessellation control/evaluation shaders, geometry and fragment shaders

A brief chronology

- Conventionally shaders were used for graphics work only with dedicated programmable processors (vertex and fragment processors)
- More and more people used shaders for non-graphics work (GPGPU) for e.g. image processing, AI calculation, dynamics etc. from 2004-2006.
- Due to this trend, NVIDIA came up with a GPGPU framework based on C called CUDA in 2007. They introduced a stream processor model which is a general purpose processor
- Khronos group then created an open specification OpenCL in 2008

Compute vs Shader mode

- Modern GPUs can work in two modes
  - compute mode or
  - shader mode
- Compute mode
  - the GPU stream processors work as general purpose processors which makes them suitable for GPGPU
- Shader mode
  - the GPU stream processors work as specialized processors for vertex, tessellation, geometry, fragment processing which makes them suitable for graphics processing
- At any point, the GPU can only be in one mode either compute or shader mode

Compute and Shader frameworks and libraries

- Graphics API
  - OpenGL/DirectX (for desktop development)
  - OpenGL ES (for mobile/tablets)
  - WebGL (for web browser on desktop/mobile/tablets)
- Shader languages
  - GLSL (OpenGL/OpenGL ES/WebGL)
  - HLSL (DirectX)
  - Cg (wrapper around GLSL for OpenGL applications and HLSL for DirectX applications)
  - Brook (a kernel based GPGPU emulation using shaders)
- Compute
  - CUDA (NVIDIA only)
  - OpenCL (general)
  - WebCL (browser based compute)
  - Direct Compute Shader (special shader type in DirectX 10 and above)
  - OpenGL Compute Shader (special shader type in OpenGL v 4.3 and above)

OpenGL ES

- A slim version of desktop OpenGL for mobile and tablet platforms
- OpenGL ES v 1.x and above
  - No immediate mode rendering (`glBegin/glEnd`) only retained mode rendering available using vertex arrays
  - Supports only 2D textures
  - Supports only triangles (`GL_TRIANGLES`)
- OpenGL ES v 2.0
  - Added shader support similar to desktop OpenGL 2.0
  - Removed fixed function pipeline completely (i.e. no transformation matrices support, no material or lighting support etc.)
- Further details: http://www.khronos.org/opengles/

WebGL

- A subset of OpenGL ES that runs in a browser window without requiring any plugin
- Supported on all major web browsers
- Direct access to the OpenGL driver through the html5 canvas element
- Calling conventions similar to desktop OpenGL

Legacy OpenGL vs Modern OpenGL

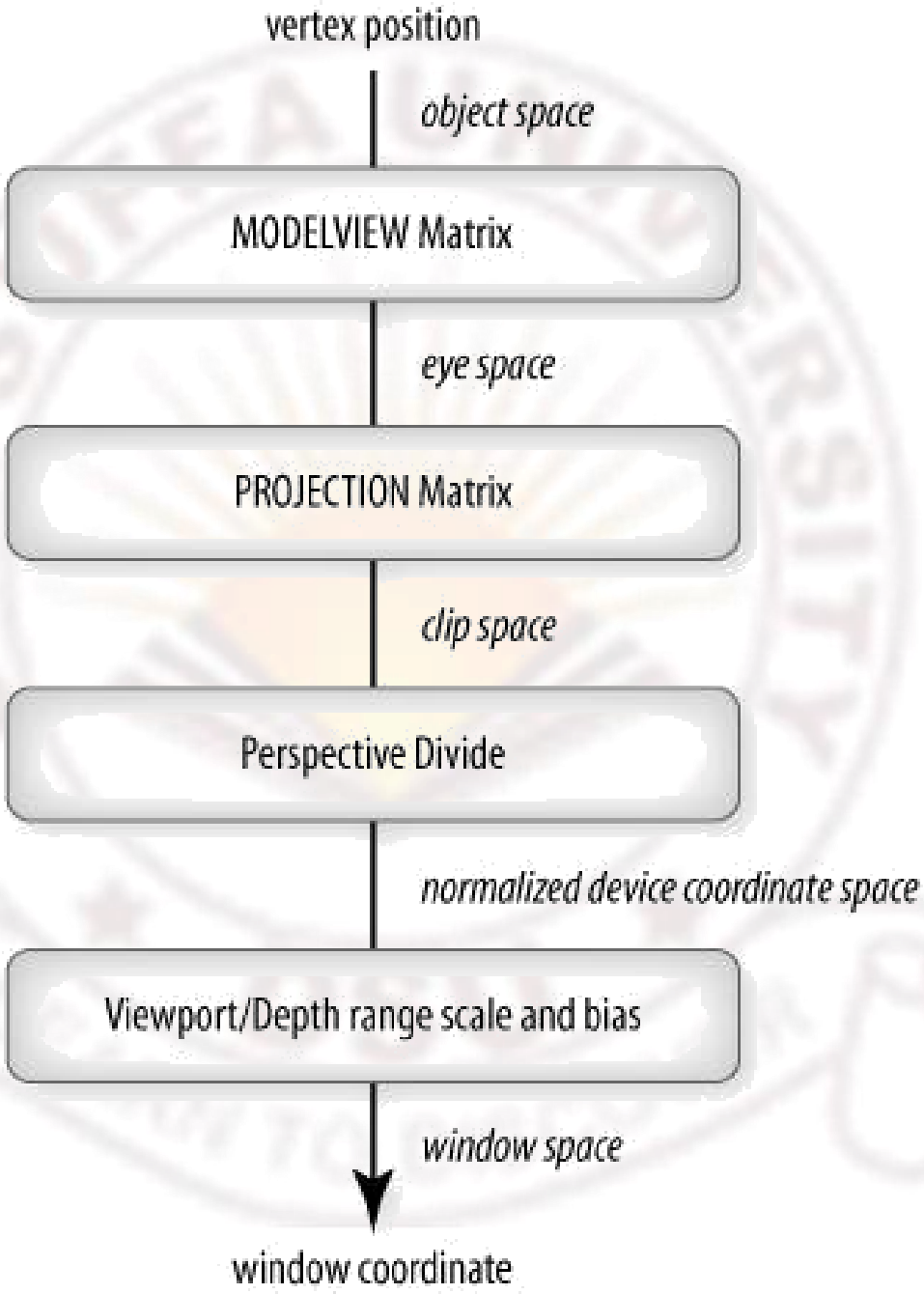
Legacy OpenGL	Modern OpenGL
Fixed function pipeline support i.e. lighting, materials etc. provided	Fixed function pipeline support in compatibility profile only and is removed from the core profile
Matrix handling and matrix stack functions provided i.e. gl{Push/Pop}Matrix, gl{Translate,Rotate,Scale}f functions	No matrices support in the core profile. User must either implement their own matrix library or use libs like glm
Immediate mode rendering provided i.e. glBegin(…)/glEnd()	Retained mode rendering provided by using either vertex arrays or vertex buffer objects (VBO)

OpenGL Extensions Mechanism

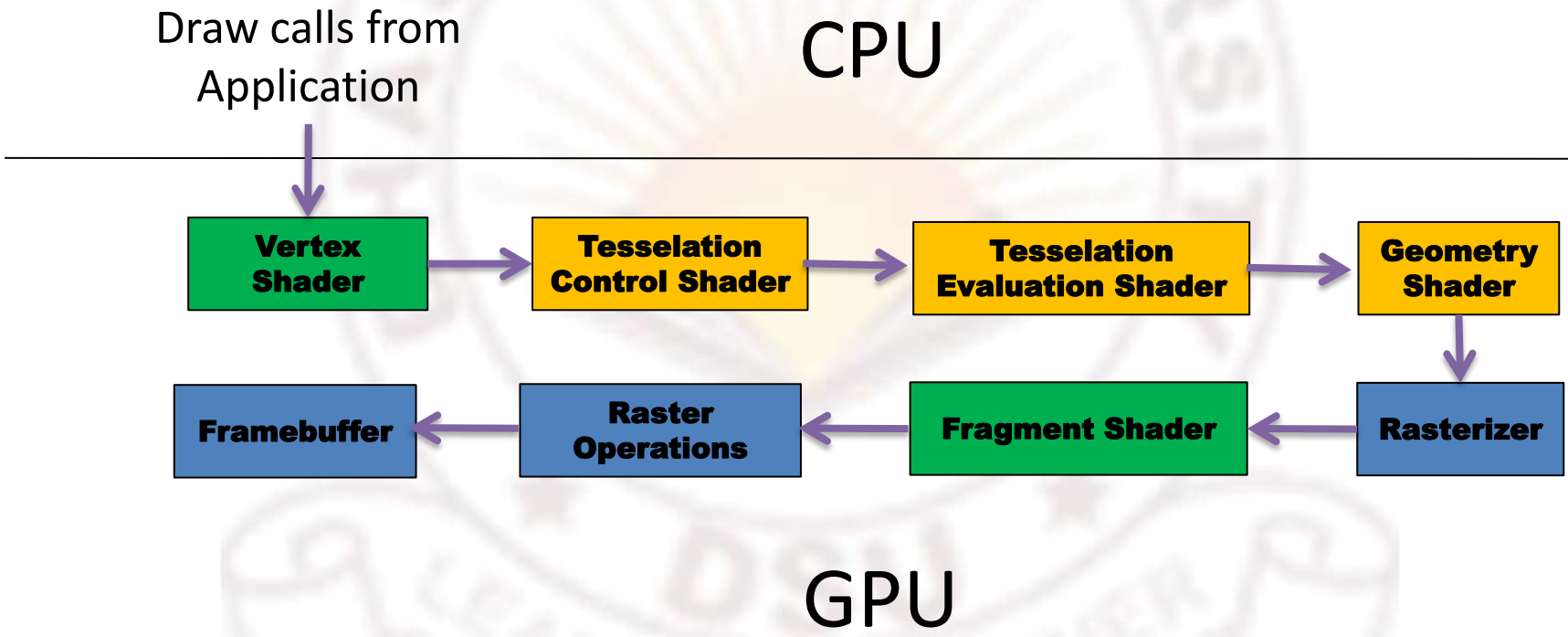
- OpenGL specifications allows GPU vendors to expose functionality which is supported by a hardware but it is not specified in the OpenGL specifications by means of extensions.
- The extensions can contain new constant fields, new functions or new formats etc. on the discretion of the hardware vendor. Such functions/fields/formats etc. are marked with EXT suffix and conventionally have the vendors initials as prefix for e.g. `GL_NV_texture_barrier`
- After a EXT remains for a while, it is discussed in the next OpenGL ARB (Architectural Review Board) meeting. If the extension has enough substance, it is promoted to ARB extension. These are marked with ARB suffix
- After a couple of revisions, the ARB extension is then merged into core OpenGL functionality for e.g. `GL_EXT_framebuffer_object` was promoted to `GL_ARB_framebuffer_object` and then to core in OpenGL v 3.0.
- After promotion to core the EXT and ARB suffix are removed from the function/field/format.
- **As a rule of thumb, programmers should always call the core functions first. If the functionality is not core yet then the ARB variant should be used and if that is not available then use the EXT functions.**
- **EXT functions should be used as the last resort since it makes the code dependent on a specific hardware that supports that extension only**

Coordinate Spaces

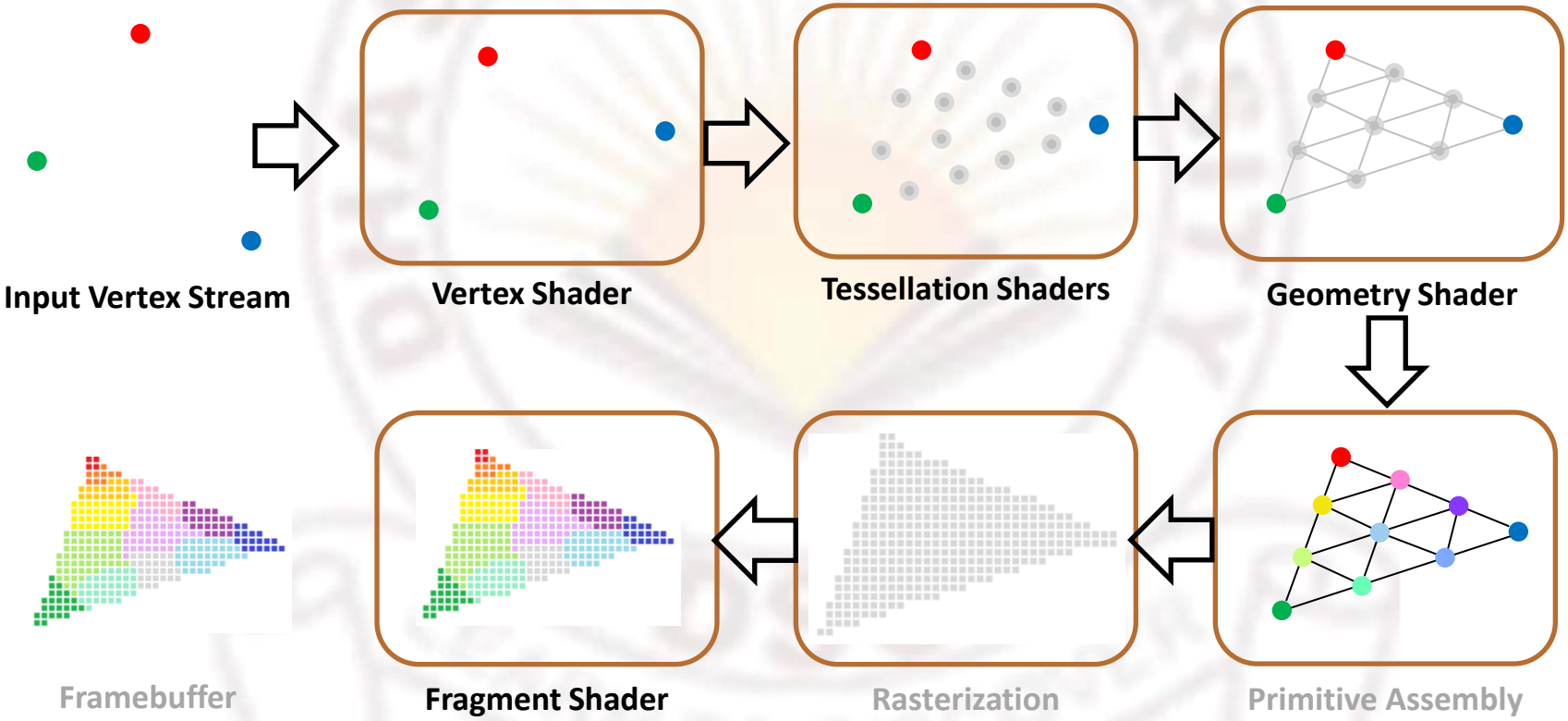
- Objects in graphics are represented using a collection of vertices that are connected to one another to create polygons
- The vertices are specified in the coordinates called **model coordinates** (also called **local coordinates** or **object space coordinates**)
- The vertices are transformed by a 4x4 **Model** matrix to bring them into **world space**.
- The world space vertex coordinates are multiplied by a 4x4 **View** matrix to get **eye space** or **view space coordinates**
- In OpenGL, the **Model** and **View** transformation matrices are combined into a single **ModelView** matrix
- The eye space coordinates are then projected to clip space using a 4x4 **Projection** matrix. The clip space coordinates are from [-1,-1,0] to [1,1,1]
- The clip space coordinates are then normalized to get **normalized device coordinates** which are from [0,0,0] to [1,1,1]
- The normalized device coordinate undergo viewport transformation to obtain **screen space coordinates also called window space coordinates**


vertex position
object space
MODELVIEW Matrix
eye space
PROJECTION Matrix
clip space
Perspective Divide
normalized device coordinate space
Viewport/Depth range scale and bias
window space
window coordinate

The Modern Programmable Graphics Pipeline



Programmable Stages of the Modern Graphics Pipeline



Libraries/Tools/SDKs

Tools and Libraries

- There is no official SDK for OpenGL
- The hardware vendor provides latest drivers which contain the required OpenGL dlls
- Both NVIDIA and ATI/AMD provide OpenGL SDKs for developers which are freely downloadable from the vendor websites
  - NVIDIA: https://developer.nvidia.com/opengl
  - ATI/AMD:
    - OpenGL SDK: http://developer.amd.com/tools-and-SDKs/graphics-development/amd-radeon-SDK/
    - OpenGL ES SDK: http://developer.amd.com/tools-and-SDKs/graphics-development/amd-opengl-es-SDK/
- The version of OpenGL dll that ships with the Windows operating system is quite old therefore developers are advised to update their drivers from their hardware vendor's website

Tools and Libraries

- Development libraries
  - glew (allows easier loading of OpenGL extensions) URL: http://glew.sourceforge.net/
  - freeglut (platform independent windowing toolkit) URL: http://freeglut.sourceforge.net/
  - glm (vector/matrix math library) URL: http://glm.g-truc.net/
- High level shader editors
  - ShaderDesigner (http://www.opengl.org/SDK/tools/ShaderDesigner/)
  - RenderMonkey (http://developer.amd.com/resources/archive/archived-tools/gpu-tools-archive/rendermonkey-toolsuite/)
- Rapid application development suite
  - MeVis Lab (an OpenInventor based IDE with built-in support of vtk/itk as well as GLSL shaders) http://www.mevislab.de/download/
- Debuggers
  - gDebugger
    - Standalone version: http://www.gremedy.com/
    - Visual studio addin: http://developer.amd.com/tools-and-SDKs/heterogeneous-computing/archived-tools/amd-gdebugger/ now replaced by AMD CodeXL
  - AMD CodeXL: http://developer.amd.com/tools-and-SDKs/heterogeneous-computing/codexl/
  - glslDevil: http://www.vis.uni-stuttgart.de/glsldevil/
  - WebGL Inspector: http://benvanik.github.io/WebGL-Inspector/
- More info: https://www.opengl.org/wiki/Debugging_Tools

Tools for Looking at Performance Bottlenecks

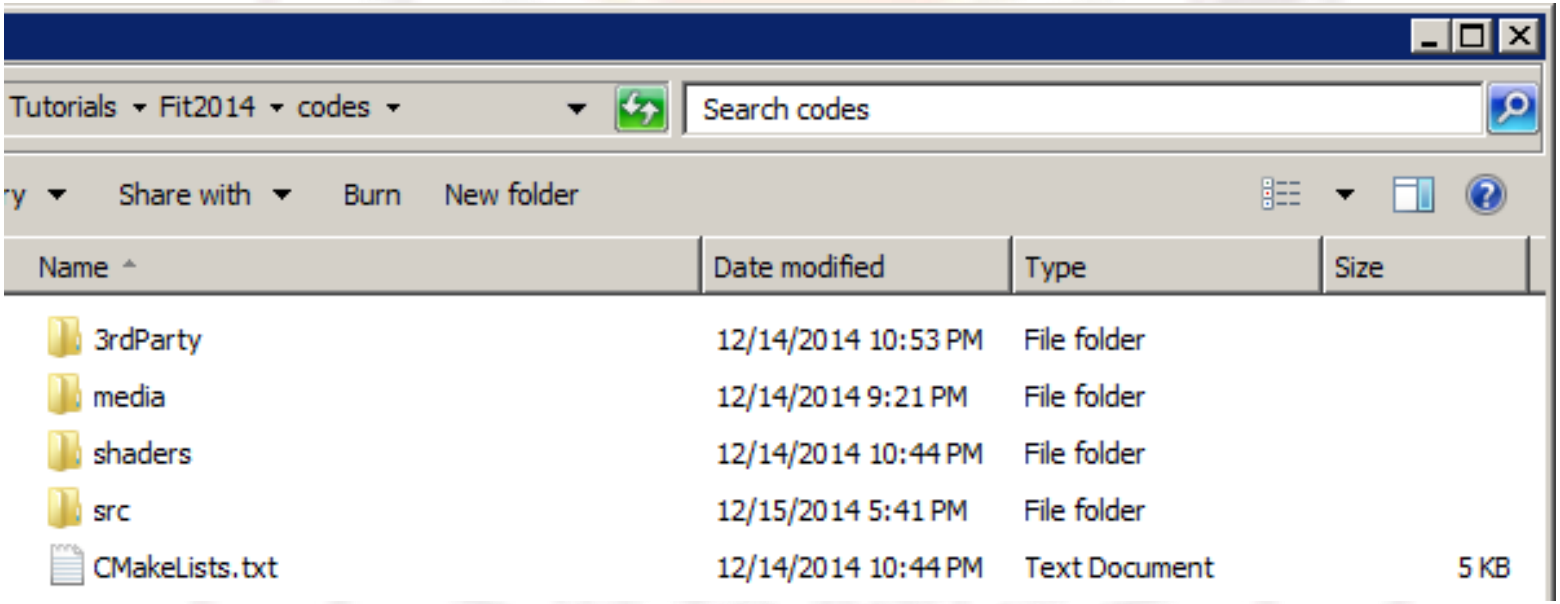
- **AMD**
  - AMD PerfStudio 2 (http://developer.amd.com/tools-and-SDKs/graphics-development/gpu-perfstudio-2/)
  - AMD GPUperfAPI (http://developer.amd.com/tools-and-SDKs/graphics-development/gpuperfapi/)
  - AMD GPU Shader Analyzer (http://developer.amd.com/tools-and-SDKs/graphics-development/gpu-shaderanalyzer/)
- **Intel**
  - Intel GPA (https://software.intel.com/en-us/vcsource/tools/intel-gpa)
- **NVIDIA**
  - NVIDIA PerfKit(https://developer.nvidia.com/nvidia-perfkit)
  - NVIDIA NSIGHT(https://developer.nvidia.com/nvidia-nsight-visual-studio-edition)

Tutorials

Basic Setup

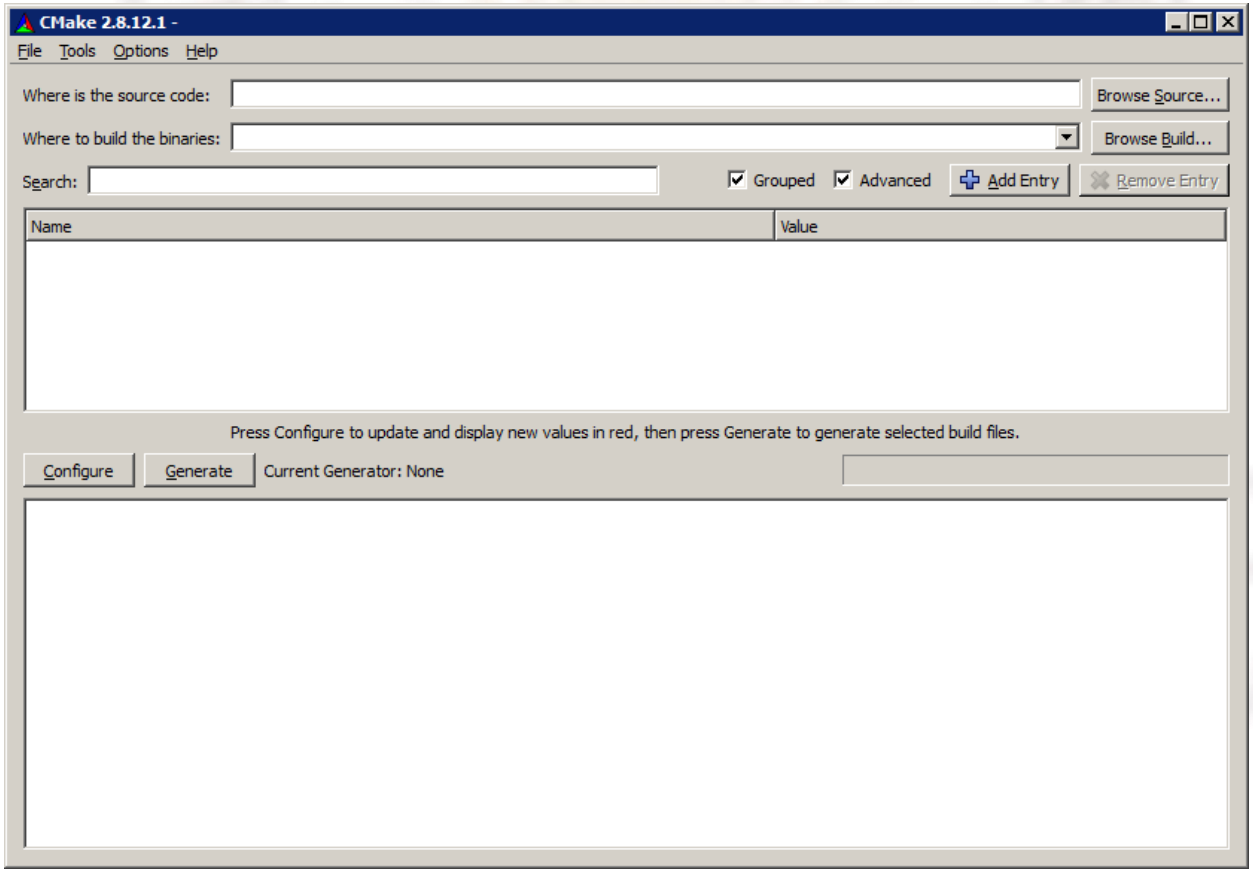
- Download cmake: http://www.cmake.org/download/
- Go to tutorial webpage http://cgv.dsu.edu.pk/tutorials/FIT2014 and download the codes.7z compressed file
- Direct Link: http://cgv.dsu.edu.pk/tutorials/FIT2014/codes.7z
- In case, our server is down, get it from github https://github.com/mmmovania/FIT2014_OpenGL_Tutorial
- The codes.7z file contains all required libraries for these tutorials
- Extract codes.7z file to a suitable location

- If you have done all the steps as directed, you directory structure should be like this

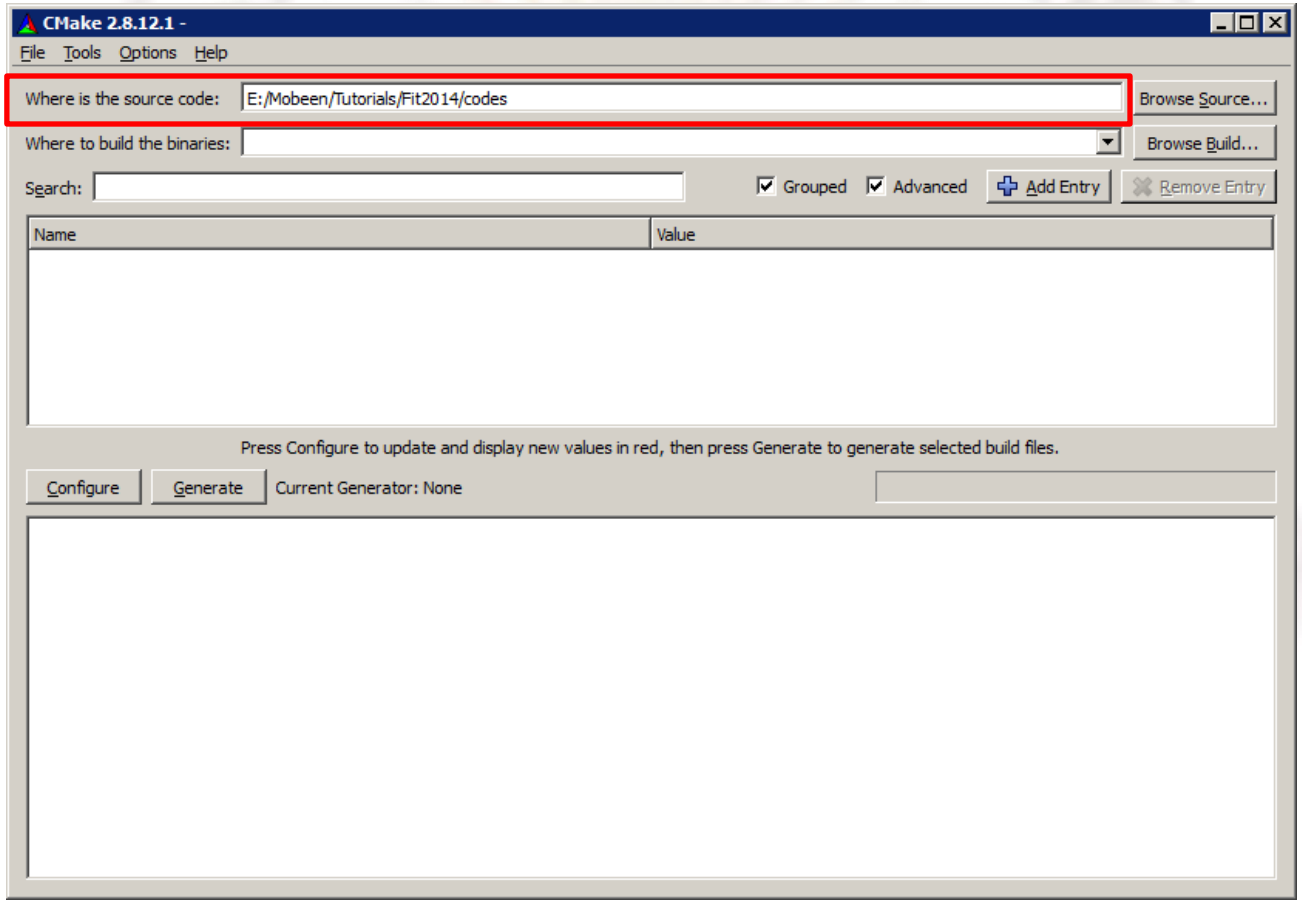


Using CMAKE to create Visual Studio Projects

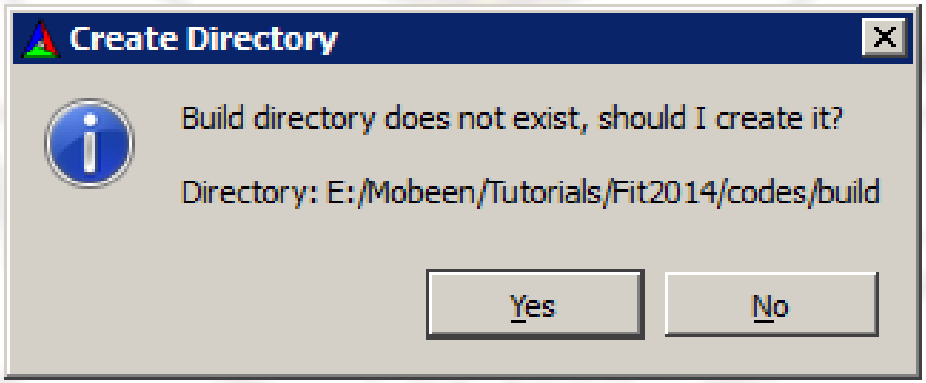
- CMAKE is a versatile cross platform make tool
- Run Cmake gui which opens a dialog box as shown below



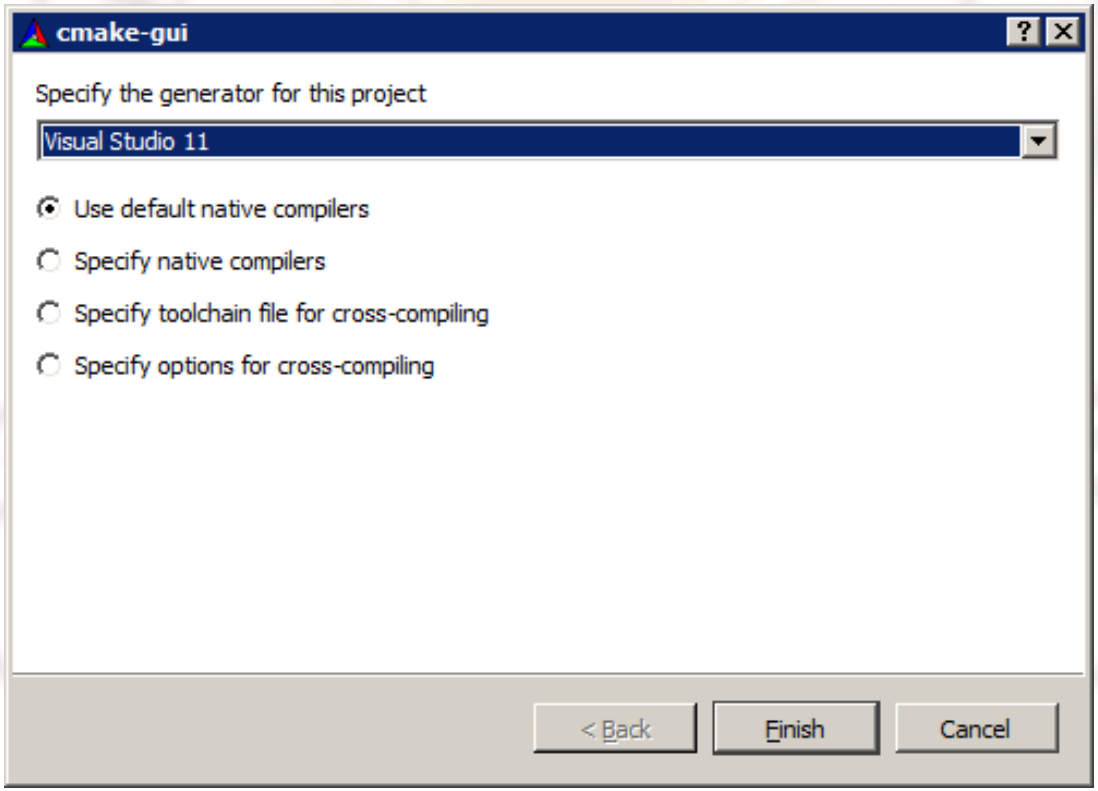
- In Cmake gui application, enter the path where you extracted codes.7z file in “where is the source code” field. **This is the folder containing CMakeLists.txt file**



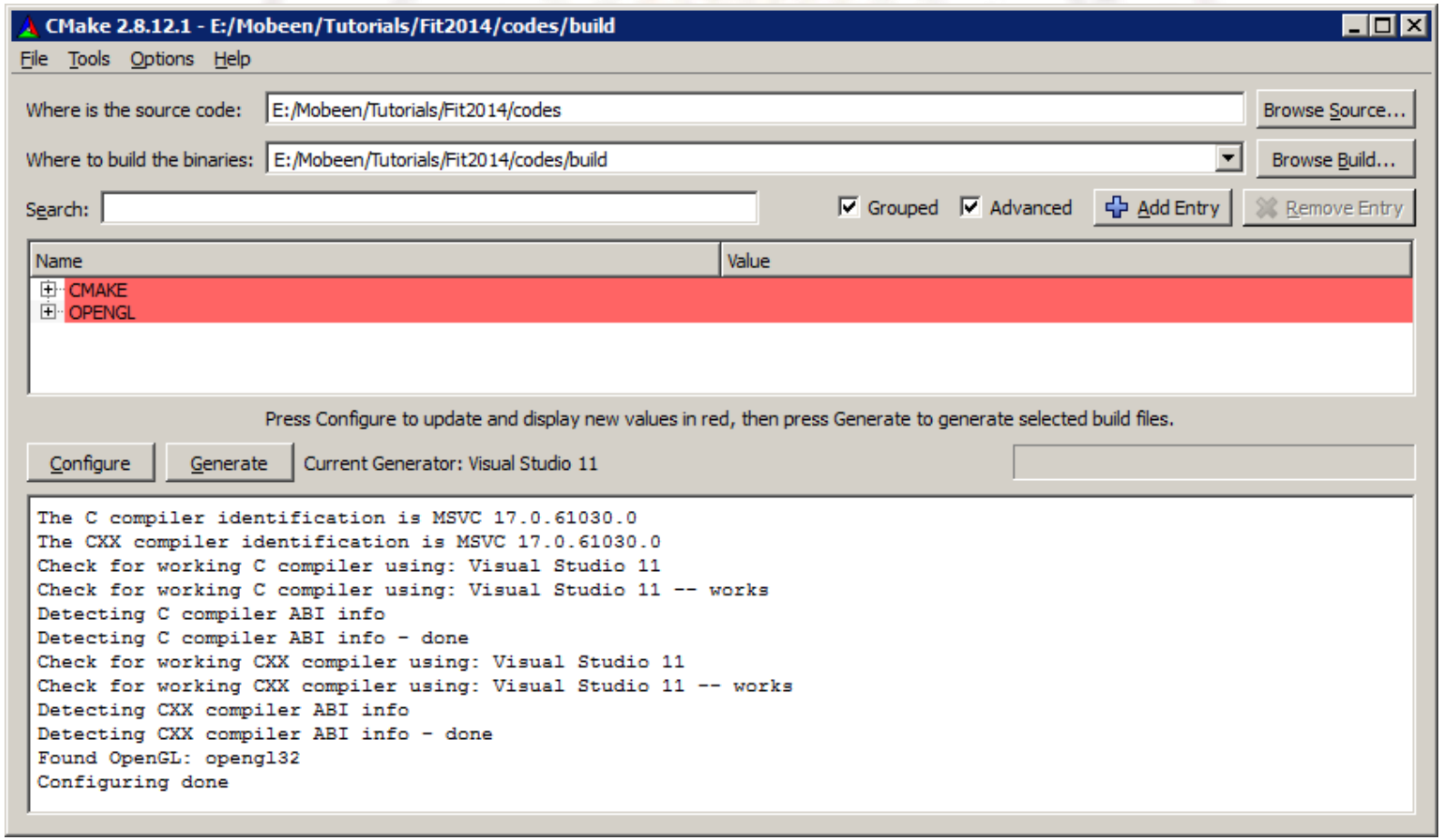
- In Cmake gui application, enter the same path as was given in the “where is the source code” field but add build sub-directory so that the project files are stored in the build sub-directory.
- Next press Configure button which will ask you to create the build sub-folder. Press Yes.



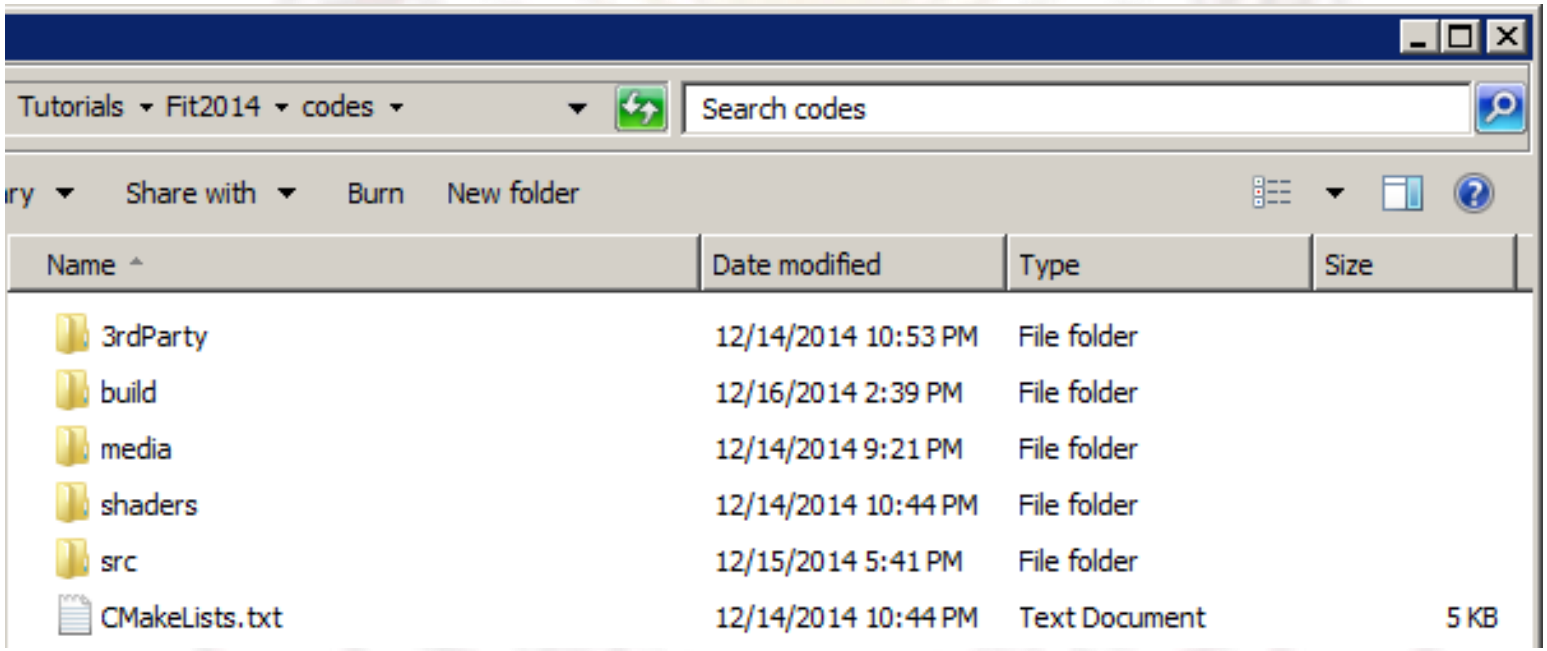
- Next a dialog box will be given for your to select a suitable generator. This is the preferred compile for which the projects will be generated. Accept the default settings and press finish.



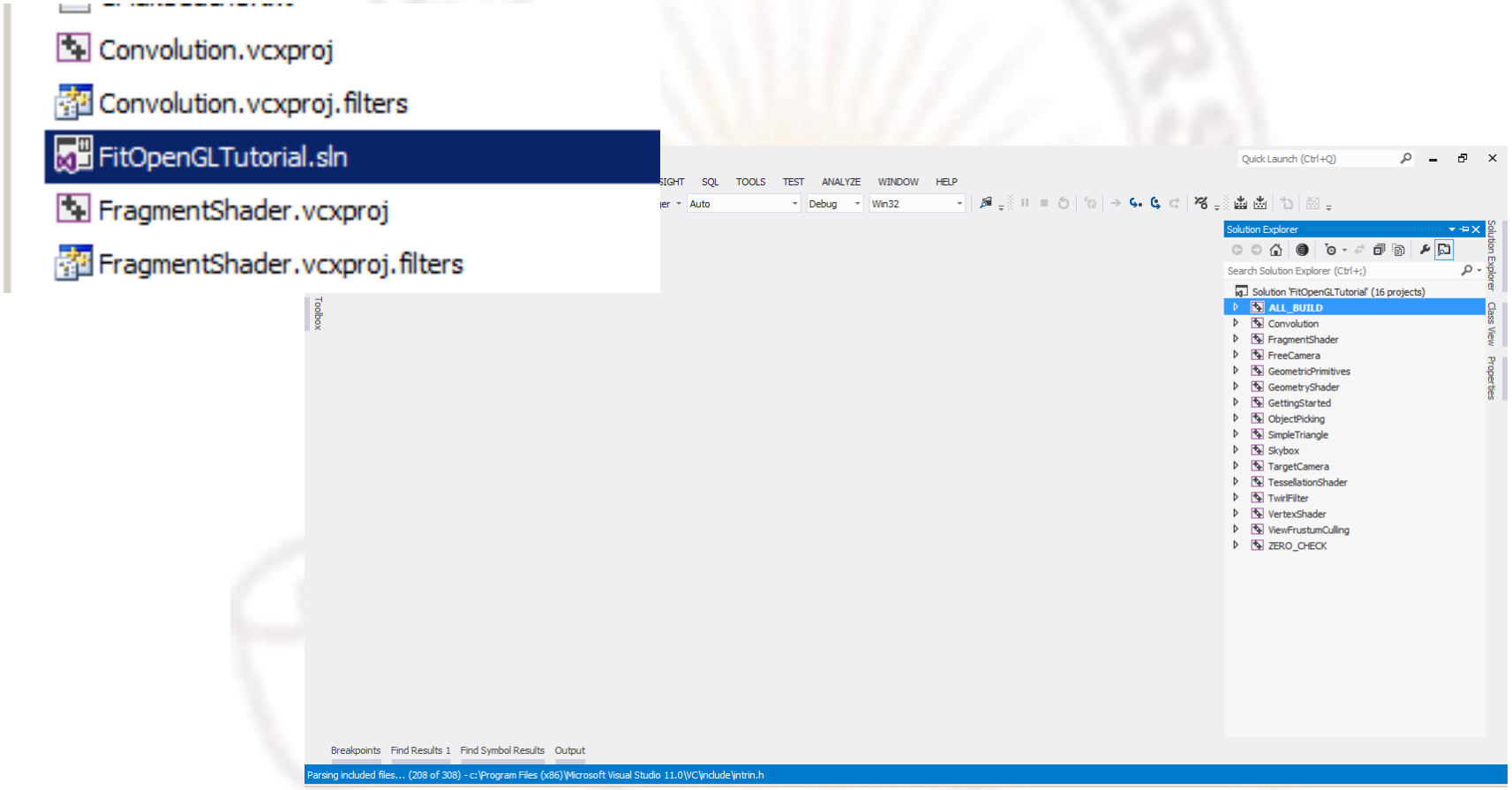
- Cmake will then print some information about the current compiler as shown below. This might take a few seconds to finish.



- Next press the Generate button which will generate the Visual Studio project files. Close Cmake gui and now your directory structure will be as follows



- Now go into the build directory and open FitOpenGLTutorial.sln Visual Studio solution.



Typical OpenGL® Application

- Create a window using the operating system specific API and specify the framebuffer format, depth and double buffering support
  - For convenience, we will use freeglut library which abstracts all of this from us in a platform independent way
- Create a valid OpenGL context for the desired OpenGL version
  - Here also freeglut provides functions which handle this in a platform independent manner.
  - Additional functions are provided for obtaining a valid OpenGL v 4.4 and above core/compatibility profile context
- Attach paint event callback function to call the OpenGL draw function
  - Freeglut provides glutDisplayFunc for this
- Invoke the main loop function
  - Freeglut provides glutMainLoop function

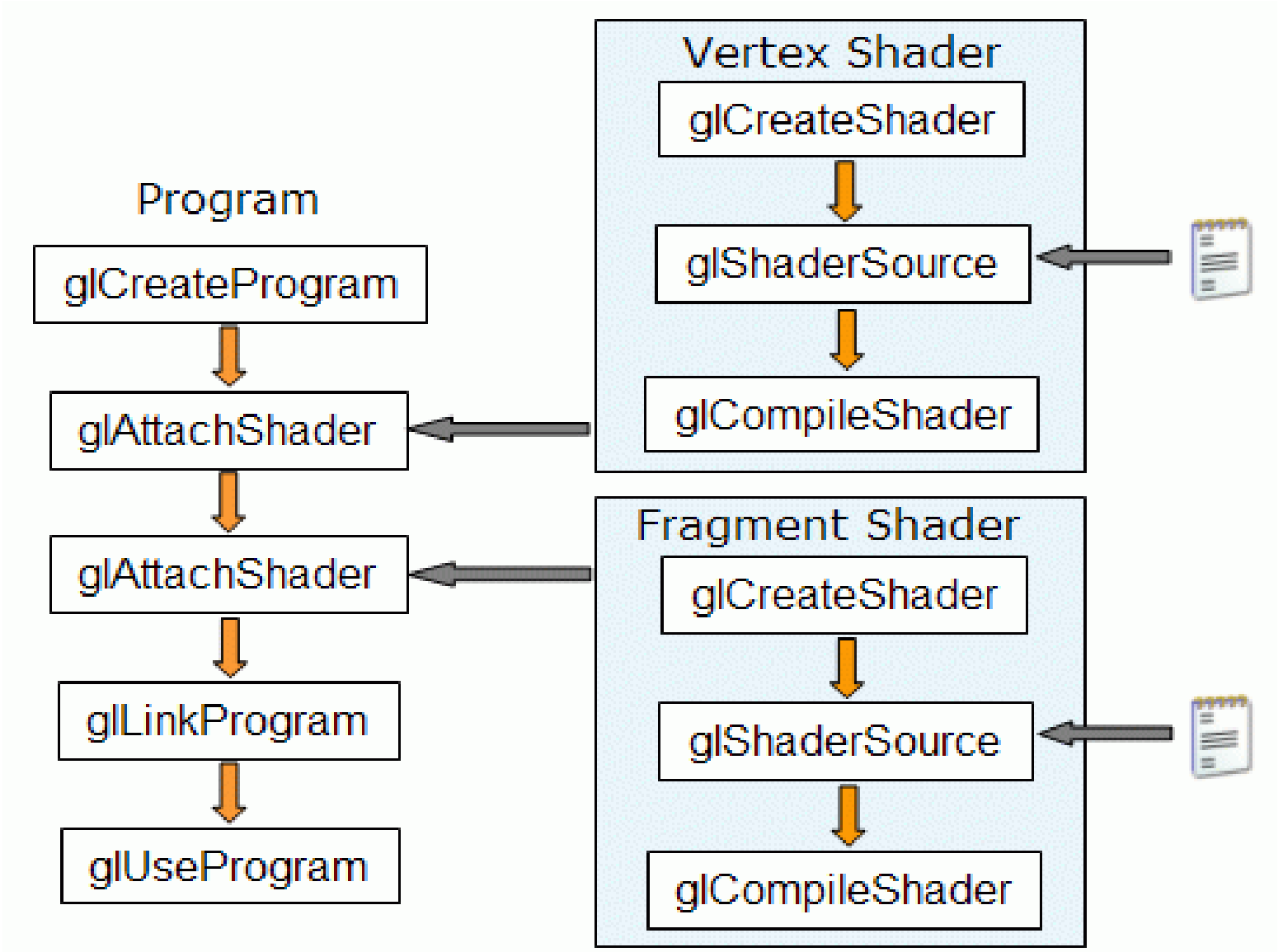
Getting Started with Modern OpenGL

- By default, freeglut provides the compatibility profile context for the version of OpenGL that is supported completely by the given hardware
- You can find out about the supported OpenGL version by using OpenGL Extensions Viewer: http://www.realtech-vr.com/glview/)
- For enabling support for modern OpenGL (v 4.4 and above) we can specify the OpenGL version by calling `glutInitContextVersion` and `glutInitContextFlags` before `glutCreateWindow`.
- To enable OpenGL v 4.4 forward compatible core profile context use

```
int main(int argc, char** argv) {
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA );
   glutInitWindowSize(SCREEN_WIDTH, SCREEN_HEIGHT);
   glutInitContextVersion (4, 4);
   glutInitContextFlags (GLUT_CORE_PROFILE);
   glutInitContextProfile(GLUT_FORWARD_COMPATIBLE);
   glutCreateWindow("Getting started with modern OpenGL");
   //rest of the code as before
}
```

Getting started with shaders

- In a typical Modern OpenGL application, the following sequence of calls are executed to create a shader object
  - `glCreateShader` (generates a new shader object)
  - `glShaderSource` (sources shader text to shader compiler)
  - `glCompileShader` (compiles a shader object)
  - `glGetShaderInfoLog` (dumps shader compile logs)
- After creation of shader object, a program object is created by using
  - `glCreateProgram` (generates a new progrcam object)
  - `glAttachShader` (attaches one or more shaders to a program object)
  - `glLinkProgram` (links a program object)
  - `glGetProgramInfoLog` (dumps program link logs)
- After a program object is linked successfully, all of the shader objects attached to the program can be deleted safely
- The program object can then be used with the rendering code by calling `glUseProgram(program_name)`


Vertex Shader
glCreateShader
glShaderSource
glCompileShader
Program
glCreateProgram
glAttachShader
glAttachShader
glLinkProgram
glUseProgram
Fragment Shader
glCreateShader
glShaderSource
glCompileShader

Wrapping the shader interface into a simple class GLSLShader

- Since the steps involved in shader compiling and linking are repetitive, we wrap the whole thing into a convenient class `GLSLShader` as follows

```
class GLSLShader {
   public:
      GLSLShader(void);
      ~GLSLShader(void);
      void LoadFromString(GLenum whichShader, const string& source);
      void LoadFromFile(GLenum whichShader, const string& filename);
      void CreateAndLinkProgram();
      void Use();
      void UnUse();
      void AddAttribute(const string& attribute);
      void AddUniform(const string& uniform);
      GLuint getAttribute(const string& attribute);
      GLuint getUniform(const string& uniform);
      void DeleteShaderProgram();
   private:
      enum ShaderType {VERTEX_SHADER, FRAGMENT_SHADER, TESSELATION_CONTROL,
                       TESSELATION_EVAL, GEOMETRY_SHADER, MAX_SHADERS};
      Gluint _program;
      int _totalShaders;
      GLuint _shaders[MAX_SHADERS];   //0->vertexshader,
                                      //1->fragmentshader,
                                      //2->tessellation control,
                                      //3->tessellation evaluation,
                                      //4->geometryshader
      map<string,GLuint> _attributeList;
      map<string,GLuint> _uniformLocationList;
};
```

Tutorial0

- Project: GettingStarted
- Outcome: How to create a Win32 Window that supports OpenGL 4.4
- Learning:
  - freeglut initialization
  - glew initialization
  - attaching callbacks with freeglut
  - clearing backbuffer and depth buffer

Freeglut Initialization

```
int main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DEPTH|GLUT_DOUBLE|GLUT_RGBA);
    glutInitContextVersion (4, 4);
    glutInitContextFlags (GLUT_CORE_PROFILE );
    glutInitContextProfile(GLUT_FORWARD_COMPATIBLE);
    glutInitWindowSize(WIDTH, HEIGHT);
    glutCreateWindow("Getting started with OpenGL 4.4");
```

Glew Initialization

- Enables easier loading of OpenGL® extensions
- glewInit to initialize glew but this should be called only after the window has been created
  - glewInit requires a valid OpenGL® context

```
glewExperimental = GL_TRUE;
GLenum err = glewInit();
if (GLEW_OK != err){
   cerr<<"Error: "<<glewGetErrorString(err)<<endl;
} else {
   if (GLEW_VERSION_4_4) {
      cout<<"Driver supports OpenGL 4.4\nDetails:"<<endl;
   } else {
      cout<<"Driver does not support OpenGL 4.4"<<endl;
      exit(EXIT_FAILURE);
   }
}
```

Attaching Callbacks

- Called by freeglut in response to event generated by Operating System
- glutCloseFunc(OnShutdown);
  - Called when the application is closed
  - Do all resource deallocation here
- glutDisplayFunc(OnRender);
  - Called whenever the window is repainted
  - Do all drawing code here
- glutReshapeFunc(OnResize);
  - Called when the window is resized
  - Do viewport and projection setting here

Clearing buffers

- Previously glClear was used like this
  - `glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);`
  - This function required setup of two OpenGL® states `glClearColor` and `glClearDepth`
  - This function is still usable but not recommended

- Modern OpenGL® provides separate functions thus there is no state anymore
  - `const GLfloat CLEAR_COLOR[4]={1.0f,0.0f,0.f,1.f}; //red color`
  - `const GLfloat CLEAR_DEPTH = 1.0f;`
  - `glClearBufferfv( GL_DEPTH, 0, &CLEAR_DEPTH );`
  - `glClearBufferfv( GL_COLOR, 0, CLEAR_COLOR );`

Tutorial1

- Project: SimpleTriangle
- Outcome: How to draw a coloured triangle in OpenGL 4.4
- Learning:
  - Using GLSLShader class
  - Using buffer objects to store triangle geometry
  - Shaders setup (vertex and fragment)
  - Setup OpenGL viewport, modelview and orthographic projection matrices
  - Drawing triangle

Using GLSLShader class

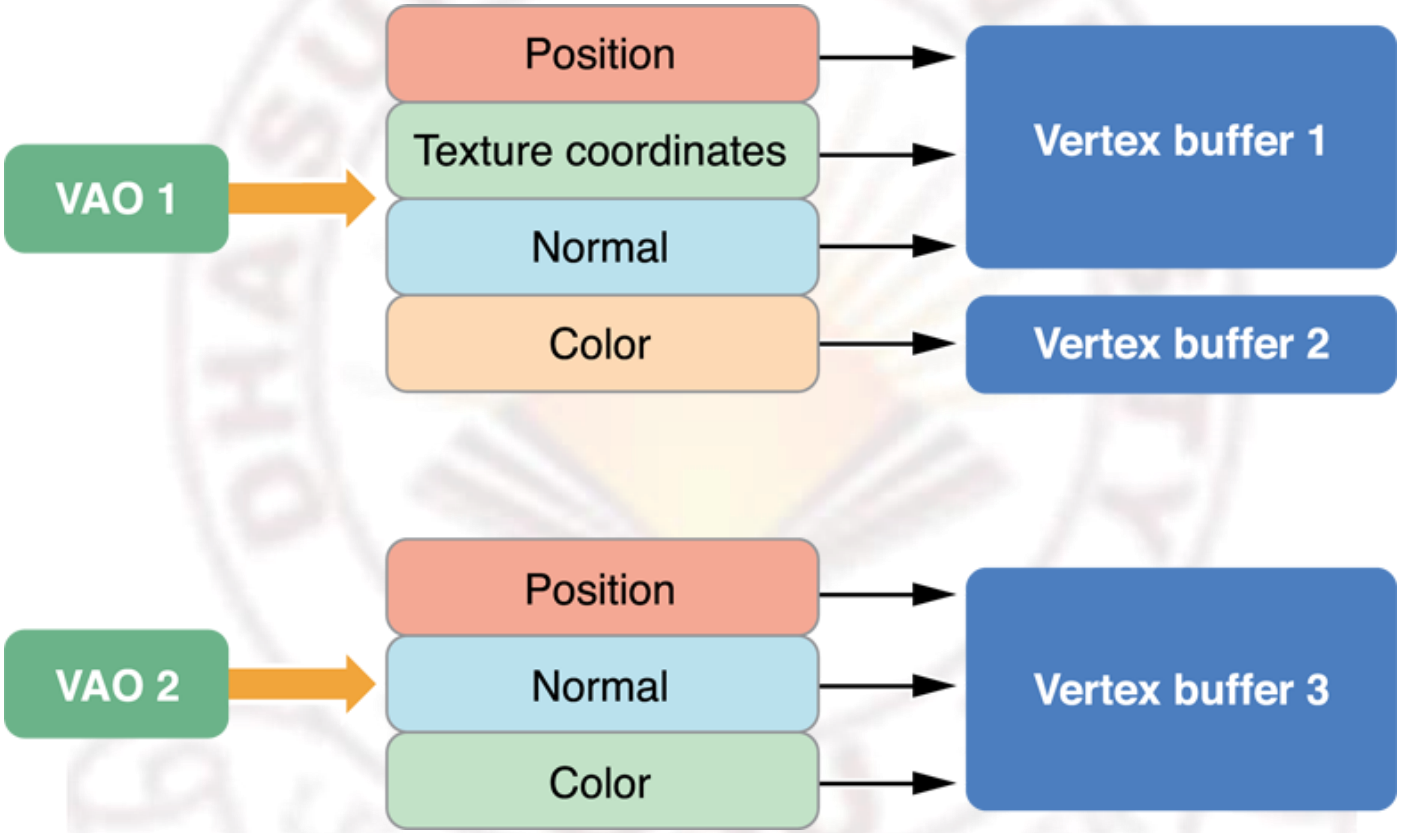
- Create a `GLSLShader` reference `GLSLShader shader;`
- Call `GLSLShader::LoadFromFile` to load the shader from file
  - `shader.LoadFromFile(GL_VERTEX_SHADER, "../shaders/ch1_2.vert");`
  - `shader.LoadFromFile(GL_FRAGMENT_SHADER, "../shaders/ch1_2.frag");`
- Compile and link shader using `shader.CreateAndLinkProgram();`
- Call `GLSLShader::Use()` to use the shader
- Call `GLSLShader::AddAttribute/AddUniform` to store the locations of attributes and uniforms
  - `shader.AddAttribute("vVertex");`
  - `shader.AddAttribute("vColor");`
  - `shader.AddUniform("MVP");`
- Set values of shader uniform that would not change during application lifetime
- Call `GLSLShader::UnUse()` to unuse the shader

What are buffer objects?

- OpenGL® objects that store an array of unformatted memory allocated by the OpenGL context (aka: the GPU)
- Used to store
  - vertex data,
  - pixel data retrieved from images or
  - framebuffer, and
  - a variety of other things

Vertex Array Object

- Vertex Array Object (VAO)is an OpenGL Object that stores all of the state needed to supply vertex data
- It stores the format of the vertex data as well as references to Buffer Objects providing the vertex data arrays
- Note that VAOs do not copy, freeze or store the *contents* of the referenced buffers - if you change any of the data in the buffers referenced by an existing VAO, those changes will be seen by users of the VAO.


VAO 1
Position
Texture coordinates
Normal
Color
Vertex buffer 1
Vertex buffer 2
VAO 2
Position
Normal
Color
Vertex buffer 3

Creating Geometry

- We create a custom Vertex struct to allow interleaving of attributes

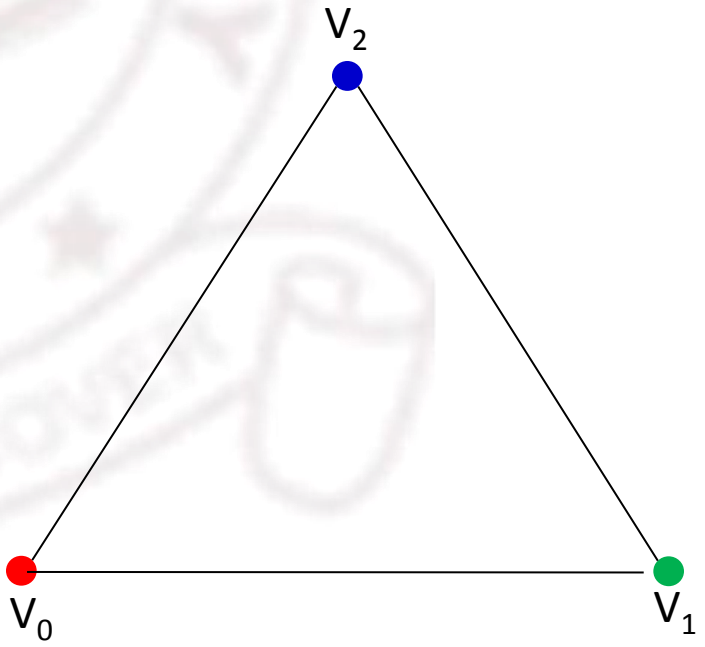
```
struct Vertex {
   glm::vec3 position;
   glm::vec3 color;
};
```

- Create an array of three Vertex and indices to contain topology

```
Vertex vertices[3];
GLushort indices[3];
```

- Fill the vertices and indices arrays

```
vertices[0].color=glm::vec3(1,0,0);
vertices[1].color=glm::vec3(0,1,0);
vertices[2].color=glm::vec3(0,0,1);
vertices[0].position=glm::vec3(-0.5,-0.5,0);
vertices[1].position=glm::vec3(0.5,-0.5,0);
vertices[2].position=glm::vec3(0,0.5,0);
indices[0] = 0;
indices[1] = 1;
indices[2] = 2;
```



Creating Vertex Array and Buffer Object

- First call glGenVertexArrays, glGenBuffers to create vertex array and buffer objects

```
glGenVertexArrays(1, &vaoID);
glGenBuffers(1, &vboVerticesID);
glGenBuffers(1, &vboIndicesID);
```

- Now bind the vertex array object

```
glBindVertexArray(vaoID);
```

- Any buffer object bound after this call will be automatically referenced by the bound VAO

Storing Data in Buffer Object

- Bind the buffer object to an appropriate binding point
  - `glBindBuffer (GL_ARRAY_BUFFER, vboVerticesID);`
- Next, pass data to buffer object memory using glBufferData
  - `glBufferData (GL_ARRAY_BUFFER,sizeof(vertices), &vertices[0], GL_STATIC_DRAW);`
- Next, enable all required attribute indices and set their data format
  - `glEnableVertexAttribArray(shader.getAttribute("vVertex"));`
  - `glVertexAttribPointer(shader.getAttribute("vVertex"), 3, GL_FLOAT, GL_FALSE,stride,0);`
  - `glEnableVertexAttribArray(shader.getAttribute("vColor"));`
  - `glVertexAttribPointer(shader.getAttribute("vColor"), 3, GL_FLOAT, GL_FALSE,stride, (const GLvoid*)offsetof(Vertex, color));`
- Next, assign element indices to a different binding point and update data
  - `glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vboIndicesID);`
  - `glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), &indices[0], GL_STATIC_DRAW);`

Vertex Shader Setup

```
#version 440 core

layout(location = 0) in vec3 vVertex;//object space vertex position
layout(location = 1) in vec3 vColor;//per-vertex colour

//output from the vertex shader
smooth out vec4 vSmoothColor;

//uniform
uniform mat4 MVP;//combined modelview projection matrix

void main()
{
   vSmoothColor = vec4(vColor,1);
   gl_Position = MVP*vec4(vVertex,1);
}
```

```
glEnableVertexAttribArray(shader.getAttribute("vVertex"));
glVertexAttribPointer(shader.getAttribute("vVertex"), 3, GL_FLOAT,GL_FALSE,
stride,0);
glEnableVertexAttribArray(shader.getAttribute("vColor"));
glVertexAttribPointer(shader.getAttribute("vColor"), 3, GL_FLOAT,GL_FALSE,stride,
(const GLvoid*)offsetof(Vertex, color));
```

Fragment Shader Setup

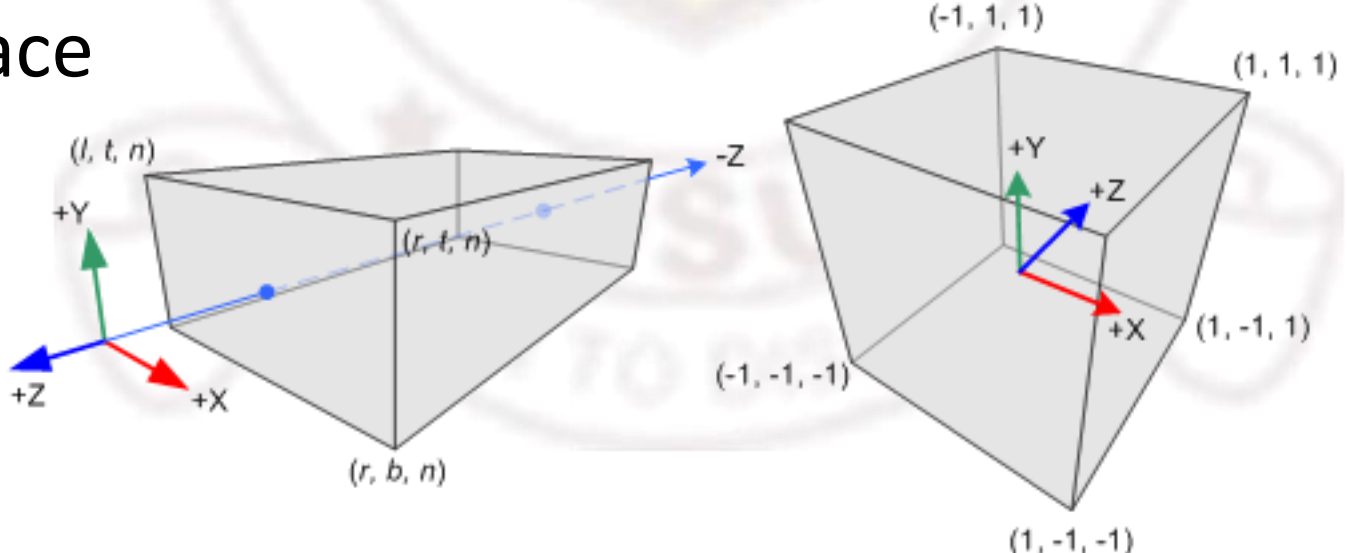
```
#version 440 core

layout(location=0) out vec4 vFragColor;

smooth in vec4 vSmoothColor; //interpolated from vertex shader

void main()
{
   vFragColor = vSmoothColor;
}
```

Setup Projection Matrices and Viewport

- Reshape callback function is called whenever the window is reshaped
  - `void OnResize(int w, int h) {`
    `glViewport(0,0,(GLsizei)w,(GLsizei) h);`
    `P = glm::ortho(-1,1,-1,1);`
  - `}`
- The new width and height are passed as parameters, we change the size of our 3D world to screen mapping using viewport

Orthographic Projection and ModelView Matrices

- Accommodates the whole 3D world into a box with coordinates [-1,-1,-1] to [1,1,1]
  - `P = glm::ortho(-1,1,-1,1);`
- The ModelView matrix is set to identity as the triangle coordinates are directly given in clip space



Internals of Orthographic Projection Matrix

- glm::ortho(l,r,t,b,n,f) defines the following 4x4 matrix

$$\begin{bmatrix} x_c \\ y_c \\ z_c \\ w_c \end{bmatrix} = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} x_e \\ y_e \\ z_e \\ w_e \end{bmatrix}$$

- $[x_c,y_c,z_c,w_c]^T$ are the clip space coordinates
- $[x_e,y_e,z_e,w_e]^T$ are the eye space coordinates

http://www.songho.ca/opengl/gl_projectionmatrix.html

Internals of Perspective Projection Matrix

- glm::perspective(fovY,aspectRatio, n, f) defines the following 4x4 matrix

$$t = \tan(fovY/2)$$

$$\begin{bmatrix} x_c \\ y_c \\ z_c \\ w_c \end{bmatrix} = \begin{bmatrix} \frac{1}{aspectRatio * t} & 0 & 0 & 0 \\ 0 & \frac{1}{t} & 0 & 0 \\ 0 & 0 & \frac{f+n}{n-f} & \frac{2fn}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_e \\ y_e \\ z_e \\ w_e \end{bmatrix}$$

- $[x_c,y_c,z_c,w_c]^T$ are the clip space coordinates
- $[x_e,y_e,z_e,w_e]^T$ are the eye space coordinates

Drawing Triangle

- Enable the current shader
  - `shader.Use();`
- Set the value of the MVP shader uniform
  - `glUniformMatrix4fv(shader.getUniform ("MVP"),1,GL_FALSE, glm::value_ptr(P*MV));`
- Make glDrawElements call
  - `glDrawElements(GL_TRIANGLES, 3, GL_UNSIGNED_SHORT, 0);`
- Unbind the shader
  - `shader.UnUse();`

Cleanup

- Delete shader program object
  - `shader.DeleteShaderProgram();`
- Delete all objects vertex array object/s, vertex buffer object/s
  - `glDeleteBuffers(1,&vboVerticesID);`
  - `glDeleteBuffers(1,&vboIndicesID);`
  - `glDeleteVertexArrays(1, &vaoID);`

Modern OpenGL Learning Resources

- Modern OpenGL Books
  - OpenGL Super Bible (6th Ed.)
  - Anton's OpenGL 4 Tutorials
  - OpenGL 4 Shading Language Cookbook (2nd Ed.)
  - OpenGL Development Cookbook
  - OpenGL Insights
  - OpenGL ES 3 Programming Guide
  - WebGL Programming Guide
  - WebGL Insights
- Other GPU Books
  - GPU Gems Series
  - GPU Pro Series
- Online Tutorials
  - Anton's OpenGL 4 Tutorials (http://antongerdelan.net/opengl/)
  - Learning Modern 3D Graphics Programming (http://arcsynthesis.org/gltut/)
  - Swiftless Tutorials (http://www.swiftless.com/opengl4tuts.html)
  - http://www.opengl-tutorial.org/
  - http://www.rastertek.com/tutgl40.html

Thanks

- Open for joint research and collaboration
- Email:
  - mova0002@e.ntu.edu.sg
  - mobeen.movania@dsu.edu.pk
- Github
  - http://github.com/mmmovania
- URLs:
  - Website: http://cgv.dsu.edu.pk
  - Blog: http://mmmovania.blogspot.com
  - Tutorial: http://cgv.dsu.edu.pk/tutorials/FIT2014/
  - Github: https://github.com/mmmovania/FIT2014_OpenGL_Tutorial

Supplementary Slides

Offscreen Rendering using Framebuffer Object (FBO)

- Enables render to texture functionality
- Often used in GPGPU computation to store intermediate results from a shader output
- Can write to multiple textures in a single pass by writing to different drawbuffers
- Floating point formats are used for greater precision
- Renderbuffers provide support for depth testing

Typical Setup Code for FBO

- Initialize the FBO by generating an FBO object (`glGenFramebuffers`) and bind it to a framebuffer target (`glBindFramebuffer`)
- Generate a texture and bind it as a color attachment of FBO
- Check for framebuffer completeness
- If depth testing is required, generate a render buffer object and bind it as a depth attachment of FBO

```
Gluint fbo;
glGenFramebuffers( 1, &fbo );
glBindFramebuffer( GL_DRAW_FRAMEBUFFER,fbo);
glFramebufferTexture2D( GL_DRAW_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
                        GL_TEXTURE_2D, texID, 0);
GLenum status = glCheckFramebufferStatus(GL_DRAW_FRAMEBUFFER);
if(status == GL_FRAMEBUFFER_COMPLETE) {
   puts("FBO setup successfull");
} else {
   puts("Problem in setting up Single FBO texture");
}
glBindFramebuffer( GL_DRAW_FRAMEBUFFER,0);
```

Typical Rendering Code for FBO

- Bind the FBO to a framebuffer target
- Set the FBO's color attachment as the current draw buffer
- Reset the viewport to the size of the FBO attachment
- Apply the shader and then draw a fullscreen quad
- Unbind the FBO and reset the default back buffer to resume rendering to screen

```
//bind FBO to enable render to texture
glBindFramebuffer( GL_DRAW_FRAMEBUFFER, fboID );
glDrawBuffer(GL_COLOR_ATTACHMENT0);
glViewport( 0, 0, imageWidth, imageHeight);

//apply the shader
shader.Use();
    //draw the fullscreen quada
    glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_SHORT, 0);
shader.UnUse();
//unbind the FBO to resume default back buffer
glBindFramebuffer( GL_DRAW_FRAMEBUFFER, 0);
glDrawBuffer(GL_BACK_LEFT);

//continue rest of the rendering code
```

Tips on Converting and Algorithm to GPU shaders

- In order to utilize the parallelism of the GPU for GPGPU calculations, the fragment shader can be used
- Identify how the given algorithm can be mapped to a fragment shader
- Use offscreen rendering to generate intermediate results. For efficiency, use a pair of textures and ping pong between them.
- Scale space generation can be easily carried out by using mipmaps along with offscreen rendering using FBO
- Minimize if/else conditions by using shader tricks. The GPU will execute both if and else branches together and the slower of them will dictate the final execution speed. e.g. assume the following shader

```
void main() {
    bool result = SomeCalculation();
    if(result)
       gl_FragColor = PerformMethod1();
    else
       gl_FragColor = PerformMethod2();
}
```

- The previous shader snippet could be rewritten by using mix GLSL function in this way

```
void main() {
    bool result = SomeCalculation();
    gl_FragColor = mix(PerformMethod2(),
                       PerformMethod1(),
                       int(result));
}
```

- Minimize use of square roots in calculation by using shader tricks and maximize the use of `dot` function as it is a single instruction in GPU hardware e.g. Circular points in Modern OpenGL

Rendering of Circular Points in Modern OpenGL

- The equation of a circle needs a square root as follows:

```
float dist = sqrt(pos.x*pos.x + pos.y*pos.y);
if(dist>=radius)
   discard; //don't render fragments outside circle
else
   gl_FragColor = color;
```

- The above code can be optimized as follows:

```
float dist2 = dot(pos, pos);
if(dist2 >= (radius*radius))
   discard; //don't render fragments outside circle
else
   gl_FragColor = color;
```