Everything in life is transient

11 OpenGL 4.0

Department of Computer Science and Software Engineering
University of Canterbury, New Zealand.

R. Mukundan (mukundan@canterbury.ac.nz)

Changes in OpenGL

- OpenGL 1.0 designed for the fixed-function pipeline is not optimal for today's hardware.
- Users must be able to choose a rendering context based on a specific OpenGL version.
- A thorough overhaul of the API began in 2007, with the design of OpenGL 3.0 in 2008, and OpenGL 4.0 in 2010
  - Fundamental changes in the rendering paradigm, suitable for hardware optimisation.
  - GPU processing given utmost importance. Allows you to create functions (**shaders**) that graphics hardware can execute.
- OpenGL 5 expected to be released later this year!

More Shader/GPU Functionality

- OpenGL 3.0 introduced a deprecation model with several functions marked for deletion in future versions.
  - All **fixed-function mode** vertex and fragment processing routines were deprecated.
  - **Immediate mode** rendering using `glBegin()-glEnd()` blocks also deprecated.
- OpenGL 3.2 divided the specification into two profiles:
  - Compatibility profile: Backward compatible, allowing access to old APIs
  - Core profile: The core API specification.

Motivation

- The ability to program the graphics hardware allows you to achieve a wider range of rendering effects.
- Traditional lighting functions and the fixed functionality of the graphics pipeline are fine only for ‘common things’. They have now been removed from the core profile.
- Developers have more freedom to define the actions to be taken at different stages of processing.

OpenGL Context: Example

```
#include <iostream>
#include <GL/glew.h>
#include <GL/freeglut.h>
using namespace std;
...

int main(int argc, char** argv)
{
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_RGB);
  glutInitWindowSize(500, 500);
  glutCreateWindow("A Triangle");
  glutInitContextVersion (4, 2);
  glutInitContextProfile (GLUT_CORE_PROFILE);
   ...
```

Getting Version Info

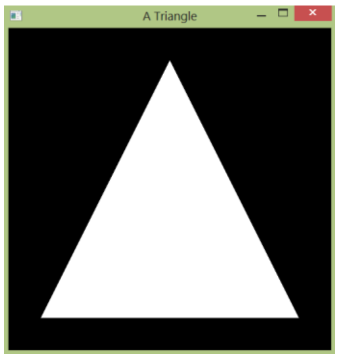
Version.cpp

```
const GLubyte *version = glGetString(GL_VERSION);
const GLubyte *renderer = glGetString(GL_RENDERER);
const GLubyte *vendor = glGetString(GL_VENDOR);
```

```
OpenGL version: 4.2.0
OpenGL vendor: NVIDIA Corporation
OpenGL renderer: GeForce 710M/PCIe/SSE2
Version (ints): 4.2
```

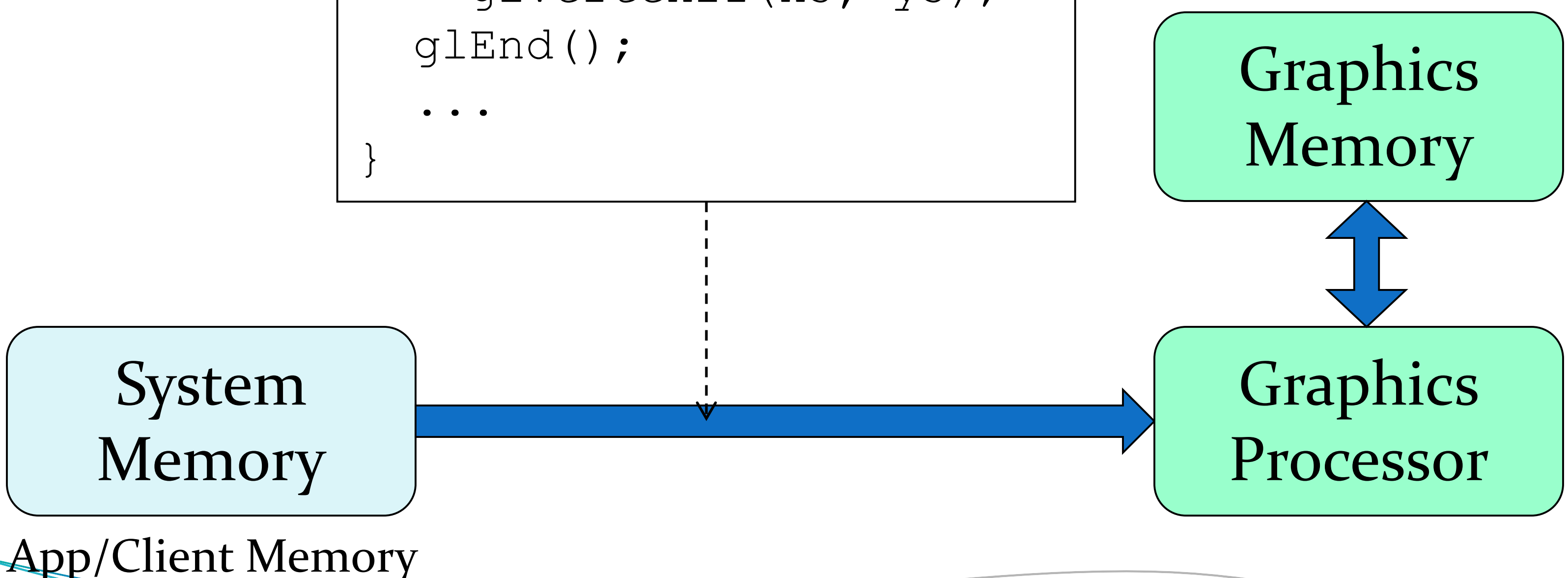
Primitive Drawing (OpenGL 1)

(Immediate Mode Rendering)




```
void display()
{
   ...
   glBegin(GL_TRIANGLES);
     glVertex2f(x1, y1);
     glVertex2f(x2, y2);
     glVertex2f(x3, y3);
   glEnd();
   ...
}
```

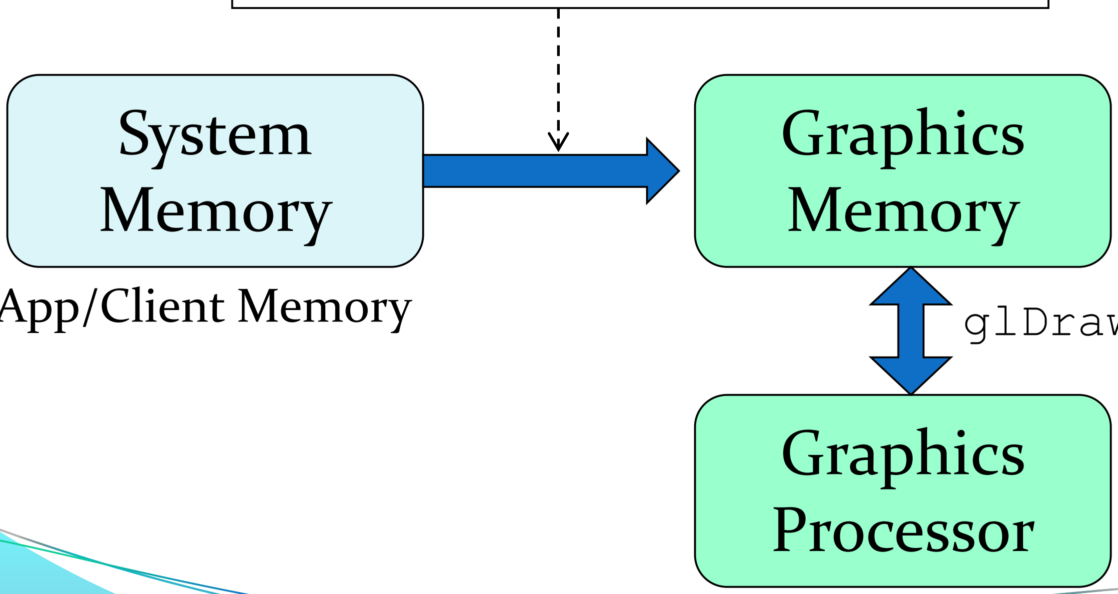
Deprecated!



Primitive Drawing (OpenGL 4)

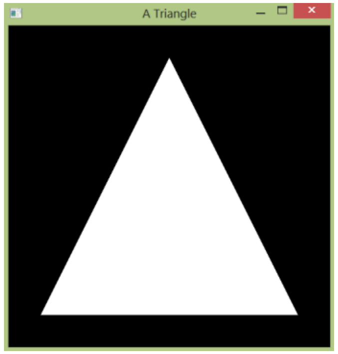
(Non-Immediate Mode Rendering)

```
void initialise()
{
    ...
    glBufferData(...);
    glBufferSubData(...);
    ...
}
```

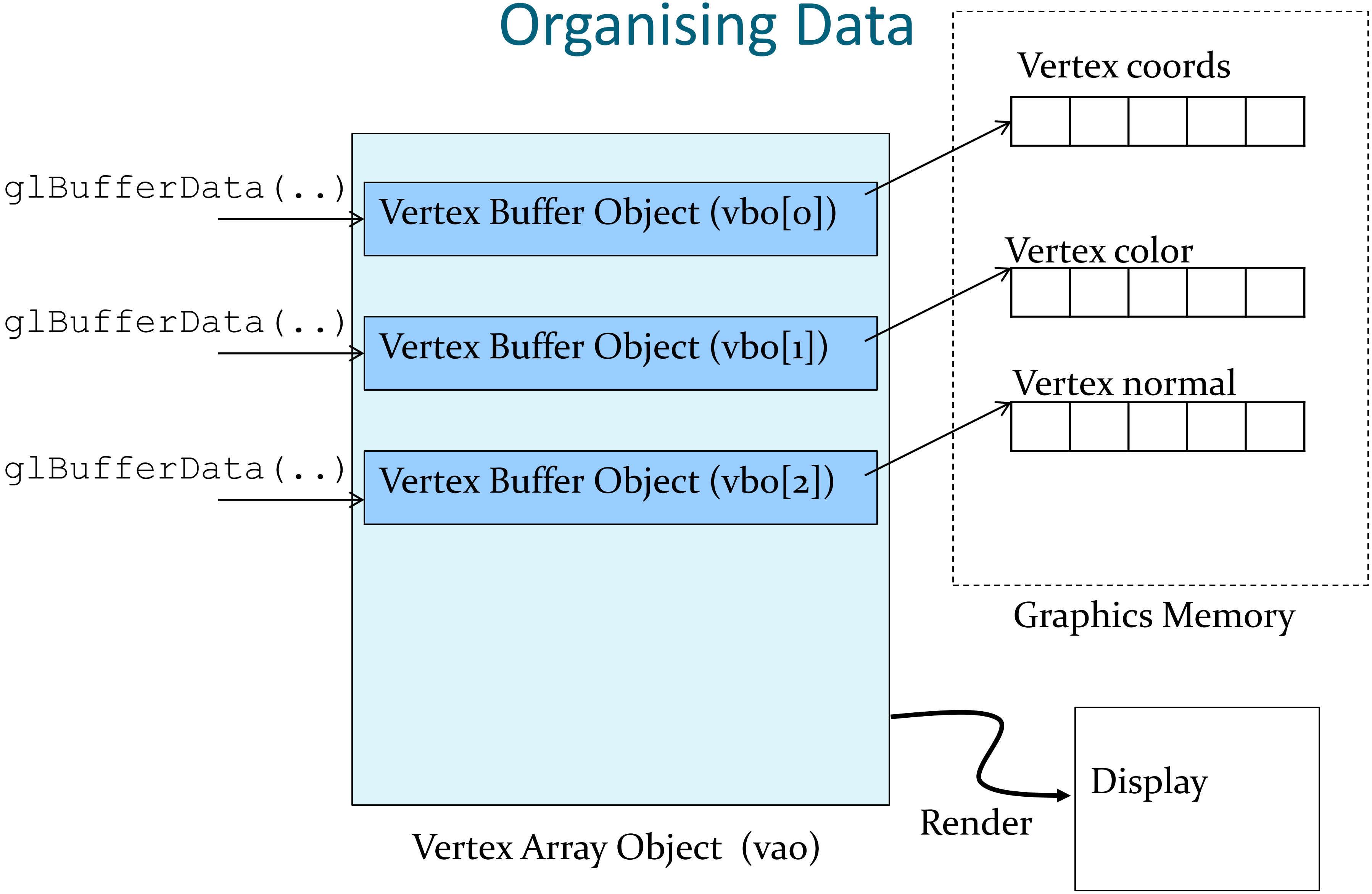


App/Client Memory

glDrawArrays(GL_TRIANGLES,0,3);

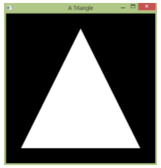


Organising Data



Vertex Buffer Objects

Draw1.cpp



- A vertex buffer object (VBO) represents the data for a particular vertex attribute in video memory.
- Creating VBOs:
  1. Generate a new buffer object "vbo"
  2. Bind the buffer object to a target
  3. Copy vertex data to the buffer

```
  GLuint vbo;
1 glGenBuffers(1, &vbo);
2 glBindBuffer(GL_ARRAY_BUFFER, vbo);

3 glBufferData(GL_ARRAY_BUFFER, sizeof(verts), verts,
                                       GL_STATIC_DRAW);
  glEnableVertexAttribArray(0);
  glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0, NULL);
```

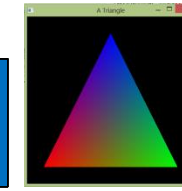
Multiple VBOs

Draw2.cpp

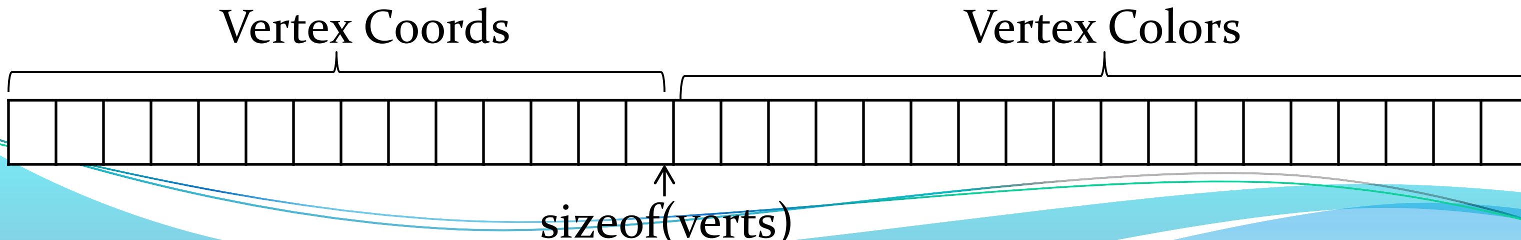
```
GLuint vbo[2];
glGenBuffers(2, vbo);    //Two VBOs

glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);  //First VBO
glBufferData(GL_ARRAY_BUFFER, sizeof(verts), verts,
                                          GL_STATIC_DRAW);
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0, NULL);

glBindBuffer(GL_ARRAY_BUFFER, vbo[1]);  //Second VBO
glBufferData(GL_ARRAY_BUFFER, sizeof(cols), cols,
                                         GL_STATIC_DRAW);
glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE, 0, NULL);
```

Packing Several Attributes in 1 VBO

Draw3.cpp



```
GLuint vbo;
glGenBuffers(1, &vbo);    //Only 1 vbo

glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, sizeof(verts)+sizeof(cols),
                                  verts,  GL_STATIC_DRAW);
glBufferSubData(GL_ARRAY_BUFFER, sizeof(verts), sizeof(cols),
                                                     cols);
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0, NULL);
glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE, 0,
                                  (GLvoid *)sizeof(verts));
```



Vertex Array Object

- A vertex array object (VAO) encapsulates all the state needed to specify vertex data of an object.
- Creating VAOs:
  1. Generate a new vertex array object “vao”
  2. Bind the vertex array object (initially empty)
  3. Create constituent VBOs and transfer data

```
1 glGenVertexArrays(1, &vao);
2 glBindVertexArray(vao);
...
3 glGenBuffers(3, vbo);
...
```

Rendering

- Bind the VAO representing the vertex data
- Render the collection of primitives using `glDrawArray()` command:

```
glBindVertexArray(vao);
glDrawArrays(GL_TRIANGLES, 0, 3);
```

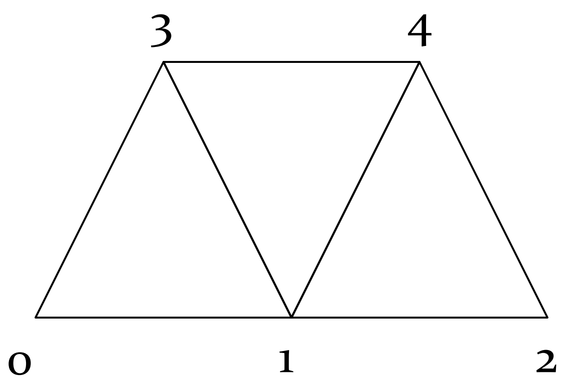
Primitive Type

Index of first primitive

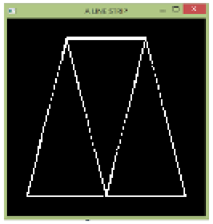
Count

Drawing Using Vertex Indices

- Mesh data is often represented using vertex indices to avoid repetition of vertices



`Draw4.cpp`



Polygonal Line : 3 0 1 3 4 1 2 4

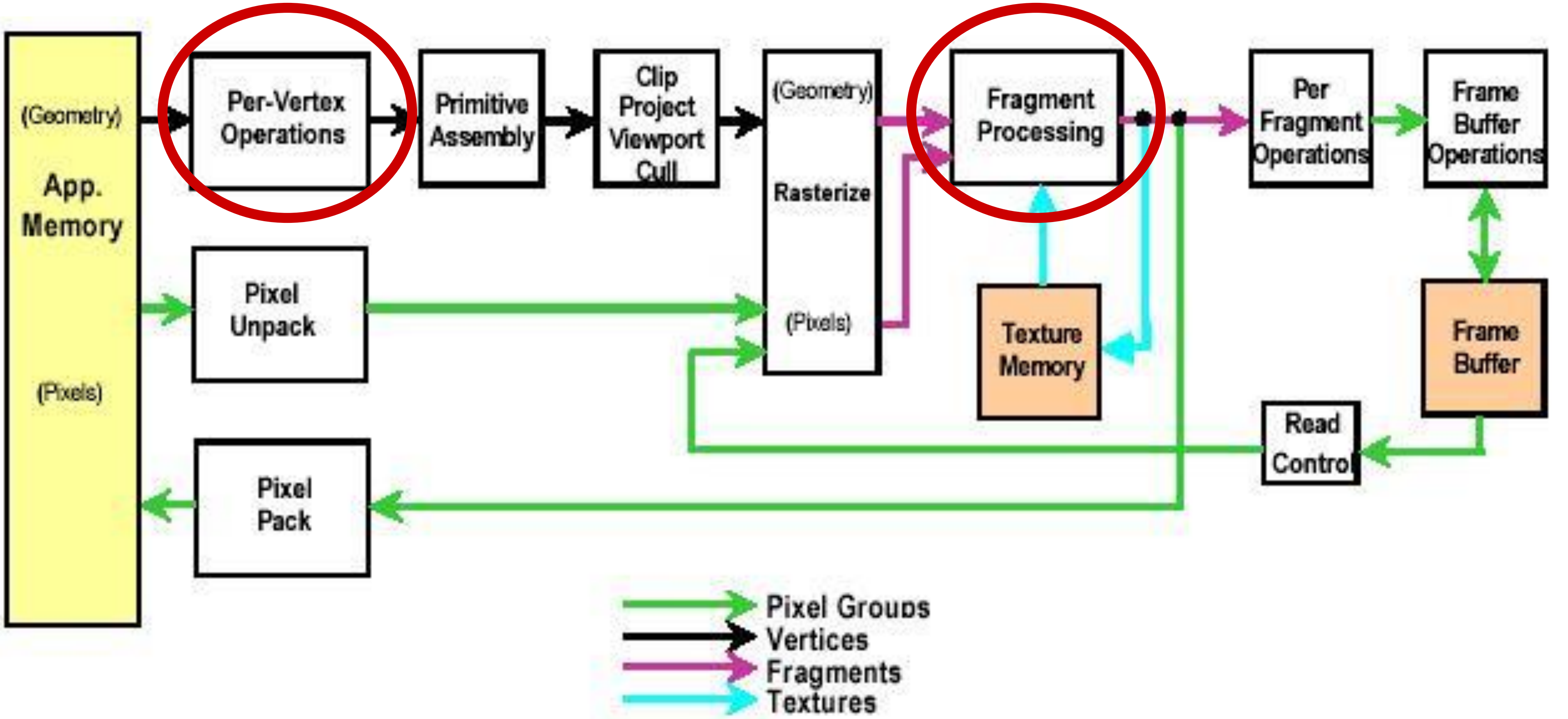
- The VBO for indices is defined using `GL_ELEMENT_ARRAY` as the target.
- Rendering of the mesh is done using the command `glDrawElements(..)`

Homework!

- Download and install
  - freeglut (http://freeglut.sourceforge.net) and
  - glew (http://glew.sourceforge.net)
- Run the following programs:
  - Version.cpp
  - Draw1.cpp
  - Draw2.cpp
  - Draw3.cpp
  - Draw4.cpp

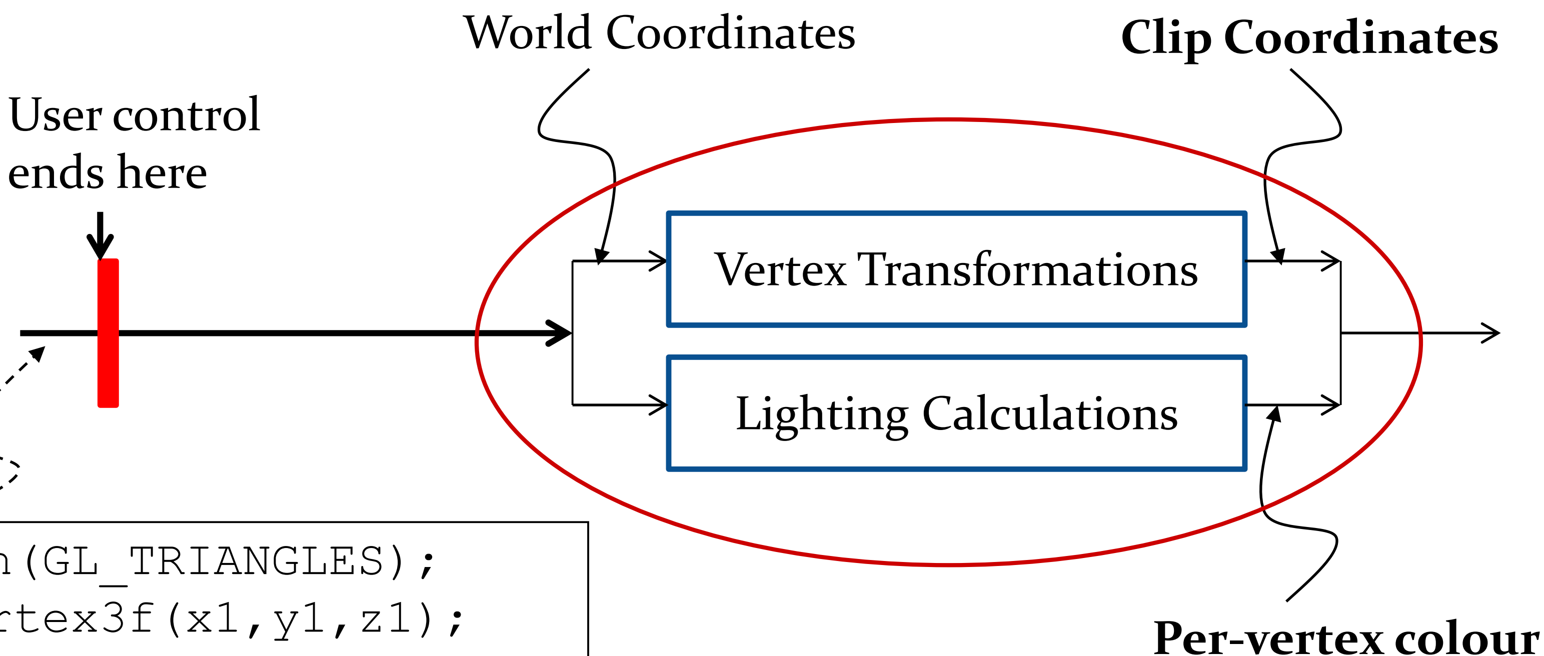
  Uses shader code `Simple.vert, Simple.frag`
- Discuss any issues using class forum

OpenGL Fixed Function Pipeline



OpenGL Fixed Function Pipeline

The Vertex Processing Stage (T&L Stage)

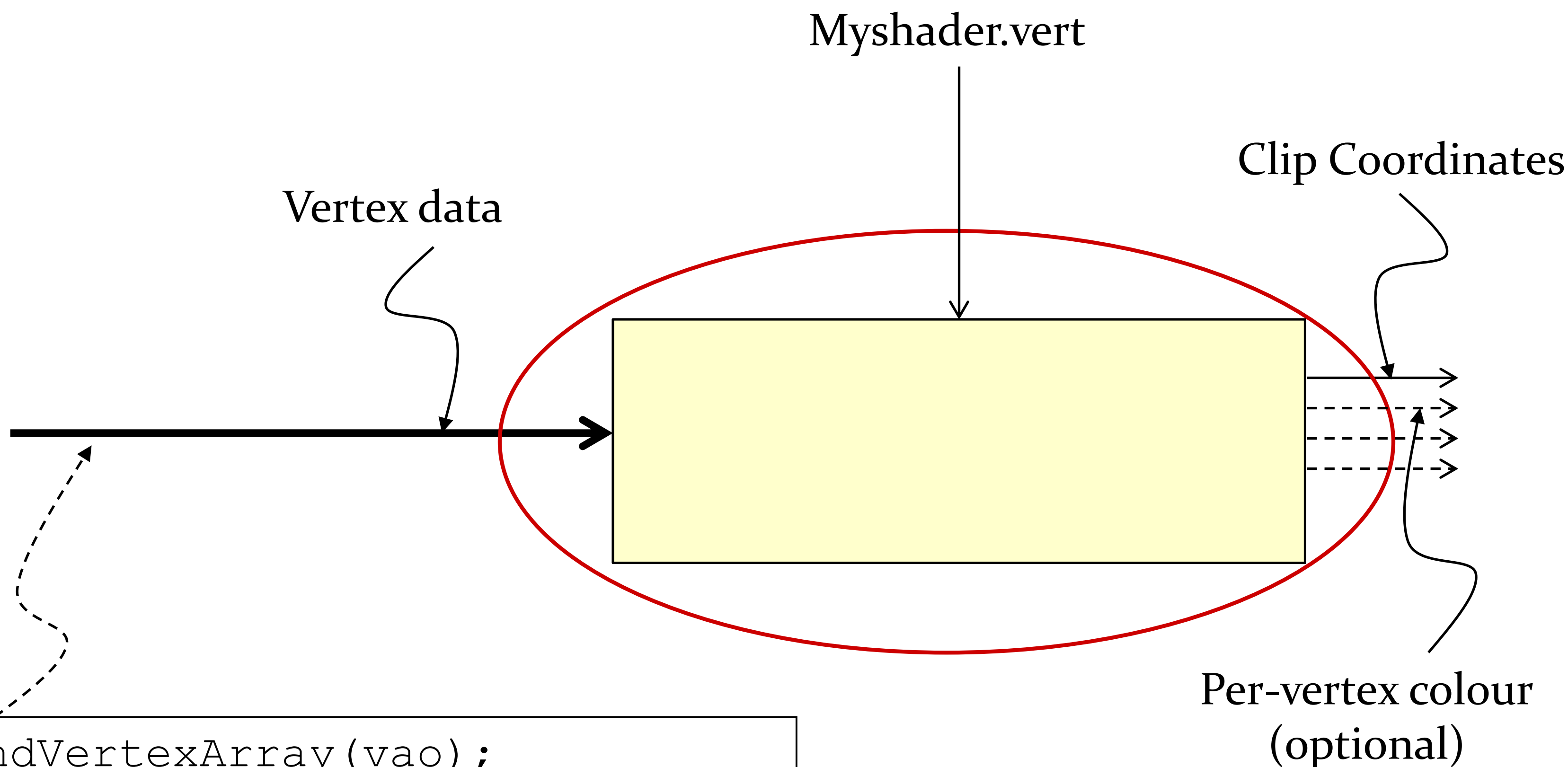



```
glBegin(GL_TRIANGLES);
  glVertex3f(x1,y1,z1);
  glVertex3f(x2,y2,z2);
  glVertex3f(x3,y3,z3);
glEnd();
```

Deprecated!

Programmable Pipeline

The Vertex Shader




```
glBindVertexArray(vao);
glDrawArrays(GL_TRIANGLES, 0, 3);
```

Vertex Shader: Example

Draw2.cpp

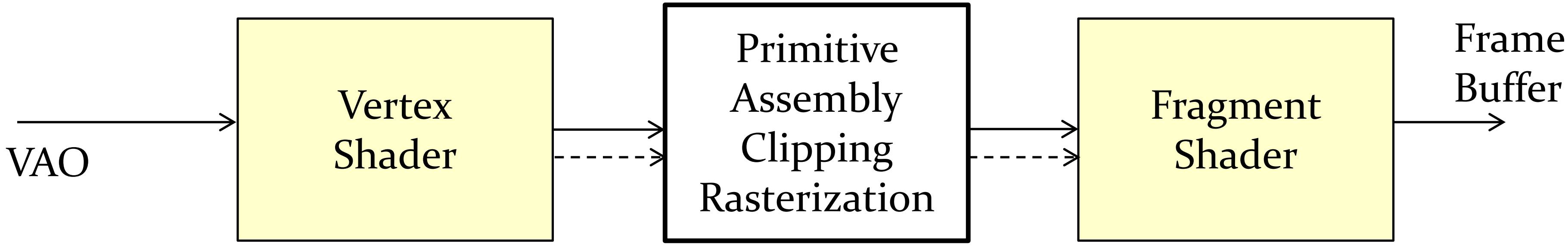
Application

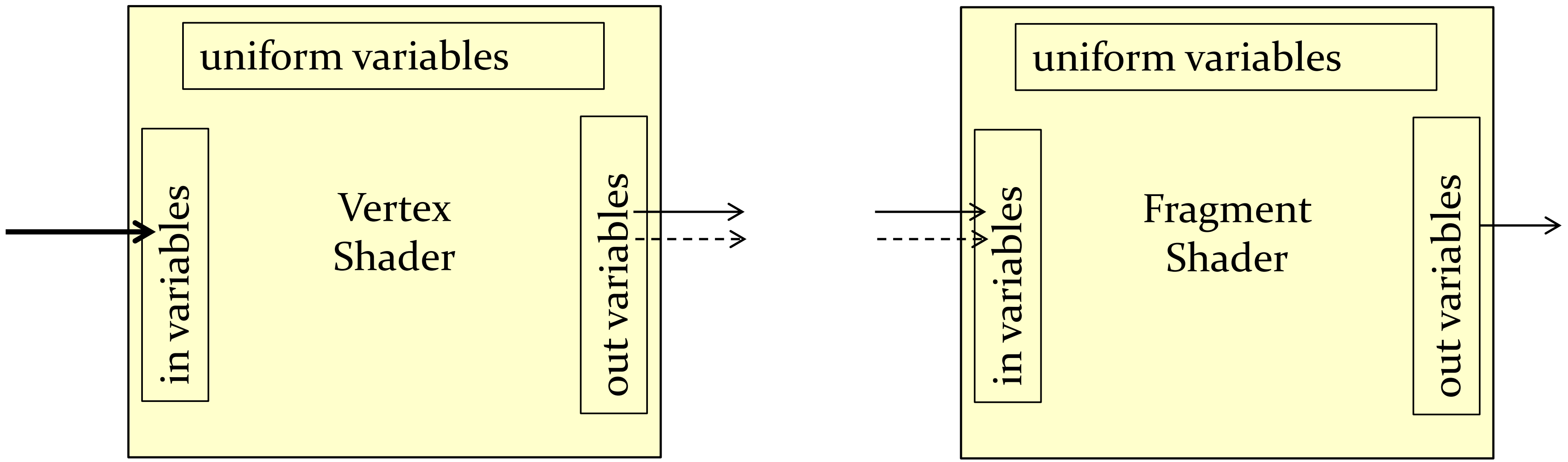
```
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0, NULL);

glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE, 0, NULL);
```

```
#version 330

layout (location = 0) in vec4 position;
layout (location = 1) in vec4 color;

out vec4 theColor;

void main()
{
        gl_Position = position;
        theColor = color;
}
```

Simple.vert

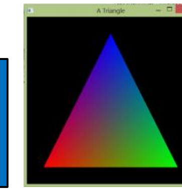
Vertex and Fragment Shaders





Fragment Shader: Example

Draw2.cpp



Vertex Shader
`Simple.vert`

```
#version 330

layout (location = 0) in vec4 position;
layout (location = 1) in vec4 color;

out vec4 theColor;

void main()
{
        gl_Position = position;
        theColor = color;
}
```

Fragment Shader
`Simple.frag`

```
#version 330

smooth in vec4 theColor;

out vec4 outputColor;

void main()
{
        outputColor = theColor;
}
```

GLSL – Language Features

Vector Types: `vec2, vec3, vec4,`
`ivec2, ivec3, ivec4`

```
vec3 v1, v2, v3;
vec4 pos1, pos2;
vec2 p;
float zcoord, d;

v1 = vec3(-1.0, 2.0, 0.5);
v2 = vec3(0.2);  //same as (0.2,0.2,0.2)
v3 = v1 + v2;
pos1 = vec4(v3, 1.0);
p = v3.xy;            //swizzle operator
zcoord = v3.z;
pos2 = pos1.ywxx;  // (2.2, 1.0, -0.8, -0.8)
d = dot(v2, v3);
```

GLSL – Language Features

Matrix Types: `mat2, mat3, mat4`

```
mat4 m1, m2;
mat2 m3;
vec4 v1,v2,v3,v4;
vec2 p;
float zcoord;

m1 = mat4(1.0);  //Identity matrix
v1 = m1[2];      //Third column of matrix m1
m2 = mat4(v1,v2,v3,v4);  //column vectors
m3 = mat2(1.0, 6.0, 0.2, 0.8)  //1st col=(1., 6.)
v4 = m2 * v3;
```

Vertex Shader

When a vertex shader is executed, the following fixed functionality operations are affected:

- Vertex coordinates are not multiplied by model-view, projection matrices
- Texture coordinates are not multiplied by texture matrices
- Normals are not transformed to eye coordinates
- Normals are not rescaled or normalized
- Per vertex lighting is not performed
- Color material computations are not performed
- Texture coordinataes are not generated automatically.

Defining Transformations

- We will need to define transformations and projections using our own functions!
- The GLM (GL Mathematics) library written by Christophe Riccio provides functionality similar to the deprecated functions.
- GLM is a header-only library that can be downloaded from http://glm.g-trunc.net

```
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
```

Defining Transformations

- The Model-view-projection matrix must be made available in the vertex shader for transforming vertices to clip coordinates.
- Uniform variables provide a mechanism for transferring matrices and other values from your application to the shader.
- Uniform variables change less frequently compared to vertex attributes. They remain constant for every primitive.
- Important matrices:
  - Model-View Matrix (VM)
  - Model-View-Projection Matrix (PVM) See next slide.

Defining Transformations

Application

Draw5.cpp

```
GLuint matrixLoc;
matrixLoc = glGetUniformLocation(program, "mvpMatrix");
```

```
void display() {
glm::mat4 proj = glm::perspective(60.0f, 1.0f, 100.0f, 1000.0f);
glm::mat4 view = glm::lookAt(glm::vec3(0.0, 0.0, 150.0),
                             glm::vec3(0.0, 0.0, 0.0),
                             glm::vec3(0.0, 1.0, 0.0));
glm::mat4 matrix = glm::mat4(1.0);    //Identity matrix
matrix = glm::rotate(matrix, angle, glm::vec3(0.0, 1.0, 0.0));
glm::mat4 prodMatrix = proj*view*matrix;
glUniformMatrix4fv(matrixLoc, 1, GL_FALSE, &prodMatrix[0][0]);
...
```

Defining Transformations

Vertex Shader

Tetrahedron.vert

```
#version 330

layout (location = 0) in vec4 position;
uniform mat4 mvpMatrix;

void main()
{
	gl_Position = mvpMatrix * position;
}
```

Output in **clip coordinates**

Input in world coordinates

Lighting Calculations (Application)

- Lighting calculations are performed in eye-coordinates.
- We compute the following (using GLM) in our application:
  - Model-View matrix (VM)
  - Light's position in eye coordinates: VML
  - Inverse transformation matrix for the normal $(VM)^{-T}$

```
void display() {
...
glm::mat4 prodMatrix1 = view*matrix;
glm::mat4 prodMatrix2 = proj*prodMatrix1;
glm::vec4 lightEye = view*light;
glm::mat4 invMatrix = glm::inverse(prodMatrix1);
glUniformMatrix4fv(matrixLoc1, 1, GL_FALSE, &prodMatrix1[0][0]);
glUniformMatrix4fv(matrixLoc2, 1, GL_FALSE, &prodMatrix2[0][0]);
glUniformMatrix4fv(matrixLoc3, 1, GL_TRUE, &invMatrix[0][0]);
glUniform4fv(lgtLoc, 1, &lightEye[0]);
```

Lighting Calculations (Vertex Shader)

- Inside the vertex shader, we add the code to output the colour value at a vertex using the Phong-Blinn model.

Vertex shader

```
layout (location = 0) in vec4 position;
layout (location = 1) in vec3 normal;
uniform mat4 mvMatrix;
uniform mat4 mvpMatrix;
uniform mat4 norMatrix;
uniform vec4 lightPos;  //in eye coords

out vec4 theColour;

void main()
{
  vec4 white = vec4(1.0);  //Light's colour (diffuse & specular)
  vec4 grey = vec4(0.2);   //Ambient light
```

Continued on next slide

Lighting Calculations (Vertex Shader)

```
    vec4 posnEye = mvMatrix * position;    //point in eye coords
    vec4 normalEye = norMatrix * vec4(normal, 0);
    vec4 lgtVec = normalize(lightPos - posnEye);
    vec4 viewVec = normalize(vec4(-posnEye.xyz, 0));
    vec4 halfVec = normalize(lgtVec + viewVec);
    vec4 material = vec4(0.0, 1.0, 1.0, 1.0);  //cyan
    vec4 ambOut = grey * material;
    float shininess = 100.0;
    float diffTerm = max(dot(lgtVec, normalEye), 0);
    vec4 diffOut = material * diffTerm;
    float specTerm = max(dot(halfVec, normalEye), 0);
    vec4 specOut = white *  pow(specTerm, shininess);

    gl_Position = mvpMatrix * position;
    theColour = ambOut + diffOut + specOut;
}
```

Multi-Texturing

Texture 1

Texture 2

Texture Unit 0

```
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE2D, tex[0]);
texLoc1 = glGetUniformLocation
              (program, "tex1");
glUniform1i(texLoc1, 0);
```

Texture Unit 1

```
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE2D, tex[1]);
texLoc2 = glGetUniformLocation
              (program, "tex2");
glUniform1i(texLoc2, 1);
```

Texture Coordinates

```
glBindBuffer(GL_ARRAY_BUFFER, vboID[2]);
glBufferData(GL_ARRAY_BUFFER, num* sizeof(float), texC, GL_STATIC_DRAW);
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 0, NULL);
glEnableVertexAttribArray(2);
```

Multi-Texturing

Vertex Shader

```
layout (location = 0) in vec3 position;
layout (location = 1) in vec3 normal;
layout (location = 2) in vec2 texCoord;

uniform mat4 mvMatrix;
uniform mat4 mvpMatrix;
uniform mat4 norMatrix;

out vec4 diffRefl;
out vec2 TexCoord;


void main()
{
	gl_Position = mvpMatrix * vec4(position, 1.0);
	           …
	diffRefl =
	TexCoord = texCoord;
}
```

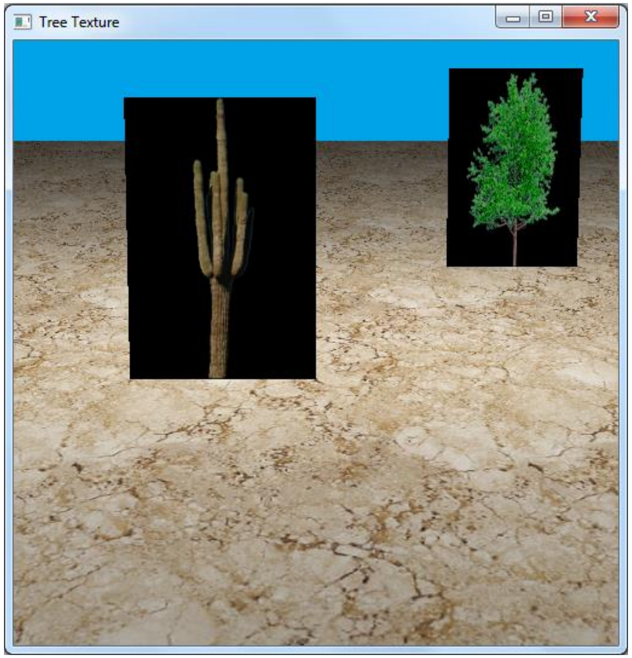
Multi-Texturing

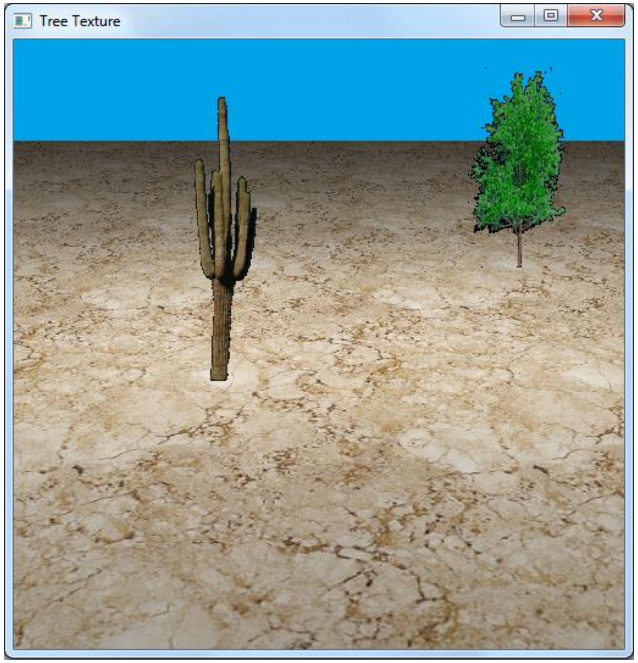
Fragment Shader:

```
uniform sampler2D tex1;
uniform sampler2D tex2;

in vec4 diffRefl;
in vec2 TexCoord;
out vec4 outputColor;

void main()
{
        vec4 tColor1 = texture(tex1, TexCoord);
        vec4 tColor2 = texture(tex2, TexCoord);

        outputColor =  diffRefl*(0.8*tColor1+ 0.2*tColor2);
}
```

Alpha Texturing

- A textured image of a tree should appear as being part of the surrounding scene, and not part of a rectangular 'board'.





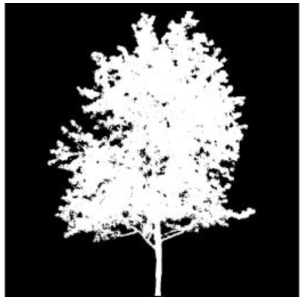
Alpha Texturing

- Use the alpha channel of an image (if available) to transfer only those pixels on the object.

Fragment Shader

```
uniform sampler2D texTree;

in vec2 TexCoord;
out vec4 outputColor;

void main()
{
  vec4 treeColor = texture(texTree, TexCoord);
  if(treeColor.a == 0) discard;
  outputColor = treeColor;
}
```

RGB



Alpha