

9

OpenGL- 4

You are in charge (of everything!)

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

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



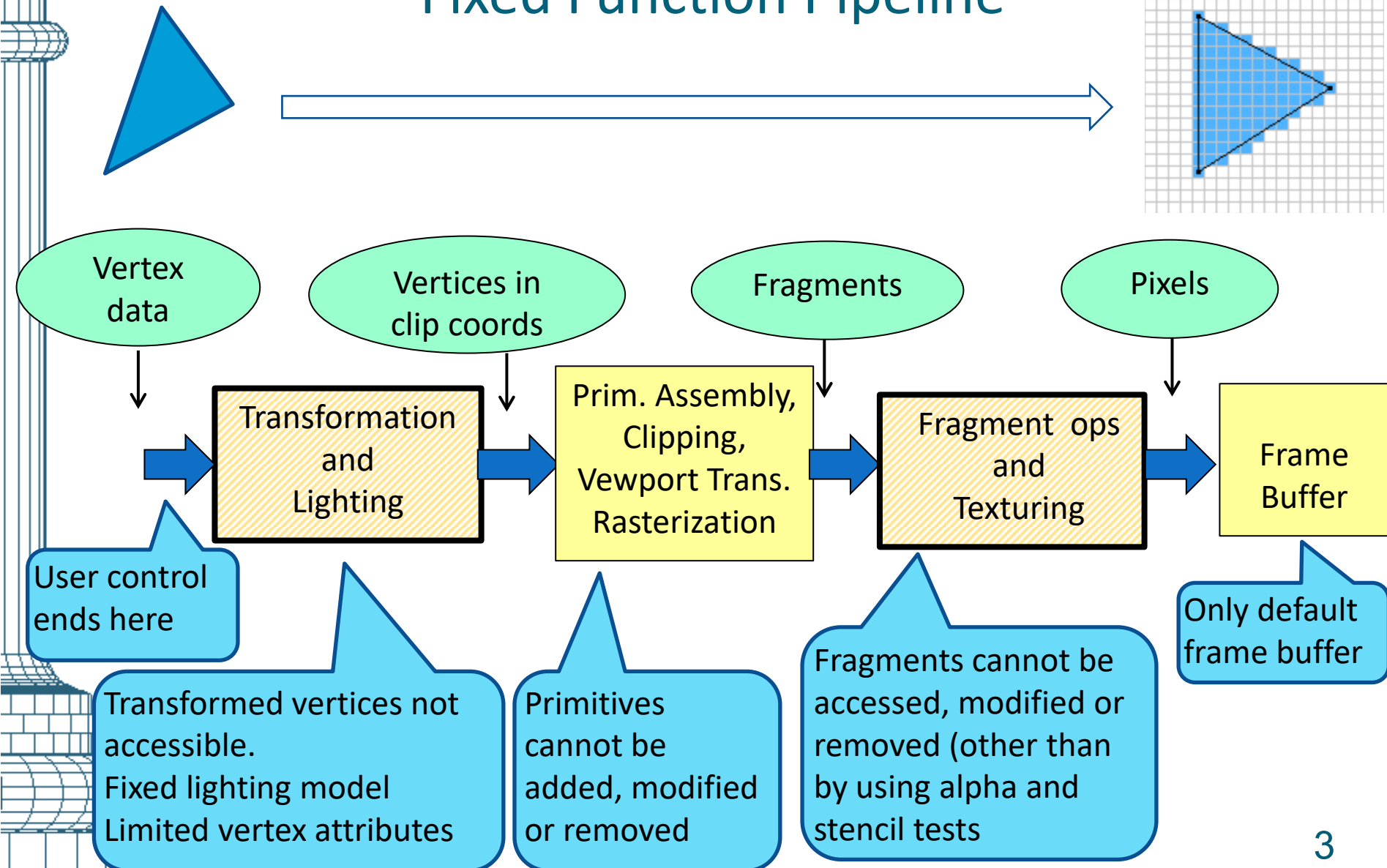
Motivation

- The ability to program the graphics hardware allows you to achieve a wider range of rendering effects that give optimal performance.
- 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.
- Downside: The user needs to specify the computations to be done at each stage.

Traditional OpenGL (OpenGL-1,2) Fixed Function Pipeline

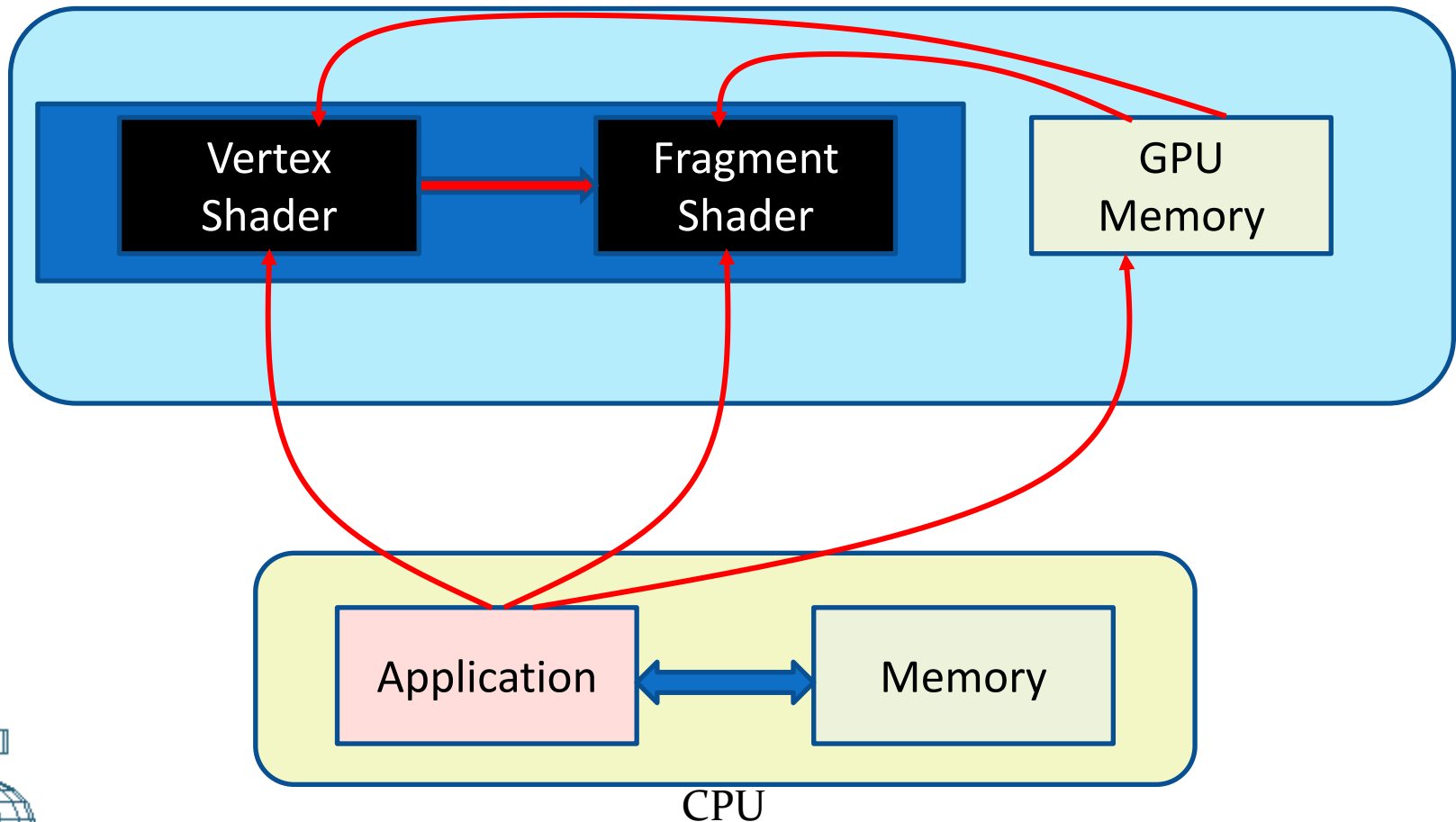
Primitives
(3D)

Pixels
(2D)

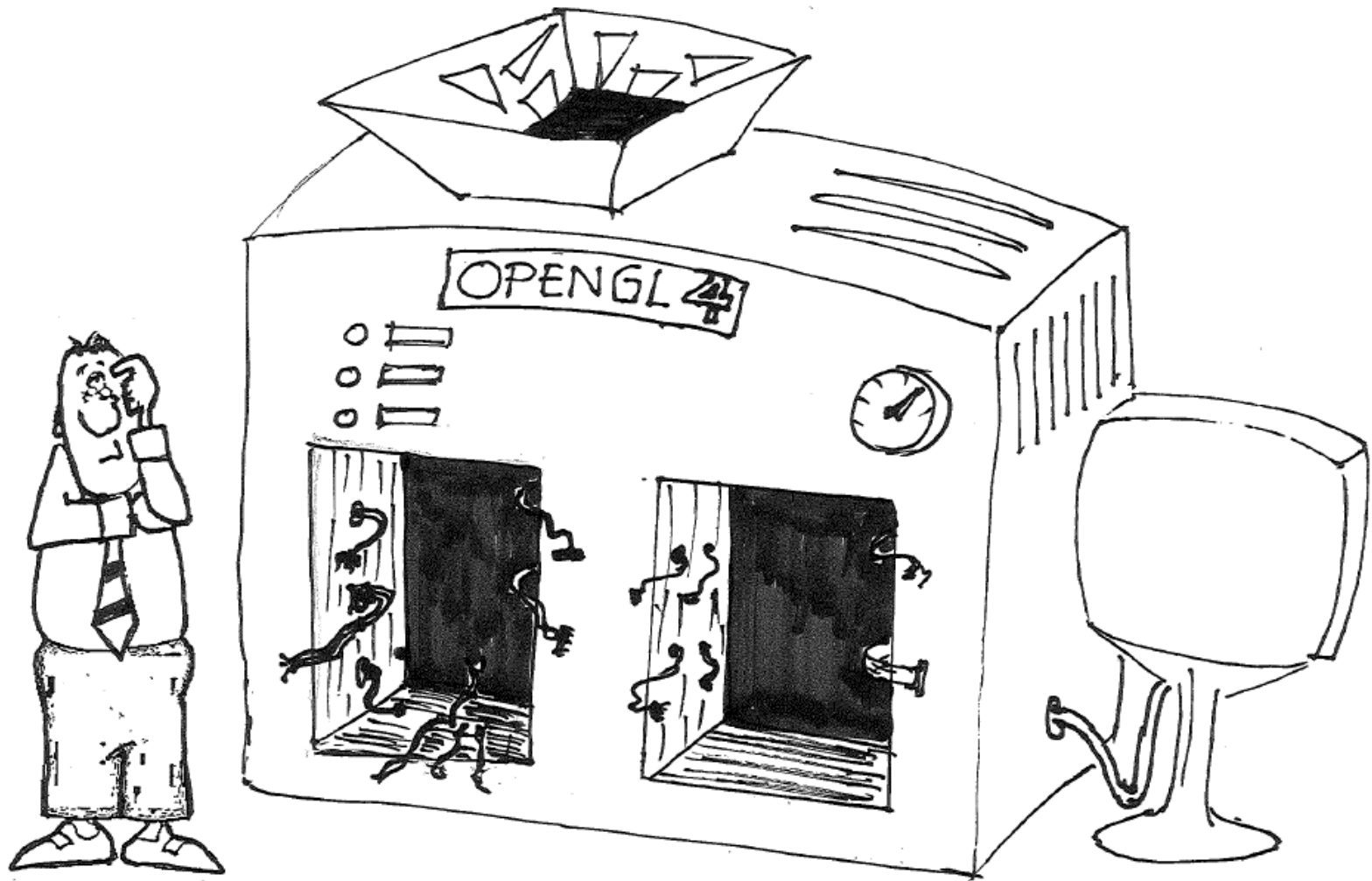


Modern OpenGL (OpenGL-4) Programmable Pipeline

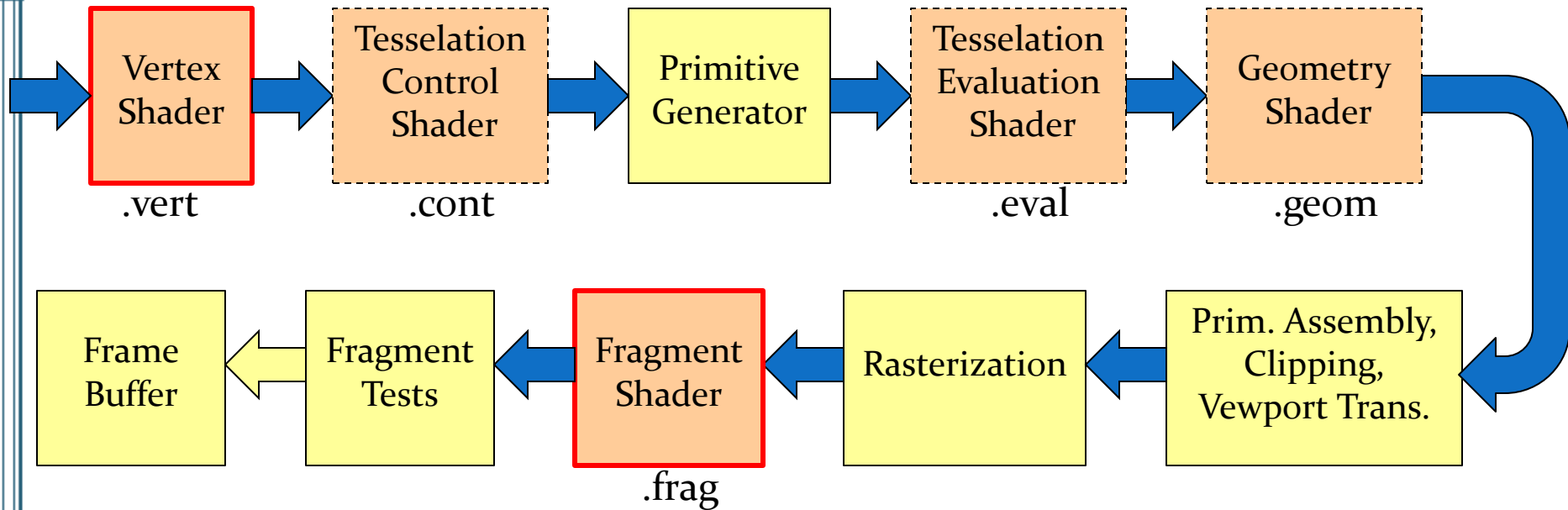
Graphics Processing Unit (GPU)



OpenGL 4 State Machine



OpenGL-4 Shader Stages



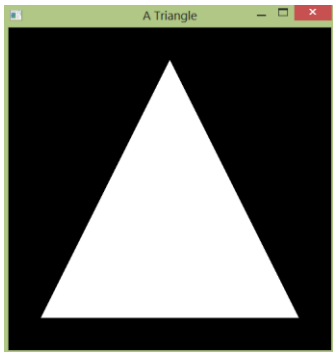
OpenGL Context, Version: Example

```
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);  
  
    const GLubyte *version = glGetString(GL_VERSION);  
    const GLubyte *renderer = glGetString(GL_RENDERER);  
    const GLubyte *vendor = glGetString(GL_VENDOR);  
  
    cout << "OpenGL version: " << version << endl;  
    cout << "OpenGL vendor: " << vendor << endl;  
    cout << "OpenGL renderer: " << renderer << endl;.
```

```
OpenGL version: 4.6.0 NVIDIA 391.35  
OpenGL vendor: NVIDIA Corporation  
OpenGL renderer: GeForce GTX 1050 Ti/PCIe/SSE2
```

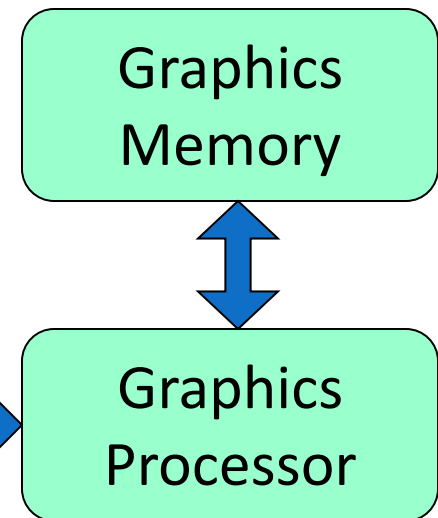
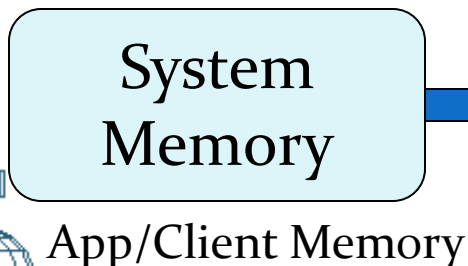
Primitive Drawing (OpenGL 1)

(Immediate Mode Rendering)



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

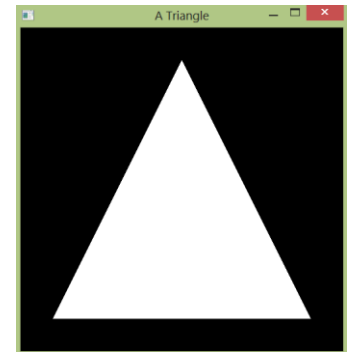
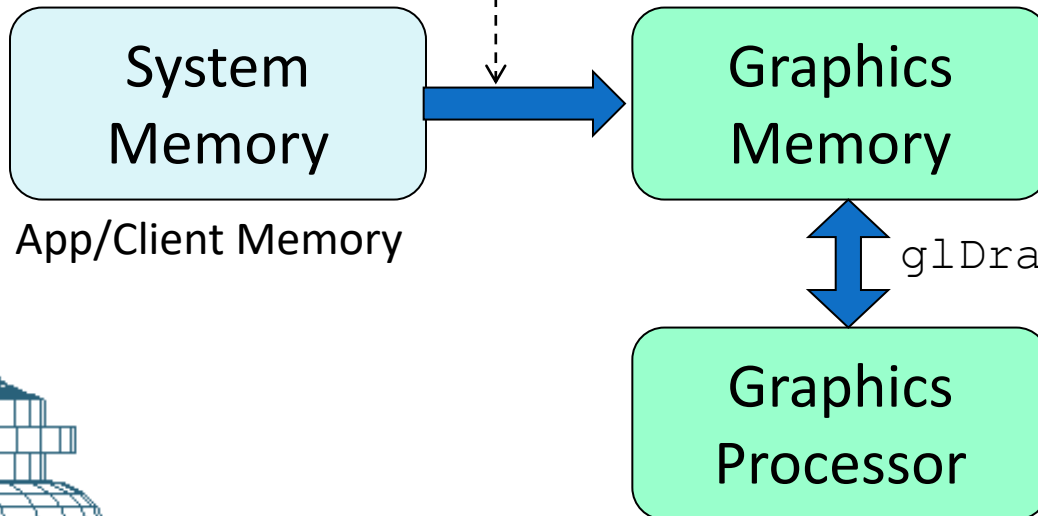
Deprecated!



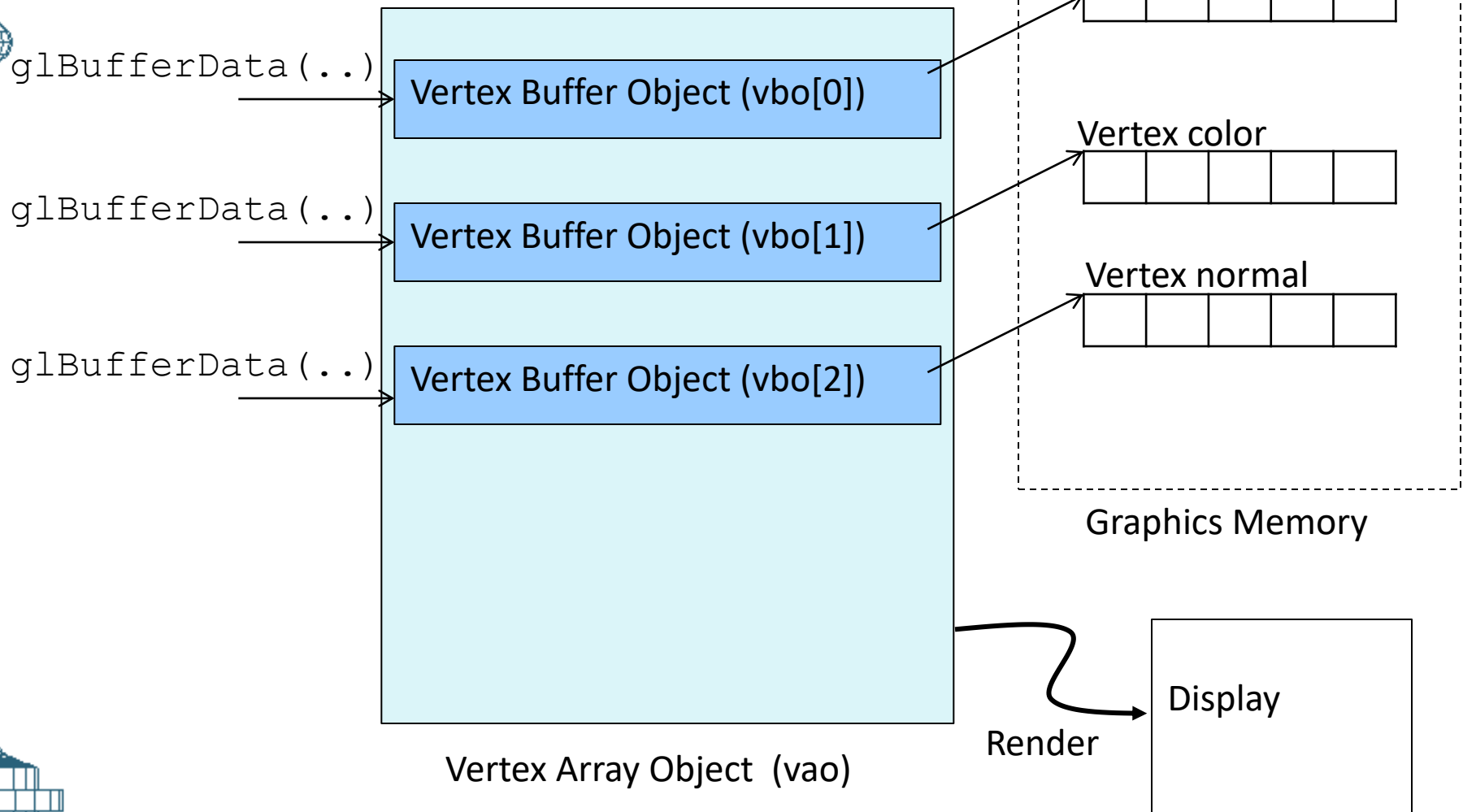
Primitive Drawing (OpenGL 4)

(Non-Immediate Mode Rendering)

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



Organising Data



Vertex Buffer Objects

Draw1.cpp

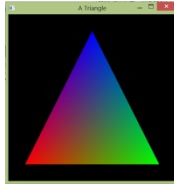


- A vertex buffer object (VBO) represents the data for a particular vertex attribute in video memory.
- Creating VBOs:
 1. Generate a new buffer object “vbo”
 2. Bind the buffer object to a target
 3. Copy vertex data to the buffer

```
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



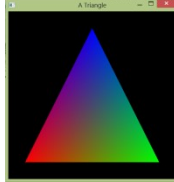
```
GLuint vbo[2];
glGenBuffers(2, vbo);    //Two VBOs

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

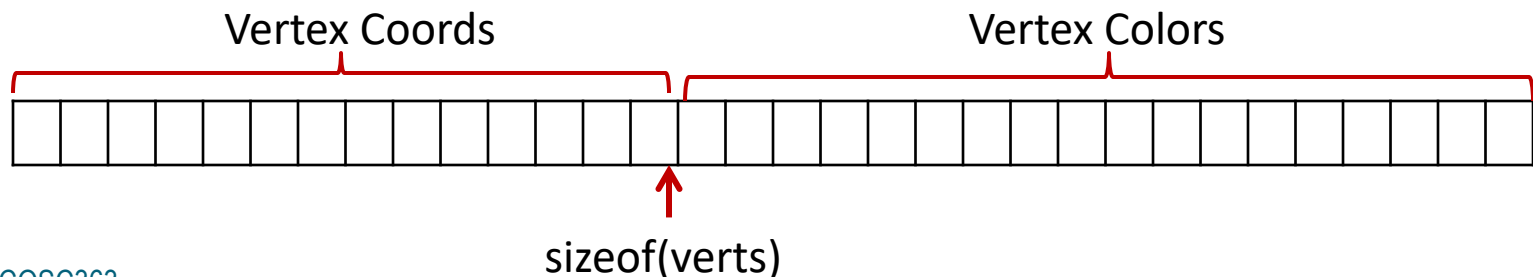
Draw3.cpp



```
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);
```

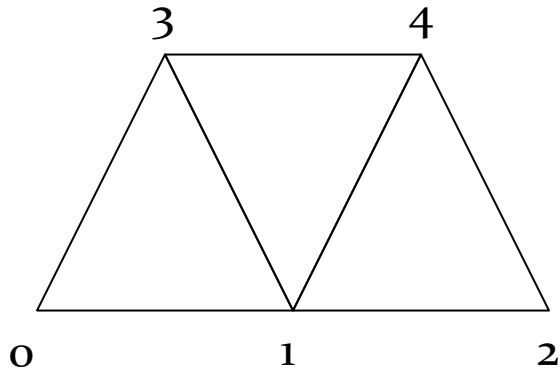
Primitive Type

Start index in the
enabled arrays

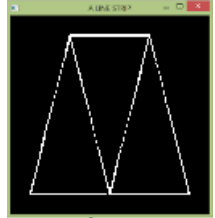
Count

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
 - glew (<http://glew.sourceforge.net>)
- Run the following programs:
 - Version.cpp
 - Draw1.cpp
 - Draw2.cpp
 - Draw3.cpp
 - Draw4.cpp

Uses shader code

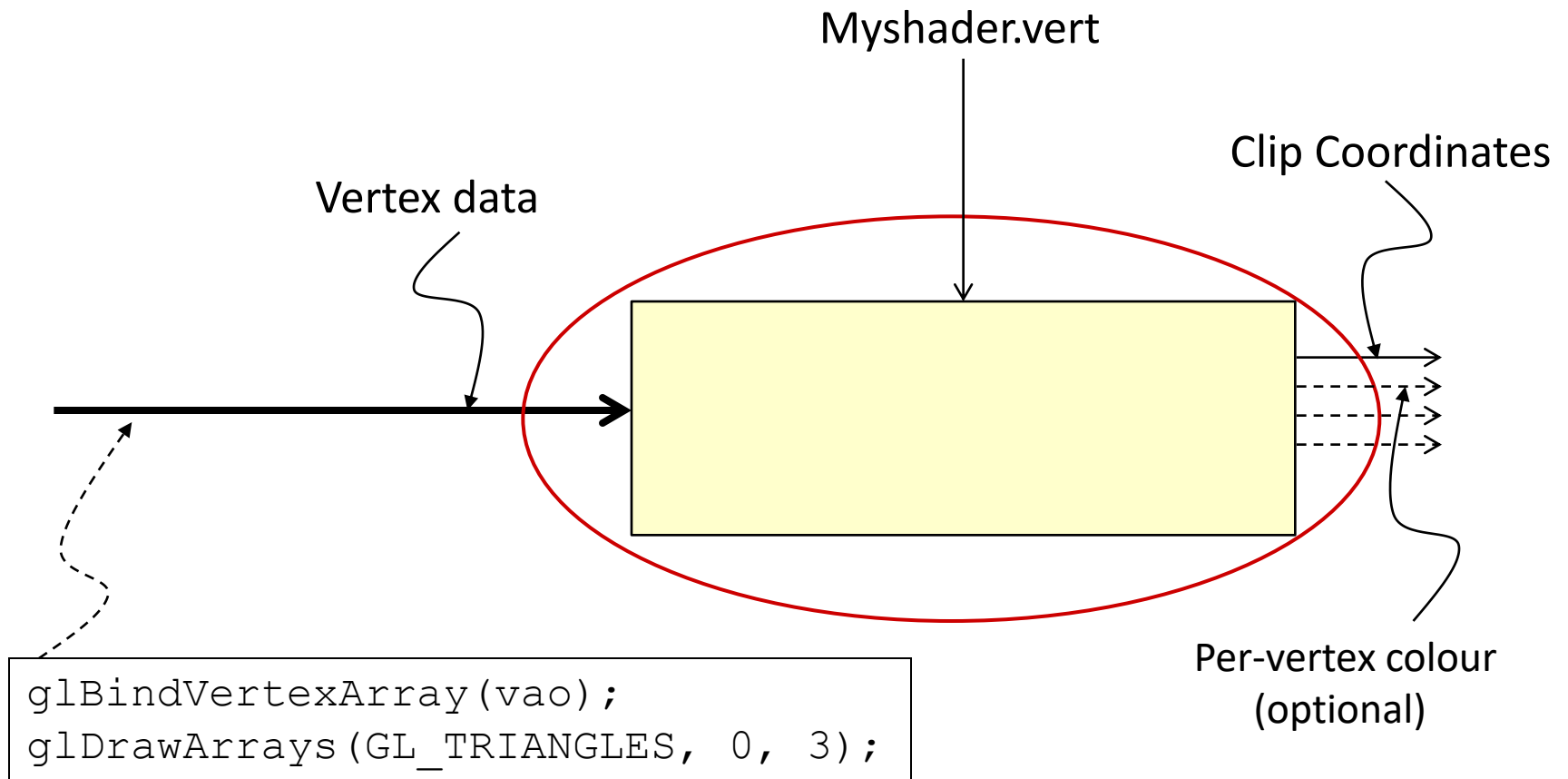
`Simple.vert, Simple.frag`
- Discuss any issues using class forum

Vertex Shader

- The vertex shader will execute once of every vertex.
- The position and any other attributes (normal, colour, texture coords etc) of the *current vertex*, if specified, will be available in the shader.
- Positions and attributes of other vertices are not available.
- A vertex shader *normally* outputs the clip coordinates of the current vertex, and also performs lighting calculations on the vertex.
- **gl_Position** is a built-in out variable for the vertex shader. A vertex shader *must* define its value.

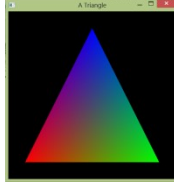
Programmable Pipeline

The Vertex Shader



Vertex Shader: Example

Draw2.cpp



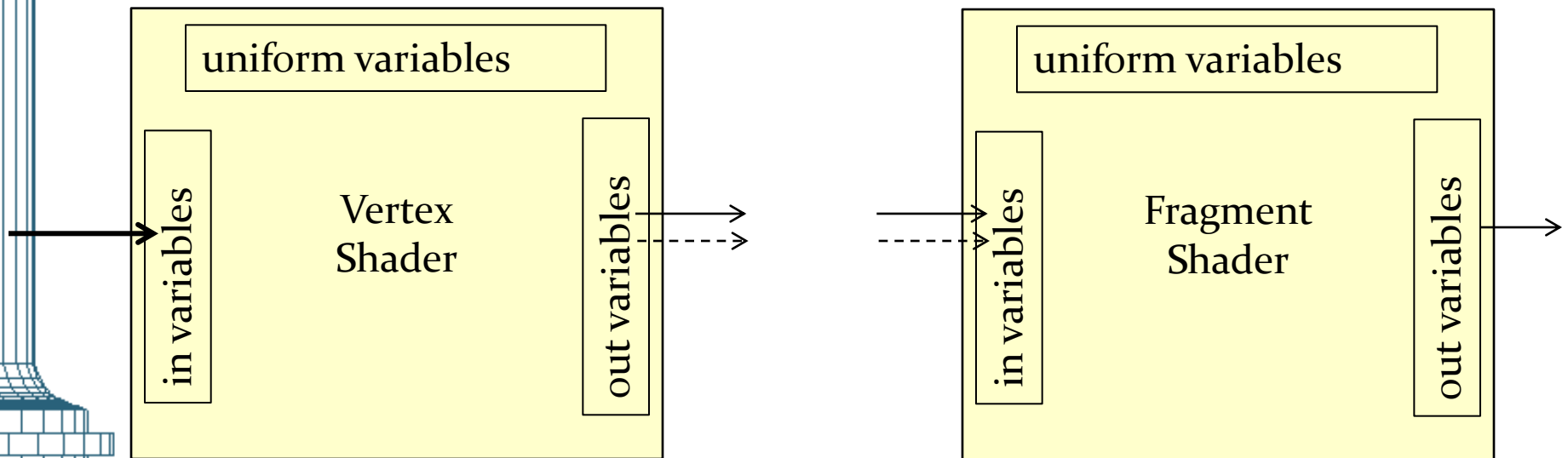
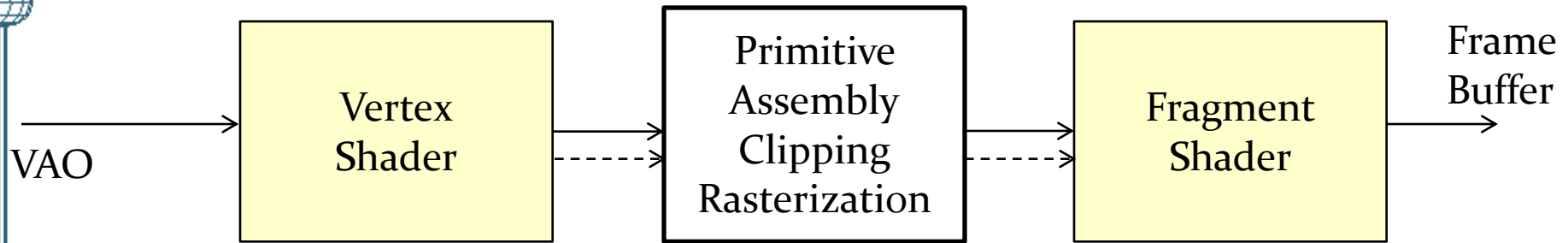
Application

```
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

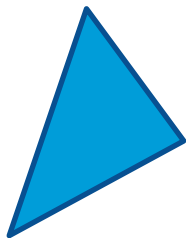
Vertex and Fragment Shaders



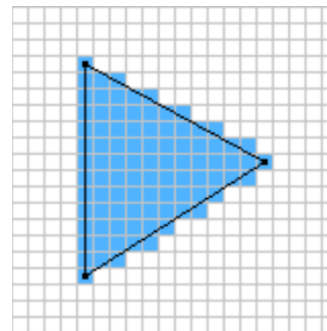
Fragments

- Rasterization is the process of scan-converting a primitive into a set of fragments.
- A fragment is a pixel-sized element that belongs to a primitive and could be potentially displayed as a pixel.
- The number of fragments generated for a primitive depends on the projected area of the primitive in the screen coordinate space.

Primitive
(in 3D space)



Fragments
(in 2D space)

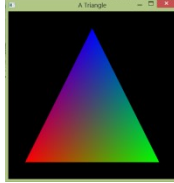


Fragment Shader

- A fragment shader is executed for each fragment generated by the rasterizer.
- A fragment shader outputs the colour of a fragment and optionally the depth value.
- Several colour computations (texture mapping, colour sum etc.), and depth offsets can be performed inside a fragment shader.
- A fragment shader can also discard a fragment.
- A fragment shader has the built-in in variable `gl_FragCoord` and built-in out variables **`gl_FragColor`** and `gl_FragDepth`

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

in vec4 theColor;

void main()
{
    gl_FragColor = theColor;
```


GLSL Aggregate Types

Vector Types: `vec2`, `vec3`, `vec4`

```
vec2 posn2D;  
vec3 grey, norm, color, view;  
vec4 posnA, posnB;  
float zcoord, d;  
posnA = vec4(-1, 2, 0.5, 1);  
posnB = vec4(posnA.yxx, 1); //Same as (2, -1, -1, 1)  
norm = normalize(vec3(1)); //(.33, .33, .33)  
view = vec3(1.6);          //(1.6, 1.6, 1.6)  
d = dot(norm, view);  
zcoord = posnA.z;          //0.5  
color = vec3(0.9, 0.2, 0.2);  
grey = vec3(0.2, color.gb); //(0.2, 0.2, 0.2)
```

Component Accessors: `(x, y, z, w)`, `(r, g, b, a)`
`(s, t, p, q)`

GLSL – Aggregate Types

Matrix Types: mat2, mat3, mat4

```
mat2 matA, matB, matC;
mat3 scale, identity;
float det;
vec2 v1, v2, v3, v4;
v1 = vec2(-6, 4);
v2 = vec2(3);
matA = mat2(3, 0, -2, 5);           //1st Column = (3, 0)
matB = mat2(v1, v2);                //v1, v2 column vectors
matC = matA * transpose(matB);      //Product matrix
v3 = matC[1];                       //Second column of matC
v4 = matA * v3;
identity = mat3(1.0);
scale = mat3(3.0);                  //3.0 along diagonal
det = determinant(matC);
matC = inverse(matC);
```

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)

Model-View-Projection Matrix

Old Version

glFrustum(...)
gluPerspective(...)
glOrtho(...)

gluLookAt(...)

glTranslatef(...)
glRotatef(...)
glScalef(...)

$$\begin{bmatrix} x_c \\ y_c \\ z_c \\ w_c \end{bmatrix} = \begin{bmatrix} \text{Projection} \\ \text{Matrix} \end{bmatrix} \begin{bmatrix} \text{View} \\ \text{Matrix} \end{bmatrix} \begin{bmatrix} \text{Transformation} \\ \text{Matrix} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Output

Vertex Position in
Clip Coordinates

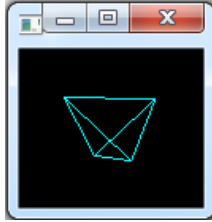
Input

Vertex Position in
World Coordinates

Defining Transformations

Application

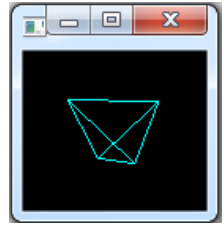
Draw5.cpp



```
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



Draw5.cpp

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

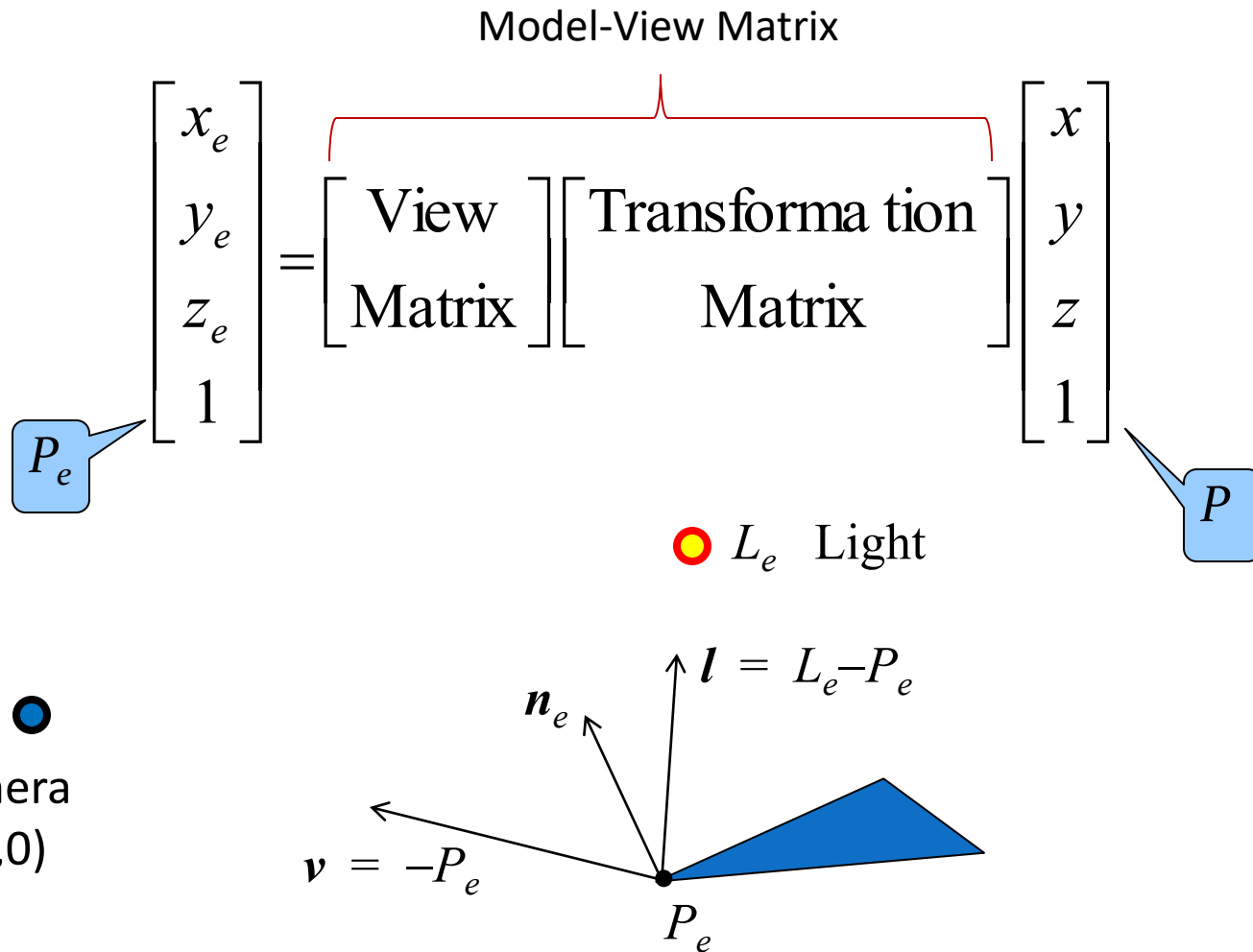
Fragment Shader

Tetrahedron.frag

```
void main()
{
    gl_FragColor = vec4(0.0, 1.0, 1.0, 1.0);
}
```

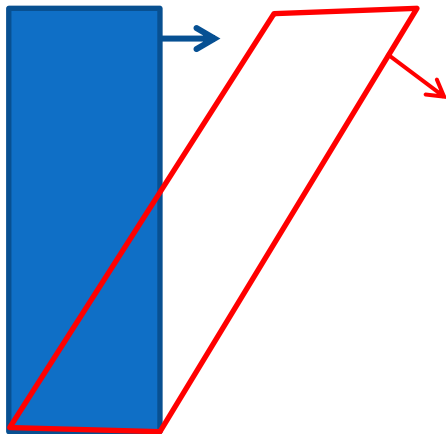
Lighting Calculations

Lighting calculations are usually performed in **eye-coordinate** space.



Transformation of Normal Vector

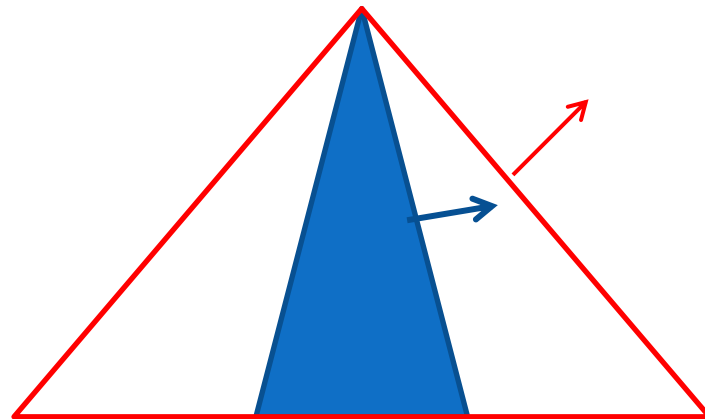
When primitives (or objects) are transformed by a matrix M , their surface normal vectors undergo a transformation by a matrix M^{-T} .



Shear transformation

$$x' = x + ky$$

$$y' = y$$



Scale transformation

$$x' = kx$$

$$y' = y$$

Transformation of Normal Vector

Consider a vector $V = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}$, and its normal vector $N = \begin{bmatrix} n_x \\ n_y \\ n_z \end{bmatrix}$

The vectors are perpendicular: $v_x n_x + v_y n_y + v_z n_z = 0$.

In matrix notation,

$$\begin{bmatrix} v_x & v_y & v_z \end{bmatrix} \begin{bmatrix} n_x \\ n_y \\ n_z \end{bmatrix} = 0 \qquad V^T N = 0.$$

Let V be transformed using matrix A , and the normal using matrix B . After the transformation, the vectors will remain perpendicular only if $(AV)^T (BN) = 0$.

Transformation of Normal Vector

The previous equation gives $V^T A^T B N = 0$

$$V^T (A^T B) N = 0.$$

But, $V^T N = 0$.

Therefore, $A^T B = I$ (identity matrix).

$$\text{Hence, } B = (A^T)^{-1}$$

The transformation applied to the normal is the ***inverse-transpose*** of the transformation applied to the vectors (or points).

For lighting calculations, we need to multiply the normal vectors by the inverse-transpose of the model-view matrix.



- 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: $L_e = VML$
 - 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 using the Phong-Blinn model.

Vertex shader:

Torus.vert

```
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;
```

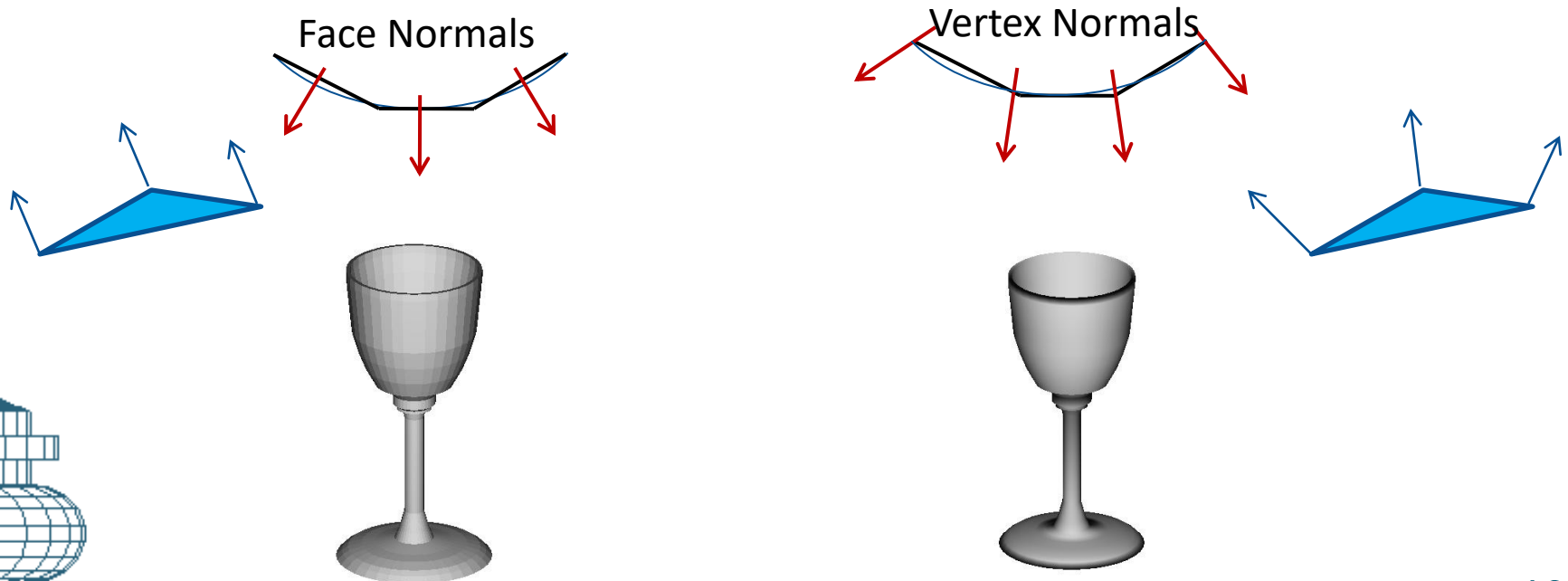


Fragment shader:
Torus.frag

```
in vec4 theColour;  
  
void main()  
{  
    gl_FragColor = theColour;  
}
```

Face Normals vs Vertex Normals

- Face normals: Each triangle or quad of a mesh model has a single normal vector representing the orientation of that face.
- Vertex normals: A planar element can be made to look curved, by assigning different normal vectors at the vertices that represent an underlying curved shape of the surface.



Modelling Using Vertex Normals

Torus.cpp

```
int nverts  = nsides * nrings;
nelms = nsides * nrings * 6;

float *verts = new float[nverts * 3];
float *normals = new float[nverts * 3];
unsigned int *elems = new unsigned int[nelms];

...

glGenBuffers(3, vboID);

glBindBuffer(GL_ARRAY_BUFFER, vboID[0]);
glBufferData(GL_ARRAY_BUFFER, indx * sizeof(float), verts,
                                                     GL_STATIC_DRAW);

glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, NULL);

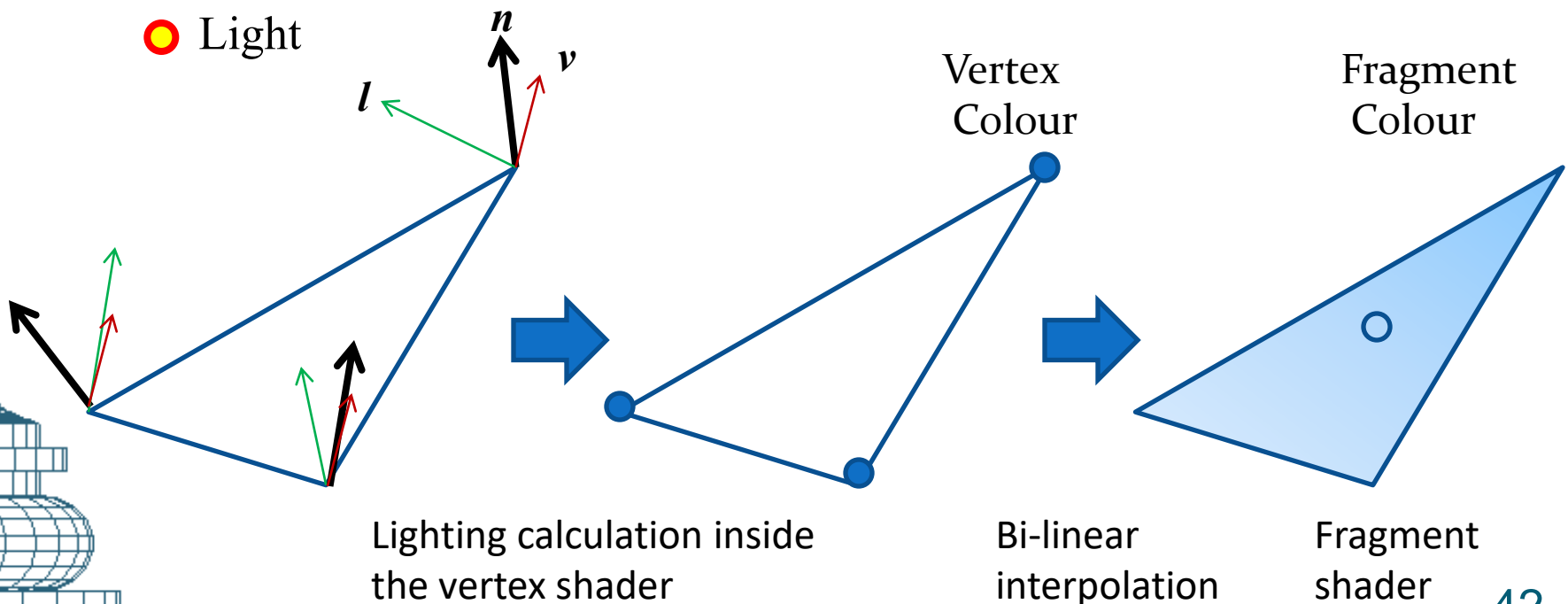
glBindBuffer(GL_ARRAY_BUFFER, vboID[1]);
glBufferData(GL_ARRAY_BUFFER, indx * sizeof(float), normals,
                                                     GL_STATIC_DRAW);

glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, NULL);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vboID[2]);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, ielndx * sizeof(unsigned int),
                                                     elems, GL_STATIC_DRAW);
```

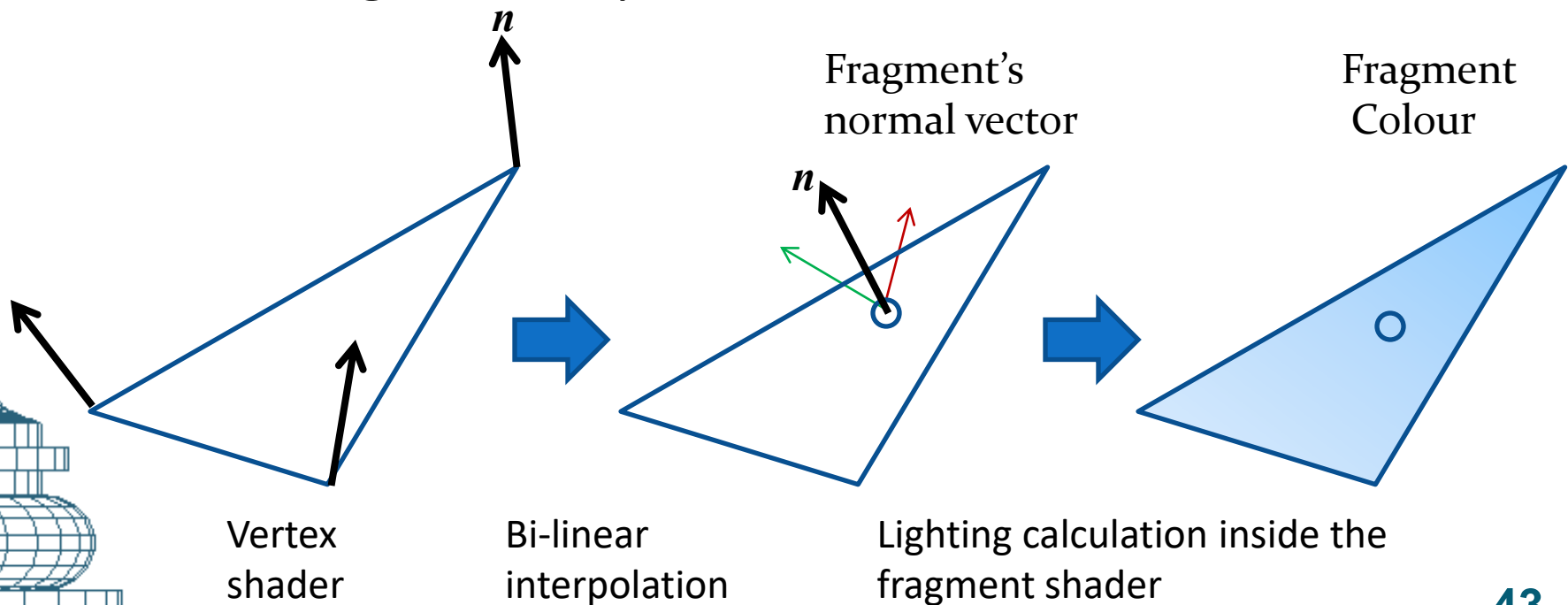
Per-Vertex (Traditional) Lighting

- The lighting calculations shown on slides 40-42 are performed for each vertex and the interpolated colour values are used in the fragment shader.
- The traditional lighting model of the fixed-function pipeline is also implemented in this way.



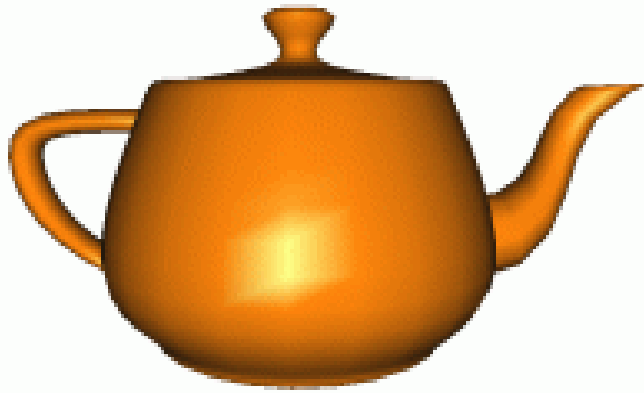
Per-Fragment Lighting (Phong Shading)

- The vertex shader outputs the normal vector to the fragment shader.
- The fragment shader receives an interpolated normal vector for each fragment.
- The lighting calculation is performed inside the fragment shader using the interpolated normal vector.

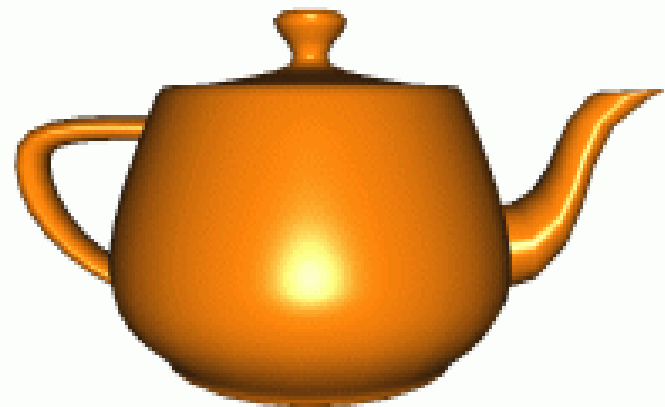


Per-Fragment Lighting

- A lighting computation implemented inside the fragment shader produces a far more accurate rendering of reflections from the surface than the traditional model.
- Per-fragment lighting is computationally very expensive compared to per-vertex lighting.



Per-Vertex Lighting



Per-Fragment Lighting

Texturing

- Select an active texture unit
- Load texture image
- Set texture parameters
- Create a `Sampler2D` variable in the fragment shader. Assign this uniform variable the index of the texture unit.

Application:

```
glGenTextures(1, &texID);  
glActiveTexture(GL_TEXTURE0);  
glBindTexture(GL_TEXTURE_2D, texID);  
loadTGA("myImage.tga");  
  
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);  
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);  
...  
GLuint texLoc = glGetUniformLocation(program, "txSampler");  
glUniform1i(texLoc, 0);
```

Texturing

Texture coordinates are stored in a vertex buffer object:

```
glBindBuffer(GL_ARRAY_BUFFER, vboID[0]);
glBufferData(GL_ARRAY_BUFFER, (indx) * sizeof(float), verts,
                                                     GL_STATIC_DRAW);

glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, NULL);

glBindBuffer(GL_ARRAY_BUFFER, vboID[1]);
glBufferData(GL_ARRAY_BUFFER, (indx) * sizeof(float), normals,
                                                     GL_STATIC_DRAW);

glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, NULL);

glBindBuffer(GL_ARRAY_BUFFER, vboID[2]);
glBufferData(GL_ARRAY_BUFFER, (indx) * sizeof(float), texCoords,
                                                     GL_STATIC_DRAW);

glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 0, NULL);
glEnableVertexAttribArray(2); // texture coords

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vboID[3]);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, ...
```

Texturing

The vertex shader passes the texture coords of each vertex to the fragment shader.

Vertex shader:

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

...

out vec4 diffRefl;
out vec2 TexCoord;

void main()
{
    gl_Position = mvpMatrix * vec4(position, 1.0);
    ... //lighting calculations
    diffRefl = ...
    TexCoord = texCoord;
}
```

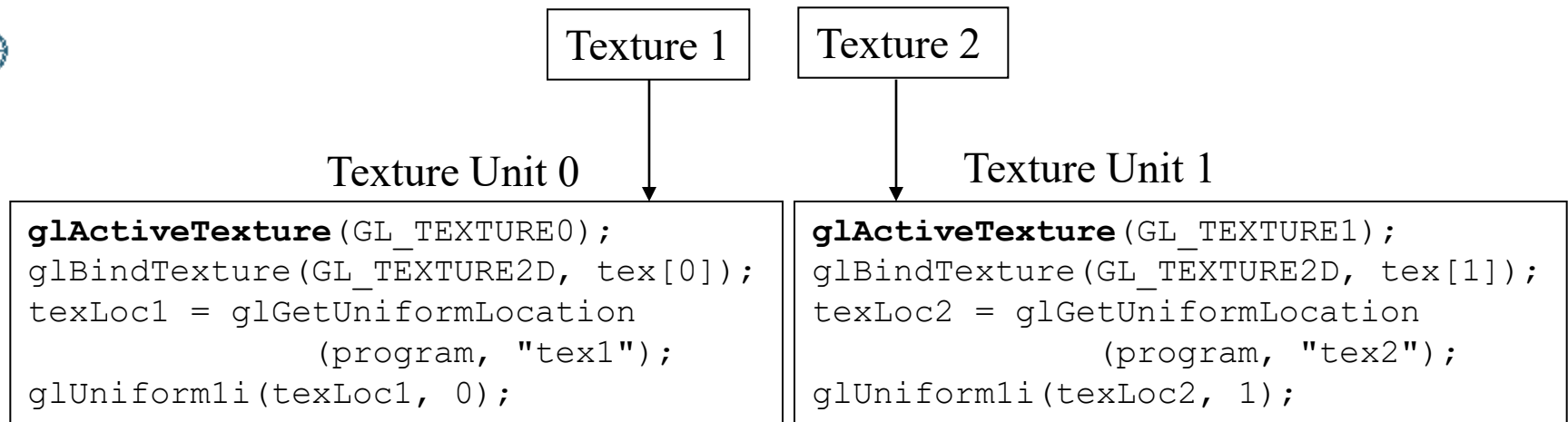
Texturing

The fragment shader receives the interpolated texture coordinates for each fragment, and uses a `Sampler2D` object to retrieve the colour values from texture memory.

Fragment shader:

```
uniform sampler2D txSampler;  
  
in vec4 diffRefl;  
in vec2 TexCoord;  
  
void main()  
{  
    vec4 tColor = texture(txSampler, TexCoord);  
    gl_FragColor = diffRefl * tColor;  
}
```


Multi-Texturing



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

Fragment Shader:

```
uniform sampler2D tex1;
uniform sampler2D tex2;

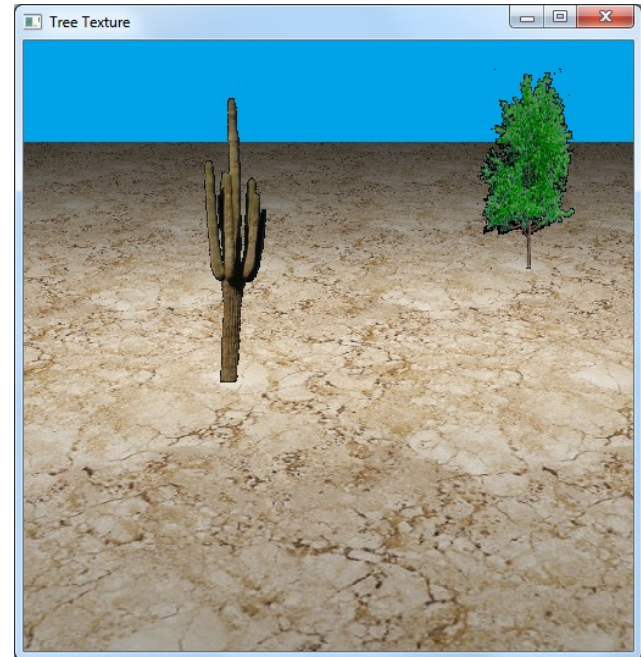
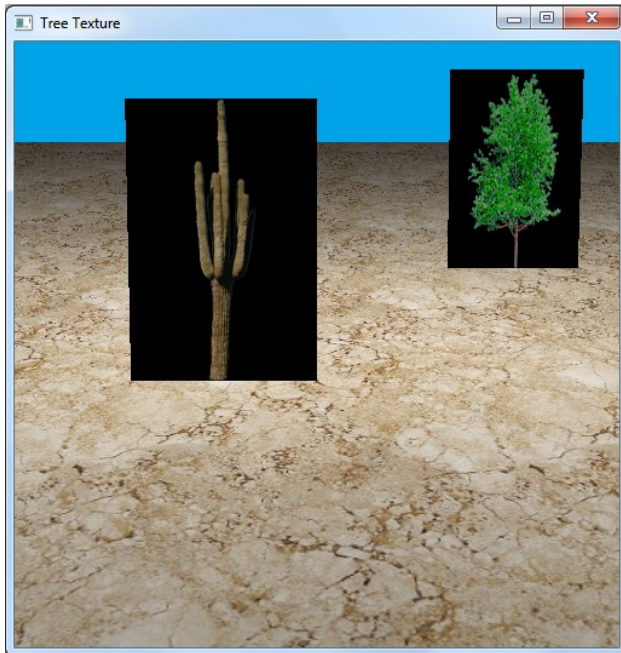
in vec4 diffRefl;
in vec2 TexCoord;

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

    gl_FragColor = 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

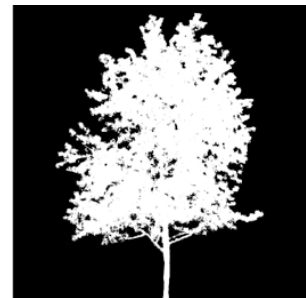
Use the alpha channel of an image (if available) to transfer only those pixels on the object.

Fragment Shader

```
uniform sampler2D texTree;  
  
in vec2 TexCoord;  
  
void main()  
{  
    vec4 treeColor = texture(texTree, TexCoord);  
    if(treeColor.a == 0) discard;  
    gl_FragColor = treeColor;  
}
```



RGB



Alpha