VICTORIA UNIVERSITY OF WELLINGTON
TE HERENGA WAKA

Lecture 5:
Shading and colour

CGRA 354 : Computer Graphics Programming

Instructor: Dr Alex Doronin
Cotton Level 3, Office 330
alex.doronin@vuw.ac.nz

With slides from: Steve Marschner, Cornell; Taehyun Rhee, CMIC; Zohar Levi, Mike Bailey, OSU, Ed Angel University of New Mexico

Next

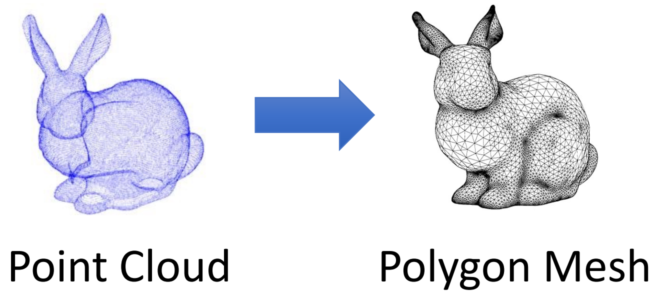
- **C++/OpenGL programming continued:**
  - **3D Geometry and GUI**
  - **Shading and color**
  - ***Introduction to Lighting***

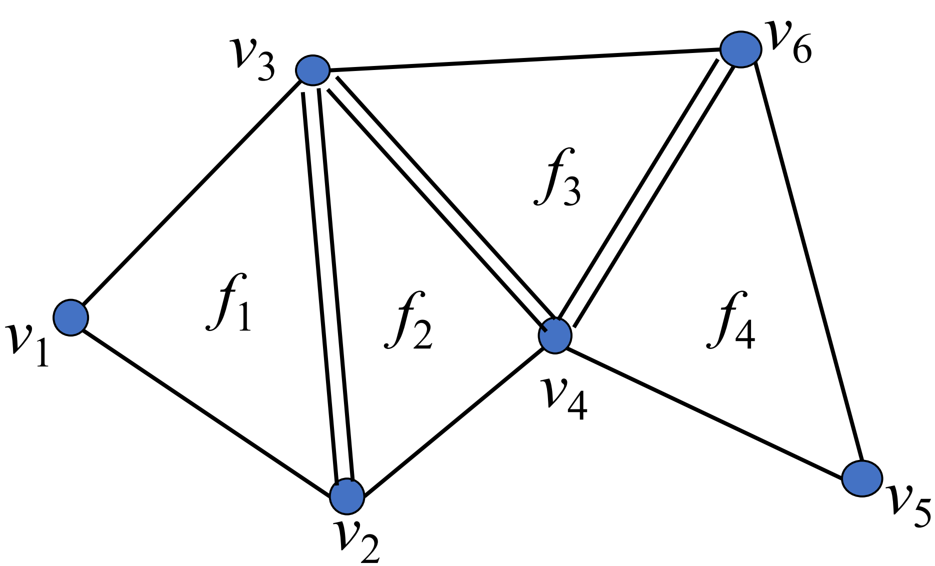
Office hours

- Thursdays **from 11am -1pm** at CO330
- Additional hours by **drop in** or **appointment**

Recap: Geometry and Polygon Mesh

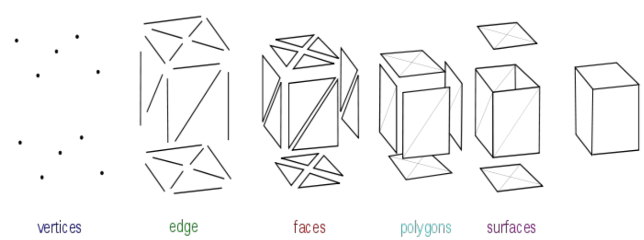
- Face list
  - Lists of coordinates
- Polygons are unrelated





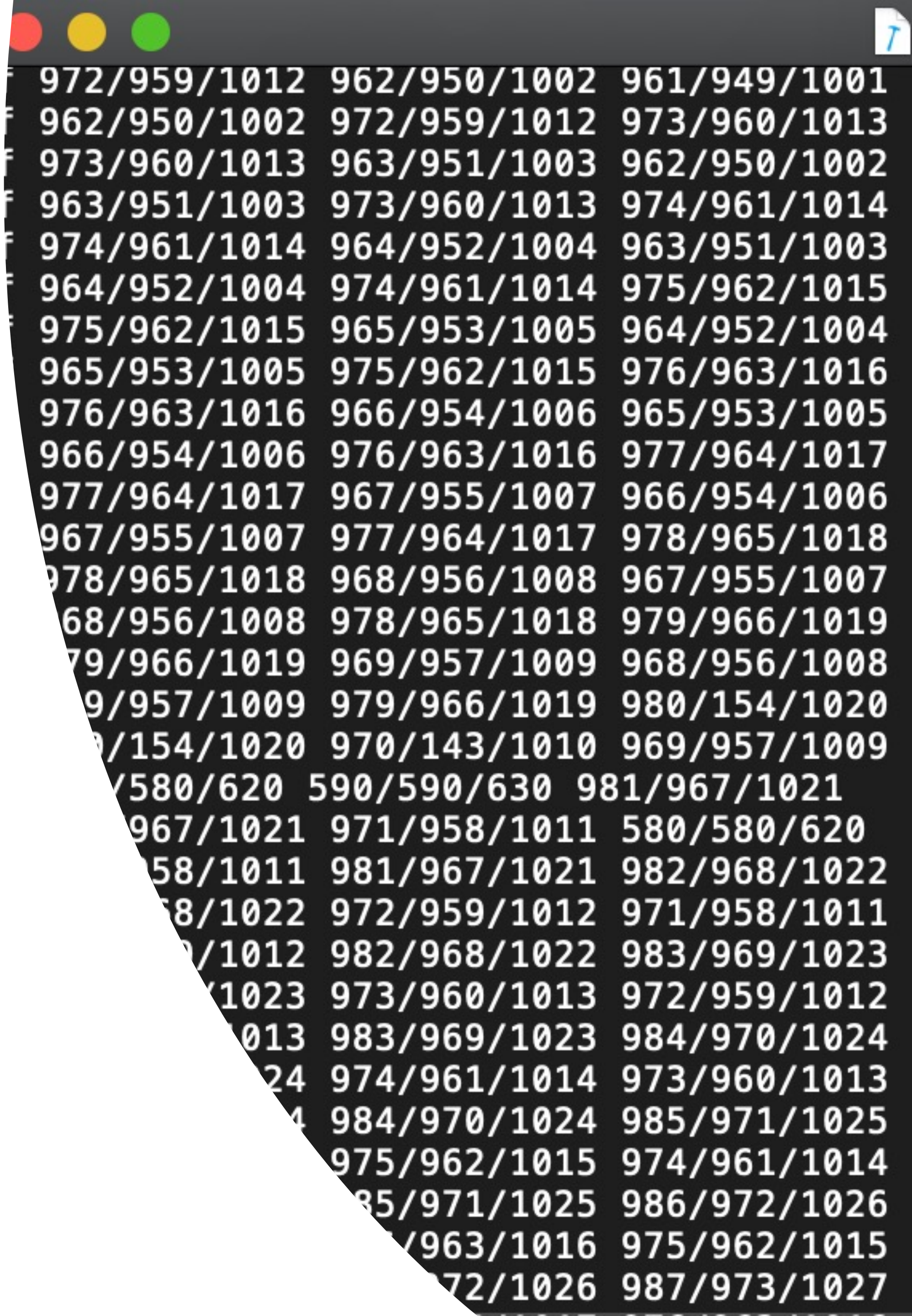
face	vertices (ccw)
f_1	(v_1, v_2, v_3)
f_2	(v_2, v_4, v_3)
...	...

- What are the nearest neighbor of a vertex ?
- What are the adjacent triangles of a vertex ?



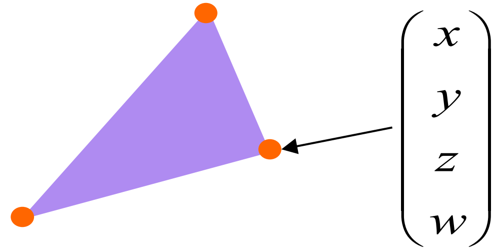
Recap: Wavefront obj file

- f (face)
  - f v1 v2 v3 …
  - f v1/vt1 v2/vt2 v3/vt3 …
  - f v1/vt1/vn1 v2/vt2/vn2 v3/vt3/vn3 …
  - f v1//vn1 v2//vn2 v3//vn3 …
- Group
  - o [object name]
  - g [group name]
- Materials
  - mtlib [external .mtl file name]



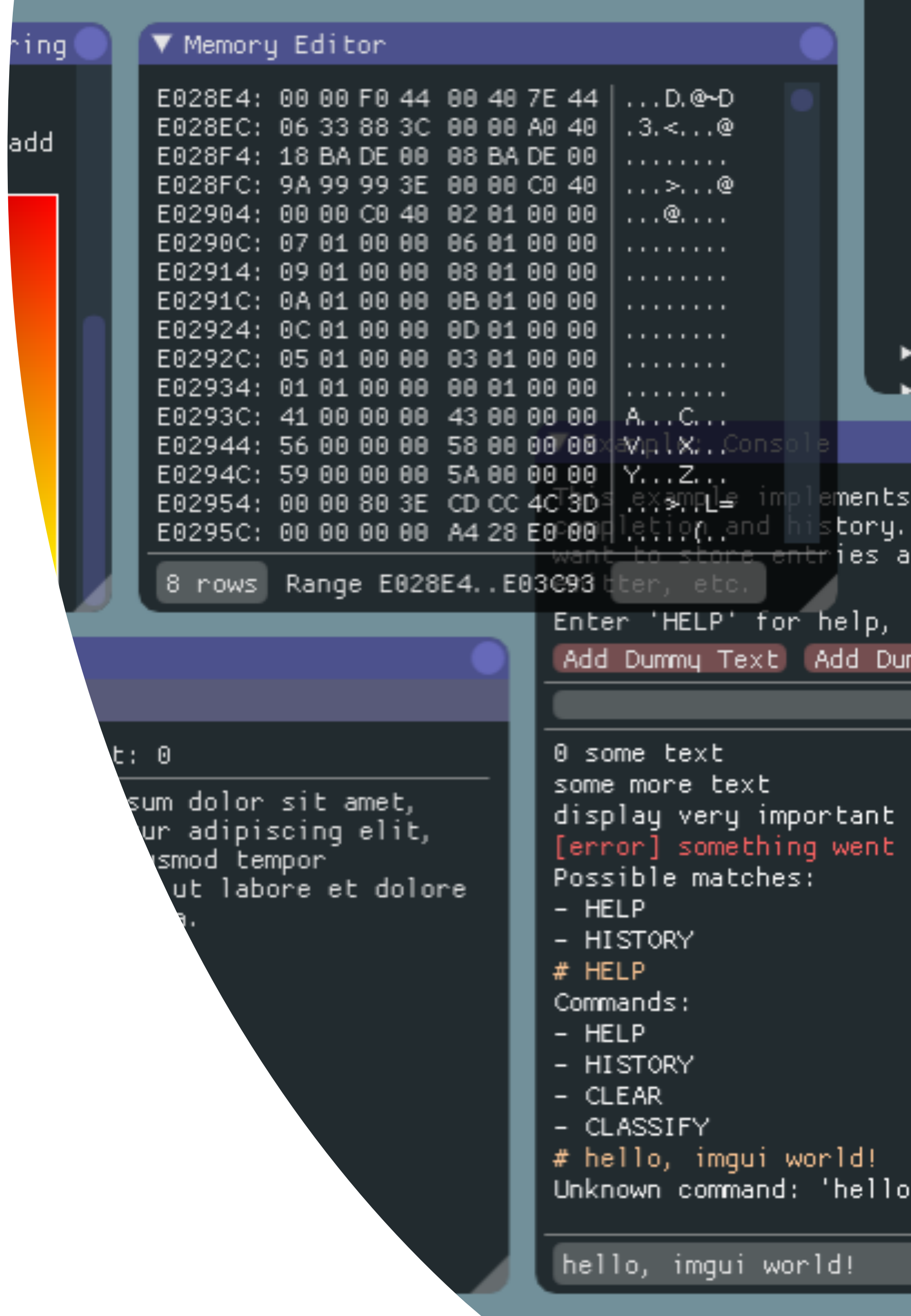
Recap: Representing Geometric Objects

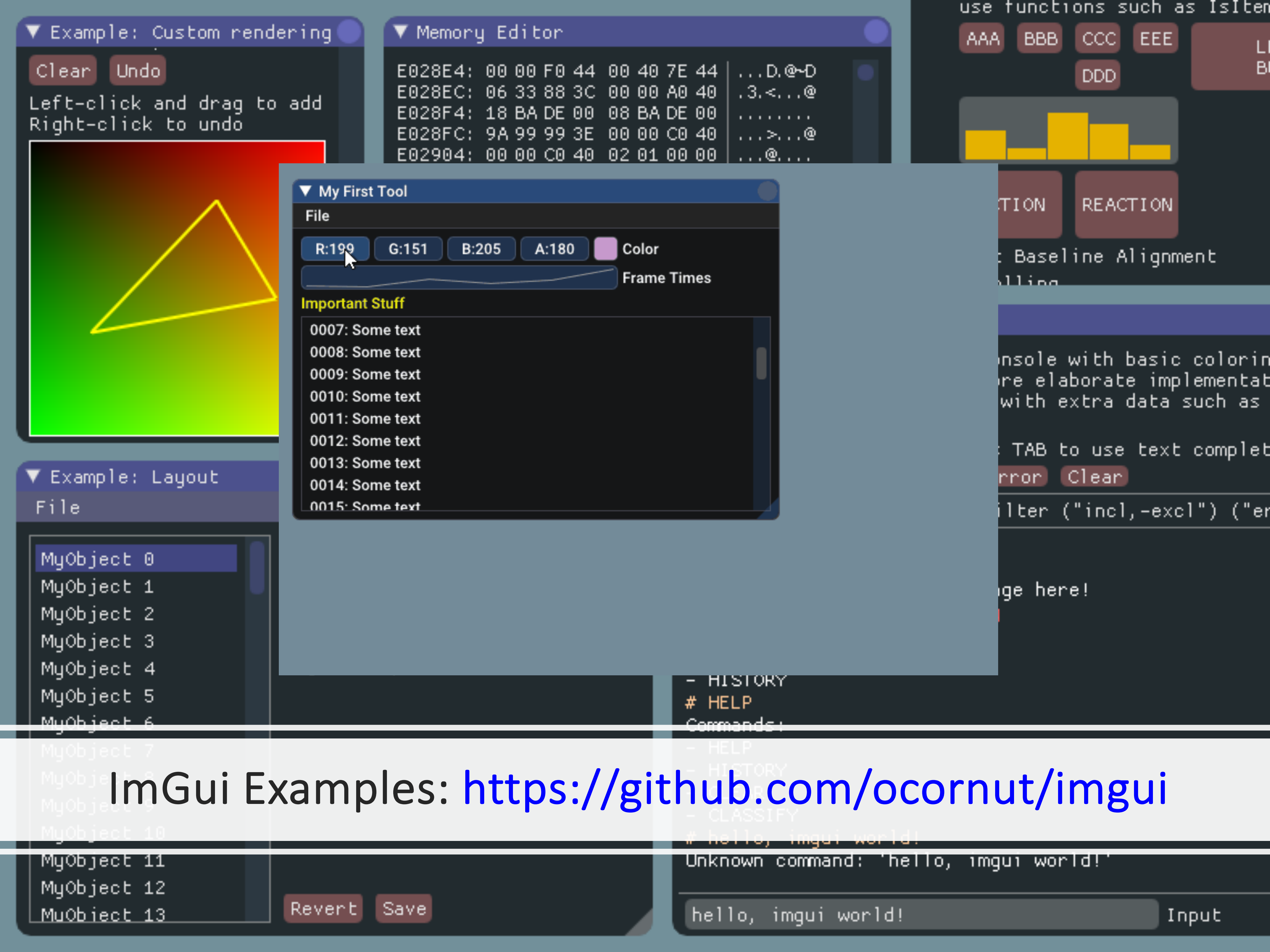
- Geometric objects are represented using *vertices*
- A vertex is a collection of generic attributes
  - positional coordinates
  - colors
  - texture coordinates
  - any other data associated with that point in space
- Position stored in 4 dimensional homogeneous coordinates
- Vertex data must be stored in vertex buffer objects (VBOs)
- VBOs must be stored in vertex array objects (VAOs)



if w == 1, then the vector (x,y,z,1) is a position in space.
If w == 0, then the vector (x,y,z,0) is a direction.

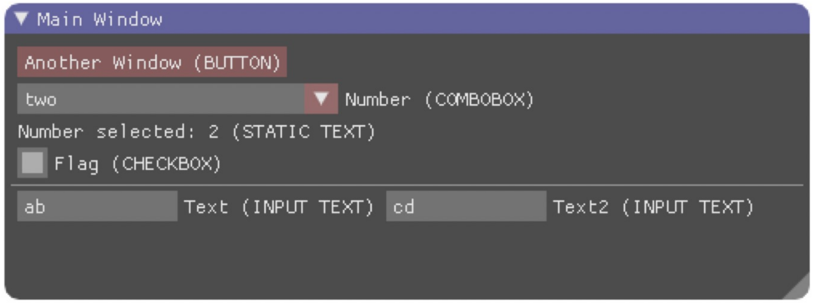
Recap: ImGUI

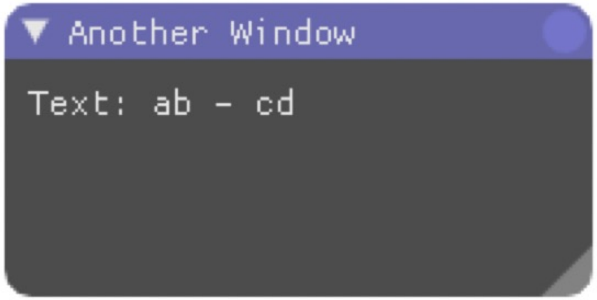

Memory Editor
E028E4: 00 00 F0 44 00 40 7E 44 ...D.@~D
E028EC: 06 33 88 3C 00 00 A0 40 .3.<...@
E028F4: 18 BA DE 00 08 BA DE 00 ....
E028FC: 9A 99 99 3E 00 00 C0 40 ...>...@
E02904: 00 00 C0 40 02 01 00 00 ...@....
E0290C: 07 01 00 00 06 01 00 00 ....
E02914: 09 01 00 00 08 01 00 00 ....
E0291C: 0A 01 00 00 0B 01 00 00 ....
E02924: 0C 01 00 00 0D 01 00 00 ....
E0292C: 05 01 00 00 03 01 00 00 ....
E02934: 01 01 00 00 00 01 00 00 ....
E0293C: 41 00 00 00 43 00 00 00 A...C...
E02944: 56 00 00 00 58 00 00 00 V...X...
E0294C: 59 00 00 00 5A 00 00 00 Y...Z...
8 rows
Range E028E4..E03C93
Add Dummy Text
0 some text
some more text
[error] something went
Possible matches:
- HELP
- HISTORY
HELP
Commands:
- HELP
- HISTORY
- CLEAR
- CLASSIFY
hello, imgui world!
hello, imgui world!



ImGui Examples: https://github.com/ocornut/imgui

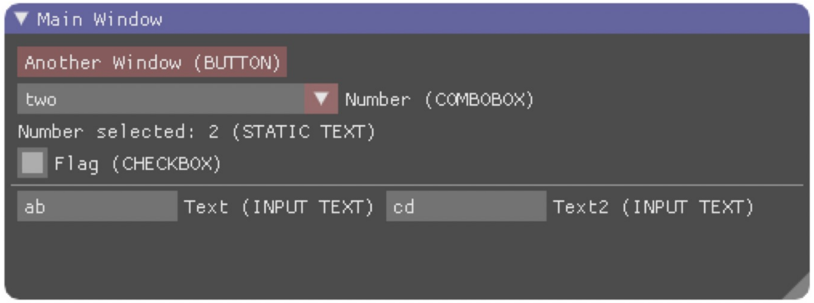
Hello, ImGui: Advanced

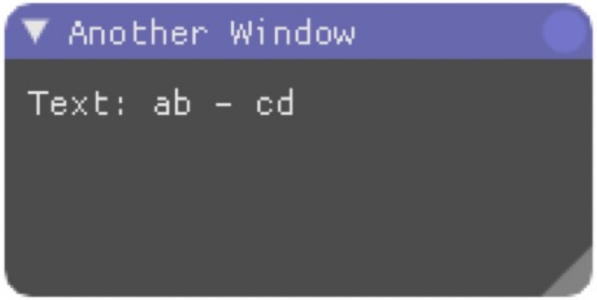





```
void main_loop()
{
   static bool show_another_window = false;
   ImGui::Begin("Main Window");

   ImGui::PushItemWidth(200); // Set width for next widgets
   if ( ImGui::Button("Another Window (BUTTON)") )
      show_another_window ^= 1;

   static int item = -1;
   const char *items[] = {"zero", "one", "two", "three"};
   ImGui::Combo("Number (COMBOBOX)", &item, items, 4, 5);
   ImGui::Text("Number selected: %d (STATIC TEXT)", item);
   static bool flag = false;
   ImGui::Checkbox("Flag (CHECKBOX)", &flag);

   ImGui::Separator();
```

Hello, ImGui: Advanced

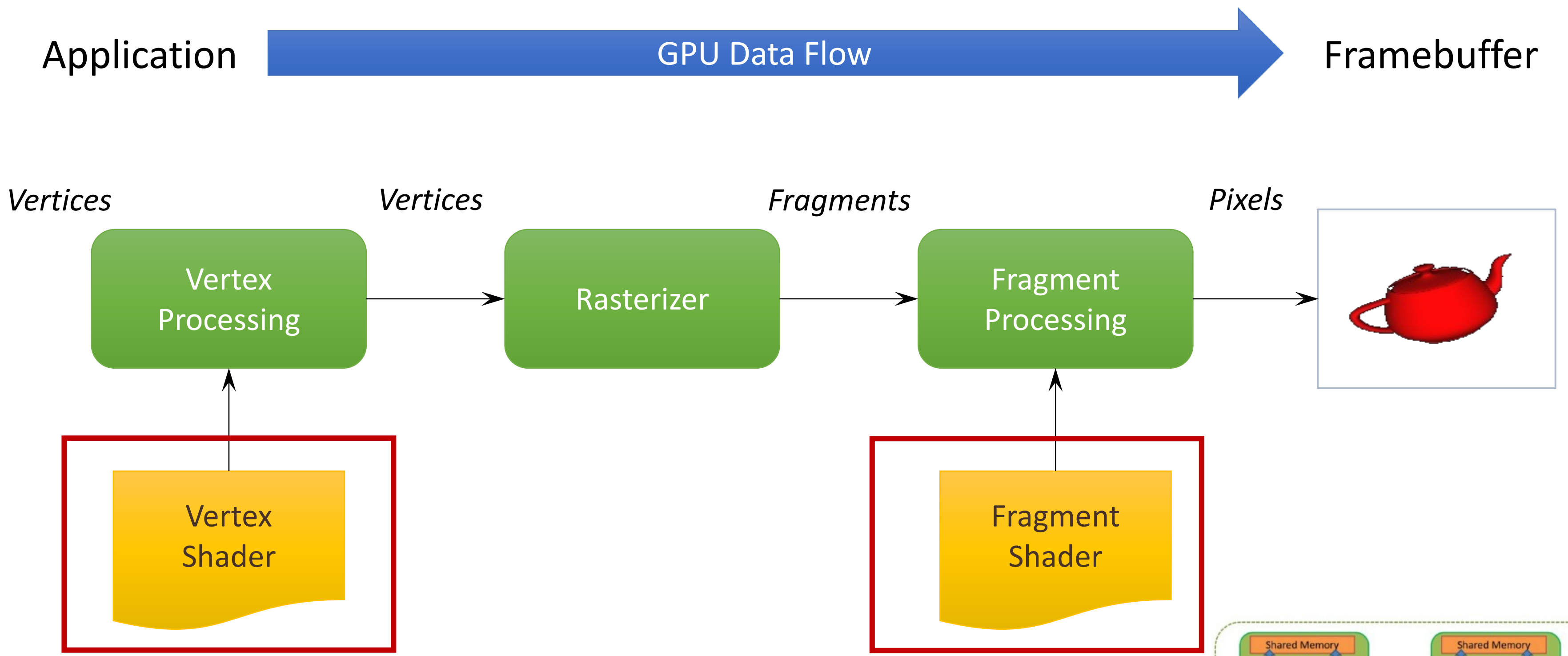




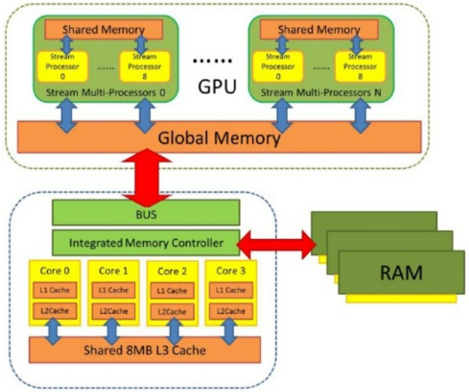

```
    static std::string text, text2;
    text.resize(50);
    text2.resize(50);
    ImGui::PushItemWidth(100); // Set width for next widgets
    ImGui::InputText("Text (INPUT TEXT)", &text[0], text.size());
    ImGui::SameLine();
    ImGui::InputText("Text2 (INPUT TEXT)", &text2[0], text2.size());
    ImGui::End();

    if ( show_another_window ) {
       ImGui::SetNextWindowSize(ImVec2(200,100), ImGuiSetCond_FirstUseEver);
       ImGui::Begin("Another Window", &show_another_window);
       ImGui::Text("Text: %s - %s", text.c_str(), text2.c_str());
       ImGui::End();
    }
}
```


RTX 2080 Ti
ACE
Graphics Command Processor
Global Data Share
Multimedia Accelerators
VCE
UVD
TrueAudio DSPs
CrossFire XDMA
Eyefinity Display Controllers
DMA Engines
PCIe 3.0 Bus Interface
Shader Engine
Geometry Processor
Rasterizer
RB
CU
L2 Cache
MC
Memory Controller
RT CORE
SM
L2 Cache
AMD
R

Recap: Graphics pipeline



- Modern OpenGL programs essentially do the following steps:
  - Create shader programs
  - Create buffer objects and load data into them
  - “Connect” data locations with shader variables
  - Render



Host (CPU) and Device (GPU) crosstalk

Vertex array objects (VAOs): Does not store any vertex data, stores references to the arrays/buffers (VBOs)

Vertex buffer objects (VBOs): Contains vertex data, etc. Configured through OpenGL calls (offsets, data types, interleaving, etc.). The order of binding VBOs and VAOs “is” important, VBOs binding changes the bound VAO.

Index buffer object (IBO): VBO Indexing, for the reuse the same vertex over again.

```
// generate buffers
glGenVertexArrays(1, &m_vao); // VAO stores information about how the buffers are set up
glGenBuffers(1, &m_vbo_pos); // VBO stores the vertex data
glGenBuffers(1, &m_ibo); // IBO stores the indices that make up primitives
```

Vertex array objects (VAOs):

Generate a buffer ID and bind:

```
// generate buffers
glGenVertexArrays(1, &m_vao); // VAO stores information about how the buffers are set up
// VAO
glBindVertexArray(m_vao);
```

Ways to think: State wrapper, tracks the actual *pointers* to VBO memory

```
void draw() {
    if (m_vao == 0) return;
    // bind our VAO which sets up all our buffers and data for us
    glBindVertexArray(m_vao);
    // tell opengl to draw our VAO using the draw mode and how many verticies to render
    glDrawElements(GL_TRIANGLES, m_indices.size(), GL_UNSIGNED_INT, 0);
}
```

https://www.khronos.org/opengl/wiki/Tutorial2:_VAOs,_VBOs,_Vertex_and_Fragment_Shaders_(C_/_SDL)

Vertex buffer objects (VBOs):

Three vertices with each vertex having a 3D position:

```
std::vector<glm::vec3> m_positions;
// vertex 1
m_positions.push_back(glm::vec3(0, 0, 0));
// vertex 2
m_positions.push_back(glm::vec3(10, 0, 0));
// vertex 3
m_positions.push_back(glm::vec3(0, 10, 0));
```

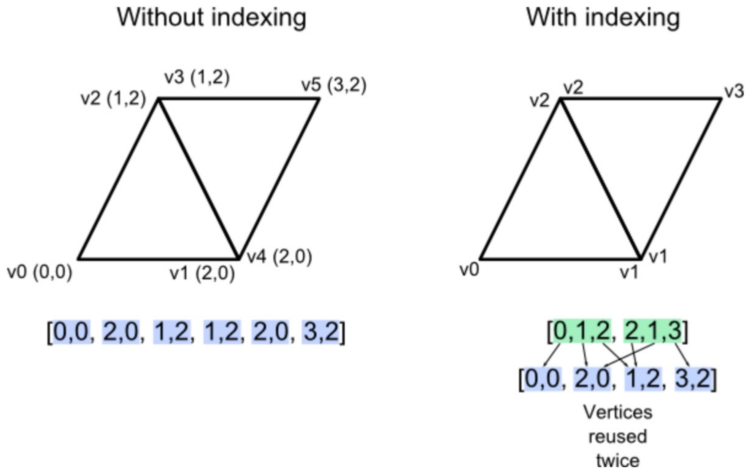
Generate a buffer ID, bind and upload data:

```
// upload Positions to VBO buffer
GLuint m_vbo_pos = 0;
glBindBuffer(GL_ARRAY_BUFFER, m_vbo_pos);
glBufferData(GL_ARRAY_BUFFER, sizeof(glm::vec3) * m_positions.size(), m_positions.data(),
    GL_STATIC_DRAW);
```

Index buffer object (IBO):

Create a triangle face by adding three vertices:

```
std::vector<unsigned int> m_indices;
m_indices.push_back(0);
m_indices.push_back(1);
m_indices.push_back(2);
```



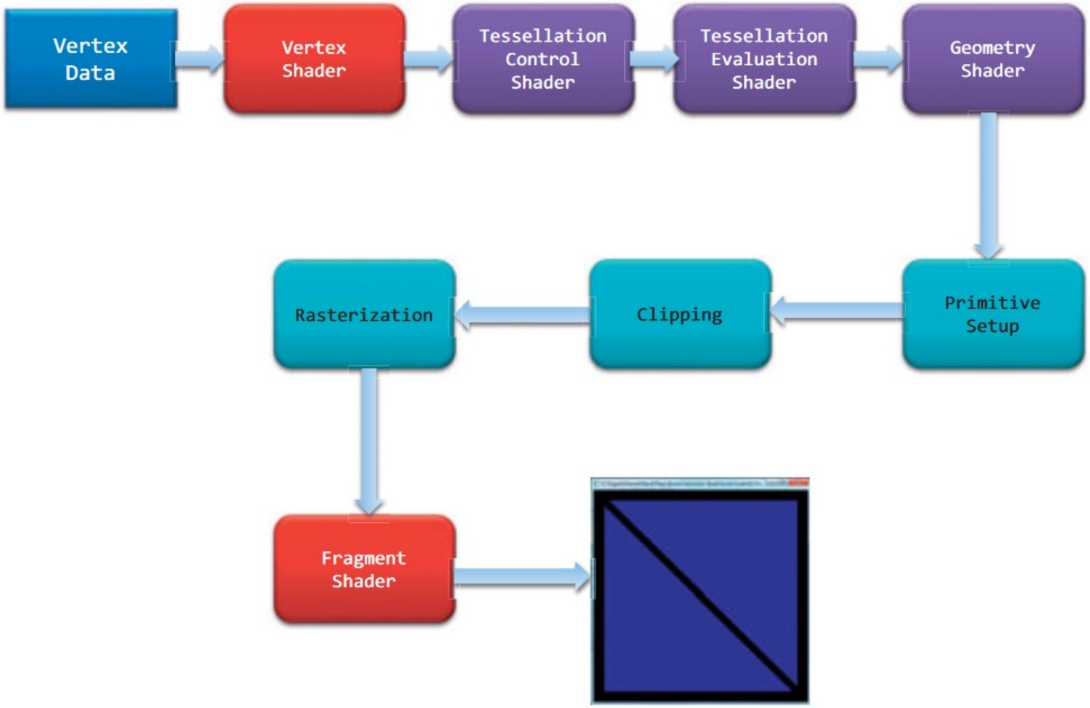
Generate a buffer ID, bind and upload data:

```
    // IBO
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, m_ibo);
    // upload the indices for drawing primitives
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(unsigned int) * m_indices.size(), m_indices.data(),
        GL_STATIC_DRAW);

    // clean up by binding VAO 0 (good practice)
    glBindVertexArray(0);
```

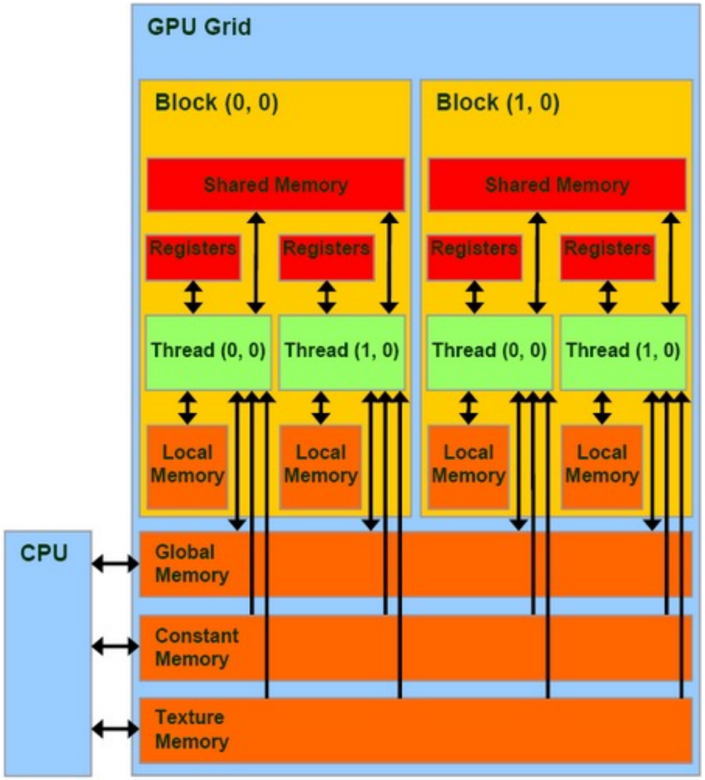
Picture credit: http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-9-vbo-indexing/#the-principle-of-indexing

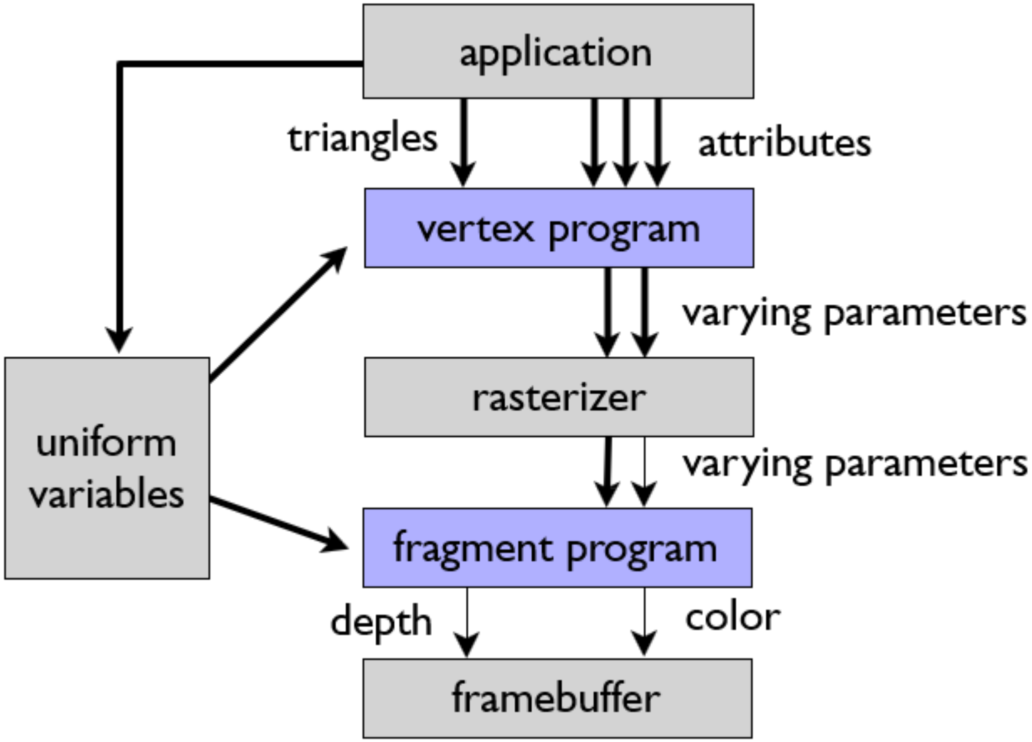
Shader Program

- A small C/C++ style program to control parts of the graphics pipeline
- Consists of 2 (or more) separate parts:
  - **Vertex shader** controls vertex transformation.
  - **Fragment shader** controls fragment shading.



GPU memory model





In the Framework

default_vert.glsl

```glsl
#version 330 core

// uniform data
uniform mat4 uProjectionMatrix;
uniform mat4 uModelViewMatrix;

// mesh data
layout(location = 0) in vec3 aPosition;
layout(location = 1) in vec3 aNormal;

// model data (this must match the input of the vertex shader)
out VertexData {
	vec3 position;
	vec3 normal;
} v_out;

void main() {
	// transform vertex data to viewspace
	v_out.position = (uModelViewMatrix * vec4(aPosition, 1)).xyz;
	v_out.normal = normalize((uModelViewMatrix * vec4(aNormal, 0)).xyz);

	// set the screenspace position (needed for converting to fragment data)
	gl_Position = uProjectionMatrix * uModelViewMatrix * vec4(aPosition, 1);
}
```

default_frag.glsl

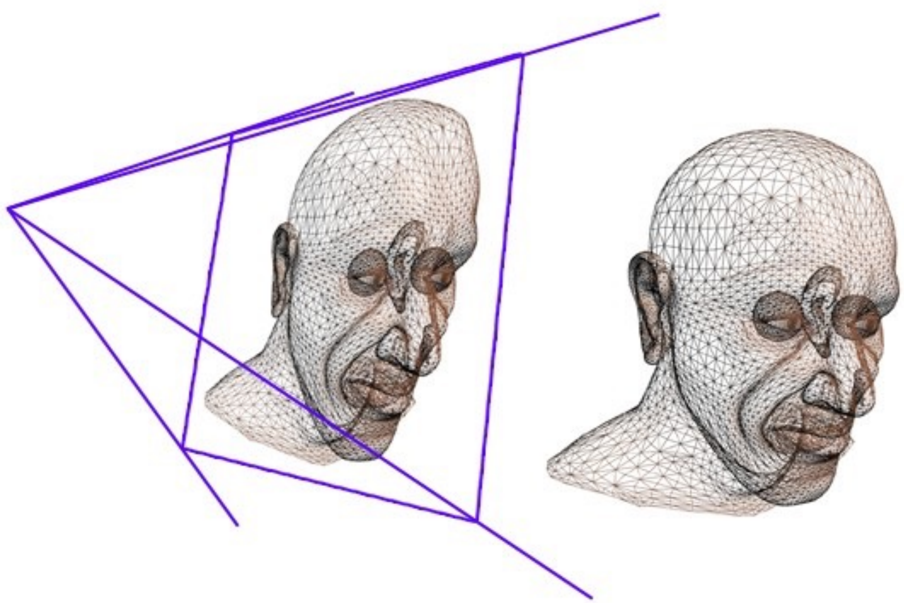
```glsl
#version 330 core

// uniform data
uniform mat4 uProjectionMatrix;
uniform mat4 uModelViewMatrix;

// viewspace data (this must match the output of the fragment shader)
in VertexData {
	vec3 position;
	vec3 normal;
} f_in;

// framebuffer output
out vec4 fb_color;

void main() {
	// calculate shading
	vec3 surfaceColor = vec3(0.066, 0.341, 0.215);
	vec3 eye = normalize(-f_in.position); // direction towards the eye
	float light = abs(dot(normalize(f_in.normal), eye)); // difference between the
surface normal and direction towards the eye
	vec3 finalColor = mix(surfaceColor / 4, surfaceColor, light);

	// output to the frambuffer
	fb_color = vec4(finalColor, 1);
}
```

```cpp
// set shader and upload variables
glUseProgram(m_shader);
glUniformMatrix4fv(glGetUniformLocation(m_shader, "uProjectionMatrix"), 1, false, value_ptr(proj));
glUniformMatrix4fv(glGetUniformLocation(m_shader, "uModelViewMatrix"), 1, false, value_ptr(view));
```

```cpp
// VBO
//
// upload Positions to this buffer
glBindBuffer(GL_ARRAY_BUFFER, m_vbo_pos);
glBufferData(GL_ARRAY_BUFFER, sizeof(glm::vec3) * m_positions.size(), m_positions.data(), GL_STATIC_DRAW);
// this buffer will use location=0 when we use our VAO
glEnableVertexAttribArray(0);
// tell opengl how to treat data in location=0 - the data is treated in lots of 3 (3 floats = vec3)
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, nullptr);

// do the same thing for Normals but bind it to location=1
glBindBuffer(GL_ARRAY_BUFFER, m_vbo_norm);
glBufferData(GL_ARRAY_BUFFER, sizeof(glm::vec3) * m_normals.size(), m_normals.data(), GL_STATIC_DRAW);
glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, nullptr);
```

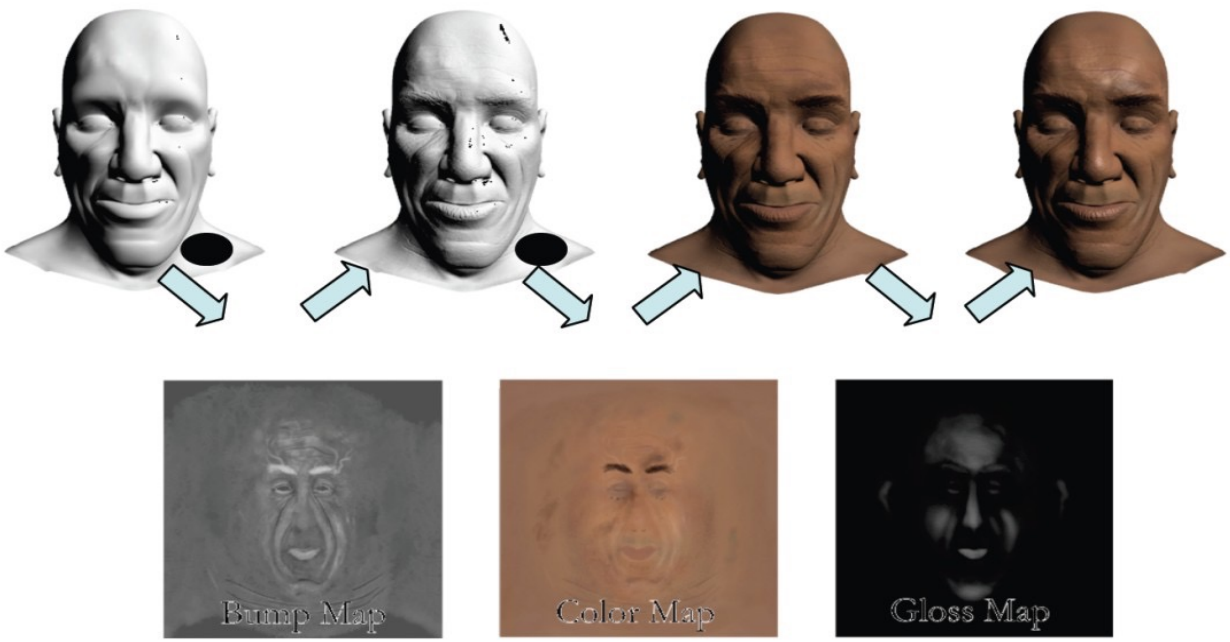
Vertex Shader

- Transform vertices from object space to clip space.
  - Conventionally modelview followed by projection
  - Can define custom transformation to clip space
- Compute other data that are interpolated with vertices.
  - Color
  - Normals
  - Texture coordinates
  - Etc.



Fragment Shader

- Compute the color of a fragment (i.e. a pixel).
- Take interpolated data from vertex shaders.
- Can read more data from:
  - Textures
  - User specified values



GLSL

- Similar to C/C++
- Used to write shaders
  - Vertex, tessellation, geometry, fragment
  - **We only cover vertex and fragment here!**
- Based on OpenGL
- First available in OpenGL 2.0 (2004)
- Competitors:
  - Nvidia Cg
  - Microsoft HLSL
  - Apple Metal Shading Language (MSL)
  - Etc.

Modern OpenGL

- We'll concentrate on the latest versions of OpenGL
- They enforce a new way to program with OpenGL
  - Allows more efficient use of GPU resources
- Modern OpenGL doesn't support many of the "classic" ways of doing things, such as
  - Fixed-function graphics operations, like vertex lighting and transformations
- All applications must use *shaders* for their graphics processing

Typical shader structure

```
/*
    Multiple-lined comment
*/

// Single-lined comment

//
// Global variable definitions
//

void main()
{
    //
    // Function body
    //
}
```

GLSL Data Types

- Scalar types: `float, int, bool`
- Vector types: `vec2, vec3, vec4`
  `ivec2, ivec3, ivec4`
  `bvec2, bvec3, bvec4`
- Matrix types: `mat2, mat3, mat4`
- Texture sampling: `sampler1D, sampler2D, sampler3D, samplerCube`
- C++ Style Constructors

```
vec3 a = vec3(1.0, 2.0, 3.0);
```

Operators

- Standard C/C++ arithmetic and logic operators
- Overloaded operators for matrix and vector operations

```
mat4 m;
vec4 a, b, c;

b = a*m;
c = m*a;
```

Components and Swizzling

- Access vector components using either:
  - [ ] (c-style array indexing)
  - `xyzw`, `rgba` or `strq` (named components)

- For example:

```
vec3 v;
v[1], v.y, v.g, v.t
```

  \- all refer to the same element

- Component swizzling:

```
vec3 a, b;
a.xy = b.yx;
```

Qualifiers

- `in`, `out`
  - Copy vertex attributes and other variable into and out of shaders

```
in  vec2 texCoord;
out vec4 color;
```

- `uniform`
  - shader-constant variable from application

```
uniform float time;
uniform vec4 rotation;
```

Functions

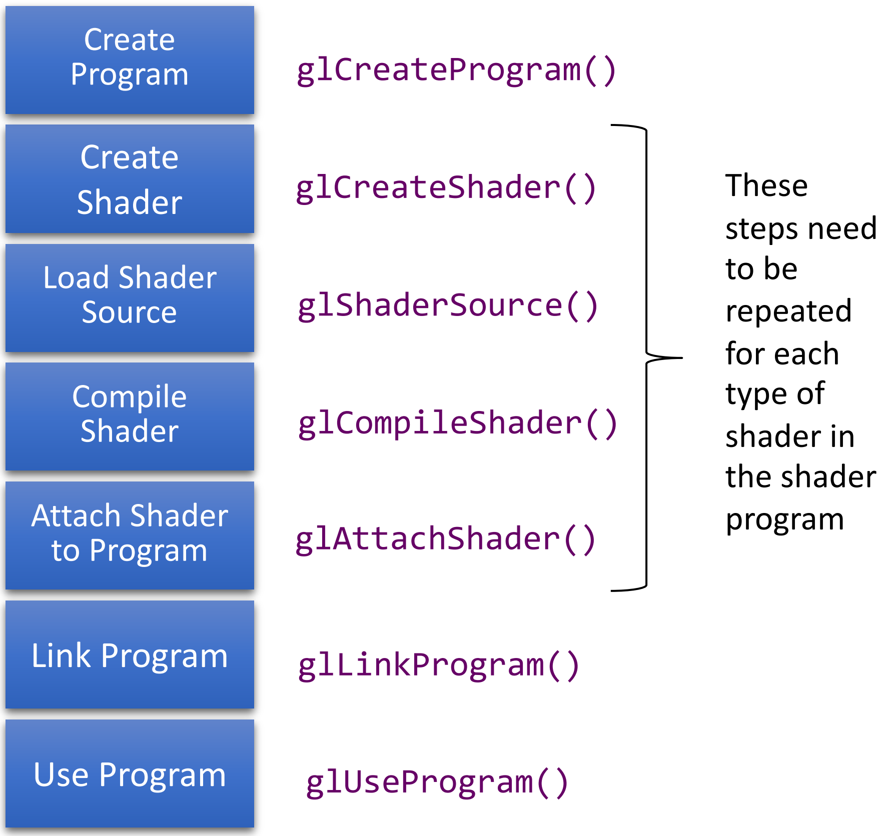
- Built in
  - Arithmetic: `sqrt`, `power`, `abs`
  - Trigonometric: `sin`, `asin`
  - Graphical: `length`, `reflect`
- User defined

Built-in Variables

- `gl_Position`
  - (required) output position from vertex shader
- `gl_FragCoord`
  - input fragment position
- `gl_FragDepth`
  - input depth value in fragment shader

Getting Your Shaders into OpenGL

- Shaders need to be compiled and linked to form an executable shader program
- OpenGL provides the compiler and linker
- A program must contain
  - vertex and fragment shaders
  - other shaders are optional



shader_builder class:

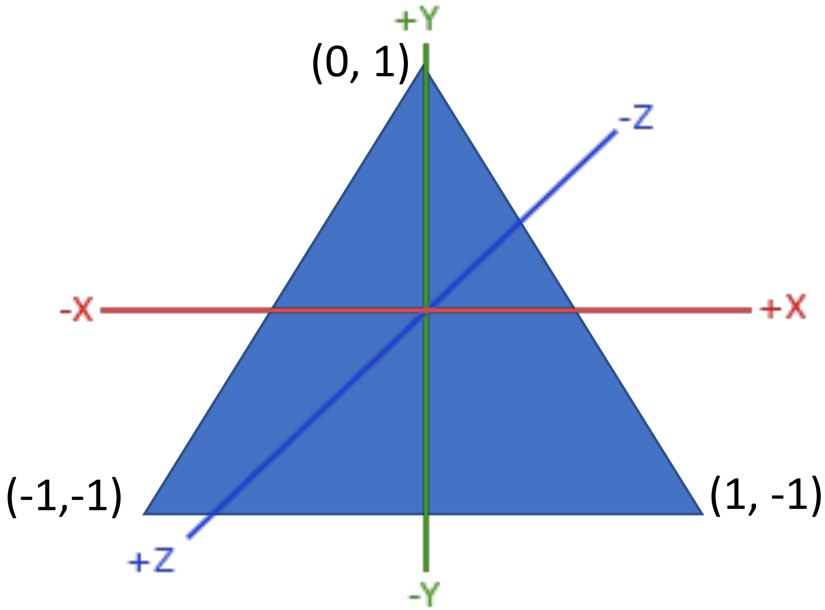
```
// build the shader
shader_builder color_sb;
color_sb.set_shader(GL_VERTEX_SHADER, CGRA_SRCDIR + std::string("//res//shaders//default_vert.glsl"));
color_sb.set_shader(GL_FRAGMENT_SHADER, CGRA_SRCDIR + std::string("//res//shaders//default_frag.glsl"));
m_shader = color_sb.build();
```

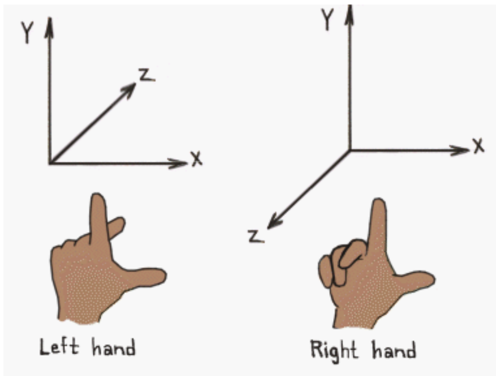
VUW color triangle

Vertex Shader

VUW.vs.glsl:

```
#version 330 core



attribute vec3 vertPosition;


void main()
{
    gl_Position = vec4(vertPosition, 1);
}
```





Vertex Shader

VUW.vs.glsl:

```
#version 330 core

attribute vec3 vertPosition;

void main()
{
    gl_Position = vec4(vertPosition, 1);
}
```

Each time the screen is drawn, this main() function is called once per vertex, as if it were in a for loop.

Vertex Shader

VUW.vs.glsl:

```
#version 330 core

attribute vec3 vertPosition;


void main()
{
    gl_Position = vec4(vertPosition, 1);
}
```

The first thing to do is specify the GLSL version. We use version 3.3 in this class. (Note: other versions can be very different!)

Vertex Shader

VUW.vs.glsl:

```
#version 330 core

attribute vec3 vertPosition;

void main()
{
    gl_Position = vec4(vertPosition, 1);
}
```

Attribute variables link to vertex attributes, or data associated with each vertex. This one is set to the vertex position buffer. Each time main() is executed, vertPosition is set to the vertex currently being processed.

Vertex Shader

VUW.vs.glsl:

```
#version 330 core

attribute vec3 vertPosition;

void main()
{
    gl_Position = vec4(vertPosition, 1);
}
```

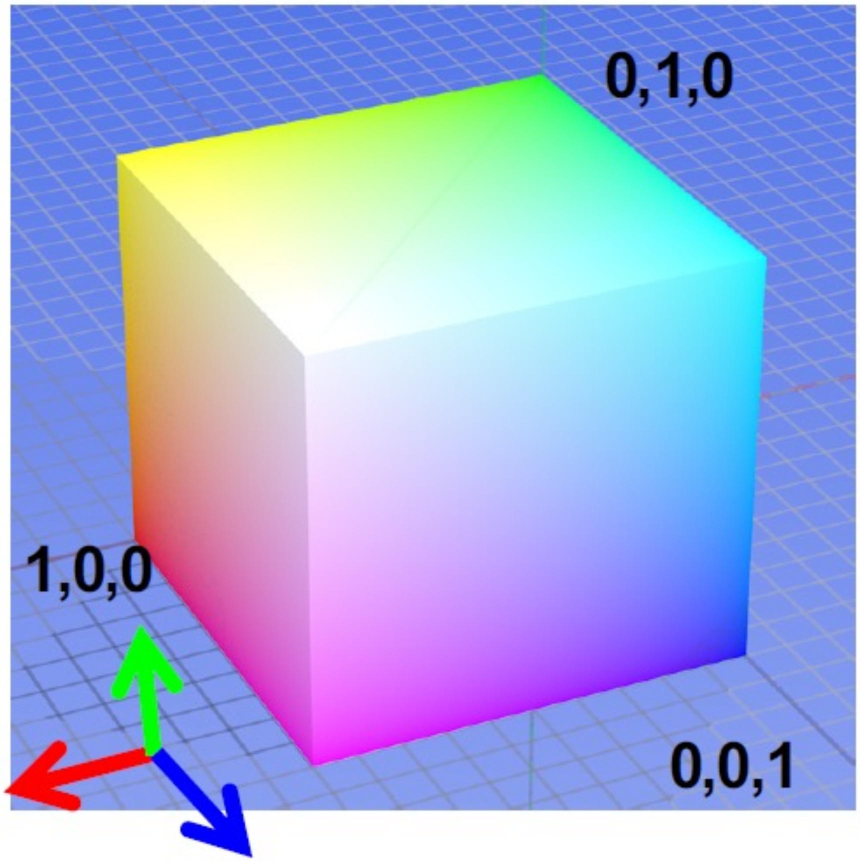
gl_Position is a special variable that holds the position of the vertex in clip space.

Since a vertex shader's main output is the position in clip space, it must **always** set gl_Position.

This vertex shader just directly gives a vec3 to vec4's constructor.

Color model: RGB

Default colour space



Some drawbacks
- Strongly correlated channels
- Non-perceptual

R
(G=0,B=0)

G
(R=0,B=0)

B
(R=0,G=0)

Fragment Shader

VUW.fs.glsl:

```
#version 330 core

out vec4 color;


void main()
{
  color = vec4(0.066, 0.341, 0.215, 1.0);
}
```

Fragment Shader

VUW.fs.glsl:

```
#version 330 core

out vec4 color;


void main()
{
  color = vec4(0.066, 0.341, 0.215, 1.0);
}
```

Each time the screen is drawn, this main() function is called once per pixel.

Fragment Shader

This is a special variable that stores the color of the output fragment.

VUW.fs.glsl:

```
#version 330 core

out vec4 color;

void main()
{
  color = vec4(0.066, 0.341, 0.215, 1.0);
}
```

Since a fragment shader computes the color of a fragment, it must **always** set the output.

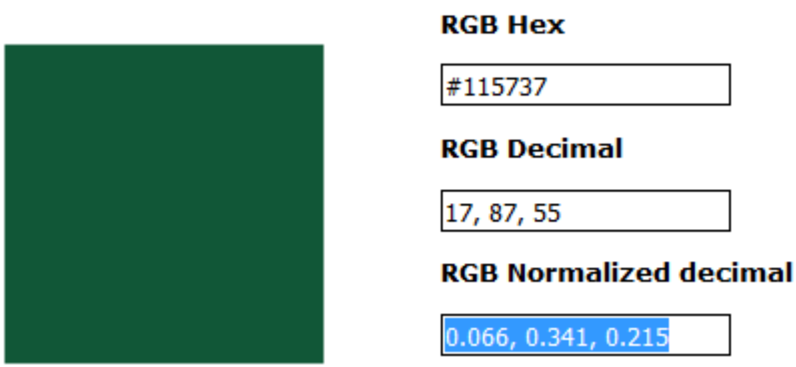
Fragment Shader

vec4 is a data type of 4D vector.

VUW.fs.glsl:

```
#version 330 core

out vec4 color;


void main()
{
   color = vec4(0.066, 0.341, 0.215, 1.0);
}
```

Can be used to store:
-homogeneous coordinates
-RGBA color



vec4(…) constructs an RGBA tuple with R=0.066, G=0.341, B=0.215, A=1, which is normalized VUW color.

Associating Shader Variables and Data

- Need to associate a shader variable with an OpenGL data source
  - vertex shader attributes → app vertex attributes
  - shader uniforms → app provided uniform values
- OpenGL relates shader variables to indices for the app to set
- Two methods for determining variable/index association
  - specify association before program linkage
  - query association after program linkage

Determining Locations After Linking

- Assumes you already know the variables' names

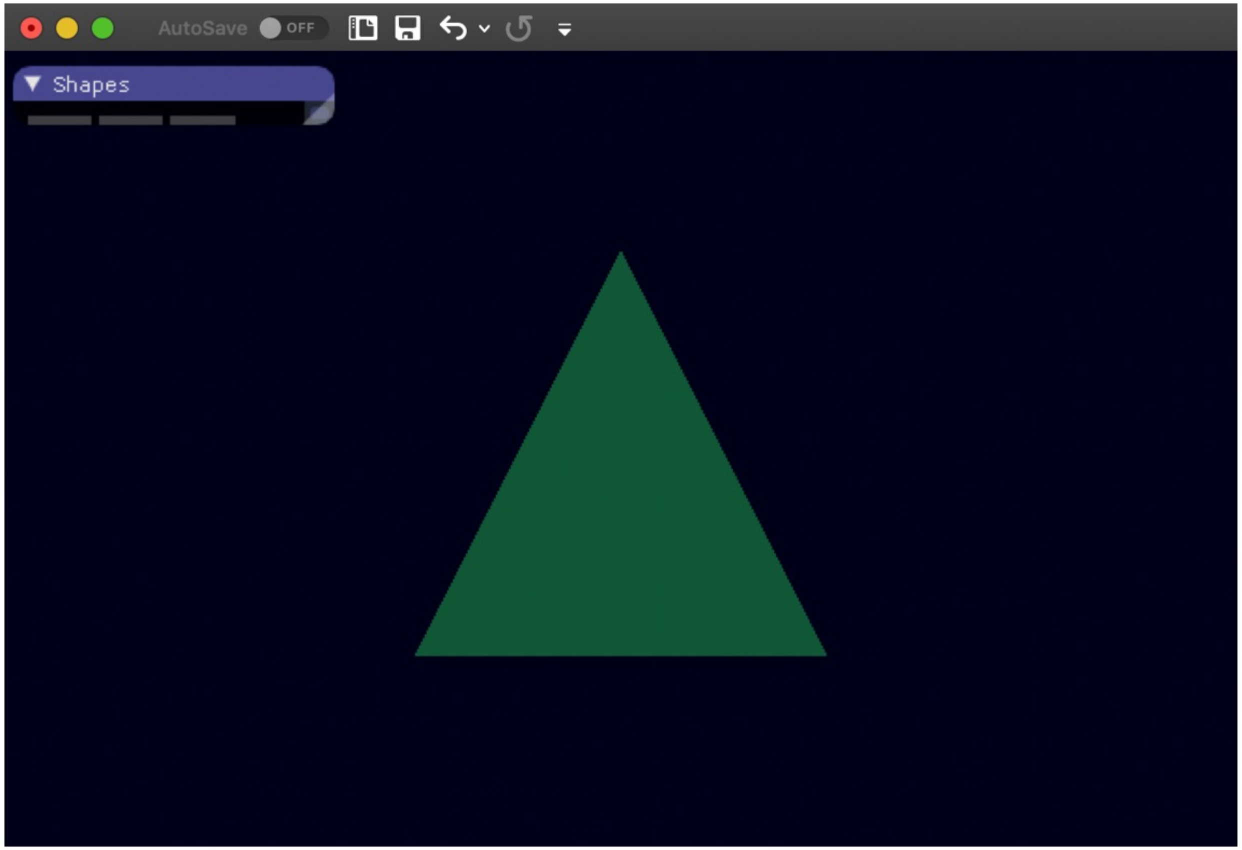
```
GLint loc = glGetAttribLocation( program, “name” );

GLint loc = glGetUniformLocation( program, “name” );
```

Initializing Uniform Variable Values

- Uniform Variables

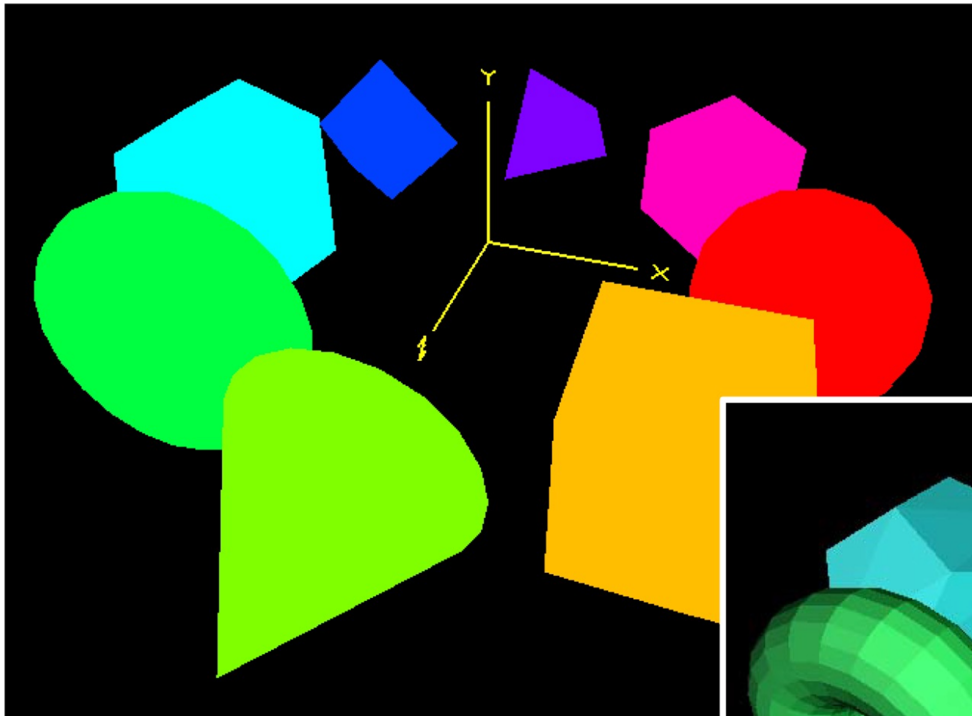
```
glUniform4f( index, x, y, z, w );

GLboolean  transpose = GL_TRUE;

GLfloat  mat[3][4][4] = { … };
glUniformMatrix4fv( index, 3, transpose, mat );
```

VUW Triangle Demo

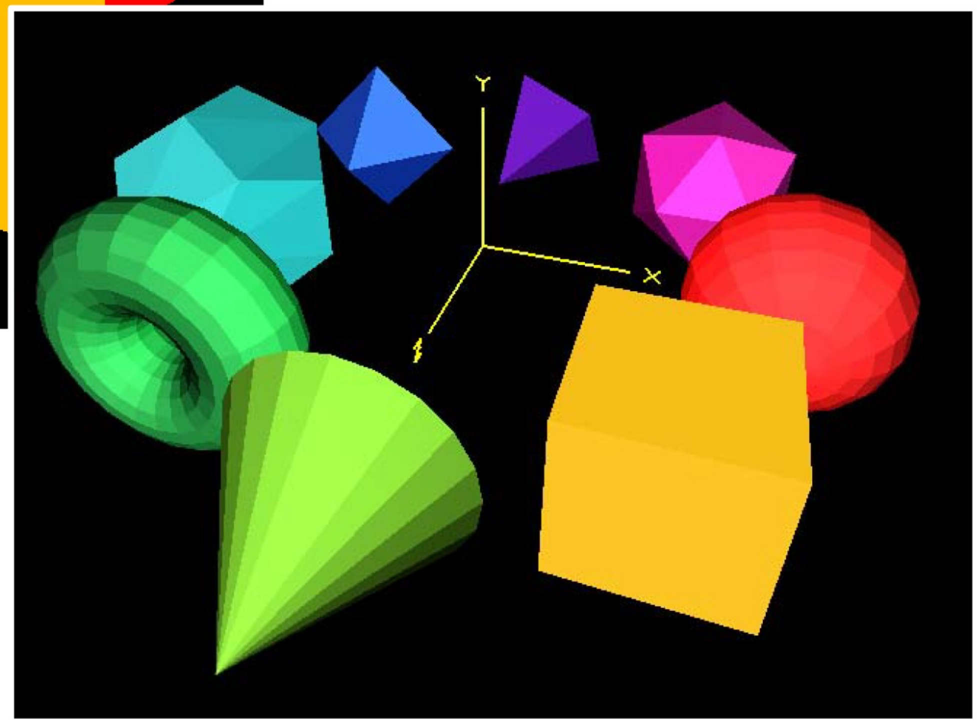


Why Do We Care About Lighting?

Lighting "dis-ambiguates" 3D scenes



Without lighting

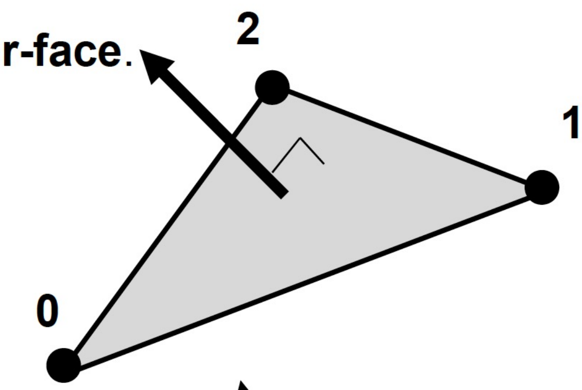


With lighting

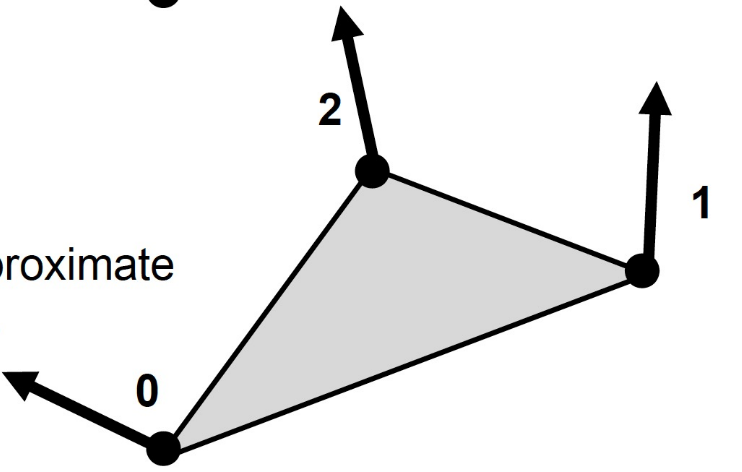
The Surface Normal

A surface normal is a vector perpendicular to the surface.

Sometimes surface normals are defined or computed **per-face**.



Sometimes they are defined **per-vertex** to best approximate the underlying surface that the face is representing.



Next

- *Introduction to Lighting: continued*