Managing 3D Graphics Data

Prof. George Wolberg
Dept. of Computer Science
City College of New York

Sending Data from C++ to Shaders

- To draw objects, vertices and transformation matrices must be sent through the OpenGL shader pipeline.

- Two methods for sending data through the pipeline:
  - through a *buffer* to a *vertex attribute*, or
  - directly to a *uniform variable*

- In OpenGL, buffer is called **VBO**(**V**ertex **B**uffer **O**bject)

- VBOs are declared and instantiated in the C++/OpenGL application

Steps for Sending Object Vertices to a Vertex Attribute (1)

Vertices are usually sent by putting them in a VBO on the C++ side and associating that buffer with a vertex attribute declared in the shader. Here are the steps:

Done once - typically in **init()**:

1. create a buffer (VBO),
2. copy the vertices into the buffer.

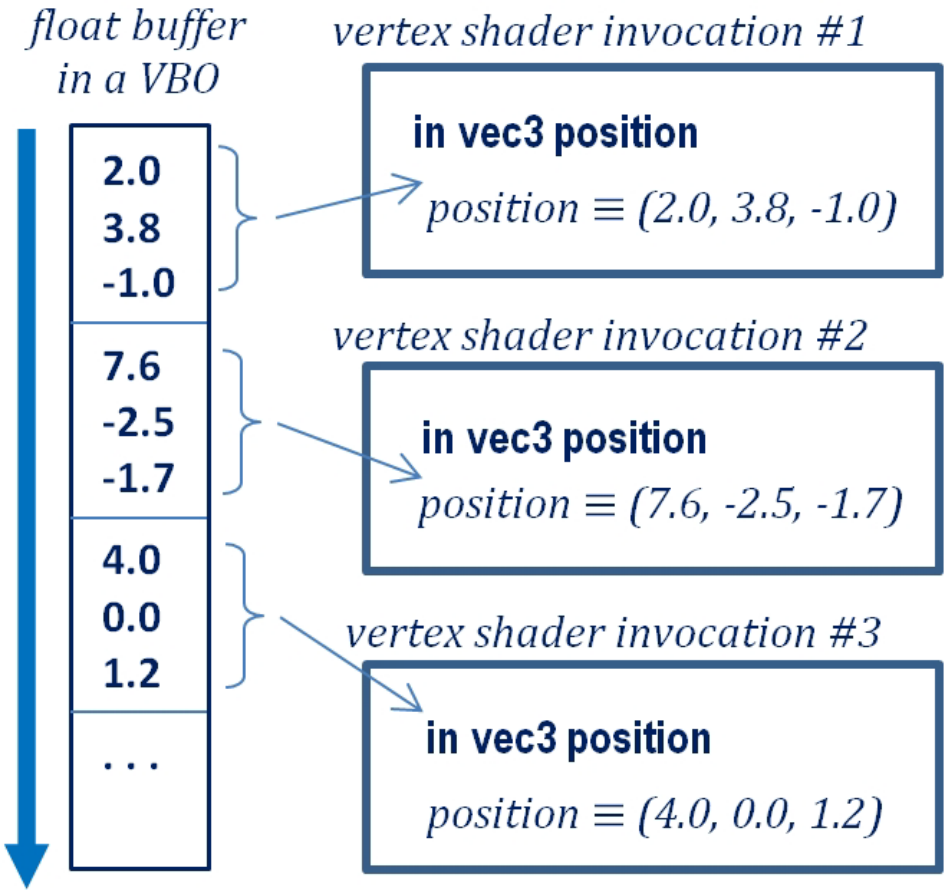
Done at each frame – typically in **display()**:

1. enable the buffer containing the vertices,
2. associate the buffer with a vertex attribute,
3. enable the vertex attribute,
4. use glDrawArrays(…) to draw the object

Steps for Sending Object Vertices to a Vertex Attribute (2)

- A scene may require many VBOs (one for each object).
- Generate and fill these buffers in init().
- A VBO interacts with a vertex attribute in a specific way.
- When glDrawArrays() is executed, the data in the buffer starts flowing through the vertex shader.
- The vertex shader executes *once per vertex*.
- A 3D vertex requires three values, so an appropriate vertex attribute in the shader would be of type **vec3.**
- For each three values in the buffer, the shader is invoked.

Data transmission between a VBO and a vertex attribute



The vertex shader runs once per vertex

Vertex Array Object (VAO)

- A related structure in OpenGL is the **VAO**(**V**ertex **A**rray **O**bject).
- VAOs organize buffers for easier manipulation in complex scenes.
- OpenGL requires that at least one VAO be created.
- For our purposes, one VAO will be sufficient.
- If we wish to display two objects, we can declare a single VAO on the C++ side, and an associated set of two VBOs (one per object).

```
GLuint vao[1];                  // declares array of one VAO
GLuint vbo[2];                  // declares array of two VBOs
...
glGenVertexArrays(1, vao);
glBindVertexArray(vao[0]);
glGenBuffers(2, vbo);           // create (instantiate) two VBOs
```

Note: a VAO is a required structure for organizing VBOs. At least one is required.

VAO and VBO

- glGenVertexArrays() and glGenBiffers() create VAOs and VBOs, respectively.
- They produce integer IDs for these items that are stored in vao[] and vbo[].
- The first function parameter refers to the number of IDs to create.
- The second parameter specifies the array to hold the returned IDs.
- The purpose of glBindVertexArrays() is to make the specified VAO “active”.
- The generated buffers will be associated with that VAO.
- A buffer needs to have a corresponding *vertex attribute* variable declared in shader.

Creating a vertex attribute (in vertex shader):

```
layout (location = 0) in vec3 position;
```

- The *in* indicates that this vertex attribute will be receiving values from a buffer.
- The *vec3* qualifier means that each invocation of the shader will grab three floats.
- The layout(location=0) portion of the command is called a layout qualifier.
- It is how we will associate the vertex attribute with a particular buffer.

Filling a VBO with Data

```
glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);          // make the 0th buffer "active"
glBufferData(GL_ARRAY_BUFFER, sizeof(vPositions), vPositions, GL_STATIC_DRAW);
```

Copy array (vPositions) into buffer

A float array containing the vertices

associating a VBO with a vertex attribute in the shader:

```
glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);          // make the 0th buffer "active"
glVertexAttribPointer(0, 3, GL_FLOAT, false, 0, 0);  // associate it with 0th vertex attribute
glEnableVertexAttribArray(0);                   // enable the 0th vertex attribute
```

*Associates the active VBO with the "**0th**" vertex attribute in the vertex shader.*
*Notice the associated "**location=0**" clause on the previous slide.*

glDrawArrays() will transmit data in the 0th VBO to the vertex attribute (in the shader) that has a layout qualifier with location 0

Steps for Using Uniform Variables

- Transformation matrices are typically sent from C++/OpenGL application to shader in a uniform variable. Send once and applies uniformly across entire primitive.

- Done in the vertex and fragment shaders:
  - Declare the name and type of the uniform variable

- Done in C++, typically in display():
  - Obtain a reference to the uniform variable,
  - Send desired data to the uniform variable

Declaring Uniform in a Shader

```
uniform  mat4  mv_matrix;
```

Obtaining a reference to the uniform:

```
mvLoc = glGetUniformLocation(renderingProgram, "mv_matrix");
```

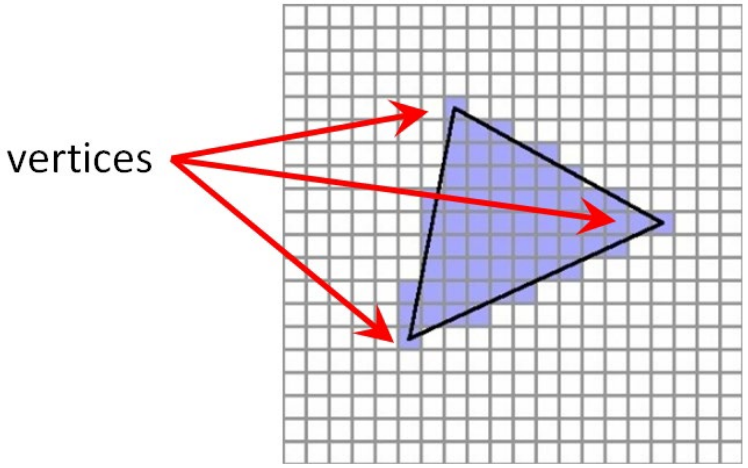
Sending data from C++ to a uniform:

```
glUniformMatrix4fv(mvLoc, 1, GL_FALSE, glm::value_ptr(mvMat));
```

Assumes that **mvMat** contains matrix data to be sent.
This can be created using GLM (described later).

Which One Should I Use?

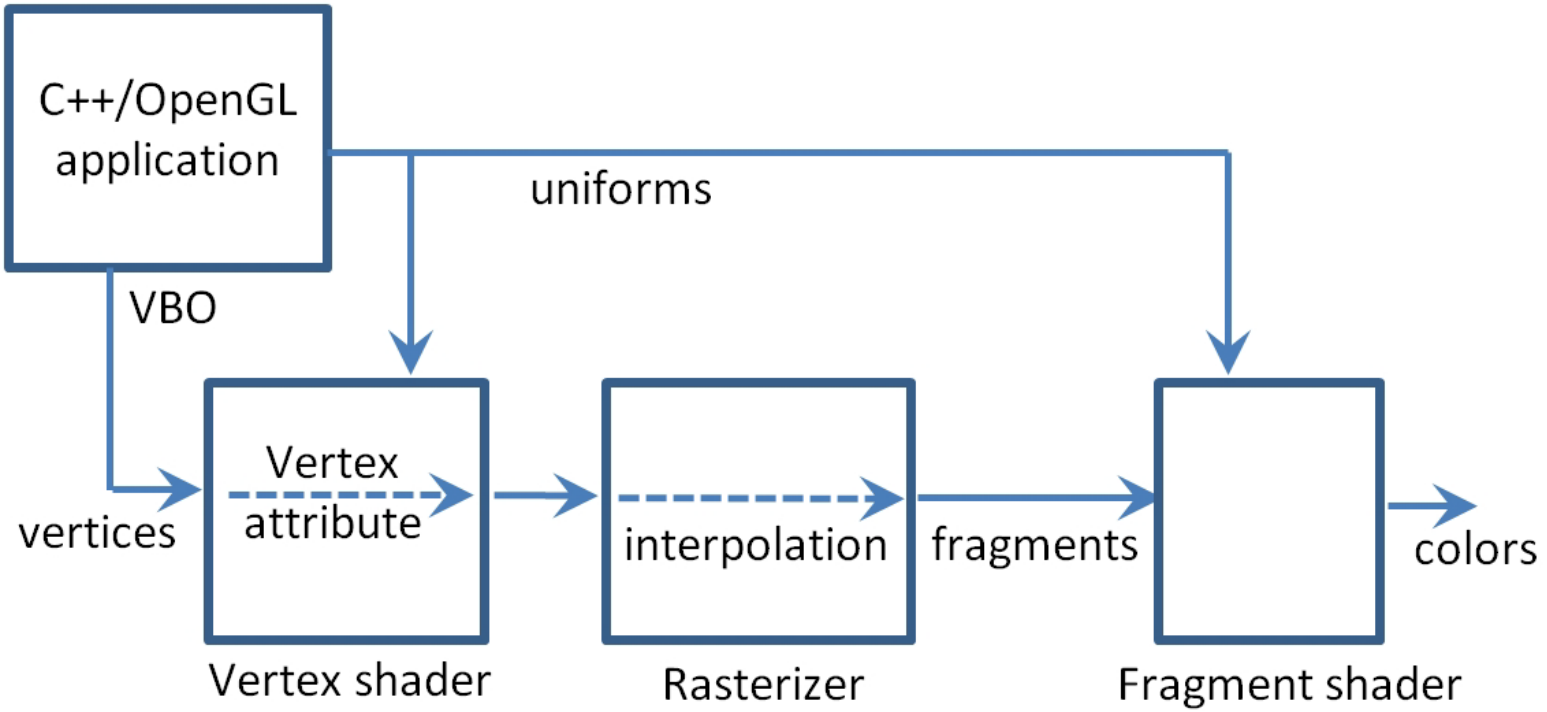
- Use a **uniform variable** for values that are constant for the entire object being drawn (such as MV and P transformation matrices)
- Use a **vertex attribute** when you want the values to be linearly interpolated by the rasterizer (such as the vertex positions, colors, normal, … in a model being drawn).



rasterization of vertices

Data Flow

Data flow for a C++/OpenGL application through pipeline shaders *(vertex attributes and uniform variables)*



Now, for a complete example…

Program to Draw a Red Cube
(many details not shown; refer to text)

C++/OpenGL Application

```
. . .
#include <glm\glm.hpp>
#include <glm\gtc\type_ptr.hpp>
#include <glm\gtc\matrix_transform.hpp>
#include "Utils.h"
. . .
float cameraX, cameraY, cameraZ;        // location of camera in scene
float cubeLocX, cubeLocY, cubeLocZ;     // location of cube object in scene
// allocate variables used in display(), so they won't need to be allocated during rendering
glm::mat4 pMat;  // perspective matrix
glm::mat4 vMat;  // view matrix
glm::mat4 mMat; // model matrix
glm::mat4 mvMat;                // model-view matrix
GLuint mvLoc, projLoc;
float aspect;

int main(void)  {
    //  same as before
}
```

Program to Draw a Red Cube

```
void init(void)  {
        renderingProgram = Utils::createShaderProgram("vertShader.glsl", "fragShader.glsl");
        cameraX = 0.0f; cameraY = 0.0f; cameraZ = 8.0f;
        cubeLocX = 0.0f; cubeLocY = -2.0f; cubeLocZ = 0.0f;  // shifted down to reveal perspective
        setupVertices();
}

void setupVertices(void)  {
    // 36 vertices of the 12 triangles making up a 2 x 2 x 2 cube centered at the origin
    float vertexPositions[108] = {
        -1.0f,  1.0f, -1.0f, -1.0f, -1.0f, -1.0f, 1.0f, -1.0f, -1.0f, 1.0f, -1.0f, -1.0f, 1.0f,  1.0f, -1.0f, -1.0f,  1.0f, -1.0f,
        1.0f, -1.0f, -1.0f, 1.0f, -1.0f,  1.0f, 1.0f,  1.0f, -1.0f, 1.0f, -1.0f,  1.0f, 1.0f,  1.0f,  1.0f, 1.0f,  1.0f, -1.0f,
        1.0f, -1.0f,  1.0f, -1.0f, -1.0f,  1.0f, 1.0f,  1.0f,  1.0f, -1.0f, -1.0f,  1.0f, -1.0f,  1.0f,  1.0f, 1.0f,  1.0f,  1.0f,
        -1.0f, -1.0f,  1.0f, -1.0f, -1.0f, -1.0f, -1.0f,  1.0f,  1.0f, -1.0f, -1.0f, -1.0f, -1.0f,  1.0f, -1.0f, -1.0f,  1.0f,  1.0f,
        -1.0f, -1.0f,  1.0f,  1.0f, -1.0f,  1.0f,  1.0f, -1.0f, -1.0f, 1.0f, -1.0f, -1.0f, -1.0f, -1.0f, -1.0f, -1.0f, -1.0f,  1.0f,
        -1.0f,  1.0f, -1.0f, 1.0f,  1.0f, -1.0f, 1.0f,  1.0f,  1.0f, 1.0f,  1.0f,  1.0f, -1.0f,  1.0f,  1.0f, -1.0f,  1.0f, -1.0f
    };

    glGenVertexArrays(1, vao);
    glBindVertexArray(vao[0]);
    glGenBuffers(numVBOs, vbo);

    glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertexPositions), vertexPositions, GL_STATIC_DRAW);
}
```

Program to Draw a Red Cube

```
void display(GLFWwindow* window, double currentTime)  {
    . . .
    // Create a perspective matrix, this one has fovy=60, aspect ratio matches screen window.
    glfwGetFramebufferSize(window, &width, &height);
    aspect = (float) width() / (float) height();
    pMat = glm::perspective(1.0472f, aspect, 0.1f, 1000.0f);      //  1.0472 radians = 60 degrees

    // build view, model, and model-view matrices
    vMat = glm::translate(glm::mat4(1.0f), glm::vec3(-cameraX, -cameraY, -cameraZ));
    mMat = glm::translate(glm::mat4(1.0f), glm::vec3(cubeLocX, cubeLocY, cubeLocZ));
    mvMat = vMat * mMat;

    // prepare uniform variables
    mvLoc = glGetUniformLocation(renderingProgram, "mv_matrix");
    projLoc = glGetUniformLocation(renderingProgram, "proj_matrix");
    gl.glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::value_ptr(pMat));
    gl.glUniformMatrix4fv(mvLoc, 1, GL_FALSE, glm::value_ptr(mvMat));

    // prepare vertex attribute containing cube vertices
    glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
    glVertexAttribPointer(0, 3, GL_FLOAT, false, 0, 0);
    glEnableVertexAttribArray(0);

    . . .
    glDrawArrays(GL_TRIANGLES, 0, 36);
}
```

Program to Draw a Red Cube

Vertex shader

```
#version 430

layout (location=0) in vec3 position;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;

void main(void) {
    gl_Position = proj_matrix * mv_matrix * vec4(position,1.0);
}
```

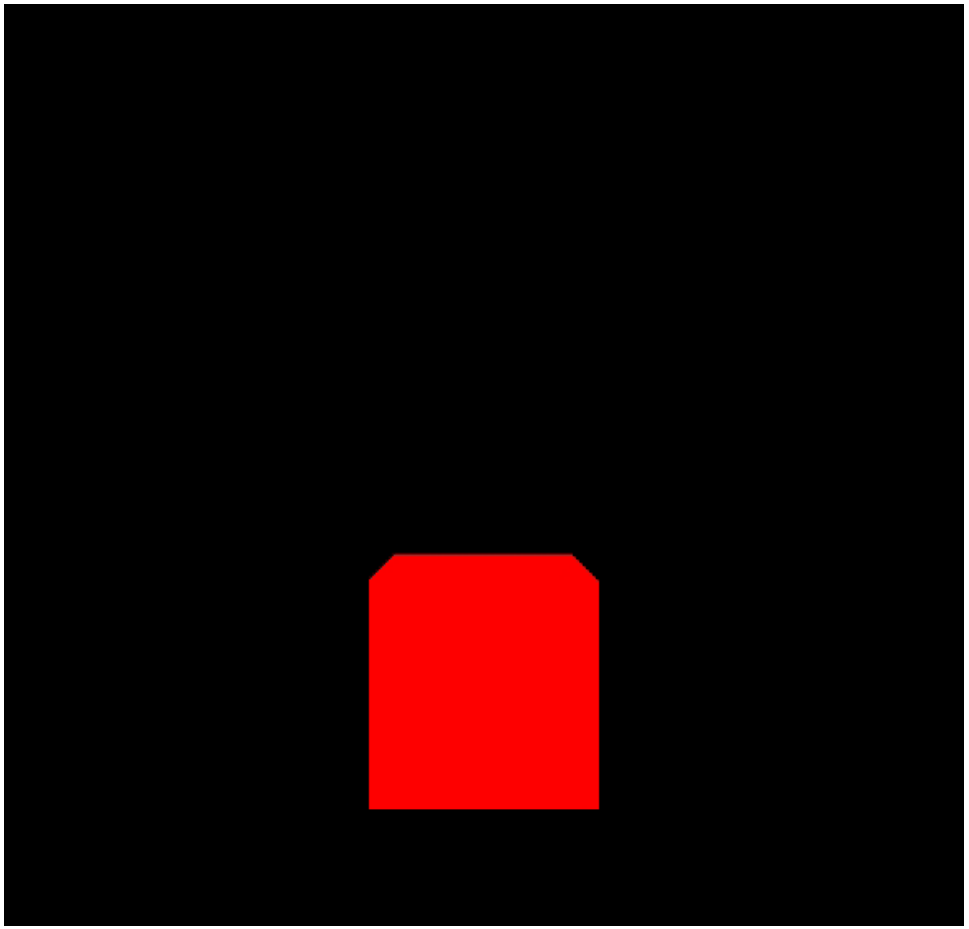
Fragment shader

```
#version 430

out vec4 color;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;

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

Program to Draw a Red Cube



Red cube positioned at (0,-2,0) viewed from (0,0,8)

Attribute Interpolation

All vertex attributes are interpolated, not just the vertex location. Example: let's add a second attribute for *color:*

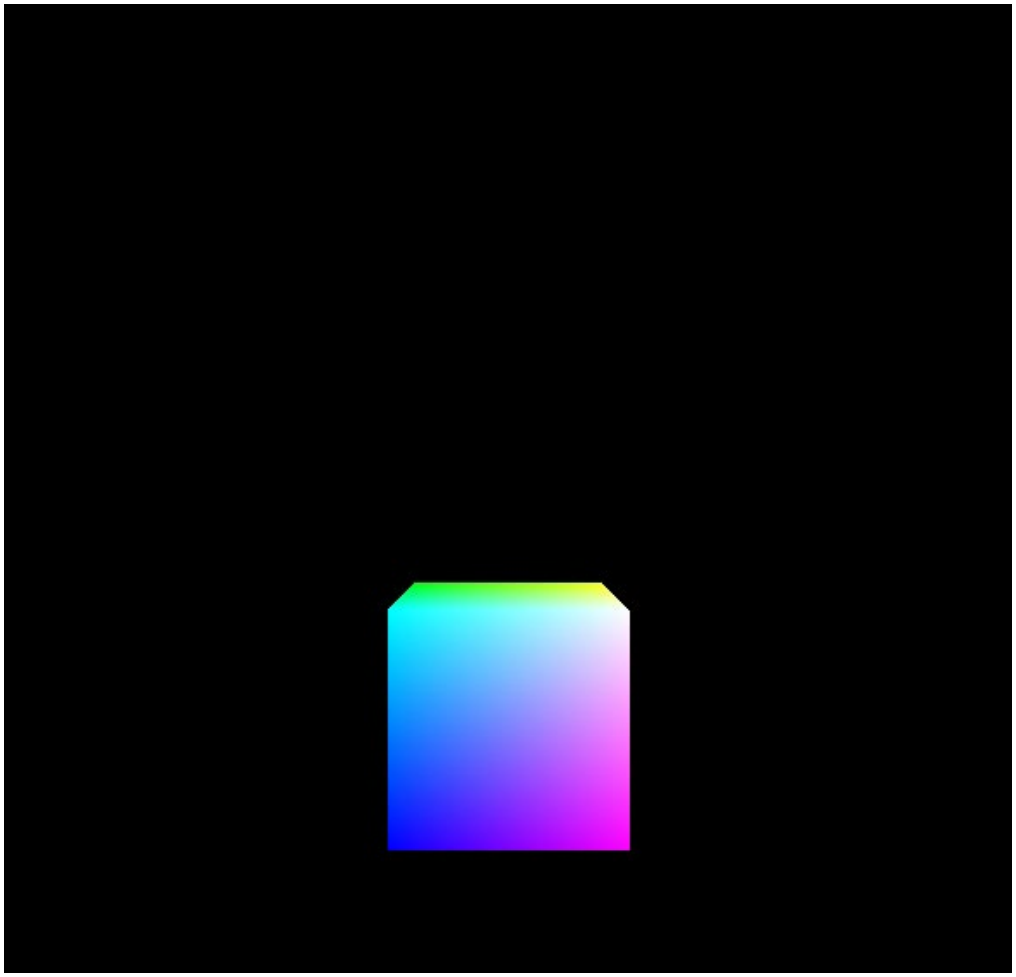
Revised vertex shader:

```
. . .
out vec4 varyingColor;          // (added)
void main(void) {
        gl_Position    = proj_matrix * mv_matrix * vec4(position,1.0);
        varyingColor = vec4(position,1.0) * 0.5 + vec4(0.5, 0.5, 0.5, 0.5);
}
```

Revised fragment shader:

```
in vec4 varyingColor;   // added
. . .
void main(void) {
        color = varyingColor;
}
```

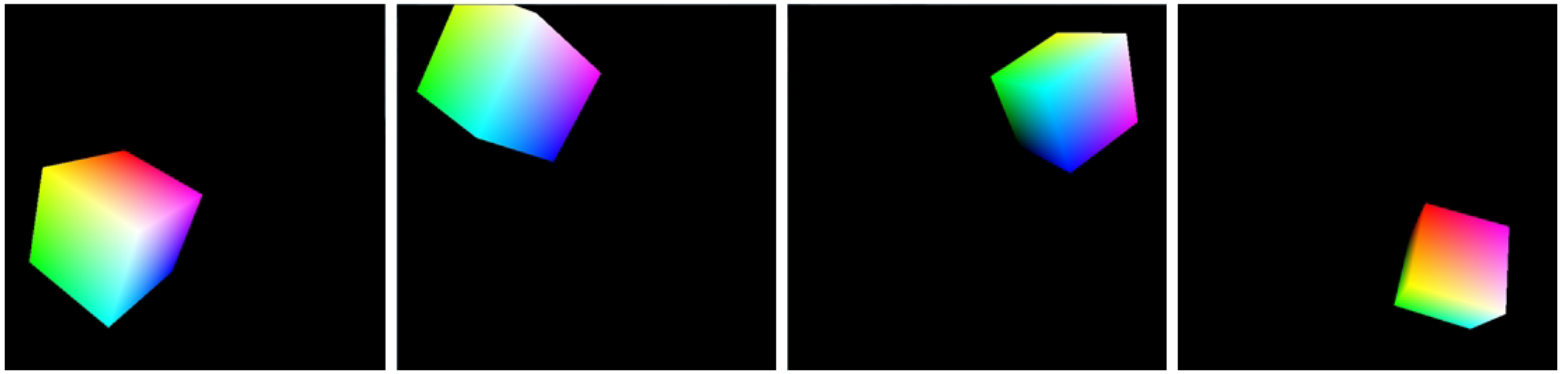
Cube with Interpolated Colors



Animating Cube

```
//    necessary to clear background each time, so that objects don't leave a trail:
glClear(GL_DEPTH_BUFFER_BIT);
glClear(GL_COLOR_BUFFER_BIT);

//    use current time – an incoming parameter of display() – to compute different
//    translations in x, y, z. The 1.75 adjusts the rotation speed
tMat = glm::translate(glm::mat4(1.0f),   glm::vec3(sin(0.35f*currentTime)*2.0f,
                                  cos(0.52f*currentTime)*2.0f, sin(0.7f*currentTime)*2.0f));
rMat = glm::eulerAngleYXZ(1.75*currentTime, 1.75*currentTime, 1.75*currentTime);
mMat = tMat * rMat;
```



Instancing

- Replace **glDrawArrays()** with **glDrawArraysInstanced()**
- This causes glDrawArrays() to be invoked a specified number of times
- The vertex shader has access to **gl_InstanceID**
- **gl_InstanceID** equals 0 in the first instance, 1 in the second, etc.

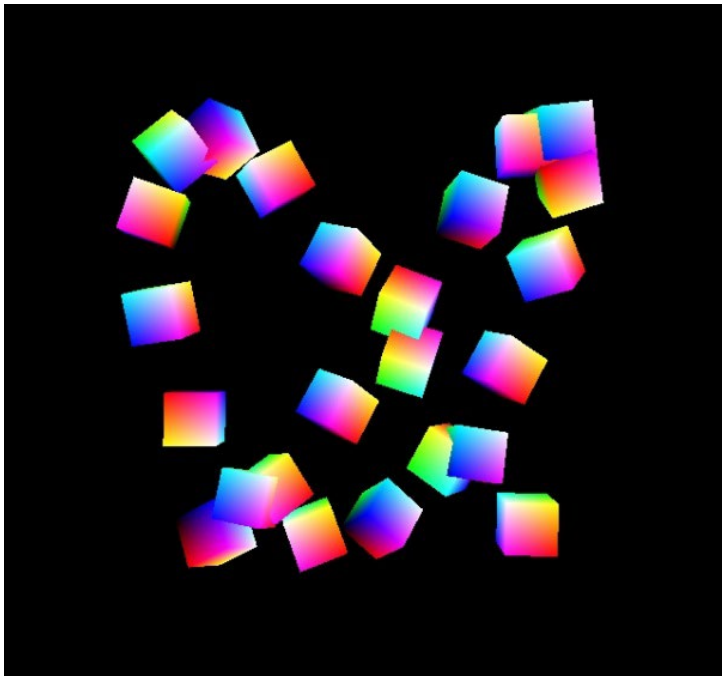
For example:

glDrawArraysInstanced(GL_TRIANGLES, 0, 36, 24);

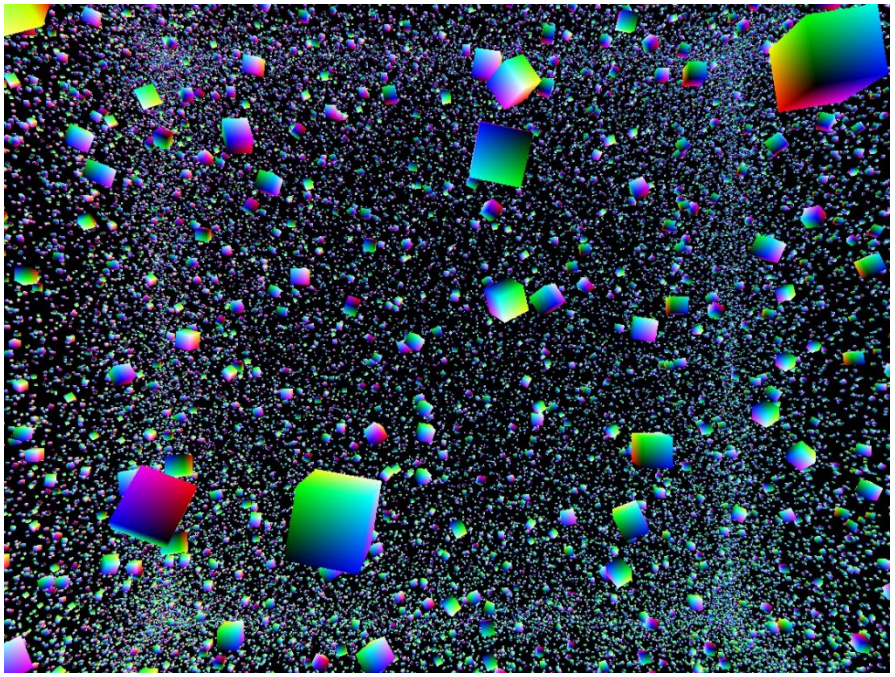
causes 24 cubes to be drawn.

Note that the vertex shader would need to use the gl_InstanceID variable to move each cube to different positions, or they would all be drawn in the same location!

Instancing Example



24 cubes



100,000 cubes

(see the textbook for complete code solutions)

Rendering Various Objects

- Place the vertices for each object in separate VBOs
- Each object will need its own Model matrix (M)
- The objects will share the same View and Projection matrices
- Call glDrawArrays() for each object being drawn

For example:

- Let's render a scene with a cube and a pyramid.
- Vertices for the cube are put in a VBO (as before).
- Vertices for the pyramid are put in a second VBO.
- Each object has its own, different Model (M) matrix.
- The same vertex and fragment shaders can be used for both objects.

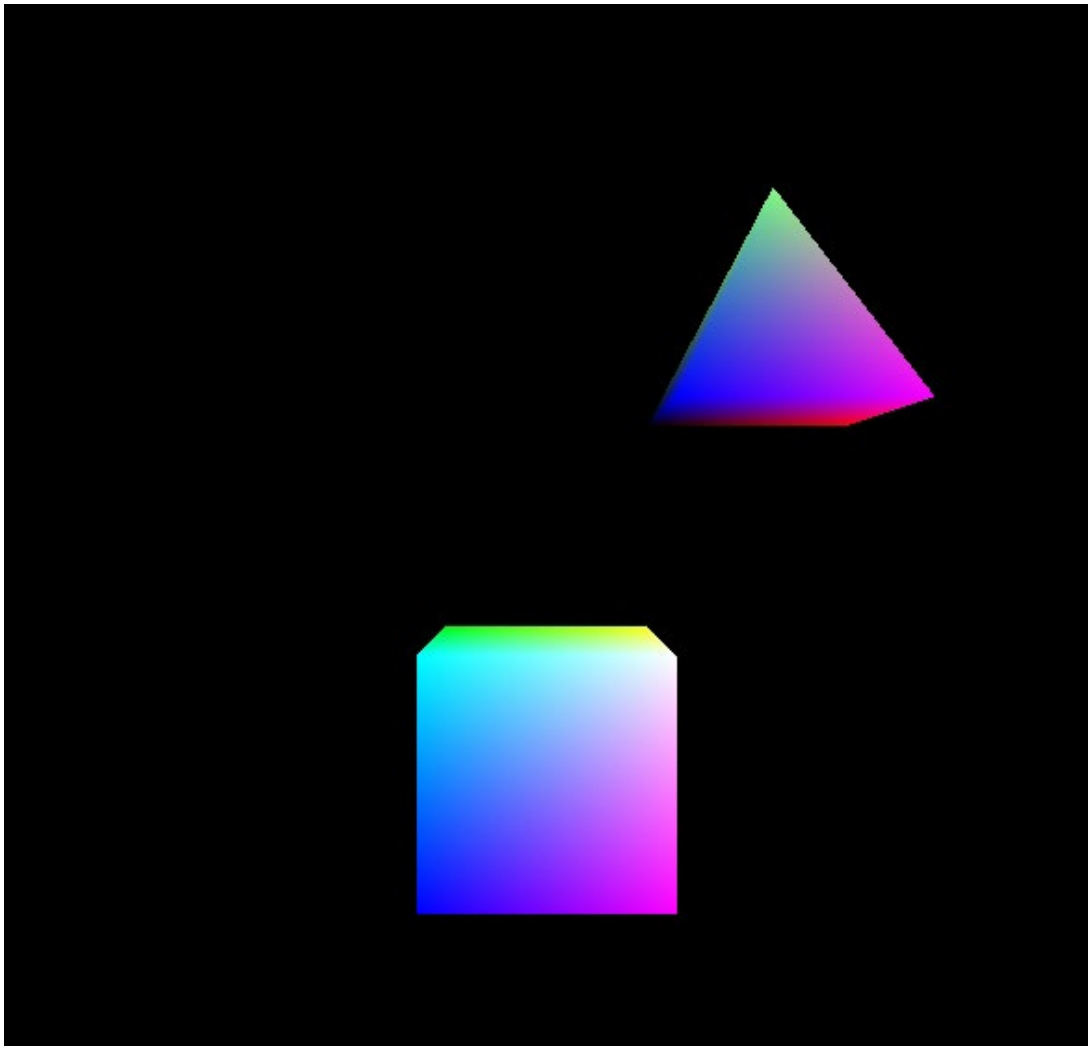
Cube and Pyramid Example (1)

```
void setupVertices()  {
    float[ cubePositions[108] = {
        -1.0f,  1.0f, -1.0f, -1.0f, -1.0f, -1.0f, 1.0f, -1.0f, -1.0f,
        1.0f, -1.0f, -1.0f, 1.0f,  1.0f, -1.0f, -1.0f,  1.0f, -1.0f,
                        … same as before, for the rest of the cube vertices
    };
    //  pyramid with 18 vertices, comprising 6 triangles (four sides, and two on the bottom)
    float pyrPositions[54] = {
        -1.0f, -1.0f, 1.0f, 1.0f, -1.0f, 1.0f, 0.0f, 1.0f, 0.0f,        // front face
        1.0f, -1.0f, 1.0f, 1.0f, -1.0f, -1.0f, 0.0f, 1.0f, 0.0f,        // right face
        1.0f, -1.0f, -1.0f, -1.0f, -1.0f, -1.0f, 0.0f, 1.0f, 0.0f,      // back face
        -1.0f, -1.0f, -1.0f, -1.0f, -1.0f, 1.0f, 0.0f, 1.0f, 0.0f,      // left face
        -1.0f, -1.0f, -1.0f, 1.0f, -1.0f, 1.0f, -1.0f, -1.0f, 1.0f,     // base – left front
        1.0f, -1.0f, 1.0f, -1.0f, -1.0f, -1.0f, 1.0f, -1.0f, -1.0f      // base – right back
    };
    //  initialize one VAO and two VBOs (as before)
    . . .
    glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);              // 0th VBO for cube
    glBufferData(GL_ARRAY_BUFFER, sizeof(cubePositions), cubePositions,GL_STATIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, vbo[1]);              // 1st VBO for pyramid
    glBufferData(GL_ARRAY_BUFFER, sizeof(pyrPositions), pyrPositions,GL_STATIC_DRAW);
}
```

Cube and Pyramid Example (2)

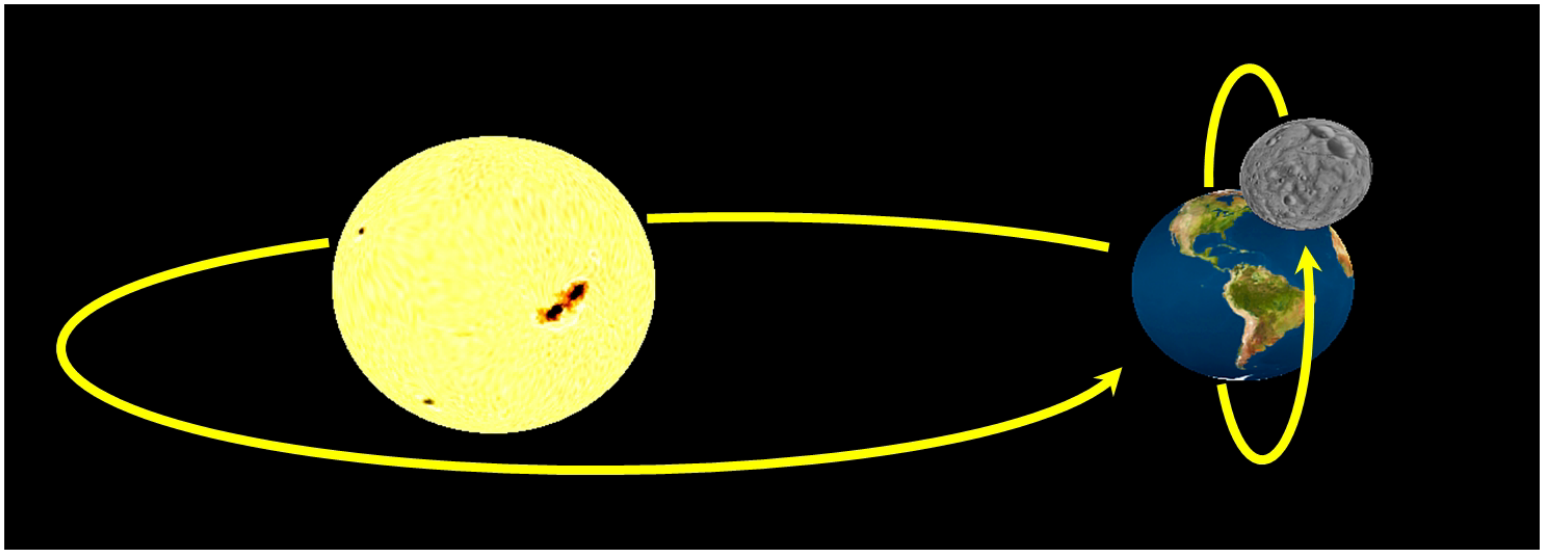
```
void display(GLFWwindow* window, double currentTime)  {
    //  set up the view matrix and rendering program, and pass uniforms
    //    to the shaders as before
    . . .
    // draw the cube using buffer #0
    mMat = glm::translate(glm::mat4(1.0f), glm::vec3(cubeLocX, cubeLocY, cubeLocZ));
    mvMat = vMat * mMat;
    . . .
    glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
    glEnableVertexAttribArray(0);
    . . .
    glDrawArrays(GL_TRIANGLES, 0, 36);
    // draw the pyramid using buffer #1
    mMat = glm::translate(glm::mat4(1.0f), glm::vec3(pyrLocX, pyrLocY, pyrLocZ));
    mvMat = vMat * mMat;
    . . .
    glBindBuffer(GL_ARRAY_BUFFER, vbo[1]);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
    glEnableVertexAttribArray(0);
    . . .
    glDrawArrays(GL_TRIANGLES, 0, 18);
}
```

Cube and Pyramid Example (3)



Matrix Stack

- Hierarchical models are useful for complex objects and scenes.
- A hierarchical scene built using a matrix stack:



The moon revolves around the earth, while the earth revolves around the sun.

C++ (STL) Stack Functions

- push() – *makes available a new entry on the top of the stack. We will use this by pushing a copy of the matrix that is at the top of the stack, and then concatenate additional transforms onto the copy.*
- pop() – *removes (and returns) the top matrix.*
- top(v) – *returns a reference to the matrix at the top of the stack, without removing it.*
- <stack>.top() *= rotate(*rotation matrix*)
- <stack>.top() *= scale(*scale matrix*)
- <stack>.top() *= translate(*trans.matrix*)

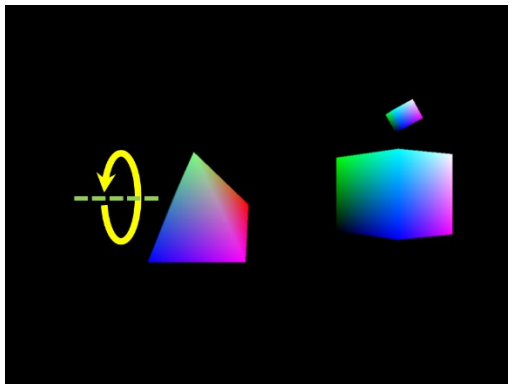
} Apply transforms directly to the top matrix in the stack

The basic approach is as follows:

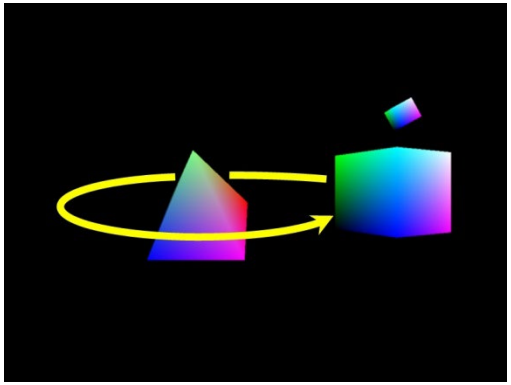
1. declare a single instance of: stack<glm::mat4>
2. when a new object is introduced relative to a parent object, do a “push” and then apply one of the transforms (rotate, scale, etc.)
3. when an object or sub-object has finished being drawn, “pop” its model-view matrix, removing it from atop the matrix stack.

Using Cubes and Pyramids

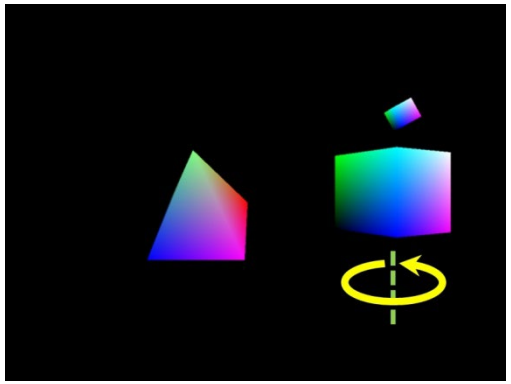
Start by pushing the view matrix **V** on the stack, then:



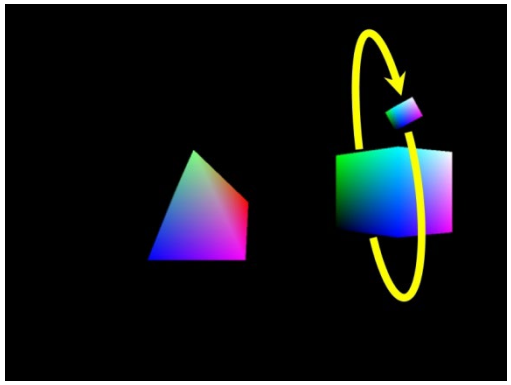
sun's rotation is pushed on the stack, and popped after drawing it



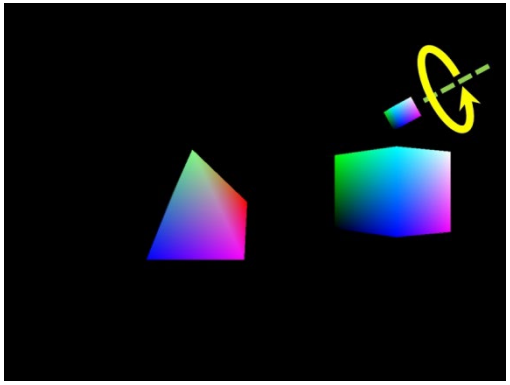
planet's revolution around the sun is pushed on the stack (but not popped)



planet's rotation is pushed on the stack, and popped after drawing it



moon's revolution around earth is pushed on the stack (but not popped)



moon's rotation is pushed on the stack

- At each drawing step, the matrix on top of the stack serves as the MV matrix.
- At the end of display(), the remaining matrices are popped off of the stack.

```
stack<glm::mat4> mvStack;
void display(GLFWwindow* window, double currentTime ) {
    . . .
    vMat = glm::translate(glm::mat4(1.0f), glm::vec3(-cameraX, -cameraY, -cameraZ));
    mvStack.push(vMat);       // push view matrix onto the stack
    . . .
    // ----------------------  pyramid == sun  ----------------------------------------
    mvStack.push(mvStack.top());
    mvStack.top() *= glm::translate(glm::mat4(1.0f), glm::vec3(0,0,0));     // sun position
    mvStack.push(mvStack.top());
    mvStack.top() *= glm::rotate(glm::mat4(1.0f), (float) currentTime, glm::vec3(1,0,0)); // sun rotation
    glUniformMatrix4fv(mvLoc, 1, GL_FALSE, glm::value_ptr(mvStack.top()));   // pass MV to shader
    glDrawArrays(GL_TRIANGLES, 0, 18);            // draw the sun
    mvStack.pop();              // remove the sun's axial rotation from the stack (but not its translation)
    //----------------------  cube == planet  ----------------------------------------
    mvStack.push(mvStack.top());
    . . .    // planet's revolution around sun goes here.  Also push a matrix for planet's axis rotation
    . . .    //  then glUniformMatrix4fv to pass MV matrix to shaders, etc.
    glDrawArrays(GL_TRIANGLES, 0, 36);            // draw the planet
    mvStack.pop();              // remove the planet's axial rotation from the stack (but not the translation)
    //----------------------  smaller cube == moon  ---------------------------------
    . . .     // moon's revolution around planet, and rotation on its axis, go here, similar as for planet and sun
    glDrawArrays(GL_TRIANGLES, 0, 36);            // draw the moon
    //  remove moon scale/rotation/position, planet position, sun position, and view matrices from stack
    mvStack.pop();  mvStack.pop();  mvStack.pop();  mvStack.pop();
}
```

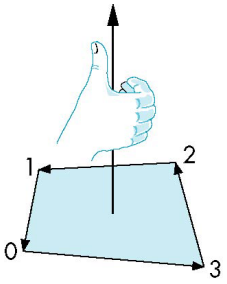
Backface Culling

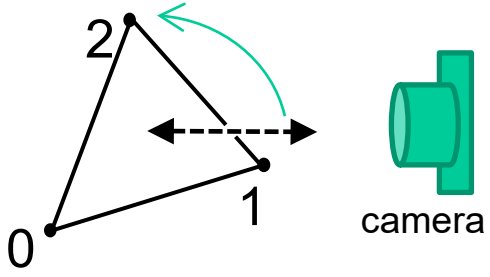
- Useful especially for entirely “closed” objects (interior is never visible)
- Backface culling saves time by not rendering faces that are hidden
- Front faces are determined by the “winding order”
- Winding order can be “clockwise” or “counter clockwise”, as viewed by the OpenGL camera
- Face culling is enabled by **glEnable(GL_CULL_FACE)**
- Winding order set by **gl_FrontFace(GL_CW)** or **gl_FrontFace(GL_CCW)**
- Can cull front or back faces, e.g.: **glCullFace(GL_BACK)**
- The default winding order is CCW.

For example, the triangle below has vertices 0, 1, 2 as shown.

If the following are set:

```
glEnable(GL_CULL_FACE);
gl_FrontFace(GL_CCW);
gl_CullFace(GL_BACK);
```

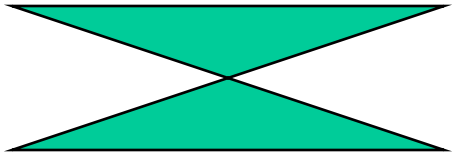




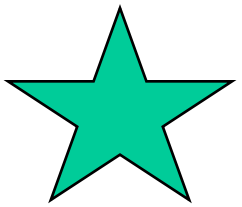
then this triangle is rendered only if its vertices progress in a counter clockwise direction, as viewed by the OpenGL camera.

Polygon Issues

- OpenGL will only display triangles
  - Simple: edges cannot cross
  - Convex: All points on line segment between two points in a polygon are also in the polygon
  - Flat: all vertices are in the same plane
- Application program must tessellate a polygon into triangles (triangulation)
- OpenGL 4.1 contains a tessellator



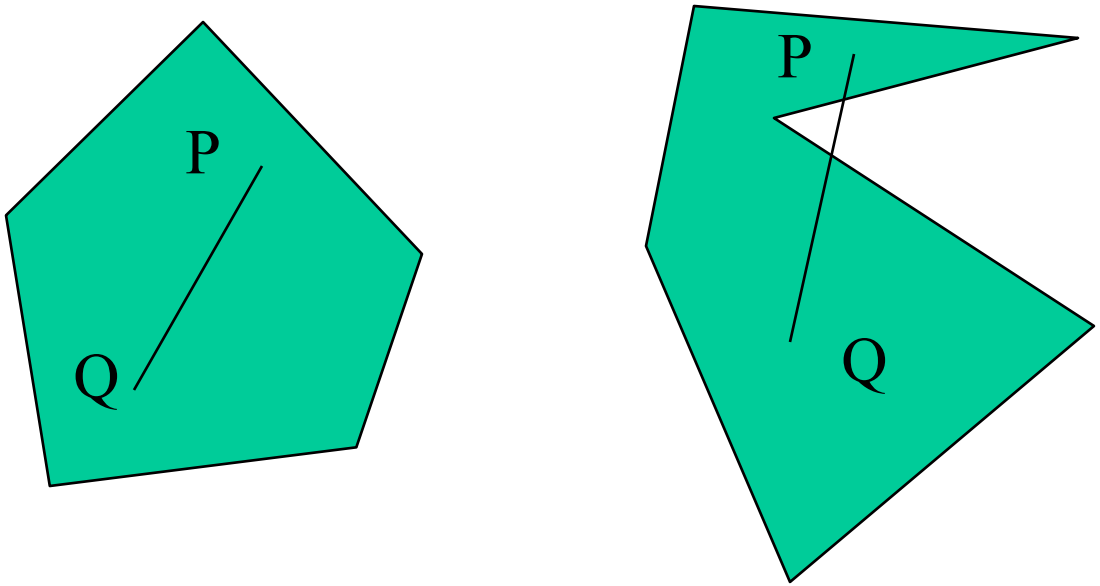
nonsimple polygon



nonconvex polygon

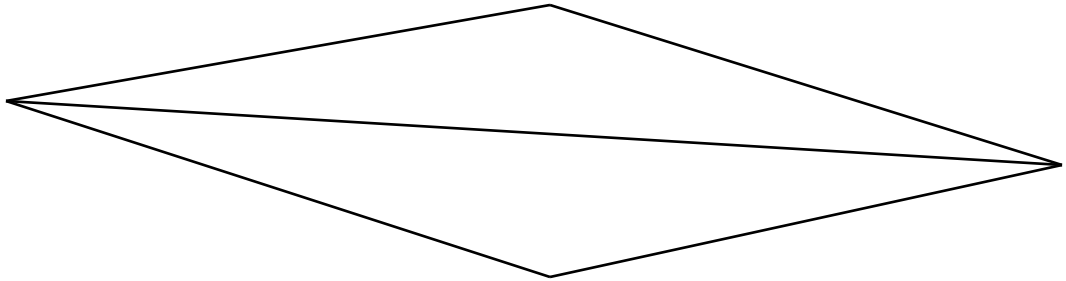
Convexity

- An object is *convex* iff for any two points in the object all points on the line segment between these points are also in the object



Good and Bad Triangles

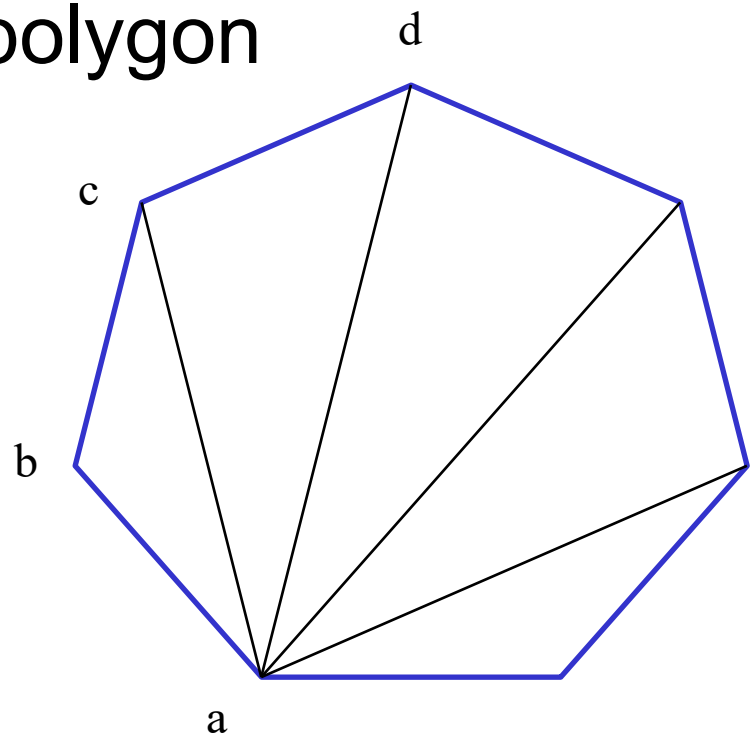
- Long thin triangles render badly



- Equilateral triangles render well
- Maximize minimum angle
- Delaunay triangulation for unstructured points

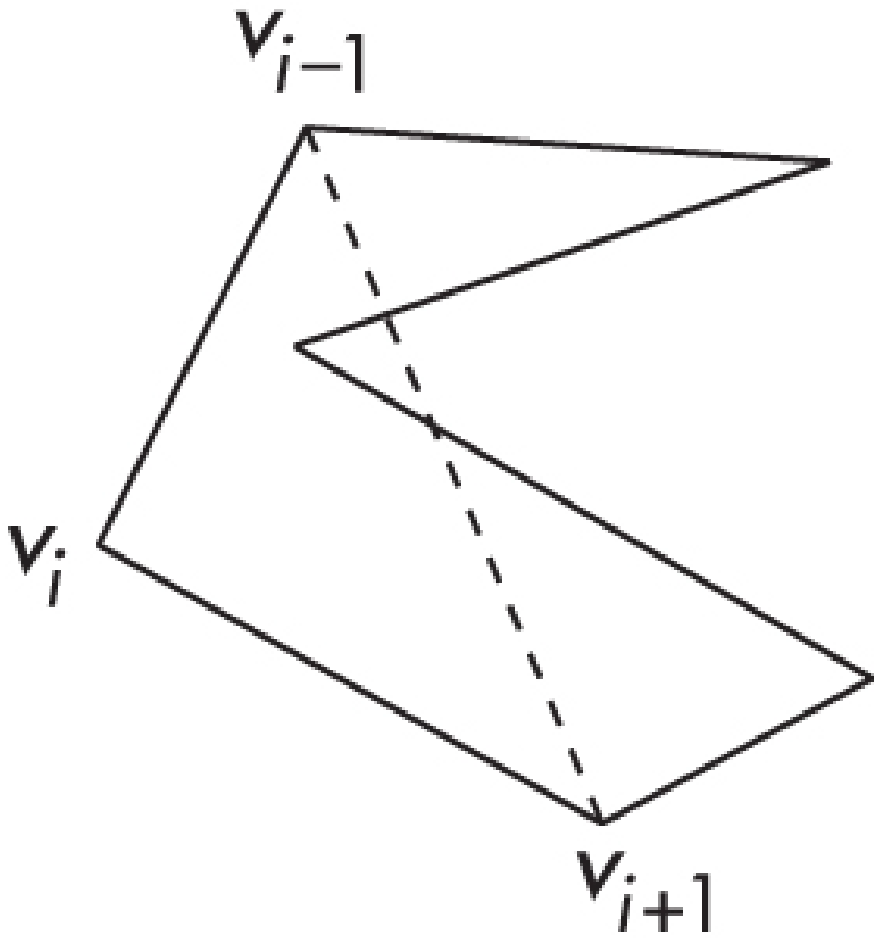
Triangularization

- Convex polygon



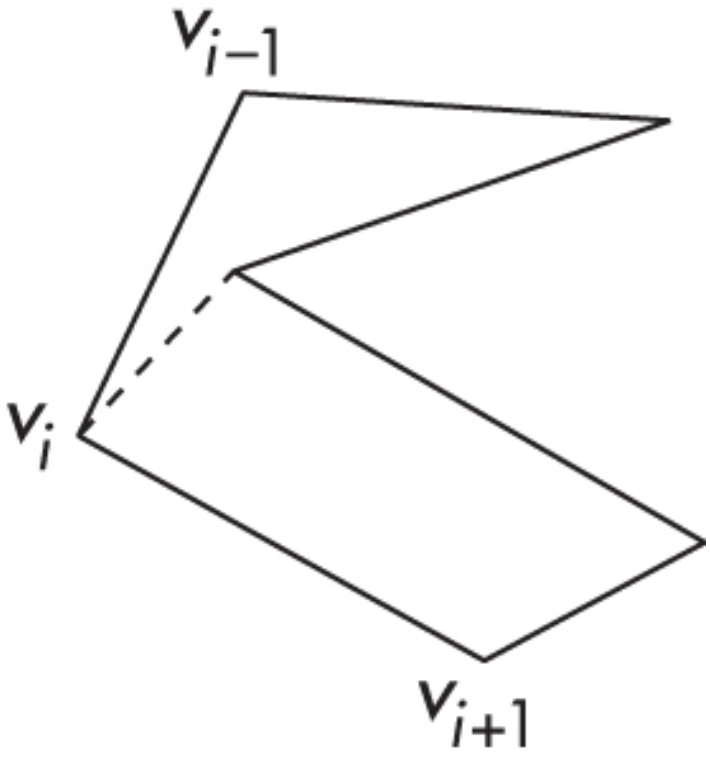
- Start with abc, remove b, then acd, ….

Non-convex (concave)



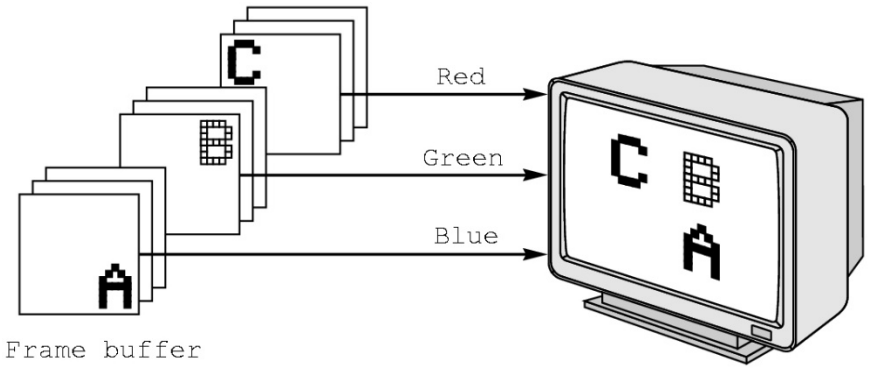
Recursive Division

- Find leftmost vertex and split



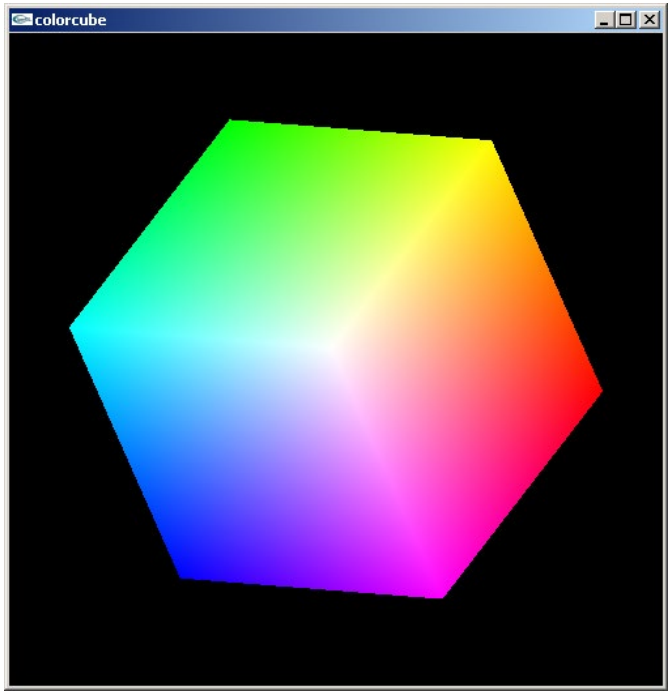
RGB color

- Each color component is stored separately in the frame buffer
- Usually 8 bits per component in buffer
- Color values can range from 0.0 (none) to 1.0 (all) using floats or over the range from 0 to 255 using unsigned bytels



Smooth Color

- Default is *smooth* shading
  - OpenGL interpolates vertex colors across visible polygons
- Alternative is *flat shading*
  - Color of first vertex determines fill color
  - Handle in shader



Setting Colors

- Colors are ultimately set in the fragment shader but can be determined in either shader or in the application
- Application color: pass to vertex shader as a uniform variable or as a vertex attribute
- Vertex shader color: pass to fragment shader as varying variable
- Fragment color: can alter via shader code