Comp 410/510

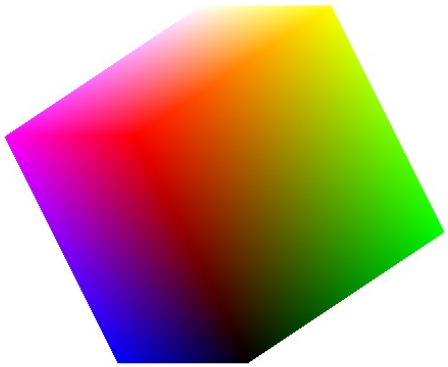
Computer Graphics

Spring 2023

# Programming with OpenGL
# Part 4: Three Dimensions

# Objectives

- Develop a bit more sophisticated 3D example
  - Rotating cube
- Introduce interaction
- Introduce hidden-surface removal
- Introduce transformations

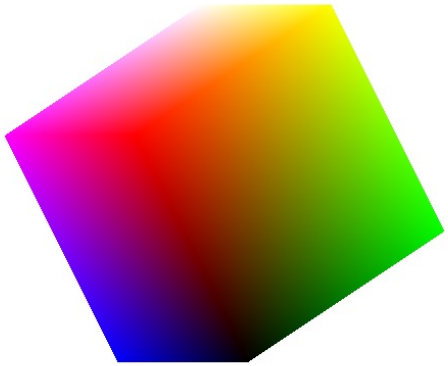


# Three-dimensional Applications

- In OpenGL, two-dimensional applications are a special case of three-dimensional graphics
- Going to 3D
  - Not much change
  - Use `vec3`, `glUniform3f`
  - Have to worry about the order in which primitives are rendered or use hidden-surface removal

# Example

- Rotating cube (see the spincube code)
- Create a 3D cube with different colors assigned to corners
- Need to rotate (continuously)
- Need to set up transformation (“modelview”) matrix
- We will also see briefly how to set up projection matrix
- A bit more about vertex shader
- Hidden surface removal

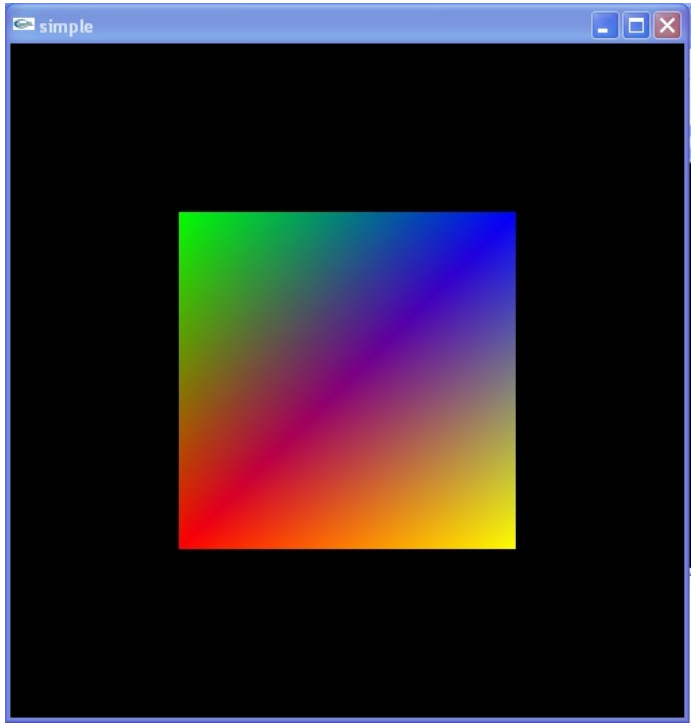


# Adding Color

- Colors are ultimately set in the fragment shader but can be determined in either shader or in the application
- If we set a color in the application, we can send it to the shaders as a vertex attribute or as a uniform variable depending on how often it changes
- In this example, associate a color with each vertex as an attribute

# Smooth Color

By default, OpenGL interpolates vertex colors across visible triangles (rasterization)



# Setting Colors

Set up a color array of same size as positions:

```
typedef vec3 color3;

color3 colors[NumVertices];
vec3 points[NumVertices];

//loop for setting positions and colors
colors[i] = ...
points[i] = ...
```

# Setting Up Buffer Object

Send color array to GPU using a VBO along with positions:

```
//need larger buffer
glBufferData(GL_ARRAY_BUFFER, sizeof(points)+ sizeof(colors),
       NULL, GL_STATIC_DRAW);

//load data separately
glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(points), points);
glBufferSubData(GL_ARRAY_BUFFER, sizeof(points),
       sizeof(colors), colors);
```

# Set Two Vertex Attribute Arrays

```
// vPosition and vColor identifiers in vertex shader

loc = glGetAttribLocation(program, "vPosition");
glEnableVertexAttribArray(loc);
glVertexAttribPointer(loc, 3, GL_FLOAT, GL_FALSE, 0,
    BUFFER_OFFSET(0));

loc2 = glGetAttribLocation(program, "vColor");
glEnableVertexAttribArray(loc2);
glVertexAttribPointer(loc2, 3, GL_FLOAT, GL_FALSE, 0,
    BUFFER_OFFSET(sizeofpoints));
```

Vertex shader

```
in vec4 vPosition, vColor;
out vec4 color;
uniform mat4 ModelView, Projection;

void main()
{
    gl_Position = Projection * ModelView * vPosition;
    color = vColor;
}
```

# Modelview and Projection

- `ModelView` and `Projection` are uniform variables
  - 4x4 matrices
  - sent as input to vertex shader

- Code from `init()` function:

```
ModelView = glGetUniformLocation(program, "ModelView");
Projection = glGetUniformLocation(program, "Projection");
```

Vertex shader

```
in vec4 vPosition, vColor;
out vec4 color;
uniform mat4 ModelView, Projection;

void main()
{
    gl_Position = Projection * ModelView * vPosition;
    color = vColor;
}
```

# Setting up Modelview Matrix

- Have to set up in `display()` since the cube keeps rotating

```
void display( void )
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    //  Generate the model-view matrix
    const vec3 displacement( 0.0, 0.0, 0.0 );
    mat4  model_view = ( Translate( displacement ) *
                         RotateX( Theta[Xaxis] ) *
                         RotateY( Theta[Yaxis] ) *
                         RotateZ( Theta[Zaxis] ) );
    glUniformMatrix4fv( ModelView, 1, GL_TRUE, model_view );
    glDrawArrays( GL_TRIANGLES, 0, NumVertices );
}
```

- `Translate()`, `RotateX()`, `RotateY()`, `RotateZ()`: user-defined functions in `mat.h`
- `Theta` is a global array

# Setting up Projection Matrix

- Better to define in `init()` since it is often set only once
- Defult is orthographic but you can use perspective as well
- Use functions from `mat.h`

```
mat4  projection;
projection = Ortho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
//projection = Perspective( 45.0, 1.0, 0.5, 3.0 );
glUniformMatrix4fv( Projection, 1, GL_TRUE, projection );
```

# Main event loop

- Update the scene at each iteration and redraw

```
while (!glfwWindowShouldClose(window))
{
    update();
    display();
    glfwSwapBuffers(window);
    glfwPollEvents();
}
```

```
#include <gl3.h>
#include <glfw3.h>


void init(){
....
}
void display(){
...
}
void update(){
...
}
void mouse_button_callback (GLFWwindow* window, int button, int action, int mods){
......
}
void key_callback (GLFWwindow* window, int key, int scancode, int action, int mods){
......
}

int main()
{
  /* window intializations*/

  glfwSetKeyCallback(window, key_callback);
  glfwSetMouseButtonCallback(window, mouse_button_callback);

  init();

  while (!glfwWindowShouldClose(window)){
    update();
    display();
    glfwSwapBuffers(window);
    glfwPollEvents();
  }
}
```

# The mouse callback

- **`glfwSetMouseButtonCallback(window, mouse_button_callback)`**
- **`void mouse_button_callback (GLFWwindow* window, int button, int action, int mods)`**

- is returned
  - which button (**`GLFW_MOUSE_BUTTON_RIGHT, GLFW_MOUSE_BUTTON_LEFT, GLFW_MOUSE_BUTTON_MIDDLE`**) causes the event
  - action that was taken (**`GLFW_PRESS, GLFW_RELEASE`**)
  - any modifier keys that were pressed, such as **`GLFW_MOD_SHIFT`** or **`GLFW_MOD_CONTROL`**
    - e.g., **`(mods & GLFW_MOD_SHIFT)`** is true when the shift key is pressed

To get cursor position in the callback, you can use

```
void glfwGetCursorPos(GLFWwindow *window, double *xpos, double *ypos)
```

# Terminating a program

We can use a simple mouse callback function to terminate the program execution through OpenGL:

```
void mouse_button_callback(GLFWwindow* window, int button, int action, int mods)
{
    if (button == GLFW_MOUSE_BUTTON_LEFT && action == GLFW_PRESS)
    {
        exit(0);
    }
}
```

# Double Buffering

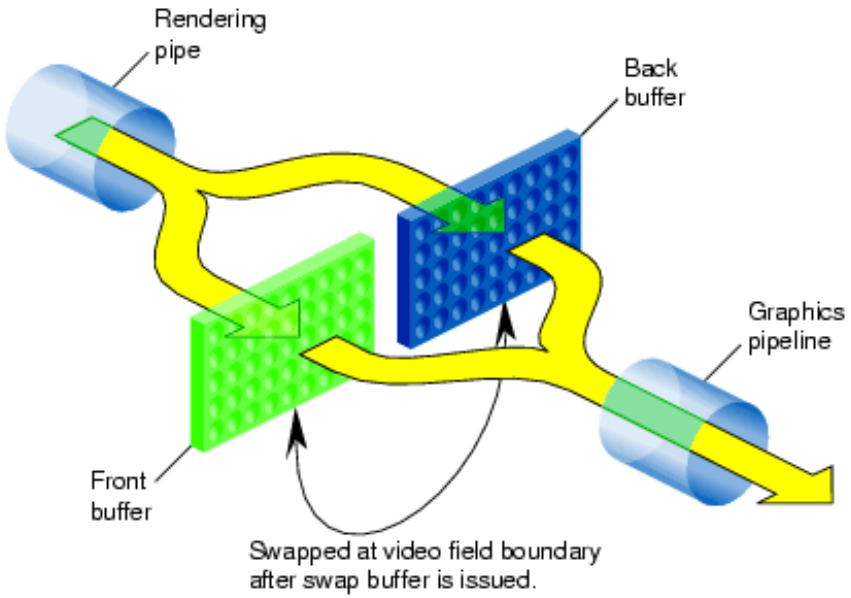
- In GLFW, the default framebuffer is a double-buffered framebuffer

- Swap buffers in the main even loop

```
while (!glfwWindowShouldClose(window))
{
    update();
    display();
    glfwSwapBuffers(window);
    glfwPollEvents();
}
```

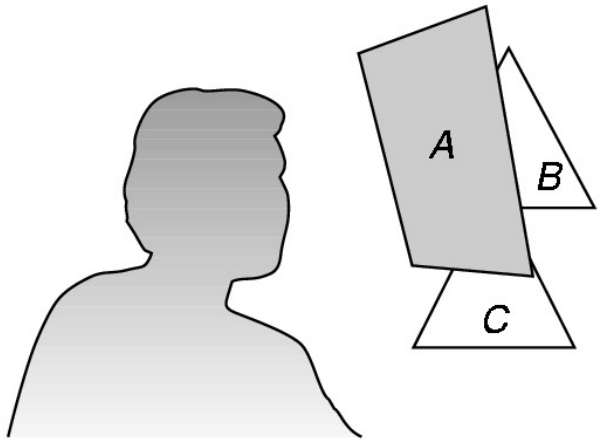
# Double Buffering

- Updating the value of a uniform variable opens the door to animation in an application
  - Execute `glUniform` in user-defined `update()` and change state
  - Force a redraw through user-defined `display()`
- Need to prevent a partially redrawn frame buffer from being displayed
- Draw into back buffer - Display front buffer
- Swap the buffers after drawing is finished



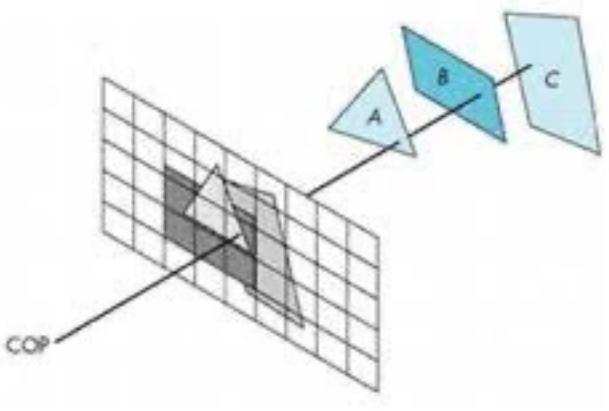
# Hidden-Surface Removal

- We want to see only those surfaces in front of other surfaces
- OpenGL uses a *hidden-surface* method called the z-buffer algorithm
- Saves depth information as objects are rendered so that only front objects appear in the image



# Hidden-Surface Removal

- We want to see only those surfaces in front of other surfaces
- OpenGL uses a *hidden-surface* method called the z-buffer algorithm
- Saves depth information as objects are rendered so that only front objects appear in the image
- Handled in fragment processor



# Using the z-buffer algorithm

- The algorithm uses an extra buffer, i.e., z-buffer, to store depth information as geometry travels down the pipeline
- In GLFW, the depth buffer is included by default in the default framebuffer of a window.
- Enabled in `init()`
    - `glEnable(GL_DEPTH_TEST)`
- Cleared in the `display()`
    - `glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)`