OpenGL Setup Tutorial

ECE 6397 (GPU Programming) – David Mayerich
Where does OpenGL fit into your application?

• OpenGL is a cross-platform API for hardware-accelerated 3D graphics
  • Data is passed to OpenGL in a vector graphics format
    – **vertices** representing triangles
    – **properties** for each vertex
  • Advanced rendering
    – texture mapping
    – lighting
    – vertex and fragment shading
OpenGL Setup

1. Create an **OpenGL Context**
   - stores all variables associated with an instance of OpenGL
   - **framebuffer**: rendering commands write data here
   - **viewport**: framebuffers are displayed here
   - **textures**: bitmaps and arrays
   - **of-screen buffers**: bitmap arrays where data can be rendered or read
     - textures can be buffers (and vice versa)
     - back-buffers are common (prevent the user from seeing things being drawn)
   - **shaders**: GPU programs that are run for each vertex or pixel

• Several cross-platform libraries can be used to create contexts
  - GLFW: [https://www.glfw.org/](https://www.glfw.org/) (recommended)
  - Qt: [https://www.qt.io/](https://www.qt.io/) (commercial development)
OpenGL Setup

2. Set up OpenGL Extensions (Windows only)
   – By default, only OpenGL 1.1 is exposed in Windows (current version is 4.6)
   – Very basic functionality – the features are there, just not loaded by default

• Extension Loading Libraries:
  – GL3W: [https://github.com/skaslev/gl3w](https://github.com/skaslev/gl3w)

3. Use OpenGL
   – After a context is created and extensions are loaded, you can use OpenGL
   – At this point – almost any online tutorial can help you
   – One exception: default settings can be platform independent
     • if something works on Windows but not Linux, it’s likely a different default setting
Libraries in Windows

• The necessary libraries (GLFW, GLEW) come with three types of files:
  – .h files (header files containing declarations and data types)
  – .lib files contain libraries that are linked with your code
  – .dll files contain runtime libraries

• Windows looks for DLLs in two different locations:
  – 64-bit DLLs are stored in /Windows/System32
  – 32-bit DLLs are stored in /Windows/SysWOW64

• In my opinion – the easiest thing for you to do is:
  1. Always compile 64-bit code
  2. Copy your DLLs into /Windows/System32
     • You need admin access to do this
     • Alternatively, you can put them in a directory and put it on your path
OpenGL Setup

• Create a project with CMake:

```cmake
project(MyProject)

# GLFW Window Manager
find_package(GLFW REQUIRED)

# GLEW (if necessary)
if(WIN32)
    find_package(GLEW REQUIRED)
endif(WIN32)

# find and set up OpenGL
find_package(OpenGL REQUIRED)

# set the include directories
include_directories(MyProject
    ${GLFW_INCLUDE_DIRS}
    ${OpenGL_INCLUDE_DIRS}
    ${GLEW_INCLUDE_DIR}
)

# specify the executable to create
add_executable(myproject
    main.cpp
    main.h
)

# set link libraries
target_link_libraries(myproject
    ${GLFW_LIBRARIES}
    ${OPENGL_gl_LIBRARY}
    ${OPENGL_glu_LIBRARY}
)

# link to GLEW if necessary
if(WIN32)
    target_link_libraries(myproject
        ${GLEW_GLEW_LIBRARY}
    )
endif(WIN32)
```
OpenGL Setup

• C/C++ code to create a window:

```c
#ifdef __WIN32__
    #include <GL/glew.h> // GLEW declarations for Windows
#endif
#include <GL/glfw3.h> // GLFW declarations
...
GLFWwindow* window; // pointer stores a global window
...
int main(int argc, char** argv){
    if(!glfwInit()) return -1; // initialize GLFW or return an error
    // create a window with a pre-determined size
    window = glfwCreateWindow(width, height, “Title”, NULL, NULL);
```
OpenGL Setup

...  
//create a window with a pre-determined size
window = glfwCreateWindow(width, height, "Title", NULL, NULL);

//create an OpenGL context (this can also apply to other APIs)
glfwMakeContextCurrent(window);

#ifdef __WIN32__
  GLenum err = glewInit();//initialize GLEW for Windows
  if(err != GLEW_OK)
    std::cout<<"GLEW Error: "<<glewGetErrorString(err)<<std::endl;
#endif
...
OpenGL Setup

At this point you have:
• an active window
• an active OpenGL context connected to that window
• access to up-to-date OpenGL features
Event Loops

- Applications that have graphical user interfaces (GUIs) have to constantly query for user input

- **event loop**: program loop that queries all possible input devices, and executes pre-directed commands based on that input (if any)

```python
function main
  init()
  while(!quit):
    for each input:
      input.callback()
```

- **callback function**: function that is executed if there is input
OpenGL Setup

• Add an event loop:

... 

while( !glfwWindowShouldClose(window) ){  
    display(); //function to render stuff  
    glfwPollEvents(); //check for any input  
} //end while  

//end GLFW (and the OpenGL context with it)  

glfwTerminate(); //terminate GLFW  
return 0; //normal return to the OS  
} //end main
OpenGL Setup

• Generate callback functions to provide input

... 

//set up a function to receive keyboard input
glfwSetKeyCallback(window, keyboard_callback);

//set up a function to receive mouse input
glfwSetMouseButtonCallback(window, mouse_button_callback);

while( !glfwWindowShouldClose(window) ){ //event loop
    ...
}

} //end main function
OpenGL Setup

```c
void key_callback(GLFWwindow* window, int key, int scancode, int action, int mods) {
    if (key == GLFW_KEY_RIGHT && action != GLFW_RELEASE) {
        right_arrow(); // do something with the right arrow
    } else if (key == GLFW_KEY_LEFT && action != GLFW_RELEASE) {
        left_arrow(); // do something with the left arrow
    }
}

void mouse_button_callback(GLFWwindow* window, int button, int action, int mods) {
    if (button == GLFW_MOUSE_BUTTON_LEFT && action == GLFW_PRESS) {
        double xpos, ypos;
        // get the position of the mouse pointer
        glfwGetCursorPos(window, &xpos, &ypos);
    }
}
```
Double Buffering

• The default framebuffer is constantly displayed on the screen
  – refresh is based on the refresh rate of the monitor
  – possible (and likely) to update the framebuffer while the screen is refreshing

• Prevent tearing, stutter, etc. with double-buffering

```python
allocate front_buffer, back_buffer
display_buffer = front_buffer
while(!quit): //event loop
    render(back_buffer) //write to back_buffer
    swap(back_buffer, front_buffer) //swap buffers
```
OpenGL Setup

• Perform buffer swapping:

```c
while( !glfwWindowShouldClose(window) ){ // event loop
    display(); // function to render stuff
    glfwSwapBuffers(window); // swap double-buffer
    glfwPollEvents(); // check for any input
}
```

// end GLFW (and the OpenGL context with it)
glfwTerminate(); // terminate GLFW
return 0; // normal return to the OS
Setting Up a Drawing Environment

• Set up a viewport:
  
  `glViewport( int x, int y,  
                     int width, int height)`
  
  – specifies an affine transformation from device coordinates to the window
  – \((x, y)\) is the lower-left corner of the window
  – width and height specify the viewport size in pixels

• Set up a projection:
  
  `glOrtho( double left, double right,  
                        double bottom, double top,  
                        double near, double far)`
  
  – specifies an orthographic projection
  – the coordinates give the edges of a cube used as the view volume
Clearing the Screen

• Set a clear color:
  
  `glClearColor( float r, float g, float b, float a )`
  
  – specifies the color used to fill the frame buffer when a clear command is sent

• Clear the frame buffer

  `glClear ( int mask )`
  
  – sets all pixels in the specified bitplane of the frame buffer to the clear color
  – GL_COLOR_BUFFER_BIT clears the color bitplane
  – GL_DEPTH_BUFFER_BIT clears the depth bitplane (discussed later)
Setting Up Matrices

• Select a matrix to update:
  
glMatrixMode( GLenum mode )
  – sets the current matrix stack
  – GL_MODELVIEW model and view matrix (most commonly updated)
  – GL_PROJECTION stores the projection
  – GL_TEXTURE manipulates texture coordinates

• Push a matrix onto the currently selected stack
  
g1Push( )

• Pop a matrix off of the currently selected stack
  
g1Pop( )

• Replace the current matrix with the identity matrix
  
g1LoadIdentity( )
Basic Viewport Example

```c
window = glfwCreateWindow(width, height, "Title", NULL, NULL);
...

glViewport(0, 0, width, height); //create a viewport

glMatrixMode(GL_PROJECTION);   //update the projection stack
glLoadIdentity();            //set projection to I
glOrtho(-1.0, 1.0, -1.0, 1.0, 0, 1);  //create an ortho projection

//update the modelview stack

//initialize to I

...  //set a clear color

glClearColor(0.0, 0.0, 0.0, 0.0);  //set a clear color

...  //clear the color buffer and depth buffer

//clear the color buffer and depth buffer

GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

...  //draw the scene based on user input

//enter the event loop

draw_scene();
```