WHERE TO START?

PUSHPAK KARNICK

CS 300: ASSIGNMENT #1

FIRST STEPS

- Download the framework off of Distance
- This presentation is included with it
- Even if you are writing your own framework, it is a great reference to getting started.
- It also has a sample executable on (roughly) how your assignment 1 should behave.

CONTACT INFO

- Instructor: Pushpak Karnick
  - Office: 2nd floor, extension #4413
  - Email: pushpak.karnick@digipen.edu
    - Include “[CS300]” as start of subject line
- TA: Information provided if available

BRIEF INTRODUCTION TO ASSIGNMENT 1

- Create an application that supports OpenGL 3
- Use VBOs and IBOs for rendering meshes
- Can load and render Wavefront OBJ files
- Can render debug normals for vertices and faces
- Compute vertex and face normals for meshes
- Have basic vertex-based directional lighting
- Have support for 2 lights
- Lighting model supports ambient and diffuse
- Support translation and rotation of the model
- Some sort of GUI support to control these features
- Anything else missing will be covered on the rubric
- Perspective projection matrix correctly implemented

FRAMEWORK

- Reminder: you can write your own
- Rest of the presentation covers the framework
- Also covers some necessary OpenGL 3 concepts
- Begin with extracting the framework, going into premake folder, and running build.bat.
- Creates the Visual Studio 2013 solution
- No support for earlier versions due to C++11

FRAMEWORK (2)

- Open up the solution and start in Main.cpp
- Entry point to the application
- Primary place where you will be doing work
- Throughout source code there are lines starting with: `// TODO(student):` and `// TODO:`
  - `// TODO(student)` refers to code you must implement in order to meet the requirements.
  - `// TODO` refers to code you may implement, but do not necessarily need to.

FRAMEWORK (3)

- Check out these files (the number in parentheses refers to the number of TODO(student) lines in that file):
  - src/math/Matrix.cpp (2)
  - src/graphics/MathFunctions.cpp (2)
  - src/graphics/MeshLoader.cpp (1)
  - src/graphics/TriangleMesh.cpp (1)
  - src/Main.cpp (20)
- Nearly the entire framework is documented and commented
- The math library is from Zero engine
  - Not documented in the same format

GETTING STARTED

C++ 11 PRIMER

C++ 11 FEATURES

- C++11 is used throughout the framework, but nothing very complicated
- You may encounter the following features:
  - std::shared_ptr and std::unique_ptr
  - Initializer lists
  - Enum classes (enum class)
  - Unordered containers
  - Deleting methods
  - override
  - auto

SHARED POINTERS AND UNIQUE POINTERS

- `std::shared_ptr` and `std::unique_ptr`
  - Shared and unique pointers are smart pointers
  - They essentially wrap around pointers and reference count them
  - When the reference count is up, they are deleted
- Initializer lists
  - Allow easy creation of STL containers, e.g.:
  - `std::vector<int> v = { 1, 2, 3, 4 };`

ENUM CLASSES

- Enum classes are very similar to regular enums, just with a few extra restrictions (see resources for more)
- Unordered containers (such as std::unordered_map)
  - Hashed data structures
  - Interface extremely similar to counterparts, e.g. std::map for the one above

DELETING METHODS

- Deleting methods
  - `MyClass(MyClass const &) = delete;`
  - Disallows the method from being implemented
  - Framework uses to disallow copying an object
- `override`
  - Specifies a virtual method in a base class is overriding one of the same name from the parent.
  - Does extra checking to verify that it worked
- `auto`: Make the compiler automatically infer the type of a variable.

C++ REFERENCES

- References:
- http://cplusplus.com
- http://www.codeproject.com/Articles/570638/Ten-Cplusplus-Features-Every-Cplusplus-Developer (covers some of these)
- http://en.wikipedia.org/wiki/Unordered_associative_containers_%28C%2B%2B%29

ASSIGNMENT 1 FEATURES

OPENGL

OPENGL FEATURES IMPLEMENTED IN THE FRAMEWORK

- Enabling/disabling GL states
- Clearing buffers on the screen framebuffer
- Changing the buffer clear color
- Managing Vertex Buffer Objects
- Managing Index Buffer Objects
- Managing Vertex Array Objects
- Managing GLSL shader programs
- Setting up vertex input layouts for the shaders
- Assigning values to shader uniform constants
- Rendering meshes in indexed drawing mode
- Most of these features will be covered shortly

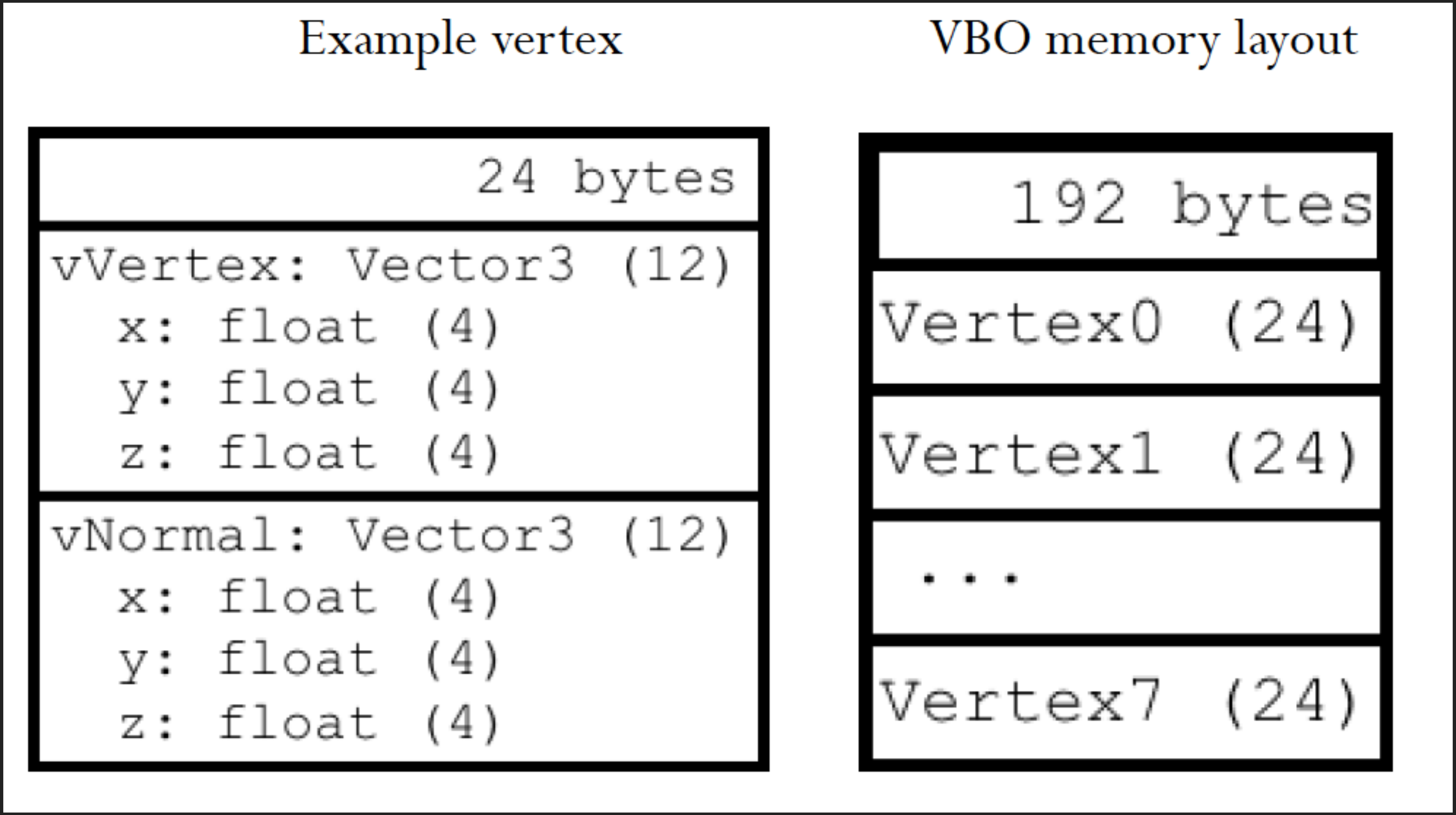
OPENGL FUNCTIONALITY

- We have already covered OpenGL basics in an earlier lecture.
- Now, we will be covering Vertex Array Objects, buffer objects and types, vertex input layouts, and interacting with shader programs.
- All of this functionality is implemented and thoroughly commented in the framework.

VERTEX BUFFER OBJECTS

- Recall how OpenGL buffers work from your last lecture.
- VBOs are a special type of buffer designed to store vertex information and attributes.
- Each vertex stored in the buffer is stored subsequently in a contiguous fashion.
- The next slide shows a sample vertex structure and, roughly, what the memory may look like for the VBO containing those vertices.
- Refer to `VertexBufferObject.h/.cpp` for more information and the syntax of needed OpenGL calls.

VBO STRUCTURE



Example vertex

```
24 bytes
vVertex: Vector3 (12)
  x: float (4)
  y: float (4)
  z: float (4)
vNormal: Vector3 (12)
  x: float (4)
  y: float (4)
  z: float (4)
```

VBO memory layout

```
192 bytes
Vertex0 (24)
Vertex1 (24)
...
Vertex7 (24)
```

INDEX BUFFER OBJECTS (IBO'S)

- Very similar to VBOs, except they store indices that are used to lookup vertices inside the VBO.
- This is referred to as indexed drawing or element drawing.
- We end up saving a lot of memory by having a separate buffer which represents the shape by simply storing indexes to the vertex, rather than the whole vertex itself.
- IBOs can store meshes of all sorts of primitives (triangles, lines, points, etc.)
- Refer to IndexBufferObject.h/.cpp for more information and syntax of the needed OpenGL calls.

VBO + IBO = RENDERING

- Once you have a VBO and IBO for a mesh created, you can render it.
- We will be assuming triangles are stored in the IBO (so 3 indices per face) and using the same vertex structure shown before.
- The following slide contains the code needed to render with those objects (it is stripped of GL conventions to keep it shorter).

RENDERING CODE

```
BindBuffer(ARRAY_BUFFER, vboHandle);
BindBuffer(ELEMENT_ARRAY_BUFFER, iboHandle);
// vertex layout: covered in greater detail later
EnableVertexAttribArray(0);
EnableVertexAttribArray(1);
// enable vVertex attribute (0)
VertexAttribPointer(0, 3, FLOAT, FALSE, sizeof(Vertex), 0);
// enable vNormal attrib. (1), offset after vVertex
VertexAttribPointer(0, 3, FLOAT, FALSE, sizeof(Vertex), (void *)sizeof(Vector3));
// draw 12 triangles
DrawElements(TRIANGLES, 12, UNSIGNED_INT, 0);
// disable vertex attribs
// unbind buffers
```

PROGRAMMABLE PIPELINE FEATURES

SHADERS

SHADERS

- Shaders are programs that assist in the transformation and rasterization pipelines on the GPU.
- In fact, almost the entire transformation pipeline in OpenGL 3 is handled manually in a vertex shader.
- Shader programs consist of one vertex shader and one fragment shader.
- The vertex shader takes in vertex attributes based on the vertex input layout (see VBO/IBO rendering code for more), transforms it to NDC space, and sends it to the fragment shader.
- The fragment shader is responsible for outputting a color value for that particular fragment/pixel.

SHADERS (2)

- This entire process is very similar to code written in CS200 and CS250, except it's in GLSL (similar to C).
- Shaders also have uniform constants
  - Variables that do not change throughout the runtime of a shader
  - Values are controlled via glUniform calls
  - See ShaderProgram::SetUniform() for more information
  - Overall, data passed to a shader must come from vertices or from uniforms
- Refer to ShaderProgram.h/.cpp for much more detail on how to create these OpenGL objects.

SHADERS AND VBO'S

- IBOs do not matter with shaders; the process of looking up vertices using indexes from an IBO is all handled internally.
- We must bind the shader before we can render VBOs.
- Recall earlier that we had to manually enable portions of the vertex buffer in order to draw (vertex input layout).
- This was based on the vertex structure example from earlier.
- The following GLSL code inside a vertex shader corresponds to the EnableVertexAttribArray calls from earlier:

VERTEX ATTRIBUTES

- layout(location = 0) in vec3 vVertex;
- layout(location = 1) in vec3 vNormal;
- // See assets/shaders/shader.vert for
- // more information on how these are used
- // and what they exactly mean.

- // vertex layout: covered in greater detail later
- EnableVertexAttribArray(0);
- EnableVertexAttribArray(1);
- // enable vVertex attribute (0)
- VertexAttribPointer(0, 3, FLOAT, FALSE, sizeof(Vertex), 0);
- // enable vNormal attrib. (1), offset after vVertex
- VertexAttribPointer(0, 3, FLOAT, FALSE, sizeof(Vertex), (void *)sizeof(Vector3));

VERTEX ARRAY OBJECTS (VAO'S)

- VAOs are not necessary to complete assignment 1, but the framework uses them.
- OpenGL 3 construct that allows n VBOs, up to 1 IBO, and the vertex input layout to all be stored within one data structure.
- For information on what OpenGL code is needed to create a VAO, see VertexArrayObject.h/.cpp.
- Rendering VAOs is trivial compared to before:

RENDERING USING VAO'S

- // shader program should already be bound
- BindVertexArray(vaoHandle);
- // draw 12 triangles using the IBO and VBO
- // of this VAO (and vertex input layout)
- DrawElements(TRIANGLES, 12, UNSIGNED_INT, 0);
- BindVertexArray(0);

ASSIGNMENT 1

USEFUL ALGORITHMS

COMPUTING FACE NORMALS

- Iterate through all triangles in the mesh
- Take two adjacent edge vectors that are starting from the same vertex and cross them.
- The cross product, normalized, represents the face normal.

COMPUTING VERTEX NORMALS

- More complicated process
- A vertex normal is the average all the face normals for each face that contains that vertex.
- There is an issue with this process: two polygons that exist on the same plane and share a vertex will end up double-affecting that vertex, which is incorrect.
- Imagine a triangulated cube; each face will have a situation where 2 triangles touch the same vertex and will produce this situation.
- This problem is solvable by keeping track of all face normals for each vertex and ensuring the same direction is not added to the vertex twice.

WAVEFRONT OBJ FILE FORMAT

- Rather than documenting the format within this presentation, the Wikipedia link on the format is extremely thorough and helpful:
- http://en.wikipedia.org/wiki/Wavefront_.obj_file
- Some implementation details:
  - You should parse vertex normals and vertex textures, but you do not need to do anything with the data (basically skip the lines if nothing else).
  - You should support parsing and skipping comments
  - You do not need to support any of the versions of the face line with slashes in it; the standard 'f v1 v2 v3' style format is fine.
  - You do not need to support parsing non-triangle faces
  - You do not need to support parsing materials

PREMAKE, IMGUI, GLEW & FREEGLUT

TOOLS

PREMAKE

- Premake is a meta-build system used to data-drive actual project files.
- It greatly simplifies dealing with projects, since we no longer need to deal with merging Visual Studio solutions, project files, etc.
- The script files are defined in Lua
- Used in this framework to keep everything normalized and minimalized.
- Similar to CMake, but more cross-platform and portable.
- See https://bitbucket.org/premake/premake-dev/wiki/Home for more.

IMGUI

- https://github.com/ocornut/imgui
- ImGui is a relatively new GUI library which is designed around the concept of immediate rendering.
- This means that if you suddenly desire adding some sort of GUI tweaking support anywhere in your code, you can do it right in that location.
- The GUI is recreated every frame and little to no state is stored within the GUI framework.
- This significantly shortens GUI code and, as a result, speeds up the process of building GUIs.

IMGUI (2)

- You are not required to use ImGui and are free to use any other GUI software you choose.
- From the TAs' experiences, though, ImGui is much faster to work with than AntTweakBar or WxWidgets.
- For excellent examples on using ImGui, see: https://github.com/ocornut/imgui/blob/master/imgui.cpp#L6962 (ShowTestWindow())

GLEW - GL EXTENSIONS WRANGLER

- http://glew.sourceforge.net/
- OpenGL is made up of iterations of changes, called extensions.
- Major versions, if supported, guarantee certain extensions are functional (e.g. OpenGL 3.3 has guaranteed functionality if fully supported).
- GLEW has the terrible responsibility of dynamically loading all of the OpenGL functions from the driver's OpenGL DLL and verifying which extensions are supported.
- It can determine if a complete GL version is supported.

GLEW (2)

- Due to how OpenGL is supported on Windows (only OpenGL 1.1 is built into Windows), GLEW is basically necessary.
- Dealing with OpenGL extensions without a library like GLEW can be very painful.
- It’s easy to setup and compatible with most libraries that can create a GL context for you (GLUT, FreeGLUT, SDL, SFML, GLFW, etc.)

FREEGLUT

- A newer implementation of the original GLUT library.
- GLUT: GL Utility Toolkit
- http://freeglut.sourceforge.net/
- Contains a bunch of legacy GLU code, including generating interesting meshes during runtime.
- Also, contains windowing support, creating an OpenGL context, and attaching this context to a window; all functionality is cross-platform.

FREEGLUT

- Also, supports various messaging functions via callback mechanisms, just like original GLUT.
- It has some nice changes and functionality that were missing in GLUT.
- Fully compatible with OpenGL 3 and newer and works correctly with GLEW.
- Alternatives to using FreeGLUT: SDL2, SFML, and GLFW (there are other options that are platform-specific, too).

WRAPPING UP

IN SUMMARY

CHECKLIST

- Before turning in, make sure your application:
    - Follows all of the points mentioned in slide 4 (introduction to Assignment 1)
    - Make sure you have at least looked at and implemented the functionality for all TODO(student) in the framework
    - Make sure your application supports all the features demonstrated in the sample (see the sample's README)
- Do not submit the sample folder
- Remember to run clean.bat before archiving; build artifacts deduct points
- If you changed premake4.lua, please email the instructor/TA
- Be sure to read the framework's README and update it
- Submission name format: ***digipen.login_cs300_1.zip***

CONCLUSION

- Remember, if you have any questions, do not hesitate to email the instructor/TA.
- The framework is (relatively) brand new and is bound to have issues; if you are getting stuck debugging code you did not write, please email us immediately.
- You always have the option of writing your own framework, as well.
- Good luck!

THANK YOU

Pushpak Karnick