15-462 Project 1: Basic OpenGL

Release Date: Thursday, January 17, 2013

Due Date: Thursday, January 31, 2013, 23:59:59

Starter Code:
http://www.cs.cmu.edu/afs/cs/academic/class/15462-s13/www/project/p1.tar.gz

1 Overview

For this project, you will have the opportunity to familiarize yourself with basic OpenGL concepts and real-time graphics programming. You will implement basic camera and lighting functionality, and render a 3D mesh and a heightmap. You will find the *OpenGL Programming Guide* (Red Book) to be extremely helpful, as it covers nearly all topics needed for this assignment in great detail.

2 Description

Welcome to this exciting spring edition of 15-462! Course staff is happy to have you on board for the next five assignments! The assignments are designed to give you an opportunity to implement some of the theoretical ideas that are discussed in lecture and a chance to program graphics applications.

For all assignments, we will direct you towards resources for help, but if you need more clarification or have a question, the best place to go is Piazza, which can be found at

`https://piazza.com/cmu/spring2013/15462`

. For this particular assignment, the OpenGL redbook will be the most helpful resource.

We will be using OpenGL for almost all of the assignments, so this lab is an introduction to the library. Your task is to use OpenGL to render a simple 3D scene: a pool with animated water, in which you can move the camera around with mouse and keyboard input.

We will give you a scene with a model for our pool. The water will be represented as a heightmap. A heightmap is a function that defines a y-position for every point in the xz-plane. We will be using this to create an animated surface of water by calculating the positions of points in the xz-plane on each time step and constructing a polygonal mesh.

3 Submission Process and Handin Instructions

Failure to follow submission instructions will negatively impact your grade.

1. Your handin directory may be found at one of the two directories below, depending on which class you are enrolled in. Graduate students should be enrolled in 15-662 and undergraduates in 15-462.

   `/afs/cs.cmu.edu/academic/class/15462-s13-users/`*andrewid*`/p1/`.

   `/afs/cs.cmu.edu/academic/class/15662-s13-users/`*andrewid*`/p1/`.

All your files should be placed here. Please make sure you have a directory and are able to write to it well before the deadline; we are not responsible if you wait until 10 minutes before the deadline and run into trouble. Also, remember that you must run `aklog cs.cmu.edu` every time you login in order to read from/write to your submission directory.

2. You should submit all files needed to build your project, as well as any textures, models, shaders, or screenshots that you used or created. Your deliverables include:

   - `src/` folder with all `.cpp` and `.hpp` files.
   - Makefile and all `*.mk` files
   - `writeup.txt`
   - Any models/textures/shaders needed to run your code.

3. Please **do not** include:

   - The `bin/` folder or any `.o` or `.d` files.
   - Executable files
   - Any other binary or intermediate files generated in the build process.

   Run `make clean` before submitting. If you were using Visual Studio, be sure to clean the solution before submitting.

4. Do not add levels of indirection when submitting. For example, your makefile should be at `.../`*andrewid*`/p1/Makefile`, **not**
   `.../`*andrewid*`/p1/myproj/Makefile` or `.../`*andrewid*`/p1/p1.tar.gz`.
   Please use the same arrangement as the handout.

5. We will enter your handin directory, and run `make clean && make`, and it should build correctly. **The code must compile and run on the GHC 5xxx cluster machines**. Be sure to check to make sure you submit all files and that it builds correctly.

6. The submission folder will be locked at the deadline. There are separate folders for late handins, one for each day. For example, if using one late day, submit to `.../`*andrewid*`/p1-late1/`. These will be locked in turn on each subsequent late day.

4 Required Tasks

A very general overview of the implementation requirements is as follows. Refer to subsequent sections of the handout for more details. Figure 1 shows a screenshot of the reference implementation.

Input: The input to the program is a hardcoded scene containing a model, and heightmap, a light, a camera, and some position and material properties, along with basic functionality for loading a window, saving screenshots, moving the camera with the mouse and keyboard, and running a rendering loop.

Output: The output is a rendering of this scene, in addition to screenshots and `writeup.txt`. Your code must compute the normals for the model and heightmap.

For your program, you must:

- Correctly set the projection and modelview matrices based on the camera's values. The user should be able to move the camera around with the mouse and keyboard.
- Generate a mesh for the heightmap.
- Compute per-vertex normals for the model and heightmap's meshes.

- Use OpenGL fixed-functionality to add lights to the scene to correctly shade the scene with the given materials, setting appropriate ambient, diffuse and specular effects.
- Submit a few screen shots of your program's renderings.
- Use good code style and document well. We *will* read your code.
- Fill out `writeup.txt` with details on your implementation.

At a minimum, you must modify `opengl/project.cpp` and `writeup.txt`, though you may modify or add additional source files.

`writeup.txt` should contain a description of your implementation, along with any information about your submission of which the graders should be aware. Provide details on which methods and algorithms you used for the various portions of the lab. Essentially, if you think the grader needs to know about it to understand your code, you should put it in this file. You should also note which source files you edited and any additional ones you have added.

Examples of things to put in `writeup.txt`:

- Mention parts of the requirements that you did not implement and why.
- Describe any complicated algorithms used or algorithms that are not described in the book/handout.
- Justify any major design decisions you made, such as why you chose a particular algorithm or method.
- List any extra work you did on top of basic requirements of which the grader should be aware.

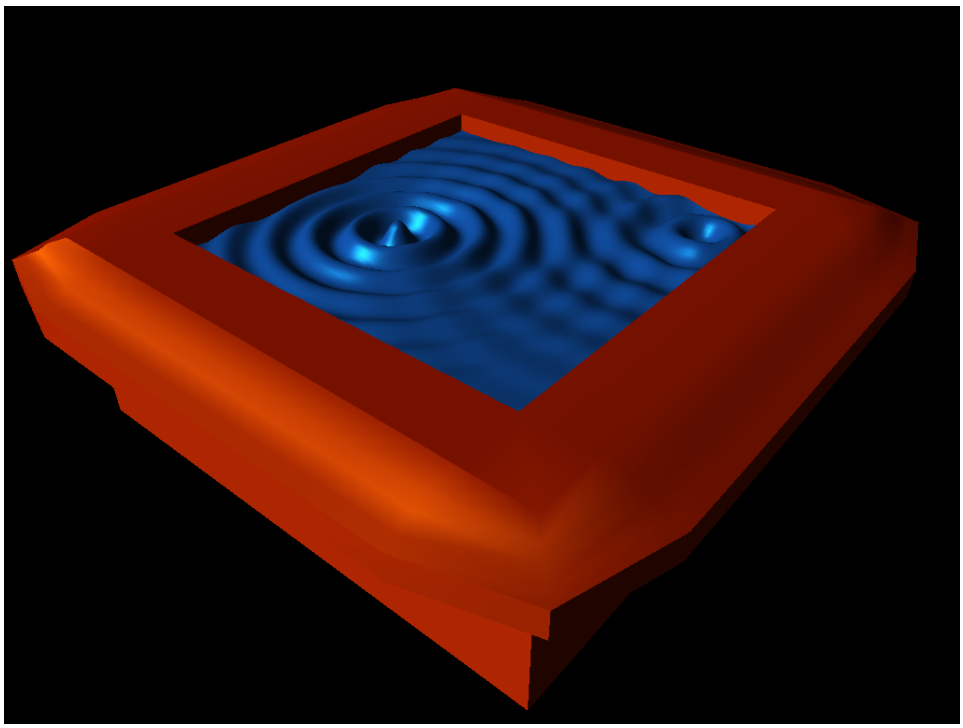

Figure 1: Example program output

5 Starter Code

It is recommended that you begin by first reviewing the starter code is provided. Though the amount you must edit is small, we are providing you with a large code base to get you started. You will need

to get familiar with it, since much of it will be used in future projects as well. The `README` gives a breakdown of each source file.

5.1 Programming Language

The code base is in C++. We know that this may be a new language to many of you, and so it is limited to C and a small subset of C++. In addition to C, the code uses classes, operator overloading, and a bit of the Standard Template Library, notably `std::string` and `std::vector`. If you are unfamiliar with these, your first task should be to learn them. The TAs are available for help.

5.2 Provided Functionality

The starter code provides a slew of functionality to get you started that is neither very interesting to write nor relevant for the course. You are free to modify any of it as you see fit, provided you do not break the application behavior. You may add new any new behavior if you wish. You will see the same code each project. Report any bugs to course staff. Here is a brief break down of what is provided; see the `README` for further details.

Matrix Algebra Structures and several helper functions for vectors, matrices, quaternions, and colors. Most basic functions you would need (e.g., dot product, matrix-vector multiplication) have been provided. Take a look before you write a basic math function to save yourself time. The function names and types are generally the same as those in the OpenGL Shading Language, which you will see in a future project.

Window Initialization/Event Handling SDL is used to initialize the window. The starter code creates a window and an OpenGL context, then runs a main loop that invokes update/render functions at a consistent framerate. SDL is also used to process keyboard and mouse events.

Mesh Loading A class that loads an OBJ mesh into a list of vertices and triangles. In this project, only positions are provided for each vertex. Not all OBJ files are supported; consult the code for more detail.

Image Saving/Loading Routines to save/load pngs into arrays of RGBA data. Also, the starter application code saves a screenshot when the '**f**' key is pressed.

5.3 Building and Running the Code

The code is designed to run and build on the Gates 5xxx machines and comes with a makefile. Consult the `README` for more detailed build and running instructions.

We have also provided a Visual Studio 2008 solution (works in Visual Studio 2010 Express too) , though it will take a bit of effort to get working since the programs have required command-line arguments. Note that there are differences in the compilers and graphics card across machines and thus the Windows solution might not be thoroughly tested for all possible machines. More details are in the `README`. If you use Windows, your project still **must** build and run on GHC Linux machines, so you will still have to test it on them before submitting. There are some differences in the compilers, so **code that compiles and works with Visual Studio may not compile or run correctly with GCC**. Make sure you test it well before the deadline. Be sure not to submit Windows binaries, either.

The program takes no arguments as the scene is hard-coded. Simply run the resulting executable.

5.4 What You Need to Implement

`opengl/project.cpp` contains some empty shell functions for you to fill in. At a minimum, you should implement the initialize, destroy, update, and render functions. Documentation for each function is in the source file. Feel free to modify any existing code or add new files, as long as you do not break the behavior of the program.

6 Grading: Visual Output and Code Style

Your project will be graded both on the visual output (both screenshots and running the program) and on the code itself. We will read the code.

In this assignment, part of your grade is on the quality of the visuals, in addition to correctness of the math. So make it look nice. Extra credit may be awarded for particularly good-looking projects.

Part of your grade is dependent on your code style, both how you structure your code and how readable it is. You should think carefully about how to implement the solution in a clean and complete manner. A correct, well-organized, and well-thought-out solution is better than a correct one which is not.

We will be looking for correct and clean usage of the C language, such as making sure memory is freed and many other common pitfalls. These can impact your grade. Additionally, we will comment on your C++-specific usage, though we will generally be more lenient with points (at least earlier in the semester). More general style and C-specific style (i.e., rules that apply in both C and C++) will, however, affect your grade.

Since we read the code, please remember that we must be able to understand what your code is doing. So you should write clearly and document well. If the grader cannot tell what you are doing, then it is difficult to provide feedback on your mistakes or assign partial credit. Good documentation is a requirement.

7 Implementation Details

The Red Book is essential for this project. The first few chapters provide all the information you need to render the scene using OpenGL. Consult the Red Book or online resources for descriptions of the API.

7.1 Update/Render Loop

The starter code runs an update/render loop at a fixed framerate. The update function is always invoked, but the render function may be skipped if the framerate is lagging. Thus, update and render may not correlate exactly. Your program must maintain a reasonable framerate.

7.2 Scene Representation

The scene is described by structs in `opengl/project.hpp`. The scene consists of a camera, a triangle mesh, and a heightmap. The triangle mesh and heightmap also both have an associated position, orientation, and scale. The following section describe each of these in more detail. Note that no material/lighting information is specified; you can use whatever you like for these. However, for all other values (including the camera), you must use all the given values.

7.3 Projection and Transformation

The camera class contains all information needed to position a camera in a scene. OpenGL matrix stack is used for transformations; consult the Red Book for more detail. The projection matrix typically contains the camera's field of view (FOV), aspect ratio, and near and far planes. You may find `gluPerspective` useful here. The modelview matrix contains the camera's position, direction, up vector, and the model's position, orientation, and scale. You may find `gluLookAt` useful here.

Note: `gluLookAt` takes a camera *target*, which is **not** the same as the camera *direction*. Read the documentation carefully.

7.4 Object Transformations

Both the mesh and heightmap come with a position, orientation and scale. These are defined as a vector, quaternion, and vector, respectively. Traditionally in OpenGL, the modelview matrix is used to store object transformations, which are applied after camera transformations. The Red Book describes this in detail, including using the modelview matrix to share camera transforms and objects transforms.

Note: `glScale` will cause normals to not be of unit length, which messes up lighting. See `GL_NORMALIZE` for details on fixing this problem.

7.4.1 About Quaternions

Quaternions are a compact, efficient way to represent 3-dimensional rotations. You do not need to understand the inner workings of quaternions, but for those interested, they parallel how complex numbers are used to represent rotations in 2 dimensions. The given implementation provides functions to convert to and from either a 3×3 matrix or a angle and rotation axis format. That should be all you need to use them for this and other labs.

7.5 Rendering Triangle Meshes

The `Mesh` has is a list of triangles and a list of vertices. A triangle is simply three indices into the vertex list. Vertices are a position in 3D space along with other attributes such as the normal. For this lab, you will actually be computing the normal yourself, so only position is provided.

The Red Book describes several methods for rendering a triangle mesh, and you may use any that you wish. If you are new to OpenGL, the course staff recommends using vertex arrays. Essentially, they work by putting all the primitive data into arrays and drawing the entire array with one function call. They are simple to use yet vastly more efficient than more basic methods (though still nowhere near the most efficient). As always, see the Red Book for more details.

Vertex Buffer Objects (VBOs) are a (slightly) more recent addition to OpenGL. As such, they are not covered in the Red Book and are not provided by the default OpenGL shared library. The starter code provided to you includes the GL Extension Wrangler (GLEW), which loads in all OpenGL extension functions that are supported by your graphics hardware. Though VBOs are an extension, they were released in 2003 and are therefore supported by virtually all graphics hardware.

A good overview of VBOs is available at http://www.songho.ca/opengl/gl_vbo.html. NOTE: The functions and constants exported by GLEW do *not* include the `ARB` suffixes. For example, use `glGenBuffers`, not `glGenBuffersARB` and `GL_STATIC_DRAW`, not `GL_STATIC_DRAW_ARB`.

More information about GLEW is available at http://glew.sourceforge.net/.

7.6 Computing the Heightmap Mesh

The heightmap will also be a mesh, only you must create it yourself. The `Heightmap` is an interface that provides a function to query the y position (height) at a given (x, z) position, from $(-1, -1)$ to $(1, 1)$. To render this, you must create a mesh from the heightmap by breaking it into discrete points on a grid, evaluating the heighmap at each point to get vertex positions, and connecting them into a mesh. The resolution of this mesh should be at least 64 vertices in each direction.

We actually provide you with an animated heightmap, so you'll have to recompute the mesh (and normals) in the update function. Most reasonable implementations will be more than fast enough, but take care that your program maintains a reasonable framerate.

Note: It is a poor idea to allocate new memory for this mesh every frame. Repeated allocations via `malloc` or `new` are *extremely* slow.

7.7 Buffers

The rasterized primitives are place in the color buffer, which is the final image placed on the screen. This color buffer is not cleared automatically each frame, and so you must do this manually. Look at `glClear` to accomplish this.

We must use the depth buffer to make sure the correct primitives are drawn. Closer primitives should be drawn on top of farther primitives. We can use OpenGL's depth test for this. You must enable this manually. As with the color buffer, the depth buffer is not automatically cleared.

7.8 Lighting the Scene

Once you render meshes correctly with the transformation matrices set correctly, you should be able to see a solid silhouette. The last task is to add some lighting to the scene to provide somewhat realistic looking shading.

The Red Book has entire chapters on lights and materials. We will only make you do the very basics for this assignment. Create at least one single light for your scene, and set some ambient, diffuse, and specular materials for you object in order to light it. You can also play with the color. Use different materials for each object.

7.9 Computing Normals

For lighting to work correctly, you will need normal data for your model. We intentionally do not load any normals from the model file; you must compute them yourself. You will be implementing a numerical approximation of the normal for each vertex:

1. Compute the normal for each triangle. Respect the winding order convention: the default for OpenGL is that primitives are in counter-clockwise order. You should normalize this vector.
2. Set the normal for a particular vertex to be the average of the normals of all triangles of which it is a part.

7.10 Suggested Sequence

We suggest you implement the assignment in the following order:

1. Create a test object to get basic rendering working.
2. Use the transformation matrices to account for the virtual camera and object transformations.
3. Render the given mesh without lighting/normals.
4. Use the matrices to follow objects' positions.
5. Compute normals for the mesh and add a light to the scene.
6. Create a mesh with normals from the heightmap and render it.
7. Create materials for each object.

8 Words of Advice

- Start early. Though this assignment is simple, it may take you time to become familiar with the OpenGL API and this style of programming. There are a lot of concepts introduced.
- Familiarize yourself with all the OpenGL concepts (transformation matrices, primitive rendering, lighting, materials, etc) before starting the assignment. The Red Book covers all these topics, as do many sources on the internet.
- Familiarize yourself with the structures in `math/` before starting, as they will be used throughout the semester. In particular, have a look at the `Vector` and `Camera` structures.
- Start with a simple object, even a hard-coded one like a triangle, to get something working.

- Make sure you have a submission directory that you can write to as soon as possible. Notify course staff if this is not the case.
- Experiment! Play around with OpenGL, even features that aren't part of the assignment. There is a lot you can do.
- While C has many pitfalls, C++ introduces even more wonderful ways to shoot yourself in the foot. It is generally wise to stay away from as many features as possible, and make sure you fully understand the features you do use.