Assignment 3: Shading and Geometric Modeling

Covers Lectures 5 and 6

CS 148 Autumn 2013-2014

Due Date: Oct. 14th

Introduction This assignment is about how to make your object look better. In this assignment, you are going to try your hand at generating triangular surface meshes of objects, writing GLSL shaders to render your objects in OpenGL and then implement a sub-division algorithm (Loop's Subdivision) to get a smoother surface.

We have already provided a sample code for loading the triangle mesh from the simple .obj file, generating simple surface meshes, shading the objects and implementing a sub-division algorithm.

Background

OBJ File The OBJ file format is a standard 3D object file format in ASCII supporting both polygonal and free-form geometry. The supported geometries include lines, polygons, and free-form curves and surfaces. Lines and polygons are represented by vertices and curves and surfaces are usually represented by control points. For the data structures used in geometry representation, you may need to review the slides in CS148. To know the complete grammar rules of OBJ format of you can refer to the wikipedia. In this assignment you are only required to parse a small part of these rules.

The OBJ file reader in our sample code is able to read a triangle mesh. An OBJ format file for triangles usually contains information of vertex locations (v), texture coordinates (vt), normals (vn), and faces (f). Here is an example:

```
# List of Vertices
v .0 .0 .0
v .0 .0 1.5
v 1.5 1.5 1.5

# Texture coordinates
vt .0 .0
vt .8 .0
vt .8 .5

# Normals
vn -.707 .707 .0

# Faces
f 1/1/1 2/2/1 3/3/1
```

This is a simplest OBJ file containing only one triangle. Comments lines begin with a hash mark (#). Blank space and blank lines can be freely added to the file to aid in formatting and readability. This file contains two types of data: vertex data (vertex locations, texture coordinates, and normals) and face data (triangles). The formats for vertex data are:

```
v x y z
vt u v
vn dx dy dz
```

Here v represents the location of a vertex, vt represents the texture coordinate, and vn represents the normal. Please notice that in OBJ file the number of elements for different vertex data components do not need to be the same, which means the number of vertices and the number of vertex normals (or textures) can be different. For example, in this file it has three vertices but only one normal (shared by all the three vertices). In this assignment we only require you to parse the token v for vertex locations, while vt for texture coordinates and vn for normals is not required.

The face data specify which element it will use for different components by defining a list of indices:

```
f v1 v2 v3
f v1/t1 v2/t2 v3/t3
f v1//n1 v2//n2 v3//n3
f v1/t1/n1 v2/t2/n2 v3/t3/n3
```

To describe a face, you need to provide the indices of its vertices in the vertex list (required) and indices for texture coordinates and normals (optional). The order of these three components should always be location, texture, and normal. Different components are split by slash. No spaces are permitted before or after the slash. Notice that the index always starts from 1 (be careful about this). There are four different formats to specify different components for a face. For this assignment, we only need vertex location(v) and face data(f). The sample code handles

```
v x y z
f v1 v2 v3
```

which is enough to read a simple .obj file. If you want to read a more complex .obj file, please modify the `STTriangleMesh::Read()` according to the information above.

Programmable Shaders We have already covered some aspects of GPU programming in the lectures, but we will cover the basics here as well. A modern GPU (Graphics Processing Unit) is responsible for computing the position of every vertex and the color of every fragment/pixel that it draws to the screen. The original definition of the OpenGL machine (as given in the Red Book) describes a particular way of computing these positions and colors that we call the *fixed-function pipeline*. To enhance or alter the fixed-function pipeline, we may insert small programs called shaders that will do the computation of vertex positions and fragment colors. In OpenGL, these programs can be specified in the C-like **OpenGL Shading Language (GLSL)**. You can find some information on GLSL in Chapter 7 of the Red Book. In addition, the OpenGL "Orange Book" is entirely about GLSL.

For this assignment, we will be using the following two types of GLSL programs:

- **Vertex shaders** allow us to determine the final position of each vertex we tell OpenGL to draw (among other things). The default OpenGL vertex shader simply multiplies the vertex coordinates by the modelview and the projection matrices. Normal tranformation/normalization/rescaling, lighting, and texture coordinate generation/transformation also occur at this point in the pipeline if they are enabled. If you have these features enabled and you replace the OpenGL's default vertex shader with a vertex shader of your own, you will need to re-implement them.
- **Fragment shaders** let us compute the color that will be written to each pixel. Textures are accessed and applied here.

Note that OpenGL also has a 3rd type of shader called a geometry shader. We will not require you to learn about these for the purposes of this assignment.

A GLSL program (i.e. *shader*) is defined by a function `main`(), just like a C program. The `main()` function of a vertex shader is responsible for computing the value of `gl_Position`, the final position of the

vertex. The `main()` function of a fragment program is responsible for computing the value of `gl_FragColor`, the final color of the fragment.

Note: The difference between 'fragments' and 'pixels' is a subtle one. Remember that geometry gets rasterized into fragments which are then composited onto pixels. There may, in general, be multiple fragments drawn to the same pixel (when things overlap). For this assignment, the difference isn't especially important, but we will do our best to use the terms correctly in our explanations.

A GLSL program can use simple C types like `int` and `float`, along with several built-in vector types: `vec2`, `vec3` and `vec4`. These vector types are a bit like the `STPoint`, `STVector` and `STColor` types all rolled into one. They support operations such as:

- Constructors – You can create a vec2 using an expression like `vec2(1.2, 3.4)`.
- Field access – You can get the components of a vec3 with `v.x, v.y, v.z`.
  - GLSL supports two variations on the field names: `x/y/z/w` and `r/g/b/a` are equivalent.
- "Swizzling" – You can extract the RGB part of an RGBA color `c` with `c.rgb`.
- Component-wise math – The standard addition, subtraction, multiplication and division operators operate component-wise on GLSL vectors.
  - Warning! Because GLSL uses the same type for vectors, points and colors, there are no restrictions on these operations like in `libST`.
  - In addition, many standard math functions (`sin()`, `sqrt()`, etc.) also operate component-wise on vectors.
- Vector operations – There are `dot()` and `cross()` functions for dot- and cross-product respectively. There is also a `length()` function that gives the length of a vector, and `normalize()` which returns a normalized copy of its vector parameter, as well as a `reflect()` function.

While the `main()` routine of a GLSL fragment program takes no arguments and returns void, it has access to a number of inputs. Each of the inputs is of one of two basic kinds:

- `uniform` parameters are specified through the OpenGL API. They may be either scalar types, vectors or texture types. They are called uniform parameters because their value is the same across all fragments being shaded.
- `varying` parameters are those whose value may be different for each fragment. In the case of a fragment program, these are the values that were written by the vertex program, and interpolated by the rasterizer.

GLSL fragment programs cannot perform input/output (no `printf`), cannot write any results except for `gl_FragColor`, and cannot use pointers or read/write memory. These restrictions all mean that a GPU can run many distinct "instances" of a GLSL program in parallel without changing their meanings. This also forces us to be much more creative when debugging.

GPU Rendering For this assignment, we will be using vertex and fragment shaders to implement the Phong reflection model. Because GPUs can run many program instances in parallel, so they run these per-pixel and per-vertex operations many times faster than a CPU of similar size/cost.

Lots of applications that do graphical processing use shaders to reduce processing time. The vast majority of video games on the market today are using shaders as well, whether they are striving for lifelike realism (Battlefield 3) or more artistic effects (Limbo, Braid, Darkness II, Flower, Journey, etc.). By completing this assignment, you will gain experience with the OpenGL tools necessary to make these kinds of cutting-edge applications in real-time graphics and visualization.

Implementing the Phong Reflection Model We have implemented the Phong Reflection model for the simple geometries we provided using the material properties that are defined in `main.cpp`. Since we don't want you to get too bogged down in the minutiae of GLSL built-ins, we have already provided the normal vectors, all of the material properties and lighting information to you in the `phong.frag` shader file. You should take a look at how the material and light properties are set up in `main.cpp` for your edification.

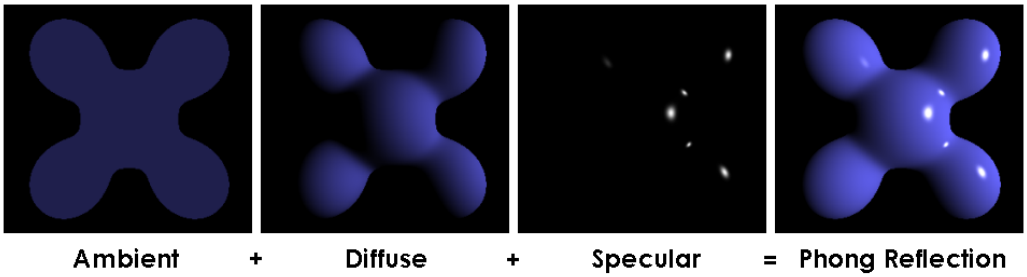

Figure 1: Phong Reflection

Details of the Phong Reflection Model The **Phong Reflection Model** distinguishes between 3 types of reflection:

1. **Specular reflection** - This type of reflection occurs when a surface reflects light from a single light source in a single outgoing direction. Specular reflection is most easily observed on mirrorlike surfaces, such as polished metal or the surface of a body of still water, but can also be observed in combination with other types of reflectance on smooth, glossy/shiny materials (e.g. plastics, polished surfaces, etc.)
2. **Diffuse reflection** - This type of reflection occurs when a surface reflects light from a single light source in multiple outgoing directions. This type of reflection typically occurs on materials that are very rough at a fine level. Examples of such surfaces include pretty much any surface that's not liquid or polished metal, but some particularly good examples might include paper, anything made out of cotton, and plaster.
3. **Ambient reflection** - This is a non-physical type of reflection (ie, one which has no physical analogue). The ambient lighting term in the Phong model accounts for the small amount of light that has reflected off of so many surfaces before hitting your eye, that as an aggregate this light doesn't really have any direction anymore. Higher "ambient reflection" usually occurs in enclosed spaces where light is trapped and has plenty of opportunities to bounce back and forth. This is why an indoor shadow will probably be softer than, say, a shadow cast by a single light source outside at night.

Check out the visual summary in Fig. 1.
The formula for the Phong reflection model (with only one light source) defined in OpenGL state terms:

$$I = M_a I_a + (\hat{\mathbf{L_m}} \cdot \hat{\mathbf{N}}) M_d I_d + (\hat{\mathbf{R_m}} \cdot \hat{\mathbf{V}})^{\text{shine}} M_s l_s \tag{1}$$

where:

- I_a, I_d, and I_s are the ambient, diffuse, and specular illuminations (In OpenGL, these are specified as a 3-vector of floats between 0 and 1, with one component for each color channel).
- M_a, M_d, and M_s are the material's ambient, diffuse and specular colors. If it seems odd to you that a material can simultaneously be 3 different colors, that's because it is odd - at least, its certainly odd to think of a material that way. But these terms don't really represent the color of the material itself, but rather the color of the material multiplied by some constants that, together, describe how the material reflects different kinds of light. In practice, we usually consider the ambient and diffuse material colors to be the same. We often set the specular material color to white (or at least something very bright) since specular reflections are usually the color of the specular light source, regardless of the color of the material.
- $\hat{\mathbf{L_m}}$ is the normalized vector from the surface point to the light source.
- $\hat{\mathbf{N}}$ is the normalized vector normal to the surface at this point.
- $\hat{\mathbf{R_m}}$ is the normalized direction to which that a light ray from the light source to this surface point would reflect (dependent on the surface normal),
- $\hat{\mathbf{V}}$ is the normalized direction pointing from the surface point towards the viewer.

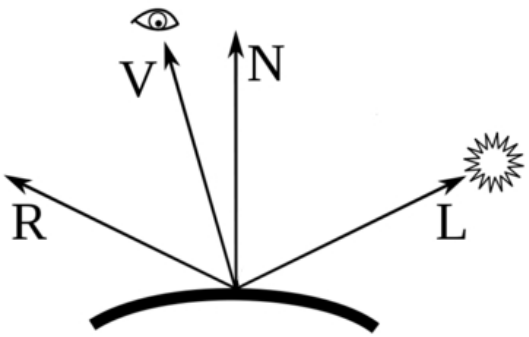

Figure 2: Phong Reflection Model

- shine is a constant representing the shininess of the material. Higher shininess means a brighter, more spatially concentrated specular reflection.

Remember that these dot products could be negative, so you'll have to clamp them to ($>= 0$).

Loop's Subdivision Loop's subdivision is a subdivision algorithm proposed by Charles Loop in his M.S. thesis "Smooth Subdivision Surfaces Based on Triangles" in 1987. If you want to know more about Loop's and other subdivision algorithms you may turn to the SIGGRAPH 99 course notes "Subdivision for Modeling and Animation".

The basic idea of Loop's subdivision is simple: refine and smooth. To refine a mesh it subdivide one triangle into four smaller triangles by adding new vertices on edges, and to smooth the mesh it calculate the new location of an vertex based on its neighbors. The new added vertices on edges and the old vertices on nodes are smoothed separately using different masks. Multiple iterations can be taken to get smoother result (as in Figure 3).

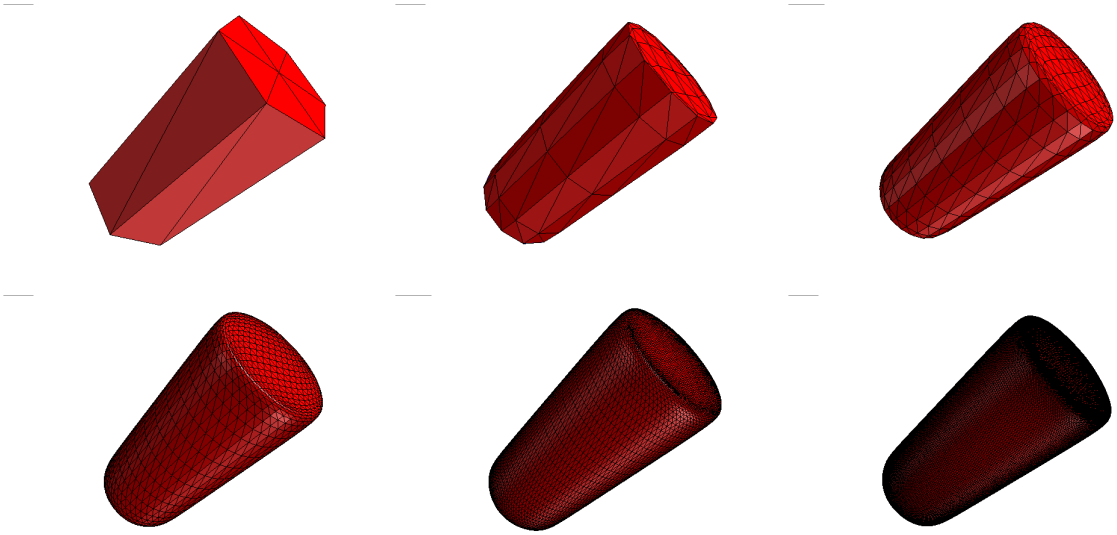

Figure 3: Applying Loop's subdivision on a hexagon cylinder.

Here is a brief description of the algorithm:

- Add new vertices at the midpoints of each edge. (These new added vertices on edges are called odd vertices, while the old vertices on nodes are called even vertices.)
- For each triangle, the three newly added odd vertices on the three edges of the triangle are connected to divide the triangle four smaller triangles.
- Calculate the new locations for both odd and even vertices based on different masks in Figure 4.

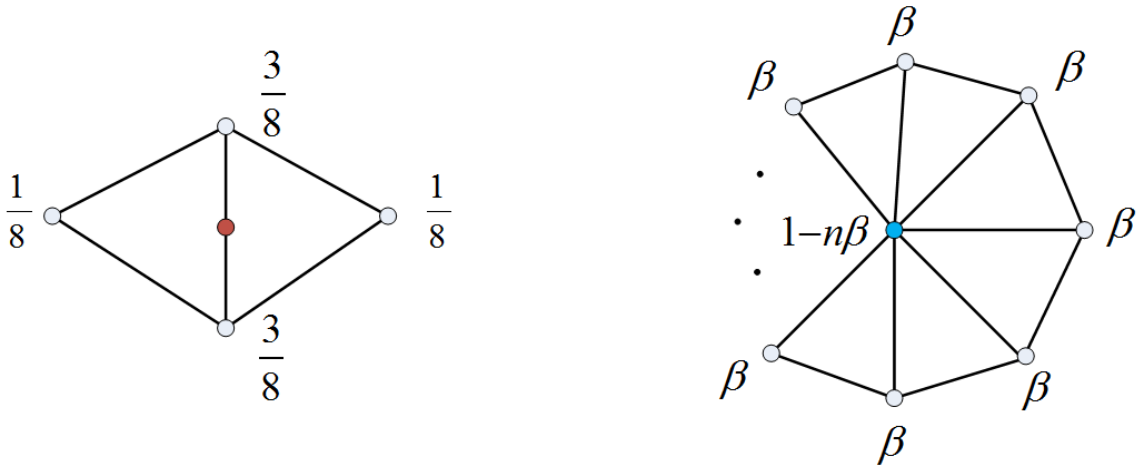

Figure 4: Different smoothing masks for odd (left) and even (right) vertices.

For an odd vertex, its new location is calculated as a linear combination of the two connected vertices of the edge it lies on and two opposite vertices on the adjacent two faces of that edge. For an even vertex, its new location is computed by averaging with all the vertices connected to it by an edge. The averaging weight β depends on the number of the neighboring vertices n which can be calculated as

$$\beta = \begin{cases} \frac{3}{8n} & n > 3 \\ \frac{3}{16} & n = 3 \end{cases} \tag{2}$$

Your Sample Code Package As usual, we have just placed all the different versions (Visual Studio, Xcode, Makefile) into one package. Just download it and ignore the files that aren't related to your system. This starter package also includes `libST`. You'll need to use this version of `libST` (not an old one from a previous assignment) because we've made some modifications to the `STShaderProgram` class and added the `STTriangleMesh` class.

Build the `assignment3` program. Before starting on the code for this assignment, you'll want to run the starter program and make sure that the machine you are using displays everything correctly.

GLEW With this project we will start using another external library, the OpenGL Extension Wrangler Library (GLEW). For Visual Studio and Xcode users, there should be no additional steps for using this package – We have tried to set up the projects so things are automatic.

For Linux users, you will need to install this library to your own system. You might be able to install the `libglew-dev` package on your machine using the package manager for your distro, or get the source code directly from GLEW on SourceForge. You may find that you need to change the project make to use `-lGLEW` instead of `-lglew` to get things to build.

For instructions on how to get the assignment running on myth machines, please see how to get GLEW running on `myth` machines in the Appendix.

Figure 5: Output of the starter `assignment3` program

Try the Starter Program Because we will be using OpenGL functionality that your machine may not support, the first step after you have built `libST` is to build the `assignment3` program and try to run it. If everything works, you should see something like Fig. 5:

If It Didn't Work If the program prints out error messages saying that your GPU supports neither OpenGL 2.0 nor ARB extensions, your GPU may not be new enough for the assignment. If you have not done it already, try updating your graphics card drivers. Depending on your machine, you may have to go to the web site of your graphics card manufacturer (NVIDIA, AMD/ATI, Intel) or the website of your computer manufacturer. If things still don't work after you've updated your drivers, this means that your GPU is incapable of handling the shaders we will write for this assignment (as of last year, even the latest revisions of Intel integrated graphics were insufficient). Try the `myth` machines (Gates B08). We have successfully tested the application in the `myth` lab. You will probably need to do the assignment there **in person** – It is unlikely that you will be able to run these advanced OpenGL programs remotely.

Understanding/Using `assignment3`

Behavior: `assignment3` takes a set of command line arguments

```
./assignment3 vertShader fragShader objMeshFile
```

Explanation:

- `vertShader` - The source file for the vertex shader program you want OpenGL to use.
- `fragShader` - The source file for the fragment shader program you want OpenGL to use.
- `objMeshFile` - The obj mesh file you want to load.

To get started, run `assignment3` with the following arguments:

```
./assignment3 kernels/default.vert kernels/phong.frag meshes/sphere.obj
```

When the application starts up, it will draw a single blue plane on a black background.

- You can rotate the object you are looking at by dragging the mouse on the screen with a left-click.

- A right-button dragging will zoom in and out.
- You can move the camera (vertically and horizontally) with the arrow keys.
- You can reset the camera (position and orientation) by hitting the 'r' key.
- You can hit 'm' to switch between the mesh created in the program and the mesh loaded from .obj file.
- You can hit 'f' to switch between flat shading and smooth shading.
- You can hit 'l' to perform Loop-Subdivision on current mesh.

The starter code also includes some screenshot functionality. Just press the 's' key to write the screen contents to a 'screenshot.jpg' file in your current working directory.

Requirements In this assignment you are required to draw or use different triangular meshes from what we provided, shade the meshes by adjusting the light colors, positions or types properly, and refine the mesh using sub-division. You should be able to interactively control the level of sub-division to change the smoothness of the mesh.

Note that there are two week left before submitting your final scanline image. It is highly recommended that in this assignment you generate all the geometries that are necessary for your final image and shade them with proper light conditions. Then next week after you have learnt textures, you will texture the objects you draw in this assignment to make them more colorful. For the final image, we expect pretty and realistic image. And the image will be graded primarily based on how pretty or realistic it looks. Therefore, it is recommended that you design your image using complex enough geometries from the real world and probably more than one geometries, and shade the objects realistically.

Grading This assignment will be graded according to the following rubric.

- + – Exceeds the requirements via one or more artistic/technical contributions
- ✓ – Meets all of the requirements
- − – Does not meet the requirements but still produces a drawing.
- 0 – The submitted solution does not produce a drawing.

Misc. Here are a list of websites where you can download the mesh data of different kinds of objects built by other people - tf3dm.com, www.turbosquid.com, archive3d.net, www.3dmodelfree.com, 3docean.net. Some of these websites are free, and some may ask you to pay. You are allowed to use the meshes generated by other people to make beautiful pictures, but for this assignment you are also required to generate some coarse meshes by yourself and refine them using sub-division.

MeshLab is a good free software to modify meshes and convert mesh files. You can use it to export .obj files with only vertex locations and face datas. (But you may want to keep the original files if they contain normal, texture information that you want to use in the next assignment). NetGen can be used to generate high-quality meshes. If you want to generate meshes of some simple geometries, you can describe them in .geo files, load .geo with NetGen, generate, then export to .stl files, and finally convert .stl to .obj with MeshLab.

How to get GLEW running on Myth machines

- Download GLEW. Grab the tarball since it has Linux config files. (To untar `file.tar.gz` or `file.tgz`, type `tar -xf file.tar.gz` or `tar -xf file.tgz`).
- Unzip the assignment. This should create the `assignment3/assignment3` directory.
- Unzip/Untar the GLEW source into the `assignment3/assignment3` directory. This will result in the creation of a `assignment3/assignment3/glew` directory. If the resulting directory has a different name such as glew-10.0, change this directory name to be glew.
- Run `make` inside `assignment3/assignment3/glew`.
- Build `libST`.

- Now you should be able to build `assignment3` as well. If you are still having problems linking to `glew`, try changing `glew` in the `Makefile` `LIBS` list to `GLEW`.
- To run `assignment3`, you will also need to add the `glew` shared objects to your dynamic library loading path. To do this, you need to modify the `LD_LIBRARY_PATH` environment variable to include the path to the `glew/lib` directory. Here is the command you would run if you had placed your `assignment3` directory directly in your home directory:

```
setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:$HOME/assignment3/assignment3/glew/lib
```

If your `assignment3` is somewhere else, modify the `$HOME/assignment3` portion of the above command to include the correct path to `assignment3`. Note that you will need to re-run this command every time you log in to a `myth` machine.

Now you should be ready to run the assignment! If this does not work, make a Piazza post.