Transformations and Simple Shading – Assignment 2

Due: April 17, 2013
Spring 2013 Computer Science 484
Principles of Computer Graphics

Instructions: Please submit your solution to the following exercise by 23:59 on April 17, 2013 via «`http://dropbox.ecs.fullerton.edu/`». Your submission must, at a minimum, include a plain ASCII text file `README`, all necessary source files, libraries, and build configuration files to allow the submission to be built and run independently by the instructor. **All files** must include a header identifying the author, author's contact information, and a brief description of the file. The `README` file must describe any bugs and features along with a description of what was or was not completed in the solution. The `README` must also describe the use of the program and if there are any external dependencies. **Do not include any object files, binary executables, or other superfluous files.**

Place your submission in a folder and name it *lastname.firstname_asgt2*. **Zip** the folder and submit the zipped folder, named *lastname.firstname_asgt2.zip*, via «`http://dropbox.ecs.fullerton.edu/`». For example, if your name was Tilly Titan, then your file would be named *Titan.Tilly_asgt2.zip*. **Do not use any other archiving method other than zip.**

Failure to follow these instructions will detrimentally effect your assignment's score.

Plagiarism and academic dishonesty is not tolerated. Correctly and properly attribute all third party material and references.

Do not make calls to `system( )`, `exec( )` or similar system calls which trigger the operating system to execute an external program.

Assignment

Objective: The objective of this assignment is to perform your own modeling and perspective transformations as well as computing the color of the model's surface at each vertex. In order to compute the color at each vertex, your program will have to calculate the model's normals as well as performing an illumination calculation per vertex. You will be building upon your previous experience with command-line PLY model processing in conjunction with your matrix and vector code. The resulting program, named `simpleshader`, will be an interactive graphics program that allows the user to interact with the model.

Rubric: The assignment is out of 100 points.

- 30 points for compiling with no errors or warnings
- 20 points for transforming the model correctly
- 15 points for rendering flat shading correctly
- 15 points for rendering smooth shading correctly
- 10 points for a correct user interface
- 5 points for reading from the described input file formats.
- 5 points for using the correct name for files, the program and documenting your code.

Not including a `README` forfeits 25 points. Your code is expected to be well organized and well documented[1]. Points will be deducted for poorly structured or undocumented code.

Prerequisites: In order to complete the exercise successfully, you will need to have the following:

- ANSI C or C++ compiler
- A text editor
- `make` or other build tools such as Microsoft Visual Studio or Apple Xcode
- The GLUT library
- The OpenGL library

[1] Well organized and well documented means sufficient comments to explain what all the functions do and even what is going on within a function

Models: The models used in this assignment will be the parameteric surfaces taht are generated programatically. Each surface is computed and loaded into an indexed face list data structure. An indexed face list contains parallel lists of vertices, indices to the vertices (triangles), triangle face normals, and vertex normals. The sample code does not compute the face nor vertex normals and must be implemented by the student.

Face normals are trivially computed by taking the cross product of two triangle edges that share a vertex. Make sure you are using the right hand rule as well as normalizing all normals to unit length.

Vertex normals are just as easily computed as the average of the face normals that are incident upon the vertex in question. A simple algorithm to achieve this with our face list data structure is

```
foreach face f
  foreach vertex v in f
    vertex_normal[v] += face_normal[f]
foreach vertex v
  normalize( vertex_normal[v] )
```

Computing the Inverse of a Matrix: In order to transform your normals, you will need to compute the inverse of your transformation matrices. A simple way to accomplish this is to divide the adjoint of the transformation matrix by its determinant (assuming the determinant is non-zero). In other words, given a matrix A,

$$A^{-1} = \frac{A^H}{|A|} \tag{1}$$

where A^H is the adjoint matrix[2] of matrix A and $|A|$ is the determinant of matrix A. The adjoint is only defined for square matrices. The adjoint can be calculated by taking the determinant of the $(n-1) \times (n-1)$ matrix obtained by deleting the i^{th} row and j^{th} column of A. The terms of this new matrix are known as the cofactors of A.

The inverse can be computed using the following pseudo-code. The pseudo-code is an adaptation of Section 5.2.4 (page 126) in the Shirley textbook. Remember that a matrix has an inverse so long as its determinant is non-zero.

```
float invert( Matrix m, Matrix *adjoint_matrix ){
  deteterminant = m.determinant( );
  if( det != 0 )
    adjoint_matrix = m.adjoint( );
    adjoint_matrix /= determinant;
  }
  return( determinant );
}
Matrix adjoint( Matrix m){
  Matrix adjoint;

  adjoint.column[0] = cofactors( -m.column[1], m.column[2], m.column[3] );
  adjoint.column[1] = cofactors( m.column[0], m.column[2], m.column[3] );
  adjoint.column[2] = cofactors( -m.column[0], m.column[1], m.column[3] );
  adjoint.column[3] = cofactors( m.column[0], m.column[1], m.column[2] );

  return( adjoint );
}

Vector4 cofactors( Vector4 t, Vector4 u, Vector4 v ){
  // Determinants 1 through 6
  float d1 = (u[2] * v[3]) - (u[3] * v[2]);
  float d2 = (u[1] * v[3]) - (u[3] * v[1]);
```

[2]The adjoint matrix is also known as the conjugate transpose, adjugate matrix, Hermitian adjoint, or Hermitian transpose.

```
  float d3 = (u[1] * v[2]) - (u[2] * v[1]);
  float d4 = (u[0] * v[3]) - (u[3] * v[0]);
  float d5 = (u[0] * v[2]) - (u[2] * v[0]);
  float d6 = (u[0] * v[1]) - (u[1] * v[0]);

  return( Vector4(- t[1] * d1 + t[2] * d2 - t[3] * d3,
                    t[0] * d1 - t[2] * d4 + t[3] * d5,
                  - t[0] * d2 + t[1] * d4 - t[3] * d6,
                    t[0] * d3 - t[1] * d5 + t[2] * d6 ) );

}
```

Transformations: OpenGL provides a number of functions to perform basic transformations. These functions are `glTranslate( )`, `glRotate( )`, `glScale( )`, `gluLookAt( )`, `glFrustum( )`, `gluPerspective( )`, and `glOrtho( )`. You will be replacing these functions with your own that will perform the necessary calculations and then insert the result into OpenGL's modelview or perspective matrix stack.

To accomplish this, create new functions which replace the `gl` with `my`. For instance, `glScale( )` will be replaced by a function you write named `myScale( )`. To learn more about the OpenGL function you are replacing, look them up in the *OpenGL Reference Manual*, also known as the Blue Book. A copy is available for viewing online at «`http://www.opengl.org/sdk/docs/man2/`»

In the sample code, a `myGL.h` is given which provides some guidance about which functions will need to be implemented. A related implementation file `myGL.cpp` is provided as well.

The following is a list of functions that you will have to implement. They should behave similar to the similarly named OpenGL functions.

Remember that OpenGL thinks of 4×4 matrices as a 16 unit array representing a column major order matrix. You should write an appropriate helper function for your matrix class to convert between your representation of a matrix to OpenGL's representation.

Here are some additional tips to help you get started:

- To find out what matrix mode OpenGL is currently in, use the following snippet of code:

```
int mode;
glGetIntegerv( GL_MATRIX_MODE, &mode );
```

- To load a 4×4 matrix as a 16 unit array representing a column major order matrix into the top of the currently active stack, use the following snippet of code:

```
float matrix[16];
glLoadMatrixf( matrix );
```

- To get a copy of the matrix from the top of the currently active stack, use the following snippet of code:

```
int mode;
float matrix[16];
glGetIntegerv( GL_MATRIX_MODE, &mode );
if( mode == GL_MODELVIEW ){
  glGetFloatv(GL_MODELVIEW_MATRIX, matrix);
}else if( mode == GL_PROJECTION ){
  glGetFloatv(GL_PROJECTION_MATRIX, matrix);
}
```

- To set the top of the currently active stack to the identity matrix, use the `glLoadIdentity( )` function.

As a rule of thumb, you can approach rewriting each of the aforementioned functions by taking the following steps:

1. Grab a copy of the top most matrix from the currently active stack.

2. Create a transformation matrix given the input to the function.
3. Multiply the two matrices together.
4. Send the product back to OpenGL.

Unlike OpenGL, print error messages immediately and exit gracefully.

Surface Shading: The fixed-function OpenGL renderer uses the Phong illumination model to calculate surface shading. In your assignment, we will use a simplified illumination model that produces a polished golden/brassy-like appearance. For this to work, you will need to disable all of OpenGL's lighting and shading.

Start by first implementing the shading as flat shading. In other words, compute one color per face using the surface's face normal. Based on the camera position, vertex, and the vertex's face's normal, you will compute the RGB color using the shading equation below. Set all the polygon's vertices' colors to the same computed color using `glColor3f( )`. This will give the model a faceted look.

Based on the camera position, vertex, and the vertex's normal, you will compute the RGB color using the shading equation below, and set the vertex's color explicitly using `glColor3f( )`. OpenGL will then bilinearly interpolate the polygon's vertex's colors, similar to how Gouraud shading is done, thus giving a smoothly shaded appearance.

The new shading equation uses a different exponent for each color channel, given by

$$C_\alpha = W_\alpha \left(\frac{\cos\theta + 1}{2} \right)^{P_\alpha} \tag{2}$$

where θ is the angle between the *eye* vector and the *reflection* vector, and α here connotes the color components, red, green and blue. Hence, there are six parameters.

By default, set (W_r, W_g, W_b) to $(1, 1, 1)$, and (P_r, P_g, P_b) to $(1, 2, 20)$. This will produce the aforementioned polished golden/brassy-like appearance. Note that all the vectors are unit vectors. An illustration of the vectors used for calculating the illumination at a vertex is shown in Figure 1.

See the section on User Interface regarding the use of the 'd' key to toggle between these two shading modes.

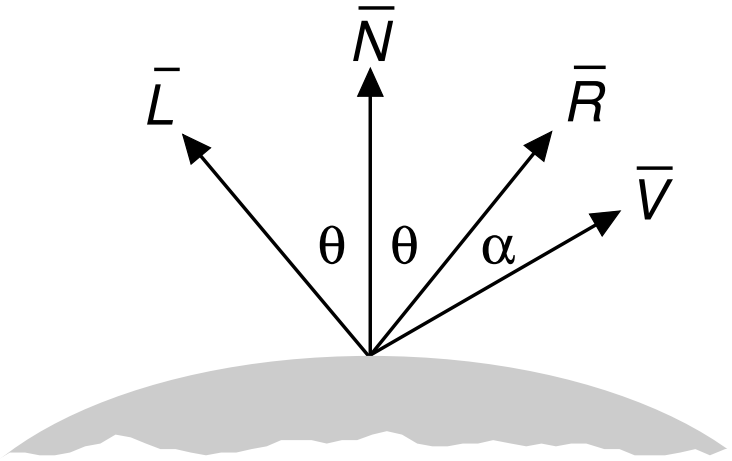

Figure 1: Vector L is from the surface to the light. Vector N is the normal to the surface. Vector R is the vector along the reflection. Vector V is the vector to the viewing position. The angle θ is the angle of reflection and incidence. The angle α is the angle between the vector of reflection, R, and the view position, vector V.

The light source is a point light sourced positioned at $(1.25, 1.25, 0.5)$. Position the camera at $(0.0, 0.0, 1.0)$ looking at the origin, $(0.0, 0.0, 0.0)$, with $(0.0, 1.0, 0.0)$ as the up vector.

Drawing: To draw the scene, you will need to submit the model's vertices and vertex colors to OpenGL's renderer. For every frame, the vertex colors need to be recomputed. If your program becomes unbearably slow, try enabling your compiler's

optimizations. Do not issue the face or vertex normals to the renderer since you are computing the surface shading on your own.

User Interface: This assignment creates a command line program named `simpleshader`, which will have a number of command line options.

An example of how the command will be used on the command line is given below.

```
prompt> simpleshader
```

The following is a list of all command line options with their descriptions. Feel free to add other options as you see fit.

- `-v` Turn on debugging messages.
- `-h` Print out a help message which tells the user how to run the program and brief description of the program.

The program should not print any debugging messages or other noise unless the `-v` option is enabled.

Once the program is running, the user will be able to use the mouse and keyboard to interact with the program. The keyboard controls are case insensitive.

- mouse left button drag: translate in XY plane
- mouse shift-left button drag: translate in XZ
- mouse crtl-left button drag: translate the camera towards/away from the origin
- arrow up/down: rotate around X
- shift+arrow up/down: rotate around Z
- arrow left/right: rotate around Y
- ' ': cycle to next surface
- D: toggle smooth/flat shading
- P: toggle between orthographic and perspective projection
- +: Positively scale the model
- -: Negatively scale the model
- esc: quit

Suggested Improvements: Consider implementing at least one of the suggested enhancements or an enhancement of your own creation once you have completely satisfied the assignment's base requirements. Document any enhancement you implement in your `README`.

1. Add a simple animation using the `glutIdleFunc( )` function
2. Read and parse other geometric models stored in text files such as *3D Studio Max (.3ds)* or *Wavefront (.obj)* file formats
3. Enable the light source to be moved or animated.
4. Enable the camera to be moved or animated.
5. Create a new shading function.
6. Implement Phong illumination.
7. Implement Phong shading.

Sample Code: The sample code is available at «`http://gamble.ecs.fullerton.edu/teaching/spring13/cs484/sample-code/`».