eecs 487

INTERACTIVE COMPUTER GRAPHICS

EECS 487 PA2: Scene, Camera, Lights!

This assignment is due on Thursday, Oct. 23nd, 2014 at 10:00 pm.
Note the change in the Submission Guidelines below.

Plain Page for Printing

Overview

In this assignment you will work on a simple model/scene viewer `view3D` that uses a simplified X3D scene graph as well as two simple programmable shaders. The provided `view3D` parses as input a scene description file in the X3D format, partially constructs a scene graph with various X3D nodes, and passes the scene to the renderer. The *full* X3D file format is described in the X3D Specification. Most of your tasks will be modifying the source file `scene.cpp`. Look for words `YOUR CODE HERE` to locate where to work. You will also modify the GLSL vertex and fragment shader files: `phong.vs` along with `phong.fs`, and `interesting.vs` along with `interesting.fs`.

You're *not* required to read most of the X3D documentation reference herein. Actually, you may find it more helpful to simply read the class definitions in `scene.h` and glace over the provided code in `scene.cpp` relevant to the class of interest. The one exception is the X3D Lighting Specification referenced below, which you do need to read carefully.

The provided code has been tested to compile under Linux, Mac OS X, and Windows. However, CAEN Linux workstations do not have great graphics drivers and the programmable GPU shaders will not work there. If you have a Linux machine with good drivers you can use the `Makefile` provided. I haven't been able to get shader support to work with Cygwin/X under Windows 7. If you manage to get it to work, please let me know. For further help with the support code, please see details below.

Graded tasks overview (100 points total)

1. Implement rendering of a Cylinder node modifying `X3Cylinder::Render()` function. (5 pts)
2. Implement simple camera navigation ("examine mode") by writing a couple of `X3Viewpoint` methods. (25 pts)
3. Implement proper modeling Transformation sequence by modifying `X3Transform::Render()` function. (10 pts)
4. Setup Point Lights by modifying `X3PointLight::SetupLights()` and `X3Transform::SetupLights()` functions. (10 pts)
5. Setup the Material properties by modifying `X3Material::Render()` function. (5 pts)
6. Compute the normals and render the IndexedFaceSet (mesh) node, modifying methods of `X3IndexedFaceSet::Render()` and `X3IndexedFaceSet::Add()`. (15 pts)
7. Implement per-vertex Gouraud shading that matches the fixed-function

HW1

HW2

Don't see the menu?

per-vertex rendering (modify `gouraud.vs`). (10 pts)

8. Implement Phong and Blinn-Phong variants of per-fragment shading (`phong.vs`, `phong.fs`, and `blinn.fs`) (10 pts)
9. Implement some interesting shader (modify `interesting.vs` and `interesting.fs`). (10 pts)

Implementation details

We provide several X3D scene description files in the `scenes` folder. All the X3D files refered to below can be found in this folder. As is, the skeletal `view3D` provided can only display the X3D files `00-cube.x3d` and `00-cone.x3d`. If you try to view any of the other files, you will simply get a blank blue window.

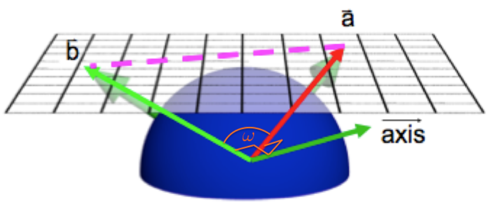
Cylinder rendering

Modify the `X3Cylinder::Render()` function to render Cylinders using all the provided data fields of the cylinder node. It may help to look at how the same function is implemented for the Cone primitive. After you're done with this task, your `view3D` should be able to display `01-cylinder.x3d`.

Camera examine mode

The simple X3D viewer and parser included with the project does not include a camera class. All the viewing information is stored within the X3Viewpoint node. The gaze center ("lookAt" coordinates) of the camera is assumed to be the origin of the camera coordinate frame. The location of the camera itself is stored in the `position_` member of `X3Viewpoint`, in camera coordinates. The camera coordinate frame is initialized to coincide with the world coordinate frame and the camera is initialy placed on the positive *z*-axis of the world coordinates (equivalentlly, the *w*-axis of the camera coordinates), with the gaze vector pointing towards the origin. The `up_` member of the `X3Viewpoint` class helps you keep track of the camera's up direction, which we assumed to be the same as the *v*-axis of the camera coordinate frame and initialy coincides with the positive *y*-axis of the world coordinates.

You will need to implement dollying in and out (mapped to dragging the mouse up and down with the *right* button pressed— FYI, vanilla GLUT doesn't support the mouse wheel). You will also need to implement camera tracking along the latitude and longitude of a spherical space around the origin of the camera coordinate frame (a.k.a. arcball or trackball camera). After each viewing transform (each call to `display()`), the camera is always at lat/lon (0, 0) in the camera coordinate frame, with its gaze towards the origin (of the camera coordinate frame, which coincides with the origin of the world coordinate frame). Dragging the mouse left and right with the *left* button pressed causes the camera to track along the equator. Dragging the mouse up and down with the *left* button pressed causes the camera to track along its longitude. In the figure below assume the mouse is dragged from point 'a' to point 'b' on the screen. You need to translate this linear movement of the mouse into a rotation of the camera on the sphere around the origin.



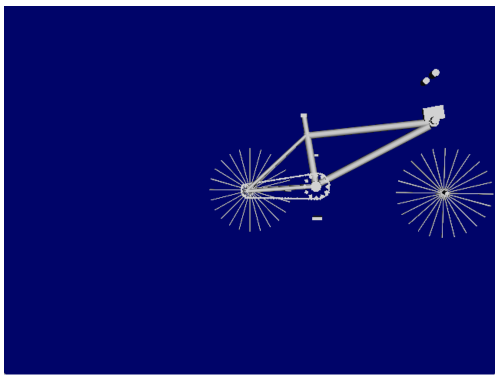
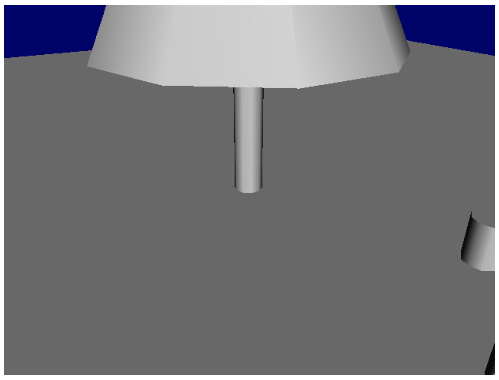
When you rotate the camera's position (or, equivalently, its gaze or *w*-axis) along the equator, you are actually rotating the camera's coordinate frame about its *v*-axis. So you need to rotate the camera coordinate frame's *u*-axis also. Similarly, dragging the camera along its longitude is equivalent to rotating the camera coordinate frame along its *u*-axis, so you need to rotate the *v*-axis (or `up_` vector) also. For consistent user experience, we will assume a constant mapping between the amount of on-screen mouse movement and the amount of camera movement. That is, we assume unit gaze vector at all times, even after the camera has been dollied closer or further away from the origin.

You will need to write `X3Viewpoint::track_latlong()` (6 lines) and `X3Viewpoint::dolly()` (2 lines). Feel free to add private members to the `X3Viewpoint` class in `scene.h` to cache frequently computed values (if you do, don't forget to initialize them in `X3Viewpoint` constructor).

Once you have interactive camera control, examine the `01-cylinder.x3d` again to verify that you have set up the top and bottom sides of the cylinder correctly. If you view `00-cube.x3d` again you'll see that it really is a cube, not just a square. Similarly, you can now "fly" about `02-ds9.x3d`—to see the space station, you first need to "undock" and "fly away."

Modeling transformation hierarchy

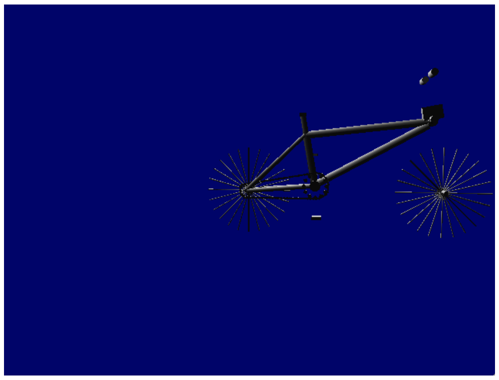
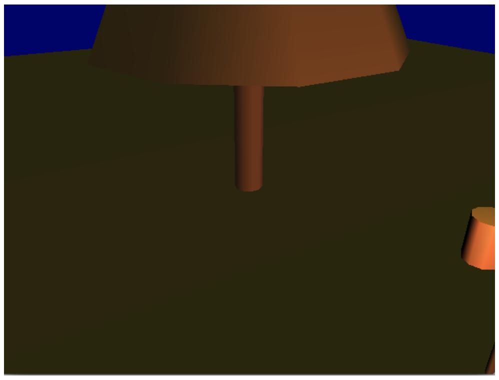
The scene graph represents the scene as a directed acyclic graph (it is not a simple tree since some nodes can be reused multiple times) and the traversal of this graph is performed when the scene is rendered. Grouping nodes such as `X3Scene`, Group, and Transform traverse their children so that each child renders itself. The `Transform` node adds the ability to specify the transformation between the child coordinate system and the parent coordinate system so that children can be specified with respect to their parent nodes. Implement `X3Transform::Render()` following the inline instructions. The inline comments pretty much give you the solution to this task, so think of it as a chance to practice your knowledge of OpenGL API calling conventions (10 lines). Practice and drill. The reward though is that you can now view `03-bikeframe.x3d`, `03-desklamp.x3d`, and `03-desktop.x3d`.

Lighting setup

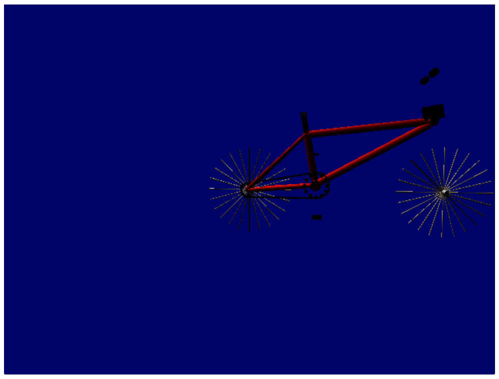
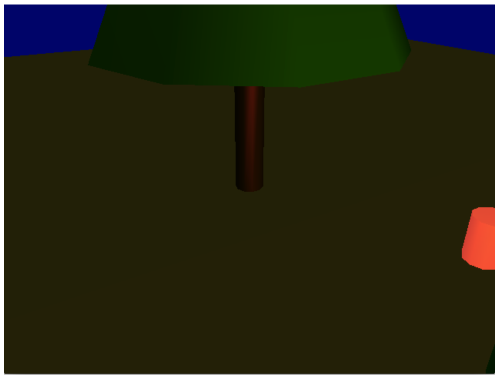
We will support only point lights in `view3D`. We will assume each point light effects the whole scene, and therefore a separate pass is needed to setup light positions and other light parameters before the scene is rendered. Each point light can be located anywhere in the scenegraph. A light attached to a child node is transformed by the modeling transformation of the child's parent(s), allowing a light to be "attached" to a particular object.

The `X3Scene::SetupLights()` method recursively calls the `SetupLights` method of children nodes down the scenegraph hierarchy. Each node's `SetupLights` should be implemented very similar to the node's `Render` method. Implement the `X3Transform::SetupLights()` method to do this for `X3Transform` node (10 lines). Then in the `X3PointLight::SetupLights()` method set up OpenGL lighting parameters for PointLight to match the X3D Lighting Specification (9 lines, you need to read this lighting specification). In our simplified viewer we have no fog and no spot light, so ignore those parameters. Be careful to match X3D lighting parameters onto OpenGL's parameters. If no lights are specified within the X3D file, a default light is added to the scene (in `display()`). Make sure to assign properties to the light with the proper id: `GL_LIGHT0 + light_index`. You can now see the effect of lighting on `03-bikeframe.x3d`, and `03-desktop.x3d`. But to really put a shine on the surfaces, you need to finish the next task.

Material setup

You set up the Material properties of your surface inside the `X3Material::Render()` method (6 lines). Lights are setup once before rendering the whole scene whereas materials vary from node to node. Be careful to match X3D material properties onto OpenGL's material properties, for instance, shininess in X3D gets multiplied by 128, so check that the resulting colors and intensities are what you intended. Now you can view `00-cone.x3d`, `00-cube.x3d`, `01-cylinder.x3d`, `03-desklamp.x3d`, `03-bikeframe.x3d`, and `03-desktop.x3d` in their glorious colors.

Computation of mesh normals and mesh rendering

IndexedFaceSet represents a polygonal mesh node in the form of a list of vertex indices. Each entry in the vertex index list refers to an entry in the list of vertex coordinates. Our simplified X3D parser supports only triangle and quad meshes. From an X3D file, the parser extracts two separate lists: one of quads and the other of triangles (see the `X3IndexedFaceSet` class and the comments in

`X3IndexedFaceSet::Add()`). The parser also assumes only coordinates are given (no normals or colors, for example). In order to compute lighting, you need to compute a normal for each vertex. The normal of a vertex is the average of the normals of the polygons sharing the vertex. To compute the normal of a vertex, you need to first compute the normal for each polygon. Each vertex then accumulates the normal computed for each of its incident polygon. After all the polygons have been processed, average out the normals of the vertices by normalizing them. Vertex normal computation is to be done in the `X3IndexedFaceSet::Add()` method.

You will also need to implement mesh rendering within the `X3IndexedFaceSet::Render()` method. For each triangle, draw triples of vertices in a loop within the `GL_TRIANGLES` rendering mode. Remember to specify `glNormal()` before each call to `glVertex()`. Similarly for quads. Review your `X3Cylinder::Render()` for an example, but specify a separate normal for each vertex. Once you are done with this task you will be able to display the famous graphics teapot model, `04-teapot.x3d` but check out the `04-coolcube.x3d`! Other files you can now view are: `04-eight.x3d`, `04-fish.x3d`, and `04-torch.x3d`. A few of the models provided have non-planar surfaces in their indexed face sets. This will cause them to have a small number of polygons missing, such as `04-fish.x3d` having a small hole in its cheek. This is caused by view3D supporting only a subset of X3D, e.g., we throw away the provided normals and colors and we only recognize triangles and quads. You don't have to fix the holes.

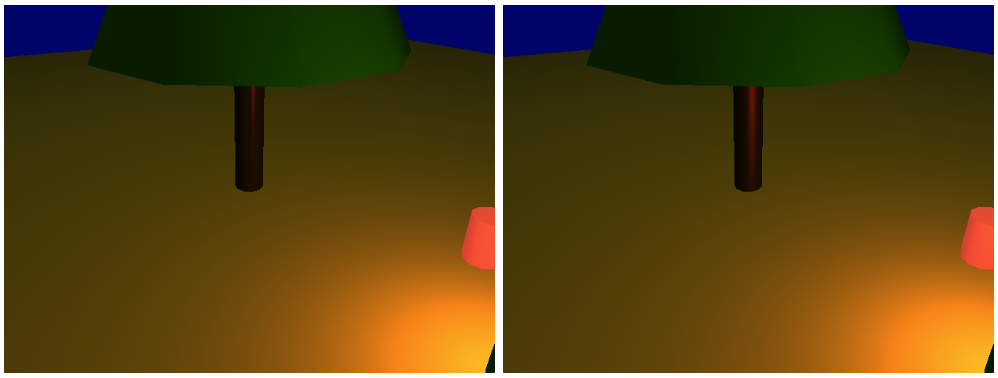
GLSL shaders

These last three tasks will be done in GLSL. Before starting this part of the assignment, you may want to review the GLSL syntax in the lecture slides on Programmable Shaders. The provided support code reads and compiles shader files. The shader files must be in the same folder from where `view3D` is run (if you're using an IDE, this could be where the project file is). The program compiles the shaders into GPU-executable program objects. The code in `shaders.cpp` performs these tasks. You are encouraged to read it to see how it traps errors and notifies the user of failed compilation and linkage. You can mix and match compatible vertex and fragment shaders. By default, the program uses OpenGL fixed-function pipeline.

First implement a per-vertex Gouraud shading that matches the smooth rendering given by OpenGL fixed-function pipeline. The `view3D` application passes the number of lights in each scene to the shaders using the `uniform` variable `numlights`. You should use this variable to compute the contribution of all lights in a scene in your shaders. Be sure to compute all components of the lighting formula, including emission, global and local ambient, and effect of attenuation. You would want to use the built-in `uniform` variables `gl_FrontMaterial`, `gl_LightModel`, and `gl_LightSource[]` (see lecture notes). The final color in the vertex shader must be assigned to the vector `gl_FrontColor.rgb`. To match OpenGL's fixed-function lighting exactly, follow the instructions in `gouraud.vs` on what to implement. You don't need to provide a fragment shader. OpenGL will default to the fixed-function fragment shader. Compare the output of your Gouraud shader for `03-desktop.x3d` against the one generated by OpenGL fixed-function pipeline. They should look identical.

Next implement per-pixel shading. Most of the work will be done in the fragment shader `phong.fs`, but the vertex shader `phong.vs` must first prepare the per-vertex attributes. Unlike with the vertex shader, the final color in the fragment shader must be assigned to the vector `gl_FragColor.rgb`. Compute the per-pixel shading using the original Phong algorithm. View the effect of your Phong shaders on `01-cylinder.x3d`, `04-coolcube.x3d`, `03-desktop.x3d`, and `04-torch.x3d`

To implement the Blinn-Phong variant of per-pixel lighting, copy your `phong.fs` to `blinn.fs` and make the slight Blinn-Phong modification (`view3D` will quietly reuse your `phong.vs` as vertex shader). Now view `04-torch.x3d` and `03-desktop.x3d` again and marvel at the difference such a slight modification to the formula can achieve. Below are the screenshots for `03-desktop.x3d` with Phong and Blinn-Phong shading respectively.



The "interesting" shader task is an open-ended task and you can implement any nice or weird looking shader. Examples of interesting shaders may include the brick procedural texture or some cartoonish looking textures (note that the provided `interesting.[vf]s` already implements some simple cartoon shading so you need to come up with something a bit more interesting). Be original; do not simply take some random dot product, multiply by 4, take the square root, and convert to rods per hogshead. Think it through and make sure to explain and understand the choices that you made.

Support Code and Required Libraries

Download the assignment archive from [`/afs/umich.edu/class/eecs487/f14/FILES/pa2.tgz`].

The program `view3D` should be called with one command-line argument—the name of the X3D scene file. We have included some in the `scenes` sub folder—there are many other on the web, and some graphics packages can export to X3D.

The following keyboard shortcuts are defined for `view3D` (you may bind other operations to keys in `view3D.cpp:kbd()`, but do not change these predefined keys):

- 'q' or ESC: quits the program
- '@': resets the camera
- 'p': activates the per-pixel Phong shaders.
- 'b': activates the per-pixel Blinn-Phong shaders.
- 'i': activates the interesting shaders.
- 'n': disables shaders (default, no shader, fixed-function pipeline mode).

The provided code must be linked with the Expat XML Parser libraries, in addition to the usual OpenGL, GLU, and GLUT libraries. On Linux and Windows platforms, the code further must be linked with GLEW, the OpenGL Extension Wrangler. Please see the course note on how to install and use GLEW and Expat, which also covers other IDE-related topics such as how to specify command line options, add header file search path, and link against external libraries and dynamically linked libraries. The use of `expat` should be completely transparent to you and you don't need to study it. Your code must not require other external libraries or include files other than the ones listed in this spec.

Submission Guidelines

As with PA1, to incorporate publicly available code in your solution is considered cheating in this course. To pass off the implementation of an algorithm as that of another is also considered cheating. For example, if the assignment asks you to implement sort using heap sort and you turn in a working program that uses insertion sort in place of the heap sort, it will be considered cheating. If you can not implement a required algorithm, you *must* inform the teaching staff when turning in your assignment, e.g., by documenting it in your writeup.

Test the compilation!
Code that does not compile will be heavily penalized.

Create a writeup in **text format** that discusses:

1. Your platform and its version - Linux, Mac OS X, or Windows.
2. Anything about your implementation that is noteworthy.
3. Feedback on the assignment.
4. Name the file writeup-*uniqname*.txt.
   For example, the person with uniqname *tarukmakto* would create *writeup-tarukmakto.txt*.

Your "*PA2 files*" then consists of your `writeup-uniqname.txt`, `gouraud.vs`, `phong.vs`, `phong.fs`, `blinn.fs`, `interesting.vs`, `interesting.fs`, `scene.cpp`, and, if modified, `scene.h`. Your code must not require other external libraries or include files other than the ones listed in this spec.

To turn in your PA2:

1. Email the GSI the SHA1's of your *PA2 files*. Use "EECS487: PA2 Submission" as your email's "Subject:" line. Once you've sent in your SHA1's, don't make any more changes to the files, or your SHA1's will become invalid.
2. Upload your *PA2 files* to your `pa2` folder on MFiles:
   `https://mfile.umich.edu/?path=/afs/umich.edu/class/eecs487/f14/FOLDERS/<uniqname>/pa2/`.
   Replace "<uniqname>" with your uniqname. Please report any problems to ITCS.
3. Keep your own backup copy! Don't make any more changes to the files once you've submitted your final SHA1's.

The timestamp on your SHA1 email will be your time of submission. If this is past the deadline, your submission will be considered late. You are allowed multiple "submissions" without late-policy implications as long as you respect the deadline. Try not to email your SHA1 to the GSI until you've finalized your code. You don't want to annoy him.

Turn in ONLY the files you have modified. Do not turn in support code we provided that you haven't modified. Do not turn in any binary files (object, executable, dll, library, or image files) with your assignment. Your code must not require other external libraries and include files other than the ones listed in this spec.

Do remove all `printf()`'s or `cout`'s and `cerr`'s you've added for debugging purposes.

General

The General Advice section from PA1 applies. Please review it if you haven't read it or would like to refresh your memory.