Project 2 Documentation

Paul Kline
EECS 700
10/20/2014

The idea behind your project and/or where you got the idea:

I chose to try to model Solomon's Temple (although he would probably 'roll over in his grave' if he saw my version!). I chose this because I wanted to do something from the Bible, and after thinking about it for a few days, I settled on Solomon's Temple. Although I admit I took some 'creative liberties' due to time constraints.

How you generated your model:

My very first step was to just get the cylinder class from the M&M example to render in my project2. That seemed to be a feat all in itself, but relative to the rest of the project, it was not too much time. Important word is relative here. It took me a long time. I haven't decided if I'm proud or embarrassed about the amount of time I spent on this project. So once I got a cylinder to render, I decided to make my own. I started with the code provided in the cylinder class, but soon decided it was too limiting for me. I wanted my cylinder (named "Column") to be able to go anywhere it wanted in any orientation. This was not easy, and I think I spent just as much time thinking about this one as I did coding it. Although I did manage to make a pretty cool looking "thing" in the process. Eventually, I ended up with a "Column" with a massively large constructor, but it provided flexibility in every sense of the word. The Column could be placed anywhere/anyhow, and offered configurable features such as color, whether or not the ends are capped, and different size radii for each end. This comes in quite useful later when building other classes. Because I'm not too sure about how much detail to include, I will go into a little detail here about the way the normal vector is calculated for a column(cylinder) with different sized radii at each end. The way in which I have the Column class arrange I get almost for free the vector from the bottom center to the current point I am trying to find the normal of. This is almost but not quite the vector I need for the normal. It is only the vector to specify the normal if the bottom and top radii are the same. Another vector I can easily calculate is that which is defined by subtracting the bottom point (points are around the circumference, top and bottom) we are interested in with the corresponding top point. Now if I cross these two vectors, This gives me a vector perpendicular to the (bottom point) - (top point) vector. I can take this new vector and cross it with the (bottom point) - (top point) vector to give me the normal! (Right hand rule accounted for of course). Also, by keeping the top points separate from the bottom points, rendering 'caps' onto the columns is relatively easy if desired.

Now that I had my "Column" I wanted to see it. But I was saddened because who knew if the "dark side of the moon" side was really there? So the next thing I wanted was a movable eye.

I admit I spent far too much time on this project configuring the dynamic eye, and the feature underwent three major revisions throughout the life of this project. "Version 1" used the handleKey method to move the eye back and forth in the x-z direction while keeping the distance to the eye the same. The Column class has an "instances" variable, and the first column that gets created gets to handle the movement of the eye. No other subsequently created columns will handle key events. Perhaps not the cleanest way to move the eye, but it worked, and that is step one (I admit the first instance of Column remains the handler of key events for all revisions. Hey, it worked! why change?). I was not satisfied with only rotating left and right. I wanted more. I then added the feature of up and down movement. My brain I think was properly warmed up at this point and I was able to make a cleaner, smaller, "more correct" way of moving the eye up and down. However now I had weird behavior if I moved up or down, and then moved laterally. This was because of the way I was handling lateral eye movement only changed the x-z coords, not relative. So finally with revision 3 of my eye movement feature, the user can use the 'asdw' keys to move left, down, right, and up, respectively. AND the viewer can hold down multiple keys for both effects-- both right('d') and up('w') for instance, to move both directions at the same time. However, to accomplish this is to not use the handleKey method, but use a glfw function (glfwGetKey(..)) that can look up a key state for you-- i.e. whether the key is pressed or not. I can then handle both eye changes in one render call achieving the 'diagonal' view shift. Additionally, the orientation of the scene can soon go awry. This is because I only give control to rotate the eye position, not the center. For this reason, the user can press the 'u' key to reset the up vector to what is traditionally accepted as 'up'. Also, since multiple key presses can be handled at once, the user can hold down 'u', and use the 'asdw' keys to rotate around the center of the scene all the while keeping the orientation of the scene straight 'up'. The 'i' and 'o' keys were also utilized to go 'in' and 'out'. However, I will say that the closer to the center of the scene a viewer gets, the 'choppier' the change in view becomes and is more drastic. This is because I implemented an eye change that is hard coded and not dependent on the distance the eye is from the center. This is a would-be feature in Revision 4..

The previously mentioned revisions of the eye position feature did happen continuously, but rather throughout the development of the rest of the project, since I thought it would be a good idea to work on that as well. After all, I hadn't gotten very far on the part I was to be graded on. This Column class, which is a bit of a misnomer I admit, but I kept it around for historical reasons, was part of bigger master plan. The reason for the name column was that I wanted to make columns for the Temple. Nearly all rendering (artistic and realistic) include Solomon's Temple having two columns in the front. Now I had a very simple column. At this point I thought it would be a good idea to create a "Block" class, that of course, could be put anywhere in any orientation. This again, obeyed Hofstadter's Law and took much longer than anticipated. However, after much math, I was ready to make some compound objects for my scene. But alas, I could not. My column was not fancy enough just yet.
I wanted to make a fancy column that is not just a cylinder all the way around, but is like the image to the left. It was the riveting running up and down the column that I wanted. That other stuff in the photo looks really hard and not "geometrical." My idea was the make half cylinders running up the length of whatever my final column class name would be (since I already used 'column').

So My next step was to create a HalfColumn class. This turned out to a (relatively) pretty easy change given that most of the hard work was done in the creating of the Column class. However, an exact half column turned out to not be desirable, and I changed it to take the span of radians as a constructor parameter. Now that I had my HalfColumn class (which is a bit a double misnomer this time since it is much more dynamic that a column or cylinder, and since it does not have to be a half but rather can span any radial value (<2*PI of course)). This was nice. Now I had the tool I needed to make a fancy column, I just had to throw a lot of math at it.

So a FancyColumn is ~~simply~~ a collection of HalfColumns arranged in a circle such that each HalfColumn spans its portion of the (FancyColumn's) circle (which was sadly not pi radians). So I thought pretty clever of myself when I was able to do this properly (ie HalfColumn normals facing inward, and each HalfColumn draws exactly it's portion such that their cumulitive outline makes a perfect circle). However the constructor of FancyColumn needs to be passed an even number of HalfColumns to make otherwise there will be a gap and everyone could see that is not actually a column and hollow.. And what is a FancyColumn without a base? This to come later. But first..

I thought I would give my Block class a little more attention, and it would be a nice change from curvy things, although I was getting the hang of it. My next goal was to make a Stairs class that is a collection of Blocks strategically placed, which of course, involved a bit of math. I think I made a few thousand new neuronal connections making this project. It involved a lot of thinking/visualizing that I was noticeably faster at towards the end of this project. My final Stairs class made pyramidal stairs with the option to make them flat-backed or not--pretty useful when one is trying to place stairs against a building, or say, a temple.

Back to the curvier things. I decided to call my final column class 'SuperFancyColumn' (Finally not a misnomer curvey thing!). I now had all the pieces to create it including the Block, FancyColumn,and the good ole Column class. After completed, the 'SuperFancyColumn' constructor simply asks for the location to put the super fancy column, direction, and size (color, etc.). Each of the sub components are then calculated to be a percentage of that height and location. Again, like all of my classes, the SuperFancyColumn can be put in any location, in any direction. The feature is not evident by simply viewing the rendering of the Temple because Temples innately do not have tilted columns unless, perhaps, in Pisa (I was greatly tempted to tilt some to exemplify their versatility).

Now that I finally had the pieces I thought I needed to render a Temple, I began the painstaking journey of construction. Although it did not take me as long as Solomon, I feel there is a direct relation -- i.e. (time to build Solomon's Temple)/(Real Life) = (Time it took me to render something similar)/(graphic life). Placing the blocks was pretty tedious. However, it paid off in the sense that one would not be able to tell that the Temple is actually made of several discontinuous Blocks(slightly noticeable from time to time when exercising the eye movement feature due to gpu rendering). However if one has the knowledge that the temple was rendered with blocks, one can make a pretty good guess at the number and location of the blocks. I would also like to point out that every component of the Temple is entirely created referencing the dimensions of one Block (or a chain that leads back to that Block--where the stairs lead) of the Temple. In this way, I could "easily" port the code in the main method to a Temple class that creates everything in the scene just by specifying a few dimension of one block. However time constraints prevent such action.

Next I included what I call a “FirePit” in the front of the Temple (still specified relative to the Temple). This class was actually not too difficult to make since I already did all the hard work creating my “smaller” classes to be very flexible. The FirePit is actually quite similar to the SuperFancyColumn class. I then added another FirePit although not historically accurate, better aligns with project specifications.

When experimenting with my Stairs class, I noticed I could do quite a few things with them (for instance, provide a negative stair height), and I placed 4 ‘Stairs’ instances around the Temple that are unrecognizable as stairs due to their strategic dimensions. They are for a decorative effect so that the Temple doesn’t just look like a box.

Finally, the very last thing I added were the golden cones (well.. Columns..) around the perimeter of the roof--again so that the Temple just doesn't look like a box. This re-introduced some C++ pointer difficulty that I experienced earlier, but have not mentioned. It was slightly easier to rectify the second time. This has been a summary of ~30hrs, and here I gloss right over the roadblocks that made this project this duration (some mental). Throughout this project I also made extensive use of the cryph utility, using points to specify... everything. That really helped when making objects located relative to others.

The way you met project specifications:

- **1.** I started from the project2 directory.
- **1.** All instances of TEMPLATE_SubClass have been removed, and shall never be spoken of.
- **1.** getMatrices method completed
- **2.** As for 3 concrete subclasses, I would argue that I have 4 ‘distinct concrete classes’. While I technically have 7 different ModelViewWithLighting subclasses, I count some as equivalent to others due to the minor amount of changes needed between them (However, I feel like I would have a case for 7 distinct subclasses since the reason they are so similar in implementation is due to the design choice of making ‘basic’ objects, very, very versatile (which adds to their complexity). As sub-bullets in each category, I list their usages.
    a. <u>Column/HalfColumn</u>
        - Column: used as a buffer between base and main column of the SuperFancyColum class, used as decorative top to Temple, used as bowl for fire pit, used as bottom of bowl in FirePit
        - HalfColumn: used for the ‘riveting’ of the FancyColumn
    b. <u>Block/Stairs</u>
        - Blocks: used as main temple components including dark doorway, used as base for SuperFancyColumn and FirePit, used for each ‘stair’
        - Stairs: used as stairs, used as decorative siding
    c. <u>FancyColumn</u>
        - FancyColumn: used in SuperFancyColumn, and FirePit

    d. SuperFancyColumn/FirePit
        ■ used as their names imply.

In total 5 sets of stairs, 2 SuperFancyColumns (also instances of FancyColumn), 2 FirePits, * Blocks, * Columns

- **3.** *Multiple colors requirement*: golden columns, different color roof than rest of temple, ground colored.
- **4.** *Eye Requirement*: The eye is initially set to be up to the left of the Temple (in addition, can be moved around during rendering). Larger surrounding parameters are supplied to help the view not get cut off when moving about the scene (although not perfect).
- **5***. Lighting Req:* Not required (and I made no lighting changes other than the actually coloring of object, but that falls under #3).
- Sophistication: As I mentioned in my process description, any class can be instantiated anywhere in any orientation. This was not easy, and I think shows that I know my way around Affine Space and geometry in general. Also, creating the Fancy/SuperFancyColum was non-trivial.
- *per-primitive/per-vertex attributes*. I made few changes to these and believe I am using them correctly.
- *Project Report*: Here it is!
- *Demonstration of understanding of the API, GLSL and use*: For a specific example of the API knowledge, I use a new GLFW method to get what keys are depressed. And in general, it all runs!

Interactive Controls:

‘a’ key -- pan left
‘d’ key -- pan right
‘s’ key -- pan down
‘w’ key --pan up
‘u’ key -- reorient the scene what really is ‘up’
‘i’ key -- zoom in (on the center of scene)
‘o’ key --Zoom out (from the center of scene)

The thing (or things) that caused you the most difficulty while developing the project:

- OpenGL errors are NOT helpful. Mostly because there aren’t any. It just gives you a black scene (if you’re lucky). This really puts one’s debugging skills to the test.
- Also, I had some C++ problems. For instance in one place in my code, the ONLY way I could find to get it to work properly was to include “**array[i] = &(*(new ...**”. I could have really used a brush up on C++ before taking this class I think. I’d say a good portion of my error are just C++ errors.

- I also found it very difficult to point to cryph::AffPoints. Perhaps again due to not completely knowing C++ and not working in it for a long time.
- Also related to arrays in another place, for some reason it seg faulted when I assigned a value to an array a values, but I could change the implementation to an array of pointers and take the pointers to the objects and it worked fine! Whyyyyyy. This would have saved probably at least 3 hrs total dealing with arrays on this project.
- I wouldn't say the math part was the hard part, but instead satisfyingly challenging.
- Something that might be helpful to put in the common problems is that to declare an array of a Class in C++, you have to implement a default constructor. The error code doesn't make that crystal clear. But,.. then again common problems is about GLSL things and this is C++.

  So I'd say the main problems I had were C++ arrays and unhelpful GPU errors (none). It sure would be nice if someone built a GPU emulator of some kind that produced helpful errors one could test a GLSL project on!

Any unique things you did in the project that you especially want me to notice while grading:

1. The movement of the eye point (left, right, up, down, up correction, in, out)
2. Anything can be put anywhere in any orientation!
3. Registering multiple button presses simultaneously
4. The entire Temple is built relatively (aka easily movable and scalable)
5. The math behind the SuperFancyColumn/FancyColumn classes (you can specify the number of HalfColumns around the outside to make it fancy. So long as the number is even!)