Programming Assignment #7

Due Dec 15 by 11:59pm **Points** 4 **Submitting** a text entry box or a file upload
Available until Dec 20 at 11:59pm

This assignment was locked Dec 20 at 11:59pm.

Due: Wednesday, December 15th (at midnight).

Late Policy: Assignment submission will remain open for 5 additional calendar days after the formal deadline. Late submissions will be penalized by a maximum of 1 point on our 4-point scale (i.e. if you submit 5 days late and your assignment was otherwise deserving a "3", you might get a grade as low as a "2" due to this penalty). Some leniency might be exercised on this general rule *(for this last assignment we might be a bit more lenient for assignments turned in no more than 1-2 days late, but no assignment will be accepted after Dec 20th, i.e. the day of the final) based* on how late the assignment was, and whether turning in an assignment was a repeating occurrence or a one-time event.

Synopsis: You get to use WebGL to create a 3-dimensional scene, with your own shaders, models, and associated data (vertex attributes, colors, normals, textures, etc).

Learning Objectives: To familiarize yourselves with the complexities of using the WebGL API and the GLSL shading language to craft a 3-dimensional scene. This includes defining vertex attributes, dispatching them to the GPU and using them within shaders, using vertex colors, normals and texture coordinates (when applicable). Also, using one or more textures to create a more rich appearance, if desired.

Evaluation: Based on our 4-point grading scheme, as discussed in our introductory lecture. You get a check (“3”) if you turn in a viable, and complete submission (even if it just draws a rectangle like the example in the tutorial). “Above and beyond” grades (i.e. a "4") will be awarded for people who have crafted something particularly cool. As a general rule, no more than 1/3 of all assignments turned in (the very best ones, that is) will be considered for a "4" grade.

Collaboration policy: This is an assignment to be done individually. Code not written by you needs to include proper attribution (see **this post (https://graphics.cs.wisc.edu/WP/cs559-fall2020/2020/09/10/collaboration-policy/)** here). It is always ok to use code provided in our in-class examples as a starting point, but you need to add your own effort to raise those examples (or other sources) to what is asked by the programming assignment (i.e. finding some code on some online forum that does all the job for you that is needed to satisfy the assignment is not the intent, if you haven't added any of your own effort to it). If you use somebody else's code (other than our GitHub examples), make sure to clarify in your submission notes what **you** did, and what you repurposed from the external source.

Hand-in: Electronic turn-in on Canvas. Make sure that you turn in all files needed for your program to

run. It is acceptable to turn in a single HTML file with your program, but even preferable to separate your code into an .html file and a separate .js file containing the JavaScript code, similar to the examples in our **GitHub repository (https://github.com/sifakis/CS559F21_Demos)** (see, e.g. Demos in Week10/ Week11/). If you submit anything else than a single HTML file, please put everything in a single ZIP archive. Feel free to use the copy of the glMatrix library included in our examples in the GitHub repository (or use them as a starting point) if it's convenient. ***It is not acceptable to submit a link to JSbin for this assignment!***

Description

our task will be to create a 3D scene, visualized using the WebGL drawing API (as opposed to the "Canvas" 2D drawing API we have used up to and including homework assignment #5). We have seen several examples, in-class, about using this interface, which you can find in subdirectories **Week10/** and **Week11/** of our **GitHub repository (https://github.com/sifakis/CS559F21_Demos)** (check out **this (https://jsbin.com/hagakel/)** and **this (https://jsbin.com/zivofiw)** URL for JSBin versions of some of these examples). When authoring such three-dimensional visualizations, it will be your responsibility to do the following, among others (while online tools like shdr.bkcore.com would do many of these things for you) :

- Write your own vertex shader and fragment shader; a relatively clean and straightforward way to do this would be to include them in the body of <script> blocks in the HTML code, as we saw in the examples above. *You could, in fact, write more than a single pair of fragment/vertex shaders, if you chose to, if you have several objects, each of which requires a different shader.*
- You should compile and link the two shaders into a "program", as we saw in our examples. *You can create more than one "program", if you wish to use a different shader routine for different objects you are drawing.*
- You should define your own vertex attributes, and uniform variables as we saw in our examples. This involves keeping pointers to attribute/uniform variables by querying the linked program (as we saw in our examples), and also defining the data associated with vertex attributes (and uniforms), dispatching that data to the GPU, and taking any other necessary operations to do the drawing.
- You should define the *geometry* of the model(s) being used, by providing a indexed set of vertices that make up triangles. This should also be buffered to the GPU as we saw in class, and used in a *drawElements()* call inside the draw loop.
- You should create all necessary transforms that the shaders might need, and dispatch those to the GPU (typically, as "uniforms").
- You should load texture images (if you chose to include texture mapping in your implementation), and furnish texture coordinates for your models.
- You should use the Z-buffer visibility mechanism to leverage the GPU's capability for performing visibility queries.

In order to secure at least a "satisfactory" ("3") grade in this assignment, you can think at a minimum of the world you created in your Programming Assignment #5, and ask yourself how you could

implement a similar appearance, but using WebGL this time (ideally, this appearance would be significantly richer!). The minimum requirements will be as follows :

- Your scene should include at least one “polyhedral” object with multiple shaded (as opposed to be drawn as “wireframe”, only by their edges) polygonal facets. Those will be typically be comprised of triangles. Your entire object cannot be all flat! (unless if you include several objects in your world, in which case it’s ok for at least one of them to not be flat). We would like to be able to visually appreciate that the Z-buffer visibility algorithm is actually working … “front facing” triangles/polygons, should hide parts of the objects that are located behind them.
- You should either include diffuse & specular shading in your objects, *or* use texture mapping (maybe both!). It will not be acceptable to have the entire polygonal object (or objects) show up as a single color, without any variation due to lighting. It is perfectly fine to use vertex colors to customize the coloration of different parts of the model.
- We would like to see you use at least **three** different vertex attributes in your shader. Those could be, for example position/color/normal, position/normal/texture-coordinates, etc. If you use more than one shader pairs in your implementation, this restriction is relaxed to: at least *one of the shader pairs* should use at least 2 vertex attributes.
- There should be a way to enact some change in the scene, *different than just controlling the camera position.* This would typically be some modeling transform that moves some part of the scene with respect to the world coordinates, some motion along a curve, a hierarchical modeling apparatus, etc. But at the very mininum, some modeling transform should be applied (in our in-class examples, this was done by a “rotation” transform applied to the cube model).
- There should be some way to affect the placement of the camera relative to the scene. In our examples, we have a slider that spins the camera around the scene … similar requirements were stated in programming assignment #5.

Of course, we would like to think you will extend beyond these bare-minimum requirements. Here is a list of possible embellishments that you might wish to consider … doing several of them, or doing them in particularly creative and aesthetically interesting ways will make you competitive for an “above and beyond” (“4”) grade.

- Combine non-trivial lighting *and* texturing. We would almost expect this as a prerequisite any submission that would have a chance to compete for a “4” … you should use both specular/diffuse lighting, and texture mapping at the same time (or for different objects).
- Consider modeling several different objects, subject to different modeling transformations, maybe implementing a hierarchical model, and maybe using different shader programs (i.e. multiple pairs of vertex/fragment shaders, each compiled into its own “program” and used via the *useProgram*() call before drawing this component). Think whether it makes sense to have multiple modeling transforms for a hierarchical model (separate “uniforms” if using a single shader?), or if you want to use different programs altogether to draw those.
- Consider using multiple textures at once, or in fancy combinations (think of the “decal” texturing **trick** **(https://jsbin.com/lonokos)** we showed in class, for example).
- Implement some complex objects, or objects with intricate motion.
- Implement some interesting camera motion.

Supplemental readings

The following supplemental/optional readings can be used to complement our in-class discussion. As far as the exams go, you will only be asked for concepts that were covered in-lecture, but the following readings could be useful in your review, or if you want to go into greater depth on some of these topics than we explicitly discussed in class.

From Foundations of Computer Graphics :

- From Chapter 4 **[Link] :** The most relevant sections are : Introduction, Section 4.1, 4.5, 4.8.
- From Chapter 8 **[Link]** : Focus on Sections 8.1 (mostly subsection 8.1.2), 8.2, and 8.3.
- From Chapter 11 [Link] : This is all good information. You may skip 11.1, and 11.3.1. For sections 11.4 through 11.7 don't worry about the formulas.

From The Big Fun Graphics Book :

- From Chapter 15 [Link] : All good material – you may skip Section 15.2. The explanations are more technical and in-depth than what we covered in class; don't worry about this, for the exam you will just need the depth of exposition we covered in lectures.
- From Chapter 16 [Link] : This is all useful and accessible (too bad the chapter is incomplete!). Again, this gets more technical with deeper mathematical foundations of the techniques, which is informative if you want to understand more about the concepts, but just the level of coverage we had in our lectures will be well sufficient for the exam.

From Real Time Rendering :

- Chapter 6 **[Link]** is a fantastic reference for texturing, although it includes *substantially* more detail than we covered in class (or needed for your exam). It's a great read if you want to learn more about these techniques though. You can very safely **skip** Summed Area Tables (p. 167) and Anisotropic Filtering (p. 168), as well as Sections 6.2.3, 6.2.5, 6.2.6, 6.3, 6.4, 6.5, 6.6., 6.7.4 and 6.7.5.