Programming Assignment #6 : Drawing in 3D revisited – WebGL style

by Eftychios Sifakis on March 20, 2018

Due: Tuesday April 3rd**.** *Note: Due to the extra time allowance, the submission system will remain open only for 2 extra days (until Thursday April 5th) for late submissions*

Synopsis:
You will create an interactive visualization of a 3D model using WebGL and GLSL shaders. In principle, you will re-create the experience that was asked of you in Program 4, but instead of doing the drawing using the (2D) routines of the HTML5 canvas, you will leverage the WebGL API and provide your own shaders to accomplish that goal. Some of the challenges that Program 4 was based on will not apply in this case (e.g. visibility will be handled using the Z-buffer, not your explicit implementation of painter's algorithm), while new concepts related to how WebGL works will now require your attention (e.g. interfacing the WebGL context with your custom shaders, communicating *attributes* and *uniforms* to your shaders from the JavaScript code, organizing your geometry data into buffers, etc.)

Learning Objectives:
To help you understand how the principles of the graphics pipeline materialize into a concrete drawing task, and understand how WebGL enables you to formalize and orchestrate this process. We hope that Program 5 has familiarized you with shaders themselves; the important next step that you will take in this assignment is to understand how they fit within the broader WebGL API (and the rest of your JavaScript application).

Evaluation: Based on our 5-point grading scheme. You get a "satisfactory grade" (3) if you turn in a viable, and complete submission. Higher scores will be awarded for people who incorporate additional features.

Handin: Will be as a Canvas assignment [Link]. Make sure that you turn in all files needed for your program to run.

Description

For this assignment you are asked to create an interactive drawing of a 3D object using the WebGL JavaScript API and your custom-implemented shaders. In terms of the geometric nature and complexity of the objects you are expected to visualize, this assignment is similar to Program 4: you will need to create a 3D object which can be relatively simple (e.g. a polyhedron, modeled as a closed surface, split into triangles — but hopefully you'll do something more interesting!) and take the appropriate actions to render a 2D view of it, using perspective projection. Similar to Program 4, the object that you choose to show needs to be at least complex enough to help you visually appreciate perspective projection, and demonstrate the need for visible surface determination (i.e. you will need something more complex than a flat object on a plane). The image rendered needs to depend on parameters that the user can interactively vary, for example using sliders, input boxes or other interface elements. These parameters at the very least need to have an effect on the 2D projected shape being drawn. For example, they can alter the placement on the object in the 3D world, alter the placement of the camera relative to the object, or vary the parameters of the perspective projection (optionally they may control even more aspects, such as colors, procedural textures, lights, or articulation of hierarchical models).

Since the visual content of what you must deliver can be quite similar to what Program 4 was asking, it is important to emphasize what you *must* do differently. In P4, you were explicitly asked to implement actions that are normally the responsibility of the graphics pipeline. For example, prior to calling any actual drawing commands, you had to explicitly transform vertices into the camera coordinate system, apply the projective transfom (using algebra, via the twgl.m4 library), apply a viewport transform to the resulting 2D values (which were in Normalized Display Coordinates), and only then actually call the canvas drawing functions. In your *final deliverable* to this assignment it is expected that the vertex data provided to the vertex shader will be in *world coordinates* (or even object *local coordinates,* if different than the world coordinates) — it will be the shader's responsibility to apply the necessary transformations to transform these 3D locations into an (x,y,z) triple appropriate for drawing (where (x,y) is the actual display location, and z will be used for visible surface determination via the Z-buffer). The transformation itself is something you will build outside of the shader code, and communicate to the shader as a *uniform* variable (of type *mat4*). Also, instead of implementing your own technique of visible surface determination (e.g. painter's algorithm), you will let the Z-buffer do this task for you — again, it will be your responsibility to produce appropriate z-values in the vertex shader in order to allow the Z-buffer to do its job.

The shaders you will implement for this assignment do not need to be overly complex. Chances are that the "simpler" of the 2 shader pairs that you crafted for Program 5 will be fully sufficient (that being said, cool looking alternatives are welcome and can give you project points). In terms of visual complexity, at a minimum you are asked to have variability in the colors you use for the triangles (or polygons) your 3D model is made of. It is ok (for this assignment) if you simply specify a single flat color to be used for any of these triangles. Or you can be more ambitious and have the coloration of model faces depend on something more intricate (e.g. a light, or the orientation of the triangle relative to the camera). Also note the differences between what you had to do for Program 5 and this assignment: in P5 the shader sandbox you used already took care of generating transformation matrices for you (and exposing them to you as *uniforms*, accessible by the vertex shader), and also took care of the details of communicating geometry data to your shaders, by populating the appropriate buffers. In this assignment, you will handle those API elements yourselves. You will have to communicate variables between the shader code and your host JavaScript program, organize your geometry data (e.g. vertex position, colors) into buffers, and bind them appropriately.

In class, we discussed that setting-up things in WebGL (e.g. compiling/linking/attaching shaders, binding attributes/uniforms, setting up buffers) can either be done by directly using the WebGL API calls that accomplish these (admittedly, sometimes pedantic) tasks, or using a toolbox (like *twgl*) to aid you in the process. This is exemplified in the code samples we discussed in class. For this assignment, you are free to follow either approach. Also, it is our expectation that you *could* (although you don't have to …) incrementally build the entirety of your program starting from one of the provided examples and only appending a handful of new lines of code at a time, while still seeing something visually informative as your output.

The description above conveys the basic features that your implementation should have. Above-and-beyond implementations could venture in the following directions:

- Use hierarchical 3D objects that can be interactively manipulated. The proper way to handle such models would be to also pass the model transformation (i.e. from *local coordinates* to *world coordinates*) down to the vertex shader (who would have the responsibility to apply it), while the geometric components of the model themselves will be entered in the vertex buffers in *local* coordinates.
- Compute normals, communicate those to the shaders, and use them to control the coloration of triangles in an interesting and non-trivial way (i.e. breaking from the simple solution of attaching a single, constant color to every triangle).
- Model several objects, using more than a single vertex/fragment shader pair for different objects of your scene.
- Implement an exciting and cool-looking shader, with graded color transitions within triangles, or implementing an interesting pseudo- (or procedural) texture.

Implementation hints and tips:

- Use Your P4 as a reference – it's the same, except that you need to draw the triangles differently. All the transformations and whatnot are the same. Your user interface for the camera can be re-used.
- Remember that the coordinates that are the output of the vertex shader are in normalized device coordinates.
- Start simple. The examples given you in class draw a single solid color triangle in screen coordinates first. Then you can slowly add more things: transformations, more triangles, passing colors to the fragment shader, passing colors to the vertex shader, passing normals to the vertex shader, …
- Remember that we pass information per vertex – not per-triangle. You can start by making 3 independent vertices for each triangle, each with the same color and normal.
- You can pass whatever uniforms you want. You may decide to pass the model, view, and projection matrices separately (so there are 3 matrices) and multiply them together in the vertex shader. This way you might retain more flexibility.
- Don't forget to turn on the depth test. Don't forget to clear the Z-Buffer when you clear the window.

Previous post: Programming Assignment #5: Shaders

Next post: Lecture notes from weeks 7-8

- **This week in 559**

  Due Thursday 3/15 11:59pm

  - Reading Assignment #3

  Due Sunday 3/18 11:59pm

  - Programming Assignment #5

- **Recent Posts**

  - Extra instructor office hours (Thursday 3/22 @ 4:00pm)
  - Grade statistics for Midterm Exam
  - One-time change to instructor office hours (3/21/18)
  - Lecture notes from weeks 7-8
  - Programming Assignment #6 : Drawing in 3D revisited – WebGL style

- **Recent Comments**

- **Archives**

  - March 2018
  - February 2018
  - January 2018

- **Categories**

  - Assignments
  - Lecture Notes
  - News
  - Uncategorized

WordPress Admin