# CMSI 371-01
## COMPUTER GRAPHICS
**Spring 2015**

## Assignment 0326a

Time to start building your very own personal 3D graphics library!

## Outcomes

This assignment will affect your proficiency measures for outcomes *1b*, *1c*, *3a*, *3d*, and *4a–4f*.

For outcome *3a*, this assignment only covers a subset of the full graphics library that is expected to come out of this class, so it will have a maximum proficiency of | for now.

Similarly, this assignment applies only to the vertex shader aspect of *3d*, so that outcome will have a maximum proficiency of | until a future assignment expands that to include the fragment shader as well.

## Not for Submission

If you have access to the Angel textbook, the following readings will add depth and detail:

- (already mentioned) *3D graphics overview and pipeline:* Sections 1.1–1.10 (pages 1–36).
- *Graphics programming:* Section 2 (pages 39–94).

## For Submission

For the following tasks, start by copying one of the *hello-webgl* bazaar samples into *homework/pipeline* on your `git` repository.

### Envision a Scene

It's a good idea to have *some* notion of what you want to render by the end of the semester. Create a stub web page that will eventually hold the final version; for now, you can use it to demonstrate and test the work listed below.

### Define a Shape Object

Refactor the ad hoc "shape" objects from the sample code into a bona fide JavaScript `Shape` object. There are no design specifics here beyond the ability to invoke `new Shape(…)` with arguments of your choosing, producing a *polygon mesh* data structure of your own design, that subsequently works with the refactored drawing code (see next section).

### Modify the Drawing Code Accordingly

Rework the scene drawing code as needed in order to work with your new `Shape` object. Remember to take into account the preprocessing that is done per shape.

*Protip:* Keep an eye out for activities that can be refactored into prototype methods. Candidates for such conversion can be found both in the main drawing code and in the current *shapes.js* utility object (*cough* conversion to WebGL-ready, mode-specific arrays *cough*).

### Implement Shape Groups

Extend your `Shape` object so that it can handle *composite* or *group* objects: that is, allow `Shape` objects to have children, each of which is also a `Shape`. Your implementation should allow for arbitrary depth, and of course the drawing code should be able to handle this without a problem. Yes, you are implementing a tree. And now you know why the data structures course is a prerequisite to this one.

### Supply Unit Tests

By all accounts, `Shape` truly is an object with methods and expected behavior, so yes you want to include unit tests with your library. No specific testing framework is required, but if you don't have a preference, QUnit will work just fine.

### Optional: Provide Import/Export

You may find it useful to implement import or export functionality. Importing allows you to represent your shapes as distinct files, as well as download some from the web or create custom models using 3D applications. Exporting allows you to potentially 3D-print your `Shape` instances, if you export to the right file format. All potentially fun **:)**

This optional functionality will be considered when assigning proficiencies, but in fairness the more complex models that will be facilitated by these features will not have an effect.

## Expand Your Shape Library

With your `Shape` all set, you can now build a library of pre-built/predefined `Shape` instances. You may use the JavaScript prototype mechanism to create "subclasses," or take a factory approach (i.e., similar to what the sample code currently does in *shapes.js*). Either way, for this assignment we want:

- A sphere implementation…because anyone who does 3D graphics simply has to; the Angel textbook has an example but it isn't too hard to come up with one on your own, either.
- At least two (2) more shapes, based on what your envisioned scene will need. Feel free to implement simpler building blocks then use the composition/grouping functionality to define more sophisticated objects.

## How to Turn It In

You will want your basic shape code to reside in a reusable *shape.js* file. Its use would be like any other JavaScript library: reference it in a `script` element, and all other JavaScript code should then be able to build `Shape` objects with it.

Unit tests should reside in separate files. Place them in a distinct *tests* folder so that they do not get mixed up with the library code.

Your shape library can take a number of approaches. If using a factory approach, a single top-level factory object with functions for producing shapes can work, although that may grow really quickly. Alternatively, you may take a similar approach to your sprites and define one pre-built shape per JavaScript file. Or anything in between will work—just remember to code it well, keep it readable and maintainable, and adhere to best practices.