Assignment #4

Due: Thurs, May 23, 2018 at 2:00pm

Points:
Collaboration: None Permitted

This assignment is a guided tour through the remaining features of your codebase and a deeper tour of WebGL programs in general, especially shader programming.

Submission: Follow the instructions carefully to avoid point reductions. Submit on CCLE a zipped file (UID.zip — e.g: 398342394.zip) that includes all the template files (not just main-scene.js). Don't include any .git folders

Setup Information: Clone the Assignment 4 code linked on Piazza. Open your project as always, by hosting it on localhost, and dragging your new repo's folder into Chrome DevTools to create a workspace. Even if the green dots show up, if you forget to create a new Chrome workspace you might accidentally edit where your A1, A2, or A3 files are stored.

Getting Started:

Write a program using our JavaScript template that **draws the solar system we specify, showing us that you know how to match our scene exactly.** Animations showing the whole solar system are included below. The structure of the scene must be as we specify. When in doubt, refer to our animated GIFs and match them.

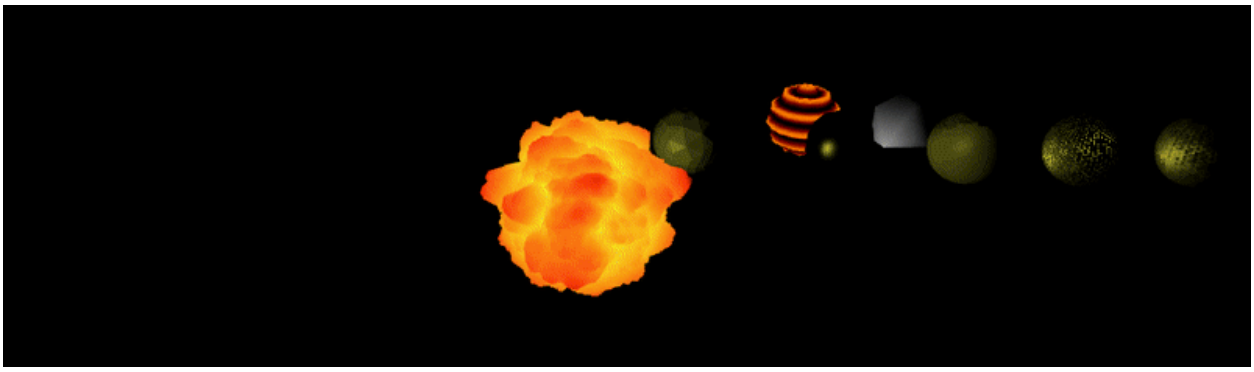
When you open your project, you will see the familiar `Transforms_Sandbox`, but with texture images. Replace this scene with the solar system we specify in detail below.

Before you work on anything, **remember to comment out the "test scene"** at the bottom of your scene's display function. Your controls and camera location won't work until you do.

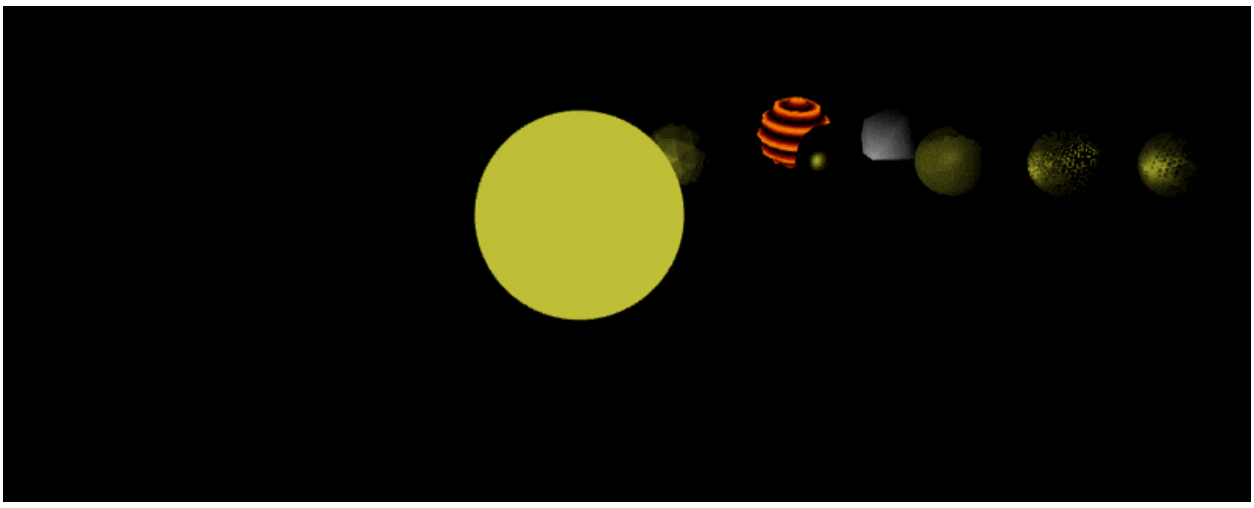
Items in your code for you to fill in are marked `TODO`. You probably won't need to edit outside of those items, or any file besides `main-scene.js`, the one where you build your solution. The lengthy instructions inside the code comments of our partial `Solar_System` class in that file should be sufficient to start implementing the required features.

Animations:

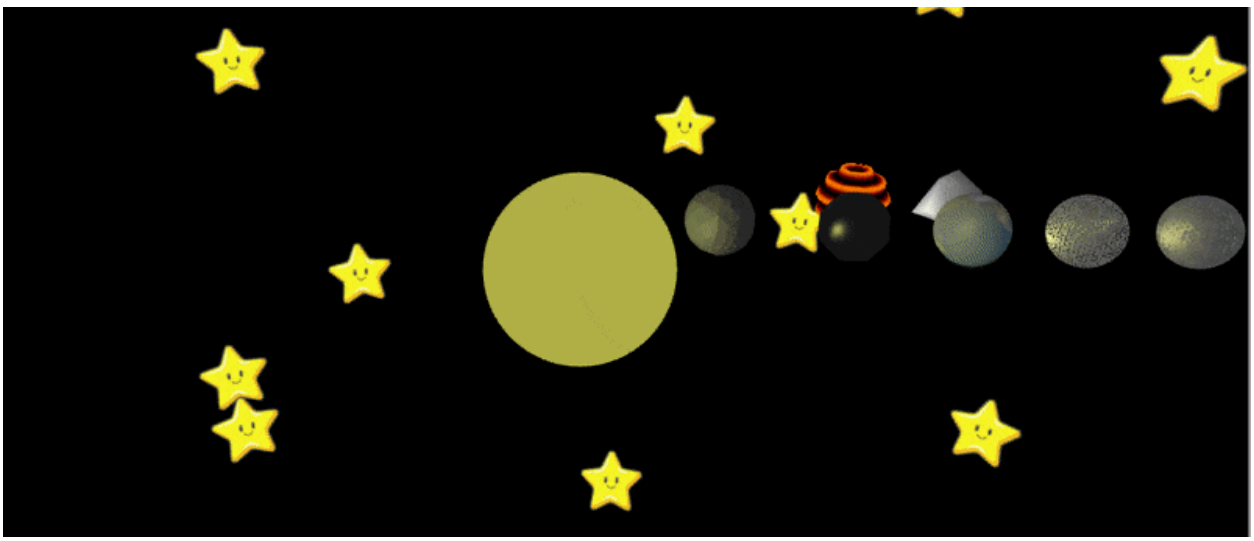
With EC2 sun:



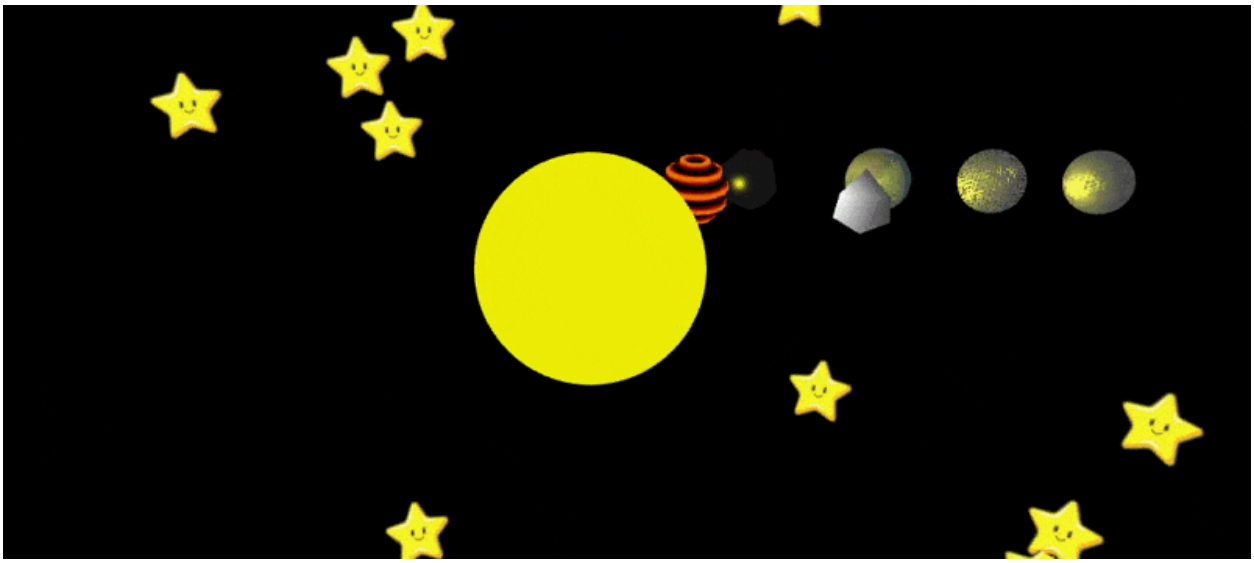
Without EC2 sun, and lights switched off:



With lights switched on:



Close ups of planets, using the Camera_Teleporter control buttons::



Remember: Read and understand the document "When your JavaScript code isn't working" from Assignment 1 for important instructions about using a debugger for this assignment and all the time!

Note: Unfortunately Chrome's debugger cannot step inside shader programs, since GLSL code runs on the GPU. When your shaders don't compile, you can at least get the compiler error message though. To do so, use your debugger to stop at a breakpoint at when the shader compilation fails. The red exception message it throws upon error contains a verbose shader compiler error message that will guide you.

- This message includes line numbers, so it would be helpful to be able to find line numbers in your shaders. Seeing line numbers is possible. Nearby your breakpoint, you should see a call to your `this.vertex_glsl_code()` or `this.fragment_glsl_code()` responsible for the failed compile. Whichever one it was that didn't compile, remember it and copy-paste it into the "console" tab of DevTools. Append `.split("\n")` to it and press enter. Running this line of code should return an array that you can expand out. This is your shader string split up by line number. Now you can see which line number the compiler complained about.

Steps:

1. **Shapes:**
    - Fill in the constructor of `Solar_System` to instantiate the shapes that you will need to draw each planet, as follows.
    - Shape list: Declare a `Cube` object, a `Planar_Star` object, and variations of `Subdivision_Sphere` objects, one for each of these complexity levels: 1,2,3,4,5, and 6.

      (A subdivision sphere takes its complexity as an integer argument. Each additional complexity level splits every triangle into four more triangles. We cannot make this number very high, because each increase quadruples the work the GPU must do to draw so many triangles - eventually causing lag).

    - Additionally, the sphere of complexity 3 should be built for flat-shading. To do so, it should not be declared as a `Subdivision_Sphere`, but as a `Subdivision_Sphere_Flat`, which is not defined yet. To define it automatically, put this earlier in your Scene's constructor code:

      ```
      const Subdivision_Sphere_Flat =
      Subdivision_Sphere.prototype.make_flat_shaded_version();
      ```

    - **[5 points]** Figure out how to modify how textures display on the subdivision sphere of complexity 5 after it is created. Increase all of its texture coordinates on both axes, multiplying by a factor of 5. See part 4iv to see why.
2. **Materials:**
    - Fill in the constructor of `Solar_System` to instantiate the materials that you will need to draw each planet, as follows.
    - Refer to the individual planet descriptions as a guide for which materials to make.
    - Unless otherwise noted, set each material's "shader" reference to point to `phong_shader` if we don't specify a texture. If we do, set it to `texture_shader` or `texture_shader_2` (which shows the effects of lights better). A shader reference is the first argument when constructing a `Material`.
    - Give materials a color by storing a `color` member in them. When you use a material, don't say `.override()` each time like we did in assignment 3; the color is already stored inside. Later on (part 5) we will use `override()` for a different reason (to use materials with a different value of `ambient)`, but we won't need it for `color`.
    - The second argument when constructing the material is a JavaScript dictionary/object {} of options (also called "options object"). These options include the coefficients to the Phong Reflection Model formula, weights

that usually range from 0 to 1. The default options specify materials as follows: They do not react to ambient light at all (set by the field `ambient`) and fully react to diffuse and specular light components (set by the fields `diffusivity` and `specularity`). The default specular exponent called `smoothness` is 40. The default `color` is opaque black (0,0,0,1).
    - For your extra credit shaders, your option objects will use different fields than those, or none at all (in the case of shaders with no customization).
3. **Sun:**
    - **[3 points]** Draw the sun centered at the origin. Use a subdivision sphere of complexity level 6.
    - **[2 points]** The sun should use a material that has full ambient color (`ambient: 1`) because it produces its own light, and needs no light sources. The sun material's `shader` member should be a regular `Phong_Shader`, unless you do extra credit part 2.
    - **[5 points]** A variable is declared for you in `display()` called `sun_size`. It varies over time. Use it or something like it to create a scale matrix for the sun. Use given variable `smoothly_varying_ratio` or something like it to create a color that turns yellower as `sun_size` increases, and bluer as it decreases. Assign this color to your sun's material.
    - **[5 points]** Assign your sun's same color to a new light source in your scene. This should be the only `Light` object stored in your `program_state.lights` array. Lights are stored in homogeneous coordinates; the light should be a point-based light, located at the origin 0,0,0. To construct a `Light()` object, pass in its coordinates, then its color, then a size.

        Note: For the light size use ten to the power of `sun_size`. That seems like a large number, but the size is used to attenuate the light source's power over the square of distance, which is itself a large number. In JavaScript, ** is the exponentiation operator. Since the light's size is changing and not the brightness, you should see the outer planets darken more than the inner ones whenever the sun shrinks.

4. **Planets:**
    - For each of the following planets, leave them their original unscaled size. They should orbit around the origin point 0,0,0. The smallest orbit shall be 5 units away from the origin and each orbit after shall be 3 units farther. The rotation speed should be no faster than t radians, with each farther planet revolving at a slower rate than the previous. Besides revolving, each planet should also slowly rotate (around its own axis).
      Note: These planets are almost uninhabited, so it's OK if they collide together during their movement.

Note 2: If outer planets swing too fast around the sun, the camera controls might jitter when following planets.

- There are 7 total moving bodies: 5 planets and two moons (one moon is extra credit).
- Leave the ambient lighting of each planet as the default value of zero.
- **Planets list:**
  i. **[5 points]** Planet 1 is made of jagged rock. To get the jaggedness, it should be flat shaded (use the flat shaded sphere of complexity 3 you made in the constructor). The rock material uses Phong shading, a medium gray color with high diffusivity (close to 1) and low specularity (close to 0). Assign those values in the material. To see how Phong shading works, observe the `Phong_Shader` class definition in your resources file. Search for function named phong_model_lights() and notice the definitions of variables `diffuse` and `specular` inside it. These are the terms of the Phong reflection model's formula.
  ii. **[2 points]** Planet 2 is totally silver, shiny almost like a black mirror. It should reflect 100% of specular light sources but have 0% diffusivity. Use a slightly darker medium gray base color and a sphere with low complexity (2). Despite the low triangle count you should still see a mostly round specular highlight reflected in it.
  iii. **[3 points]** Planet 3 is just the earth, of course. Use a fairly complex sphere (4). Texture it with the earth's image by assigning to the material a texture of `new Texture( "assets/earth.gif" )`. The earth isn't very shiny so use mostly diffuse light reflection. Assign a medium gray base color - the texture image will take care of the rest. To see how texturing works, observe the `Textured_Phong` class in your resources file, and how its fragment shader program differs from the `Phong_Shader` by passing a `Sampler2D` variable to the fragment process. A `Sampler2D` provides an interface for careful sorts of sampling operations into your image file.
  iv. **[5 points]** Planet 4 is made of bricks (our "bricks.png" image). Use `texture_shader_2` for this one, which includes a bump-mapping-like effect to make the light seem to shine off individual bricks. The material should use full diffusivity and specularity (1). Since Planet 4 is very far away from the light, the reflection of our point light source would normally be tiny (not showing our bricks well), but fortunately, our brick planet is rougher than the other planets and this creates a larger shiny spot than normal. To accomplish that, set the `smoothness` exponent argument of the material to 10 (the default was 40).

Our included bricks image doesn't show very many bricks in the picture; that doesn't look good up close to anyone walking on the planet. To give it a higher brick count, use the subdivision sphere of complexity 5 that we already modified in part (1d) to have higher texture coordinate numbers (multiplying each U and V coord by 5).

> Note: The texture coordinates of this sphere should now range from 0 to 5 on both axes, instead of from 0 to 1. By default this causes the texture to repeat itself, since texture coords always get modulo'd into the final 0...1 range so that they can still sample somewhere in the picture's bounds even if the provided coordinates go out of this range.

Additionally for Planet 4, let's experiment with texture sampling methods. You must choose an alternate algorithm (nearest neighbor) for looking up texture pixel values. By default, our textures use a good sampling method. For this part, we'll override that to use a bad method.

- Note: The default (good) method is tri-linear filtering using mip maps. Tri-linear filtering samples your image smoothly by doing linear blending along three different axes: X, Y, and scale. Blending scale is accomplished by blending samples from different mip maps, which are progressively smaller (representing bigger pieces of the original image, instead of more locally-relevant pixel details). The worst sampling method, however, is called nearest neighbor - it simply samples one pixel of the texture image for every screen pixel, capturing only extremely local data.

Let's use that - pass the WebGL setting name in when declaring the texture, like `new Texture( "assets/bricks.png", "NEAREST" )`. Note that nearest neighbor sampling still has its uses - it is the only way to always still show the image's original full color or brightness level, without any blending or smearing.

v. **[5 points]** Planet 5 is identical to Planet 4, except we'll create a duplicate bricks material for it that does not pass in `"NEAREST"`. Both planets should look identical up close under the same light. But from far away, the differing WebGL "MIN_FILTER" settings of each texture will cause a wildly different appearance. Whereas Planet 4 will show extreme moiré patterns due to aliasing, Planet 5 will look smooth and not suffer from this (even far away), although the colors and brightnesses will no longer be as sharp.

5. **Stars:**

- **[5 points]** The `Planar_Star` class definition we already gave you in your file uses a for loop to build a very simple five-pointed star shape. The normal vectors for this planar star shape are trivial (all pointing along Z), but what should its texture coordinates be? You must fill them in yourself. Add code to the shape's constructor. You can use code similar to how the normals were built -- it's easiest to calculate texture coordinates directly from the positions using a mapping rather than trying to work them out point by point. Image coordinates must span from 0 to 1, but your star has an outer radius of 7 and therefore spans the range -7 to 7 on both axes. That should be enough for you to find the mapping and finish the texture coordinates.

  > Note: Don't try to turn a `position` vector into a `texture_coord` vector using nothing but vector operations, because a `position` is 3D whereas a `texture_coord` is 2D. There is no function that drops to 2D. You must create a new `Vector` manually with the two values in it that you need.

  Your constructor already stores 30 good random locations to draw this star shape in the variable `star_matrices`. Your `display()` function should loop through this array and draw one star at each matrix. This code must only run when the `this.lights_on` variable is true (we'll set that variable in the next part). Draw each star with a material that uses full ambient, no diffusivity or specularity (no interaction with light sources), and fully black color (but color opacity still should be 1). Assign a texture value of `new Texture( "assets/star_face.png" )` to the material so that it uses that image from your folder.

- **[5 points]** Create an interactive button on your Scene's visible control panel. Find the method `make_control_panel()` of your `Scene` (located in your own scene's class definition, not the one anywhere else). Within that, call the function `this.key_triggered_button()` to append a button to our panel. When you call it, pass in three arguments: A label (a string like `"Lights on/off"`), an array of keyboard keys that activate the button if pressed simultaneously (just use `['l']`, so that the lowercase L key controls it), and thirdly a callback function for the button to execute. Pass in a callback function that flips the boolean value of `this.lights_on`. Use an arrow function so that JavaScript doesn't lose track of `this`. The button on your page should now toggle the stars on and off when clicked.
- **[5 points]** We want the stars to light up the rest of the scene, especially so that we can see the texture images on planets 3, 4, and 5 better. This requires some amount of *ambient* light in the scene. What we really want is for each of these planets' materials to individually have a higher ambient

term when `this.lights_on` is true. Fortunately, rather than defining twice as many materials (another, higher ambient version of each one) we can instead create temporary materials with overridden values. Use the `override` function of a Material and pass in an object with the new setting. We have already created such a variable for you in your `display`, called `modifier`. Throughout your program whenever drawing any planet or moon (passing a material into draw), pass in a modified material instead. Call `.override( modifier )` on the material to generate a temporary new one that uses the setting in `modifier`, replacing the ambient term with a new value, thereby giving our scene its own working "light switch" the user can press. The `override` function works on colors and other settings too for coding convenient one-liners.

6. **Moons:**
    - Each moon should orbit at a distance of 2 units away from its planet, and be revolving (around its planet) and rotating (around its own axis).
    - **Moons list:**
        i. Moon 1 is for extra credit part 1. You can omit Moon 1 if you don't do that part. Otherwise, look farther down for Moon 1's instructions.
        ii. **[10 points]** Moon 2 uses a very jagged sphere (complexity 1) to give it the appearance of a meteor. For low-poly spheres like this, smooth Phong shading does not look quite right - the reflection of light off of it rounds out the edges, which looks unnaturally sphere-like for our jagged shape. To fix this, Moon 2 uses Gouraud shading. You must write the Gouraud shader below your Scene class. A skeleton for a `Gouraud_Shader` class is filled in for you in your file.

            Remember that with Gouraud shading, the fragment shader interpolates colors; with smooth shading, the fragment shader interpolates normals. For Gouraud, we can therefore compute our final Phong color before the vertex shader even finishes, moving the call to the `phong_model_lights` function to the vertex shader. The final color will be passed on to the fragment shader, which will do almost nothing now. Follow these steps to correctly move the Phong calculation:

            1. In the `shared_glsl_code()` function (for code that is included in both shaders), simply copy the `Phong_Shader` class's implementation of `shared_glsl_code()`, with one modification: change the two "`varying`" `vec3`s declared in it to just one `vec4`, called `color`. Color is all that we'll interpolate between vertices.

2. Copy the `Phong_Shader` class's implementation of `vertex_glsl_code()`, but change the declarations of `N` and `vertex_worldspace`. Those need to be moved into function `main()`. Declare them as `vec3`s, not `varying`, that are local to `main()`. By eliminating two `varying` variables, there will now be two fewer outputs to the fragment shader. Finally, copy over the entire fragment shader code from `Phong_Shader` to the end of the vertex shader's `main()` as well, since the Phong calculation should happen there now.

   As you do that, modify any lines that assign to `gl_FragColor`, to assign them to "`color`", the varying you made, instead. You cannot assign to `gl_FragColor` from within the vertex shader (because it is a special variable for outputting final fragment shader color), but you can assign to varyings that will be sent as outputs to the fragment shader.

3. Leave the fragment shader's `main()` function almost blank, except assign `gl_FragColor` to just equal "`color`", the varying you made earlier.
4. The moon is made of ice. It is fully white, with very high diffuse reflectivity and medium specular reflectivity.

7. **Camera:**
   - Use our provided initial camera matrix that looks diagonally down at the scene, far back enough to see the entire scene.
   - **[5 points]** At the end of your scene's whole `display()` function, store the matrix for each moving body and the sun inside the array called `this.camera_teleporter.cameras`. Since camera matrices work oppositely from the matrices of shapes that we draw, wrap each matrix in a call to `Mat4.inverse()` that you add into `this.camera_teleporter.cameras`.
     Performing that step for each planet/body allows our child/helper scene (called Camera Teleporter) to work, necessary for you and our graders to see your planets up close. It provides additional buttons on the page that smoothly move the camera between places. Any matrices externally added to its `cameras` member can be selected with these buttons. Upon selection, the `program_state`'s camera matrix slowly (smoothly) linearly interpolates itself until it matches the selected matrix.
     i. Tip: You don't want to warp the camera to the dead center of a planet, because then you'll only see its inside. Rather than storing the exact (inverse!) matrix of each planet, tweak each matrix a bit

before the inversion so you can see the planet, or maybe appear to be standing on it.

Remember to add the moons to this list when you make them too.

At our blending speed of .01, you will still have some leeway to control the camera while attached (especially mouse steering), although it will tend to pull you back to viewing the selected planet.

- As you teleport the camera, see if you can notice any undesired effects of blending matrices linearly this way to generate intermediate camera matrices. A subtle problem can be seen because our code snippet above uses linear blending instead of quaternion blending. Notice how the scene collapses sometimes.

Extra Credit

1. **Extra Credit Part 1:**

   [10 points] Make moon 1 be an animated black hole. Implement a particular custom shader and draw moon 1 with it. Don't add any properties to the material, and draw it with a subdivision sphere of complexity at least 4. This simple shader will not use any kind of lighting model; it will choose colors as a pure sinusoidal function of the ball's UV coordinates as they vary from 0 to 1 along the ball's parametric space (UV here is the typical spherical coordinate system). Besides assigning colors, let's use our vertex shader for something. We'll make a displacement shader, which disturbs the final vertex positions from their usual locations. We'll use a sinusoid for this as well, creating wave effects traveling up the latitude lines. The moving waves should vary from black to red, creating a black hole appearance.

   a. In your resources file, observe the class definition of `Funny_Shader` for an example of a simple shader that acts upon UV coordinates to generate colors arbitrarily. You can try drawing something with that shader to test it.
   b. Begin with the small code skeleton we have defined in your file called `Black_Hole_Shader`. For this shader we'll need to decide which uniform variables the shader program should receive from our JavaScript code. In your shader's `update_GPU()` function, pass JavaScript values into shader uniform variables:

      Send the GPU the only matrix it will need for this shader: The product of the projection, camera, and model matrices. Finally, pass in the `animation_time`, found inside `program_state`. Use the `update_GPU()` function from class `Funny_Shader` for reference. You

don't need to allow custom materials for this part so you don't need to forward any values from the `material` object.

> Note: When you're editing `update_GPU()` now or in future assignments, sometimes you need to tell JavaScript to send a value to some new shader variable that you made. How do you get the pointer to a brand new GPU variable from JavaScript? It turns out that the `graphics_addresses` object already present in our library already knows. By the time we run, it will have already retrieved pointers to all variables declared in your shader. Let's say you made a shader variable called `my_uniform`; you can just say `graphics_addresses.my_uniform` to denote the pointer you want to send a value to.

c. Now onto the actual shader program definitions. Specifically, the `void main()` is blank for both the vertex and fragment shader programs; fill these in to cause the GPU to store within the special GPU address called `gl_Position` the correct final resting place of the vertex, and store into `gl_FragColor` the correct final pixel color. For testing, you can try storing simple placeholder values into those special variables -- such as the original model space position value, converted from a `vec3` to a `vec4` like this: `vec4( object_space_pos, 1)`. You might find this page helpful for dealing with GLSL data types.

   For both shaders (or the shared glsl section), declare a `varying vec2` to pass a texture coordinate between your shaders (from the vertex program to the fragment program). Also make sure both shaders have an `animation_time` input (a `uniform`).

d. In the vertex shader, calculate the matrices times the `position` attribute as the existing shaders do. Store that in a new variable instead of using it for the final `gl_Position` value of the vertex, which we will base on a displacement function instead. The displacement should scale the point to change its distance from the origin (and thus from the sphere's surface). Vary this factor sinusoidally by a small amount as a function of V (from UV) and `animation_time`.

   When doing displacements, remember that your position is stored in homogeneous coordinates, and the w term (w=1) must be protected! If you scale the whole position value you'll mess up w. Scale the xyz components only.

   The input to your call to GLSL's `sin` function should not grow out of bounds or the math quickly loses precision; cap it using `mod( rate,`

`radians( 360. ) );`. Your rate can be based on linear factors of both the V texture coordinate (measuring latitude) and animation time.

Remember your vertex shader inputs: The current vertex's stored position and texture coord (UV), `animation_time`, and the final product of the projection, camera, and model matrices.

e. Lastly in the vertex shader, pass your texture coordinate to the next shader.

> Note: When editing the shaders, understand the difference between `texture_coord` and `f_tex_coord`. The first is an attribute (the value stored onto a vertex from your JavaScript). The second is a `varying` - an output your vertex shader must send to your fragment shader. Since it's an output, do not read from it! If you try to use `f_tex_coord` in any of your formulas from within the vertex shader, it will be undefined, and you will probably get a blank result.

f. In the fragment shader, using a similar function of the input UV texture coordinates and `animation_time`, generate RGB components of a color. It's probably best to use the GLSL `max` function to ensure no color component goes below zero. Store the result in gl_FragColor. You should see waves of color/brightness that move vertically along the sphere.

2. **Extra Credit Part 2:**
[20 points] Use everything you've learned so far to adapt a particular shader from online into your code. We will use the fireball shader from here:

https://www.clicktorelease.com/blog/vertex-displacement-noise-3d-webgl-glsl-three-js/

a. Read the nicely-made tutorial at that link. It describes how they use their fireball shader code as an input to the popular three.js JavaScript library. We will instead use the same shader code to fill in one of our subclasses of our familiar `Shader` object in our simpler library. The process is a little different, so you can ignore what the tutorial says about three.js and simply focus on the GLSL code. Fill in the empty `Sun_Shader` class definition in your file.

b. For adapting the code, their version of the shader code itself is not the most convenient version available posted online. Instead take the code from this live coding website linked by somebody in the comments instead:

https://shaderfrog.com/app/view/30?view=fragment

There you can see the fragment and vertex shaders, and can also run the shader and see output, with adjustable sliders for the uniforms. This might

be very informative to see, and their code is better.

Rather than passing in that many uniforms from JavaScript, it's fine for you to hard-code them in as constants in your vertex shader. Type in whatever values seemed to work best using the adjustable sliders. Then set them as constant values in global scope. Leave off the "uniform" keyword.

Make sure you're actually setting all the uniform constant values expected by the vertex program!

c. Additionally, the most convenient version of the fragment shader came from somewhere else: The comment section of the tutorial. There I found a version that does not depend on looking up a texture image just to get a collection of possible colors (which is excessive anyway; we can use simple math to make colors). Here is their suggestion for the entire `main()` function:

```
vec3 color = vec3((1.-disp), (0.1-disp*0.2)+0.1,
(0.1-disp*0.1)+0.1*abs(sin(disp)));

gl_FragColor = vec4( color.rgb, 1.0 );

gl_FragColor *= sun_color;
```

Where `disp` is the displacement distance from the origin, which needs to be figured out in the vertex shader and passed from there as a varying, and where `sun_color` is a uniform equalling the sun's material color we used previously in our JavaScript. I added in the last line, for applying it (the material color) to the fireball effect.

Those two variables plus another uniform we must provide for `animation_time` will be enough to cover all our shader inputs. Make sure they are declared in whichever shaders need them.

d. For the vertex shader, directly paste in all of the Perlin Noise code linked by the tutorial at the beginning of the program. Next, in `main()`, after any necessary declarations, use the vertex shader from the above shaderfrog link. This calculates a `displacement` float value. Pass `displacement` (scaled if necessary) on to the fragment shader as our `disp` varying so it can be used to affect color.

Also, use `displacement` to create an offset `vec3` based on our `position` attribute. Remember that this attribute is stored in object coordinates (the sphere's own system) which is perfect for our plan to

displace it from the center. Simply add the normal vector to it times `displacement` to generate our new value. You don't need any additional shader inputs to get the normal vector; just derive the normal from position using the fact that we start with a perfect sphere. Finally, use the new point. Apply the product of your projection, camera, and model matrices on the new point instead of on the original `position` attribute.

> Note: For some reason the tutorial's line of code for multiplying `position` by all the matrices also sets "2" for the w coordinate of position. Don't do that; it will mess up your sun size.

Your sphere's positions should now be moving around over time according to the smooth noise function described in the tutorial. Your fireball effect is complete.

Each required part must successfully draw and show up onscreen in order to count. **There is no partial credit on any individual requirement.** Implement the assignment in clean and understandable code.

If any parts are unclear, ask on Piazza.

Good luck, have fun!