TNM061 3D Computer Graphics
WebGL with three.js
Lab instructions 2024

Martin Falk

January 19, 2024

Contents

1 Introduction

During this lab exercise, you will be introduced to the three.js graphics API (threejs.org). three.js is a high-level *scene graph framework* for 3D graphics written in JavaScript. It builds on top of WebGL, a JavaScript API for rendering 2D and 3D graphics directly in your browser. Topics covered in this lab:

- WebGL in combination with three.js
- Scene graphs and transformations (see also previous lab)
- Vertex and Fragment Shaders
- Illumination (Lambertian shading and Phong illumination)
- Specular reflectance mapping

2 Theory

The Phong illumination model is quite often used in real-time rendering. It is a simple model which approximates illumination effects (see 04-real-time-graphics.pdf, slide 21). The model consists of material coefficients k_a, k_d, and k_s which scale the ambient, diffuse, and specular terms, respectively. The illumination depends on the surface normal $\vec{n}$, the light direction $\vec{L}$, and the view direction toward the camera $\vec{V}$. The direction of reflected light $\vec{R}$ is obtained by reflecting $\vec{L}$ on the normal $\vec{n}$.

$$I = k_a + k_d(\vec{n} \cdot \vec{L}) + k_s(\vec{R} \cdot \vec{V})^\alpha$$

Two vectors are combined using the dot product $\cdot$, also known as scalar product.

1. When does the diffuse term of the Phong model reach its maximum and when its minimum?
2. Explain why it is important to normalize all vectors when calculating the illumination for example with the Phong model?
3. Describe what shading is in the context of real-time rendering? What is the difference between Phong illumination and Phong shading?

3 Assignments

Similar to the previous lab, you will be modeling parts of the solar system. This time with three.js. Your task is to create and utilize a scene graph provided by three.js. You then illuminate the scene with a single light source. Vertex and fragment shaders are used for lighting computations. You should complete all of the steps below and your final rendering should be similar to the one depicted in Figure 1.

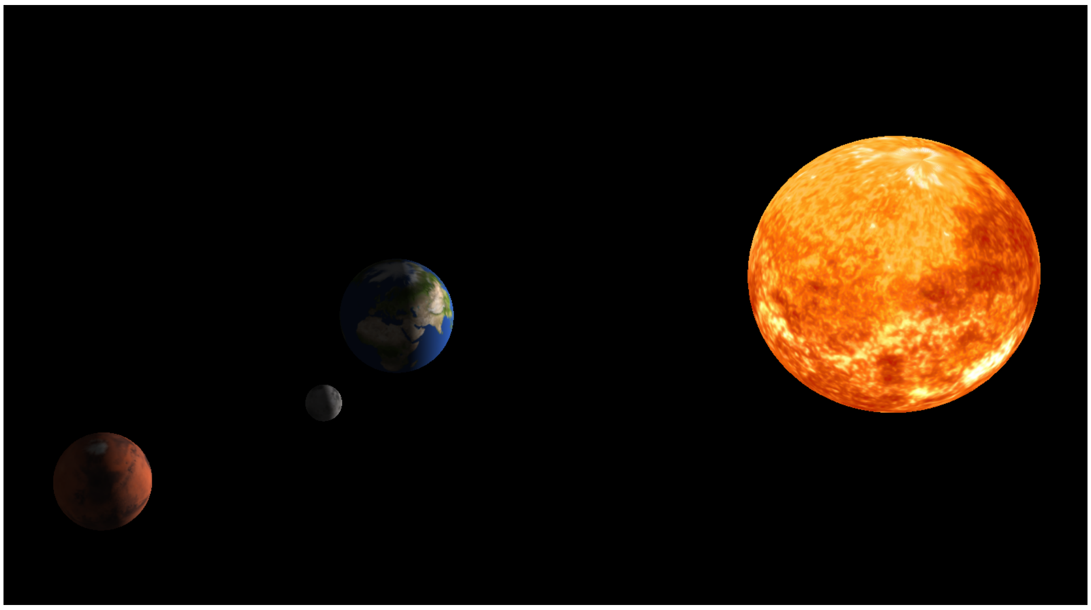

Figure 1: The resulting planetary system with basic illumination.

1. Get started with three.js. Familiarize yourself with the source files for this lab, `lab2.html` and `lab2.js`. Then open `lab2.html` in your browser. The HTML file contains an empty container which serves as canvas for WebGL. There also two basic shaders, a vertex shader and a fragment shader, which you are extending in step 7.
2. Enable loading the texture for the cube earth by uncommenting the following lines in `lab2.js`:

   ```
   const earthTexture = texloader.load('tex/2k_earth_daymap.jpg');
   materialEarth.map = earthTexture;
   ```

   Note

   When accessing things like textures stored on disk, you need to run a local web server. Otherwise the files cannot be loaded. This is due to browsers' *same origin policy security restrictions.* You can do this for example using python by opening a command prompt, changing to the folder containing your source files, and then running `python -m http.server`. The website can then be accessed via http://localhost:8000. Alternatively, you can also put the files in your student web folder and then access it via http (not the local file).

   See also mozilla.org: Set up a local testing server and three.js: How to run things locally.

3. Replace the rotating box with a proper sphere (`THREE.SphereGeometry`). Experiment with the number of subdivisions and try to find an appropriate setting. Note that a high number of subdivision can severely impact the triangle count (check out the wireframe setting).

4. Add another transformation to the existing scene graph (see `createSceneGraph()`) to consider the fact that Earth's axis is tilted by 23.44 degrees.

   In three.js, the scene graph nodes are represented by `THREE.Group` objects. These Group objects have a transformation matrix which can be modified.

   ```
   var group = new THREE.Group();

   group.scale.set(0.3, 0.3, 0.3);

   // three ways to set the position
   group.position.set(2.5, 0.0, 0.0);
   group.position.y = 10.0;
   group.position = new THREE.Vector3(1.0, 2.0, 3.0);

   // rotation around z axis in radians
   group.rotation.z = 90.0 * Math.PI / 180.0;

   group.add(myMesh);  // add other groups or meshes as children
   ```

   However, inside a Group the order of scaling, rotation, and translation is fixed. So it might be necessary to split up transformations over multiple groups. Use `add()` to append child nodes, which can be other Group nodes or mesh objects.

   Continue and add the Moon (`tex/2k_moon.jpg`) and let it rotate around Earth. Optionally, if you want, you can also incorporate the tilt of the Moon's orbit by 5.15 degrees.

5. Extend the scene graph further by adding the Sun (use `tex/2k_sun.jpg` for texturing) and at least another planet.

   Draw the scene graph on paper.

   Ensure that the Earth rotates around the Sun in 365 seconds and the Sun does a full turn every 25 seconds. The Earth should spin once every second while the Moon takes 27.3 seconds to orbit around the Earth (with the same side facing Earth at all times).

   Assuming that rendering the scene is fast enough, `requestAnimationFrame()` will cap the frame rate at 60 fps (or more if your display has a higher refresh rate). For now, assume that we render the scene at 60 fps. This means that a rotation of 360 degrees in one second requires a rotation of 360/60 degrees per frame. **Note:** keep in mind that three.js uses radians instead of degrees for angles.

6. Now it is time to apply some light. Luckily three.js provides a lot of functionality in that respect. Add a `PointLight`, located inside the Sun, to the scene. The `MeshBasicMaterial` used so far, however, is not affected by any light sources. Thus you need to change it to a `MeshLambertMaterial`. This will give the objects in the scene a soft appearance with only diffuse lighting applied. You can leave the Sun as it is to maintain the glowing look.

   You might want to add an `AmbientLight` as well to brighten up the dark spots. It does not need to be too bright; set its light color to `0x202020`.

7. It is also possible to use your own custom shaders in three.js. The **HTML file** already contains a vertex shader and a fragment shader for diffuse illumination. The specular part of the Phong model is missing.

   In three.js, shaders are part of a material (`ShaderMaterial`). Uncomment the custom `shaderMaterial` in the JavaScript file and apply it only to the Earth mesh. There should be no difference between the shader material and the original `materialEarth`.

   Continue by adding the specular reflection term of the Phong illumination model to the shader. Hint: you can use the `reflect()` function in GLSL to compute the reflection of a vector.

   three.js provides a number of uniforms for the shaders and set the correct values, like the `cameraPosition` as a `vec3`. For more built-in uniforms and attributes see threejs.org/docs/#api/en/renderers/webgl/WebGLProgram.

   See also gpumentalmodel.pdf located in the lab folder in Lisam for a brief introduction to GLSL shaders.

8. As you might know, the Earth is quite unique due to its large bodies of water. We now want to limit the specular reflection only to water surfaces since land masses are better represented with diffuse lighting only. This can be achieved by utilizing a *specular map*, a gray-scale texture, to scale the specular term of the illumination model (white corresponding to high specular reflection, black to none).

   Start by adding another texture (`tex/2k_earth_specular_map.jpg`) to the uniforms of the shaderMaterial and add the corresponding uniform sampler2D in the fragment shader. Then scale the specular term by using this texture. Specular highlights should now only be visible on the water.

9. (Optional) The scene navigation with the mouse is quite basic. Feel free to improve it by adding scene transformation which are modified when moving the mouse. This can include a simple translation and basic rotation around the center of the solar system.

   Hint: in order to map the xy movement of the mouse (given in screen coordinates) back to xyz world coordinates, you need the inverse matrix of the camera transformations (that is the inverse of the projection matrix).

4 Additional Links

- Getting started in three.js `https://threejs.org/docs/#manual/en/introduction/Creating-a-scene`
- Built-in uniforms and attributes of shaders `https://threejs.org/docs/#api/en/renderers/webgl/WebGLProgram`