CPSC 314
Assignment 3: Shaders

Due 11:59PM, March 10, 2024

1 Introduction

In this Assignment, you will utilize your knowledge of lighting and shading on 3D models. Here we will study how to implement various shading algorithms: Blinn-Phong, PBR (Physically Based Rendering) as well as some other fun shaders. This is a rather interesting assignment, so we hope you will have fun with it!

1.1 Getting the Code

Assignment code is hosted on the UBC Students GitHub. To retrieve it onto your local machine, navigate to the folder on your machine where you intend to keep your assignment code, and run the following command from the terminal or command line:

```
git clone https://github.students.cs.ubc.ca/CPSC314-2024W-T2/a3-release.git
```

1.2 Template

- The file `A3.html` is the launcher of the assignment. Open it in your preferred browser to run the assigment, to get started.
- The file `A3.js` contains the JavaScript code used to set up the scene and the rendering environment. You may need to modify it.
- The folder `glsl/` contains vertex and fragment shaders for the Snowman and other geometry. You may need to modify shader files in there.
- The folder `js/` contains the required JavaScript libraries. Generally, you do not need to change anything here unless explicity asked.
- The folder `gltf/` contains geometric model(s) we will use for the assignment, as well as texture images to be applied on the model(s).

2 Work to be done (100 pts)

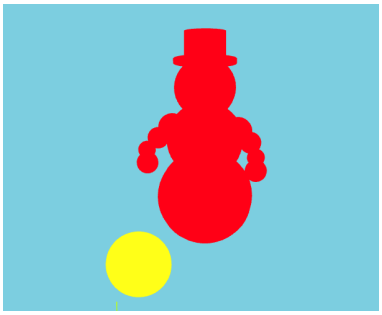

Figure 1: Initial configuration

Here we assume you already have a working development environment which allows you to run your code from a local server. If you do not, check out instructions from Assignment 1 for details. The initial scene should look as in Figure 1. Once you have set up the environment, Study the template to get a sense of what and how values are passed to each shader file. There are four scenes, and you may toggle between them using the number keys 1, 2, and 3: 1 - Blinn-Phong, 2 - Ray Marching, 3 - Three.JS PBR.
The default scene is set to 1. See `let mode = shaders.PHONG.key;` in A3.js. You may find it convenient during your development to change this default value to the scene containing the shader that you are currently working on (e.g. let mode = shaders.TOON.key; for question 1b).

2.1 Part 1: Required Features

a. **25 pts** Scene 1: Phong Reflection

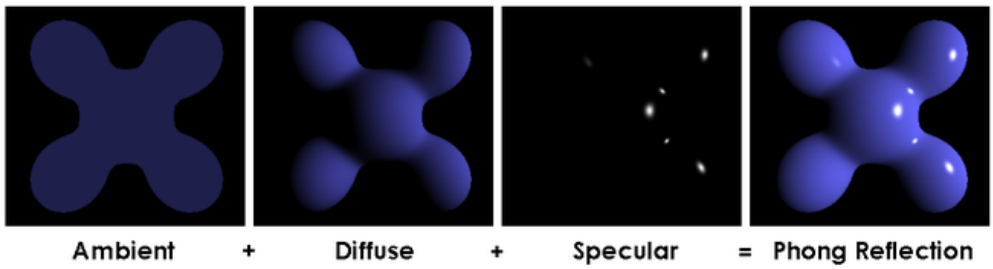

Figure 2: Phong reflection model.

First of all, note that the Phong reflection model is a different type of thing than the Phong shading model; they just happen to be named after the same person. The latter improves on the Gouraud shading by computing the lighting per fragment, rather than per vertex. This is done by using the interpolated values of the fragment's position and normal. In this scene, you will implement Blinn-Phong version of the Phong reflection

model. Figure 2, taken from the Wikipedia article on this model, shows how different components look either individually or when summed together:

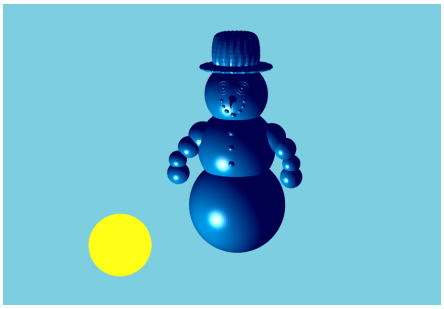

Figure 3: Question a: Blinn-Phong Snowman.

Your task here is to complete the code in `phong.vs.glsl` and `phong.fs.glsl` to shade the Snowman in Scene 1 using the Blinn-Phong reflection algorithm. While the majority of computation should happen in the fragment shader, you still need to add some code to the vertex shader in order to pass appropriate information to your fragment shader. Your resulting Snowman should look like the one shown in Figure 3.

b. **35 pts** Scene 2: Ray Marching

In this section, we explore *Ray Marching*, a rendering technique commonly used in computer graphics to determine ray-scene intersections, particularly for implicit surfaces like Signed Distance Fields (SDFs). Since fragment shaders execute per pixel, they efficiently compute ray-object intersections by iteratively stepping along a ray until it reaches a surface.

In essence, the Three.js scene here consists of a simple plane. The scene itself is not made of traditional 3D objects but is instead defined procedurally using mathematical functions within the fragment shader. These functions determine the fragment color, effectively "texturing" the plane with a ray-marched representation of the scene.

Figure 4: Question b: Ray Marched Snowman.

For this assignment, your task is to model a new Snowman (see fig. 4) by blending primitive shapes with boolean operations; a technique known as constructive solid geometry (CSG) modelling.

To begin, in the *raymarching.fs.glsl* fragment shader, implement the *rayMarch()* function to perform ray marching. Ray marching is a technique for determining the intersection of a ray with a scene by iteratively stepping along the ray's direction.

At each step, the algorithm computes the signed distance function (SDF) at the current point, which represents the minimum distance to the nearest surface. This distance dictates the step size for the next iteration—ensuring that the ray progresses efficiently while avoiding unnecessary computations. As the ray approaches a surface, the step size decreases, eventually reaching a threshold where the intersection is considered valid.

The loop terminates when the distance to the scene falls below the HIT_DIST threshold, indicating a successful intersection, or when either the maximum number of iterations (MAX_STEPS) or the maximum travel distance (MAX_DIST) is reached, preventing excessive computation.

Hint 1: Use *getSceneDist()* to get the nearest distance in the scene.
Hint 2: A helpful video guide `https://www.youtube.com/watch?v=PGtv-dBi2wE`

Once you have successfully implemented, the raymarch loop, you should be able to render a basic scene consisting of a snowy plane and a blue sky (see fig. 5).

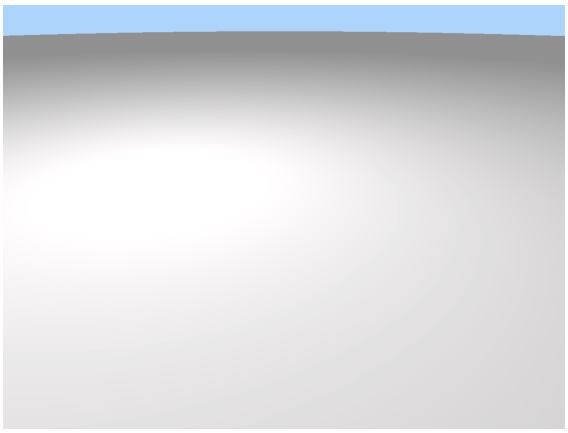

Figure 5: Question b: Blank Ray Marched Scene.

Now model the snowman! In *snowman()*, you will model the snowman by defining shape SDFs and blending them using the stubbed helper functions. In particular, you will need to create spheres (*Sphere()*), cylinders (*Cylinder()*), and a cone (*Cone()*), then blend them using boolean operations like smooth (*smoothUnionSDF()*) or hard (*(unionSDF())* unions. Although we do not expect your result to exactly match ours, you will be expected to include: three snowballs for the body, three buttons, two eyes, a nose, and a hat!

Hint 1: Inigo Quilez, is well known for creating incredibly immersive and detailed scenes using implicit functions, his website (`https://iquilezles.org/articles/`) has a num-

ber of useful articles relevant to this section.
Hint 2: If your machine is struggling to render the scene, look for comments labelled `NOTE`, to reduce visual complexity.
Hint 3: Note the camera (0,0,0) *looks at* (0,1,5), which is where we centered our snowman. Feel free to modify this position according to your snowman, or make sure your snowman roughly corresponds to this position.

c. **25 pts** Scene 3: Three.JS PBR and texturing

In this part, we'll get our first look at how to texture an object using Three.JS, and to create a physically based rendering of a damaged, sci-fi themed helmet.

Many objects have many small details and facets of the surface (e.g., wood grain, scratches on metal, freckles on skin). These are very difficult, if not impossible, to model as a single material lit by a Phong-like model. In order to efficiently simulate these materials we usually use texture mapping. In basic texture mapping, UV coordinates are stored as vertex attributes in the vertex buffer (Three.js provides them in the `vec2 uv` attribute for default geometries such as planes and spheres). UV coordinates allow you to look up sampled data, such as colors or normals, stored in a texture image, as discussed in class. Figure 6 shows the textures we will use for rendering the damaged helmet.

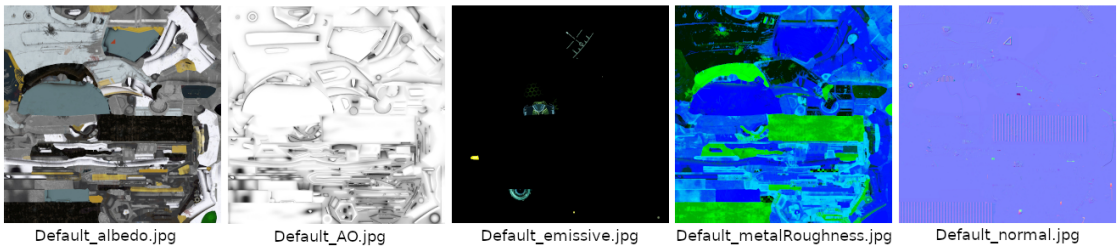

Figure 6: Textures used for the rendering the damaged helmet.

PBR or Physically Based Rendering is a technique that simulates the interaction of light with materials in a realistic manner, considering physical properties such as the reflectance (or albedo), roughness, and metalness to achieve lifelike visual representations. This information is stored in a set of texture images. For more information, you can take a look at this great writeup: https://learnopengl.com/PBR/Theory.

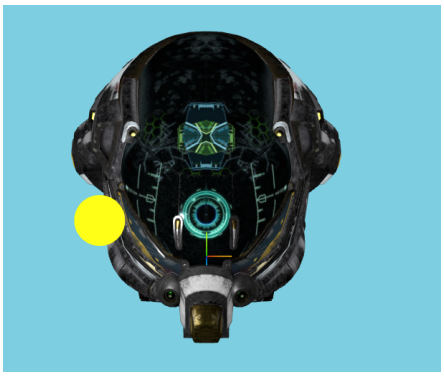

Figure 7: Question c: PBR with `MeshStandardMaterial`.

Your task is applying material textures (`.jpg` files) under `gltf/` to `helmetPBRMaterial`, which is already defined as Three.JS's built-in type `MeshStandardMaterial`. For this, modify `A3.js`. The result should look like Figure 7.

Textures in GLSL are specified as `sampler2D` uniforms, and the values can be looked up using the `texture()` function; Three.js' built-in materials can do this for you. And you need to load the relevant textures using `THREE.TextureLoader`. You can consult the Three.JS documentation for more information.

d. **15 pts** Feature Extension.

In this part, you are required to extend the assignment to add a feature of your own choosing. The goal is to encourage you to explore the capabilities of Three.js and WebGL. For full credit for this part, the feature does not have to be complex or creative, but must be non-trivial (e.g., not just changing a color). Roughly requiring about 10 lines of new code.

For example, extend or modify the raymarching scene to your imagination! You might consider creating other primitives, using different boolean operations, changing colours or making use of the *time* uniform to animate objects. As mentioned before, consider taking a look at the work of Inigo Quilez for endless inspiration (`www.shadertoy.com/user/iq`).

Alternatively, an easy way to prototype stunning visual effects with PBR: probe into the helmet's material properties, change its property values "on the fly" in your browser's console (F12). This allows you to interactively play around with all the properties, i.e. metalness, roughness, color, before identifying which combination of values looks the best to you and coding it up. For example, as shown in Figure 8, we can either change the `g` and `b` component to 0 so that the helmet becomes red-ish, or set `emissiveIntensity` to be 0 so that the emissive part of the helmet disappears.

For this question, first duplicate your current work into a new directory named 'part2', then implement your feature in this new directory. Write a brief description of your feature in the README file. You will be graded on both how it works and how well you can explain your feature.

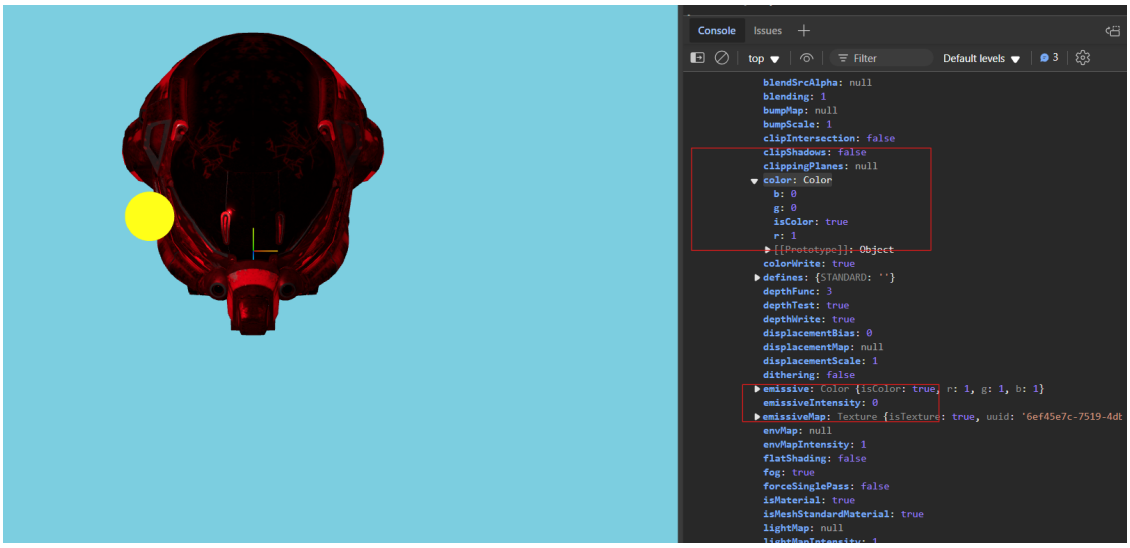

Figure 8: Modify material properties in the browser's console to prototype stunning visual effects.

3 Bonus Points

You have many opportunities to unleash your creativity in computer graphics! In particular, if you create a particularly novel and unique feature extension, you may be awarded bonus points. A small number of exceptional extensions may be shown in class, with the student's permission.

4 Submission Instructions

4.1 Directory Structure

Your submission should contain two subdirectories - the first should be named 'part1' and should contain all parts before the feature extension; the second subdirectory should be named 'part2' and should contain your feature extension. For each of the two subdirectories, include all the source files and everything else (e.g. assets) needed so each part can be run independently. Do not create more sub-directories than the ones already provided.
You must also write a clear `README.txt` file which includes your name, student number, and CWL username, instructions on how to use the program (keyboard actions, etc.) and any information you would like to pass on to the marker. Place the file under the root directory of your assigment.

4.2 Submission Methods

Please compress everything under the root directory of your assignment into `a3.zip` and submit it on Canvas. You can make multiple submissions, but we will grade only the last

one.

5 Grading

5.1 Face-to-face (F2F) Grading

Submitting the assignment is not the end of the game; to get a grade for the assignment, you must meet face-to-face with a TA in an 8-min slot during or outside lab hours, on Zoom, to demonstrate that you understand how your program works. To schedule that meeting, we will provide you with an online sign-up sheet. Grading slots and instructions on how to sign up will be announced on Canvas and on Piazza. During the meeting, the TA will (1) ask you to run your code and inspect the correctness of the program; (2) ask you to explain parts of your code; (3) ask you some questions about the assignment, and you will need to answer them in a limited timeframe. The questions will mostly be based on ThreeJS or WebGL concepts that you must have come across while working on the assignment; but also you may get conceptual questions based on lecture materials that are relevant to the assignment, or technicalities that you may not have thought about unless you really "digged deep" into the assignment. But no need to be nerveous! We evaluate your response based mainly if not entirely on the coherence of your thoughts, rather than how complete or long your response is: e.g. if the full answer to a question includes A+B+C+D and you mentioned A+B only in the provided time, but your thread of thought is logical then you may still get full marks.

5.2 Point Allocation

All questions before Feature Extension has a total of 85 points. The points are warranted based on

- The functional correctness of your program, i.e. how visually close your results are to expected results;
- The algorithmic correctness of your program, e.g. applying transformation matrices in the right order;
- Your answers to TAs' questions during face-to-face (F2F) grading.

The Feature Extension component is valued at 15 points. You will earn some points as long as you implement at least one extension that the grading TA considers novel and unique. However, full marks may not be awarded if your implementation introduces significant visual artifacts that diminish the product's appeal, or includes critical bugs that render the system unusable (e.g., failing to respond to user commands). Exceptional feature extensions may also be eligible for bonus marks, allowing for a score exceeding 100 on an assignment. However, note that your overall course grade will be capped at 100.

5.3 Penalties

Aside from penalties from incorrect solution or plagiarism, we may apply the following penalties to each assignment:
Late penalty. You are entitled up to three grace (calendar) days in total throughout the term. No penalties would be applied for using them. However once you have used up the grace days, a deduction of 10 points would be applied to each extra late day. Note that

1. The three grace days are given for all assignments, **not per assignment**, so please use them wisely;

2. We check the time of your last submission to determine if you are late or not.

No-show penalty. Please sign up for a grading slot on the provided sign-up spreadsheet (link will be posted later on Piazza) before the submission deadline of the assignment, and show up to your slot on time. A 10-point deduction would be applied to each of the following circumstances:

1. Not signing up a grading slot before the sign-up period closes. Unless otherwise stated, the period closes at the same time as the submission deadline.

2. Not showing up at your grading slot.

If none of the provided slots work for you, or if you have already missed your slot, follow instructions outlined in this Piazza post to get graded after the F2F grading period ends. Also, please note that

1. you'll need to explain to your TA why you're getting graded late, and may be asked to present documents to justify your hardship. The TA may remove the no-show penalty as long as he/she deems the justification to be reasonable.

2. In the past some students reported that their names disappeared mysteriously due to technical glitches, and the spreadsheet's edit history has no trace of it. So double check that your name is on the sign-up sheet after you sign up by refreshing the page.