CPSC 314
Assignment 3: Shaders

Due 11:59PM, March 15, 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, Toon, 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/CPSC314-2023W-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 Armadillo 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)

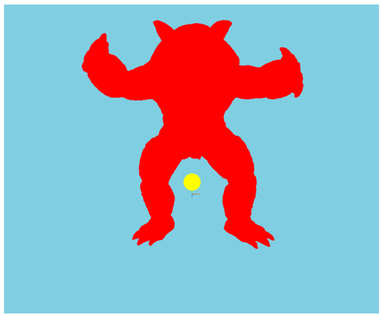


Figure 1: Initial configuration

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, 3, and 4: 1 - Blinn-Phong, 2 - Toon, 3 - Polka dots, 4 - 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

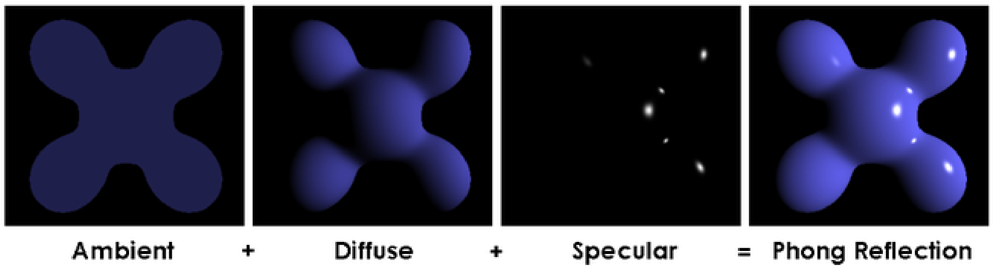


Figure 2: Phong reflection model.

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:

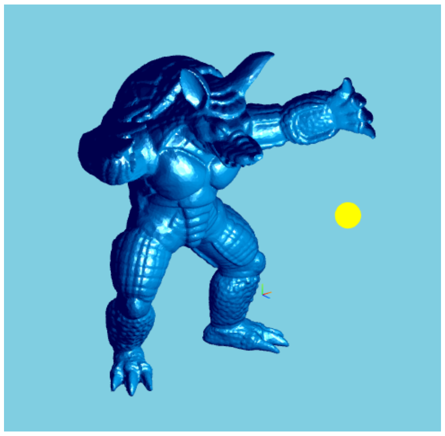


Figure 3: Question a: Blinn-Phong Armadillo.

Your task here is to complete the code in `phong.vs.glsl` and `phong.fs.glsl` to shade the Armadillo 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 Armadillo should look like the one shown in Figure 3.

b. **25 pts** Scene 2: Toon

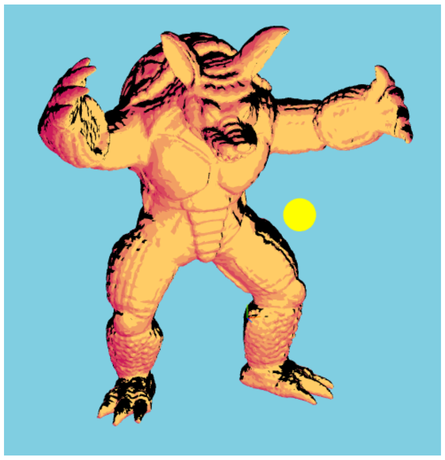


Figure 4: Question b: Toon Armadillo.

Send the Armadillo to the realm of action and superheros! Unlike the smooth, realistic shading in the previous questions, Toon shading gives a non-photorealistic result. It emulates the way cartoons use very few colors for shading, and the color changes abruptly, while still providing a sense of 3D for the model. This can be implemented by quantizing light intensity across the surface of the object. Instead of making the intensity vary smoothly, you should quantize this variation into a number of steps for each "layer" of toon shading. Use two tones to colour the Armadillo; red for the darker areas (lower light intensity) and yellow for the lighter areas (higher light intensity), as shown in Figure 4. Complete the code in the shaders.

Hint 1: This is most easily done by interpolating between two predefined colours. Lastly, draw dark silhouette outlines on the Armadillo. For each fragment, you can determine whether it is a part of the object's outline by computing the cosine of the angle betwen the fragment's normal and the viewing direction: fragments that are "edgy" enough should be outlines. If you need some inspiration, the following movies and video games were rendered with toon shading (also called cell shading) techniques:

`http://en.wikipedia.org/wiki/List_of_cel-shaded_video_games`

Hint 2: Since the silhouette is determined using the surface normal and the viewing direction, and does not depend the light direction, moving the light source should not affect its location.

c. **30 pts** Scene 3: Polka Dots

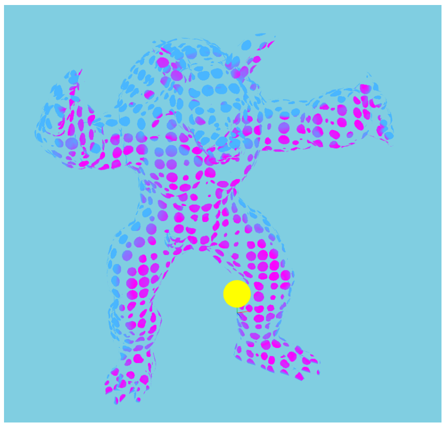


Figure 5: Question c: Polka-dotted Armadillo.

The possibilities are endless! Here, we ask you to (i) give the Armadillo polka dots by checking whether a fragment is close enough to points on a regular grid in the 3D space around the Armadillo; (ii) make the polka dots smoothly "roll" down the body over time; and (iii) make the dots smoothly change colour. To implement these effects, you will need to modify `dots.fs.glsl`, so it shades the fragment depending on a time "ticks" and the local vertex position, and discards fragments that you don't want to render: see the empty space between the dots. Complete the code in the shaders. The final result should look like what is shown in Figure 5. To check out the combined effects of (i), (ii) and (iii) when animated, see this video clip. Or look at the combined effects of (i) and (ii) shown in this video clip. If you are unable to implement rolling dots, you can still get partial marks for completing (iii) as shown in this video clip.

Hint 1: Make use of some oscillatory mathematical function(s) to implement the color change.

Hint 2: The `discard` statement in GLSL throws away the current fragment, so it is not rendered.

Hint 3: You may find the GLSL `mod()` function to be useful. It computes the modulus of two floats. Example: `mod(0.3, 0.2) = 0.1`.

Hint 4: The Armadillo's model coordinates may be smaller than you think. If your Armadillo disappears completely (not because of some syntax error) try using smaller numbers when computing regions to discard.

d. **20 pts** Scene 4: 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.

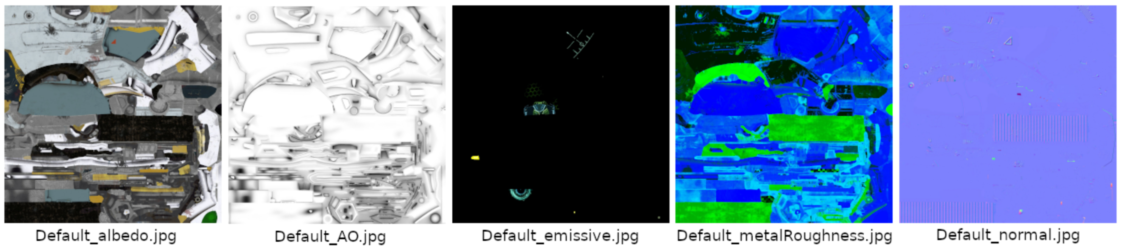


Figure 6: Textures used for the rendering the damaged helmet.

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.

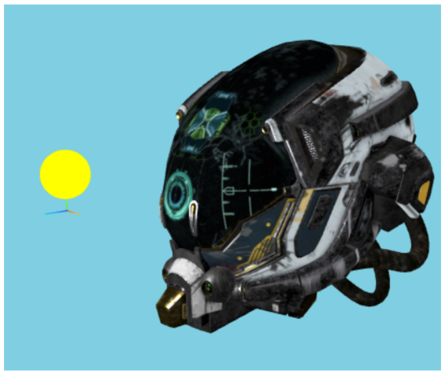
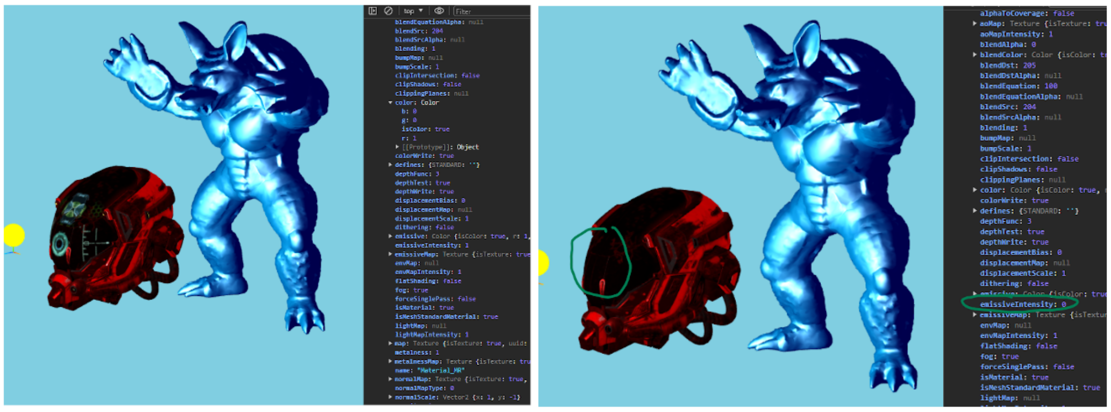


Figure 7: Question d: PBR with `MeshStandardMaterial`.

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.

2.2 Part 2: Creative License (Optional)



```
blendEquationAlpha: null
blendSrc: 204
blendSrcAlpha: null
blending: 1
bumpMap: null
bumpScale: 1
clipIntersection: false
clipShadows: false
clippingPlanes: null
▾ color: Color
    b: 0
    g: 0
    isColor: true
    r: 1
  ▸ [[Prototype]]: Object
colorWrite: true
▸ defines: {STANDARD: ''}
depthFunc: 3
depthTest: true
depthWrite: true
displacementBias: 0
displacementMap: null
displacementScale: 1
dithering: false
▸ emissive: Color {isColor: true, r: 1,
emissiveIntensity: 1
▸ emissiveMap: Texture {isTexture: true
envMap: null
envMapIntensity: 1
flatShading: false
fog: true
forceSinglePass: false
isMaterial: true
isMeshStandardMaterial: true
lightMap: null
lightMapIntensity: 1
▸ map: Texture {isTexture: true, uuid:
metalness: 1
▸ metalnessMap: Texture {isTexture: tru
name: "Material_MR"
▸ normalMap: Texture {isTexture: true,
normalMapType: 0
▸ normalScale: Vector2 {x: 1, y: -1}
```

Toggling g and b component.

```
alphaToCoverage: false
▸ aoMap: Texture {isTexture: tru
aoMapIntensity: 1
blendAlpha: 0
▸ blendColor: Color {isColor: tr
blendDst: 205
blendDstAlpha: null
blendEquation: 100
blendEquationAlpha: null
blendSrc: 204
blendSrcAlpha: null
blending: 1
bumpMap: null
bumpScale: 1
clipIntersection: false
clipShadows: false
clippingPlanes: null
▸ color: Color {isColor: true,
colorWrite: true
▸ defines: {STANDARD: ''}
depthFunc: 3
depthTest: true
depthWrite: true
displacementBias: 0
displacementMap: null
displacementScale: 1
dithering: false
▸ emissive: Color {isColor: true
emissiveIntensity: 0
▸ emissiveMap: Texture {isTextur
envMap: null
envMapIntensity: 1
flatShading: false
fog: true
forceSinglePass: false
isMaterial: true
isMeshStandardMaterial: true
lightMap: null
```

Nullifying emissiveIntensity.

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

You have many opportunities to unleash your creativity in computer graphics! In this **optional** section, and you are invited to extend the assignment in fun and creative ways. A small number of exceptional contributions may be awarded bonus points; we will also highlight some of the best work in class.

Here's 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.

3 Submission Instructions

3.1 Directory Structure

Under the root directory of your assignment, create two subdirectories named "part1" and "part2", put all the source files and everything else required to run each part in the respective folder. 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.

3.2 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.

4 Grading

4.1 Point Allocation

Part 1 has 100 points in total; the points are warranted holistically, 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;
- Your answers to TAs' questions during F2F grading.

Part 2 is optional and you can get bonus points (0-10 points) at the instruction team's discretion. The max score for each assignment is 110 points.

4.2 Face-to-face (F2F) Grading

For each assignment, you are required to meet face-to-face with a TA during or outside lab hours to demonstrate that you understand how your program works. To schedule that meeting, we will provide you with an online sign-up sheet. Details regarding when and how to sign up will be announced on Canvas and on Piazza.

4.3 Penalties

Aside from penalties from incorrect solution or plagiarism, we may apply the following penalties to each assignment:

a. **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

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

   (b) We check the time of your last submission to determine if you are late or not;

   (c) We do not consider Part 1 and Part 2 submissions separately. Say if you submitted Part 1 on time but updated your submission for Part 2 one day after the deadline, that counts one late day.

b. **No-show penalty.** Please sign up for a grading slot at least one day before F2F grading starts, and show up to your slot on time. So a 10-point deduction would be applied to each of the following circumstances:

   (a) Not signing up a grading slot before the sign-up period closes;

   (b) Not showing up at your grading slot.

   If none of the provided slots work for you, please contact the course staff on Piazza before the sign-up closes. Also, please note that

   (a) we would not apply the no-show penalty if you are unable to sign up/show up on time because of a personal health emergency, and in such cases we would like to see a written proof of the situation.

   (b) if you decide to make a late submission and only get graded after the F2F grading period ends, you need to show the TA your submission time on Canvas so he/she knows you submitted late and will not apply the no-show penalty.

   (c) In the past some students reported that they got their names overwritten by others, or their names disappeared mysteriously due to technical glitches. Therefore we suggest you to take a screenshot of your slot after sign-up just to prove you have done it.