UTD CS 4361 Assignment 4: Shaders

Due on 11:59pm on Nov 4, 2025

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 released on the E-learning page. Please download it to your local machine, navigate to the folder where you intend to keep your assignment code, and run the following command from the terminal or command line:

```
unzip a4-release.zip
```

1.2 Template

- The file `A4.html` is the launcher of the assignment. Open it in your preferred browser to run the assignment, to get started.
- The file `A4.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 dragon 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 explicitly 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).

1.3 Anti-AI/Plagiarism Measures

To prevent plagiarism, 10 students will be randomly selected. These students are required to present their final submitted code to TAs during office hours and point out the locations of core code for the corresponding problems.

2 Work to be done (100 points)

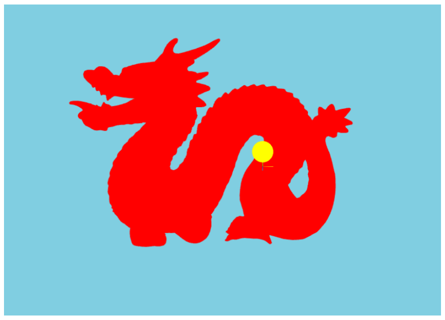
Figure 1: Initial configuration

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.BLINNPHONG.key;` in A4.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 points)** Scene 1: Phong Reflection

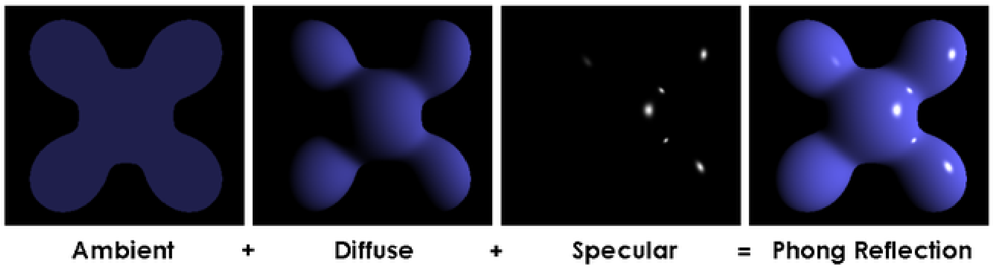

Figure 2: Phong reflection model.

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:

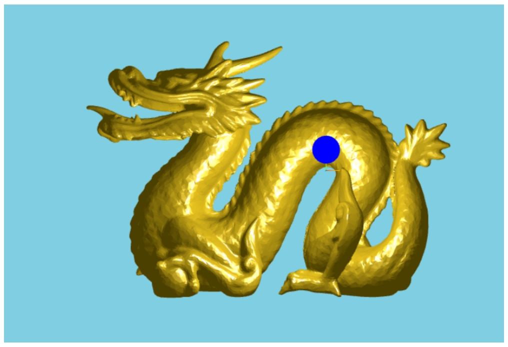

Figure 3: Question a: Blinn-Phong Dragon.

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

b. **(25 points)** Scene 2: Toon

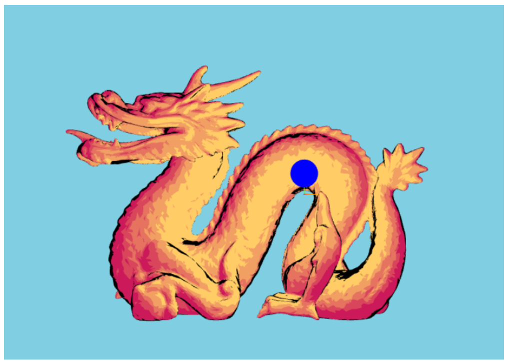

Figure 4: Question b: Toon Dragon.

Send the Dragon 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 Dragon; 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 Dragon. For each fragment, you can determine whether it is a part of the object's outline by computing the cosine of the angle between 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 points)** Scene 3: Polka Dots

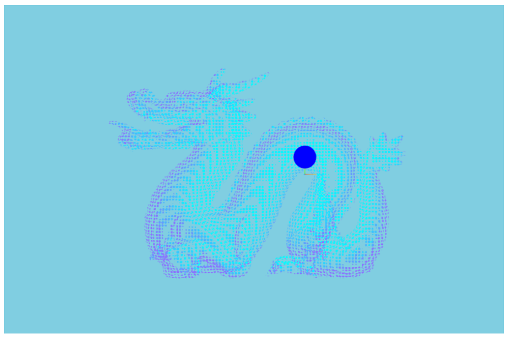

Figure 5: Question c: Polka-dotted Dragon.

The possibilities are endless! Here, we ask you to (i) give the Dragon polka dots by checking whether a fragment is close enough to points on a regular grid in the 3D space around the Dragon; (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. If you are unable to implement rolling dots, you can still get partial marks for completing (iii).

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

d. **(20 points)** 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.

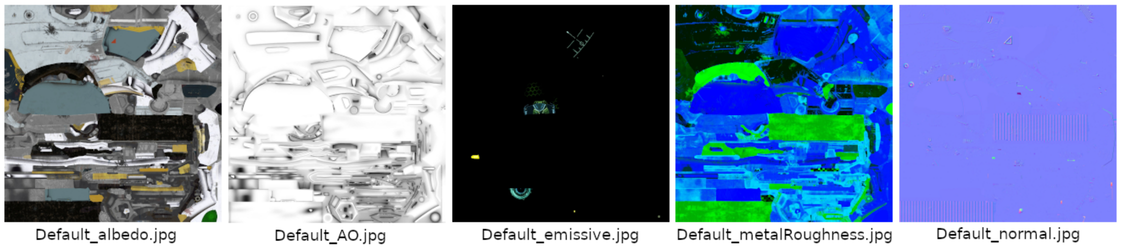

Figure 6: Textures used for the rendering the damaged helmet.

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`.

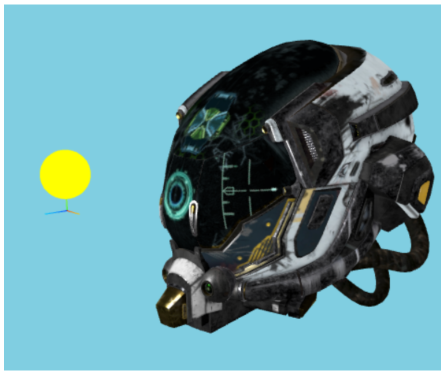

Figure 7: Question d: PBR with `MeshStandardMaterial`.

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 `A4.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.

3 Submission Instructions

3.1 Directory Structure

You must submit your final code and write a clear `README` file which includes your name, student number, and the core idea/changed code/explanation/screenshot of each question. The TAs strongly encourage you to read the template of answers in template.md and use a similar template to submit your answers.
Your README file can be in PDF, Word, or Markdown format. It must be placed in the root directory and will serve as the basis for grading your work. **Missing a README file will result in a 30-point deduction.**

3.2 Submission Methods

Please compress everything under the root directory of your assignment into `a4.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

Each assignment has 100 points for Part 1. 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. **Missing a README file will result in a 30-point deduction. Please make sure the README file includes the screenshot of each question.**
TAs will carefully review your README file and evaluate based on idea/code/explanation/screenshot. If the screenshots show correct rendering effects, full points will be graded.

4.2 Anti-AI/Plagiarism Measures

To prevent plagiarism/AI cheating, 10 students will be randomly selected. These students are required to present their final submitted code to TAs during office hours and point out the locations of core code for the corresponding problems. TAs will typically notify selected students via email with meeting links, offline addresses, and a schedule for students to book appointments. Each student will have 5 minutes to answer questions about their code.

4.3 Penalties

Aside from penalties from incorrect solution or plagiarism, we may apply the following penalties to each assignment:
Late penalty. A deduction of 10 points will be applied for each late day. Note that

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

AI/Plagiarism penalty. If a student cannot point out where their code is located for solving specific problems (e.g., which lines of GLSL code correspond to the answer for a particular problem) and cannot clearly articulate their approach to solving the problem, this will result in a complete deduction of points for the corresponding assignment.