Online-Hausarbeit 3: Implementing a Rendering Pipeline

Bearbeitungszeitraum: 20.07.2020 00:00 Uhr - 31.07.2020 23:59Uhr

General Informations

- There are three graded assignments allowing for a total score of 200 regular points and 20 bonus points. You will need 100 points to pass with 4.0, every additional 10 points increase the grade to the next step, while a score of 190 points or greater results in 1.0.
- It is prohibited to exchange solutions for the graded assignments with other students during the examination period. You must work on the graded assignments alone and independently and submit your own solution. If we discover any fraud or plagiarism in the submission, both parties will be excluded from the course.
- For programming tasks, you must organize and comment your code well by following additional comments/instructions in the code skeleton. Weird coding style or changing the provided template may not be well tested or graded. In addition, any comments in your code must be written in English.
- For non-programming tasks, you can answer in German or English or mix.
- If you have any questions regarding technical issues, please contact the course assistants immediately.

1 Overview

This is the last graded assignment for the CG1. You can achieve a maximum of 110 points from this graded assignment. The estimated completion time for this assignment is about 4 to 8 hours. Depending on your familiarity with the subject and possible coding issues, your time might increase accordingly. If you struggle with one task, get yourself some fresh air and come back to it later. To reduce the time pressure, you can submit your work until 31.07.2020 23:59.

For our own information (and without any influence on the grade!), please roughly record the actual time (in hours) you spent on this assignment and let us know on the last page below your signature.

Remember we told you that the rasterization rendering pipeline is *the* most important concept in this course? This assignment aims to check whether you really understood the pipeline or just scratched the surface of the topic. Therfore, we believe the best way is to ask for an implementation of all crucial components of the rasterization pipeline. As a result you should be able to render a bunny with a texture totally from scratch, *without using any graphics APIs.*

Before you take a look at the skeleton, let's do a quick review of the pipeline. First, the pipeline starts by initializing two buffers, then a mesh model is transformed into world space. Next, camera transformation to camera space, then projection transformations to projection space. The transformed coordinates are used for shading and occlusion testing. To shade the pixels, we interpolate the needed information, then query color values from the given texture. Lastly, we compute an illumination model for the final shading. Although this is a strongly simplified process, it should still present a reasonable, but manageable challenge.

2 Code Skeleton

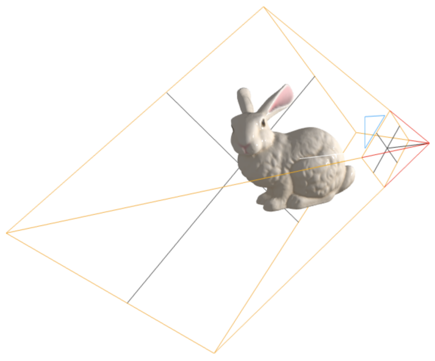
Since you are already familiar with the graded assignment workflow, you can directly start using **`npm i`** to install dependencies and **`npm start`** to run the code skeleton. The skeleton includes many different files: the `src/assets/` folder contains the bunny model and its texture. The `package.json`, `package-lock.json`, `webpack.config.js`, `babel.config.js` are files that you don't care.

To let you work dircetly with pixels, the skeleton supplies an 800x500 "hardware display", which takes all your calculated and stored color values, and flushes them from your frame buffer (in the `src/rasterizer.js`) into the display. You can read the `src/main.js` file for more understanding of how the "hardware display" is constructed, but it is not really necessary for your rasterizer implementation.

For this graded assignment, all of your tasks are specified within the corresponding `// TODO:` comments. You will work in `src/vec.js`, `src/mat.js`, and `src/rasterizer.js`, which computes the pixel colors and stores them in frame buffer using basic arithmetic: addition, subtraction, multiplication, and division.

3 Result Reference

Figure 1a shows the scene setup, including a textured bunny in a camera view frustum with Blinn-Phong shading effects. This scene rendered by `three.js`. Once you completed this graded assignment, you should get a result that looks almost exactly as Figure 1b. As always, you can find an online interactive demo to compare your implementation and the expected reference result at `https://www.medien.ifi.lmu.de/lehre/ss20/cg1/demo/grading/rasterizer/`. It might take some time to load and render the result, depending on your network and hardware speed.


(a) A simple scene that contains a textured bunny using Blinn-Phong shading.

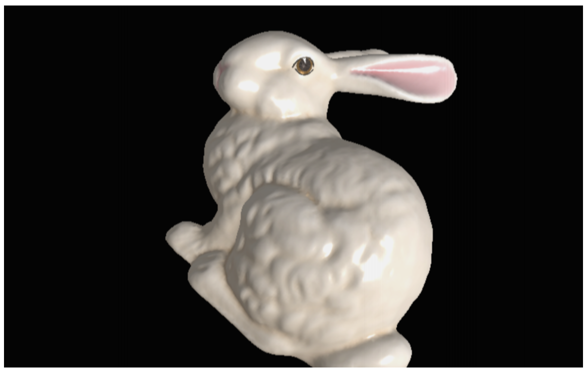

(b) The final rendering result based on the settings on the left.

Figure 1: Result reference: your final result should looks almost exactly the same.

4 Grading Details (100 Points)

This section lists the grading details. You can always skip (or postpone) some of the features (or tasks) just as you can skip some of the tasks in a traditional written exam if you have no idea how to implement them correctly. However, in some cases, a subsequent feature relies on a previous feature. A general hint is to figure out which feature you need to implement first, then make sure your implementation is correct before you pursue the next one. In this way, you can prevent most of the mistakes that could affect subsequent implementations.

As said, **you are not allowed to use any APIs**. Before you start working on the implementation of the pipeline, you need vector and matrix operations support, which serve as a warmup task in `src/vec.js` and `src/mat.js`:

- Implement the vector and matrix operations in the `Vector` and `Matrix` class (**total: 13p**)
  - Implement vector addition (1p), subtraction (1p), scalar multiplication (1p), dot (1p) and cross products (2p), normalization (1p), and 4x4 matrix by 4x1 vector multiplication (2p)
  - Implement matrix elements' set method (1p), and 4x4 by 4x4 matrix multiplication (3p)

Next, use the implemented `Vector` and `Matrix` classes to continue the implementation of your rasterizer in `src/rasterizer.js`:

1. Implement buffers initialization in the `initBuffers` method (**total: 4p**)
   - Init the frame buffer by *black* color (2p) and depth buffer by an appropriate value (2p)
2. Prepare transformation matrices in the `initTransformation` method (**total: 21p**)
   - Init model matrix using params in `this.model` and assign to `this.Tmodel` (6p)
   - Init view matrix using params in `this.camera` and assign to `this.Tcamera` (6p)
   - Init perspective matrix using params in `this.camera` and assign to `this.Tpersp` (6p)
   - Init viewport matrix using params in `this.screen` and assign to `this.Tviewport` (3p)
3. Prepare the needed informations in the `render()` method for drawing (**total: 10p**)
   - Invoke the needed initializations (1p). For all triangles, implement vertex generation process by accessing vertices coordinates (3p), UVs (2p), and normals (2p) from the loaded geometry `this.model.geometry`, then pass them as arguments to the `draw` method (2p)
4. Implement a rendering pipeline in the `draw` method with vertex and fragment shader support (**total: 30p**)
   - Process each vertex using vertex shader (1p). Implement a culling approach that helps generating fragments based on the processed vertices (5p). For all generated fragments, computes barycentric coordinates (6p) and skips those fragments that outside the processing triangle (3p).
   - Implement occlusion testing (4p), interpolate UVs (2p), fragment position (2p), and normals (4p) for each processed fragment. Then update the depth buffer (1p) and update the frame buffer by fragment shader (2p)

5. Implement the vertex shader and fragment shader (**total: 22p**)
   - In `this.vertexShader`, transform a vertex from model space to projection space (5p)
   - In `this.fragmentShader`, query texture color (5p), then compute the Blinn-Phong reflectance model in Phong shading frequency for the final shading (12p)

It is worth mentioning that performance is not the goal of this assignment. Correct and clean implementation is more preferred. However, if you complete quickly and feel bored over the lengthy weeks, we encourage you to think about the bottleneck and try to optimize your implementation for better performance. Performance optimization is just for fun, and no points are rewarded.

5 Submission Instructions (Bonus: 10 Points)

Please use the provided work package to create your submission, and **make sure that your submission is complete. In particular, the signed "Erklärung über die eigenständige Bearbeitung" must be included in your submission. A submission without signature will not be graded, and a re-submission is NOT possible.** Your signature must be a non-editable element in the document. Specifically, you can either print the document, sign then rescan the document as a PDF file that replaces this document with your hand signature; Or, you can also add your electronic signature in this document by using the Preview app on macOS or Adobe Acrobat Reader DC on Windows (Linux user can find their own tools). Following the additional submission instructions below will reward you with 10 bonus points.

- Submission works without compiling errors using `npm run build` (1p) and without runtime errors in the Chrome/Firefox browser's console (1p).
- Adding your implementation directly after the `// TODO:` comments without changing other places of the code skeleton (1p) and documenting your implementation by adding additional short, clean comments whenever you think they are helpful (1p). Note that comments serve the purpose of explaining non-trivial code, and you do not need to explain trivial code (e.g. `s += 1 // increase s by 1`). This also means that commented code, which is your unfinished work, will not be processed or read. We hope you learned this basic rule from your former coding class.
- Remove all unnecessary, unused, and unfinished constants, variables, and comments in your implementation. Exclude any unrelated files (e.g. `node_modules`). Your submission folder structure should be exactly the same as the showed folder structure in the below (1p).
- Report your implementation in the `CHECKLIST.md` by checking your implemented features among the listed features. Your checked features in the `CHECKLIST.md` should match your actual implementation (1p).
- Create an `images` folder, and attach intermediate results by taking screenshots of 1) a white bunny without mapped texture 2) a bunny that is colored by its depth value (i.e. grayscale depth map) 3) a bunny with mapped texture without using the Blinn-Phong model. Your screenshots must be in `.jpg` format (3p).
- Rename your folder name to `abgabe3-<your matriculation number>`. Then submit your solution as a `.zip` file via Uni2Work (1p).

As an example, assume your matriculation number is 12345678, then your ZIP filename should be `abgabe3-12345678.zip`, and the decompressed folder structure should look exactly like this:

```
abgabe3-12345678/
├── CHECKLIST.md
├── README.pdf           // with your signature
├── babel.config.js
├── images               // create and attach your intermediate results
│   ├── 1.jpg
│   ├── 2.jpg
│   └── 3.jpg
├── package-lock.json
├── package.json
├── src
│   ├── assets
│   │   ├── bunny.obj
│   │   └── texture.jpg
│   ├── main.js
│   ├── mat.js
│   ├── rasterizer.js
│   └── vec.js
└── webpack.config.js
```

Erklärung über die eigenständige Bearbeitung

Online-Hausarbeit 3

Ich erkläre hiermit, dass ich die vorliegende Arbeit vollständig selbstständig angefertigt habe. Quellen und Hilfsmittel über den Rahmen der Vorlesungen/Übungen hinaus sind als solche markiert und angegeben. Ich bin mir darüber im Klaren, dass Verstöße durch Plagiate oder Zusammenarbeit mit Dritten zum Ausschluss von der Veranstaltung führen.

Ort, Datum Unterschrift

Ich habe insgesamt etwa _____ Stunden Arbeitszeit für diese Abgabe aufgewendet. (Diese Information ist freiwillig, rein informativ und hat keinen Einfluss auf die Benotung.)