THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

CSIT6000L - Advanced Digital Design

Group 3 - Project Report

Minecraft-like Game Scene Renderer With Web UI

HAN, Chi Chiu	**20533742**
TANG, Quan	**20835815**
YUAN, Dian	**20785278**

Email: {cchanau, qtangac, dyuanaf}@connect.ust.hk

May 2022

Project Summary

Recent in-game graphic rendering qualities show there is certainly room for improvement of the current methods suggested in assignment 4, such as the sense of hierarchy and pixel quality. With the goal of better sense construction, this project aims to make the characters and objects look similar to Minecraft. In particular, the project strives to provide an easy access and user-friendly platform for sense creation by combining the existing features in assignment 4 with more advanced ones.

Objectives

In order to achieve the project goal, the following objectives are selected under in-deep consideration:

1. Implementing the depth of field blurring for simulating the real-world scenarios of human eyes and physical camera.

2. Attaching normal mapping and environment mapping to enrich the current sense rendering effects while maintaining a low computation cost.

3. Reproducing Minecraft like pixel-art themed by optimizing the texture with pixel-perfect

4. Delivering the real-time sense rendering and construction platform by porting the project to a web browser environment.

Design Process & Methodology

We started working on our project after the completion of assignment 5, during which we reorganized the source code to make it much more extensible. With the source files thoughtfully divided into multiple sub-directories and the redesigned building and testing procedures, we were ready to implement our objectives and integrate them into the existing project.

After some observation and planning, we decided to implement the DOF feature as an optional command argument flag, the normal and environment mapping features as extensions of the scene definition file format. This design aligns well with the existing features and thus keeps the old test cases functional. The feature of pixel-perfect texture, on the other hand, was discovered to be a rather trivial task, which can be achieved by defining a toggle flag and some minor adjustments of the Texture class.

As for the web browser transplantation, we first made a feasibility study about WebAssembly and related technologies. Then, a concept-proof example was built to port the command-line program into the browser console, which turned out to be a success. Therefore, we could work on multiple possible paths to improve the user experience of the Web UI, and we resulted in some outstanding outcomes.

Technology

We used C++ for the renderer project, and Emscripten toolchain for WebAssembly transplantation. For the Web UI, we used TypeScript and ReactJS as the framework, Ant Design, Monaco Editor, Chevrotain for implementing related functionalities.

Implementation 1: Depth of field Blurring

Depth of field (DOF) is used to describe a range of distance such that the objects inside the range are imaged with acceptable sharpness [1]. Contrastingly, the objects outside the focus distance are blurred to some extent when imaging. Given the aperture size of camera lens N, circle of confuse C, focal length f and focus distance D, the approximate DOF can be derived by the formula [1]:

$$d = \frac{2NCD^2}{f^2}$$

From the formula, one can realize that the smaller the aperture size of the lens, the less the DOF resulted in the image. This shows the pinhole camera in assignment 4 is not suitable for simulating DOF as it considers the lens as a single point such that the rays pass through the lens directly. Thus, all the objects in the sense share the same sharpness no matter the distances.

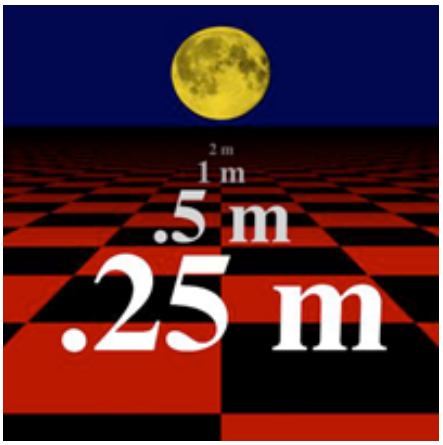

Image 1: Sense with same sharpness in pinhole camera [2]

To simulate the physical camera in the real world, we introduce the Thin-lens camera in our implementation. With the Thin-lens camera, the lens is a disk that allows the rays to pass through with a different degree of refraction. It is clear that the objects inside the focus distance have a higher possibility hit by rays and thus produce a sharper image. The below image shows the Thin-lens camera model, in which the focus distance is the distance between the lens and the point in focus.

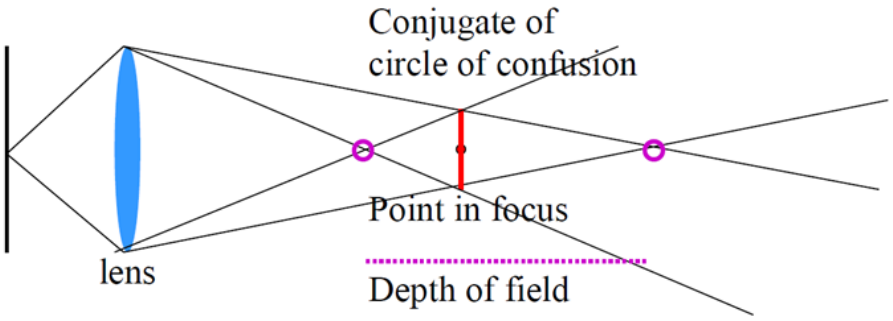

Image 2: DOF with Thin-lens camera [1]

For simplicity, our implementation ignores the focal length and circle of confusion by fixing the camera in the same position with the aperture size set to 0.5f. This causes the value of DOF fully dependent on the focus distance provided, so one can adjust the range of DOF by providing different focus distances. For example, both images 3 & 4 illustrate the blurry effects with various focus distances. The orange cube is blurred in image 3 because it is imaging beyond the image plane, while the purple cube is blurred in image 4 because it is imaging behind the image plane.

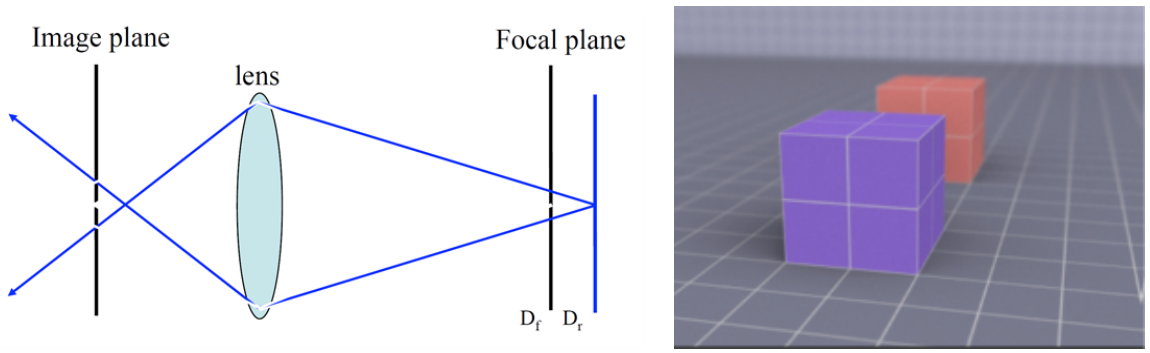

Image 3: Distant object burring [2]

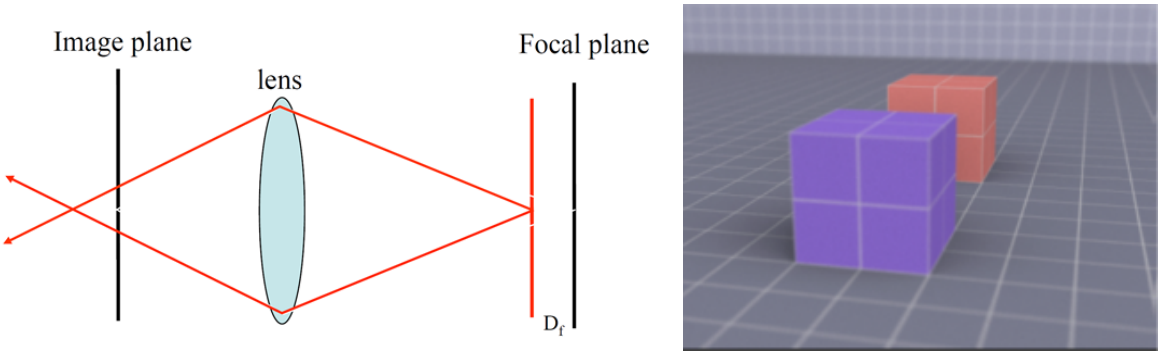

Image 4: Close object burring [2]

Pseudocode

1. Draw a disk (lens) center at a pixel of image panel
2. Pick a focus point S with a given focus distance
3. Sample N rays on the disk with direction toward S
4. Sum up the color returned by the rays and take the average.
5. Repeat step 1 - 4 for every pixel in image panel

Result

The following image displays the result of the DOF feature, the output starting from the left are one from the original assignment 4, our implementation with focus distance 6.5 and 3.5 respectively.

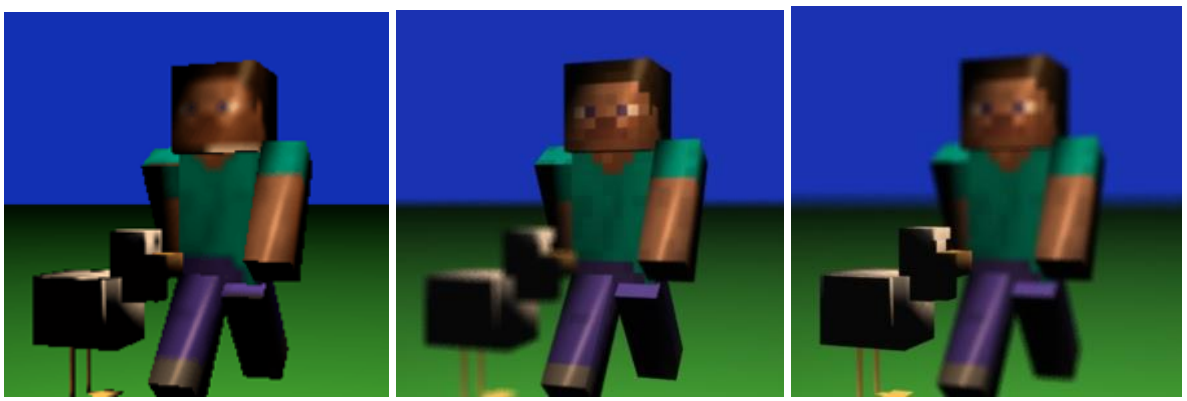

Image 5: DOF blurring results

Implementation 2: Normal Mapping & Environment Mapping

Normal Mapping

A normal map is a texture mapping technique used for faking bumps and dents. It can be used to add details without adding more polygons. It is commonly used to significantly enhance the appearance and details of a low polygon model by generating a normal map from a high polygon model. The basic idea is mapping the normal vectors to RGB vectors (map from range [-1, 1] to [0, 1]), and then reversing the process of mapping normals to RGB colors by remapping the sampled normal color from [0, 1] back to [-1, 1], and then use the sampled normal vectors for the upcoming calculations. Finally we get a sense of depth. The process of normal mapping is illustrated below (from left to right):

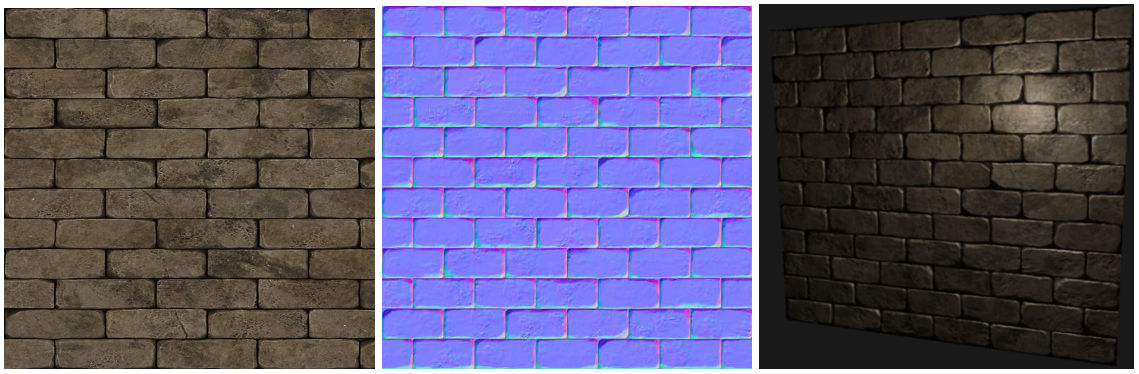

Image 6: Normal Mapping process [3]

However, there is a little caveat here, the normal vectors and the surface normal may point in different directions, which is not what we wanted. For example, in the picture below, the surface normal is pointing towards z direction whereas the normal vectors are pointing towards y direction, as a result, the lighting is not correct:

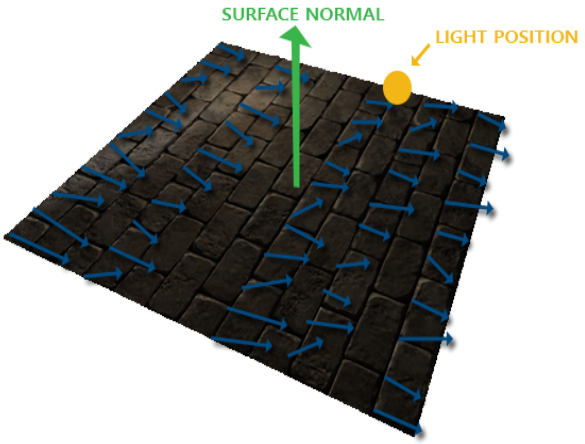

Image 7: Inconsistent pointing directions between surface normal and normal vectors [3]

To resolve this issue, we implemented a mathematical method to calculate the TBN (Tangent, Bitangent, Normal) matrix converting normal vectors in tangent space to real space. The implementation is as follows:

```
void Triangle::setTbn(Hit &h) {
    auto &[p0, p1, p2] = texCoords;
    auto e0 = b - a, e1 = c - a;
    auto invDuDv = Matrix2f(p1 - p0, p2 - p0, false).inverse();
    auto tbx = invDuDv * Vector2f(e0.x(), e1.x());
    auto tby = invDuDv * Vector2f(e0.y(), e1.y());
    auto tbz = invDuDv * Vector2f(e0.z(), e1.z());
    auto t = Vector3f(tbx.x(), tby.x(), tbz.x()).normalized();
    auto b = Vector3f(tbx.y(), tby.y(), tbz.y()).normalized();
    auto n = Vector3f::cross(t, b).normalized();
    h.setTbn(Matrix3f(t, b, n));
}
```

Image 8: C++ implementation of TBN matrix

The implementations in image 8 are based on the mathematical equation below:

$$\begin{bmatrix} \Delta U_1 & \Delta V_1 \\ \Delta U_2 & \Delta V_2 \end{bmatrix}^{-1} \begin{bmatrix} E_{1x} & E_{1y} & E_{1z} \\ E_{2x} & E_{2y} & E_{2z} \end{bmatrix} = \begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix}$$

Image 9: TBN Matrix Calculation

The images below are our final results:

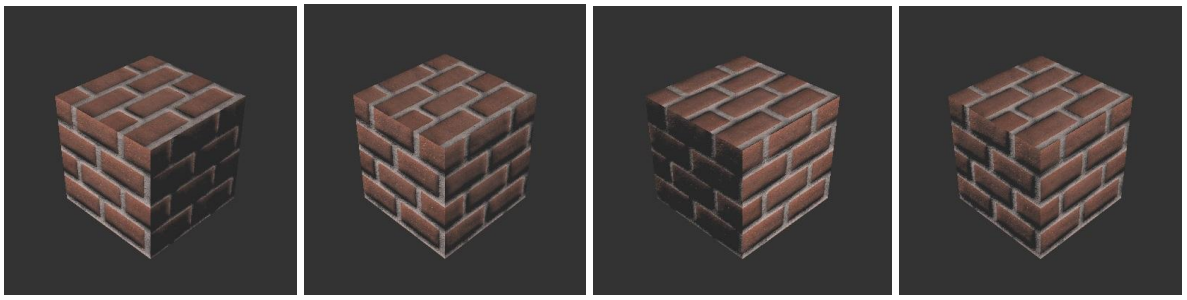

Image 10: Results for normal mapping implementation

Environment Mapping

Environment mapping is a fast, image based lighting method. The idea is placing the object inside a cube-map with an environment texture applied:

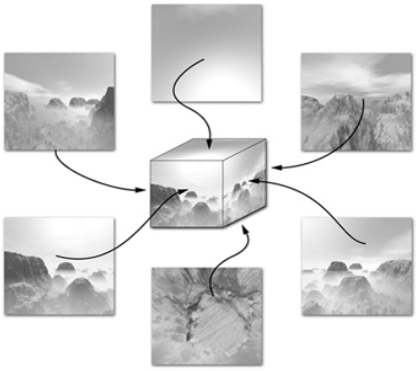

Image 11: Cube Map Textures[4]

The object would reflect the incident ray and the reflection ray would hit a pixel of the environment texture, whose color is more or less equal to the corresponding pixel's color on the object. To illustrate this, please see image 12 below:

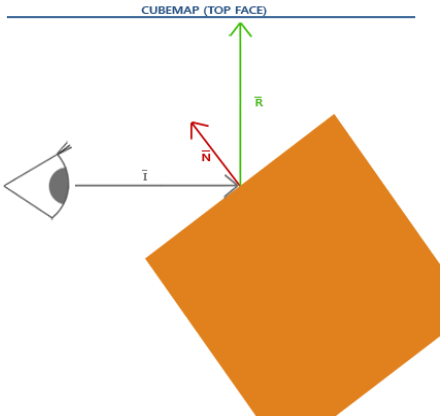

Image 12: Incident ray and reflection ray [5]

The orange object in image 12 is the object we would like to perform environment mapping on. The incident ray originates from the camera hitting the object and the corresponding reflection ray would hit the texture, intersecting with the texture on a pixel, the color of this pixel is the color that needs to be mapped to the pixel where the incident ray intersects with the orange object. As a result, we could simulate an object reflecting its surroundings in environment mapping.

Note that Environment mapping assumes that the surrounding of the object is infinitely distant from it. The reason is that variations of positions should never affect the reflected appearance of surfaces, so everything in the environment should be infinitely far away from the object such that the position doesn't cause any mapping changes.

The key implementation is illustrated below (We preserved part of the original texture of the object):

```
Vector3f Material::getEnvironmentColor(const Ray &ray, const Hit &hit) const {
    auto N = hit.getNormal();
    auto d = ray.getDirection();
    auto reflection =  d - 2 * Vector3f::dot(d, N) * N;
    auto environmentColor = 0.5 * cubemap->operator()(reflection);
    auto textureColor = 0.5 * noise.getColor(ray.getOrigin() + ray.getDirection() * hit.getT());
    return environmentColor + textureColor;
}
```

Image 13: Environment Mapping Implementation

The image below is the implementation result:

Image 14: Result of implementing environment mapping

Implementation 3: Web UI & Other Features

No matter how many fancy features we have implemented, it would be meaningless if no one wants to give it a try. Therefore, we decided to build an all-in-one Web UI, freeing ordinary users from the need to use command-line interface, making our project truly accessible to anyone with a modern browser.

Transplantation

First, we need to transplant the whole project into the browser environment. WebAssembly fits best in our use case: it's a pure front-end technology and does not require any support from the back-end server, reducing the cost to maintain our website to zero (when it's hosted by GitHub page). Also, we don't need to design extra communication protocols and server logics if no back-end is involved.

After some investigation, we chose the Emscripten toolchain to build our WebAssembly module of the project, for that it allows minimum modification in our source code. Its *embind* library also supports a highly flexible coding style, and we could use JavaScript values directly in our C++ binding code, making the interaction between the Web UI and the project very easy to implement.

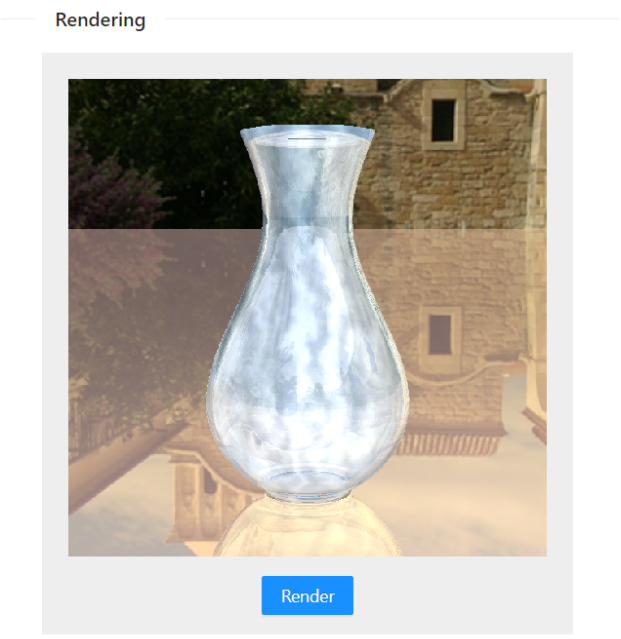

Image 15: The in-browser renderer of Web UI

The result of our transplanted renderer is shown in the Image 15 above. Some optimizations become necessary as we implement the transplantation. The progress bar and the stop button are particularly helpful, since in some configurations the rendering can be very time consuming.

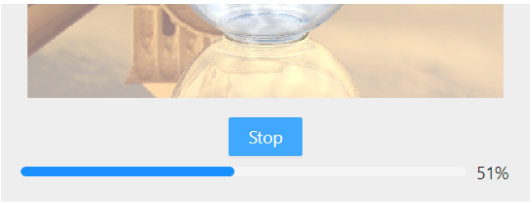

Image 16: The progress bar and stop button during rendering

However, the JavaScript language running inside the browser has a single-threaded nature, and running the time-consuming renderer directly would freeze the UI completely. Not only would the stop button and the progress bar never work, the browser might also force-terminate an unresponsive web page after some while.

Therefore, periodic interruption of the rendering is required, introducing the concept of *feedback frequency*, which defines how many times in a second to interrupt the rendering, reporting the progress and accepting the stop signal. The feedback frequency is provided as a configuration so that the user can set a trade-off between rendering time and UI responsiveness.

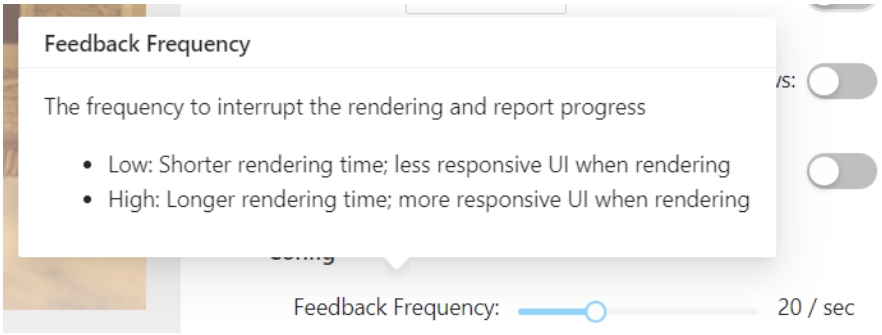

Image 17: The feedback frequency configuration

Argument Settings Panel

To fulfill our promise of improving user experience in Web UI, we provide its first major feature, the argument settings panel, which can set all the command line arguments in an easy and intuitive way.

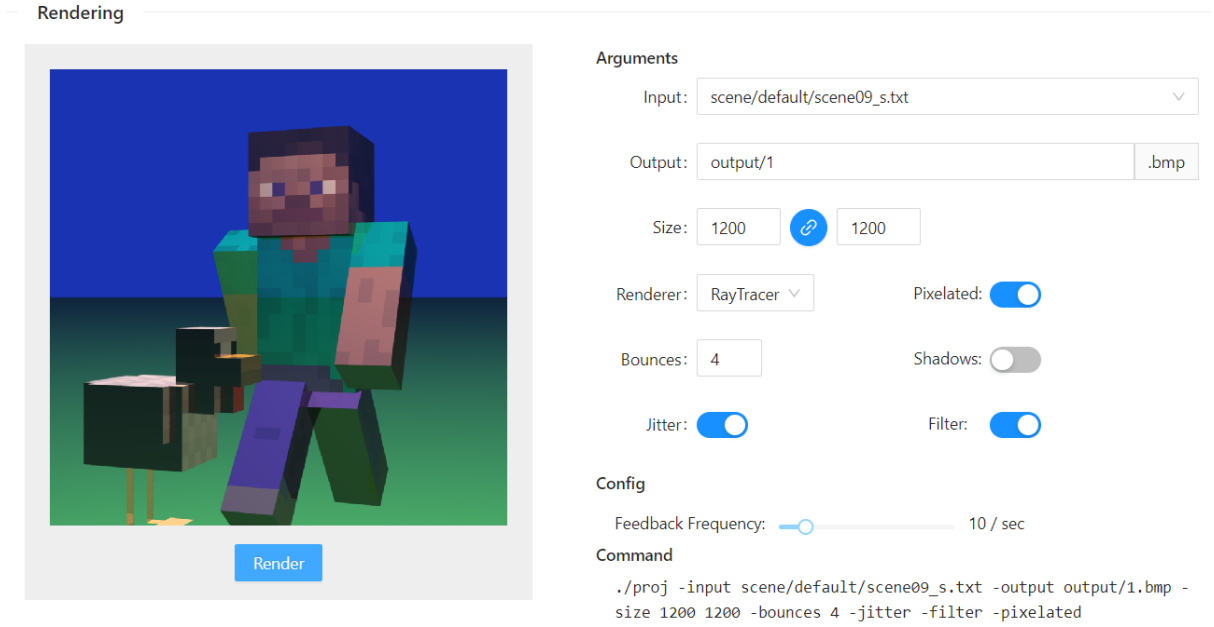

Image 18: The argument settings panel

To explain its design principle, take these two examples: the input scene file selector as an example, which traverses the scene definition files and guarantees that the user's input is valid.

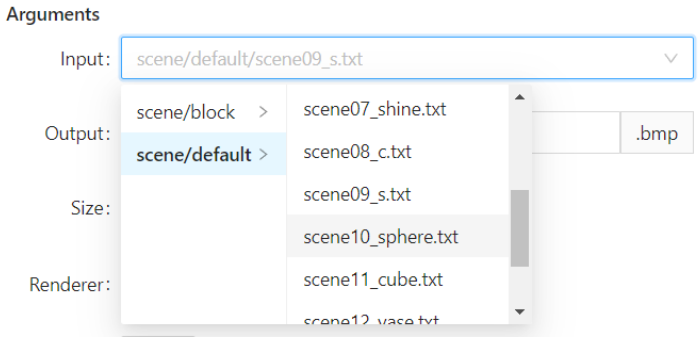

Image 19: The input scene file selector

The command panel, which auto-generates corresponding commands according to the arguments settings panel and makes the CLI also easier to use.

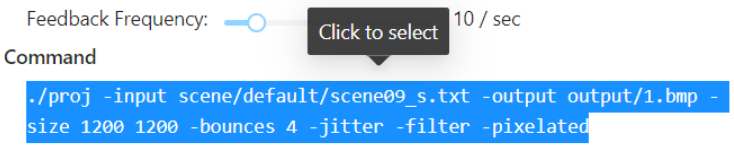

Image 20: The command panel

We then moved on to the next feature of the Web UI, the file system sandbox. It utilized the File System API of the Emscripten toolchain and provides a clean execution environment, as well as an easy way to manage scene definition files and output images.

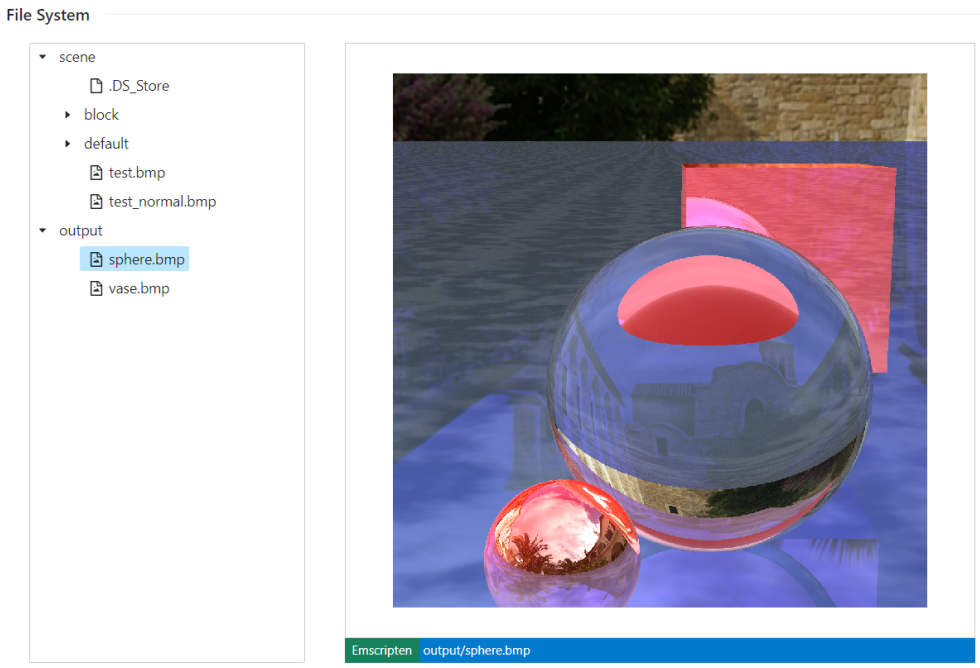

Image 21: The file system sandbox

Scene Definition Editor

With the file system sandbox, we are not satisfied to just allow the users to view the scene files. To make the most of our project, the users should be able to get their hands dirty with scene files and see how their edits impact. However, editing the scene definition file is no easy task, due to the lack of code highlighting, error suggestions, etc. We don't want the users to be bothered by these problems.

Therefore, we propose the second major feature of our Web UI, the scene definition editor, supporting a full range of functionalities including code highlighting, error suggestions, and auto-formatting. These functionalities are all made possible by parsing and analyzing the scene definition domain-specific language (DSL) that was defined in our assignments. The parsing process is powered by the Chevrotain Parser Building Toolkit and consists of 4 stages: plaintext, lexed tokens, Concrete Syntax Tree (CST), and our final product, Abstract Syntax Tree (AST).

In the Image 22, the parsing program structure is visualized on the left side. The scene definition text is first lexed into tokens, and then parsed by Chevrotain to build the CST. Finally, the hand-built AST parser generates corresponding AST nodes and binds them with the CST. The reason for the two-way binding between the CST and the AST is to allow generic traversing of the AST using the CST. On the right side, there's a rail diagram showing how the Chevrotain CST parser analyzes the tokens.

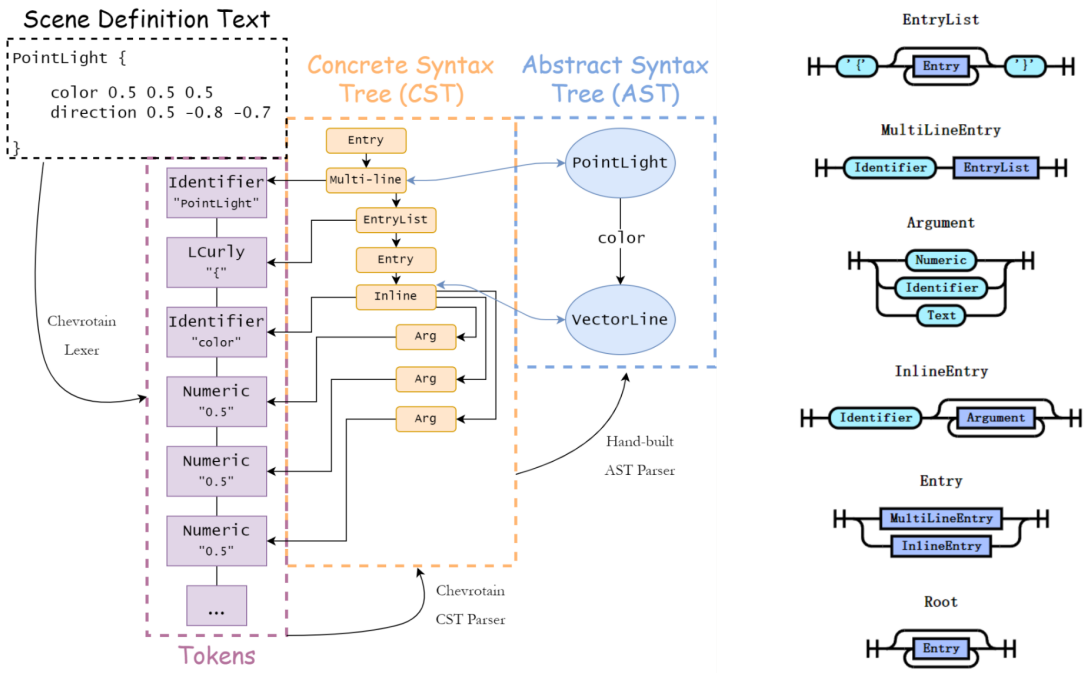

Image 22: The parsing of the scene definition file

With the AST built, we can easily implement the desired functionalities. We used the Monaco Editor library which provides various relevant API. Starting with the code highlighting, the editor colors each word according to the semantic tokens generated using our AST. We can have a customizable color for every scene object and arguments.

```
1
2   PerspectiveCamera {
3       center 0 6 20
4       direction 0 -2 -20
5       up 0 1 0
6       angle 30
7   }
8   Lights {
9       numLights 2
10      PointLight {
11          position 7 15 10
12          color 0.5 0.5 0.5
13          falloff 0.001
14      }
15      DirectionalLight {
16          direction 0.5 -0.8 -0.7
17          color 0.5 0.5 0.5
18      }
19  }
20  Background {
21      color 0.4 0.2 0.8
22      ambientLight 0.4 0.4 0.4
23      cubeMap tex/church
24  }
25  Materials {
26      numMaterials 2
27      PhongMaterial {
28          cubeMap tex/church
29          diffuseColor 0.6 0.5 0.5
30          specularColor 0.5 0.5 0.3
Emscripten  scene/default/scene12_vase.txt
```

Image 23: The code highlighting example

The error suggestion is another powerful feature of our scene definition editor. With the power of AST, the accuracy and level of detail is far better than what simple regular expression can provide. Images 24-27 show 4 examples of the error suggestion.

Image 24: Error suggestion: Unexpected argument

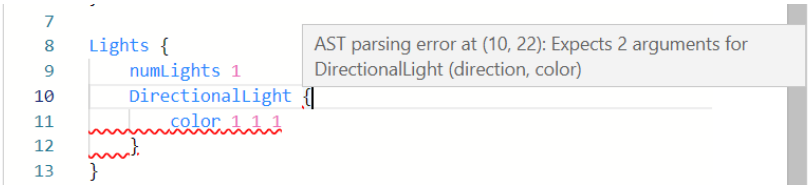

Image 25: Error suggestion: Missing argument

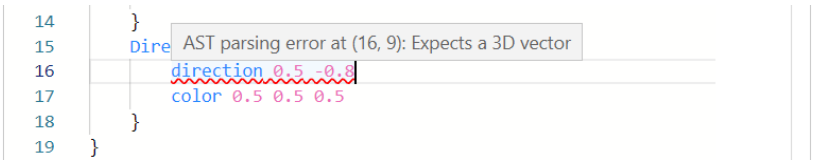

Image 26: Error suggestion: Mismatched value type

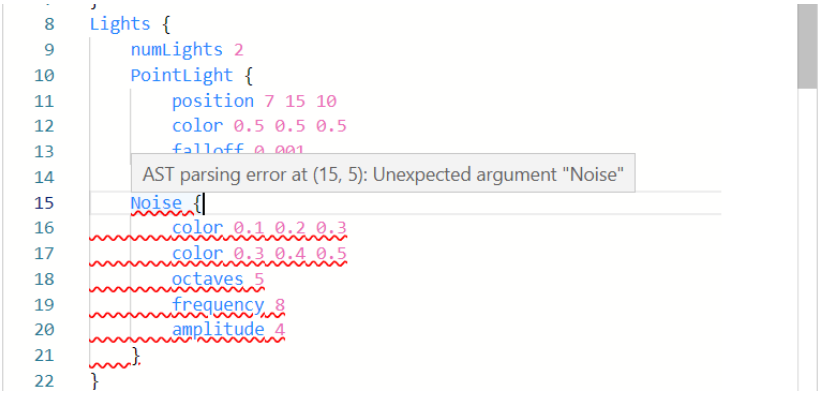

Image 27: Error suggestion: Misplaced block

The last functionality we implemented is the auto formatter. The user only needs to press Shift+Alt+F and the auto-formatting happens.

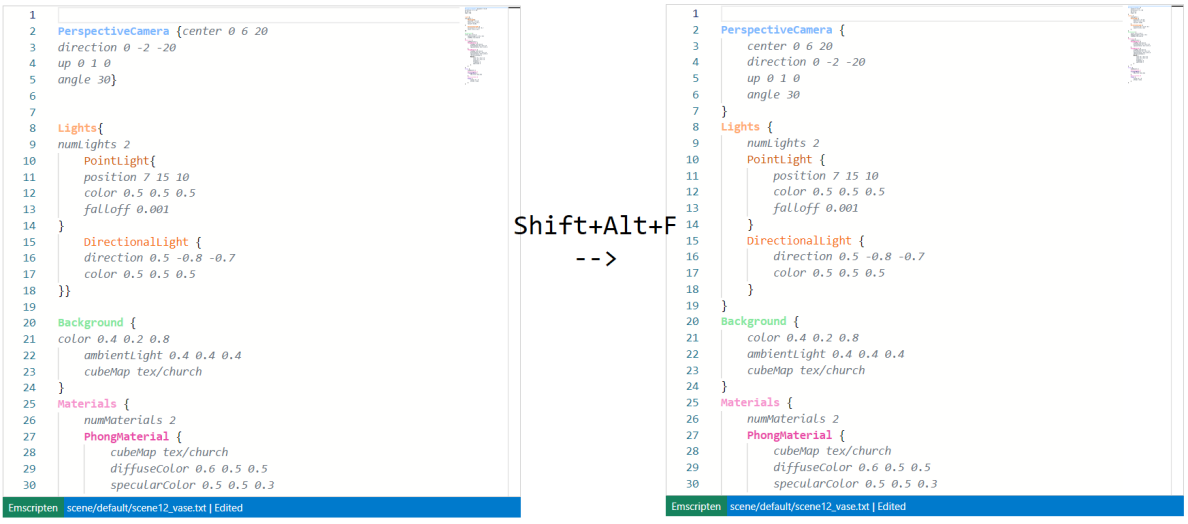

Image 28: Auto-formatter example

Future Improvements

1. Build more customizable scenes that are related to our topic, which is the Minecraft-like Game Scene Renderer. We have not implemented custom minecraft-like scenes because it would involve converting in-game assets and would be time-consuming.
2. For the blurring feature, our implementation simply applies depth of field blurring, which may not be applicable for simulating those moving objects. We plan to add a motion blurring feature for this case.
3. Regarding environment mapping, in our previous implementation, we only considered one object and the cube map, we could extend our implementation to support environment mapping of multiple objects.
4. Extend the features of the scene definition editor to build an even more sophisticated IDE, which includes detailed visualization feedback as the user edits the scene definition: visualized color palette, vector direction, material samples, etc.
5. Implement an obj file editor similar to the scene definition file editor, supporting the full range of functionalities based on AST parsing.
6. Improve the interaction between the renderer module and the Web UI to support more detailed feedback, such as intersection positions on geometrics, which can be the basis for building some rendering process demonstrations for teaching purposes.

References

[1] F. Durand and B. Freeman, 'Focus and Depth of Field', Massachusetts Institute of Technology, n.d.

[2] H. Shen, 'Distributed Ray Tracing', The Ohio State University, 2012.

[3] Joey de Vries, June 2014. Normal Mapping, LEARN OpenGL. https://learnopengl.com/Advanced-Lighting/Normal-Mapping

[4] Randima Fernando, Mark J. Kilgard, Chapter 7. Environment Mapping Techniques, The CG Tutorial

[5] Joey de Vries, June 2014. Cubemaps, LEARN OpenGL. https://learnopengl.com/Advanced-OpenGL/Cubemaps