Rasterizer

Jason Haciepiri, Maria Marinova

Rasterization is a way of computing a 2D image representation of a 3D scene. It is faster than raytracing, and hence typically used for real-time visualisation. However, it is not as accurate as raytracing.

1. Compiling and running the code

The project includes a `Makefile`, which compiles the code. It contains the `-O3` flag in `CC_Opts` so that the code is compiled in the most optimised way. The `-lX11` flag is added to the linker options `LL_Opts` to allow communication with the X11 Windows Manager. The flag `-fopenmp` is added to the `CC` flags. The last two flags allow for parallelism in the code, which will be elaborated on in Section 3.

The project is compiled and run using the command:

```
make && .\Build\skeleton
```

2. Parts 1 and 2

The first aspect implemented was projecting points from 3D to 2D in perspective. As the pinhole camera model is used, the relation is derived using similar triangles; the width and height of the screen are also taken into account.

The camera can be moved via the keyboard – translation and rotation are implemented in the `Update()` function.

The vertices of the triangles which compose the 3D scene are therefore projected to the 2D image. Interpolation is used to draw the edges between them, and hence produce the respective triangles in the 2D image.

A depth buffer is implemented to ensure the 2D representation reflects the position of the objects in the 3D scene.

The illumination can be computed per vertex or per pixel – the former is faster but less accurate. It is computed as a combination of direct and indirect light, and is also affected by the objects reflectance. While in the base case the reflectance is simply the colour of the objects, it can be extended to hold the reflectance properties of the materials.

In order to simulate illumination, a light source which can be moved via the keyboard is added.

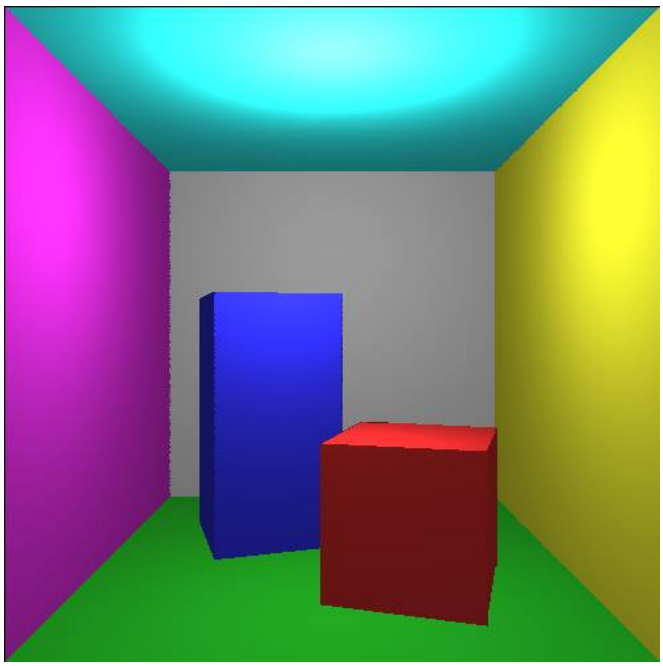

Figure 1: *Rasterizer base*

3. Extensions

Importation of OBJ Files

OBJ files provide a specification for 3D geometry, with a simple format for defining the positions of vertices and the faces that they belong to. Support for OBJ files was implemented with a custom parser in the `LoadOBJModel` method, which takes the path of an OBJ file, the path of a material library, and an empty Triangle vector. It works by scanning for key tokens such as vertices, normals, faces and material names in the OBJ file. Through parsing the OBJ faces, each of the scanned vertices can be assigned to a triangle and rendered within our scene, with a corresponding material. Faces are written to the OBJ file underneath the material name that they are assigned to, and so the parser must only keep track of the last read material. When it comes across a new material name, the parser accesses a separate material library file and scans for a matching name. Materials can have a wide number of attributes however for diffusion it is only necessary to return a single 3D colour vector from the material library, which is then assigned to the face's triangle struct.

As a result of this extension, it was possible to create models in Blender and import them into the scene. Examples are given in Figures 2 and 3. To render Figure 3 in the submitted

project, comment out the default `LoadTestModel` method call on line 75 of the `skeleton.cpp` file, and uncomment the `LoadOBJModel` method call on line 76.

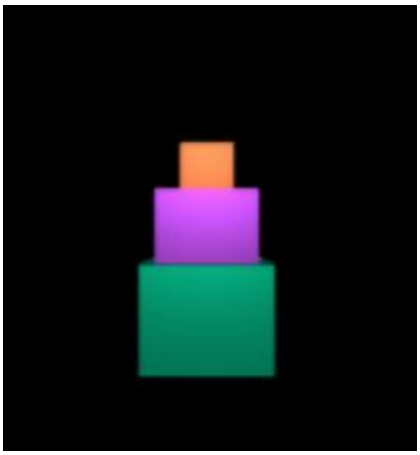

Figure 2: *Imported box stack from Blender OBJ file.*

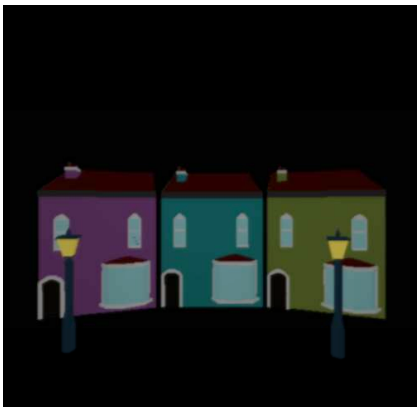

Figure 3: *Imported house models and gaslamps from Blender OBJ file.*

Bresenham's Line Algorithm

The project implements Bresenham's Line Algorithm in the `Bresenham_Interpolate` method, replacing the usage of the naive interpolation used in the `DrawPolygonRows` method. It is more efficient than other line approximation algorithms such as Wu's algorithm, due to only using integer addition, subtraction and bit shifting as opposed to more expensive operations.

Depth of Field

As with the raytracer, a depth of field effect was created using a 5 x 5 Gassuian filter with focus and slack variables. However, unlike the raytracer, storing the depth information of pixels was already necessary in order to render the scene correctly, and so this feature did not require extra memory.

$$kernel = \frac{1}{273}\begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix}^{[1]}$$

Parallelisation via OpenMP

Similarly to the raytracer, the render time was improved by parallelising loops with no data dependencies. This was done with means of OpenMP – `pragma omp parallel for` was used above the main `for` loops in the `Draw()` function. Until that point, `vec3 reflectance` was a global variable, however, to make the loop parallelisable, it was passed as an argument – `vec3 currentReflectance` – to all relevant functions.

Additionally, the `for` loop in the `DrawDepthField()` function was also parallelised once completed.

As mentioned above, the `-lX11` is necessary to allow communication with the X11 Windows Manager. The flag `-fopenmp` and the header `#include <X11/Xlib.h>` were also needed to allow usage of OpenMP.

References

1. `http://homepages.inf.ed.ac.uk/rbf/HIPR2/gsmooth.htm`