Lab 2 report

Victor Hansjons Vegeborn
victorhv@kth.se

DH2323 Computer Graphics and Interaction
KTH Royal Institute of Technology

The report focuses on the main parts of the lab and is presented in the order the lab was solved. Equations are referenced from the lab instructions with their corresponding index and therefore is assumed to be known to the reader of this report and is never rewritten, unless they are needed to further explain the solution to this lab.

1 Intersection of Ray and Triangle

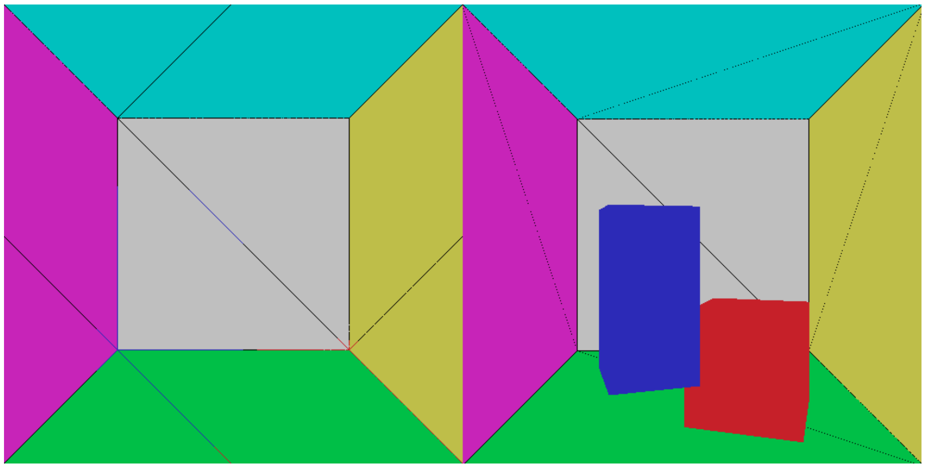
Fig. 1: Left: rendering error occurring when not utilizing the correct conditions for the inequalities 7,8,9. right: the blue box is rendered over the red due to rendering the first encountered surface.

Intersections between a line and a plane is given mathematically by equations 10-23. However, as a line spans infinitely in one dimension and a plane span infinitely in two dimensions, the inequalities 7,8,9 and 11 is used to limit the intersection to a triangle as defined within the lab.

When implementing the ClosestIntersection-function, two problems was encountered. The first is illustrated in figure 1, on the left. The inequalities written in the lab instructions rendered the diagonal on each triangle black. The correct inequalities used within the solution to this lab are the following:

$$0 \leq u \tag{7}$$
$$0 \leq v \tag{8}$$
$$u + v \leq 1 \tag{9}$$

u and v are scalars in $\mathbb{R}$ to the perpendicular edges of the triangles that should be rendered. As the points on the triangles boundary should be rendered, the updated inequalities above assures this by allowing one edge to be 1, while the other is 0, and all possible values in between. This also includes the diagonal of the triangle.

The other render error occurred when returning the closest intersection from the camera. On the right in figure 1, the blue box is overlapping the red. The error was generated due to returning the first encountered intersection, not the closest. This was handled by iterating through all triangles, not just until the ray intersected with the first encountered, and comparing the distances between the first ray and the next (if there was one).

2 Tracing Rays

Tracing rays was done by sending out rays from each pixel of the screen. For every pixel, a 3d-space point $\mathbf{d}$ of the view plane was calculated, and was sent to the ClosestIntersection-fucntion. This point was assumed to be the direction of the ray, from the camera position.

By setting the camera to $(0, 0, -3)$ and the focal length $f = 500$ pixels (the same as the screen height and width), the whole room could be rendered in a single frame. The vertical and horizontal field of view, with the focal length and height/width ratio set as stated above, is both calculated to 90 degrees. If the room is defined by the dimension in equations 1,2 and 3, then the screen width and height of 500 pixels should cover 2 world space units. Be offsetting the camera by -3 on the z-axis, a full coverage of the room is rendered as a result of the 90 degree field of view. This relationship is illustrated for the horizontal field of view in a orthographic manner in figure 2.

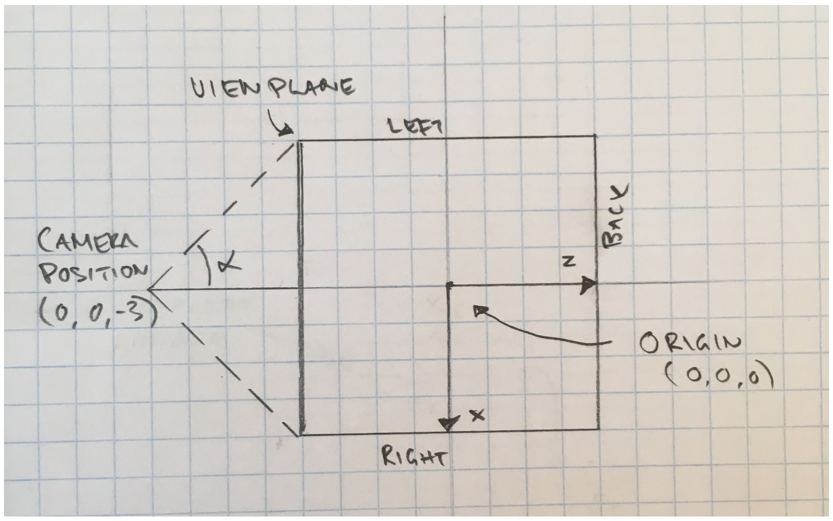

Fig. 2: If W is the screen width and f is the focal length, the horizontal field of view fov is given by: $fov = 2\alpha$, where $\alpha = \tan \frac{W}{2f}$. The offset of the camera places the view plane at -1 on the z-axis.

3 Moving the Camera

To move the camera around with keystrokes, a rotation matrix $\mathbf{R}$ was calculated for each time a keystroke was pressed in the update function. To yaw the camera with some degree y in the y-axis, the rotation matrix $\mathbf{R}$ was defined by:

$$\mathbf{R}(y) = \begin{bmatrix} \cos(y) & 0 & \sin(y) \\ 0 & 1 & 0 \\ -\sin(y) & 0 & \cos(y) \end{bmatrix}$$

To actually rotate the camera, a new ray direction $\mathbf{d}'$ was calculated in order to keep the view plane relative to cameras rotation. The new direction was computed by:

$$\mathbf{d}' = \mathbf{d} \cdot \mathbf{R}$$

The translation for the camera had to be coherent with the rotation of the camera. This was done by adding multiplying some movement velocity to different columns of the matrix, then adding them to the camera position, depending on the direction the keystroke implied.

4 Illumination

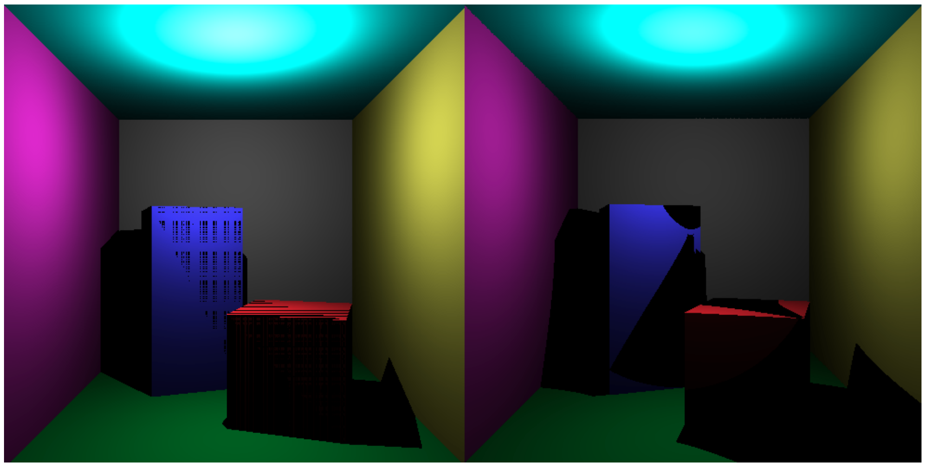

Fig. 3: These are images rendered when trying to debug the error that occurred when reusing the ClosestIntersection-function for shadows.

The illumination part of the lab was probably the hardest to solve due to the rendering error that is illustrated in figure 3. To implement shadows, the function ClosestIntersection had to be reused as it could send a ray from any surface point to the defined light source.

The errors in figure 3 originated from a floating point error and not normalizing the surface-to-light direction. The surface point from which the ray was sent, was considered a intersecting surface. The solution to this was to offset the starting point i.e the surface point, in the direction of the ray with a tiny amount (0.00001 in the implementation) and by normalizing the direction.

5 Result

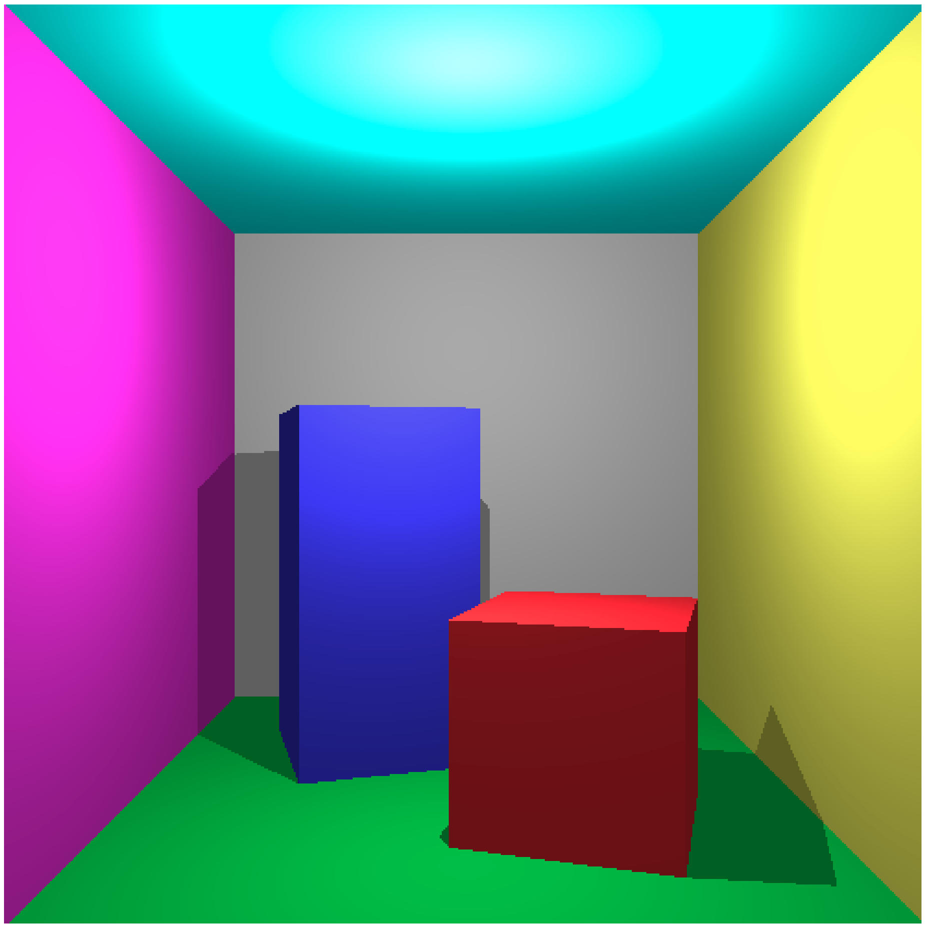

Fig. 4: The output of this lab. A still frame computed by ray tracing each pixel with working illumination.