Kungliga Tekniska Högskolan
Valhallavägen 79
100 44 Stockholm

Computer graphics and interaction

« Rasterization »

April 24^{th} - May 19^{th}

Author
Rémi Domingues

Teachers
Christopher Peters

Scholar year 2014-2015

Introduction

This report describes the implementation and results of a rasterization algorithm applied to a wel-known 3D model. The algorithm uses a pinhole camera and supports direct illumination with camera rotation and translation,

2 Drawing vertices

Vertices of the 3D model are projected on the screen according to the following formula with R the rotation matrix describing the camera direction, C the camera position, P the 3D coordinates of the vertex and P' the 2D coordinates of the pixel on the screen.

$$P' = (P - C) * R \tag{1}$$

Vertices x and y coordinates are then updated according to the pinhole camera model in order to obtain a perspective projection (f focal length) :

$$\begin{aligned} x &= f\frac{x}{z} + \frac{width}{2} \\ x &= f\frac{y}{z} + \frac{height}{2} \end{aligned} \tag{2}$$

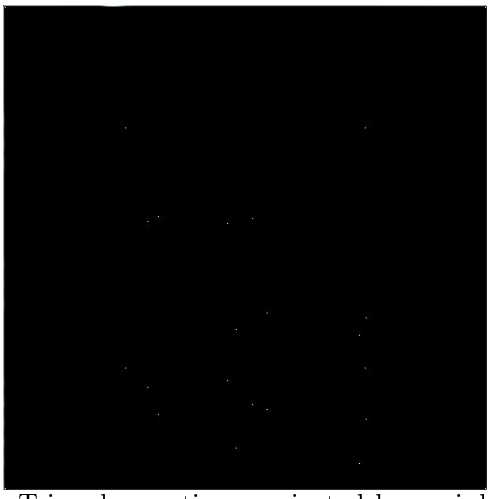

FIGURE 1 – Triangles vertices projected by a pinhole camera

3 Drawing edges

Once the 2D position of the vertices is computed, we interpolate the 2D position of each point of the edges.

The camera translation and rotation using the arrow keys or the mouse has also been implemented. Rotation is however only applied to the Y axis.

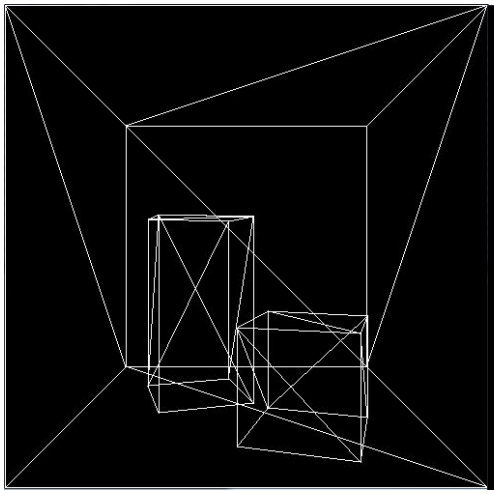

FIGURE 2 – Triangles edges

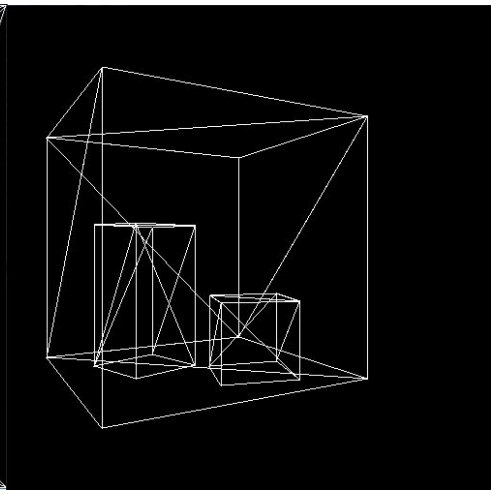

FIGURE 3 – Triangles edges with camera rotation and translation)

Due to the current absence of clipping process in this algorithm, undesirable visual effects (figure 4) may appear when rotating the camera of 90°. Those effects come with higher computation time.

Figure 5 shows a previous bug which happened for a rotation of 180° and was caused by pixels displayed with coordinates out of range.

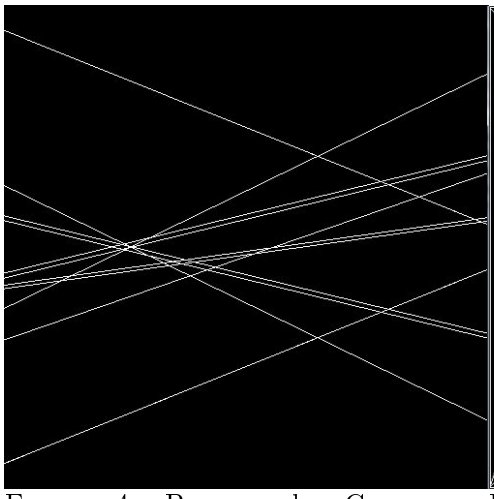

FIGURE 4 – Bug sample - Camera rotation on the Y axis of 90°

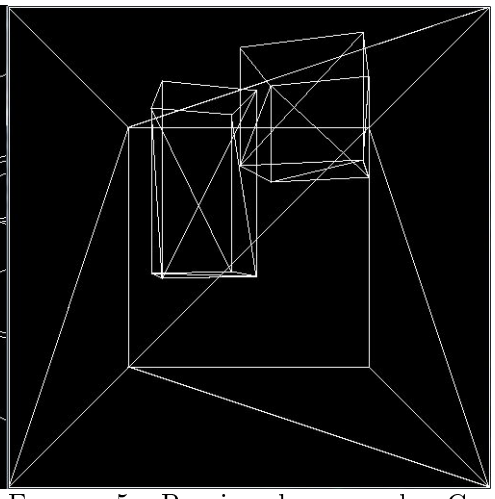

FIGURE 5 – Previous bug sample - Camera rotation on the Y axis of 180°

4 Filling triangles

In order to fill triangles, the rasterization algorithm computes rows of pixels describing a given triangle.

A row is an array of pixels positions interpolated from the row boundaries. Those boundaries have the same Y coordinate but different X coordinates.

The boundaries are obtained by looping through the pixels of the edges in order to retrieve the minimum and maximum X for a given Y in the specified triangle.

By interpolating the pixels positions between the boundaries in each row, we eventually display every pixel of a triangle according to the triangle color.

Figure 6 shows a rendering of the algorithm, computed in 8 ms. For a comparison, a similar rendering was obtained with the raytracer in 370 ms. We can observe that the blue box is displayed on top of the red box. This issue should be solve in the next section using a depth buffer.

Figure 7 is a previous bug sample obtained when rotating the camera and caused by invalid pixels positions when triangles vertices were out of the field of view.

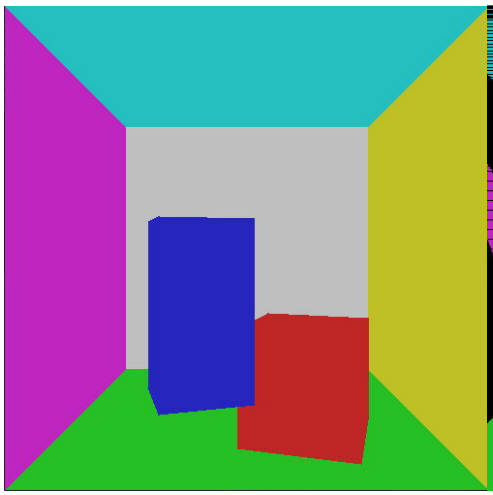

FIGURE 6 – Filled triangles

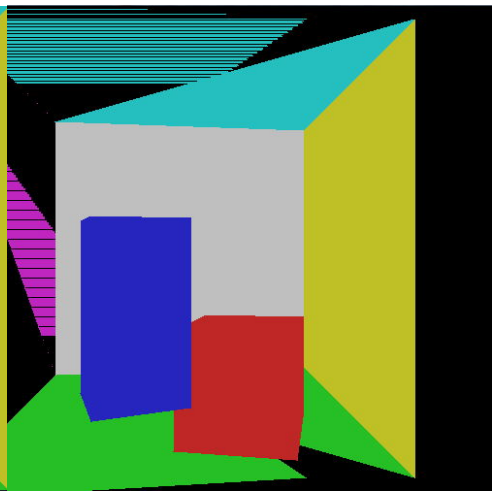

FIGURE 7 – Previous bug sample

5 Depth buffer

The depth buffer is a structure containing the depth of each pixel displayed on the screen. When computing a new pixel, the rasterization algorithm should draw the corresponding pixel only if this one is closer to the camera than the on currently displayed.

This could be achieved by storing the interpolated Z coordinate of each pixel displayed in the depth buffer. However, due to the perpective projection applied by the pinhole camera model, this coordinate does not vary linearly. This is why we use here $\frac{1}{Z}$ instead, and display a new pixel when its $\frac{1}{Z}$ coordinate is larger.

In order to retrieve the initial Z coordinate, we could simply divide a linearly interpolated Z coordinate by $\frac{1}{Z}$. This value will be used in section 6.2.

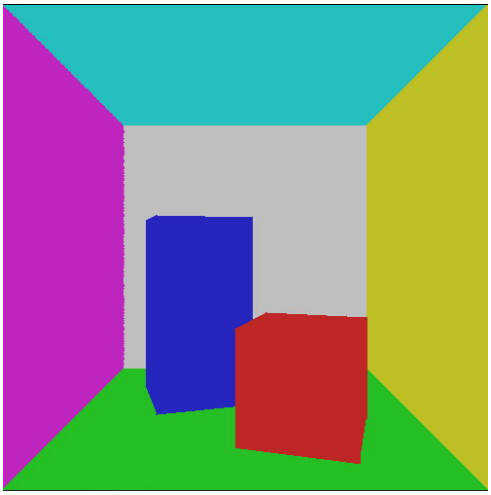

FIGURE 8 – Filled triangles with depth buffer

6 Illumination

6.1 Per vertex - Interpolated light

An illumination method giving speed renderings but poor realism (especially with large triangles) is to compute the illumination of every vertex and then to interpolate the illumination of the pixels inside the triangles.

The illumination model, same as the direct light model in the raytracer, is as follows, with D the light power at the vertex, P the power of the light source, i.e. energy per time unit for each color component (14.f * vec3(1, 1, 1) for example), r the distance from the vertex to the light source, $\hat{n}$ the normal of the triangle and $\hat{r}$ a unit vector describing the direction from the vertex to the light source.

$$D = \frac{P * max(\hat{r}.\hat{n}, 0)}{4\pi r^2} \tag{3}$$

The color of a vertex is then computed according to N the indirect illumination constant and ρ the RGB vector of a triangle.

$$R = \rho * (D + N) \tag{4}$$

Once the vertices illumination R is computed, we interpolate it for the pixels of the triangles edges, and then for the pixels of each row.

The rendering displayed in figure 9 is computed in 14 ms.

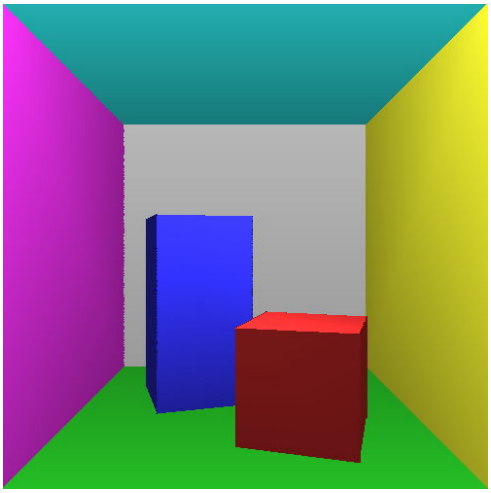

FIGURE 9 – Illumination with interpolated light

6.2 Per pixel - Interpolated positions

Finally and to make our rendering more realistic, we apply the illumination model to every pixel instead of every triangles vertices.

To do so, we interpolate the 3D position of the pixels in addition to their 2D position. Nevertheless, interpolating the Z coordinate linearly gives the renderings below.

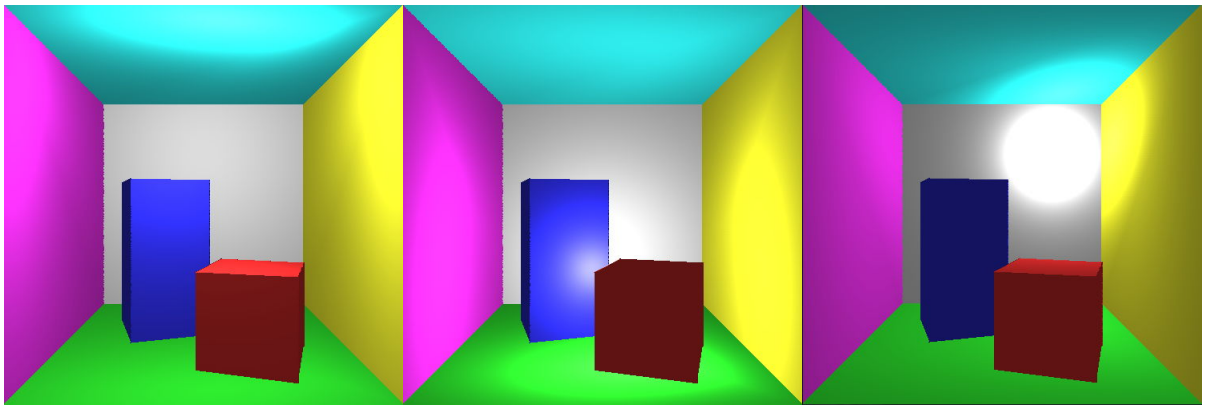

FIGURE 10 – Previous bug sample - Illumination - Interpolated position

Those renderings have an inaccurate light surface (figure 10.1) which may result in unrealistic results (figure 10.3) when the light moves. The light translation is triggered by the keys W, A, S, D, Q and E.

This is why we must divide the interpolated Z by $\frac{1}{Z}$. The final renderings are computed in 30 ms, while the raytracer gives simlar results (yet including shadows) in 750 ms.

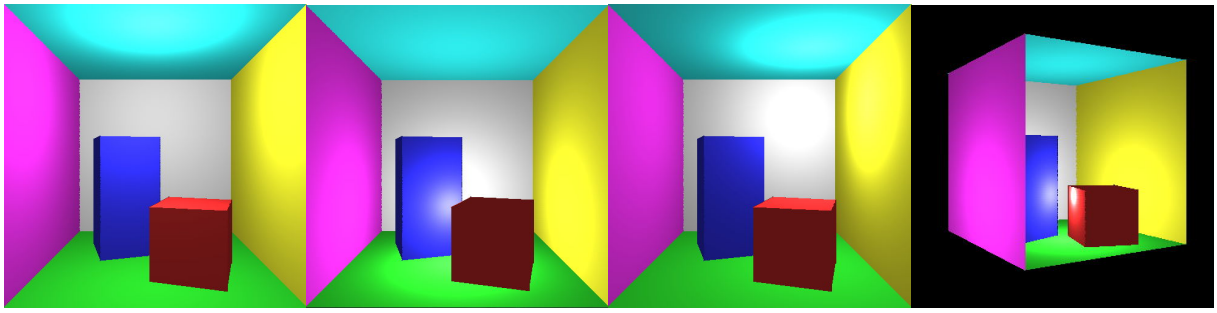

FIGURE 11 – Illumination - Interpolated position