Non-photorealistic Rendering

Jakub Olejnik Trevina Litchmore

Lund University
Sweden

Abstract

For this project, we implemented a non-physically based renderer that produced images that mimicked sketches. The basis of our implementation was the paper by [Ježek et al. 2018] in which multiple passes were used to generate this effect. We optimized the algorithm by implementing it with one pass through the geometry shader instead of three separate passes. Additionally, we combined some aspects of the sketch renderer with diffuse shading and dithering to create a renderer that would output images in the style of the famous Pop Art artist, Roy Lichtenstein. Our scene is diffuse shaded, but the shadows are rendered using a dithering technique adapted from an online article by [sysrpl]. For relatively large scenes, our renderer performed quite well. However, this aspect of our program was not rigorously tested, and it would have been interesting to see how well it would have performed with larger instances of geometry (i.e. scenes with hundreds of thousands of triangles) or even in combination with animation.

1 Introduction

Realistic computer graphics were always associated with the most advanced technology and memorable experiences. Nowadays this is definitely still true, although the advancement in the field of 3D graphics is not as noticeable (especially by people outside the industry) as in early 2000s. Hence, the term photo-realism became deprecated, since somewhat photo-realistic graphics are easily achievable on almost every hardware we use today and are not considered a nuisance unless they are ultra photo-realistic. This led to creators using unconventional ways to stylize their work, to make it more distinguishable from others. In the past, this stylization was often the consequence of hardware limitation,but today it is is a conscious choice on part of the 3D graphics artist: non-photorealistic rendering.This phrase can be used to describe a variety of styles, but in this project we focus on an implementation of a pencil sketch shader and a toon shader with dithering.

2 Application

Silhouette detection is a crucial part of our pipeline. In order to perform computation properly, appropriate data representation is required, which in our case is in the form of adjacent triangles. When a 3D model is loaded from an .obj file, instead of saving the object's vertex indices directly to the index buffer, a few additional operations are needed.

To build this triangle adjacency data, we used a hash table. The implementation of this hash table was documented within an online discussion [Reed]. We used the std::unordered_map data structure, where the key is an edge of a triangle, and the value is a vertex adjacent to this edge.

For removing duplicate edges, we based our solution on the article by [Meiri] where another hash table is used for removing indices pointing to an already saved vertex. This me method was much faster than the one proposed in [Ježek et al. 2018]. In this paper, removing duplicate edges involved iterating over the triangle adjacency data multiple times by comparing all the saved edges. In terms of efficiency, the hash table was a much faster way to achieve the same result.

Once the data was properly processed, it was run through the rendering pipeline. During the first pass a depth mask was generated and stored in a texture for later use. In the second pass we performed silhouette detection. Triangle adjacency data was used as an input to the geometry shader. All of the edges were tested for visibility and significance based on the depth test and the result of the dot product between their normal and light direction. Before being emitted, the lines were divided at a random point between them, and then the vertices were randomly chosen to be displaced by a random vector. Both random values were read from the noise texture, which was generated at the initialization of the program. It was a 2D array of RGB values, in the range from 0 to 1, generated using uniform std::uniform_real_distribution. Achieving the pop art effect required one additional step.

During the first pass, a basic diffuse texture was also created. It was built in a fragment shader that used the object's material diffuse data to apply a flat color to every surface. In the next step, the shade of the calculated color was used to compute the intensity of the dithering effect. To obtain an outline that was just a single fine line, the step of displacing vertices was omitted.

3 Results

Final effects of the project can be seen at Fig. 1 and Fig. 2.

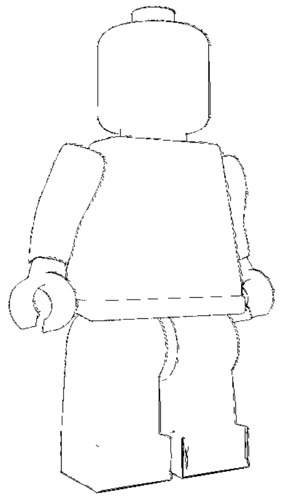

Figure 1: Final result of a model being rendered using the implemented pipeline - Sketch effect

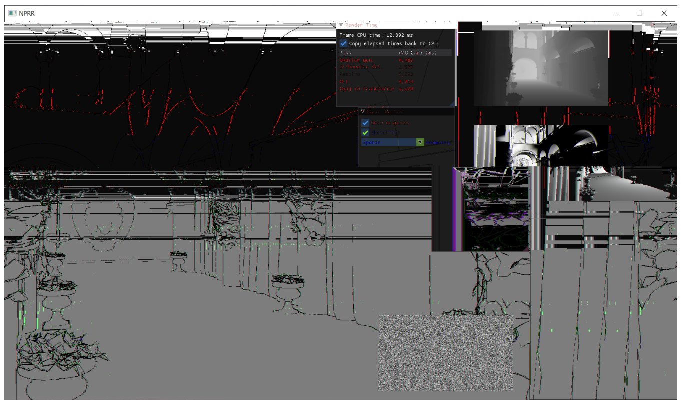

Figure 3: Sponza scene with controls and textures visualizing steps in the pipeline

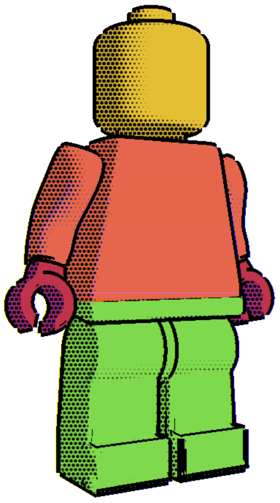

Figure 2: Final result of a model being rendered using the implemented pipeline - Pop Art effect

When it comes to performance, the project was mainly tested in two environments:

A) a desktop PC with a dedicated GPU - Nvidia GeForce GTX 1050 Ti,

B) a laptop with integrated GPU - Intel HD Graphics 520.

In environment A, the project rendered in under 20 ms for each model in both styles, even for the most complex one - sponza (see Fig. 3). It was only for this scene that part of the pipeline responsible for silhouette detection was working longer than 1 ms. Depending on the intensity of the camera movement, time elapsed varied from 2 ms to 5 ms.

In environment B, the rendering time was still satisfactory for simple models with 1000 vertices or fewer, but when more complex one were introduced the drop in performance was noticeable. Sponza was rendered in under 23 ms when the camera was still and around 29-34 ms when it was in movement. The time to render silhouette detection increased by 13 ms.

4 Discussion

When it came to implementing the pipeline as described in Ježek et. al (2018), the first part of the implementation went relatively smoothly. The noise texture and depth mask were generated according to what was written in the paper. However, creating the triangle adjacency list was the most difficult aspect of the implementation because OpenGL does not provide a function to generate such a list like the High Level Shading Language (HLSL) from DirectX [White et al.]. Thus, we had to program an algorithm that would easily generate a triangle with adjacent vertices, which went through a few iterations.

At first we created a 2xN matrix, where we had N-columns with two rows. Each column and index represented a triangle within the mesh. We iterated over this matrix to find the adjacent points of each triangle. However, it was too computationally expensive when we tried it with larger models such as the sponza geometry that we used in our project. Afterwards, we implemented a hash table of edges that allowed us to quickly determine the adjacent vertices for each triangle. With this version of the algorithm, we were able to quickly render the geometry.

However, when it came to implementing the other two graphics pipelines (the one for the segmentation of edges and the other for the transformation of those edges into strokes), we were not successful. In this case, we programmed a workaround that was a good approximation of what we wanted to accomplish. Instead of creating the two additional pipelines, we computed the strokes in the same geometry shader where we tested the edges against the depth masks and removed the minor edges (edges where there is no significant change in geometry). In this case, we were able to save on the amount of processing needed to generate the same effect.

For the sketch generation component of the project, one thing that could be improved would be the ability of the algorithm to capture finer edge detail. When looking at the sponza geometry capture, the lines on the face of the lion were not captured well by the algorithm.

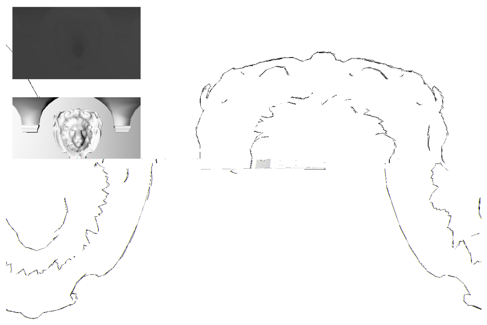

Figure 4: Lion head model from sponza scene after being passed through the sketch renderer pipeline.

Perhaps altering the increasing the limits of when we render the lines or adding a second pass for those finer lines, would be a way to improve the algorithm.

5 Conclusion

With the optimized sketch rendering algorithm combined with dithering and diffuse shading, it would be interesting to see how well it performs with animated objects. For each render it redraws the stroke lines, as well as the circles for dithering. Animation also includes quite a bit of computation as well. The combination of the computational complexity could make it difficult to apply such a renderer in those cases, but it would be an interesting avenue to pursue. Additionally, as has been evidenced in many films both within the art house genre and in mainstream entertainment, the unique styles generated by such shaders will continue to be in-demand due to the need for unique visuals to standout from the competition.

References

JEŽEK, B., HORÁČEK, D., VANĚK, J., AND ANTONIN, S. 2018. Non-photorealistic Rendering and Sketching Supported by GPU: 5th International Conference, AVR 2018, Otranto, Italy, June 24–27, 2018, Proceedings, Part I. 07, 447–463.

MEIRI, E. Tutorial 39: Silhouette Detection. `https://ogldev.org/www/tutorial39/tutorial39.html`. Accesed: 2021-12-07.

REED, N. Building triangle adjacency data. `https://gamedev.stackexchange.com/questions/62097/building-triangle-adjacency-data`. Accesed: 2021-12-08.

SYSRPL. Hatching Shaders. `https://www.codebot.org/articles/?doc=9525`. Accesed: 2021-12-10.

WHITE, S., MARTINEZ, J., BATCHELOR, D., AND SATRAN, M. Geometry-Shader Object. `https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl-geometry-shader#return-value`. Accessed:2021-30-06.