COSC 4306

Final Report

Mathieu Carriere
Bradley McFadden
Joy McGibbon

April 4, 2022

Abstract

Non-photorealistic rendering is a combination of computer graphics and artistic techniques used in a variety of applications to emphasize information from a photorealistic rendering or present it in a different style. This can be done either through artistic techniques that create a particular mood, or by emphasizing details of the image/model, or both.

Our goal is to create a system and algorithm to render 3D models in a style similar to 2D animation using OpenGL and shaders. Transforming the rendered model will be achieved through cel shading, lighting effects, modifying textures, and adding contour lines to the image. We will also examine the effect of applying these transformations to the model in both the vertex and fragment shaders.

Introduction

In the field of computer graphics, non-photorealistic rendering is a somewhat circular term that refers to the use of rendering techniques to achieve a non-photorealistic effect.

Prior and related work

Description

Cel shading

Suggestive contours

Outlines

Implementation

The described features were implemented into a real-time rendering system. The system itself was written in OpenGL3. Specifically, we used LWJGL's implementation of OpenGL. We took advantage of the vertex and fragment shaders that OpenGL3 requires to be used. The use of vertex and fragment shaders allows us to implement most algorithms on the GPU, which saves CPU cycles and increases the performance of the application.

Essentially, our system is a real-time renderer for 3D models that handles lighting, arbitrary textures, and allows shading effects to be toggled on or off for comparison. A camera is supported so that the model can be examined from a variety of viewing angles. A high-level overview of the system is presented in Figure 1.

The first stage of our system takes in .obj files and reformats parts of the file in order to make it easier to parse. This was implemented as a separate Python program, since it needs only be done once per model. The second stage of the

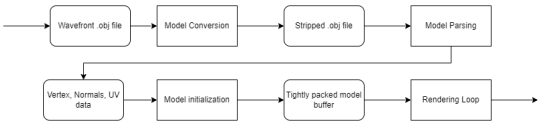

Figure 1: An overview of stages in the system

system parses a .obj file to produce a prototype of a Model that our system is later able to render. In doing so, the vertices, vertex normals, and texture coordinates from the file are packed into the prototype model. Additionally, the indices that use this data to define faces of the model are also read. At this stage, textures are also read in for the model from the .mtl file referenced by the .obj file. One caveat is that imported models should be triangulated, as our system has no procedure in place to triangulate polygons. The next stage of the system packs the Model data into a format that the vertex shader can understand. An strided array of nine floating point values per vertex is created. The first three values each stride represent the vertex data, the next three are the vertex normal, and the final three contain texture coordinates. These values are passed as attributes to the vertex shader. The buffer only ever has to be calculated once per model, so this stage does not repeat. Every frame, each model is rendered. The render proceeds by passing light and camera parameters to the vertex shader, along with the vertex, vertex normal and texture coordinates from the previous stage. Two rendering stages happen for our program, the outline render loop, and the model render loop. These stages are shown visually in Figure 2.

The outline render loop draws only the back faces of the model in a black wire-frame. Edges are drawn at a size of 5 pixels instead of the default 1 pixel, so they stick out from the model. Furthermore, the polygons of the model are draw slightly offset to further accentuate the outlines. One issue with this approach to drawing outlines is that the size of the outlines is constant no matter the distance to the model, so far away objects end up with much more noticeable outlines than closer models.

The model render loop handles the lighting and the cel shading of the object, and the its work is done in the vertex and frament shaders. The steps are detailed below.

The vertex shader is the entry point for the OpenGL pipeline, so its main role is to forward the parameters that the fragment shader uses, and to specify the position of

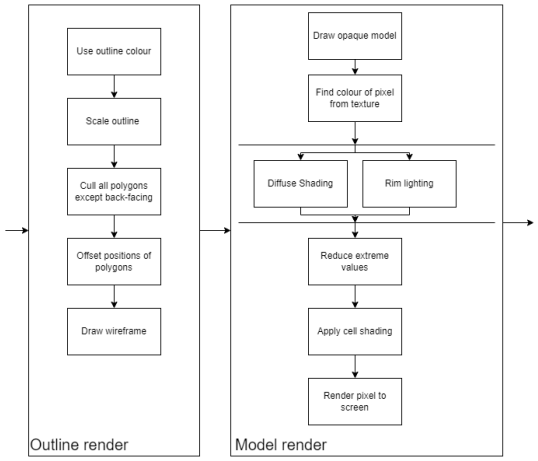

Figure 2: An overview of stages in the rendering step

each vertex sent to it. Additionally, the vertex shader calculates the lighting normal.

The fragment shader does the bulk of the work of our application. In the fragment shader, two main procedures happen. First, the colour of the lighting for the fragment is found. This is a multi-step process that involves applying diffuse lighting, rim lighting, and reducing extreme light and dark values in order to preserve the colour of the model. Afterward, cel shading is applied to discretize the colours of the model.

Figure 3 shows the rendering process at three stages, drawing the outline, drawing the inner model, and the combined final result. One drawback to this approach of drawing outlines is that the rendering pipeline is used twice per model. However, it may be not quite as intensive since only back faces are drawn, and the wireframe of the model is not filled.

(a) Outline render (b) Model render (c) Final render

Figure 3: Comparison of output at each rendering stage

Experimental results

Conclusions

Bibliography

[1] Angel, E. Shreiner, D. (2012). *Interactive Computer Graphics: A top-down approach using OpenGL*. Pearson.