COSC 4306

Final Report

Mathieu Carriere
Bradley McFadden
Joy McGibbon

April 4, 2022

Abstract

Non-photorealistic rendering (NPR) is a combination of computer graphics and artistic techniques used in a variety of applications to emphasize visual 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.

In this project, we investigate techniques used for NPR, and apply them to our own renderings of 3D models. In particular, we attempt to reproduce the style of 2D animation with 3D models by applying cel shading, lighting effects, sampling textures, contours, and suggestive contours. For our implementation, we use the OpenGL pipeline to apply a rendering process with several stages, in order to produce our final renderings of our models.

Introduction

In the field of computer graphics, non-photorealistic rendering (NPR) is a somewhat circular term that refers to the use of rendering techniques to achieve a non-photorealistic effect. There are several situations in which a non-photorealistic rendering is advantageous.

The first field where this is commonly applied is technical illustration, as described by Gooch et al[5]. Technical illustrations are commonly found in manuals for products that need to be disassembled for repairs, or in training documentation for a particular device. As such, the depictions of the device are static images, and it is often the goal of the illustrators to convey as much information as possible with these static images. Often, realistic shading is omitted, and image are drawn with an isometric perspective, so the scale and shape of geometry is easier to read. In addition, outlines are commonly added. If a light source is used, sometimes a technique called "rim lighting" is applied that highlights edges of faces that is pointed away from a light source.

Another field in which NPR is commonly used, is in interactive media. Games and animation that use NPR often apply it as a design choice. A realistic rendering may be seen as too visually busy, or does not provide the user with as much visual information as the designers wish. It may also be that designers wish to imitate a style closer to that of cartoons, for one reason or another. Cartoons are generally drawn in a wide variety of styles, but most have the following characteristics: Foreground objects such as characters are drawn with outlines. Most foreground objects are drawn with a single colour, and shading is shown with a darker version of this colour. This technique is referred to as cel shading, and is very common in many stylized modern games, including examples such as Sable.

The goal behind this project is to write a program that applies common NPR effects to 3D models, to create our own NPR rendering of the models. In particular, we will implement cel shading, object contours, and rim highlights to achieve our final NPR rendering of our models.

Prior and related work

Gooch et al.'s work in technical illustration demonstrates many common non-photorealistic rendering effects [5]. The authors made the observation that there was a disconnect between the style of artists and illustrators, and the style created by computer graphics programmers. Where computer graphics artists were concerned with techniques to create a realistic image, illustrators of technical manuals opt for a style that emphasizes the geometry of an object, while removing extraneous detail. The authors note that technical illustration needs this characteristic, as the viewer does not have the ability to move around a printed image to get more information. To produce a shaded technical rendering of an object, Gooch et al. render objects with black edge lines, no shadowing, intensities far from black or white, colour indicated by surface normal, and a single light source that provides white highlights (also known as rim lighting). It is interesting to note that the authors provide a method for approximating their model using Phong shading with negative light colours.

The work of Gooch is very similar to [11]. Saito and Takahashi also sought to create more clearly comprehensible renderings of 3D shapes, in particular renderings that maintained their clarity after multiples passes through a copy machine. Their work focuses on image processing techniques that occur after an existing rendering has been produced. The authors combine image enhancement techniques together to produce more clear images. They find edges, contours, discontinuities, and curved hatching together to produce images of 3D objects with enhanced edges, and more clear shape from curved hatching. Additionally, they demonstrate their results on topographical data, to produce easy to read topographical maps of terrain.

The work of DeCarlo et al. starts from a similar question, how can the shape of a 3D object be conveyed with just lines[4]? The research describes two techniques to display an object's shape, namely contours and suggestive contours. Contours describe an area of an image where a surface turns away from the viewer, becoming invisible. Similarly, suggestive contours are like contours, but describe areas where the surface of an object turns away from viewer, yet remains visible. Additionally, Decarlo et al. provide mathematical definitions and algorithms that detail how contours and suggestive contours can be produced on a model.

Decaudin et al. draw upon several techniques to render 3D scenes in a cartoon style [3]. Their work presents a rendering algorithm that proceeds in four stages. Firstly, they render the scene with ambient lighting. Next, they find the outlines of each object in the scene. For each light source, they render the scene as illuminated by it, and then find the project shadows from other objects. Finally, they combine the steps to produce a cartoon style image. Notably, their ambient lighting stage renders each object with a single uniform colour, and the shadows of the scene are also a single colour, which seems like a form of cel shading.

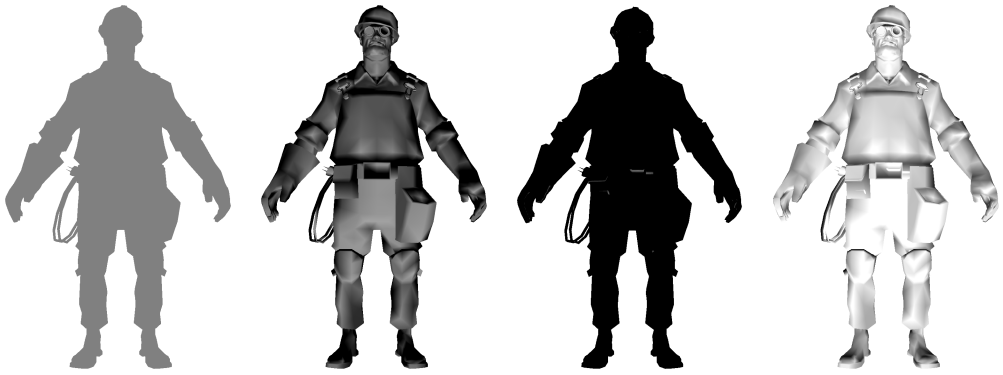
Work by Mitchell et al. in [8] describes how a modified Phong shading model can be used to increase visual clarity of rendered models in a first-person actions game like Team Fortress 2. One technique employed is a transformation on the dot product of vector n and

I in the diffuse lighting equation. The transformation prevents models from losing shape information on faces opposite a particular light source. The authors transform the Phong model to add a function that takes the scale produced by diffuse lighting, and breaks it into three regions, a dark gray end ground, a light gray start zone, and a middle ground with a slight red component. This is done from observations that artists tended to favour use light grays and dark grays instead of black and white, and preferred to mix in warm colours to their mid tones. The authors also add a dedicated rim lighting term to their Phong model in order to make upward facing surfaces more likely to be rim shaded.

To create images that are even closer to traditional hand drawn line renderings Al-Rousan et al used Laplacian smoothing to reduce rough and sharp edges of models, creating a simplified model to reduce details in a realistic way before using other algorithms to outline contours of the model [10]. Similarly Lee et al focussed on improving directional lighting through dividing objects into curved surface patches, then applying lighting effects to each surface patch independently to optimally highlight details and draw the viewer's eye [7]. In both cases simplification and modification of the initial model allowed for clearer renderings, but also required complicated pre-processing to the model.

Description

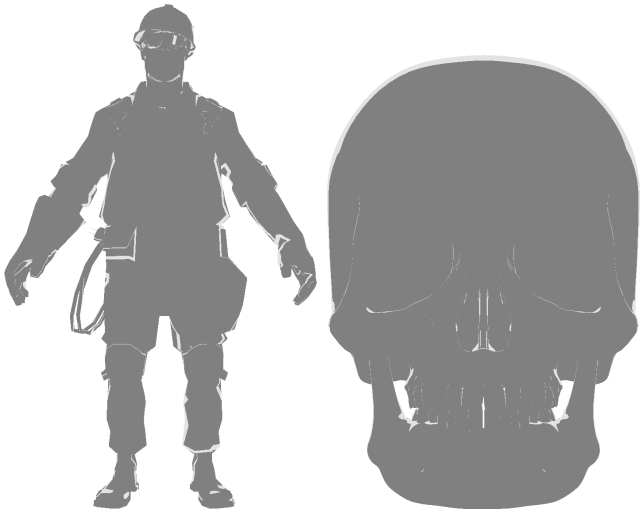
Lighting



(a) Ambient (b) Diffuse (c) Specular (d) All

Figure 1: Phong Lighting

The Phong lighting model is used, with specular, diffuse, and ambient light. Specular light concentrates light in one direction, providing highlights, diffuse light scatters light equally, providing shadows, and ambient light provides an equal amount of light to all parts of the surface from every direction.



(a) Engineer (b) Skull

Figure 2: Rim Lighting

Rim lighting is also added to provide additional highlights and increase the cartoon appearance of the render as specular highlights are not that effective. It highlights edges of the model perpendicular to the light within a range. This is likely as properties of the model's material is consistent throughout it's entire texture. It highlights the edge of the

object, and as a result improves the rendering from some angles, but appears incorrect from some models that do not have smooth curves, and is highly dependant on the distance of the light from the object.

```
d = dot(ptLightNorm, ptNorm);

// add Phong lighting
ambientPortion = ambientColour;
diffusePortion = diffuseColour * d;
halfway = normalize(ptLightNorm - ptNorm);
s = max(dot(ptNorm, halfway), 0.0) ^ 2;
specularPortion = specularColour  * s;

// add rim lighting
r = 1 - d;
if (r > 0.9)
    rimPortion = vec4(0.4, 0.4, 0.4, 1);

lighting = ambientPortion + diffusePortion
         + specularPortion + rimPortion;
```

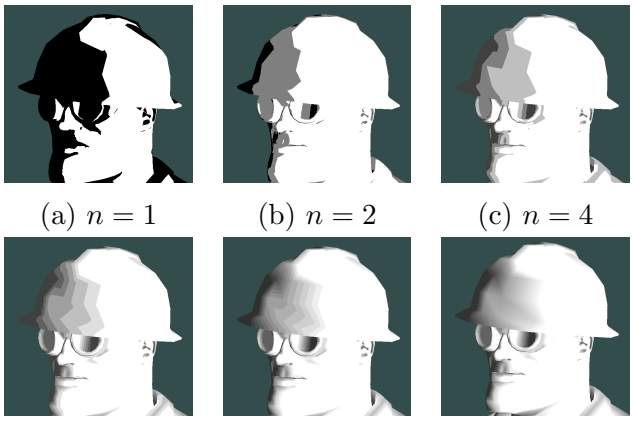
Lighting is implemented in the fragment shader, rather than in the vertex shader, decreasing efficiency as it must be calculated for every fragment rather than the relatively small number of vertices, but providing more precise results. Our implementation of Phong lighting is as described By Angel [1], and the model can be displayed with either a directional or point light source.

Cel shading

Cel shading refers to an NPR technique where to better simulate a cartoon art style, the tones of an image are broken down into bands. The colour palette of a scene is reduced to a handful of colours. The technique simulates how an artist might draw an image, where only a few shades are used, and few separate colours are used.

To perform cel shading, a number of discrete bands n must be chosen. Higher values for n produce more bands of colour, which leads to less harsh contrast in the image, but fewer bands create more contrast, and therefore a more dramatic effect. To get the final colour of a pixel, the RGB values are divided by $n + 1$, which returns the index of the band. Then, the index is multiplied by the size of the band, which produces the final colour. Examples of cel shading for a particular model are shown in Figure 3.

One can notice sharp edges on the model. This is implementation specific, and happens because the colour passed to the fragment shader is interpolated between vertices by OpenGL. Normally, this creates a smooth effect and saves processing power, but when per-

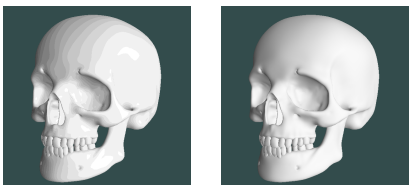


(a) $n = 1$ (b) $n = 2$ (c) $n = 4$

(d) $n = 8$ (e) $n = 16$ (f) None

Figure 3: Comparison of number of cel shading bands

forming cel shading, it can destroy the effect, so it is turned off. Unfortunately, doing so creates jagged lines in models with a low polygon count, but the effect is not as noticeable for higher polygon count models, such as in Figure 4.



(a) $n = 18$ (b) None

Figure 4: Cel shading with a high polygon count model

Outlines

This is done by drawing lines on all the outer edges of the model. So as to clearly break it up form the background. As well as to give it a more stylized look. It is a commonly used technique in none photorealstic rendering. Even more so when trying to emulate a hand draw style.

Suggestive contours

Suggestive Contours are a little more complex. Properly stated, " Suggestive contours are lines drawn on clearly visible parts of the surface, where a true contour would first appear with a minimal change in viewpoint." [2] Simplified, what this means is that suggestive contours are lines drawn on top of a model where the model is highly concave. So as the show deapth.

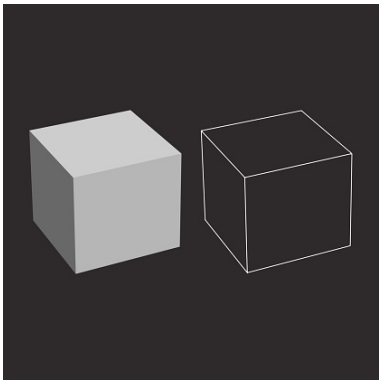


Figure 5: An overview of stages in the rendering step

This helps to break up the form of the model in a way similar to how one dose when drawing with pencil and paper. This technique if most comonly used in situations where the model is atempting to look like a darwing or was made with a limited color palette.

Figure 6: An overview of stages in the rendering step

Textures

(a) Engineer (b) 6 Levels (c) 8 Levels (d) Skull (e) 8 Levels

Figure 7: Cel Shaded Textures

Texture coordinates are loaded from the initial .obj file and stored with each vertex, and can then be used both to render the model with its original texture or to replace the texture with another image. In the fragment shader textures are used to map fragments to the color at the fragment's texture coordinate interpolated from the stored vertex coordinate. The color mapped by the coordinate in the model's textures can then be further modified through cel shading so photo realistic versions are rendered in a consistent toon style.

(a) Initial Texture (b) Skull (c) Engineer

Figure 8: Replaced Textures

Using a photo-realistic image transformed through cel shading as the new texture can produce interesting renders with the same process used for the original textures.

System overview

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 9.

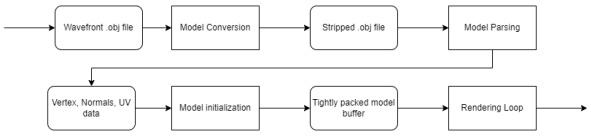


Figure 9: An overview of stages in the system

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 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 17.

The outline render loop draws only the back faces of the model in a black wireframe. 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 each

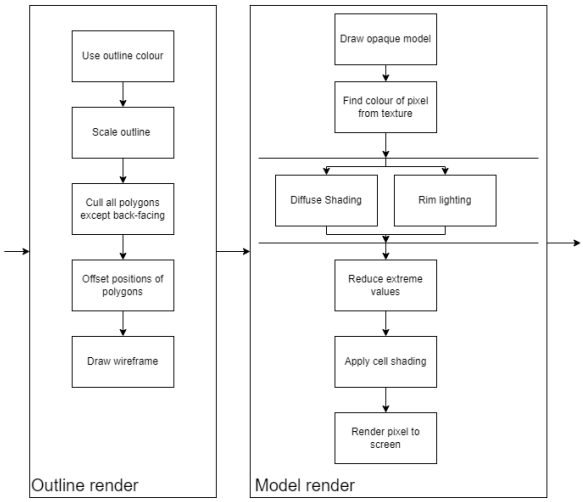


Figure 10: An overview of stages in the rendering step

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.

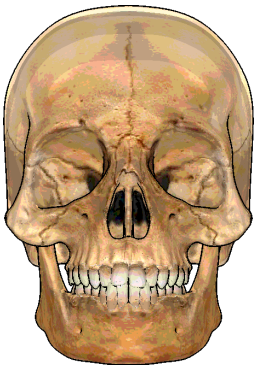
Figure 11 shows the rendering process at three stages, drawing the outline, drawing the inner model, and the combined final result. One drawback to this approach of drawing outlines is that the rendering pipeline is used twice per model. However, it may be not quite as intensive since only back faces are drawn, and the wireframe of the model is not filled.

(a) Outline (b) Unshaded (c) Shading (d) Final

Figure 11: Comparison of output at each rendering stage

Experimental results

Our system was able to produce the best results using models with higher vertex counts and smooth curves like the skull, while the engineer was more problematic, but increasing the vertex count also significantly increased the time to produce the render. Many aspects needed to be customized to particular models to get optimal results.

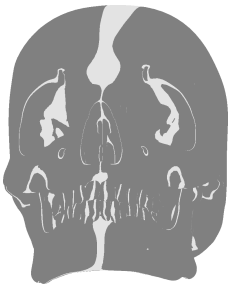


(a) Skull

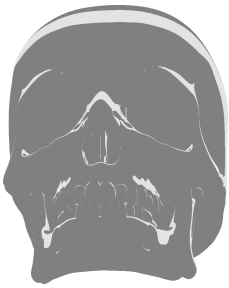
(b) Engineer

Figure 12: Final Renders

When our simple Cel Shading algorithm is applied to some textures the colours output need correction especially at lower levels as in Figure 13, but combined with cel shaded lighting using more levels for the texture the desired effect can be produced as shown in Figure 12b.



(a) Directional



(b) Point

Figure 13: Rim Lighting and Light Position

Positioning of the light for optimally rending the lighting is highly dependant on the model, and the area of the model that is being viewed. Depending on the positioning of the lighting the effect can vary greatly, demonstrated in Figure 13 where the rim lighting on the model appears dramatically different depending on the distance away it is.

As for adding in our contours and suggestive contours. We have cycled through three methods.

Our first implementation was only able to handle contours and was done using OpenGLs built - in PolygonMode system. More specifically, it was done by re-rendering the model in GL_Line mode. Before offsetting in the Z axis relative to the camera. Theoretically making it so that the only part visible would be the lines outlining the model.

Figure 14: An overview of stages in the rendering step

The problem with this is implementation, next to its simplicity, is that it produces a lot of artifacts. With this problem being most pronounced around the face of the model, which has the highest density of polygons. But they can be seen through the model. Flickering in and out of existence as the model is rotated.

Figure 15: An overview of stages in the rendering step

Seeing that this was not going to work, we then moved onto our second implementation using the stencil buffer. For contours, we set the stencil buffer to store the position of any pixel that was drawn before rendering the model to the screen. Before disabling writing to the buffer and inverting its data. Making it so that only areas outside that in which the model was drawn could be rendered to. Before once more rending the outline of the model.

Figure 16: An overview of stages in the rendering step

While this was happening, suggestive contours would be calculated inside of the shaders and stored in a matrix, to be passed into the stencil buffer for a third and final pass. But it was as we were going over this that we realized that since we were already doing calculations inside of the fragment shader, it would be easier to manually set the color of the fragment to be black.

Figure 17: An overview of stages in the rendering step

This leads into implementation three. For this implementation, we realized that it would be better to instead do all the work inside of the shaders. As most of the calculations would need to be done inside of them anyways. And by doing it this way, we would be able to generate the Contours in a single pass, instead of needing 3. While also being able to handle both Contours and Suggestive Contours in a near identical way to Contours.

For the actual implementation, we used the dot product of the normal vector of each face and the view angle. With any result of zero being a contour. While any none zero none negative result being compared to its sourdning faces. With any faces found to be significantly lower to be registerd as a Sujestive Contour. With Contours and Sujestive Contours being set to render as black once they reach the Fragment Shader.

Conclusions

Our implementation can produce a cartoon-style rendering from a variety of models and allows for customization, but when inspecting details of the model there are some obvious issues, especially with models that are not smooth.

The jagged lines produced by applying cel shading or applying rim lighting to models with lower vertex counts could potentially be fixed by increasing the vertex count to make the model smoother, but would make rendering the model much less efficient. Methods proposed in [10] to smooth the model vertices could also be used to improve rendering of models with rough edges.

Lighting, and rim lighting in particular, could be further improved by geometry dependent lighting methods described in [7]. This would enable automatic customization of the lighting to enhance particular aspects of the model and increase the overall quality of the image generated, and also give a more consistent appearance for rim lighting as the model would consist of smooth curves.

Cel shading textures could have been improved by defining a specific discrete range of colors to map the initial colours to, allowing for a larger variety of customization and more accurately coloured outputs. This is especially the case for models with textures that had colors that were not smoothly distributed.

The cartoon renderings produced from 3D models were not optimal in all cases, but models with smooth curves, colouring, and a large number of vertices produced more consistent and visually appealing results. Through further modification to the initial model the same simple techniques could be applied successfully to a larger variety of models.

Bibliography

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

[2] Baert, Jerone. Jeroen Baert's Blog, 3 Sept. 2010. *h*ttps://www.forceflow.be/thesis/thesis-code/. Suggestive Contours for Conveying Shape (princeton.edu)

[3] Decaudin, P. (1996). *Cartoon-looking rendering of 3D scenes*

[4] DeCarlo, D., Finkelstein, A., Rusinkiewicz, S., & Santella, A. (2003). *Suggestive contours for conveying shape*. In ACM SIGGRAPH 2003 Papers (pp. 848–855).

[5] Gooch, A., Gooch, B., Shirley, P., & Cohen, E. (1998). *A non-photorealistic lighting model for automatic technical illustration*. Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques, 447–452.

[6] Graf, C. (2002). *NPR-Shading*.

[7] Lee, CH., Hao, X., Varshney, A. (2006). *Geometry-dependent lighting*. IEEE Trans Vis Comput Graph. vol 12(2), 197-207.

[8] Mitchell, J., Francke, M., & Eng, D. (2007). *Illustrative rendering in Team Fortress 2*. Proceedings of the 5th International Symposium on Non-Photorealistic Animation and Rendering, 71–76.

[9] Phong, B. T. (1975). *Illumination for computer generated pictures*. Communications of the ACM, 18(6), 311–317.

[10] Al-Rousan, R., Sunar, M., Kolivand, H. (2016). *Interactive toon shading using mesh smoothing*. Int. J. Intelligent Systems Technologies and Applications. vol 15, 218-229.

[11] Saito, T., & Takahashi, T. (1990). *Comprehensible rendering of 3-D shapes*. Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques, 197–206. https://doi.org/10.1145/97879.97901