Comic book shader implementation

Alexandre Flores
pg50165@alunos.uminho.pt
Pedro Alves
a93272@alunos.uminho.pt
Rui Armada
pg50737@alunos.uminho.pt

***Abstract*—Stylistic, non-photo-realistic shaders are often used in media to automatically convey varied art styles without having to individually model every aspect of a scene within said art style. A very common example of this kind of shader usage are cartoon like shaders, which may render models with a more limited color palette among other stylistic effects. In this document, we describe our process for implementing a comic-book like shader and showcase our results step-by-step.**

***Index Terms*—lightning, shading, real time rendering, deferred shading, non-photo-realistic rendering, comic book art style**

I. Introduction

With the recent mass resurgence of comic book related media, stylistic 3D rendering is more relevant than ever. Big animation studios like Sony Pictures Animation and even more recently Disney are recognizing the value of creative and unique looking animation, different from the standard style usually seen in 3D animated movies. As such, the development of stylist non-photo-realistic shaders has gained a lot of relevance. The video game industry has also seen a increase in the value of stylistic graphics; stylized graphics tend to age better than a purely realistic art style as technology develops. In addition, heavily stylized visuals may serve to obscure technological limitations by utilizing simplistic textures and models that will still look good under a carefully developed shader.

In this document, the process of development of a stylized comic-book like shader is detailed. The goal was to replicate the appearance of class comics by implementing four specific elements of the art style:

- Line shading
- Two-tone shading
- Dithering
- Outlines

Fig. 1. Sample comic panel with line shading, two-tone shading, and outlines

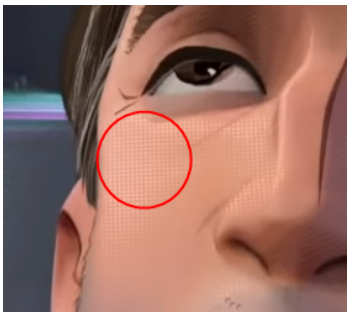

Fig. 2. Example of dithering for shading

II. Outlines

A. Implementation

Perhaps the most complex aspect of this shader, outlines require the implementation of position differentiation in the fragment shader. The implemented algorithm is an adaptation of the algorithm described in [1], with differences made for the sake of simplicity or stylization.

The first step to implementing this shader is the implementation of an information gathering shader that will map each fragment's color to its position in view space. The resulting render, which is referred to as **positionTexture**, will be used for differentiation in the main shader.

Now in the main shader, we implement differentiation as a cycle that will iterate through a square of fragment around the current fragment. In each iteration, the view space position of the current fragment is compared to the view space position of a neighboring fragment. The size of the neighboring "square" around the current fragment which is check determines the width of the outline. Throughout this cycle, the maximum

distance found between two fragments is updated. Afterwards, this maximum distance is utilized in a smoothstep which will get a position in the curve between the minimum outline defining distance and the maximum outline defining distance. If the calculated distance is below the curve's minimum value, no outline is drawn at this fragment. Otherwise, the *smoothstep* value is used as the alpha value for the outline's color. This means that bigger distances will result is harsher outlines while smaller distances will blend the outline's color with what would otherwise be the color of the fragment.

Fig. 3. Resulting outlines

The example render in Fig. 3 uses parameters that result in harsher, less varied outlines. Namely, the minimum and maximum curve values are very close together, resulting in a much smaller transition from the harshest outline to no outline. This transition is visible in the top and bottom of the teapot's handle.

Finally, an outline distortion algorithm was implemented to simulate an inky/wavy outline. A Perlin noise texture is first generated. In the fragment shader, a color vector is obtained from this texture using the fragment coordinates as converted texture coordinates. The red and blue color values are utilized as distortion vector coordinates. When positions are consulted in the **positionTexture**, the distortion vector is added to the fragment coordinates of the current fragment and the neighboring fragment. The effect is a continuous, varying distortion across the outline.

Fig. 4. Resulting distorted outlines

B. Alternatives

One detriment to this implementation is the shape of the neighboring area that is checked during differentiation. The fact that a square is checked around the fragment (rather than a circle) coupled with the lack of antialiasing makes the outlines looks jagged.

Additionally, since every fragment is treated the same independently of the distance to the camera's position, outlines will look messy and overpopulate the resulting image in large/complex scenes. To remedy this, one might use the a depth buffer and vary the neighboring fragment interval according to the depth. This would result in thinner outlines for objects far away from the camera, and larger outlines for objects close to it. The algorithm described in [1] utilizes a similar strategy to produce these dynamic outlines.

Also in [1]'s algorithm, the outline color is not a static color, but rather a darker variant of the current diffuse color. This approach is perfectly adequate, but we found that a static outline would be more appropriate to simulate a grittier art style.

As for the distorted outlines, the dependence on fragment coordinates makes the effect warp and contort as the camera moves around. This can prove disorienting and inadequate for real time rendering scenarios. To alleviate this, one might use a different method for obtaining a color vector from the noise texture.

III. Two-tone and line shading

A. Implementation

Two-tone shading is a rather simple form of lighting shading that simply clamps color values to two possible outputs. For this implementation, the two possible colors (exclusive outline color) for any given fragment are the diffuse colour and black. In order to implement this, line shading was utilized to color any sufficiently dark fragments in black in a striped pattern.

First, the diffuse lighting intensity is calculated using Phong's lighting model. If the intensity is found to be above a certain threshold, the calculated colour for this step is simply the diffuse colour of the current fragment. If the intensity is below or equal to said threshold, the line shading algorithm is used instead.

The line shading algorithm begins by calculating the line's width at the current fragment. The intent is that as a model gets darker, the lines painted on it should grow thicker, until they eventually completely dominate the model and paint it completely black. The line width is the calculated by performing a smoothstep between 1 and 0.1 of the division between the light intensity and the darkness threshold. This value is then elevated to a power of 1/x, where x is a user defined line growth factor which controls how fast the lines change width within the darkness.

With the line width now calculated, the line shading algorithm utilizes the texture coordinates of the model to check if the current fragment would fit within a line. The texture coordinates are first transformed into a vector that repeats

itself n times across the model. The value of n controls the line density on the model. The x and y coordinates are then subtracted (in the actual implementation the equation is slightly altered to flip the diagonal) to see their difference - a perfect diagonal occurs when they are the same. Finally, this difference is checked against the line width. Larger line widths are more lenient to bigger deviations from the perfect diagonal, and as such result in larger lines and darker shading. If the difference value places this fragment outside of a line, then the diffuse color is used.

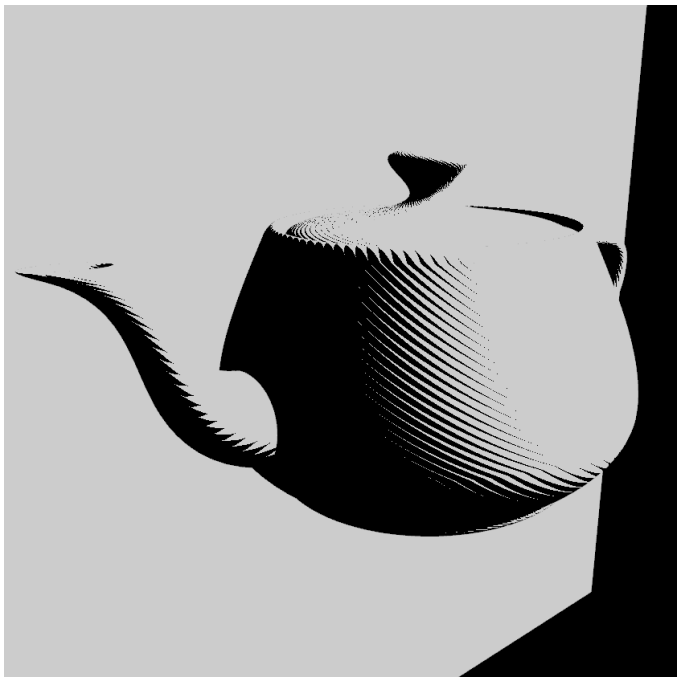

Fig. 5. Resulting line shading

B. Alternatives

Once again, the lack of antialiasing hurts this effect. As the model grows lighter and lines get thinner, it's hard not to single out individual fragments, which harms the effectiveness of the shader.

Additionally, since texture coordinates are used as the basis for calculating whether a fragment belongs in a line or not, complex scenes of varying object scales can become messy looking. A smaller object placed next to a large object can create an undesirable difference between the line widths and line density of the two. This effect may also be used to stylistic effect, as it is not uncommon to shade different objects with varied lines in comics.

Finally, while this shader utilizes black as its darkness color, it is perfectly viable and even adequate to instead use a darker shade of the diffuse color. The black color was simply chosen in order to simulate a grittier look.

IV. Dithering

A. Implementation

Dithering is an staple of comic books where two colours are alternated (usually in a dotted pattern, with dots as one color and the background as another) in order to create the illusion of color blending. In this shader, dithering is used in order to showcase specular lighting with an algorithm similar to the one used for line shading.

The dithering algorithm only applies in areas of the model not below the darkness threshold. In these areas, the specular light intensity is calculated utilizing Phong's lighting model. The texture coordinates for this fragment are then converted in the same way as line shading (with, generally, far more repetition for a better effect). The converted coordinates are compared to the coordinates of the center of a circle within the repeating pattern, and the distance that is obtained is compared to a user defined radius (for simplicity's sake, this parameter is shared with the line growth factor for line shading). If the fragment is found to be withing a dot's radius, it is coloured with the specular color. Otherwise, the diffuse color is used.

Fig. 6. Dithered specular lighting (with the other effects enabled)

B. Alternatives

The only major issue with this implementation (besides the usual lack of antialiasing) is how the dots get cut off where specular intensity drops to zero. In order to combat this, we could use a process similar to the one used in the fragment differentiation for outlining. A quad would first need to be generated which has the specular intensity of each fragment. In the main shader, instead of only checking if the current fragment is within a "lit" circle, the fragments within the same circle as this one would also be checked. If any of them are

found to have specular light intensity, this fragment would ”light” as well. That way, if the current fragment belongs to a specularly lit circle it will always ”light”, even if it is outside of the specularly lit area of the model.

V. Conclusion and Improvements

We are satisfied with the developed shader and think it has adequately reached the goal of simulating a comic book art style. The shader is not perfect for every scenario and is ideal for simple renders of individual models. The parametrization of the used algorithms would allow for artists to experiment and find ideal conditions for a given model. The main improvements to this algorithm would be to further fine tuning to these effects, such as the mentioned dynamic outline widths and smarter specular illumination, as well as obviously antialiasing and performance. While this shader performs fine (albeit under limited testing conditions), there are optimizations to be exploited in, for example, differentiation (using a calculated position from a depth buffer instead of using a position quad).

Fig. 7. Spider-man model [2] with our shader after parameter adjusting

References

[1] D. Lettier, “Outlining — 3d game shaders for beginners.” [Online]. Available: https://lettier.github.io/3d-game-shaders-for-beginners/outlining.html

[2] B. Foster, *Spider-man obj*. Spider-man - Download Free 3D Model By Bobby Foster (@littleshaolin) [db660fd]. [Online]. Available: https://sketchfab.com/3d-models/spider-man-db660fd9c2474577b857973c4240cf2c