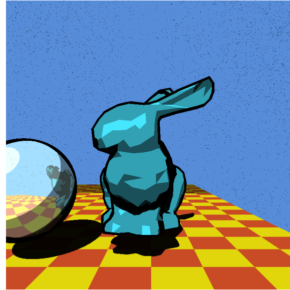
Toon Shading Using Distributed Raytracing

Amy Burnett, Toshi Piazza



Abstract

In this paper, we describe a method of producing ”toon” shading effects on a raytraced scene. Toon shading is a common non-photorealistic rendering technique that produces seemingly 2d cartoon drawings from a 3d scene. We demonstrate an implementation of toon shading which makes use of distributed ray tracing, as opposed to traditional toon shading implementations.

In particular, this implementation better captures global effects as shading and edges, where other toon shading algorithms fall short, especially with respect to reflection.

1 Introduction

1.1 Toon Shading

”Toon” or cell shading describes a general rendering technique which currently drives many of the non-photorealistic effects found in video games and popular cartoons. It mimics an older style of hand-drawn cartoons watermarked by thick borders and discretized color palettes. One such example can be found in the popular video game ”Borderlands,” whose effect was generated by hand; this can be considered the ideal in toon shading.

In general, cell shading often culminates in two distinct features, which we strive to reproduce: the physical cell shading and color discretization, and edge detection. The former, color discretization is often implemented in a GLSL shader or other equivalent GPU kernel program, and operates under the principle that colors should be weighted against the intensity of light that hits that particular point on a surface. One may simply compute the intensity of light I on a point with the following:

$$I = NL$$

where N represents the normal of the plane on which the point resides, and L denotes the ray from the point to the light. There is a common caveat here, however; for a mesh completely comprised of triangles, it is necessary to compute the phong-interpolation of the vertex normals along the polygonal face on which the point resides. Once the intensity has been computed, the accepted pixel colour should be weighted against discretized intensity thresholds, for example 95%, 50%, 30% and 5%.

Edge detection also has a common solution; the edges are often computed in image space, after the entire scene has been rendered. More concretely, the OpenGL Depth Buffer is reused and fed into an instance of the Sobel operator, and the results of this operator are interpolated with the original cell shaded image (Gao, Lei, Xiaoguang, Huizhong).

Unfortunately, there exists no commonly accepted solution to shadows and other global effects, such as reflection for toon shading. Because toon shading is often implemented for its local effects via GLSL shaders, there is no way to compute shadows without some sort of shadow approximation, such as shadow volumes or maps. There also exists no commonly accepted method of handling reflections and glass, and indeed very few non-photorealistic renderings which make use of toon shading also support reflective objects in scenes.

1.2 Ray Tracing

Ray tracing, as described by the seminal text *An Improved Illumination Model for Shaded Display*, makes use of a physical interpretation of the world to posit a rendering implementation which is especially robust in the way of global effects such as shadows and reflection.

In order to accurately and physically simulate light, the ray tracing technique involves a per-pixel recursive algorithm. In this algorithm, a ray of light is shot out from the immediate view of the camera, and is intersected with all objects in the scene (modulo implementation-specific optimizations, such as acceleration data structures). The fraction of diffuse, reflective and refractive color is computed, recursively by shooting another ray if need be for the recursive and refractive cases.

Shadows can be computed as well by simply performing occlusion testing, *using the same ray tracing implementation* to determine if any light source illuminates a particular point on the object. This is performed in a Monte Carlo fashion, in that many rays are cast around a distribution, to achieve a desirable soft shadow effect as in *Distributed Ray Tracing*.

Finally, Monte Carlo can simulate other effects such as aliasing, depth of field and motion blur, all of which are solved by shooting multiple rays per call site, which comes at a great runtime cost but for the sake of much more accurate renderings.

2 Cell Shading

We propose multiple solutions to the cell shading problem which are compatible with the ray tracing paradigm. In particular, we explore in depth two possible implementations, namely a color-based

discretization technique, and an intensity-based color discretization technique.

We first discuss a possible solution to the cell shading problem, which involves inferring the intensity of light on a pixel by using the RGB value itself. The possible motivations for this include simpler and potentially cheaper computation, as this discretization only occurs once per pixel as opposed to once per ray. Unfortunately, this technique produces visually worse images, as are thoroughly explained in the Implementation section.

A more successful solution to the cell shading problem involves the traditional intensity calculation, applied recursively to fit the general Whitted ray tracing algorithm. This algorithm describes a method for computing the intensity simultaneously with the ray tracing result, i.e. alongside with the Phong shading calculation; we then discretize this physically correct result with the computed intensity to output the final toon-shaded result per recursive ray iteration. We again discuss the ramifications of this in the Implementation section.

Finally, in order for the ray tracing to compute a much smoother model given an arbitrary and rough triangular mesh, it becomes completely necessary to perform phong normal interpolation along the face of the triangle; this causes the image to become much smoother. However, we implement instead a very close approximation to a phong shaded model, by ignoring the intensity when calculating the indiscreet (pre-toon shaded) color of a pixel as hit by an object. The ramifications of this approximation can be seen in the Results section.

3 Edge Detection

3.1 Distributed Raytracing Edge Detection

We performed edge detection by creating a distribution of rays on a normal diffuse ray hit. These rays are cast parallel to the original ray from distance d from the surface. Initially this distribution was uniformly generated at random over the disc radius r orthogonal to the ray at the hit point. The length of these rays would then map a depth map over this radius of the model. Several heuristics can then be applied to this map to determine if the area contained an edge of the model.

If a heuristic decides an edge is within the radius, the algorithm will set the result color of the raytrace to a solid edge color. This also allows for solid edges around reflective or refractive materials. The size of r determines how large the outline will be, since a larger radius will trigger on points further from the edge. d should be small enough to allow rays to be cast with length larger than ϵ_{hit} but small enough not to intersect any other models when casting.

3.2 Edge Heuristics

The simplest heuristic we applied checked if any of the rays missed the model. This works well for models with no edges, such as a sphere. For models with edges, this functions as an outlining heuristic. This may be useful in some applications, but was not the goal of toon shading.

To capture non-outline edges, the actual depth of the rays needed to be taken into account. We attempted the naive method for checking for any distances above a threshold ϵ_{depth}. This would locate these internal edges, but is susceptible to inaccuracies with sloping faces. We attempted to correct this by adjusting the ϵ_{depth} value. If to small an ϵ_{depth} is used, the majority of points on the slope will be falsely detected. Too large and ϵ and the real edges will not be found. Figure 1 shows a sphere rendered using this method. The line around the sphere varies slightly depending on the slope of the sphere compared to the angle of the ray cast, notably at the bottom left of the sphere. Note the highly sloped ground also suffers from false positives along the back end.

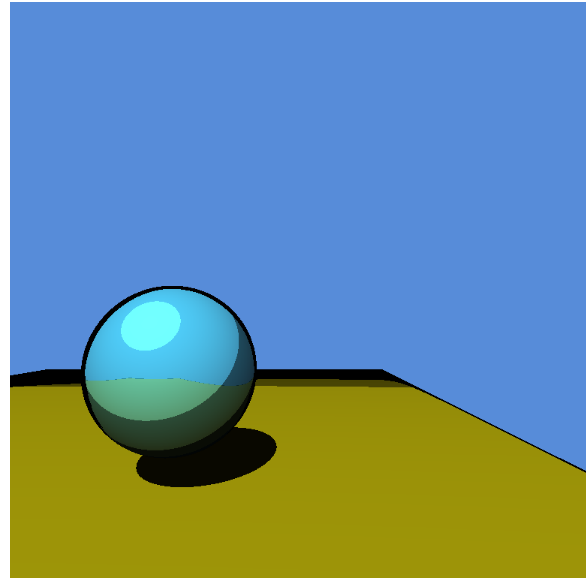


Figure 1: *Sphere rendered with ϵ_{depth} depth testing. The edge changes as the angle of the slope compared to the ray changes.*

3.3 Slope Edge Heuristic

To correctly determine edges on sloped faces, we used a heuristic based on the angle of the slope itself. The slope of the faces in the radius of the disk is calculated by finding the slope between adjacent points in the depth map. These data points are functionally the first derivative of the surface we mapped over. To determine if there is an edge within this map we looked for changes in slope larger than ϵ_{slope}. These were calculated by finding change in slope between adjacent points, similar to depth. Large second derivative values were used as indicators that the radius contained an edge.

Care should be taken when adjusting ϵ_{slope}. Curved faces are most susceptible, since they may have sharper curves, which may need several derivations before having a good indicator of an edge. Setting ϵ_{slope} determines the cut off for these curves.

Figure 2 shows examples of running this procedure against several different surfaces. The graphs show the depth (dotted line) and slope (solid line). The red sections indicate parts of the curve that the algorithm has determined to be edges.

4 Implementation

The NVIDIA OptiX-based programming framework was used in the implementations of the following toon shading attempts. The OptiX framework specifically aids in the rapid prototyping of real-time GPU programming, particularly in the field of ray tracing; by giving the user accessible buffers and an API capable of attaching callbacks to common ray-tracing operations, we were able to make use of the full optimizations available to NVIDIA cards, as are documented in *OptiX: A General Purpose Ray Tracing Engine*. We rendered our scenes with an NVIDIA GTX 1070 graphics card.

In particular, for our demos we constructed a general purpose shader program which computes the toon shaded effect, as well as

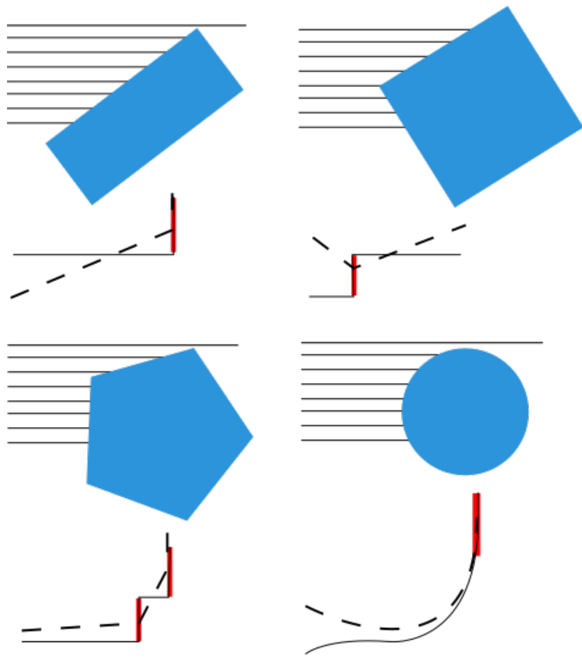


Figure 2: *Examples of casting rays against different edges. Dotted line is depth, Solid line is slope. Red marks large slope changes marked as edges.*

another shader which solely computes the distributed edges; these shaders became the *main* focus of our papers, and were used in conjunction with a plethora of other shaders which were developed and installed to intersect arbitrary meshes, in tandem with spheres and other primitives.

In our color-based discretization scheme, we initially claimed that the intensity of light could be inferred by the result passed down by the ray tracing; we could then perform some discretization on color as opposed to on intensity. In order to effect this implementation, we converted from RGB to LAB space, from which we could then infer intensity.

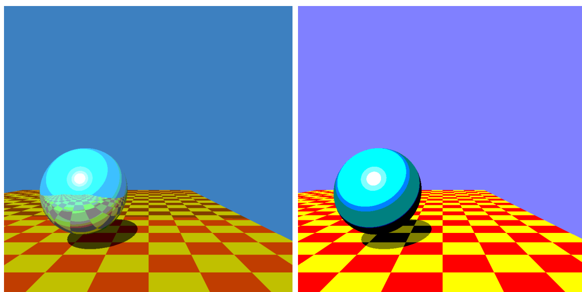


Figure 3: *Depicted above is a ray tracing sample of a diffuse and reflective sphere, using the RGBA to LAB conversion. As we can see, neither image looks particularly convincing, and are not good approximations to the problem*

As we see in figure 3, the RGB to LAB conversions are not particularly convincing, as the bands of discretized color are not distinct enough and are far too close together. As a result, we reject this implementation in favor of a more traditional approach, where we compute the discretized result of the ray tracing for a single ray. This is computed at the point of contact, when the normal and the light sources are all known; the algorithm proceeds to compute the intensity, and discretize the result of traditional Phong shading based on these intensity values. The immediate results of this computation can be seen in Figure 4.

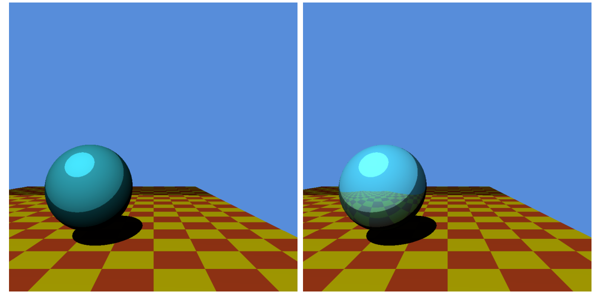


Figure 4: *Here we see the final form of the shading code, which is robust to both reflections and diffusion, as we see in both figures.*

The distributed raytracing edge detection algorithm was implemented in the recursive raytracing pipeline. After a diffuse color hit on selected models, the edge rays would be cast and slopes calculated. If an edge was detected, the trace was cut short, and the black edge color returned. Since this was done for every diffuse hit, a larger number of rays would increase render time. We ended up using 100 rays across a disc with radius of 0.02.

5 Results

The renderings we have performed look plausible, and are charming for non-photorealistic renderings. The discretization of the meshes are not jarring, and are very smooth even in the face of a rough triangular mesh, due to the interpolation mentioned earlier in the paper, as in Figure 5.

Similarly, we achieved all the goals we sought over traditional ray tracing; we were able to achieve toon style renderings even in the face of reflective test scenes, with even the outlines themselves visible through these reflections (Figure 6). This creates an effect that does not jar the eye, and more importantly looks very natural in the context of the rest of the rendering, even in renderings with low-count polygons (see Figure 7).

Finally, all images support stellar shadows, a global effect with no common solution for toon shading when applied as a local effect (i.e., in a GLSL shader). Because we implement a full ray tracer, the shadows occur naturally, and integrates fully and naturally with the toon shading algorithm.

6 Implementation Problems & Future Work

While implementing these algorithms, we ran into a few small problems. A majority of these problems were not caused by our algorithms, but rather by time constraints while using Optix for the first time.

When attempting to render refractive glass models with optix, we had problems reimplementing code that originally functioned correctly. This produced invalid ray casts and blotchy results, as seen in the right of Figure 8. From the parts that traced correctly, we can see that our toon shading was refracted correctly. With a correct Optix glass implementation, this problem would be resolved.

When loading custom objects as triangular meshes, each triangle was loaded as an individual geometry group. This causes rays to attempt to intersect all triangles, instead of just plausible ones. When detecting edges, this causes a slowdown of $O(n*m)$ where n is the

Figure 5: *Here we see shading and edge detection applied to the bunny model. Although this is a low-res bunny, notice the consistent shading across most of the bunny, a direct result of the interpolation we perform.*

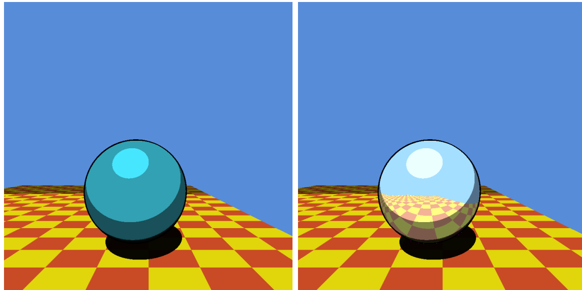


Figure 6: *Depicted here are different renderings of the same sphere; one with a diffuse sphere (left), and one with a reflective sphere (right).*

number of faces and m is the number of rays. This made rendering of higher poly mesh take a enormous amount of time. Correctly importing triangles would resolve this, though OptiX does not interface any high level utilities to do this; instead, one must manually construct a shader which understands the format of a single geometry unit with many primitives, and much construct an appropriate bounding box and intersection algorithm, which is non-trivial for arbitrary meshes.

When applying the the edge detection to a face almost parallel to the ray, a very small radius is needed to prevent rays from missing. This is a problem for large faces that are a long way from the ray source, as seen in the left of Figure 8 (which was rendered with a larger radius). The ground further away had a linear slope below ϵ_{slope}, but had rays that missed the plane due to angle and radius. Some sort of application of the face's normal may be used to resolve this problem.

Finally, the black artifacts on many of the triangular mesh images are puzzling; probably due to some deficient ϵ tolerance in the bounding box calculation of the triangles, we see that the ray tracer often returns black, as if nothing was hit. It is currently unknown

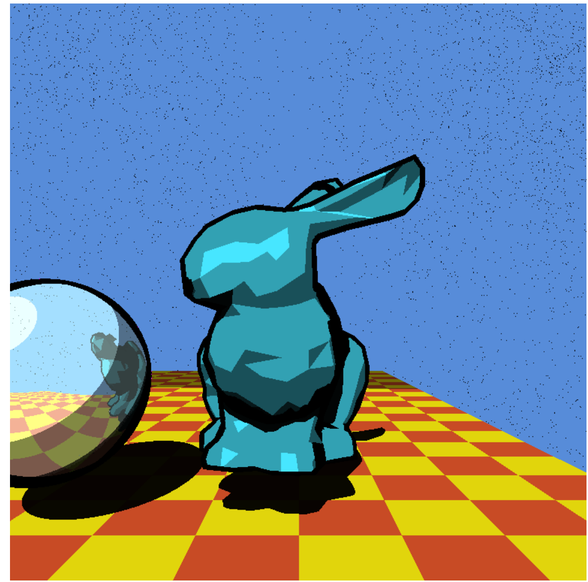


Figure 7: *Reflecting toon bunny on a metal sphere using a thicker outline. The outline is also visible in the reflection.*

why this problem persists for the triangular meshes, but is not a problem for the other primitives.

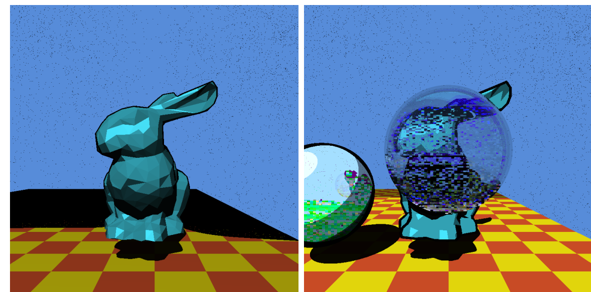


Figure 8: *Here we see two bloopers in the rendering process. Specifically, on the left we see that the surface is almost parallel to the rays causing over-detection. On the right we see OptiX mishandling glass refraction, i.e. by producing a deficient ϵ*

7 Work Distribution

Amy created and implemented the algorithms for the distributed edge detection, as well as set up the rendering build system. Toshi implemented the discretization of the colors and lights, and write the Optix object and scene import code.

References

Whitted, Turner. *An Improved Illumination Model for Shaded Display*. Publication. Vol. 26. N.p.: n.p., n.d. Print. Communications of the ACM.

Cook, Robert L., Thomas Porter, and Loren Carpenter. *Distributed Ray Tracing*. Rep. 3rd ed. Vol. 18. N.p.: n.p., n.d. Print. Computer Graphic.

Parker, Steven G., James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan

McGuire, Keith Morley, Austin Robison, Martin Stich, NVIDIA, and Williams College. OptiX: A General Purpose Ray Tracing Engine. Rep. N.p.: n.p., n.d. Print.

Gao, Wenshuo, Lei Yang, Xiaoguang Zhang, and Huizhong Liu. *An Improved Sobel Edge Detection*. Publication. N.p.: n.p., n.d. Print.