Ray Marching Distance Fields in Real-time on WebGL

Prutsdom Jiarathanakul*
University of Pennsylvania

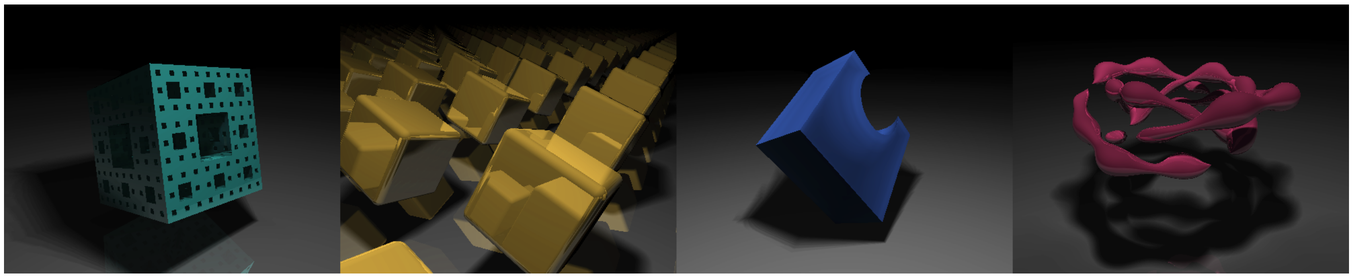

Figure 1: *Ray Marching Distance Fields*

Abstract

Ray marching, also known as sphere tracing, is an efficient empirical method for rendering implicit surfaces using distance fields. The method marches along the ray with step lengths, provided by the distance field, that are guaranteed not to penetrate the scene. As a result, it provides an efficient method of rendering implicit surfaces, such as constructive solid geometry, recursive shapes, and fractals, as well as producing cheap empirical visual effects, such as ambient occlusion, subsurface scattering, and soft shadows.

The goal of this project is to bring interactive ray marching to the web platform. The project will focus on the robustness of the render itself. It should run with reasonable performance in real-time and provide an interface where the user can interactively change the viewing angle and modify rendering options. It is also expected to run on the latest WebGL supported browser, on any machine.

CR Categories: I.3.3 [Computer Graphics]: Three-Dimensional Graphics and Realism—Display Algorithms

Keywords: distance fields, sphere tracing, ray marching, real-time, WebGL, GLSL, Three.js

Links: DL PDF WEB DATA CODE

1 Introduction

1.1 Motivation

One type of surface modeling is implicit surfaces. They can produce interesting mathematically-based shapes that are not easily reproduced through other modeling methods such as the popular polygonal meshes. Implicit surface functions can become complicated, in cases such as recursive surfaces and fractals, and therefore a method to efficiently visualize them is needed.

1.2 Overview

Ray marching, or sphere tracing in technical literature, is a fast, approximated method of directly visualizing implicit surfaces. The method is similar to ray tracing, in which for each pixel of the output, a ray is cast from the eye through the pixel into the scene. Instead of checking for a mathematical intersection of the ray with the scene, ray marching is an empirical method which takes steps along the ray until it hits the scene. It does this with the use of a distance field, or a distance estimator. Composed of one or more distance functions, the distance field is a function that describes the shortest distance from any given point to the surface of the scene. Since this distance is equivalent to the maximum distance the ray can step without intersecting any surface in the scene, the algorithm uses this distance to march along the ray towards the scene, and is guaranteed not to penetrate the surface of the scene. The process can be thought of as computing an ”unbounding volume” which is a sphere that is guaranteed not to intersect the scene. [Hart et al. 1989] See Figure 2.[1] The algorithm then repeats the process of sampling the distance field and stepping until it cannot take any more steps. This adaptive step size is fundamental to the efficiency of this method.

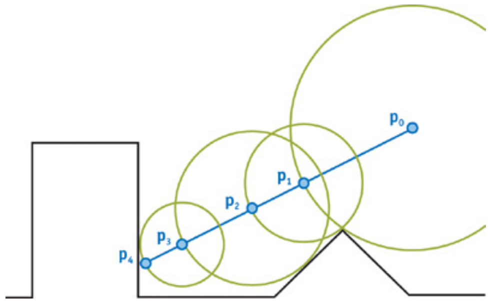

Figure 2: *Ray marching method.*

Moreover, because of the algorithm's empirical repeated process of sampling and stepping, ray marching can be extended to em-

*website: `http://www.iamnop.com/`

[1] http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter08.html

pirically produce other visual effects inexpensively. These include ambient occlusion, soft shadows, glow, and subsurface scattering.

Evidently, the fundamental limitation of this algorithm is that it is an approximated method, and as with such methods, it suffers from numerical errors. These numerical errors take the form of fuzzy glows, banding patterns, and image distortions. To eliminate these errors, the algorithm can simply take more steps to sample the scene, which will inevitably decrease efficiency. Therefore, it is important to balance between quality and performance.

2 Related Work

The method of sphere tracing was first proposed by John Hart in 1989 as a means to efficiently render 3-D fractals. [Hart et al. 1989] Afterward, he found that the method proved to be useful and published another paper in 1994 focusing on the new technique. [Hart 1994]

Despite the qualities of the ray marching technique, it took another ten years before it gained popularity. In the recent years, the technique has become popularized in the demoscene circles, due to its efficiency and its ability to produce complicated abstract shapes and infinite scenes. The most prominent figure is Iñigo Quilez who initially brought this technique to the demoscene field with his work in 4k demos. [Quilez 2008b] Through these demos, he popularized a technique where the whole scene is rendered through ray marching on a fragment shader and simply displayed on a viewing quad. [Quilez 2008c] Today, with the rising adoption of WebGL, there are many implementations of this technique online using GLSL fragment shaders.

There have also been various GPU implementations, for instance, ray tracing of Julia Quaternion Sets on GPU by CalTech in 2004. [Crane 2004]

3 Method

3.1 Framework Technology

The project is implemented on Javascript using the WebGL graphics library. WebGL is hardware-accelerated and the shader is written in GLSL using the ES 1.0 specification. In addition, the project uses Three.js, a third-party Javascript library useful for building 3D applications on WebGL.

3.2 Pipeline Architecture

On the CPU side, a viewing scene is constructed, consisting of an orthogonal camera and a single quad which acts as a screen to display rendered results. In addition, there is a Three.js Trackball Camera that is controlled by the user. All the rendering computation is done on the GPU through a fragment shader that is attached to that viewing quad. Necessary data such as the camera and scene options are passed down from the CPU as shader uniform variables.

3.3 Ray Marching

The computation of output color is done per pixel. For each visible pixel, a ray direction is determined by casting a ray from the camera, through the pixel, into the scene. The renderer first samples the distance function and then takes a step along the ray direction equal to that distance. The process is repeated until the sampled distance is below a certain small epsilon threshold, meaning that the ray has come sufficiently close to the surface of the scene, or until the maximum number of steps is reached. This loop structure benefits performance since it allows for early ray termination when the ray has hit the surface. When a large enough region of pixels terminates the loop early, coherent warp termination kicks in and there is a visible performance gain.

3.3.1 Smoothing

Smoothing is implemented to eliminate numerical errors that cause banding distortions. This is done by scaling the length of each step taken by a certain coefficient, so not the whole length of the estimated distance would be taken. The smoothing coefficient is set between 0.0 and 1.0, 1.0 being no smoothing at all. Setting this value is a balance between quality and performance. Smoothing eliminates the rippling distortion, but decreases performance since more steps are needed to be taken in order to reach the scene and produce the same quality image. A value I empirically settled at was 0.7 which provided good results while maintaining performance.

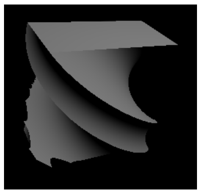

Figure 3: *Rippling distortion effect at lower left corner*

3.3.2 Rendering Modes

Three modes of rendering output are implemented. The first mode outputs the resulting computed color value for each pixel. The second mode outputs a depth map, rendering each pixel value proportionally to the distance of the camera to the scene. The last mode outputs each pixel value as the number of steps that the ray had to march in order to reach the scene.

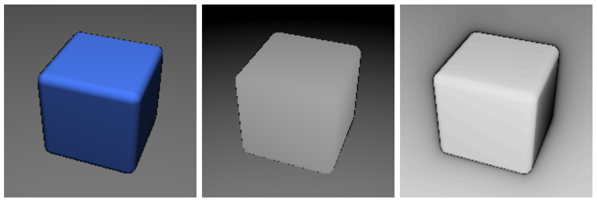

Figure 4: *Rendering modes: color, distance, steps, respectively.*

3.3.3 Bounds Optimization

A spherical bounding volume was implemented as an optimization feature. For each pixel, the renderer first checks if the ray through that pixel intersects the bounding volume, and only proceeds to ray marching if it does. Ray marching only happens within the bounds of the volume, starting at the near intersection point and stopping when it has reached the far intersection point. This technique results in a large performance gain, since a lot of work can be eliminated for the pixels that are outside the bounding volume. Coherent warp termination takes effect for the large regions outside the volume and a large performance boost can be observed.

3.4 Modeling the Scene

The scene is modeled using a distance field, or a distance estimator function. It is constructed of one or more individual distance functions that describe surfaces in the scene.

The focus of the project is on the rendering aspect, and therefore not much work is done on the modeling of the scenes. Distance functions used in the renderer are taken from various publicly available sources, many are from a great article written by Iñigo Quilez about modeling primitives with distance functions. [Quilez 2008a]

3.5 Visual Effects

This section discusses the various visual effect passes implemented as part of computing the final output color. Note that for ambient occlusion, subsurface scattering, and soft shadows, the effects are computed through purely empirical means of ray marching and therefore are not physically based.

3.5.1 Diffuse Lighting

Simple diffused lighting is computed using the local illumination formula.

$$I = k_a + I_L k_d(\hat{l} \cdot \hat{n}) + I_L k_s(\hat{h} \cdot \hat{n})^n \qquad (1)$$

3.5.2 Reflection

Mirror reflection is implemented by simply performing another ray marching color computation on a reflected ray once the initial ray hits a surface.

3.5.3 Fog

Fog in this case refers to the darkening of pixel values according to the distance marched. The goal of this effect is to hide numerical errors in the far distance where ray marching terminates from reaching the maximum number of steps. The amount of fog scales with distance as given by the following formula.

$$fog(x) = e^{-0.05x} \qquad (2)$$

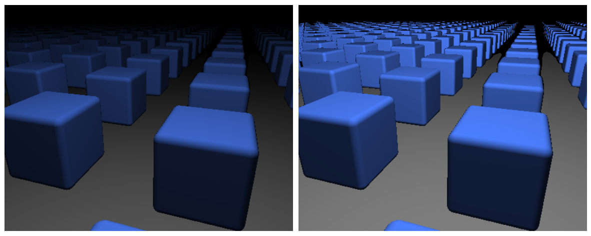

Figure 5: *Using fog (left) to hide numerical errors (right).*

3.5.4 Ambient Occlusion

The method is proposed by Iñigo Quilez [Quilez 2008c] and is originally based on another paper by Alex Evans. [Evans 2006]

The ambient occlusion coefficient at each surface point is computed by taking samples of the distance field at varying distances above the point long its normal. This essentially estimates the "openness" of the region above the point. This coefficient is simply multiplied to the output color to darken it. A mathematical description of the method is below.

$$ao = 1 - k \cdot \sum_{i=1}^{5} \frac{1}{2^i}(i \cdot \Delta - distfield(p + n \cdot i \cdot \Delta)) \qquad (3)$$

3.5.5 Subsurface Scattering

The technique is similar to the one described for ambient occlusion. The difference is that it takes samples along the direction of the marching ray. Because the technique relies on negative sampled distance, it only works on signed distance fields.

3.5.6 Soft Shadows

Soft shadows is implemented by performing a ray march from the surface to the light. If the ray to light hits another surface, the shadow is full black. Else, the shadow coefficient is determined by obtaining the smallest distance the ray to light is from the scene. The technique is credited to `the.savage@hotmail.co.uk`.

4 Results

A few final rendered images can be seen in Figure 1 on the first page. From left to right: Menger sponge, infinite plane of round boxes, U-Shape, knot with displacement function. Below is the user interface for controlling various options of the renderer.

The following section will discuss the results of the method, mainly focusing on comparing performance of the various options of the renderer. For the purpose of the comparisons, all renders discussed are at 500 by 400 pixels, running on Mozilla Firefox 11.0 and NVIDIA GeForce G 103M. Every scene has a ground plane, and unless noted otherwise, is viewed from approximately 45 degrees above horizontal. The FPS rates in the following graphs are recorded as the "stable FPS", the rate at which the renderer stabilizes, ignoring any random fluctuations.

The number of maximum steps is the most fundamental determinant of quality and performance. As more steps are allowed to be taken, the amount of numerical error is smaller and the quality improves, but the performance also decreases. Below is a graph showing the performance measure in FPS for different objects. The objects are listed roughly in order of function complexity. Notice that round box is slightly faster than regular box. This is because it is an unsigned function and requires fewer instructions.

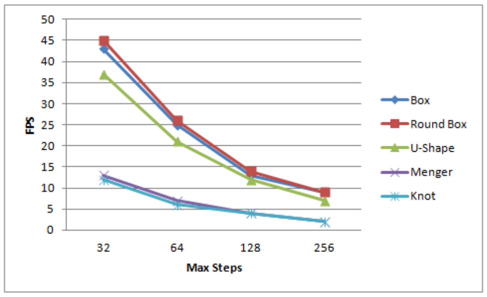

Figure 6: *Performance vs maximum steps of ray march*

The next fundamental determinant of performance is the visual effects. Ambient occlusion and subsurface scattering consume very little performance as they only take 5 samples per calculation. The heavy effects are the soft shadows and the mirror reflections, which require another ray march, and latter case, another set of color computations.

Aside from the two fundamental factors, there are a few other factors that were observed to affect FPS. The first, rather unexpected, is animation. Enabling animation (e.g., the rotating animation) seems to boost FPS, as if WebGL knows to produce a higher frame rate

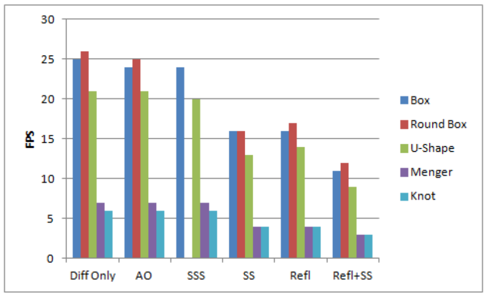

Figure 7: *Performance comparison for various visual effects*

because the scene is changing. On average, turning on animation increases the FPS by approximately 18%, with the Menger sponge gaining over a 40% boost. The other factor is early ray termination, which can be triggered by pointing the camera down in the $-y$ direction, such that rays would terminate early when hitting the ground plane. However, the expected FPS boost was not observed immediately; the boost kicked in only when animation is turned on. This behavior is indeed puzzling and there isnt a good explanation yet. The graph below shows the described results. Notice that the knot is unaffected because it is a special shape that is already animated by default. (ERT stands for early ray termination.)

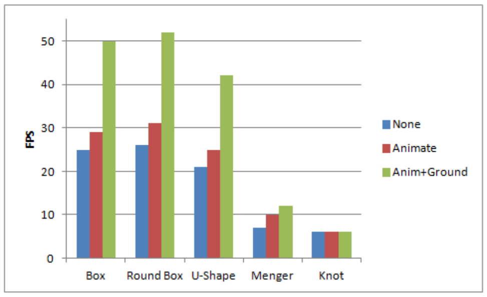

Figure 8: *Performance comparison when animation is enabled*

Lastly, the bounding volume optimization proved to be significant. On average, scenes experience a 64% gain in performance when running ray march with bounds checked. Two components contributed to this performance gain. First is the bypassing of ray marching completely when it does not intersect the bounding volume, and second is the shortened range for ray marching. The first component is especially significant for two reasons: 1) the entire workload for color computation is eliminated, and 2) pixel areas that are outside the bounding volume are contiguous. These two reasons allow for very noticeable early warp termination, which can be observed by the FPS boost as the camera zooms out and the bounding volume takes up less space of the screen.

Despite the large performance gains, there is an obvious drawback for this technique, which is the inability to render large scenes that extend outside the bounding volume, such as an infinite plane of cubes. Therefore, the optimization is implemented as an optional feature that can be enabled when appropriate.

5 Conclusion

I am quite satisfied with the final results. All of the planned visual effects were implemented and the renderer is able run at reasonable performance. Nonetheless, the project is not complete. There are complex scenes that the renderer cannot handle well, either in terms of quality or performance. There are also still a few known bugs to look into. The following is a list of possible future improvements.

- Optimizing the ray marching step. I believe more can be done since there exist other similar renderers which can produce complex fractals at good performance
- Support more complex shapes. This is related to optimizing and making the renderer more powerful and versatile.
- Support user defined scenes.
- Focal blur. Currently a work in progress, enabled on the renderer but does not produce good results.
- Glow. Very easy to implement based on current framework.
- Anti-aliasing

References

CRANE, K. 2004. Ray tracing quaternion julia sets on the gpu. http://users.cms.caltech.edu/keenan/project_qjulia.html.

EVANS, A. 2006. Fast approximations for global illumination on dynamic scenes. In *ACM SIGGRAPH 2006 Courses*, ACM, New York, NY, USA, SIGGRAPH '06, 153–171.

HART, J. C., SANDIN, D. J., AND KAUFFMAN, L. H. 1989. Ray tracing deterministic 3-d fractals. In *Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, SIGGRAPH '89, 289–296.

HART, J. C. 1994. Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer 12*, 527–545.

QUILEZ, I. 2008. Modeling with distance functions. http://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm.

QUILEZ, I. 2008. Raymarching distance fields. http://www.iquilezles.org/www/articles/raymarchingdf/raymarchingdf.htm.

QUILEZ, I. 2008. Rendering worlds with two triangles with raytracing on the gpu. http://www.iquilezles.org/www/material/nvscene2008/rwwtt.pdf, August.