Ray Marched Objects in a THREE.js Scene

Tyler Evans

December 19, 2017

1 Project Objective

The goal of this project was to display complex objects that cannot be defined by a traditional polygon mesh in such a way that they appear 3D inside of a THREE.js scene. In order to accomplish this, we have a 2D plane in the scene that is textured with a ray marched rendering. This texture and the plane that it is drawn on are manipulated in such a way that the object appears to be 3D within the scene. A GUI was provided to customize the objects.

2 Background Information

In traditional ray tracing, we find where the ray (line that passes from the camera through a particular pixel in the image plane) intersects the closest object in our scene. This approach works great for objects that can be defined by polygon meshes since we can easily find intersections between our ray and the triangles/vertices of our models. However, what can be done to try and render objects that can not be described by such polygon meshes? The most prevalent example of such a case is the rendering of 3D fractals. The problem here is that we cannot render these objects using a traditional ray tracer because we most often cannot solve for an intersection with our object due to it's complex or continuous nature.

2.1 Ray Marching

Ray marching is a technique that can allow us to render complicated objects that may not be able to be represented by a traditional polygon mesh. It is a variant of ray tracing in which instead of finding the intersection point between a ray and our object, we iteratively "march" along the ray direction, checking each time if we have collided with the object or not, up to a maximum defined number of steps and distance (in which case we conclude that this ray does not intersect the object). See [JCH89] for one of the first papers to document this approach.

In order to accomplish this, we need a way to determine the distance between our current position along the ray and our object.

2.2 Signed Distance Functions

Signed distance functions are those that return the distance from the point in question to the closest point on the boundary of our object. Points outside of the object are assigned positive values, points on the boundary of the object map to zero, and points inside of the object map to negative values. Using these functions, we can now march along our ray, checking the distance function each step, stopping when the value is as close as we want to zero.

[Note that SDF's return the distance to the closest point on the object, not the closest point to the object along the marching direction. For example, a ray that is passing slightly above the face of a cube in a direction tangent to this face will march very slowly even though it will never intersect with the cube.]

One of the simplest examples of a signed distance function is for a sphere of radius r centered at the origin. We are interested in the distance from this sphere to a point $p \in \mathbb{R}^3$.

$$\text{sphereSDF}(p) = \|p\| - r$$

It is easy to see that the properties of distance functions hold in this example, and that points of length r are on the boundary of the sphere and are indeed mapped to zero.

Note that if we wanted to translate this sphere to a point q in our scene, we can simply evaluate our distance function at the point $p - q$. Other similar operations can be done to scale or skew the object as well.

Once we have a distance function for a desired shape, we can also combine the functions in ways that can generate more complex objects. For example, to union two objects, intuitively the distance to this new object is simply the minimum of the two distances to the original objects.

See [Qui] for a comprehensive list of distance functions for primitive shapes as well as some of the basic operations that can be used to combine and distort these shapes.

2.3 Mandel Bulb

The Mandel Bulb Fractal is the showcase object used to display the techniques of this project. Estimating the distance function for this object is based on taking a particular sequence of points in the complex plane and determining if this sequence converges or diverges. See [JCH89] for a description of how distance functions for fractals like this are estimated. I won't pretend to understand how these things actually work, so an adaptation of the method seen at [Chr] was used for this project. See figure 3 for the Mandel Bulb.

2.4 Shading

Shading in approaches such as the Phong Illumination Model require normals to be interpolated across surfaces of the object. This is easy when our object is represented as a polygon model because we have the vertices of our object and can solve and interpolate based on these vertices. However, the signed distance functions that we are ray marching do not have vertices that we can sample and find the normals for. Luckily, signed distance functions posses a convenient property that we can exploit to estimate the normal at any point on the object.

Signed distance functions are positive outside of the boundary of the object and negative inside. Due to this, in order to find the normal at a given point on the boundary of the object, we can find the direction in which the function changes most rapidly from negative to positive. This direction can be calculated by finding the gradient of our function at the given point [JCH89]. The gradient is easy enough to estimate, we can find it by sampling differences between nearby points in all three spacial directions. Once we have the normal, we can employ a lighting model of our choice.

2.5 Optimizations

Instead of marching with a constant marching step, many implementations of ray marching use the "marching spheres" approach. In this approach, since signed distance functions return the distance to the closest point on the boundary of the object, we know that we can safely march this distance without overshooting the object.

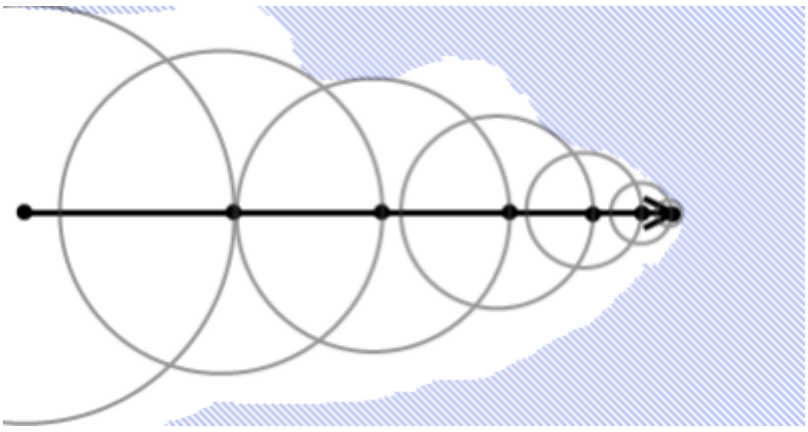
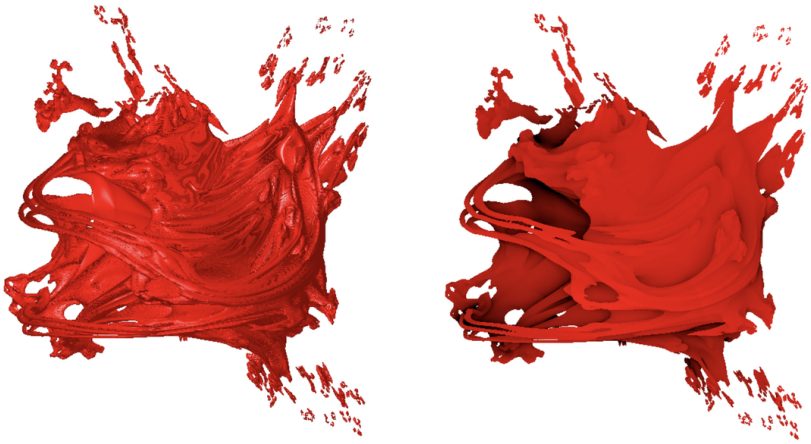


Figure 1: Visualization of Ray Marching with Varying Step Size [src]

Another optimization is to cheaply shade the object based on the number of steps taken to intersect the object [Chr]. In this "march shading" approach, we can assign a certain color to pixels whose rays to not intersect with any object (the background), then for those that do intersect, assign a color to the point with darkness as a function of the number of steps taken. This results in an image that has apparent depth, giving the object a 3D appearance, and can also serve as a hack to simulate ambient occlusion. This approach is cheap since the number of steps is calculated automatically when we ray march, and no lighting equations or estimation of normals are required.

This approach gives the object a very matte and smooth appearance, and I found it to be a good shading for very complex objects. The Phong model often would produce visual artifacts and ripples in the image. My guess is that since these complex objects may not have very "nice" derivatives, we may get incorrect results when estimating the normal by finding the gradient. Shading the object based on the number of steps eliminated these artifacts because the normal of the surfaced is not involved. Figure 2 shows the differences.



(a) Phong Model (b) Step Based "March Shading"

Figure 2: Shading Models on a Power 2 Mandel Bulb

Figure 3: The Mandel Bulb Fractal (Power 8 Mandel Bulb)

3 Implementation

The ray march algorithm is a modified version of the one found in [Won]. All shapes except for the Mandel Bulb were created by using combinations of primitive functions and operations found at [Qui]. The algorithm for the Mandel Bulb was modified from [Chr].

My Contribution

- Figure out how to incorporate the ray marched renderings into a THREE.js scene.
- The manipulations required to make these renderings appear 3D within the scene.
  - Map the shader to texture coordinates within the scene, communicate these to the ray march fragment shader with a vertex shader and transformations.
  - Manipulate the texture in such a way that it appears to have depth when the camera moves.
  - Move the ray march origin and the texture canvas based on the position of the THREE.js camera to act as a view window into the ray marched rendering.
- Design GUI for the rendering and object parameters.
- Modify the ray marcher to be customizable by the GUI parameters.
- Communication between the javascript program and the shaders, uniforms and control logic.
- Create signed distance functions for various objects (except for the "twisted torus" and "Mandel Bulb", see [Qui] and [Chr]).
- Customize the Mandel Bulb algorithm so that it works with the chosen ray marcher and is customizable by the GUI parameters.
- Implement "march shading" for the Mandel Bulb
- Figure out how to incorporate multiple ray marched object instances into a single scene.
- Create the demo scenes.

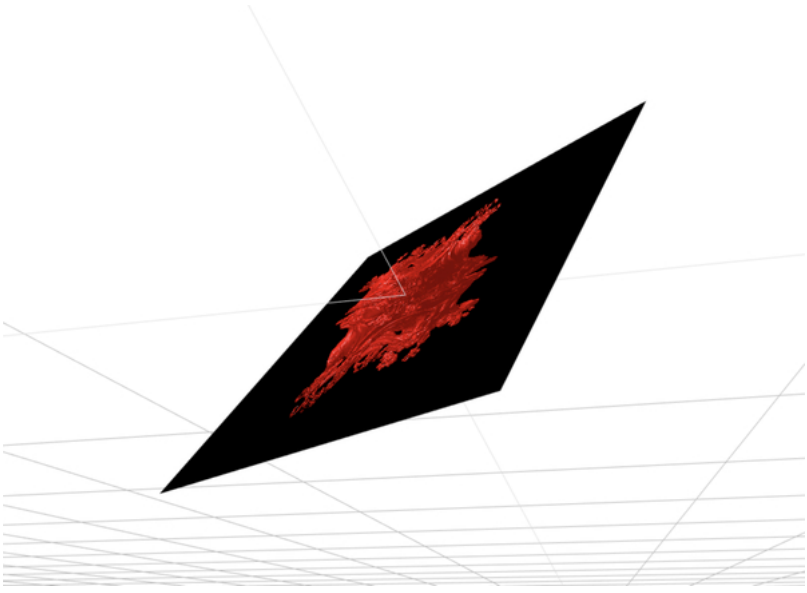


Figure 4: Object with Manipulations Disabled

I have prepared three scenes in THREE.js to showcase the above efforts.

***raymarchMain.html/js*:** The main scene where the approach was first implemented. It acts as a viewer for the various ray marched objects and contains the GUI to modify and select these objects. In the GUI, certain options disable intermediate steps in the technique which can aid the user in visualizing how this all works. I recommend the following steps to gain insight into what is happening in the scene:

1. Start with all of the default options (you can refresh the page if any were changed).
2. Use the mouse to move the camera around the object, it appears 3D in the scene.
3. Check "Grid", this helps show that the object is indeed in a THREE.js scene.
4. Check "Shader Plane". This will help show the canvas that the texture is being drawn to. Zoom out so that the entire black plane is visible.
5. Un-check "Plane to Camera" and "Eye to Camera". Move the camera around. With these settings it is easy to see that all we have is a texture on a plane. Your view should resemble figure 4.
6. Check "Eye to Camera". Move the camera around and notice that for viewing angles relatively orthogonal to the plane, the object appears to have depth.
7. Check "Plane to Camera". Move the camera around and notice how the object appears 3D within the black canvas.
8. Un-check "Shader Plane" to get back to default and see the full 3D effect in the scene.
9. Play around with the settings in the GUI.

***raymarchMultiple.html/js*:** This file shows how we can have multiple instances of these ray marched objects in the same scene. To accomplish this, each texture is stored in a global list and is assigned its own copy of uniform variables. When we change the view, update each texture in the list accordingly.

***raymarchInScene.html/js*:** Finally, a simple demo scene in which we have a combination of ray marched objects and traditional THREE.js objects (a stripped down version of the claw machine used in assignment 1). This scene shows the contrast between traditional meshes and ray marched objects, as well as how these ray marched objects can be placed in any scene as easily as the traditional ones are.

Note that the html files for each of the three different scenes all contain the same vertex shader and fragment shader code. These shaders are what render the ray marched object on the THREE.js textures in each scene.

4 GUI Options

Here are the less self explanatory options available in the GUI that change uniform values being sent to the shaders. The first three options listed below are intended to help the user visualize how the process of making the objects appear 3D in the scene works. The final three options are intended to be exclusive the the Mandel Bulb object.

Shader Plane:	Toggles the alpha value of the points in the shader that do not reach the object. This allows the user to better visualize that the object is just an image on a plane.
Plane To Camera:	Toggles whether or not the plane that the shader is drawn on follows normal to the camera. Helps show why this step is necessary in making the object appear 3D.
Eye To Camera:	Toggles whether or not the origin of the ray march (micro camera) follows the THREE.js (macro) camera. Helps show why this step is necessary in making the object appear 3D.
Fractal Power:	Power used to generate the distance function estimator sequences for the Mandel Bulb fractal.
Morph Fractal:	Animates the fractal power from values between 0 and 6. Showcases the fact that this is not a static model, but rather a continuous, mutable function.
March Shade:	True enables the cheap shading trick based on how many steps away a point is. The parameters for this shading were tuned specifically for the Mandel Bulb: enabling this shading for the other objects is not intended. Default of false uses the Phong Model.

5 Conclusion

In this project, an approach to place ray marched renderings of complex objects inside of a THREE.js scene was implemented. Three files were presented which showed how this method could be customized with GUI parameters, extended to include multiple ray marched objects, and how these objects could be placed alongside traditional objects in a scene. Options to break apart these steps were also available which helped visualize the concept of having a scene being rendered within a scene being rendered.

References

[Chr] Mikael Hvidtfeldt Christensen. Distance estimated 3d fractals. http://blog.hvidtfeldts.net/index.php/2011/09/distance-estimated-3d-fractals-v-the-mandelbulb-different-de-approximations/. Accessed: 2017-12-10.

[JCH89] Louis H. Kauffman John C. Hart, Daniel J. Sandin. Ray tracing deterministic 3-d fractals. https://graphics.cs.illinois.edu/sites/default/files/rtqjs.pdf, 1989.

[Qui] Inigo Quilez. Modeling with distance functions. http://iquilezles.org/www/articles/distfunctions/distfunctions.htm. Accessed: 2017-12-5.

[src] https://www.alanzucconi.com/2016/07/01/raymarching/.

[Won] Jamie Wong. Ray marching and signed distance functions. http://jamie-wong.com/2016/07/15/ray-marching-signed-distance-functions/. Accessed: 2017-12-12.