Rendering Opalescent Materials using Photon Mapping and Ray Tracing

Daniel Janikowski

Abstract

This paper describes a method for representing opalescent materials in a semi-accurate and efficient way to allow for fast rendering via standard ray tracing along with photon mapping. Opalescent crystals display a wide array of very bright, nearly emissive colors. While the crystals can be quite thick, their opalescent layers are usually only fractions of a millimeter thick. To efficiently represent such a structure, a simple polygonal face with a texture a normal map can be used to create a realistic rendering. The microstructure of opal is well approximated by the normal map while the texture provides the color for each patch.

1. Introduction

Ray tracing has come such a long way allowing for increasingly accurate renderings of real-life phenomena. Opalescence is an example which seems to have been overlooked leaving room for improvement. With the phenomena resulting from a combination of several complex interactions, producing an accurate model may seem like a difficult task, however, this may not be the case. Opal crystals produce their magnificent color because of their dense micro-structure of silica spheres. These spheres settle and form large structures called photonic crystals. These crystals can be thought of as thin sheets of varying thicknesses and orientations. Taking all this into account, textures are a perfect, efficient choice for storing such information. Sampling textures can be done incredibly quickly by modern computers and so storing these two compoenents in a normal and texture map seems like an obvious choice. By producing a normal map which replicates the semi-random structure of the photonic crystals, photon mapping and ray tracing can be used to replicate the glinting and 'play of color' that one would see while looking at an opal.

1.1. Previous Work

With opal containing such complex internal structure the most obvious approach to model it would be to generate an approximation using procedural modeling. However, to obtain a realistic result would require a large number of surfaces causing very poor performance from most naive ray tracing algorithms. Efficient volumetric data structures would need to be implemented in order to reduce the time spent looping over the geometry to determine ray intersections.

A previous method to render opal and its 'play of color' properties was put forth by Imura et al. [1] where they took a similar approach to the one described above. Their attempt divided the volume of the provided model and randomly filled it with geometry called 'clusters'. Each cluster was randomly placed within the model up to a count such that the cluster density matched real opal. The clusters had properties such as color, size, and orientation. Each cluster is assumed to have uniform density which allows them to have the same reflectance properties.

Of course, such a representation hits some of the same efficiency issues as discussed earlier. Although, the model is 3-dimensional and clusters are created within the entirety of its volume the typical flat shape of opals results in renderings that do not show off this depth. This leads to the trivial simplification of storing the same information within 2-dimensional textures.

2. Ray Tracer

My implementation for ray tracing began with the final submission code for homework 3. My first goal was to implement refraction since much of the material within an opal is silica which is mostly transparent. I quickly realized a large portion of the algorithm was unnecessary such as multi-sampled shadows and glossy looking material. This combined with the knowledge of Monte-Carlo path tracing resulted in a complete rewriting of my 'traceRay' function. This reduced the number of arguments and greatly simplified the recursive tracing function. Shadow, reflection and refraction rays are now always single samples however higher quality results can be achieved by increasing the number of 'anti-aliasing' rays. The general recursive trace ray function proceeds as follows:

Algorithm 1 Recursive Trace Ray

```
1:  procedure TRACERAY(ray, hit, bounce)
2:      intersect = CastRay(ray, hit)
3:      if !intersect then
4:          return background
5:      end if
6:      if hit.material == diffuse then
7:          color += ComputeDiffuseColor(hit.coords)
8:      end if
9:      color = [0, 0, 0]
10:     if hit.material == reflective then
11:         color += TraceRay(reflect, nhit, bounce+1)
12:     end if
13:     if hit.material == reflectiveAndRefractive then
14:         ratio = CalcFresnelRatio(norm, ray.dir, ior1, ior2)
15:         color += ratio * TraceRay(reflect, nhit, bounce+1)
16:         refractColor = TraceRay(refract, nhit, bounce+1)
17:         color += (1 - ratio) * refractColor
18:     end if
19:     if hit.material == opal then
20:         color += OpalColor(hit.coords)
21:     end if
22:     return color;
23: end procedure
```

The logic for computing the color of a diffuse hit simply follows the Phong lighting model:

$$C = C_E + C_L * C_D * cos\theta + C_L * C_S * (cos\theta')^{100} * cos\theta \quad (1)$$

C	Surface color
C_E	Emitted surface color
C_L	Light color illuminating current surface
C_D	Diffuse surface color
C_S	Specular surface color
θ	Angle between light and surface normal
θ'	Angle of ideally reflected light ray

If a surface has a non-zero reflective component then this diffuse color is summed with a recursive call to the *TraceRay* function using a ray reflected about the surface normal from the direction of the previous incoming ray.

2.1. Material Properties and Refraction

To handle intersection with transparent materials, logic had to be added for the calculation of the Fresnel ratio along with the direction of refraction rays. This logic would be very similar to that of the reflective rays however these needed to know which side of the face was being intersected. All models I used contained a version of an intersection function which provided the direction of the surface normal at any point. The ray direction is dotted with this surface normal to determine which side of the face the ray was hitting.

The homework 3 code provided a basic material class. Each model within the scene had an associated material object containing the following properties: diffuse/texture color, reflective color, emitted color. To account for refraction, variables for both the index of refraction and refractive color had to be added. Handling refraction proceeds as follows:

Algorithm 2 Refraction

```
cosi = Dot(ray.direction, normal)
ior1 = 1.0
ior2 = hit.material.ior  ▷ Index of refraction of the material
if cosi < 0 then  ▷ Ray intersects front side of face
    cosi = cosi * -1.0
    iorRatio = ior1 / ior2
end if
if cosi >= 0 then
    normal = normal * -1.0
    iorRatio = ior2 / ior1
    Swap(ior1, ior2)
end if
ratio = CalcFresnelRatio(normal, ray.direction, ior1, ior2)
if ratio ¡ 1.0 then
    refractDir = RefractDir(cosi, iorRatio, ray.direction, normal)
    refractColor = hit.material.refractiveColor *
                TraceRay(refract, nhit, bounce+1)
end if
continue steps of previous algorithm
```

The first part of the logic checks to see which side of the face the ray intersects. To handle a ray coming from within a volume and hitting the back face, the indices of refractions are simply swapped to get the proper ratio. Fresnel's equation is used to get the ratio of reflected to refracted light. As the incident viewing angle to the surface of a transparent material increases the amount of light reflected also increases. If there is some amount of light being refracted across the surface then the direction of the refracted ray is computed using the following formulas:

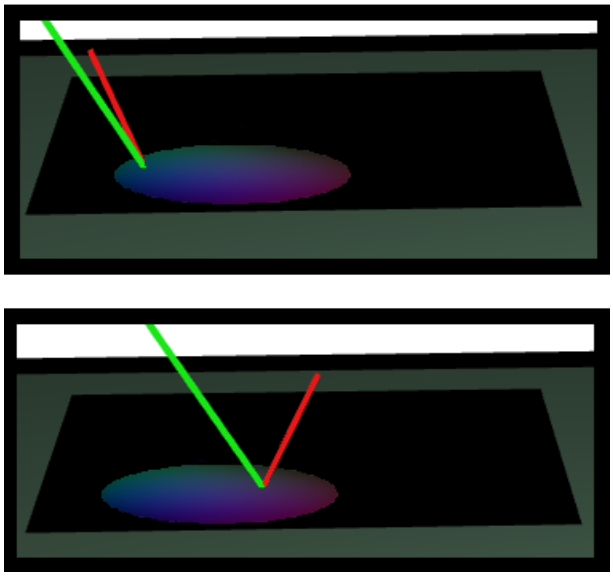

Figure 1: *Surface normal vector (red) changing direction according to normal map respective to intersection point of incident ray and face.*

$$k = 1.0 - rr^2 * (1.0 - cosi^2)$$

$$\text{if } (k < 0) \quad V_R = rr * V_I + (rr * cosi - \sqrt{k}) * V_N$$

Where rr is the ratio of indices of refraction, V_I is the incident ray direction and V_N is the surface normal.

A further extension I made to the material class was adding an enumerated type which represented the class of material: light, diffuse, reflective, reflectiveAndRefractive, opalescent. The material type was decided during the materials creation where depending on the magnitudes of the color types (i.e. emitted color or diffuse color) a material property was given. This removed the need for unnecessary logic within the ray tracing functions.

2.2. Opal Material

As discussed previously my idea to efficiently represent the photonic crystals within an opal was to store the crystal face normals and colors within texture files. This made the same presumptions as stated in [1]. A further assumption was that the thickness of the crystal layer was negligible as the cloudiness and size of the stone itself results in a uniform looking depth plane of color.

Since the material class already contained a variable to store an image to represent the color of a surface, I extended this by adding variable to store the normal map texture. This variable would only be used if the material type was set to be opal. With this the normal of the surface could be computed anywhere simply by sampling the color at the point from the texture file. To convert the given color to a direction I used the standard method where each color channel is mapped to a given axis. Assuming color values are provided in a range from 0 to 255 then red is mapped to -1 to +1 on the x axis, green is mapped to -1 to +1 on the z axis (since OpenGL considers the vertical direction to be the y axis) and blue is mapped to 0 to 1 on the y axis. A vector is then created with these values, normalized and returned to represent the normal at that point.

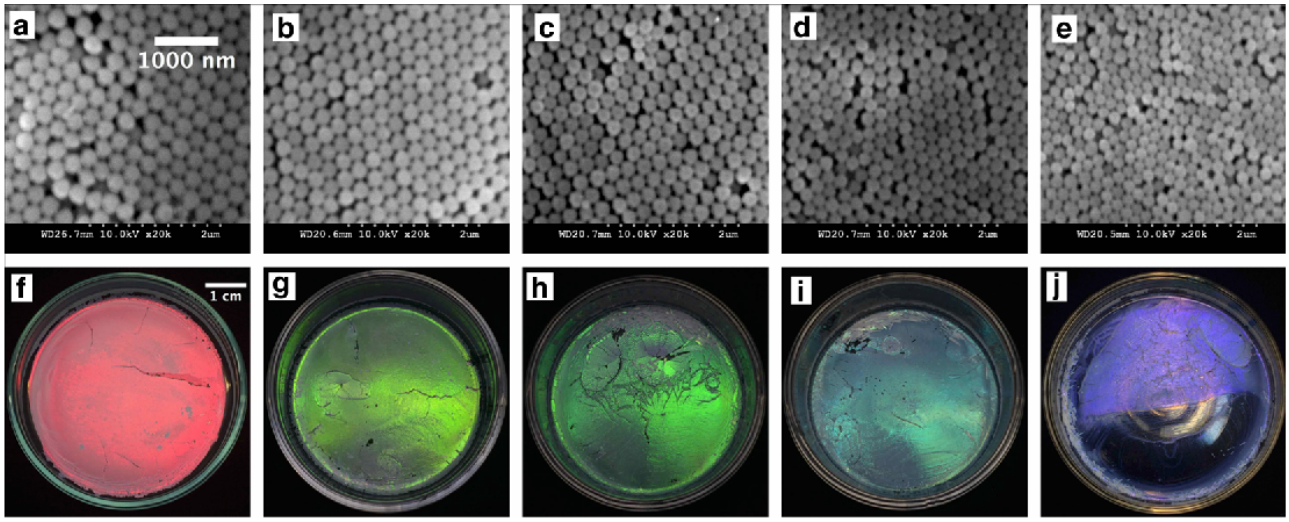

Figure 2: *Photonic crystal color based on silica micro-structure arrangement [2]*

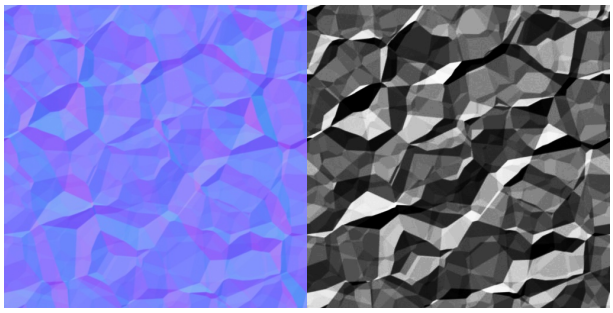

Figure 3: *Example normal map (left) wavelength/texture map (right).*

To test this implementation I used a simple normal map containing a hemisphere. As seen in Figure 1 when the incident ray (green) moves across the surface the normal direction (red) also changes even though the face is planar.

My method of interpreting the color at an intersection point was derived from the way in which photonic crystals function in real life, with some simplifications. The structure of an opal photonic crystal is made of small silica spheres on the order of a few hundred nanometer. That size is also roughly the scale of the wavelengths of visible light. Thus, when photons enter the crystal they either bounce off the surface of the spheres or pass down to a further layer then reflect. As the light passes up and down between the layers a combination of diffraction and interference results in certain wavelengths being amplified. In reality these crystals can produce a variety of colors based on the viewing angle due to the chaotic nature of the crystal layers. I simplified this by assuming each crystal was regular meaning only one color would be visible. This is also the assumption made from the 2003 opal paper [1] and can be seen in the real experiment done by Gao et al. [2] in Figure 2.

To encode this I thought of using the normal map, since this already defined the crystal boundaries, then have a color texture made from the normal map. Since real photonic crystals have an associated wavelength I chose to simply map the visible spectrum ($380 \leq \lambda \leq 780$) down to the range 0 to 1 encoding them as grey scale values. For rendering, this process is done in reverse. Figure 3 shows an example of a crystal-like normal map along with its grey scale color map. With this a ray intersects the plane, the texture is sampled, the greyscale value is extracted and finally converted to RGB. To convert to RGB I used a piece-wise approximation function.

3. Photon Mapping

Rendering the 'play of color' of opal requires drastic intensity changes dependent on the incident angle to the internal structural faces. The effect is amplified by refraction through the transparent outer layers of opals. This requires the use of global illumination techniques of which I chose to use photon mapping, specifically based on the work by Jensen [3]. Once again I re-used much of the code from homework 3 although I now had to alter the *TracePhoton* to match the functionality of *TraceRay*. For a minor improvement to performance I did not add photons which have not yet bounced as the ray tracing algorithm by itself can handle direct lighting. For reflective/refractive surfaces I use a Monte-Carlo technique where I use the Fresnel ratio and a random sample to decide whether the photon should reflect or refract. From the homework the photon class is able to store the incident direction. This is useful during the collection/tracing faze as it allows rays to only accumulate photons which are deposited on surfaces facing towards the incident ray. For any material other than opal the photon is stored with its incident direction.

The color layer within opals is highly sensitive to the viewing and lighting angle. As discussed in the work from Imura et al. [1] preprocessing is done to store the intensity values of each cluster based on their orientation to the light soruce. When forward ray tracing is executed and a ray intersects a cluster, the clusters intensity, incident direction and normal are used in the Phong model to produce the rendered color. Using this idea, I store the reflected angle of the photon off the opal surface within the incident direction variable. Then, when collecting photons for global illumination I make sure to only collect photons which reflected directions are within some $d\theta$ of the incident view ray. The color of the photon is simply multiplied by the texture color at that location.

4. Parallelization

The major throttling points for such an application are definitely during ray tracing and photon mapping. The recursive nature of ray tracing leads to an explosive number of computations. Using Monte-Carlo path tracing greatly speeds up the application while still achieving some of the nice costly visual effects such as soft shadows. However, parallelization would further decrease the total render time. I faced two issues while trying to parallelize the ray tracer. First was that drawing to the screen is usually impossible to do in parallel as the screen buffer can only be written to sequentially. Luckily, writing directly to the screen was not necessary. A sort of OpenGL hack where instead of drawing directly to screen pixels were instead created as quads. The algorithm then proceeds by determining its color through recursive tracing and finally adding this color along with its coordinates to a vector. The information within the vector is then used to create the quads. This algorithm can be parallelized although care has to be taken when writing to the vector.

The other more minor issue was that ray tracing is innately recursive which does not translate well to a GPU. Because of this I decided to do the parallelization on the CPU as the more powerful (smarter) cores can more efficiently manage recursion.

The other part of the algorithm to be parallelized was photon mapping. This is a much simpler procedure as it just scatters photons into the scene in varying directions. When photons intersect with specific surfaces they are then stored to the spatial data structure (in this case a k-d tree). Care must also be taken when adding to this data structure as it is also composed of vectors.

To handle the parallelization I took advantage of the OpenMP commands for parallelization on the CPU. Firstly, running the ray tracer scan line loop in parallel ensuring that all pixel color push backs were encapsulated within a critical block. Standard vectors are not thread safe and so all push back calls needed to be handled one at a time. Once a full horizontal line is scanned then the quad adding algorithm is run to create all the necessary 'pixels'. A similar procedure is done for the photon mapping however the loop is done all at once. For the given number of photons to be traced a loop is done in parallel to trace all photon paths. Adding photons to the k-d tree is also done in a critical block since the structure contains vectors at each branch.

5. Results

For the number of simplifications made to the true model of lighting interactions with opals the results were very promising. For the majority of my results I used a very simple scene consisting of a textured ground plane along with an opal textured plane and a sphere just clipping through the top portion to give a dome shaped appearance.

Originally, I tried to only use the Phong lighting model with different amounts of specular reflections in hopes to get a decent result. As seen in Figure 4 there is too much regularity visible and the faces do not produce that shine which is so characteristic of opal. The normal/color maps for this image are the same as seen in Figure 3. I modified the color mapping to span mainly the blue and green part of the spectrum.

Once photon mapping was implemented I saw a great improvement in the results. I decided to use a much more detailed normal/texture to see if I could get a better play of color which can be seen in Figure 5. Figure 6 then shows the photon mapping results using these two textures. Figure 7 uses the same

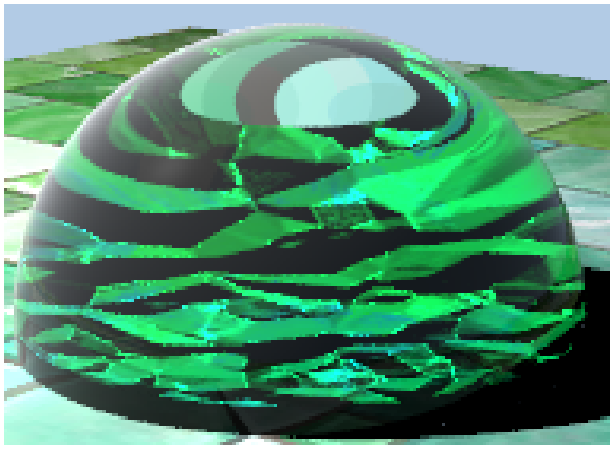

Figure 4: *Initial result without photon mapping.*

textures but with a full spectrum mapping. The main issue with these is the normal map does not produce enough variation in color as the angle changes. I also figured out the parameters for the photon mapping were not optimized as well as the collecting direction.

I finally found the best configuration of textures and photon mapping parameters in my final results (Figure 8). I went back to using the larger crystal normal map from Figure 3 and set my final number of photons to map to be $5,000,000$, collecting 15 and only collecting photons within a 15 degree arc of the reflected direction. The color changes drastically with smaller movements in the viewing angle resulting in much brighter patches.

5.1. Timing Results

Here I consolidated two tables of timing results for both the ray tracing algorithm as well as for photon map by sequential calculations and with parallelism. All timings being measured in seconds. All renderings were performed on my personal computer which has a AMD Ryzen 7 3700x, 3.6 GHz, 8 cores and 16 threads.

Ray Tracing: Scene consists of a refractive glass ball over a textured plane with a singular light source. Scene parameters are 9 anti-aliasing rays, 1 shadow ray, and a max of 6 bounces:

Window Size	200x200	400x400	600x600
Sequential	13.19	71.64	196.13
Parallel	8.97	48.68	137.53

Photon Mapping: Scene consists of the geometry seen in Figure 8. Parameters include range from shooting $500,000 - 10,000,000$ photons:

Photon #	0.5 mil	1 mil	5 mil	10 mil
Sequential	3.94	7.65	37.64	73.65
Parallel	0.69	1.08	4.69	9.24

From the above results we see that the photon mapping very nicely parallelizes in comparison to the ray tracing. Both the sequential and parallel methods show a steady linear increase in computation relative to the number of photons with the parallel method being around 6x faster on average. The ray tracing algorithm does not parallelize as well exhibiting only about a 50% increase in speed.

6. Future Work

My biggest complaint about the results of this project were the inaccuracies in the resemblances in normal and color maps. It would be a better idea to create an algorithm similar to the one described in [1] but creating a 2D projection of the geometries into a normal map and color map. The method I used of creating the grey scale image from the normal map resulted in all faces of the same orientation having the same color (as seen in Figure 8, as the model is rotated the majority of the red/orange faces turn green).

Given more time I would prefer to create a more robust and accurate model including the rendering of translucent materials for the substance for which the color is suspended in. For this I would like to work off of Henrik Wann Jensens paper [4]. Also using more accurate light tracing methods such as full spectrum ray tracing.

Regarding parallelism I would like to further optimize or completely rewrite the animation loop in order to have less wasted time restarting threads to render the image.

7. References

[1] M. Imura, T. Abe, I. Kanaya, Y. Yasumuro, Y. Manabe, and K. Chihara, "Rendering of 'play of color' using stratified model based on amorphous structure of opal," *Digital Image Computing: Techniques and Applications*, vol. 1, no. 7, pp. 349–358, Dec. 2003.

[2] W. Gao, M. Rigout, and H. Owens, "Facile control of silica nanoparticles using a novel solvent varying method for the fabrication of artificial opal photonic crystals," *Journal of Nanoparticle Research*, vol. 18, 2016.

[3] H. W. Jensen, "Global illumination using photon maps," *Rendering Techniques*, pp. 21–30, Jun. 1996.

[4] H. W. Jensen, S. R. Marchner, M. Levoy, and P. Hanrahan, "A practical model for subsurface light transport," in *Proceedings SIGGRAPH 2001*, Los Angeles, California, Aug. 2021, pp. 511–518.

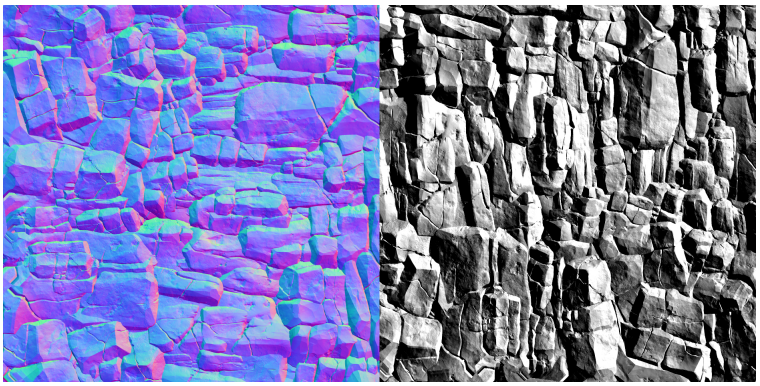

Figure 5: *Second attempted normal and color maps. Images are produced by rotating the camera around to model by 90 degree intervals clock-wise.*

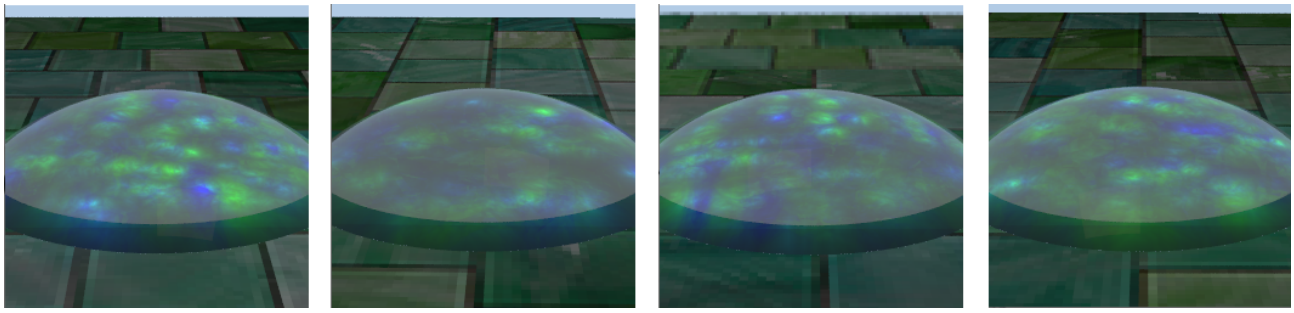

Figure 6: *Photon mapping results using Figure 5 textures. Images produced by panning over the top of the model.*

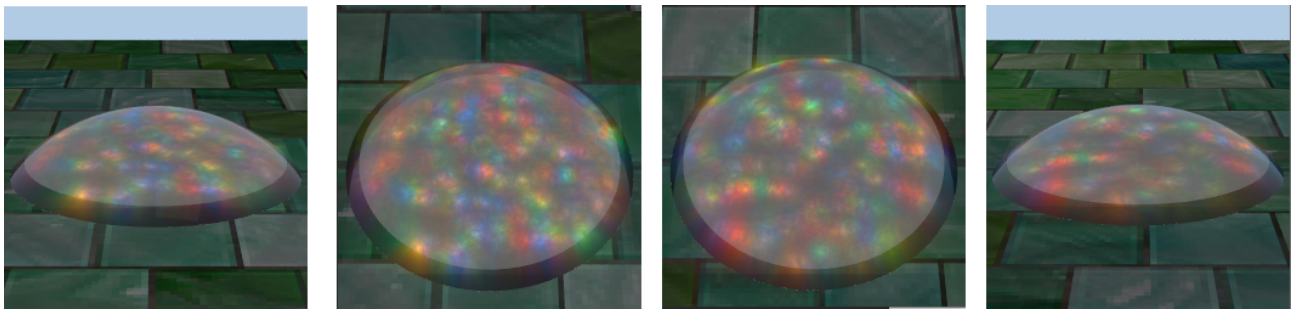

Figure 7: *Photon mapping results using Figure 5 textures.*

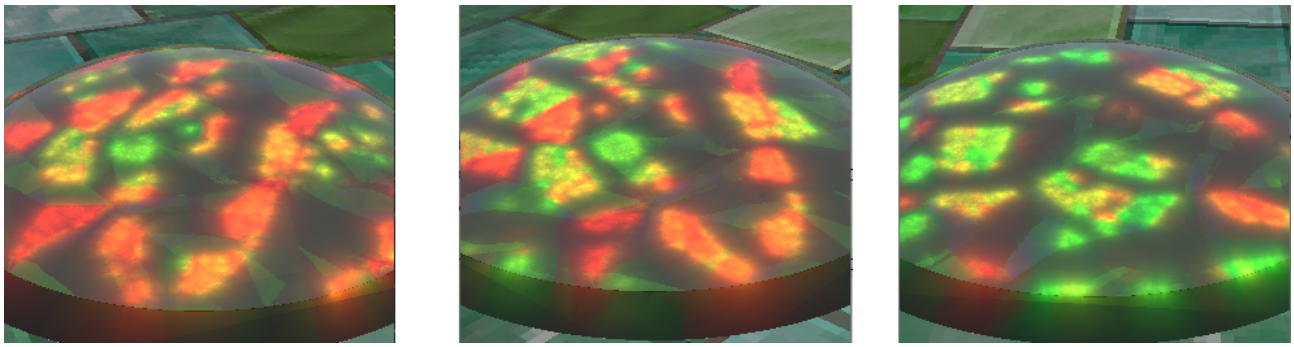

Figure 8: *Photon mapping results using Figure 3 textures. Model is rotated by roughly 20 degrees counter-clock-wise between each image.*