Computer Graphics
Coursework Three:

"Pondering The Dsphere"

by s1832137

S1832137

Table of Contents:

S1832137

Implementation:

1.1 Introduction:

This section of the report describes the methods used to extend the rendering tool PBRT-v3 [1] to include the functionality to display "displacement mapped" [2] spheres.

1.2 Added and Changed Files:

There are only 2 added files to the PBRT-v3 source code structure. These are **"DisplacedSphere.h"** and **"DisplacedSphere.cpp"**. These files constitute the vast majority of the changes to the source code. They are to be added to the **"src/shapes"** folder.

The file **"api.cpp"** is the only other file that has been changed. This changed file should overwrite the one in the original source code. It is provided within the zip that contains this report.

The makefile should not need modified.

1.3 Finding the Displacement:

One of the key problems in implementing the displacement mapped sphere is determining the amount of displacement at a given point on the sphere. This is typically solved by using a **UV Map** [5]. A UV map on a sphere maps a square onto a sphere, with the letters 'u' and 'v' denoting the axes of the square. We can treat the UV map as a *heightmap*, that is, the value at (u,v) is the amount of displacement of the sphere at that point. Given a normalised point in object space (x, y, z), we can find the coordinates of the corresponding (u,v) point by using the following equations [5]:

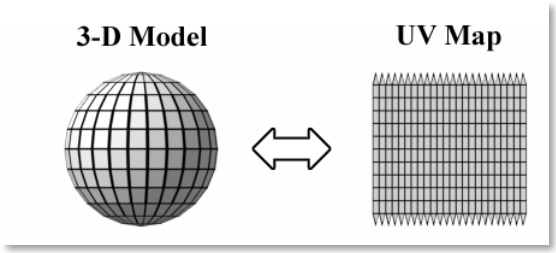

Figure 1: UV Mapping [5]

$u = 0.5 + (\arctan2(dx, dz) / 2\pi)$,

$v = 0.5 - (\arcsin(dy) / \pi)$.

Now, whenever we have a point in object space, we can normalise that point and find the displacement of the sphere at the line that passes through the sphere's centre and that point. This will be useful for our intersection test.

Implementation:

1.4 Reading in Images:

Because we can map a square onto the sphere, we can now read images and treat them as heightmaps. The displacement at a given (u,v) will be the sum of the image's r, g and b values, divided by 3 (greyscale) and multiplied by the maximum displacement allowed (discussed later).

1.5 Intersection Test:

Given that we do not have a precise function that describes the surface of the shape, **ray marching** [3] was used to render the displaced sphere.

The ray marching algorithm does not attempt to find an exact solution to the ray/object intersection. It instead approximates it by taking 'steps' along the ray until it is deemed acceptably close to the object to determine that it has hit, or by terminating the ray once it has travelled too far (see right).

Ray marching is a powerful algorithm in that it can - so long as there is a known distance function to the object - render any shape. However, the quality of that render depends on how precise the minimum distance is, and the lower that value, the more computationally expensive the algorithm is.

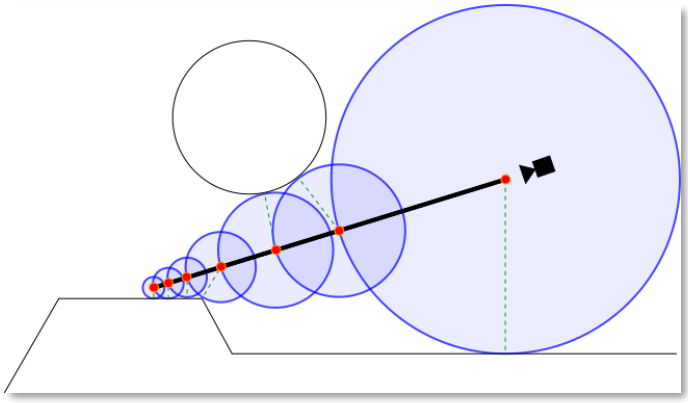

Figure 2: Ray Marching [4]

A key drawback to the ray marching approach is that - as mentioned previously - it does not find the exact intersection point. For most purposes, however, an approximation is acceptable.

Some pseudocode for ray marching:

```
distance = 0
while(ray has not gone too far & has not yet hit)
    step = distance * ray direction
    distance += step - radius - displacement(step)
intersection = distance * ray direction.
```

Implementation:

Given a ray in the coordinate space defined by the object's origin, the ray's distance from the sphere can be found be taking the magnitude of the ray's origin vector and subtracting the sum of the radius of the underlying sphere and the displacement at the point on the sphere the ray passes through, which is found using the equations provided in **1.3**.

1.6 Displacement Sphere Normals:

The normals of an intersection are normally found by taking the cross product of the derivates of position with respect to the **u** and **v** parameters of the sphere (confusingly, this u and v are not the u and v of UV mapping). It is necessary approximate these derivates of position, as there is not a known function for p(u,v) as there is with a regular sphere. Thus, the finite difference method is used [6].

Given a point of intersection of the displaced sphere, we can map that point (x,y,z) onto the underlying sphere by subtracting the displacement at that point. We can find the u and v parameters using the following equations [7]:

$u = \Phi/\Phi_{max}$,

$v = (\Theta - \Theta_{min}) / (\Theta_{max} - \Theta_{min})$.

We can then add a small amount (epsilon) to these values, and recalculate the values of Φ and Θ, then calculate the (x,y,z) of this small change in u and v.

dpdu and dpdv is then found by dividing the difference between the found points by epsilon.

We then calculate the normals by taking the cross product of these two values.

S1832137

User Guide:

2.1 Introduction:

This section of the report describes how to use the extension to PBRT-v3.

2.2 Parameters:

The Displaced Sphere takes 3 parameters: '**maxdispl'**, **'radius'**, and **'displacementmap'**.

maxdispl - float. The maximum percentage of the radius a point can be displaced from the radius. For example, a maxdispl of 15 would denote a maximum displacement of 15% of the radius.

radius - float. The radius of the underlying sphere.

displacementmap - string. A string pointing towards the relative file position from pbrt.exe of a .ppm file for reading.

2.3 Documentation:

The DisplacedSphere's functions are separated into three categories: sphere functions - denoting functions that have not been changed from sphere.cpp, modified functions - denoting functions that have been altered from sphere.cpp but carry the same name, and dsphere specific functions - denoting functions exclusive to the displacement sphere.

The **unmodified** functions are as follows:

Area, Sample, Sample, PDF and SolidAngle.

As these functions are not necessary for a displacement sphere, they will not be discussed further.

The **modified** functions are as follows:

Intersect, IntersectP, ObjectBound.

These functions are discussed in section **2.4**.

Finally, the **added** functions are as follows:

RayMarch, DistanceToDsphere, Displacement, DisplacementMap, DisplacementAnalytical, VectorToUV, ReadPPM.
These functions are discussed in section **2.5**.

2.4 Modified Functions:

2.4.1 `Bounds3f Dsphere::ObjectBound() const`

The `ObjectBound` function has been altered to include the maximum displacement. Without this change the renderer will only render within the underlying sphere, and the displacement will not be shown in the resulting image.

2.4.2 `bool Dsphere::IntersectP(const Ray &r, bool testAlphaTexture) const`

The `IntersectP` function has been changed totally. It will now pass a `bool` into `RayMarch` and returns that `bool` which will be true if `RayMarch` has detected an intersection, otherwise false.

2.4.3 `bool Dsphere::Intersect(const Ray &r, Float *tHit, SurfaceInteraction *isect, bool testAlphaTexture) const`

```
// As in sphere.cpp, we check to see if a ray hits sphere
// with radius = radius + maxDisplacement
EFloat ox(ray.o.x, oErr.x), oy(ray.o.y, oErr.y), oz(ray.o.z, oErr.z);
EFloat dx(ray.d.x, dErr.x), dy(ray.d.y, dErr.y), dz(ray.d.z, dErr.z);
EFloat a = dx * dx + dy * dy + dz * dz;
EFloat b = 2 * (dx * ox + dy * oy + dz * oz);
EFloat c = ox * ox + oy * oy + oz * oz - EFloat(radius + maxDisplacement) * EFloat(radius + maxDisplacement);

// Solve quadratic equation for _t_ values
EFloat t0, t1;
if (!Quadratic(a, b, c, &t0, &t1)) return false;

EFloat tShapeHit = t0;
if (tShapeHit.LowerBound() <= 0)
{
    tShapeHit = t1;
    if (tShapeHit.UpperBound() > ray.tMax) return false;
}
```

The `Intersection` function begins by using the same logic as sphere.cpp to determine whether or not the ray will pass through the area the displacement sphere may occupy. This is a small optimisation step to save us from raymarching where the sphere can never be [7].

```
bool intersect;
*tHit = RayMarch(ray, &intersect);
if (intersect)
{
```

Next, we pass a `bool` into `RayMarch`, and assign the distance returned by `RayMarch` to the float `*tHit`. We check whether or not `RayMarch` detected an intersection, and if it has we move onto the code within the `if` statement.

The three hit variables store information about the point at which we hit the displacement sphere.

```
// Hit variables:
Point3f  pHit = ray(*tHit);
Vector3f vHit = Vector3f(pHit);
Vector3f rHit = vHit - (Displacement(vHit) * Normalise(vHit));
```

`pHit` and `vHit` store the point at which we hit the sphere (with `vHit` storing it as a `Vector3f`), while `rHit` stores the point on the underlying sphere corresponding to `pHit` which is found by subtracting the point by the amount of displacement at that point.

The rest of the code within the `if` statement is dedicated to finding the normals, as discussed in section **1.6**.

The function then returns the `bool` that was passed into `RayMarch` which will be true if an intersection was detected, otherwise false.

2.5 Added Functions:

2.5.1 `Float Dsphere::RayMarch(Ray ray, bool* intersect) const`

The `RayMarch` function implements the described behaviour in section **1.5**. It follows the logic found at [8]

We can safely determine that the length of a single step should never be larger than the distance from the camera to the sphere, else the ray has moved away from the sphere, so we can initialise the variable `max` to be equal to the magnitude of the ray's origin. We add a small value (epsilon) to guard against any floating point issues. We need to choose an acceptably low number for the minimum distance, which if the step distance falls below we classify the ray as hitting the sphere. This value, stored in the global `float minDist` was set to its current value by observing the behaviour of the algorithm and determining a value that gave the best results.

We then use the raymarching algorithm to find the approximate distance along the ray the intersection is and set the `bool` to true if the step distance fell below the minimum, otherwise false.

2.5.2 `Float Dsphere::DistanceToDsphere(Vector3f vector) const`

The `DistanceToDsphere` function takes in a vector and returns the distance from the end of that vector to the sphere.

2.5.3 `Float Dsphere::Displacement(Vector3f vector) const`

The `Displacement` function will determine whether to pass the input vector to `DisplacementAnalytical` or `DisplacementMap` depending on the value of the `bool analytical`.

2.5.4 `Float Dsphere::DisplacementMap(Vector3f vector) const`

The `DisplacementMap` function takes in a vector, and returns the displacement of the sphere at the point on the sphere that lies on the line between the sphere's centre and the vector's end, determined by the UV map read in at the DisplacedSphere's creation.

First, it calculates the **u** and **v** coordinates on the UV map using the equations described in section **1.3**. These values are between 0 and 1, so we must multiply them by the image dimensions and round to the nearest integer so we can index the image correctly.

This rounding loses some information, however. The UV values - before rounding - will fall between two pixels. We can take the weighted sum of these two pixels by multiplying by the difference between the rounded value and the unrounded value by the first pixel, and one minus this difference by the second. This step is entirely optional, but may help smooth the UV map for low resolution heightmaps.

2.5.5 `Float Dsphere::DisplacementAnalytical(Vector3f vector) const`

The `DisplacementAnalytical` function takes in a vector, and returns the displacement of the sphere at the point on the sphere that lies on the line between the sphere's centre and the vector's end, determined by the function given in the coursework document.

It calculates the **u** and **v** coordinates on the analytical UV map using the equations described in section **1.3**.

2.5.6 `Float* Dsphere::VectorToUV(Vector3f input) const`

The `VectorToUV` function takes a vector and returns it's UV map coordinates as described in section **1.3**.

2.5.7 `void ReadPPM(const std::string &fname)`

The `ReadPPM` function follows the logic found at [9]. It takes the image data and stores it into a 1d array storing the greyscale values of the image. Additionally, it performs a gaussian blur across the image in order to eliminate noise.

S1832137

Evaluation:

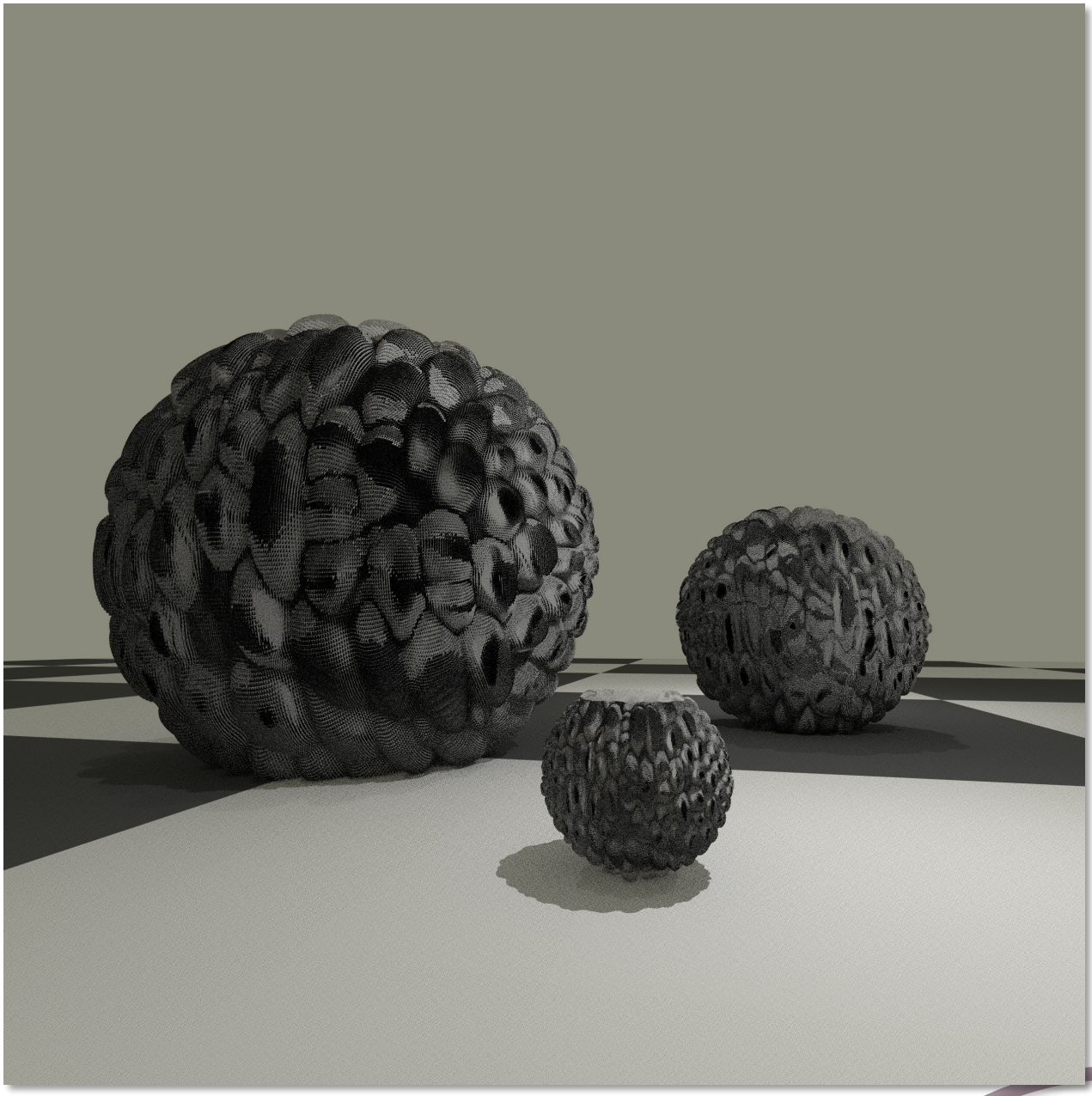

Figure 3: Final Scene "finalscene(.exr.pbrt)".

S1832137

Evaluation:

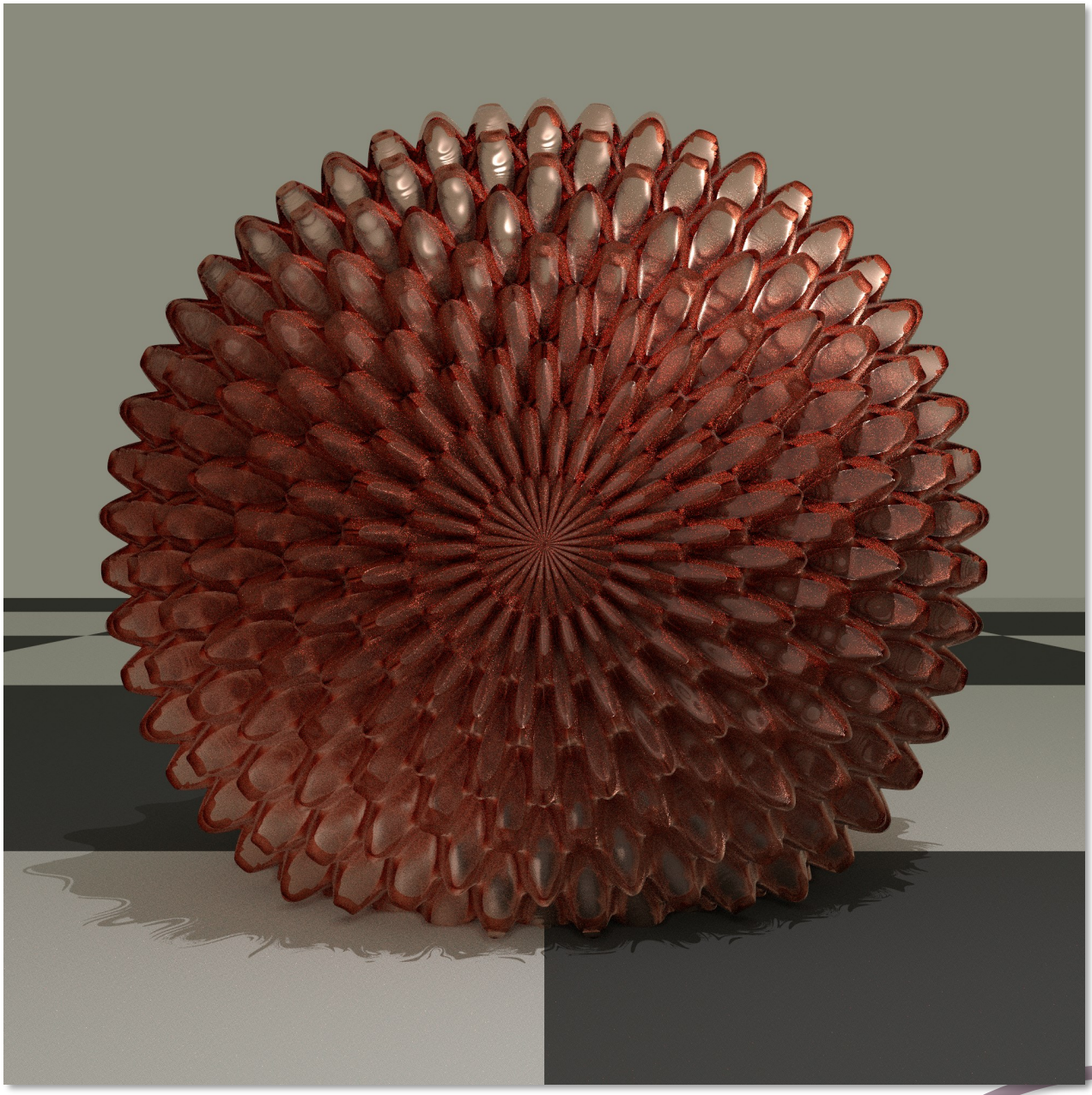

Figure 4: Metal-Sphere (analytical solution)

"metal-sphere(.png/.pbrt)".

S1832137

Evaluation:

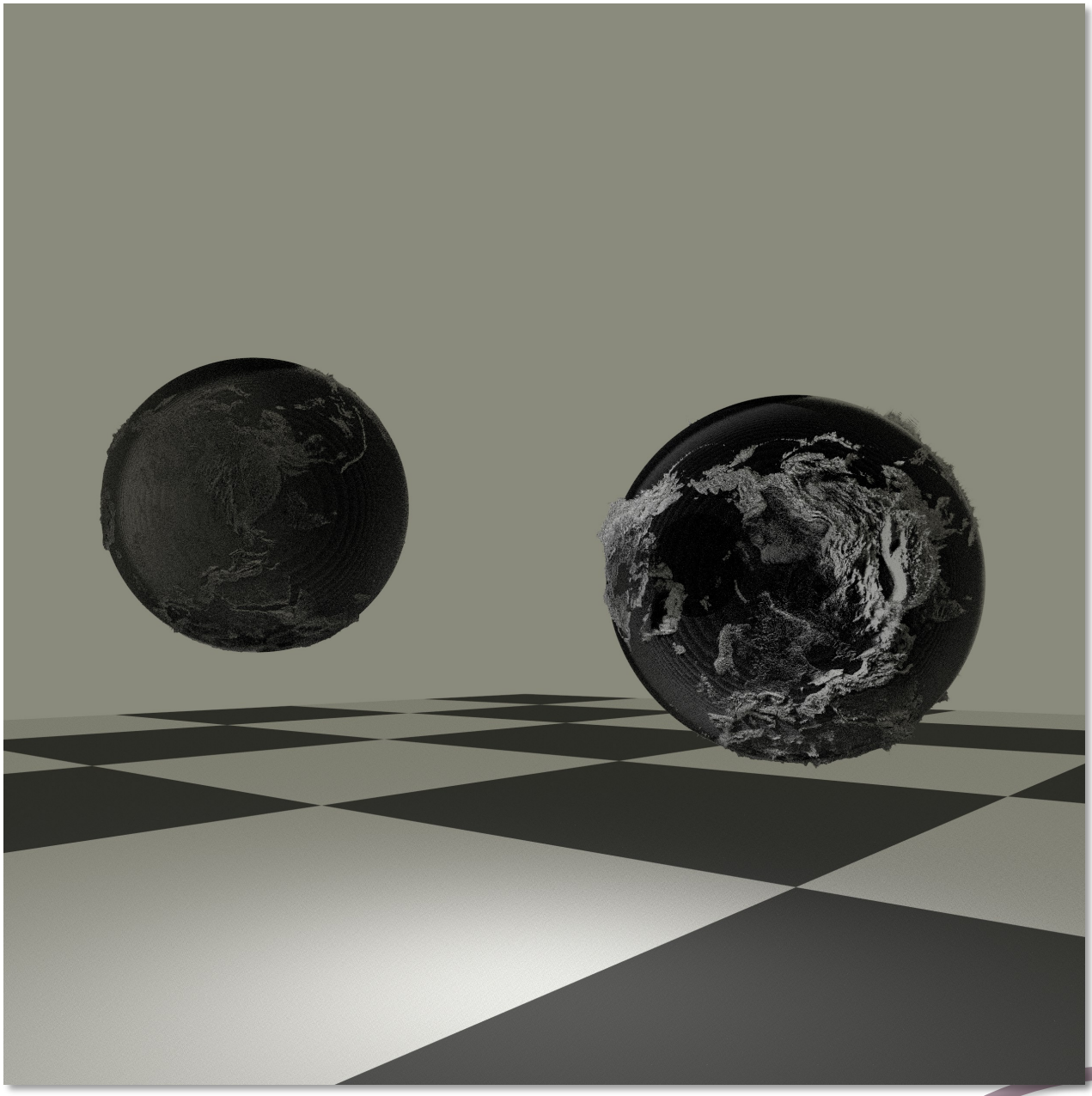

Figure 5: Big-earth-little-earth

"earth(.png.pbrt)".

S1832137

Evaluation:

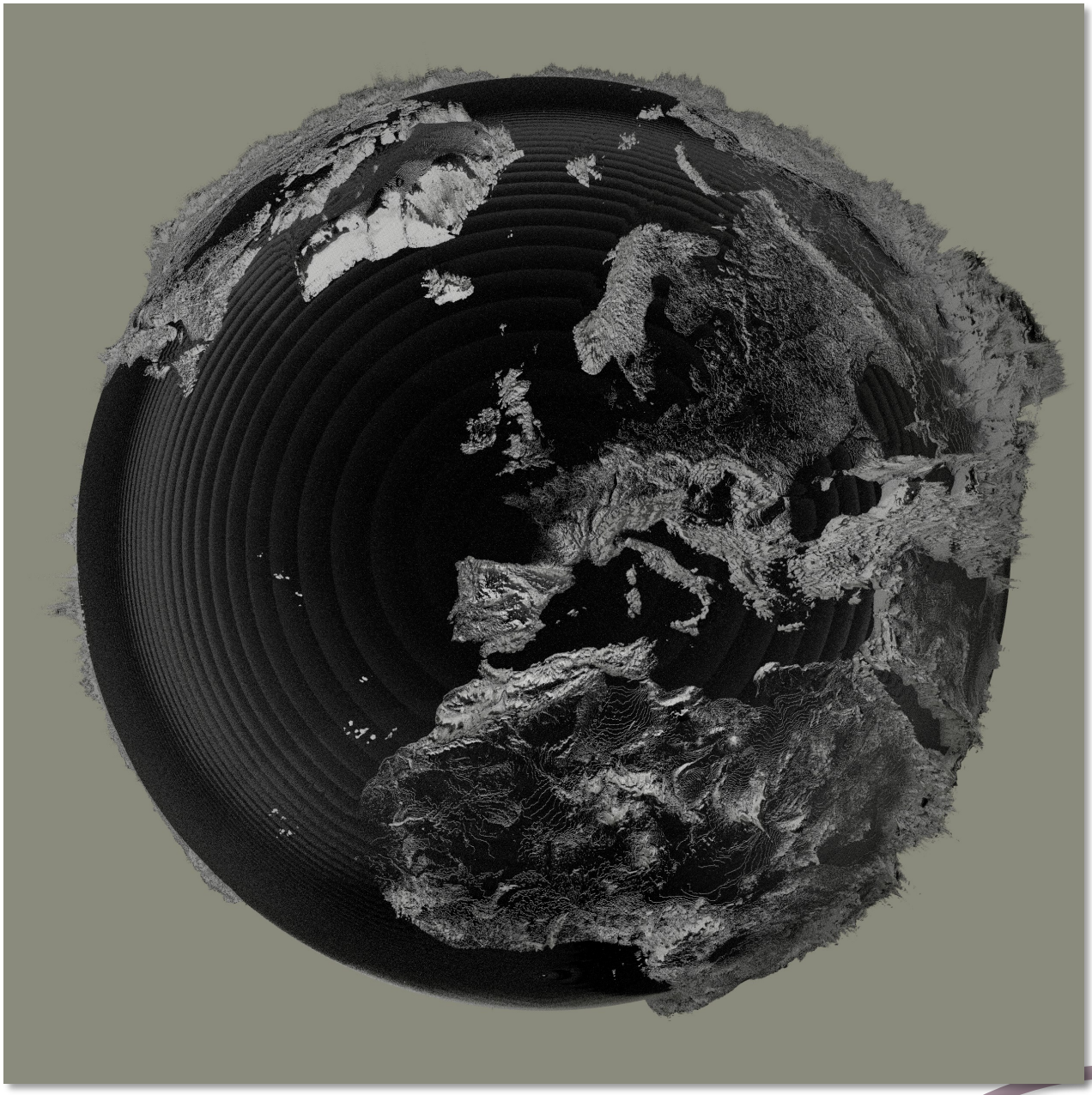

Figure 6: Europe

“europe(.png/.pbrt)”.

S1832137

Evaluation:

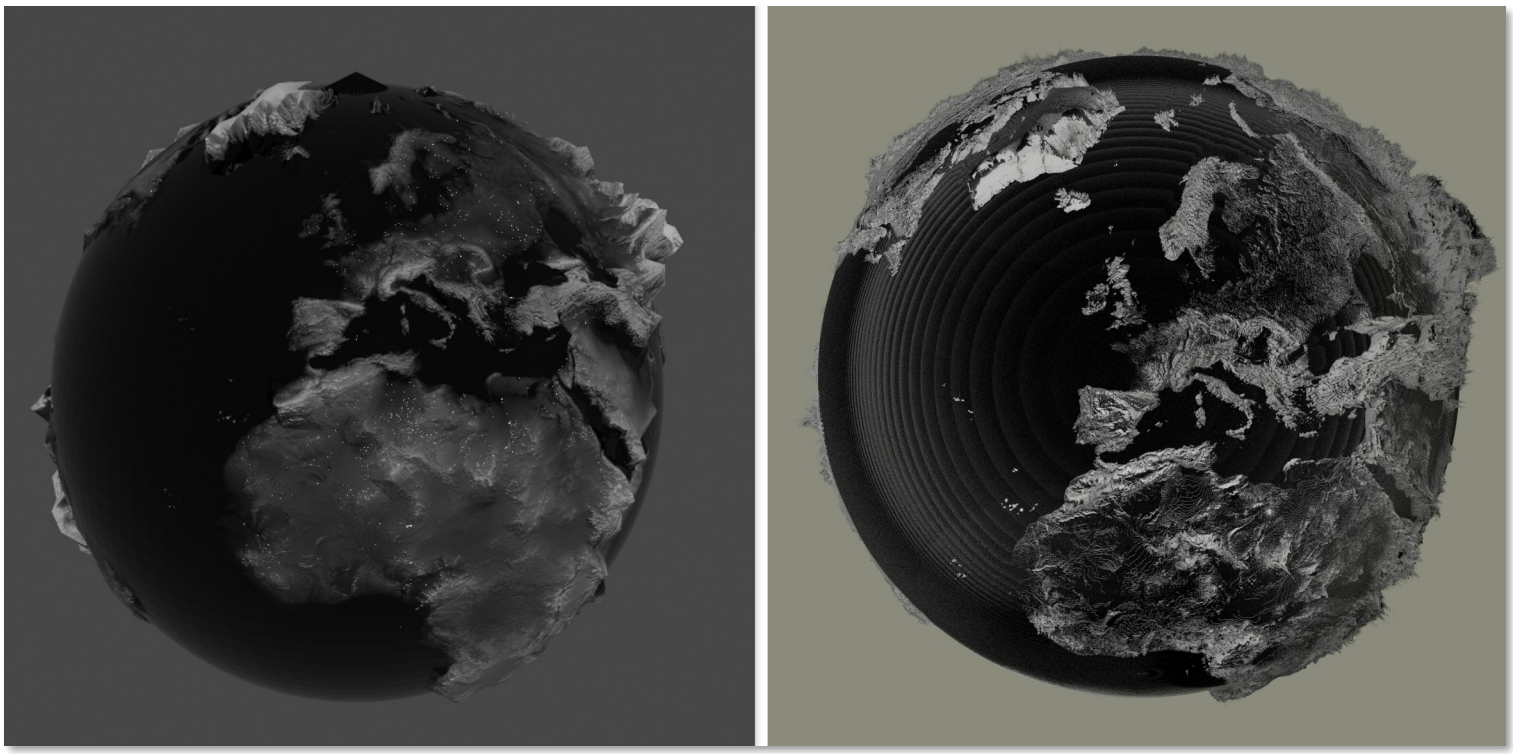

Figure 7: Blender Cycles Engine (left) vs PBRT-v3 DisplacedSphere render (right)

Evaluation:

3.1 Image Discussion:

Figure 3 displays the three key aspects of the Displaced Sphere: occlusion, normals and silhouettes.

For occlusion and silhouettes, we see that the ray-marching algorithm correctly renders the sphere. The edges are well defined, and there is a cohesive and correct looking shadow.

We can see a white area on the top of one of the spheres. I am unsure as to what caused this, but I suspect it is due to the sphere's proximity to the light in the scene.

The scene shows that the sphere needs some anti-aliasing. I suspect this could be done by extrapolating the rgb output of the ppm file into a longer array. That is, create intermediate points between two pixels that linearly interpolate between the two values those pixels hold.

It is best to evaluate the normals using Figure 7, where we see a side by side comparison with the Blender Cycles Engine rendering the same map. We can see the PBRT-v3 solution matches well with the normals of the Cycles Engine, with any differences seen being attributable to different lighting. The figure does show concentric rings moving through the PBRT-v3 render, but I suspect this is caused by the proximity of the sphere to the light source in the scene.

In Figure 4, the analytical solution is used. We can see some interesting shadows. This effect only happens with the analytical solution, and I suspect it can be explained by the nature of the function used. The general shape of the shadow appears correct.

3.2 Optimisation of Intersection Test:

The intersection test was optimised by optimising the values of:

- The minimum and maximum distance a ray can travel.
- The step distance for each step.
- Removing rays that will never intersect with the sphere.

As previously mentioned, the raymarching intersection test was optimised by first ruling out any rays that will never pass through the area the Displaced Sphere using the quadratic solutions displayed in sphere.cpp. In testing I determined this change resulted in a 16% decrease in rendering time. This value of course depends on the amount of the scene the Displaced Sphere occupies.

A further optimisation step that could be used is to begin the raymarching at the first solution of the quadratic - where the ray hits the sphere of radius equal to the radius of the underlying sphere plus the maximum displacement. However this did not work in testing for unknown reasons.

Raymarching depends on the step distance, the minimum distance to the sphere and the maximum distance a ray can travel.

A greater step distance would result in less steps being taken, meaning a faster algorithm. The step distance should never be greater than the distance from the ray to the sphere, however, as any larger could result in the ray overshooting its intersection.

An acceptable minimum distance was found using observation. I sought the lowest value possible while maintaining acceptable rendering times, as the more accurate the intersection, the more accurate the final render. The resulting value is remarkably low, at $3e^{-6}$. You can still notice some distortions and missed intersections with high displacements, unfortunately. Much lower began to noticeably impact rendering times.

The maximum distance metric was replaced by asserting that a single step should never be greater than the distance from the camera to the sphere. We can safely assume that a ray has missed if it begins to overstep this value. This technique is preferable to an arbitrary value, as it is dynamic - allowing the Displaced Sphere to be arbitrarily close or far without taking a performance penalty.

S1832137

References:

1. https://github.com/mmp/pbrt-v3
2. https://en.wikipedia.org/wiki/Displacement_mapping
3. https://michaelwalczyk.com/blog-ray-marching.html
4. https://adrianb.io/img/2016-10-01-raymarching/figure3.png
5. https://en.wikipedia.org/wiki/UV_mapping
6. https://en.wikipedia.org/wiki/Finite_difference_method
7. https://www.pbr-book.org/3ed-2018/Shapes/Spheres
8. https://www.shadertoy.com/view/4ldGzB
9. https://github.com/sol-prog/Perlin_Noise/blob/master/ppm.cpp