Sphere Tracing, Distance Fields, and Fractals

Alexander Simes

Advisor: Angus Forbes
Secondary: Andrew Johnson

Fall 2014 - 654108177

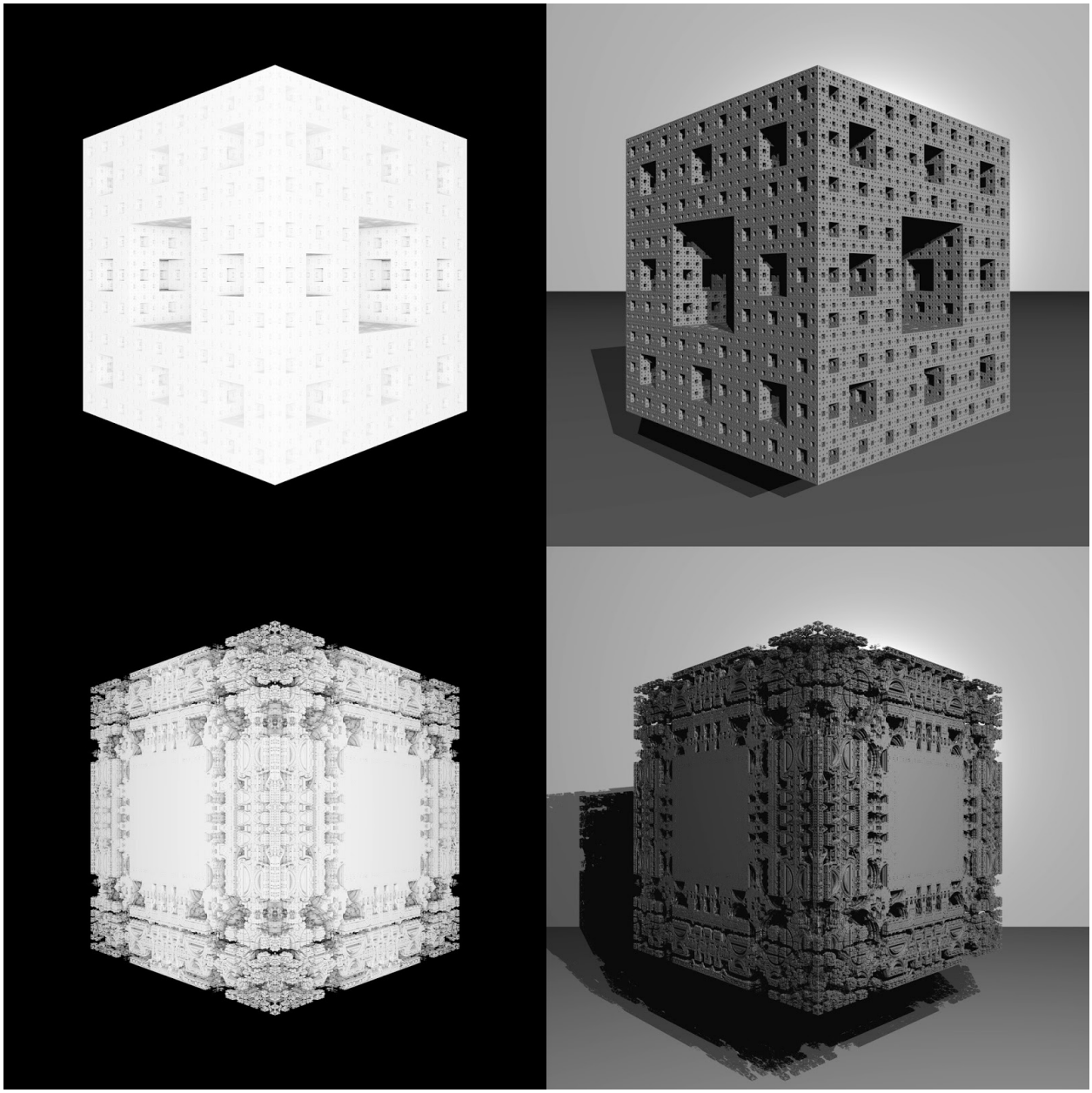


Figure 1: Sphere Traced images of Menger Cubes and Mandelboxes shaded by ambient occlusion approximation on the left and Blinn-Phong with shadows on the right

Abstract

Methods to realistically display complex surfaces which are not practical to visualize using traditional techniques are presented. Additionally an application is presented which is capable of utilizing some of these techniques in real time. Properties of these surfaces and their implications to a real time application are discussed.

Table of Contents

1 Introduction

Sphere Tracing is a rendering technique for visualizing surfaces using geometric distance. Typically surfaces applicable to Sphere Tracing have no explicit geometry and are implicitly defined by a distance field. Distance fields define volumes and a Sphere Tracer visualizes the surfaces of these volumes.

A general Sphere Tracing application was created for SAGE2 (Scalable Amplified Group Environment) to visualize distance fields on high-resolution multiple display devices. SAGE2 is a browser based middleware that works in concert with many visualization applications, enabling users to simultaneously access, stream, and juxtapose multiple high-resolution visualizations on high-resolution tiled display walls. SAGE2 allows the presented application to be supported by any device with a browser and GPU in addition to coordinating multiple instances of the application.

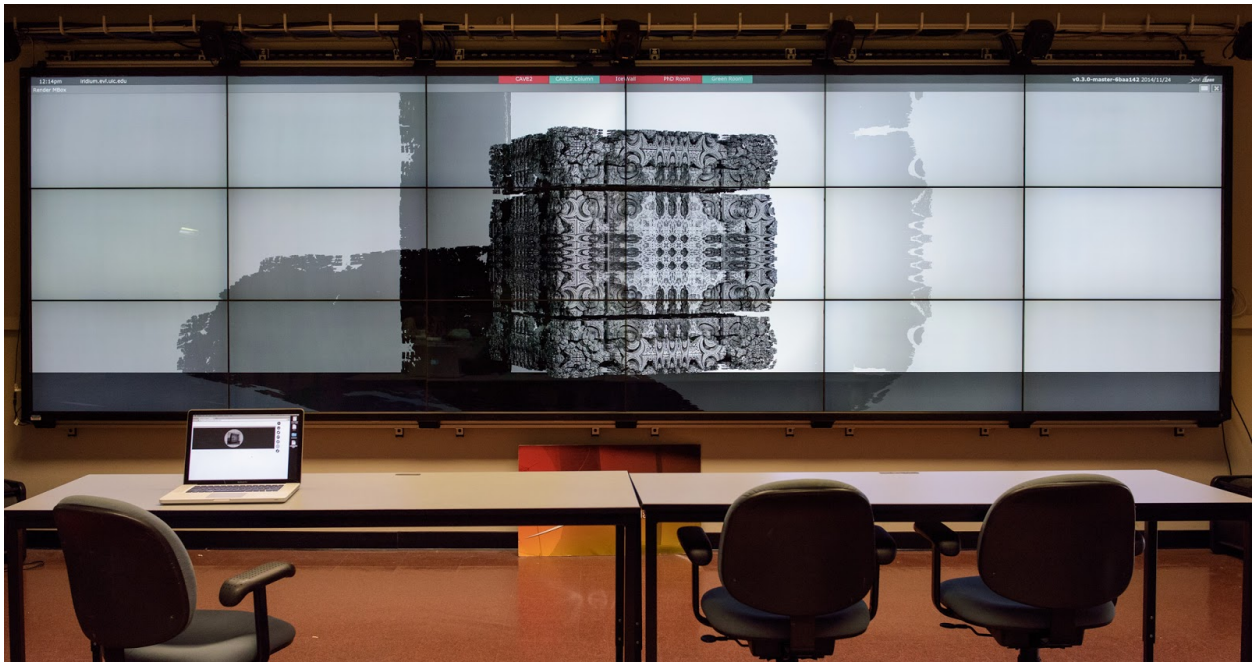


Figure 2: Instances of the application running on a high-resolution display wall. The display wall is controlled by six computers with separate GPUs. Photo provided by Lance Long

Each instance of the application receives global information from SAGE2 regarding the relative positions and resolution of the displays in JavaScript. Application instances use this information to manipulate display specific HTML canvases. Additionally application instances compile WebGL GLSL shaders which then display the Sphere Tracing content to the HTML canvases. Although this process is targeted towards simultaneously displaying content to multiple tiled displays it can also be used for a single display device.

2 Minimal CPU Sphere Tracing Model

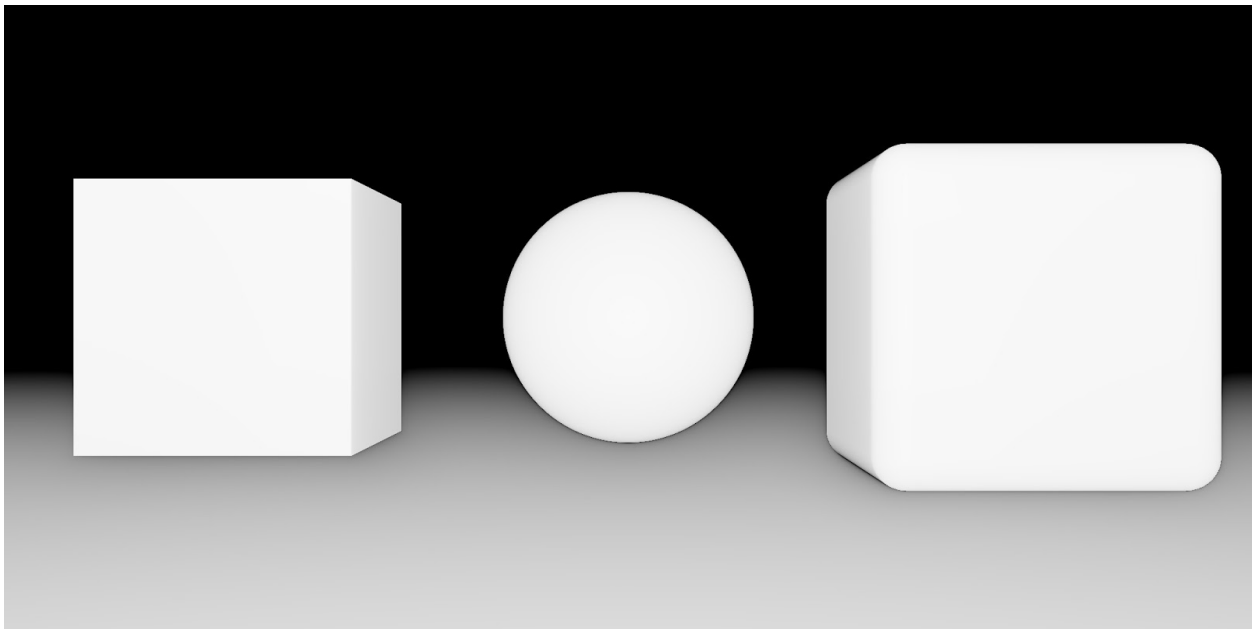


Figure 3: A Sphere Traced scene composed of simple Euclidean objects shaded by ambient occlusion approximation

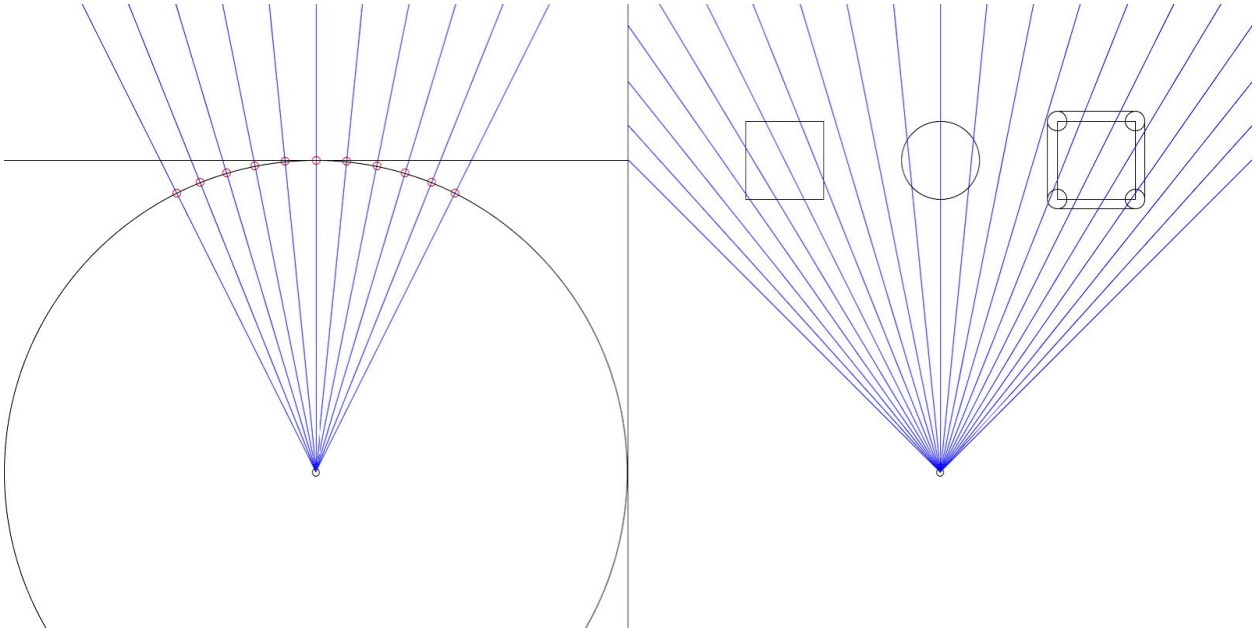
2.1 Camera Model

A simple virtual camera model is presented which will be generalized. This model requires a single three component vector to represent the camera's position under the assumption that the camera is viewing along the z-axis. This model uses a logical virtual screen containing m*n pixels one unit in front of the camera. Camera direction vectors are then calculated for each pixel ij as:

```
dir.x = i * (1.0 / width) - 0.5;
dir.y = j * (1.0 / height) - 0.5;
dir.z = 1.0;
```

Each of these direction vectors are then normalized. **Figure 4** shows a top down diagram for a camera where m is 11: the virtual screen is represented as the horizontal black line, the blue lines are the ij camera direction vectors, and the red circles represent the normalized ij camera direction vectors. This calculation assumed that *width* and *height* are equal but can be modified for a non-square virtual screen by instead calculating *dir.y* as:

```
dir.y = (j * (1.0 / height) - 0.5) * (height / width);
```



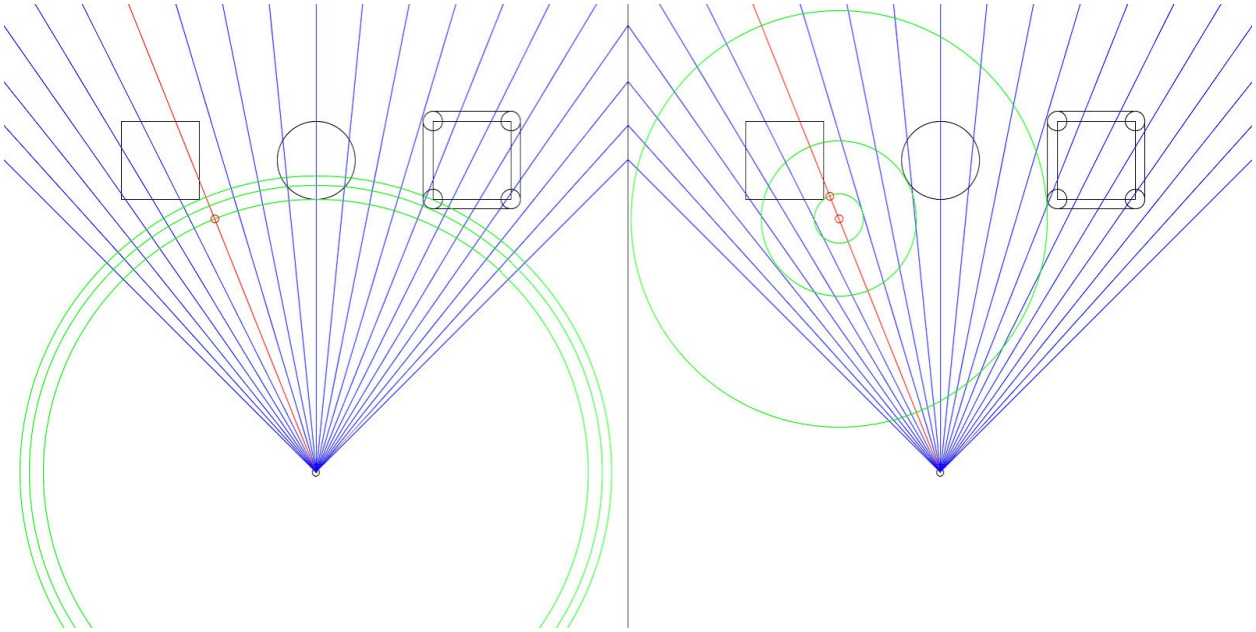
Figures 4 and 5: Top down diagrams of a camera for a 11 pixel wide screen on the left and the simple Euclidean scene on the right

Viewing in a direction that is not aligned with the z-axis requires rotating the normalized camera direction vectors about the camera position to face an arbitrary direction. Finally near and far clipping spheres are still needed and are later discussed.

2.2 Marching with Distance Fields Introduction

Each pixel will be shaded based on whether the corresponding ij camera direction vector points to a distance field within the scene or not. The simple Euclidean scene in **Figure 5** has a few distance fields including a box, a sphere, and a roundbox (the ground plane is omitted). Each distance field has a corresponding distance function which returns the geometric distance between some point to the distance field's implicitly defined surface.

Without explicit geometry exact surface intersections cannot be found in general. Instead the distance functions are used to approach the underlying implicit surface in iterations. **Figures 6** and 7 show the first two iterations of some ij camera direction vector as an example. A position vector is created as a copy of the camera position which will be referred to as the marching point. During the first iteration the distance from the marching point to each distance field is found using each's distance function. The minimum of these distances represent a sphere around the marching point which is guaranteed to not contain any implicitly defined surfaces. In **Figure 6** the distance to the sphere is the smallest distance and so the marching point advances by that amount along the corresponding ij camera direction vector. During the second iteration the distance from the marching point's new position to each distance field is found. In **Figure 7** the distance to the box is the smallest distance and so the marching point advances by that amount.



Figures 6 and 7: Top down diagrams of the first and second marching iterations of the red ij camera direction vector. The green circles represent the distance from the marching point to each implicit surface

In this example the marching point is approaching the surface of the box and it will continue to iterate until the distance is less than some minimum distance ε. The value of ε is arbitrary but serves the purpose of placing a limitation on marching iteration. For reference the box in **Figure 3** has a length of 1.0 and the value used for ε was 0.0001. The position of the marching point after being within a distance of ε to an implicit surface is an approximation of the surface intersection for the corresponding ij camera direction vector.

2.3 Ambient Occlusion Approximation

In this minimal Sphere Tracing model marching iterations are limited to a maximum of 255 iterations, the choice of 255 is arbitrary but will be later explained. If the marching point is not within a distance of ε after 255 iterations then no surface intersection approximation is made and the corresponding ij camera direction vector is considered to not be pointing to a distance field. It is possible for this assumption to be incorrect which visually results in holes of the overall image.

Finally one additional limitation is placed on marching iteration which is the far clipping sphere. If the current distance from the marching point to the camera position is ever greater than a user specified clipping value then the corresponding ij camera direction vector is considered to not be pointing to a distance field. This limitation serves the purpose of ending iteration when an ij camera direction vector would waste computation time iterating in a direction with no distance fields.

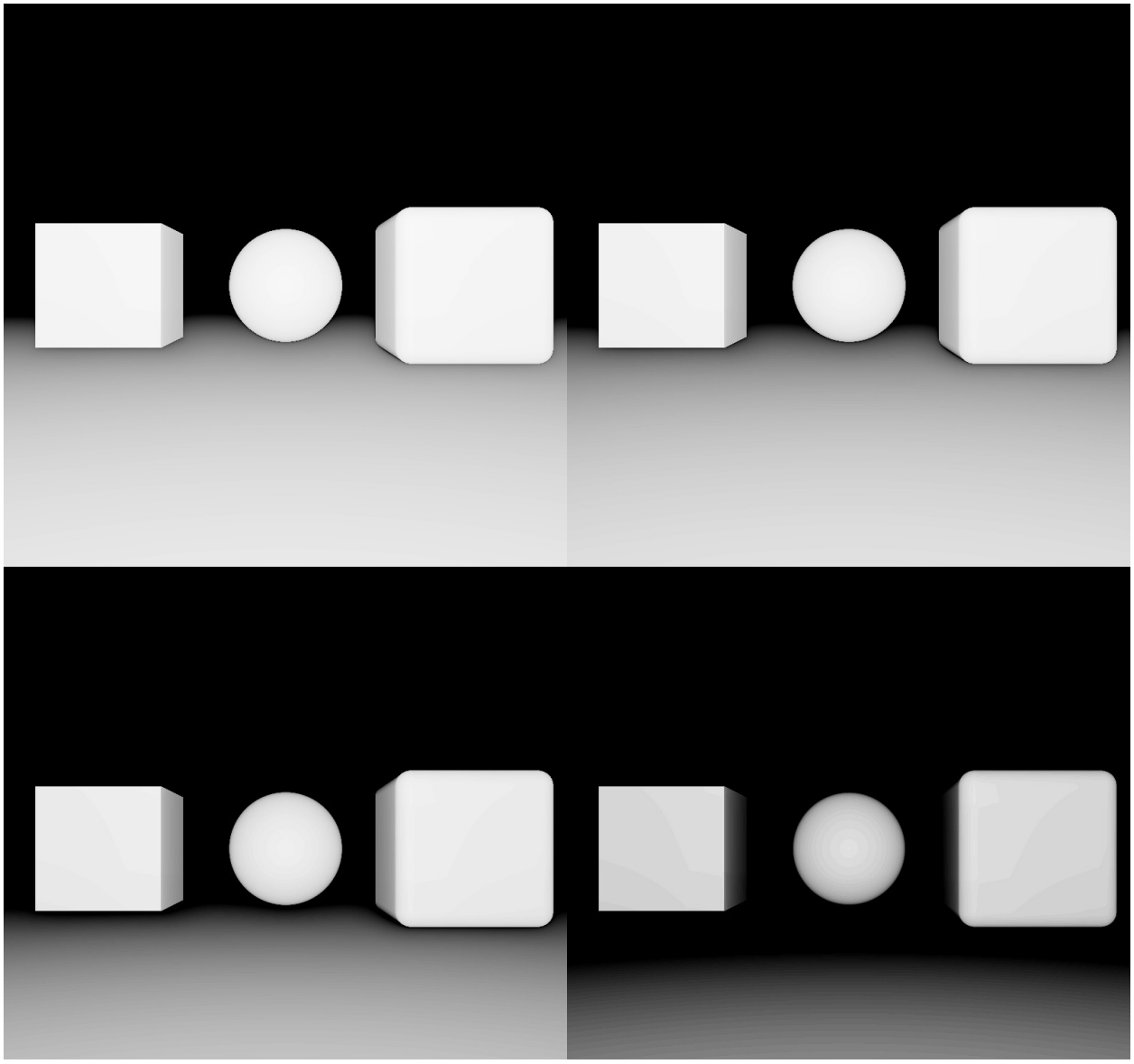


Figure 8: The simple Euclidean scene shown with maximum marching iterations of 200, 150, 100, and 50

The choice of 255 maximum marching iterations was chosen due to pixels being able to represent 255 values per channel. Under this limitation a shading value for some ij pixel can be calculated as:

val = (255.0 - iterations) / 255.0;

Due to the distance traveled reducing per iteration when the marching point approaches a surface this shading results in an effect similar to ambient occlusion. However it is not truly accurate ambient occlusion, it is only an approximation. **Figure 8** demonstrates visual artifacts related to this approximation by reducing the maximum number of marching

iterations. The most noticeable effects come from an ij camera direction vector being nearby a distance field without reaching a distance less than ε. Additionally banding becomes more apparent at lower maximum iterations.

3 Distance Fields

Iñigo Quilez provides GLSL code for several Euclidean distance fields and operations on distance fields. Understanding how a distance field represents an implicit surface is rarely a simple mental exercise and so only two simple distance fields will be explained. In addition Quilez provides several operations that can be performed on distance fields which will be explained before covering iterative distance field functions. The code examples below are modifications to the functions Quilez provides for clarity.

3.1 Signed Sphere

Quilez uses a three component position vector in his sample functions which represents the position of the marching point minus the center of a distance field. Below *radius* is the radius of an implicitly defined sphere:

```
float signed_sphere(vec3 position, float radius) {
    return length(position) - radius;
}
```

The above length function then represents the distance between the marching point and the center of the sphere. Subtracting the radius from this distance returns the distance from marching point to the implicit surface of the sphere if the marching point is outside the sphere. If the marching point is inside the sphere then a negative distance is returned and so this function returns a signed distance. Signed distance will later be convenient for Blinn-Phong shading.

3.2 Unsigned Box

The *position* vector below has the same meaning as before. The implicitly defined box requires a width, height, and depth which the three components of vector *edgeLength* represents:

```
float unsigned_box(vec3 position, vec3 edgeLength) {
    return length(max(abs(position) - edgeLength), 0.0));
}
```

The above abs function places the position vector in the positive quadrant. This is done before subtracting each component of *edgeLength* from the corresponding component of position to maximize the values. If any of these values become negative after the subtraction then the max function will replace the value with 0.0. Finally the length of the currently calculated vector is found which represents the distance to the implicitly defined box if the marching point is outside the box. If the marching point is inside the box then 0.0 is returned and so this function returns an unsigned distance. Unsigned distances will later require backstepping for Blinn-Phong shading in order to create a surface normal.

3.3 Distance Field Operations

Union: The union of any number of distance fields can be found by taking the minimum of their distance functions. The union of all distance fields in a scene is the world distance function and is the safe amount for a marching point to advance during marching iteration.

Intersection: The intersection of two distance fields can be found by taking the maximum of their distance functions.

Subtraction: Some distance field, “A”, can be subtracted from another distance field, “B”, by taking maximum returned values of negative “A” and “B” (pseudo code):

float subtraction = max(-A, B);

Repetition: Distance functions can be tiled infinitely at nearly no computational expense. The “trick” is to take the modulus of the position vector before using it as the input of a distance function. A three component vector representing the modulus of each dimension is created and applied as:

```
float repetition(vec3 position, vec3 modulus) {
    return primitive( mod(position, modulus) - 0.5 * modulus );
}
```

In the above code the function primitive is a placeholder for the appropriate distance function. For example, primitive may be a sphere distance function, box distance function, etc.

Rotation: Distance fields are world aligned and so are not rotated. Instead the marching point is rotated about a distance field to give the illusion of the distance field rotating. First the distance field is translated to the origin and the marching point is translated by the same amount. The marching point is rotated about the origin and the distance field’s distance function is called capturing a snapshot of the implicit surface from a new angle. The distance field and marching point are then placed back to their original locations. This last step could have been avoided by instead having used copies of the marching point and distance field.

4 Blinn-Phong Shadow Sphere Tracing Model

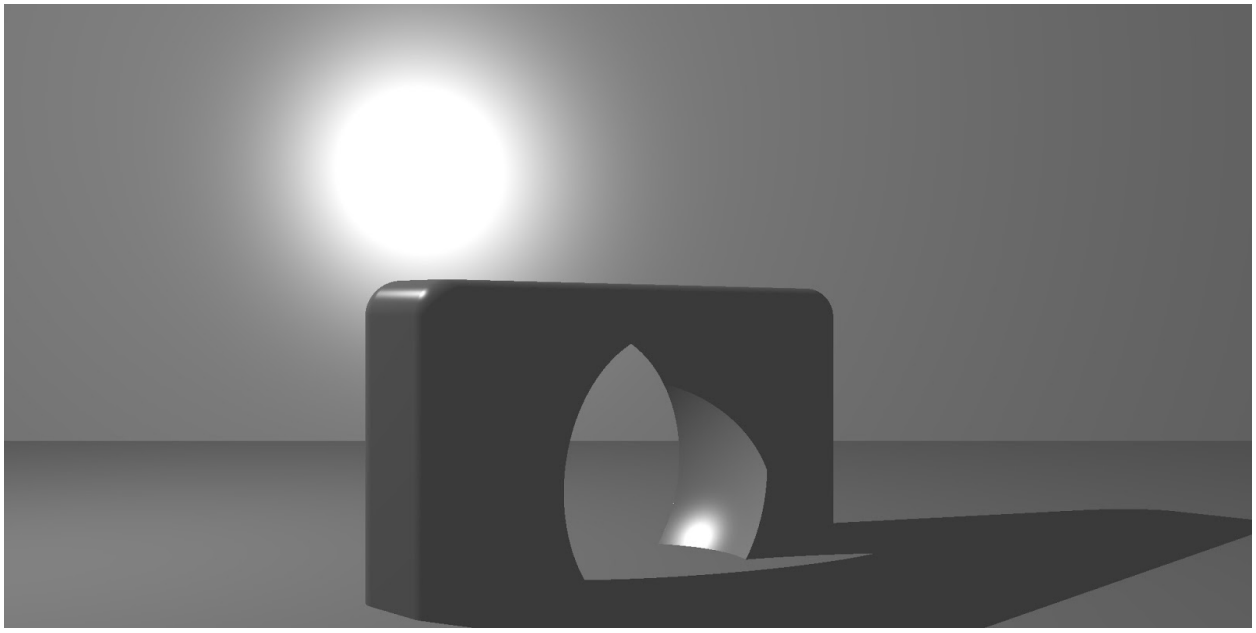


Figure 9: A Sphere Traced scene using Blinn-Phong shading in addition to hard shadows

4.1 Scene Composition

The scene depicted below in **Figure 10** consists of three distance fields: a roundbox, a sphere, and a plane. The roundbox has been rotated by 45° and the sphere has been subtracted from the rotated roundbox.

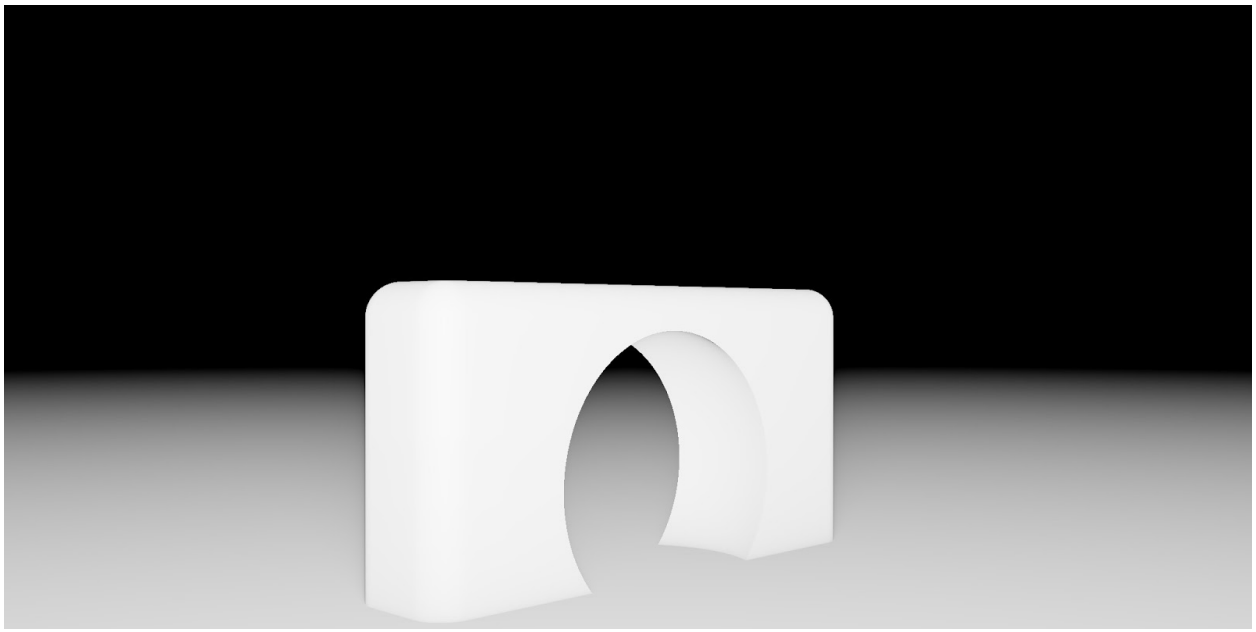


Figure 10: A scene shaded by the Minimal CPU Sphere Tracing Model

The world distance function for this scene would then be (pseudo code):

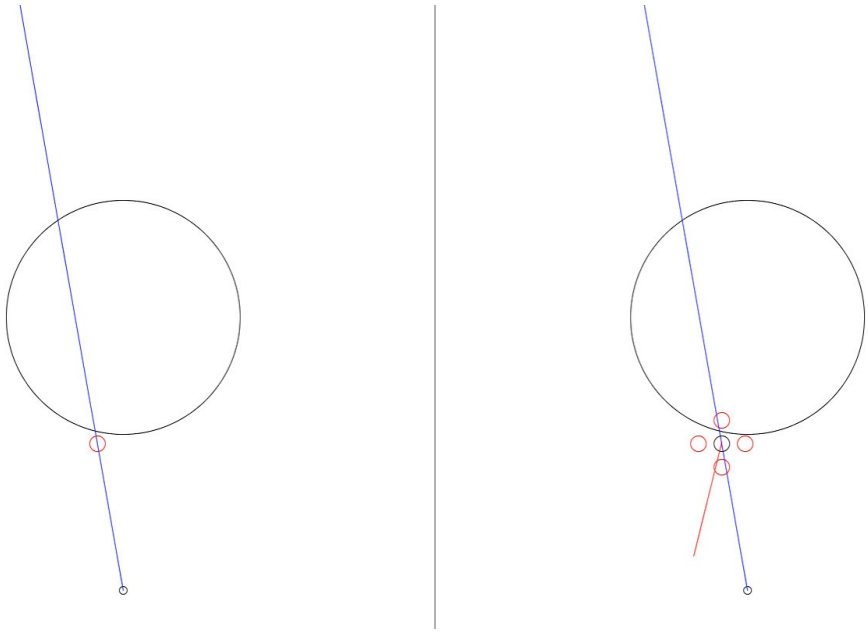
float worldDistance = max(rotateY(roundbox), -sphere);
worldDistance = min(worldDistance, plane);

4.2 Maximum Marching Iteration Limitation

Imposing a limitation to the maximum number of marching iterations in the Minimal CPU Sphere Tracing Model served the purpose of making ambient occlusion approximation shading possible. However this limitation will be removed to avoid potentially stopping iteration before reaching an implicitly defined surface along some ij camera direction vector. Without this limitation only the minimum distance ε and maximum distance far clipping sphere remain to end iteration. Therefore each ij camera direction vector either will end iteration within a distance ε of an implicit surface or safely not reach an implicit surface in which case the corresponding pixel will be shaded black (**Figure 10** does not have a backplane but **Figure 9** does so that each pixel shades a distance field). These two conditions are sufficient to avoid infinite iteration but generally cause an increase in computational time per pixel.

4.3 Surface Normals

After marching iteration terminates the world distance function is tested to verify if whether or not an ij marching point is within a distance ε to an implicit surface. If it is then the marching point represents a surface intersection approximation that will be Blinn-Phong shaded. Blinn-Phong shading requires a surface normal and **Figures 11** and **12** diagram finding the surface normal for an example ij camera direction vector.



Figures 11 and 12: Top down diagrams of a single ij camera direction vector with a marching point within a distance ε to an implicit sphere surface. To the right the sphere distance function is sampled to create a surface normal (assume a signed sphere distance function)

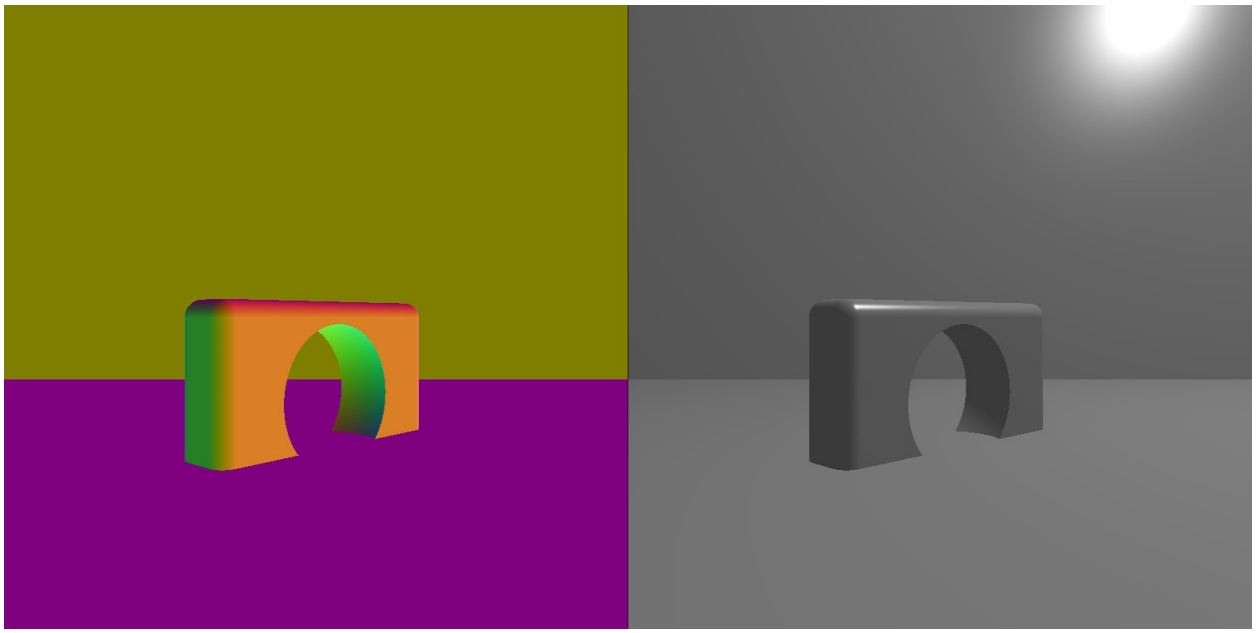
Assuming all distance functions are signed, the world distance function is sampled in the negative and positive direction of each dimension (-x, +x, -y, +y, -z, and +z). Each positive sample minus the corresponding negative sample is used to produce a vector as (pseudo code):

```
vec3 surfaceNormal = normalize(vec3(
    positiveSampleX - negativeSampleX,
    positiveSampleY - negativeSampleY,
    positiveSampleZ - negativeSampleZ
));
```

After normalizing the above vector it represents the surface normal of the nearest distance functions' implicit surface. However, if the nearest distance function was unsigned and a sample was taken from inside the implicit surface then the surface normal will be inaccurate (0.0 is returned for samples inside unsigned distance functions' implicit surface). Unsigned distance functions can typically be handled by backstepping along the corresponding ij camera direction vector before sampling.

4.4 Blinn-Phong Shading

Blinn-Phong shading typically involves three components that contribute to a pixel's shading: ambient, diffuse, and specular lighting. **Figure 14** the shows the result of applying this shading technique to the scene from **Figure 10** (a backplane has been added to the scene).



Figures 13 and 14: On the left surface normal (x, y, z) values are mapped to pixel channels (r, g, b). On the right is the resulting Blinn-Phong shading with a single point light source

Blinn-Phong shading is a common shading technique and so will not be discussed in detail. For each pixel ij an ambient value is added if the corresponding ij camera direction vector ended marching iteration within a distance ε of an implicit surface. Additionally the surface normal approximation is used in the Blinn-Phong lighting equation to calculate diffuse and specular values. Both diffuse and specular values are added unconditionally to each pixel in **Figure 14** and so there are no hard shadows.

4.5 Hard Shadows

Hard shadows are implemented by only adding diffuse and specular values to a pixel ij if the corresponding surface intersection approximation is not occluded from a light in the scene. In order to determine if a surface intersection approximation is occluded a secondary march iteration occurs for each light in the scene. These secondary marches will be referred to as light marches and originate from the surface intersection approximation.

Assume for simplicity that there is a single point light source within the scene (as in **Figure 9**) and so only a single light march. The marching point is made to be a copy of the surface intersection approximation and then moved a distance ε in the direction of the point light source. Moving the marching point a distance ε before iteration prevents termination during the first iteration of the light march unless another surface is now within a distance ε of the marching point (if another surface is now within this distance it is considered to occlude the surface intersection approximation).

The light march follows the same termination rules as the original marching iteration with the exception that its maximum distance is the distance between the surface intersection approximation and the point light source. If the marching point travels a greater distance than this then the surface intersection approximation is not occluded by another implicit surface. Otherwise the light march would have reached a distance less than ε to some other implicit surface and so the surface intersection approximation should be occluded.

4.6 Translation to GPU

Each pixel ij of the mentioned Sphere Tracing models calculated a shading value independently of the other pixel which makes translation to GLSL trivial. The initial direction of an ij camera direction vector can instead be calculated as the vec3 below and the remainder of the Sphere Tracing pipeline remains unchanged:

```
vec3 dir = normalize(vec3(
    gl_FragCoord.x / width - 0.5,
    (gl_FragCoord.y / height - 0.5) * (height / width),
    1.0
));
```

Additionally viewport cropping can be handled by adding appropriate uniform offsets to *gl_FragCoord.x* and *gl_FragCoord.y* before they are divided by *width* and *height* respectively.

5 Menger Cube

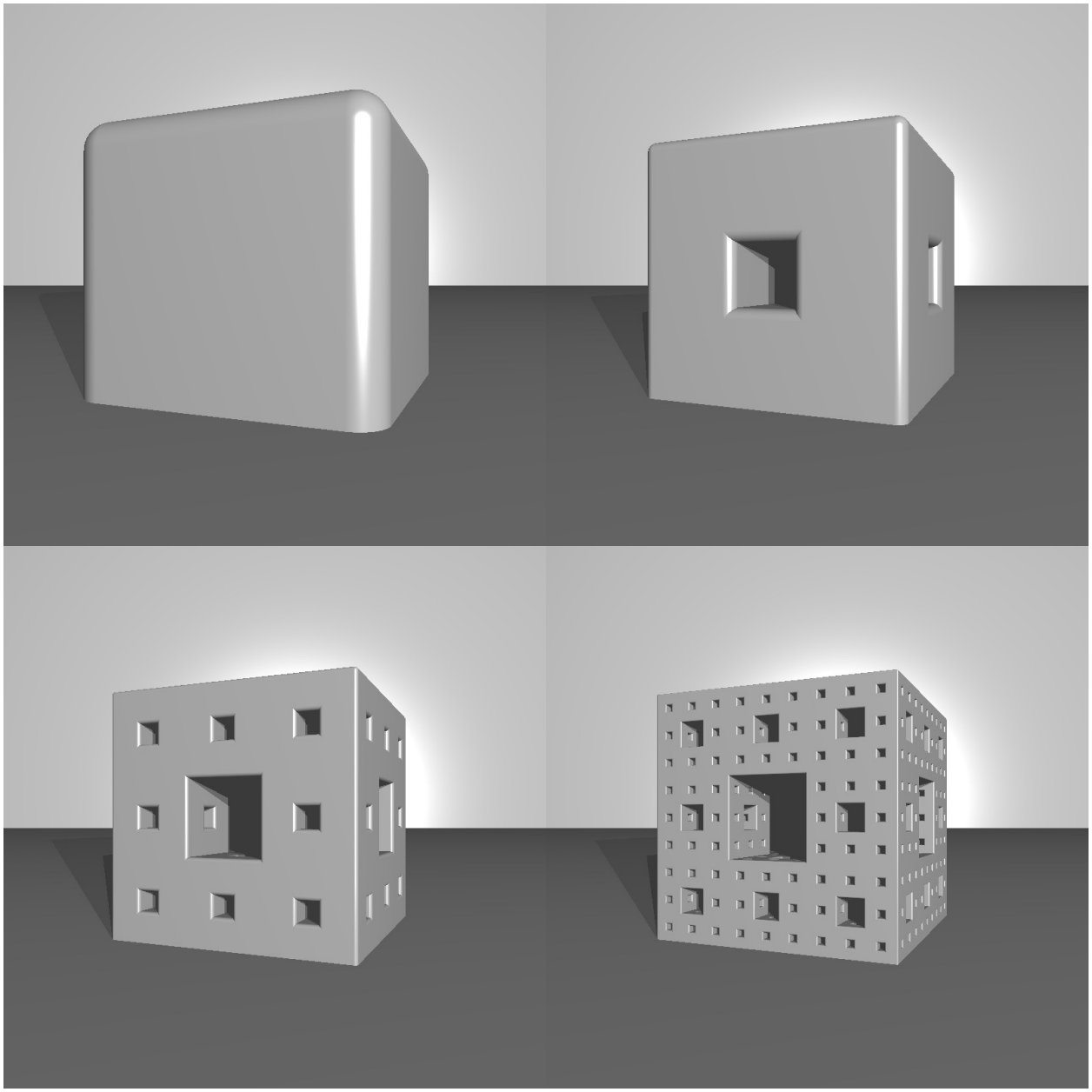


Figure 15: A Menger Cube after 0, 1, 2, and 3 iterations. A roundbox was chosen for the underlying distance field for aesthetic reasons

5.1 Introduction

The Menger Cube is a fractal that can be constructed by iteratively applying transformation operations to the previously mentioned Euclidean distance fields. Without any iteration the Menger Cube's distance function returns the same value as its underlying primitive distance field. Typically a box is used for as the underlying primitive distance field but in **Figure 15** a roundbox was used for aesthetic reasons.

5.2 Iterative Definition
The iteration loop of a Menger Cube can be defined as:

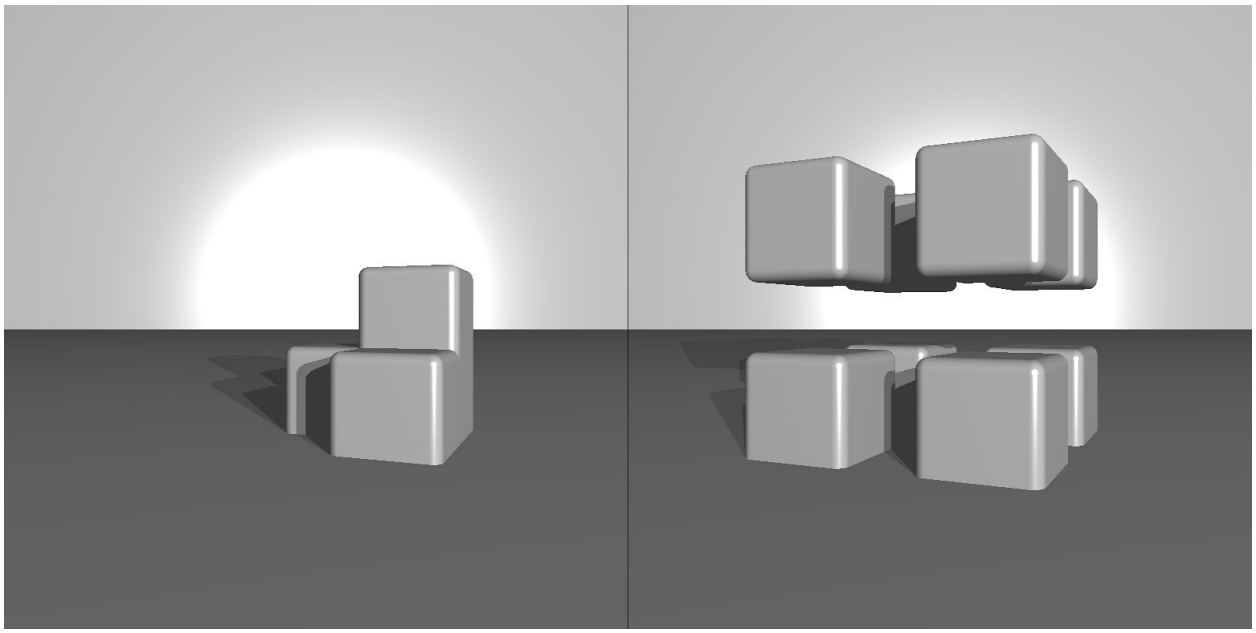
```
// Conditional mirroring
position = abs(position);

// Conditional swapping
position.xy = position.x < position.y ? position.yx : position.xy;
position.yz = position.y < position.z ? position.zy : position.yz;
position.xy = position.x < position.y ? position.yx : position.xy;

// Scale and translate
position = position*3.0-2.0;

// Conditional translation
position.z = position.z < -1.0 ? position.z+2.0 : position.z;
```

Often it is simpler to understand the distance function operations visually and so the result of a single iteration of the above code with the exception of *position = abs(position);* is shown in **Figure 16** (positive x is right, positive y is down, and positive z is forward). The section of Menger Cube in **Figure 16** is in the positive quadrant but the abs function will mirror marching points in negative quadrants so that they still approach this section of Menger Cube. As a result the abs function behaves like a conditional mirroring of a distance field.



Figures 16 and 17: On the left is the result of a single iteration of the Menger Cube without *position = abs(position);*. On the right is the the result of a single iteration just before *position.z = position.z < -1.0 ? position.z+2.0 : position.z;*

Figure 17 shows the result of all operations except *position.z = position.z < -1.0 ? position.z+2.0 : position.z;* which is normally a conditional translation along the z-axis. However, the previous mirroring from the abs function and conditional swapping cause duplicates of the small roundboxes to instead translate towards one another. As a result the gaps between the small roundboxes are filled with these translated copies forming a complete single iteration of the Menger Cube.

6 Mandelbox

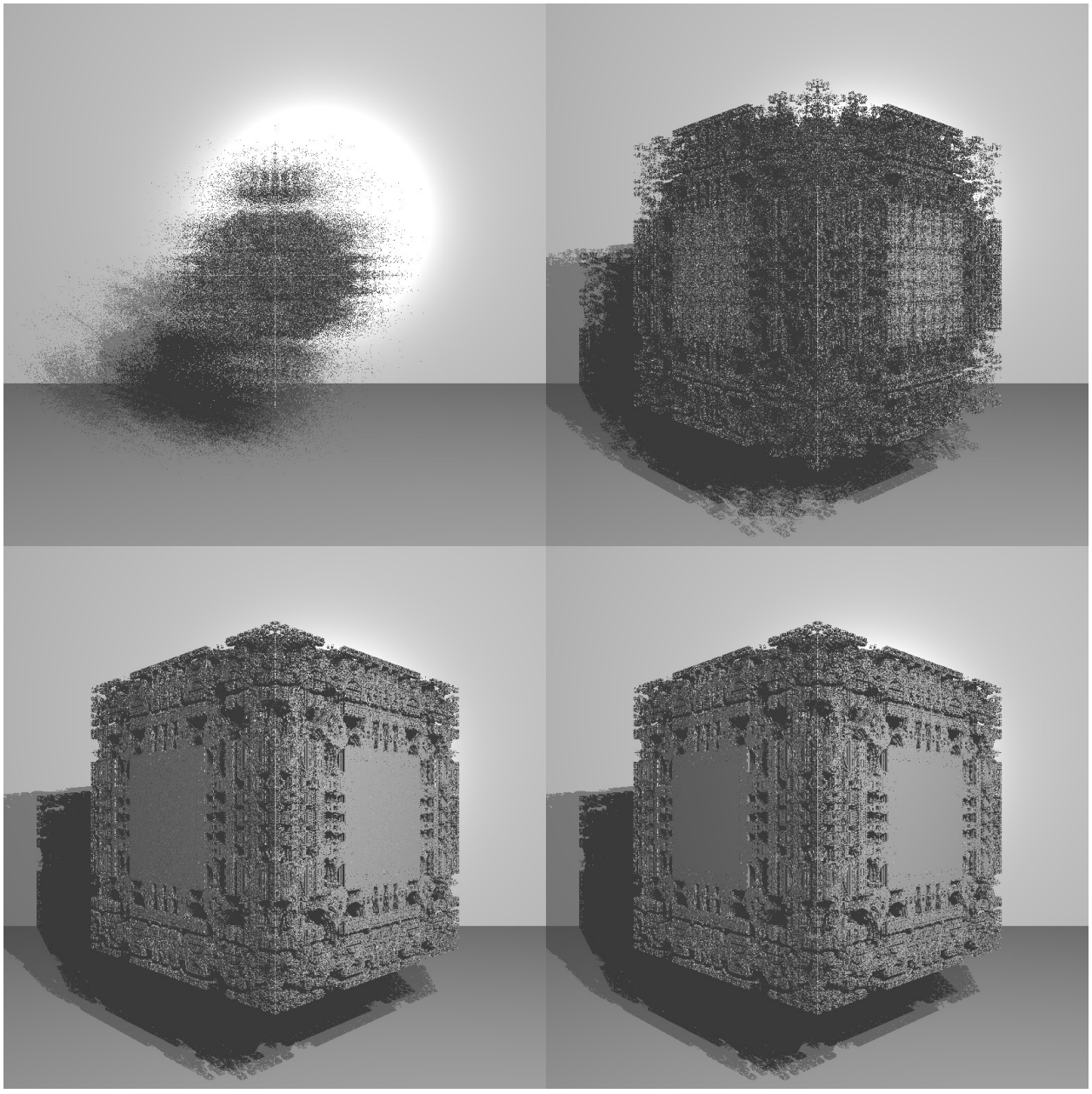


Figure 18: A Mandelbox with commonly used parameters after 8, 12, 16, and 20 iterations

6.1 Introduction

The Mandelbox is a generalization of the Mandelbrot Set that can exist in any number of dimensions although three dimensions will be discussed here (the Mandelbrot Set exists in two dimensions). The Mandelbrot Set uses the equation:

$$z = z^2 + c$$

The Mandelbox uses the equation (where *z* corresponds to *position* in the previous chapters):

$$z = SCALE * sphereFold(boxFold(z)) + c$$

In the above equation *SCALE* is a user specified parameter but a common value is 2.0. Before any iteration occurs *c* is assigned as a three component vector with the same values as *z* and serves the purpose of being a constant offset (as in the Mandelbrot Set equation). Although not shown in the above equation, the boxFold and sphereFold functions also involve user specified parameters and common values are used in **Figure 18**. Additionally calculating the distance to the Mandelbox surface involves maintaining a running derivative value, *d*, initialized as 1.0 which will be later discussed.

6.2 boxFold()

For each dimension (x, y, z) the boxFold function mirrors a point about a positive and negative user specified value, *BOX_FOLD*, which is commonly 1.0. Below is the corresponding code for *z.x*, the code is identical for *z.y* and *z.z* and so is not shown:

```
if (z.x > BOX_FOLD) z.x = 2.0 * BOX_FOLD - z.x;
else if (z.x < -BOX_FOLD) z.x = -2.0 * BOX_FOLD - z.x;
```

Mirroring a three component vector point about positive and negative *BOX_FOLD* can be done in a single GLSL statement (for each dimension):

```
z = clamp(z, -BOX_FOLD, BOX_FOLD) * 2.0 - z;
```

6.3 sphereFold()

This function serves two purposes: linearly scaling points too close to the origin away from the origin and applying a sphere inversion. Points at a distance less than *MIN_RADIUS* from the origin are linearly scaled. Points that are not affected by the initial condition but are at a distance less than *FIXED_RADIUS* from the origin have a sphere inversion applied. Common values for *MIN_RADIUS* and *FIXED_RADIUS* are 0.5 and 1.0 respectively. Additionally the running derivative value, *d*, has the same conditional operation performed as *z*:

```
float zDot = dot(z, z);
if (zDot < MIN_RADIUS) {
    z *= FIXED_RADIUS / MIN_RADIUS;
    d *= FIXED_RADIUS / MIN_RADIUS;
}
else if (zDot < FIXED_RADIUS) {
    z *= FIXED_RADIUS / zDot;
    d *= FIXED_RADIUS / zDot;
}
```

6.4 Scale and Translate

Following the boxFold and sphereFold functions both *z* and *d* are scaled and translated as shown below:

```
z = SCALE * z + c;
d = abs(SCALE) * d + 1.0;
```

As previously mentioned, *c* is a three component vector with the initial values of *z* before any iteration occurs. Similarly *d*'s offset is its initial value, 1.0. Because a value of 2.0 was used for *SCALE* in **Figure 18** the abs function above is unnecessary but it is possible that a negative user value is used instead.

6.5 Distance Function

The above processes described in sections **6.2**, **6.3**, and **6.4** are iterated some number of times (as shown in **Figure 18**, sixteen iterations produces a sufficiently dense distance field). Following these iterations which modify *z* and *d*, the below value is returned:

```
return length(z) / abs(d);
```

Admittedly why dividing the length of the iteratively transformed vector, *z*, by the running derivative, *d*, produces the implicit surface of the Mandelbox is beyond my understanding. However, the resulting distance function's implicit surface can be approached by a marching point with the previously mentioned Sphere Tracing models.

6.6 Computational Efficiency

The computations involved with the Mandelbox distance field are significantly more expensive than the Euclidean distance fields and Menger Cube. Although the Menger Cube is also iterative it produces a coherent implicit surface at low iterations while the Mandelbox does not. The bottleneck in computation of the Mandelbox distance field is its iteration loop (sections **6.2**, **6.3**, and **6.4**). Reducing this bottleneck can be done by using a four component vector as *z* where *z.w* is the running derivative, *d*. The complete GLSL code for this style of computation is provided below:

```
float mandelbox(vec3 position) {
    // The running derivative is z.w
    vec4 z = vec4(position, 1.0);
    vec4 c = z;

    for (int i = 0; i < 16; i++) {
        // Boxfold
        z.xyz = clamp(z.xyz, -BOX_FOLD, BOX_FOLD) * 2.0 - z.xyz;

        // Spherefold
        float zDot = dot(z.xyz, z.xyz);
        if (zDot < MIN_RADIUS) z *= LINEAR_SCALE;
        else if (zDot < FIXED_RADIUS) z *= FIXED_RADIUS / zDot;

        z = SCALE * z + c;
    }

    return length(z.xyz) / abs(z.w);
}
```

In the above code *LINEAR_SCALE* is defined as *FIXED_RADIUS / MIN_RADIUS*. Code similar to the above was used to make a real time high-resolution render of the Mandelbox in a 2k x 2k pixel window using ambient occlusion approximation as shown in **Figure 19**. The real time renderer made use of six GPUs and display synchronization was handled by SAGE2. Ambient occlusion shading was chosen as each pixel only then requires a single march loop while Blinn-Phong with shadows requires at least eight (one march to reach the implicit surface, six to create a surface normal approximation, and one per light in the scene).

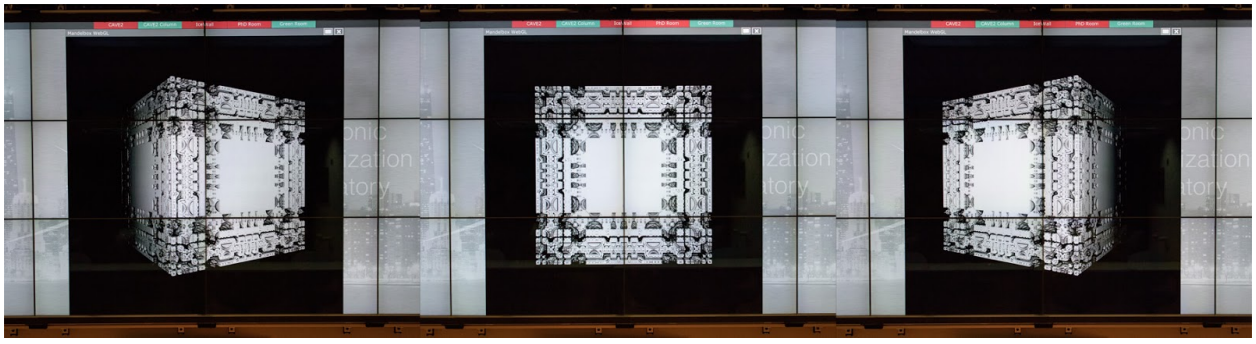


Figure 19: Three photos as a time-lapse of the 2k x 2k pixel resolution Mandelbox running in real time under ambient occlusion approximation shading. Photos provided by Lance Long

7 Conclusion

The Sphere Tracing models presented are applicable to arbitrary scenes composed of distance fields. Complex scenes can be created with simple Euclidean distance fields transformed through the operations presented in section **3.3**. Several of the distance fields presented are impractical if not impossible to represent with traditional rendering techniques. Additionally scenes involving surfaces of infinite complexity at their iterative limit can be visualized in real time using the presented methods. Due to distance fields being defined by equations rather than geometry the memory requirements of even highly complex scenes is trivial.

The presented application makes use of available GPUs of a device to display distance field scenes in real time. However the time required to render a scene is dependant on several aspects. To illustrate the significance of each aspect a scene was composed of two planes and a Menger Cube that rotates and changes size in a repeating pattern during runtime (the Menger Cube's distance function is set to perform three iterations, refer to **Figure 15**). The Blinn-Phong shading model was used with a single light for shadows. All testing was performed on my laptop which has an "AMD Radeon HD 6490M 256 MB" graphics card and a resolution of 800 x 600 pixels was chosen. Under these conditions the scene - which contains a fairly complex distance field - averaged a frame every 129.3 milliseconds (this will be abbreviated to mspf, milliseconds per frame).

Display Resolution: Decreasing the display resolution by a factor of 2.0 (400 x 300 pixels) resulted in an average of 37.8 mspf (the resolution was decreased rather than increased due to my laptop's resolution only being 1440 x 900 pixels). The speedup should theoretically have a quadratic relationship with the resize factor.

Shading Model: Using ambient occlusion approximation rather than Blinn-Phong with a single light source shadow resulted in an average of 55.6 mspf. This was initially surprising considering that Blinn-Phong with one shadow involves eight marches. However the six marches to find the surface normal have initial marching points nearby a distance field and so involve few iterations. As a result only two of the eight marches are relatively expensive and this is reflected in Blinn-Phong taking slightly more than twice as long as ambient occlusion approximation.

ε: Increasing ε by a magnitude (from 0.0001 to 0.001) resulted in an average of 104.7 mspf. Increasing ε shortens the number of iterations for all eight marches.

Near and Far Clipping Spheres: Using knowledge about the scene to place the near and far clipping spheres as closely as possible to the scene without affecting the visual result gave an average of 126.1 mspf. However, this scene is a poor example of this optimization because it contains background planes. Removing the planes from the scene resulted in an average of

27.8 mspf and then again setting the near and far clipping planes based on knowledge of the new scene resulted in an average of 21.6 mspf.

It may initially seem surprising that the background planes in the scene are responsible for the majority of the computation time despite a fairly complex surface, the Menger Cube, being within the scene. The reason for this is due to ij camera direction vectors being relatively nearby the ground plane as they travel towards the back plane. None of the marching iterations advance a large distance along the corresponding ij camera direction vector because the distance to the ground plane is always small. In addition each iteration of the marches then involves calculating the Menger Cube as it is a part of the world distance function even though it may not be nearby.

There are other scene dependant aspects that can significantly affect computational time such as the quantity of distance fields, the complexity of distance fields, and the relative positions of the implicit surfaces relative to the camera. Under the testing settings previously mentioned the most complex distance field discussed, the Mandelbox, runs at an average of 818.6 fpms using Blinn-Phong shading with a single shadow (its distance function was set to perform sixteen iterations, refer to **Figure 18**). This average fpms can be reduced to 759.9 by manipulating the near and far clipping spheres based on knowledge of the scene. Additionally increasing ε based on knowing the target resolution (without affecting the visual result) reduced the average fpms to 400.1. When instead using ambient occlusion approximation the average fpms becomes 181.2.

On a high-resolution display wall with six high performance GPUs a real time Mandelbox could be displayed using ambient occlusion approximation in real time at a resolution of roughly 2k x 2k pixels as shown in **Figure 19**. Real time display was not maintainable for a Mandelbox using Blinn-Phong shading with a single shadow at 2k x 2k resolution, the result in **Figure 2** is a single pass render.

References

Christensen, M. 2011, *Distance Estimated 3D Fractals (II): Lighting and Coloring*
http://blog.hvidtfeldts.net/index.php/2011/08/distance-estimated-3d-fractals-ii-lighting-and-coloring/

Christensen, M. 2011, *Distance Estimated 3D Fractals (VI): The Mandelbox*
http://blog.hvidtfeldts.net/index.php/2011/11/distance-estimated-3d-fractals-vi-the-mandelbox/

Lowe, T. 2011, *What is a Mandelbox*
https://sites.google.com/site/mandelbox/what-is-a-mandelbox

McGuire, M., 2014, *Numerical Methods for Ray Tracing Implicitly Defined Surfaces*
http://graphics.williams.edu/courses/cs371/f14/reading/implicit.pdf

Quilez, I. 2008, *modeling with distance functions*
http://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm

Quilez, I. 2010, *free penumbra shadows for raymarching distance fields*
http://www.iquilezles.org/www/articles/rmshadows/rmshadows.htm