CSCI4110U - Final - Rendering Signed Distance Fields With Ray Marching

2024-12-10

Jaime Barillas - 100505421

The Program

The program renders a scene built entirely with Signed Distance Field representations of primitive shapes and acompanying operators. A ray marcher is used to render spheres, boxes, cones, and other shapes into the form of the pokemon Magnemite floating in an outdoor field. The scene is lightly animated with Magnemite periodically approaching the camera and a gentle breeze bending the grass.

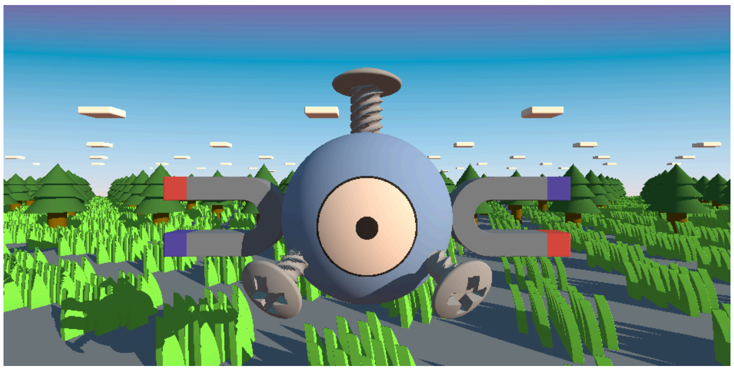
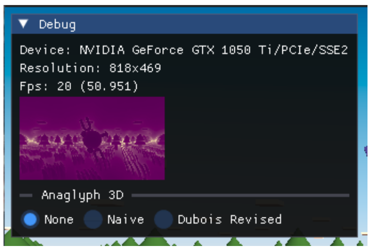


Figure 1: Pokemon #81: Magnemite, up close to the camera.

The program also provides a GUI (enabled with the `spacebar` key) which displays a texture showing how costly a particular pixel is to draw and radio buttons to turn on/off the 3D mode of the scene.



```
Debug
Device: NVIDIA GeForce GTX 1050 Ti/PCIe/SSE2
Resolution: 818x469
Fps: 20 (50.951)
```

Figure 2: Pressing the `spacebar` key brings up this menu.

OpenGL Setup

OpenGL is setup to render the scene to a Framebuffer with an additional colour attachment texture which is used to generate an “iterations” texture which represents how many times a ray had to march forward, per pixel, until it hit an object. Both textures are regenerated when the window size changes.

```
    // Safe since 0's and non-existant textures are silently ignored.
    glDeleteTextures(1, &image_texture);
    glDeleteTextures(1, &iterations_texture);
    glBindFramebuffer(GL_FRAMEBUFFER, fbo);

    glGenTextures(1, &image_texture);
    glBindTexture(GL_TEXTURE_2D, image_texture);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, resolution.x, resolution.y, 0, GL_RGBA,
GL_UNSIGNED_BYTE, NULL);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glBindTexture(GL_TEXTURE_2D, 0);

    glGenTextures(1, &iterations_texture);
    glBindTexture(GL_TEXTURE_2D, iterations_texture);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, resolution.x, resolution.y, 0, GL_RGBA,
GL_UNSIGNED_BYTE, NULL);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glBindTexture(GL_TEXTURE_2D, 0);

    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, image_texture, 0);
    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT1, GL_TEXTURE_2D, iterations_texture,
0);
```

Listing 1: Two colour attachments generated for the Framebuffer. One to render the scene, the other to render an "iterations" texture.

At the start of the draw function, the Framebuffer is bound and the shader which contains the scene code and the ray marcher is selected. The scene shader consists of four files, three of which are important: `shaders/sdf.glsl` contains the various SDFs and useful operators, `shaders/ray_marcher.glsl` contains the ray marching code and writes pixel values to the main texture and the iterations texture, and `shaders/scene.glsl` which combines the various SDF primitives and operators into a scene.

```
shader_manager.compileAndWatch({
  .name = "scene",
  .shaders = {
    Shader{.path = "shaders/vert.glsl",        .type = GL_VERTEX_SHADER},
    Shader{.path = "shaders/sdf.glsl",         .type = GL_FRAGMENT_SHADER},
    Shader{.path = "shaders/ray_marcher.glsl", .type = GL_FRAGMENT_SHADER},
    Shader{.path = "shaders/scene.glsl",       .type = GL_FRAGMENT_SHADER}
  }
});
shader_manager.compileAndWatch({
  .name = "screen",
  .shaders = {
    Shader{.path = "shaders/screen-vert.glsl", .type = GL_VERTEX_SHADER},
    Shader{.path = "shaders/screen-frag.glsl", .type = GL_FRAGMENT_SHADER},
  }
});
```

Listing 2: The `scene` and `screen` shaders. When one of their corresponding files is modified, the shader will be automatically reloaded.

Next, the various uniforms used by the scene shader are set, after which, the scene itself is drawn to the Framebuffer. Lastly, the Framebuffer is drawn to the screen by binding Framebuffer 0, activating the `screen` shader, and issuing draw calls for both the screen and the GUI.

In order to draw the scene defined within the fragment shader, a screen sized quad is generated and uploaded to the GPU.

Ray Marching The Scene

Rendering the various SDFs is done through ray marching where a ray per pixel is iteratively traversed until it hits an object. The combination of SDFs for the scene, evaluated at the current point along the ray, give a safe upper bound on how far the ray can advance before intersecting an object. One nice benefit of SDFs is that combining simple primitive shapes into more complex ones can be done fairly simply with a small collection of operators. Additionally, morphing the shapes of objects a similarly easy task.

The iterative traversal of the scene by rays has the opportunity to be costly. This happens whenever a ray passes nearby an object without intersecting it. In this case, the distance returned by an SDF will be small causing the ray to advance in small steps, which in turn cause additional evaluations of the SDF until either the ray finally passes the object a maximum number of evaluations is reached.

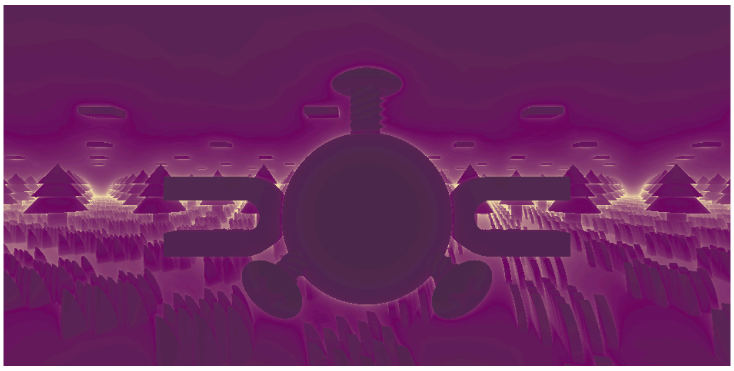


Figure 3: An image showcasing how many steps the rays marched for each pixel. More steps means more SDF evaluations. Purple areas mean the ray either hit an object or the far-plane in few steps. Bright yellow means many steps were required for that pixel.

This report will cover only the more interesting shapes within the scene, specifically, how they are defined, combined, morphed, and coloured. All the SDF definitions for the various shapes and operators were taken from Inigo Quilez's articles on 2D[1] and 3D[2] distance functions.

Ray Marching, Normals, and Lighting

The ray marching, normal calculation, and lighting code are available in the `castRay`, `sceneNormal`, and `sceneLighting` functions in the `shaders/ray_marcher.glsl` file. Of particular interest may be the normal calculation which uses the so-called "Tetrahedron Technique" as defined in the "normals for an SDF" article by Inigo Quilez[3]. This is numerical method by which the normal is calculated from four additional evaluations of the scene.

The values used in the scene lighting are best explained by the video they are based on "LIVE Coding "Happy Jumping""[4]. In short, the values and calculations used are intended to make the behaviour of the lighting intuitive. This video was used to quickly get a working ray marcher and a scene with a single sphere before working on the Magnemite model. The entire scene is made from diffuse materials, no specular reflections is exists.

Shadows are implemented as a single extra ray march from the point of intersection in the direction of the sun light (which is a directional light). Only hard shadows are implemented but it is possible to implement soft shadows with little extra computational cost[5].

Trees

The trees are made from two SDFs: The line segment and the cone. The tree trunk, made from the line segment is made thick by offsetting the distance calculated by the SDF by some amount. This introduces the first operator used in the scene: The `rounding` operator.

```
float sdfVerticalCapsule(in vec3 point, in float height, in float offset) {
    // height = height from bottom.
    point.y -= clamp(point.y, 0.0, height);
    return length(point) - offset;
}
```

Listing 3: Notice how `offset` is subtracted from the calculated distance. This produces a larger, rounder shape encompassing the original.

The trees are repeated infinitely in the x and z directions using the `repeat2D` operator (covered in the section on grass). Evaluation of the tree SDF is guarded by a check against the z coordinate of the current ray so that they are only drawn at a distance in the background.

```
if (point.z < -2) {
    vec3 tree_point = point;
    tree_point.y -= -0.8;
    tree_point *= 0.5;
    tree_point.xz = sdfOpRepeat2D(tree_point.xz, vec2(0.8));
    vec2 tree = drawTree(tree_point);
    tree.x /= 0.5;
    if (tree.x < res.x) res = tree;
}
```

Listing 4:

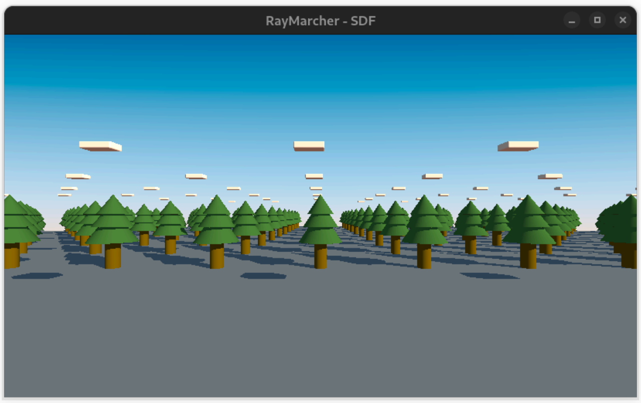


Figure 4:

The code which comines the two shapes into a tree is available in the `shaders/scene.glsl` file at line ~70.

Grass Patches

The grass patches are made from a collection of vesica piscis extruded to 3D. The vesica SDF is defined as a 2D SDF (line ~98 of `shaders/sdf.glsl`) and converted to a 3D shape using the following `extrusion` operator:

```
float sdfOpExtrude(in vec3 point, in float d, in float amount) {
    vec2 w = vec2(d, abs(point.z) - amount);
    return min(max(w.x,w.y),0.0) + length(max(w,0.0));
}
```

Listing 5: See Inigo Quilez's "distance functions" [2].

The vesica shape is repeated in the x and z directions using a `repeat2DClamped` operator. This operator allows limiting the number of repititions which is used to mimic the "square" patches of grass found in the older Pokemon games. These patches are then repeated infinitely in the x and z directions with the simpler `repeat2D` operator.

```
vec2 sdfOpRepeat2D(in vec2 point, in vec2 scale) {
    return point - (scale * round(point / scale));
}
```

Listing 6: The simpler `sdfOpRepeat2D`.

```
vec2 sdfOpRepeat2DClamped(in vec2 point, in vec2 scale, in vec2 limit) {
    return point - (scale * clamp(round(point / scale), -limit, limit));
}
```

Listing 7: `sdfOpRepeat2DClamped` uses `clamp` to limit the number of repititions.

Finally, the grass is bent periodically with the code bellow. `bend_factor` is squared to avoid negative values of sine, which would bend the grass in the oposite direction of the fictional wind within the scene. `fy` dictates how much of the bend applies to the vesica at each y level up to the tip. A cubic curve is employed to ensure the vesica bends instead of skews. This bending unfortunately distorts the distance field and causes visual glitches in grass as can be seen on patches on the left-hand-side of the image below.

```
float bend_factor = sin(itime/2);
float fy = fract(point.y);
grass_point.x -= -fy*fy*fy * (bend_factor*bend_factor);
float grass = drawGrass(grass_point);
```

Listing 8: `grass` holds the value of the grass SDF.

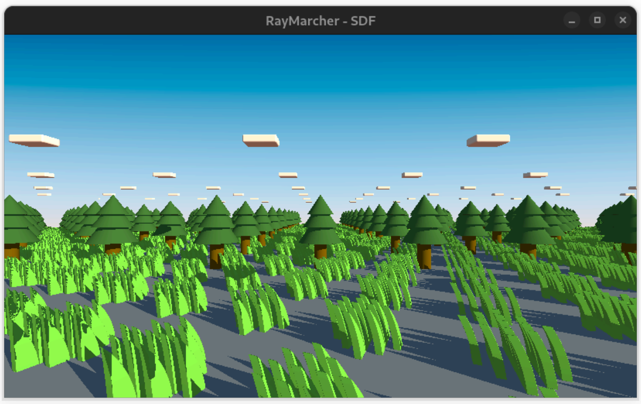


Figure 5: The grass bends in response to (non-existant) wind. Visual glitches can be seen in the grass as it bends.

Magnemite

The most interesting parts of the Magnemite model are the horseshoe magnet arms and the various screws attached to its body.

Horseshoe Magnets

The SDF for the horseshoe magnets use the same trick as the grass to create a 3D SDF from the 2D definition (See line ~56 in `shaders/sdf.glsl`). Symmetry in the shape of Magnemite is used to draw both the left and right horseshoe magnet with a single SDF evaluation. The symmetry is implemented by taking the absolute value of the x coordinate of the current point along the ray.

Screws

The screws attached to Magnemite's body are more complex. The screw body is created from a box that has been twisted about the y axis. Much like when bending the grass, the `twistY` operator distorts the distance field and visual glitches can be seen in the shadows generated on the screws. The screw heads are made from the SDF of a cut sphere and a pair of boxes arranged in a "+" pattern. Where all the previously mentioned compound shapes use a `min`/union operator to comine the shapes, the screw head uses a `max`/subtraction operation to carve out the "+"-shaped hole.

```
vec3 sdfOpTwistY(in vec3 point, in float amount) {
    float c  = cos(amount * point.y);
    float s  = sin(amount * point.y);
    mat3 rot = mat3(
         c, 0, s,
         0, 1,  0,
        -s, 0,  c);
    //vec3 new_point = vec3(rot * point.xz, point.y);
    return rot * point;
}
```

Listing 9: Twisting about the y axis.

Evaluating the shapes of the screws happens to be the most costly of part of the scene. One way to mitigate some of the cost is to avoid computing this part of the SDF for every pixel or step along the ray. The spherical body of the Magnemite (plus an offset to cover the rest of the model) can be used as a bounding volume to implement this optimization. This can prove a very effective technique: Running the program on an NVidia GeForce GTX 1050 TI Mobile GPU at a resolution of 1512x985 resulted in a rendering performance of 38 fps before the bounding volume optimization and 60fps after (capped by VSync). It should be noted that the SDF for the screws is not written in the most optimized manner to begin with, so perhaps the bounding volume optimization was not needed. Regardless, these results show how effective the technique can be.

```
float body_radius = 0.15;
float body = sdfSphere(magnemite_point, body_radius);
res = vec2(body, 1.0);

// Use body as bounding volume
if (body - 1 < 0) {
  // Rest of Magnemite SDF...
}
```

Listing 10: Use of the Magnemite body as a bounding volume for the rest of the model. Line 145 in `shaders/scene.glsl`

Colouring

Each "top-level" SDF (essentially, any shape that requires its own colour) returns both the distance to its surface and a material ID use to select the colour of the object. The `sceneColor` function in `shaders/scene.glsl` branches on this ID to determine the colour of the object. The most interesting of these is Magnemite's body due to how the eye colour is chosen. The point at which the ray intersects the body is transformed by a matrix which contains the various translations and rotations applied to Magnemite as part of its animation. The resulting vector is then normalized and its dot product against the positive z axis is taken to get the cosine of the angle between the two. This value is then used to section of parts of the body as the eye.

```
float body_radius = 0.15;
float body = sdfSphere(magnemite_point, body_radius);
res = vec2(body, 1.0);
```

Listing 11: Magnemite's body sets the return value of `scene` function to a vec2 containing the distance to the body and the body's material ID.

```
vec4 transformed_point = vec4(point, 1.0) * magnemite_tx;
vec3 n = normalize(transformed_point.xyz);

float d = dot(n, vec3(0, 0, 1.0));
if (d > 0.995) {
    material = vec3(0.005); // Black pupil
} else if (d > 0.9) {
    material = vec3(0.2); // White sclera
} else if (d > 0.89) {
    material = vec3(0.005); // Black outline
}
```

Listing 12: Part of the `sceneColor` function that colours the eye. `material` is a local variable returned by the function (the colour). `id` and `point` are parameters, `magnemite_tx` is the transformation matrix (a global) for Magnemite.

3D

Pressing the `spacebar` key displays the menu from which the anaglyph 3D mode can be turned on. There are two 3D options: Naive and Dubois. Naive mode simply filters out the red/green-blue portions of the colour for the part of the image meant for the left/right eye. The Dubois 3D mode uses Dubois' least squares method, specifically, the matrix specificed in Sanders' and McAllister's "Producing Anaglyphs from Synthetic Images"[6].

Between the two modes, the naive method provides better colour accuracy but produces an image that is somewhat uncomfortable to look at. In particular, reds are hard to look at since they appear white to the left eye, but appear red-ish to the right eye. The Dubois method produces a more comfortable image, but loses significant colour accuracy (reds are pretty much gone from the image.) The code for the 3D filtering is available in the `main` function of the `shaders/ray_marcher.glsl` file.

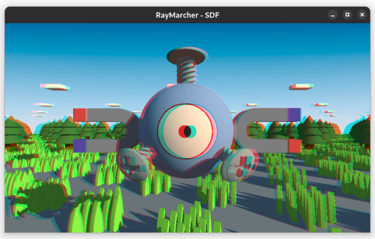


Figure 6: The naive (Photoshop) method.

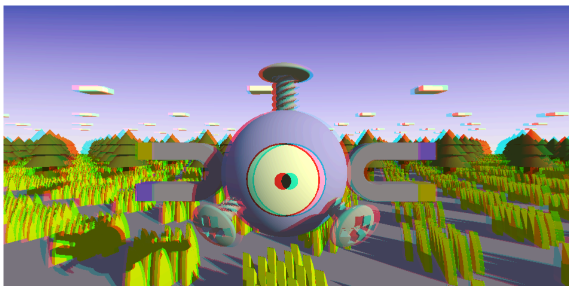


Figure 7: The Dubois method.

Bibliography

[1] Inigo Quilez, "2D distance functions." [Online]. Available: https://iquilezles.org/articles/distfunctions2d/

[2] Inigo Quilez, "distance functions." [Online]. Available: https://iquilezles.org/articles/distfunctions/

[3] Inigo Quilez, “normals for an SDF.” [Online]. Available: https://iquilezles.org/articles/normalsSDF/

[4] Inigo Quilez, *LIVE Coding "Happy Jumping"*, (Aug. 13, 2019). [OnlineVideo]. Available: https://www.youtube.com/live/Cfe5UQ-1L9Q

[5] Inigo Quilez, “soft shadows in raymarched SDFs.” [Online]. Available: https://iquilezles.org/articles/rmshadows/

[6] W. Sanders and D. F. McAllister, “Producing Anaglyphs from Synthetic Images.” [Online]. Available: https://ocul-it.primo.exlibrisgroup.com/permalink/01OCUL_IT/7cp26g/cdi_spie_primary_PSISDG00500600001000348000001