Menger Marcher by Cristobal Sciutto

Bare-bones Ray Marching of Shaded Fractal Signed Distance Functions

Code: *github.com/tobyshooters/mengermarcher*

Project Description

In this project, I implemented the ray-marching rendering method from scratch and create interesting patterns with signed distance functions. The goal was to explore implicit geometry representations, and the facility of describing recursive structures with them. Another central aim was to understand more fundamentally the systems side of these implementations. In this vein, I implemented all classes used (e.g. Vec3, Mat3, animation), and avoided using abstractions such as OpenGL APIs.

Developments

Below I present a walkthrough of the majority of features implemented. The full codebase is available at *github.com/tobyshooters/mengermarcher* for further inspection. Beyond the features elucidated below, the code also:

- Provides Supersampling capabilities
- Generates PPM images from an array of pixel values
- Implements Vec3 and Mat3
- Implements Semaphore and Threadpool (based on CS110) for concurrency of rendering

Ray-marching Algorithm

To generate an image using ray-marching, one projects a ray from the camera, through an image-plane, onto a scene. Each ray corresponds to a sample of the scene, and the rays are distributed uniformly across the pixels. The key to ray-marching is the usage of implicit definitions of surfaces, i.e. surfaces defined as level-sets of vector-valued functions.

Namely, a signed distance function consists of a function that evaluates to zero outside of the surface, postive outside of the surface, and negative inside of the surface. For example, a sphere centered at `c` and with radius `r` can be described by the function below. Given a point `p`, it returns the distance between `p` and the surface.

```
def SDF_sphere(Vec3 p, Vec3 c, double r) {
    return (p - c).norm() - r;
}
```

Thus, ray-marching consists of evaluating the distance from origin of the ray, and then "marching" across the ray that distance. This is best visualized in Figure 1.

Note that while we march along the direction of ray, the closest point to the surface might not be in that direction. Regardless, we are guaranteed to not pass the surface since we walk at most the minimum distance.

This algorithm is implemented below, for a generic signed distance function SDF:

```
double march_ray(const Vec3& origin, const Vec3& direction, double (*SDF)(const Vec3&)) {
  double t = 0.001;
  for (int i = 0; i < MARCH_ITERATIONS; i++) {
    double d = SDF(origin + t * direction);
```

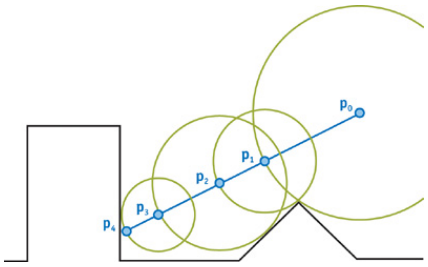

Figure 1: ray-marching illustration

```
    if (d < 0.0001) return t;
    t += d;
  }
  return 0;
}
```

Gven the `t` value returned from marching, one can deduce the location of collision with the surface by evaluating `origin + t * direction`.

Signed Distance Function Techniques

Beyond the simple sphere described above, the Box and Plane primitives were implemented. The box is defined a Vec3 from the origin to a corner of the box. In the case of the plane, it is defined by a center point, and a normal to the point.

```
inline double SDF_box(const Vec3& p, const Vec3& s) {
  return vmax(abs(p) - s);
}

inline double SDF_plane(const Vec3& p, const Vec3& c, const Vec3& n) {
  return dot(p - c, n);
}
```

An interesting technique is the usage of trigonometric function to parametrically vary the radius of a sphere, allowing for a spiked surface. This technique is know as displacement is exemplified by:

```
double SDF_hedgehog(const Vec3& p, const double sphere_radius, const double noise_amplitude) {
  // Generates spikes using interweaving sine functions
  Vec3 s = Vec3(p).normalize(sphere_radius);
  double delta = sin(16 * s.x) * sin(16 * s.y) * sin(16 * s.z);
  return p.norm() - (sphere_radius + delta * noise_amplitude);
}
```

Another technique is the repetiton of objects by using the modulus operator. Intuitively, what happens when we compute the mod of a position in 3-space by m is that range is collapsed to `(-m/2, m/2)` and repeated. This is exemplified below:

```
double SDF_sphere_repeated(const Vec3& p, const double sphere_radius, const double spread) {
  Vec3 repeated = mod(p, spread) - Vec3(spread / 2);
  return SDF_sphere(repeated, sphere_radius);
}
```

Finally, we can combine different signed distance using boolean algebra with the functions below:

```
double union      (double d_a, double d_b) { return min(d_a,      d_b); }
double intersect  (double d_a, double d_b) { return max(d_a,      d_b); }
double difference (double d_a, double d_b) { return max(d_a, -1 * d_b); }
```

Using all of these techniques together, we can define the Menger sponge. This is done by recursively subtracting off crosses from a cube.

```
double SDF_cross(const Vec3& p) {
  double inf = 1000000.0;
  double box1 = SDF_box(p, Vec3(inf, 1.0, 1.0));
  double box2 = SDF_box(p, Vec3(1.0, inf, 1.0));
  double box3 = SDF_box(p, Vec3(1.0, 1.0, inf));
  return SDF_union(box1, SDF_union(box2, box3));
}

double SDF_menger(const Vec3& p, int iterations) {
  double d = SDF_box(p, Vec3(1.0));
  double s = 1.0;

  for (int i = 0; i < iterations; i++) {
    Vec3 a = mod(p * s, 2.0) - 1.0;
    Vec3 r = Vec3(1.0) - 3.0 * abs(a);
    s *= 3.0;
    double c = SDF_cross(r) / s;
    d = max(d, c);
  }
  return d;
}
```

Normals, Lighting, and Multiple Light Sources

The signed distance function allows for easy computation of normal vectors. Given a position on the surface of a SDF, one can approximate the normal to the SDF by calculating the gradient. This is because the direction increase in SDF value will be normal to the surface. The code below numerically approximates the gradient by evaluating the SDF at small steps in each direction.

```
Vec3 SDF_normal(const Vec3& pos, double (*SDF)(const Vec3&)) {
  const double eps = 0.001;
  double d = SDF(pos);
  double normal_x = SDF(pos + Vec3(eps, 0, 0)) - d;
  double normal_y = SDF(pos + Vec3(0, eps, 0)) - d;
  double normal_z = SDF(pos + Vec3(0, 0, eps)) - d;
  return Vec3(normal_x, normal_y, normal_z).normalize();
}
```

Given a normal to the surface, we can calculate lighting crudely by comparing the normal of the surface to the direction from the that point to the light source.

```
double calculate_intensity(Vec3& light_pos, Vec3& collision_pos, double (*SDF)(const Vec3&)) {
  Vec3 light_dir = (light_pos - collision_pos).normalize();
  return max(0.4, dot(light_dir, SDF_normal(collision_pos, SDF)));
}
```

More interestingly, we can use the normal, along with the camera and lighting position to calculate the relevant vectors for the Phong reflection model.

```
Vec3 phong_reflection(Vec3& diffuse_color,
                     double attenuation,
                     Vec3& light_pos,
                     Vec3& collision_pos,
                     Vec3& camera_pos,
                     double (*SDF)(const Vec3&)) {

  Vec3 specular_color = Vec3(1.0, 1.0, 1.0) * attenuation;
  double specular_exponent = 50;

  Vec3 L = (light_pos - collision_pos).normalize();
  Vec3 N = SDF_normal(collision_pos, SDF);
  Vec3 R = (N * dot(L, N) * 2.0) - L;
  Vec3 V = (camera_pos - collision_pos).normalize();

  double diffuse_factor = clamp(dot(L, N), 0.0, 1.0);
  double specular_factor = pow(clamp(dot(R, V), 0.0, 1.0), specular_exponent);

  Vec3 diffuse = attenuation * diffuse_factor * diffuse_color;
  Vec3 specular = attenuation * specular_factor * specular_color;
  return diffuse + specular;
}
```

Furthermore, attenuation of the light source can be easily simulated once we have the collision position of the ray, with the code:

```
double atten = 1.0 / (1 + 0.1 * (light_pos - collision_pos).norm());
```

These lighting techniques can be easily expanded to several light sources by iterating over them, and adding up their contributions to the scene's illumination.

Shading, and Approximating Soft Shadows

In order to shade a certain point on the surface described by the SDF, we must know if it is illuminated directly by light. In other words, starting from the point we desire to shade, in the direction of the light source, we must know if there are any objects in between. Luckily, we have the infrastructure to do this via the ray-marching algorithm. If we arrive at a negative SDF, then there is in fact an object.

Signed distance functions also provide us an easy way to approximate soft shadows. When marching from the surface to a light source, we can keep track of how close we passed the surface, and shade proportionally. Furthermore, the closer we are to the object providing the darker the shade (i.e. attenuation).

```
double compute_shading(Vec3& light_pos, Vec3& collision_pos, double (*SDF)(const Vec3&)) {
  int k = 1;
  const Vec3 direction = light_pos - collision_pos;
  double res = 1.0;
  double t = 0.001;

  for (int i = 0; i < SHADE_ITERATIONS; i++) {
    double d = SDF(collision_pos + t * direction);
    if (d < 0.0001) return 0.0;
    res = min(res, k * d / t);
    t += d;
```

```
  }
  return res;
}
```

This technqiue is from Inigo Quilez (https://iquilezles.org/www/articles/rmshadows/rmshadows.htm).

Camera Movement

In the creation of demoscenes, movement is central. In order to enable the movment of camera, I implemented a `Dolly` class. This facilitates the generation of animations. To do so, I render an image for each camera position/frame in the animation, and the stitch them together into a GIF.

The API allows for scheduling certain camera movements in order, and then requesting frame locations from which to render the scene. Namely, a camera is defined by a position and a direction. The movements implemented are:

- `Dolly(position, direction)`: initialize the position and direction of the camera
- `rotate(radius, degrees, steps)`: rotates the camera around a center located `radius` units in its current direction.
- `translate(destination, steps)`: moves the position of the camera to destination, while maintaining the cameras direction constant. This allows for linear pedestal, dolly, and truck shots.
- `pan(degrees, steps)`: rotates the camera's direciton, while maintaining its position constant.

A sample usage of the API is:

```
Dolly camera_rig(Vec3(0, 0, 3), Vec3(0, 0, -1));
camera_rig.set_rotate(3, 180, 10);
camera_rig.set_pan(270, 9);
camera_rig.set_translate(Vec3(0, 1, -3), 5);
//...
for (int n_frame = 0; n_frame < num_frames; n_frame++) {
  frame_t next_frame = camera_rig.get_next_frame();
  render(frame_id, next_frame.pos, next_frame.dir);
}
```

Sources

- Ray Marching and SDFs: http://jamie-wong.com/2016/07/15/ray-marching-signed-distance-functions/
- TinyKaboom: https://github.com/ssloy/tinykaboom/wiki/KABOOM!-in-180-lines-of-code
- Fractal SDFs: http://blog.hvidtfeldts.net/index.php/2011/06/distance-estimated-3d-fractals-part-i/
- Inigo Quilez Ray Marching: https://www.iquilezles.org/www/articles/raymarchingdf/raymarchingdf.htm