HeyGears SDF Documentation

Austin Lau

February 2021

1 Introduction

The Shape Diameter Function (SDF) is used to calculate the local thickness of an input mesh at every vertex in the mesh. For every vertex, a user-specified number of rays are cast in a cone centered around its inward-facing normal. The weighted average of the (non-outlier) distance values collected from these raycasts is then taken and added to a vector that contains the SDF values of all vertices.

SDF::computeSDF takes the following parameter inputs:

Input	Description
Mesh& **mesh**	Input mesh
double **angle**	Bounding cone angle on $(0, \frac{\pi}{2}]$ for raycasting
int **rayCount**	Number of rays cast per vertex
double **epsilon**	Minimum distance allowed for successful raycast
vector <double>& **result**	Empty vector to be filled with SDF values

2 Algorithm for Raycasting in a Cone

In order to cast a cone of rays from a vertex, we need a way to pick *uniformly-distributed* random unit vectors within a bounded cone region to be used as direction vectors for each raycast.

First, let us consider the simpler problem of generating a *uniformly-distributed* random unit vector in a cone with angle θ_{max} centered on the z-axis in 3d space. (Note that θ_{max} is equivalent to **angle** in the code input above.)

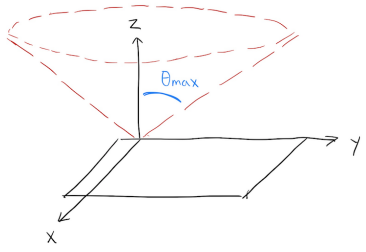

Figure 1: The bounding cone defined by θ_{max}

Consider a random unit vector $\mathbf{A}$ within the volume bounded by the cone. In spherical coordinates, $\mathbf{A}$ is defined by 2 values: θ (angle from the z-axis) and ϕ (angle from the x-axis). We need not consider the radial distance value because $\mathbf{A}$ is a unit vector so it is always 1.

To obtain the random vector $\mathbf{A}$, we need to choose:

$$\phi \sim Uniform[0, 2\pi] \text{ and}$$

$$\theta = \arccos z, \text{where } z \sim Uniform[\cos\theta_{max}, 1]$$

Choosing θ like this ensures the vectors are uniformly distributed in the cone, as directly using $\theta \sim Uniform[0, \theta_{max}]$ will give vectors in the right range but with an incorrect distribution. For further reading on why this works, see this analogous Wolfram article on picking points on spheres.

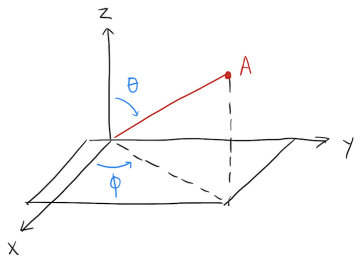

Figure 2: Unit Vector $\mathbf{A}$ in a cone centered on (0, 0, 1)

The Cartesian coordinates for $\mathbf{A}$ are:

$$x = \sin\theta\cos\phi$$

$$y = \sin\theta\sin\phi$$

$$z = \cos\theta$$

In vector form:

$$\mathbf{A} = (\sin\theta\cos\phi\,\mathbf{u}, \sin\theta\sin\phi\,\mathbf{v}, \cos\theta\,\mathbf{w})$$

Where $\mathbf{u} = (1, 0, 0), \mathbf{v} = (0, 1, 0), \mathbf{w} = (0, 0, 1)$. $\mathbf{u}, \mathbf{v}, \mathbf{w}$ are an orthonormal basis, in this case composed of the standard unit vectors along the cardinal axes.

The above vector equation is the general form of any vector $\mathbf{A}$ with respect to an *arbitrary* coordinate system, as long as $\mathbf{u}, \mathbf{v}, \mathbf{w}$ are an orthonormal basis, θ is the angle between $\mathbf{A}$ and $\mathbf{w}$, and ϕ is the angle between the projection of $\mathbf{A}$ on the plane specified by $\mathbf{u}$ and $\mathbf{v}$, and $\mathbf{u}$.

Thus, to generate a random vector $\mathbf{A}$ in a cone centered around an *arbitrary* vector, we can call our center vector $\mathbf{w}$. Next, we generate $\mathbf{u}$ and $\mathbf{v}$ such that $\mathbf{u}, \mathbf{v}, \mathbf{w}$ are an orthonormal basis. Finally, we randomly pick ϕ on Uniform$[0, 2\pi]$, pick z on Uniform$[\cos\theta_{max}, 1]$ and set $\theta = \arccos z$. Our random vector follows the same equation as before, but with new $\mathbf{u}, \mathbf{v}, \mathbf{w}$:

$$\mathbf{A} = (\sin\theta\cos\phi\,\mathbf{u}, \sin\theta\sin\phi\,\mathbf{v}, \cos\theta\,\mathbf{w})$$

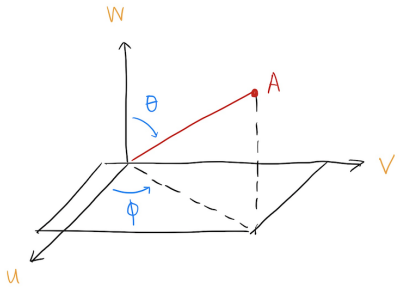

Figure 3: Unit Vector $\mathbf{A}$ in a cone centered on an arbitrary $\mathbf{w}$

The C++ code to generate a random ray in a cone is as follows, where **rayNormal** is the center of the cone. Note that **rayNormal** in this code is equivalent to the vector $\mathbf{w}$, and **randomRay** is equivalent to $\mathbf{A}$ from the equations above.

```
// pick u to be normal to rayNormal
Vector3 u = Vector3(-rayNormal.z(), 0, rayNormal.x());
u.Normalize();
Vector3 v = rayNormal.Cross(u);
v.Normalize();
double phi = randomDouble(0, 2 * PI);
double z = randomDouble(cos(angle), 1);
double theta = acos(z);
Vector3 randomRay = (u * cos(phi) + v * sin(phi)) * sin(theta) +
    rayNormal * cos(theta);
randomRay.Normalize();
```

3 Final Thickness Calculations

As rays are cast for a vertex, *successful* subdistances and their corresponding weights are recorded in a vector. Only subdistances from raycasts where the angle between the normal of the raycast origin and the normal of the face of intersection is greater than $\frac{\pi}{2}$ are considered *successful*. This is done to filter out false intersections with faces on the same side of the mesh as the origin vertex.

(The code also filters out intersections with any face directly neighboring the origin vertex.)

After all **rayCount** rays have been cast for a single vertex, the SDF is calculated from the collected subdistance values using a weighted average. Only subdistances within 1 standard deviation of the median subdistance are included in the weighted average, in order to remove outliers.

The weight formula used is $\frac{1}{\theta}$, where θ is the angle between the center of the cone and the direction of the raycast.

If we denote:

- d_i = the subdistance of the i^{th} successful raycast (from *KDTreeSAH*)
- θ_i = the angle (with respect to the cone center) of the i^{th} successful raycast
- $w_i := \frac{1}{\theta_i}$ = the weight of the i^{th} successful raycast
- d_m = the median of all subdistances for this vertex
- σ = the standard deviation of all subdistances for this vertex

$$\text{Thickness} = \frac{\sum\limits_{\substack{i \\ d_m-\sigma \leq d_i \leq d_m+\sigma}} d_i * w_i}{\sum\limits_{\substack{i \\ d_m-\sigma \leq d_i \leq d_m+\sigma}} w_i}$$

In the event a thickness is unable to be calculated,

$$\text{Thickness} = \begin{cases} -1 & \text{all raycasts are missed intersections} \\ -2 & \text{Thickness} > 100 \text{ from the formula above} \end{cases}$$

This process of generating random vectors in cones and calculating SDF Thicknesses is repeated for each vertex in the input mesh.

4 Parameter Selection

angle

angle can take values on the interval $(0, \frac{\pi}{2}]$, but $\frac{\pi}{3}$ performs well on most models. Setting **angle** too low can result in over-sensitivity to local mesh features. However, setting **angle** too high can result in taking measurements from unwanted regions of the mesh far from the target area.

Casting too wide a cone can result in an overestimation or underestimation of results. Consider the case of a sphere, where increasing **angle** will result in an

underestimate of thickness. As **angle** is increased, the rays cast will intersect at shorter and shorter distances (red lines in figure 4). However, the ground truth thickness is actually the diameter of the sphere (blue line in figure 4). As **angle** is increased, the shorter red lines will outweigh the blue line by a greater margin in the weighted average calculation and result in an underestimate.

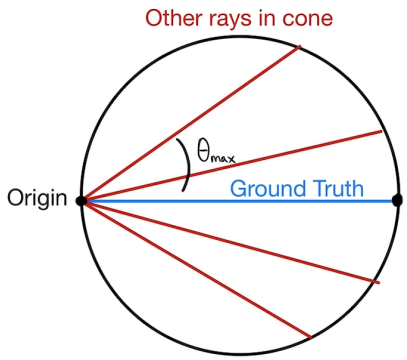

Figure 4: Distances from a large cone are smaller than the ground truth, bringing down the weighted average

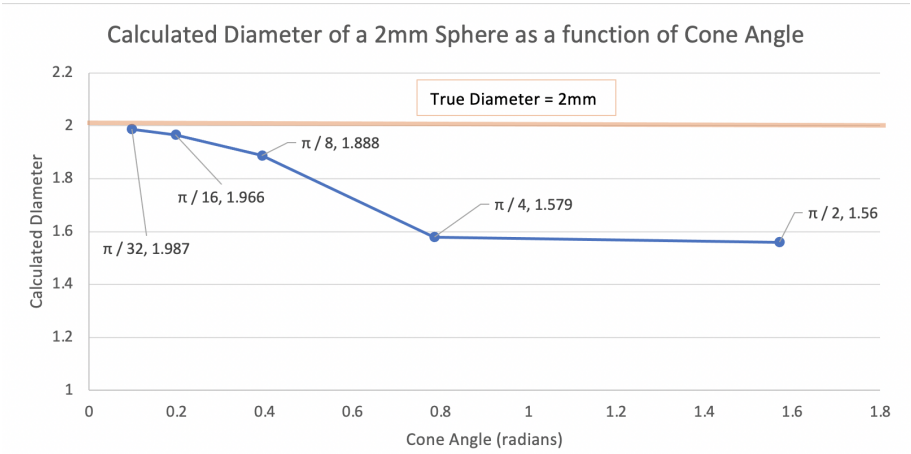

Figure 5: Increasing **angle** leads to an underestimation of thickness for a spherical mesh

The graph above illustrates how the SDF-calculated diameter of a 2mm sphere model decreases as **angle** is increased. The following code was used for these calculations, with all parameters remaining constant besides **angle**:

```
sdf.computeSDF(mesh, angle, 50, 0.001, thicknesses);
```

Though the ideal **angle** for a sphere would be as close to 0 as possible, this is not the case for all models, especially those that are not as smooth as a sphere and have lots of detail and local features. For example, using tiny angle gives very inaccurate results on dental mesh inputs.

rayCount

Increasing **rayCount** will increase accuracy at the cost of a longer runtime. If this parameter is too small there is a higher probability of having missed raycasts. Setting **rayCount** between 25 and 50 is sufficient for most models.

Figure 6 shows the effect of varying **rayCount** on the number of missed vertex intersections when raycasting. The following code was used for these calculation, with all parameters remaining constant besides **rayCount**:

```
sdf.computeSDF(mesh, PI/3, rayCount, 0.001, thicknesses);
```

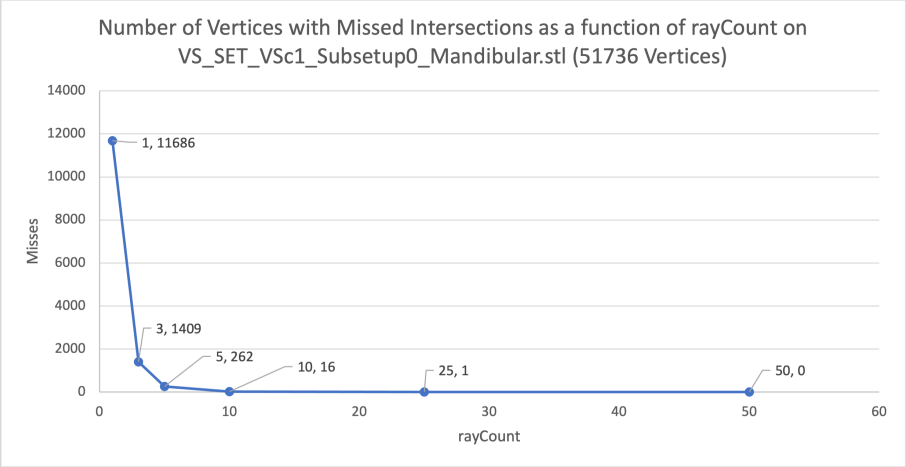

Figure 6: The effect of increasing rayCount on the number of missed intersections

epsilon

epsilon is a parameter that is passed through to the raycasting intersection function from *KDTreeSAH*. If the distance returned by this function is less than **epsilon**, the raycast is counted as a missed intersection. (Keeping this value as small as possible will keep results accurate.)

5 References and Other Notes

The SDF code is based off a description of the Shape Diameter Function from the paper *Consistent Mesh Partitioning and Skeletonisation using the Shape Diameter Function* by Lior Shapira et al. Link to paper.

This research paper defines the weight used in the weighted average as $\frac{1}{\theta}$, where θ is the angle between the raycast direction and the center of the cone it was cast within. Though I have tested other weight formulas such as $\frac{1}{\theta^2}$, other powers of θ, and the dot product of the ray normal with the normal of the face it intersects (to emphasize rays that intersect sections of the mesh more normal to the origin region), I could not prove these to be better than the weight described in the paper. However, it may be beneficial to look into different weight formulas in the future.

The paper is also where my formula of only counting subdistances within 1 standard deviation of the median distance comes from. Changing these bounds (for example to the first and third quartile) is another factor to consider experimenting with.

The greatest current weakness of this code is the influence of input cone **angle** size on results. There is no perfect value of **angle** that works best on all models, as seen in the sphere example previously. To increase the robustness of the SDF, it may be beneficial to consider an addition to this algorithm that resizes the raycasting cone depending on the region that rays are being cast on to.

List of Related Papers:

Fast and robust shape diameter function

- Computes the SDF on offset surface objects instead of the original input mesh.

Fast Approximation of the Shape Diameter Function

- Uses the Poisson Equation to approximate the SDF and reduce computation time.

GPU-based Approaches for Shape Diameter Function Computation and Its Applications Focused on Skeleton Extraction

- Uses depth peeling and the GPU to adaptively compute thickness values instead of casting a cone of rays.