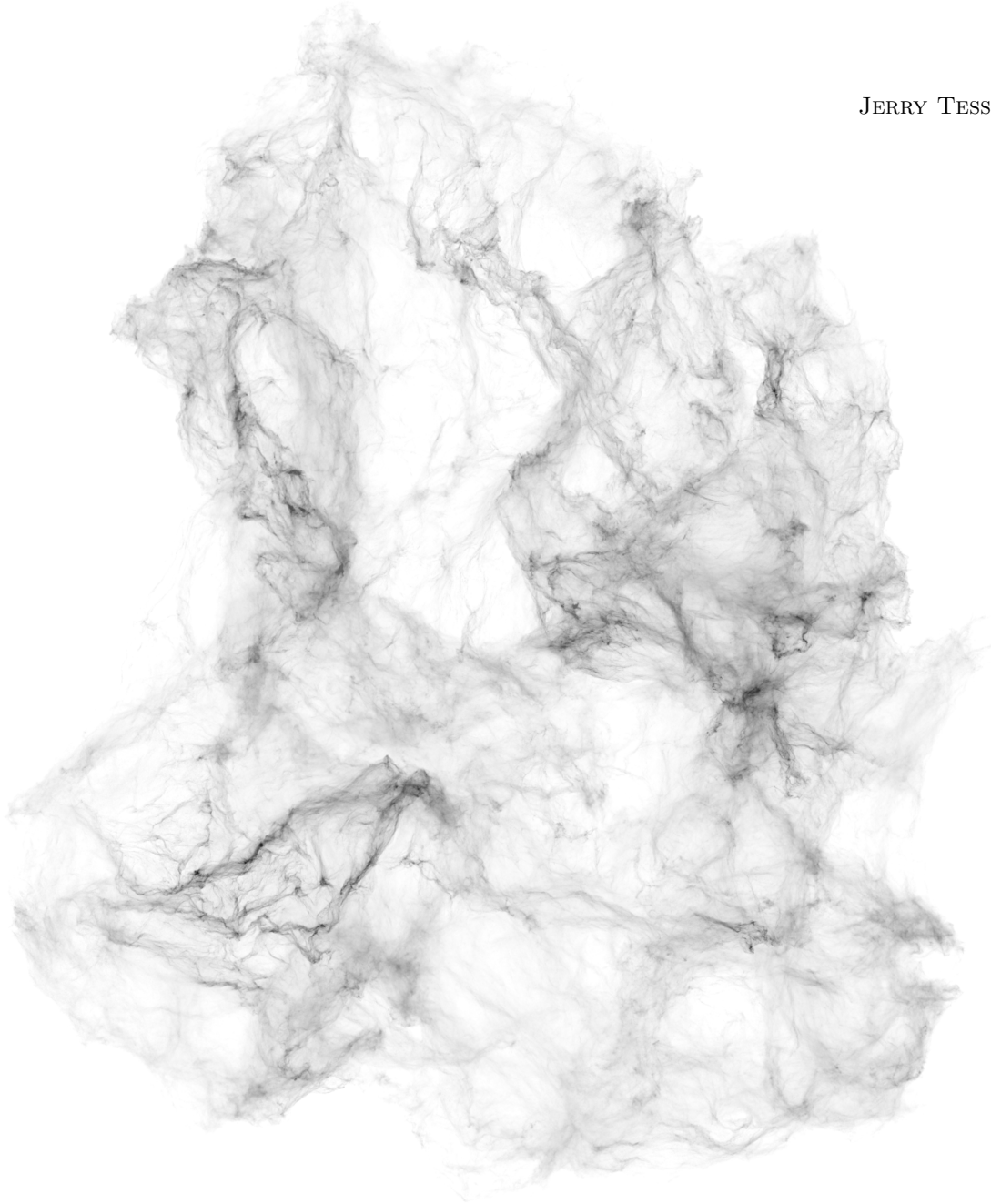


VOLUME MODELING AND RENDERING

JERRY TESSENDORF

2019



CONTENTS

1	Fields, Algebra, Transformations	7
1.1	Basic Definition of Fields	7
1.2	Gradients	8
1.3	Algebra	8
1.4	Transformations	10
2	Implicit Functions	15
2.1	Starters	15
2.2	A library of implicit functions	16
2.3	Constructive solid geometry (CSG)	18
2.4	Using implicit functions as density in volume rendering	22
3	Volume Rendering	23
3.1	Rendering Equation	23
3.2	Ray Marching	27
3.3	Deep Shadow Maps	29
3.4	Camera Model	29
3.5	Accelerating Ray Marching with Axis-Aligned Bounding Boxes	30
3.6	Common Rendering Artifacts	33
3.7	Multiple Scattering	33
3.8	Blackbody Emission	33
3.9	Curved Ray Marching	33
4	3D Grids	34
4.1	Full grids	35
4.2	Sparse Grids	36
4.3	Frustum-Shaped Grids	36
4.4	Spherical Solid Angle Grids	38
4.5	Wrapping Grids in Fields	38
4.6	Stamping Fields into Grids	39
4.7	Higher Order Interpolation of Gridded Data	40
4.8	Finite Difference Gradient	43

5	Signed Distance Functions and Level Sets	45
5.1	Signed Distance Function	45
5.2	Level Set	46
5.3	Geometry vs Level Set	47
6	Noise	48
6.1	Pseudo-Random Number Generators	48
6.2	Spatial Noise	49
6.3	Fractal Sums of Spatial Noise	49
6.4	Terrain Noise	49
6.5	Implicit Function + Noise	49
6.6	Pyroclastic Sphere	49
7	More Implicit Function Manipulations	50
7.1	Smoothing	50
7.2	Roughening	50
7.3	Distortion	50
7.4	Fracturing Geometry	50
8	Particles	51
8.1	Particle Attributes	51
8.2	Guide and Child Particles	51
8.3	Random Walks	51
9	Stamping Noise into Grids	52
9.1	Noise Clouds	52
9.2	Spline and Surface Noise	52
9.3	Wisps	52
9.4	Spline and Surface Wisps	52
9.5	3D Fractal Flame Wisps	52
10	Vector Fields	54
10.1	Closest Point Transform	54
10.2	Near Point Transform	55
10.3	Composing Maps	57
10.4	Texture Coordinate Transfer	57
10.5	Velocity Fields	57
10.6	Incompressibility	57
11	Displacements	58
11.1	Pyroclastic Sphere	58
11.2	Pyroclastic Displacement of Arbitrary Geometry	60
11.3	Cumulo	61
11.4	Displacement via Local Transformations	64

12 Advection	69
12.1 Semi-Lagrangian	69
12.2 BFECC	69
12.3 Modified-MacCormack	69
12.4 Characteristic Maps	69
12.5 Vaporous Emission Using Advection	69
13 Puffy Clouds	70
14 Fluids	71
14.1 Navier-Stokes Equations and Solver Splitting	72
14.2 Advection	72
14.3 Incompressibility	72
14.4 Boundary Conditions and Objects in Flows	72
14.5 Reflection Solver	72
14.6 Examples	72
14.7 Gridless Advection	72
14.8 Semi-Lagrangian Mapping	72
14.9 Rendering Extreme Detail	72
14.10 Compressible Gases	72
14.11 Bernoulli Waves on Arbitrary Surfaces	72
15 Volumetric Compositing	73
15.1 Value Compositing	73
15.2 Map Compositing	73
A Particle Editing	74
A.1 Notes on Editing Particle Databases to Create Visual Detail	74
A.2 A Small Gallery of Particles	105
B Sparse Grid Volume Rendering on a GPU	112
B.1 Introduction	112
B.2 Background	113
B.3 C++/C implementations	114
B.4 OpenCL implementations	119
B.5 Results	121
B.6 Conclusion	124

LIST OF FIGURES

2.1	Volume render of the implicit function of a sphere.	16
2.2	CSG union of two spheres.	19
2.3	CSG intersection of two spheres.	19
2.4	CSG cutout of one sphere from another.	20
2.5	An illustration of blending of two implicit functions. Left: A torus and box, unioned together. Right: The Blinn blend of the torus and box.	21
2.6	A shell sphere, cut away to show the thickness of the shell.	22
3.1	The Henyey Greenstein phase function for $g = 0.99, 0.5, -0.5, -0.99$	25
3.2	The Fournier-Forand phase function for $\mu = 0.35, 0.4, 0.45, 0.5$. The parameter n has the value 1.05. Petzold's measured phase functions for clear, coastal, and turbid ocean waters are shown also.	26
4.1	A frame from the DPA MFA Thesis project <i>3D Fractal Flame Wisps</i> by Yujie Shu [22]. Note the extremely fine distribution of particles from the wisp-like algorithms employed. The wisp data was stored in a frustum shaped volume matching the camera frustum.	37
4.2	Multiple spheres stamped into a volume with various selections of blending operation.	39
5.1	Demonstration of the conversion steps polygon \rightarrow level set \rightarrow polygon, for well-matched surface and grid. (a) Original polygonal surface; (b) Level set in OpenVDB format, showing the active voxels; (c) Polygon surface generated from level set. Note that (a) and (c) are identical surfaces.	47
5.2	Demonstration of the conversion steps polygon \rightarrow level set \rightarrow polygon, for a "low-resolution" grid. (a) Original polygonal surface; (b) Level set in OpenVDB format, showing the active voxels; (c) Polygon surface generated from level set. The surface in (c) has lost details and features contained in the original surface.	47
9.1	Several frames from a short film of evolving 3D fractal flame wisps. Artist: Yujie Shu.	53
11.1	Examples of classic pyroclastically displaced implicit spheres.	59
11.2	Pyroclastic displacement of a sphere using (a) positive displacement; (b) negative displacement; (c) both positive and negative displacement	60
11.3	Pyroclastic displacement as a function of γ . (a) $\gamma = 0.2$; (b) $\gamma = 1$; (c) $\gamma = 2$	61
11.4	Implicit surfaces and their pyroclastic displacements. (a) Torus; (b) Box.	62

11.5	Cumulus clouds with multiple layers of bumps.	63
11.6	Pyroclastic sphere generated from a PPT with Perlin noise normal displacement.	65
11.7	Pyroclastic torus generated from a PPT with Perlin noise normal displacement.	66
B.1	A set of volume rendered clouds	113
B.2	The rendered 512^3 grid used for testing	117
B.3	CPU Grid Implementations - Time	118
B.4	CPU Grid Implementations - Memory	119
B.5	A $4x4x2$ grid with a partition size of 2	120
B.6	An $8x8x4$ double sparse grid	121
B.7	Voxel Allocation in Sparse Renderers	122
B.8	Render Times Using Different Grid Sizes	123

LIST OF ALGORITHMS

1	Raymarch through many segments.	32
2	Finding the closest segment.	32
3	Replacing a field with a stamped version	39
4	Algorithm for Stamping Multiple Fields into a Grid.	39
5	Iterative Computation of the CPT from the NPT.	56
6	Computing the density inside a pyroclastic sphere.	59

FIELDS, ALGEBRA, TRANSFORMATIONS

1.1 Basic Definition of Fields

The central objects of volume modeling are *fields*, which are mathematical functions in 3D space that provide a value at any point in space. There are five kinds of fields of interest:

SCALAR FIELD A field that returns a scalar value, i.e. a floating-point value.

VECTOR FIELD A field that returns a vector-value. The vector is more than just a collection of scalar values, in that it has particular transformation properties under spatial transformations of the field. More about transformations below.

MATRIX FIELD A field that returns a matrix value at any point in space. This kind of field also has particular transformation properties under spatial transformations, as discussed below.

COLOR FIELD A field that returns a color value at any point in space.

SIGNED DISTANCE FIELD An implicit function field that represents the distance of any point from the surface it represents. In these notes the convention is for positive values inside the surface, and negative values outside the surface. Signed distance fields are different from the others because in some circumstances they act like scalar fields, and in others they act like vector fields.

A scalar field $f(\mathbf{x})$ has a float value at a point \mathbf{x} in three-dimensional space, a vector field $\mathbf{f}(\mathbf{x})$ has a 3-component vector as a value at any point \mathbf{x} , a matrix field $\mathbf{f}(\mathbf{x})$ has a 3×3 matrix as its value at \mathbf{x} , a color field \mathbf{F} has a tuple of values representing color information. For most of these notes, color is represented concretely as an rgb-valued triplet at the point \mathbf{x} . A signed distance field \mathbf{f} has a float value at any point.

For our purposes, fields are defined everywhere in space, and any field we construct must have a value for any point in space. In most situations, we are interested in a value for the field in only limited regions. But we require that some value be returned from a field at any point in space as a safety precaution against unforeseen events. In most cases, a field can return 0 or the equivalent outside of the region of interest.

1.2 Gradients

We also assume that fields are smooth, in the sense of calculus, so that derivatives can be performed. This is not always true, as in the case of some representations of constructive solid geometry, but we will ignore this and pretend that in all situations fields are sufficiently smooth for the algorithms we want to execute. Should be choose to not ignore it, much can be done by introducing distribution-valued fields like Dirac delta functions.

Given a scalar field $f(\mathbf{x})$, we can immediately construct a vector field and a matrix field from it by taking one and two gradients of it. Hence we have the gradient vector field $\mathbf{g}(\mathbf{x}) = \nabla f(\mathbf{x})$ and the matrix field $\mathbf{m}(\mathbf{x}) = \nabla \mathbf{g}(\mathbf{x}) = \nabla \nabla f(\mathbf{x})$. Both of these gradients arise in volumetric algorithms discussed in these notes.

Computationally gradients are handled in two ways, but represented in a single, unified framework. Gradients of analytical volumes are computed exactly, by defining the outcome of the gradient operation along with the definition of the field values. This approach provides a mechanism for analytically handling the chain rule of calculus and even L'Hopital's rule. For fields with values represented by gridded data and an interpolation scheme (see Chapter 4), gradients are evaluated with finite-difference calculation(s) and, if appropriate, interpolation.

The exceptions to this are MATRIX FIELDS and COLOR FIELDS. In all of the algorithms discussed in these notes, a gradient will never be needed for these two fields, and so it is not necessary to build such capability in them.

1.3 Algebra

Fields can be combined in a variety of ways, which we can think of as something like an algebra. All of these combinations will come into play in later chapters of these notes. Many of them are obvious, but we want to make sure you have all of them in mind.

Fields operating with like-fields

Obviously scalar fields can be added, subtracted, multiplied, and divided. Given scalar fields f and g , the operations $f+g$, $f-g$, fg , and f/g make sense in the usual way. This leads to being able to do more advance math operations like exponentiation e^f and trigonometric operations like $\cos(f)$. The gradient operation for these fields is the obvious one, e.g. $\nabla \cos(F) = -\sin(f) \nabla f$.

There are also operations that came from constructive solid geometry, but which apply more generally to manipulating fields. We will discuss CSG in detail later (Chapter 2.3). The union operation between two fields returns the maximum of the two:

$$(f \cup g)(\mathbf{x}) = \max \{f(\mathbf{x}), g(\mathbf{x})\} \tag{1.1}$$

and the intersection is the minium of the two

$$(f \cap g)(\mathbf{x}) = \min \{f(\mathbf{x}), g(\mathbf{x})\} \tag{1.2}$$

Also worth mentioning is the cutout operation

$$(f \wedge g)(\mathbf{x}) = \min \{f(\mathbf{x}), -g(\mathbf{x})\} \tag{1.3}$$

These operations are not differentiable, and so violate our assumption that we can take derivatives whenever we want. To battle this, smooth versions of these operations have been constructed define[19]. Alternatively, we can introduce the Dirac delta function to create the gradient operation.

Vector fields combine linearly with each other and scalarfields. Given scalar fields f and g , and vector fields \mathbf{v} and \mathbf{w} , the combination

$$\mathbf{u} = f \mathbf{v} + g \mathbf{w} \tag{1.4}$$

is also a vector field. This same property holds for matrix and color fields, so

$$\mathbf{m} = f \mathbf{n} + g \mathbf{o} \tag{1.5}$$

is a valid matrix field given matrix fields \mathbf{n} and \mathbf{o} , and

$$\mathbf{C} = f \mathbf{A} + g \mathbf{B} \tag{1.6}$$

is a valid color field from color fields \mathbf{A} and \mathbf{B} .

Linear algebra

As with ordinary vectors, vector fields have an inner product ($\mathbf{v} \cdot \mathbf{w}$) that produces a scalar field and a cross product ($\mathbf{v} \times \mathbf{w}$) that produces a vector field.

Vector fields and matrix fields can be multiplied to produce vector fields. The product

$$\mathbf{u} = \mathbf{m} \cdot \mathbf{v} \tag{1.7}$$

is a vector field and

$$\mathbf{w} = \mathbf{v} \cdot \mathbf{m} \tag{1.8}$$

is a different vector field. Matrix fields can be multiplied together to produce new matrix fields

$$\mathbf{m} = \mathbf{n} \cdot \mathbf{o} \tag{1.9}$$

Matrix fields can also be produced from an outer product of two vector fields:

$$\mathbf{m} = \mathbf{v} \mathbf{w} \tag{1.10}$$

Color fields also have a product property. The multiplication of two color fields \mathbf{A} and \mathbf{B} produces a color field $\mathbf{C} = \mathbf{A} \mathbf{B}$ that is a component-by-component multiply of the color values.

Composing fields

Most of the operations this far are natural extensions of what numbers, vectors, and matrices do. A new operation that fields have is *spatial composition*. Given a vector field \mathbf{X} , and an arbitrary field f , which may be a scalar, vector, matrix, or color field, a new field f' can be build from this operation:

$$f'(\mathbf{x}) = f(\mathbf{X}(\mathbf{x})) \tag{1.11}$$

The shorthand notation for this operation is $f' = f \circ \mathbf{X}$. For gradients, the chain rule is called up. For scalarfield g , with composition $h = g \circ \mathbf{X}$, the gradient of the composition is

$$\nabla h = (\nabla \mathbf{X}) \cdot ((\nabla g) \circ \mathbf{X}) \tag{1.12}$$

and $\nabla \mathbf{X}$ is a MATRIX FIELD, and $(\nabla g) \circ \mathbf{X}$ is the gradient of the pre-composition SCALAR FIELD g , followed by composition.

Composition is a fundamental operation for volume modeling. There are many applications of it in a variety of problems.

1.4 Transformations

Just as geometry can be transformed via translation, scaling, and rotations, fields have the same operations. Vector and matrix fields have some additional rules as part of these transformations also. The mathematics of transformations for fields is not the same as for geometry however. In some aspects, the transformations appear in a way that is inverse that for geometry.

Translations

Suppose you want to translate a field $f(\mathbf{x})$ by $\Delta\mathbf{x}$. For geometry, you would change a vertex at \mathbf{x} to the new position $\mathbf{x} + \Delta\mathbf{x}$. But for fields, the translation is

$$f(\mathbf{x}) \longrightarrow f(\mathbf{x} - \Delta\mathbf{x}) \quad (1.13)$$

This moves spatial features in the field properly by $\Delta\mathbf{x}$. One way to intuitively see how this works is to imagine that the field has some feature that you are aware of occurring near the location $\mathbf{x} = \mathbf{x}^*$ in space. The translated field would have that feature at $\mathbf{x} - \Delta\mathbf{x} = \mathbf{x}^*$, which is at the position $\mathbf{x} = \mathbf{x}^* + \Delta\mathbf{x}$, or in other words the feature has translated from the position \mathbf{x}^* to $\mathbf{x}^* + \Delta\mathbf{x}$.

If we use the operator symbol $T(\Delta\mathbf{x})$ for translation, then it transforms the field f to a new field h by

$$h(\mathbf{x}) = T(\Delta\mathbf{x}) f(\mathbf{x}) = f(\mathbf{x} - \Delta\mathbf{x}) \quad (1.14)$$

Translations apply to all field types in this way.

Scaling

With scaling, the field structure grows or shrinks around a point in space. For example you might expand the field, leaving it anchored at one particular point. So scaling carries with it both the notion of expansion/contraction, and the notion of a point around which this happens. Imagine again that a field has a feature occurring at $\mathbf{x} = \mathbf{x}^*$. If we expand the field in all directions around the point \mathbf{x}_t by the factor t , then the feature will occur at $\mathbf{x} = (\mathbf{x}^* - \mathbf{x}_t)t + \mathbf{x}_t$. For a scalar field, is accomplished by transforming the field in this way

$$f(\mathbf{x}) \longrightarrow f\left(\frac{(\mathbf{x} - \mathbf{x}_t)}{t} + \mathbf{x}_t\right) \quad (1.15)$$

Suppose we define the "pure" scaling operator $S(t)$ by

$$S(t) f(\mathbf{x}) = f(\mathbf{x}/t) \quad (1.16)$$

then a proper scaling operation is

$$T(-\mathbf{x}_t) S(t) T(\mathbf{x}_t) f(\mathbf{x}) = f\left(\frac{(\mathbf{x} - \mathbf{x}_t)}{t} + \mathbf{x}_t\right) \quad (1.17)$$

We can also scale component-by-component by introducing separate scaling coefficients for each direction.

Vector fields use this scaling behavior, but also have scaling applied to the components of the vector value. One way to understand the need for this is to look at a particular type of vector field, the gradient of a scalar field. If we built a vector field via a gradient, i.e.

$$\mathbf{v}(\mathbf{x}) = \nabla f(\mathbf{x}) \quad (1.18)$$

then the gradient of the scaled field is

$$\nabla f\left(\frac{(\mathbf{x} - \mathbf{x}_t)}{t} + \mathbf{x}_t\right) = \frac{1}{t} (\nabla f)\left(\frac{(\mathbf{x} - \mathbf{x}_t)}{t} + \mathbf{x}_t\right) \quad (1.19)$$

This means that if we want the gradient of a scalar field to transform under scaling as if it is the gradient of a scaled scalar field, then the value of the vector field itself needs to be scaled. This can be accomplished by defining the "pure" scaling for vector fields as

$$S(t) \mathbf{v}(\mathbf{x}) = t \mathbf{v}(\mathbf{x}/t) \quad (1.20)$$

so that a complete scaling around a scaling point is

$$T(-\mathbf{x}_t) S(t) T(\mathbf{x}_t) \mathbf{v}(\mathbf{x}) = t \mathbf{v}\left(\frac{(\mathbf{x} - \mathbf{x}_t)}{t} + \mathbf{x}_t\right) \quad (1.21)$$

Matrix fields extend this behavior with another factor of t :

$$S(t) \mathbf{m}(\mathbf{x}) = t^2 \mathbf{m}(\mathbf{x}/t) \quad (1.22)$$

Color fields do not have a similar relationship between the components of the color and spatial behavior, so color transforms like a scalar field under scaling:

$$S(t) A(\mathbf{x}) = A(\mathbf{x}/t) \quad (1.23)$$

Signed distance fields transform like scalar fields in all situations except scaling. Since the SDF represents distance, scaling the structure of the field requires also scaling the value of the field in order to retain its meaning as distance to SDFs scale like vector fields

$$S(t) d(\mathbf{x}) = t d(\mathbf{x}/t) \quad (1.24)$$

Rotations

The same logic can be used to deduce how rotations are applied. If the field is rotated about the point \mathbf{x}_r by the rotation matrix \mathbf{R} , the feature at $\mathbf{x} = \mathbf{x}^*$ moves to

$$\mathbf{x} = \mathbf{R} \cdot (\mathbf{x}^* - \mathbf{x}_r) + \mathbf{x}_r \quad (1.25)$$

so the field dependence on location changes as

$$f(\mathbf{x}) \longrightarrow f(\mathbf{R}^{-1} \cdot (\mathbf{x} - \mathbf{x}_r) + \mathbf{x}_r) \quad (1.26)$$

and we want to define the "pure" rotation operator for scalar fields as

$$R(\mathbf{R}) f(\mathbf{x}) = f(\mathbf{R}^{-1} \cdot \mathbf{x}) \quad (1.27)$$

As with scaling, vector fields also have a component-wise modification due to rotations:

$$R(\mathbf{R}) \mathbf{v}(\mathbf{x}) = \mathbf{R} \cdot \mathbf{v}(\mathbf{R}^{-1} \cdot \mathbf{x}) \quad (1.28)$$

Matrix fields have a double application, but take into account the matrix nature of rotations:

$$R(\mathbf{R}) \mathbf{m}(\mathbf{x}) = \mathbf{R} \cdot \mathbf{v}(\mathbf{R}^{-1} \cdot \mathbf{x}) \cdot \mathbf{R}^T \quad (1.29)$$

As with scaling transformations, color fields do not have a component-wise impact and act like scalar fields:

$$R(\mathbf{R}) A(\mathbf{x}) = A(\mathbf{R}^{-1} \cdot \mathbf{x}) \quad (1.30)$$

Rotation matrix

Rotations are generally represented as 3×3 orthogonal matrices that multiply 3-component vectors. This is perfectly reasonable, but constructing them can be cumbersome. Here we provide a relatively simple recipe for constructing them.

A "pure" rotation (i.e. putting aside the translations to/from the rotation pivot point) is defined by a rotation axis $\hat{\alpha}$ and a rotation angle α . The rotation axis is a unit vector that points in the direction that the rotation pivots around. Suppose a vector \mathbf{v} is rotated. Then the rotated vector \mathbf{v}_r is constructed from the old one and the rotation axis vector. This means that the rotation vector can only have this form:

$$\mathbf{v}_r = \mathbf{v}A + \hat{\alpha}B + \mathbf{v} \times \hat{\alpha}C \quad (1.31)$$

with A , B , and C being coefficients that we need to figure out. To help us figure them out, we have three guiding principles:

1. The length of the rotated vector is the same as the unrotated vector
2. The component of the rotated vector in the direction of rotation is the same as the unrotated vector
3. The angle between the rotated vector perpendicular to the rotation axis, and the unrotated vector perpendicular to the rotation axis, is α .

The second and third items relate to components of the rotated and unrotated vector along and perpendicular to the rotation axis. The components along the rotation axis are $\mathbf{v} \cdot \hat{\alpha}$ and $\mathbf{v}_r \cdot \hat{\alpha}$. Item 2 says that these must be the same. Using equation 1.31, this expands explicitly into the condition:

$$\mathbf{v} \cdot \hat{\alpha} = \mathbf{v} \cdot \hat{\alpha}A + B \quad (1.32)$$

which solves for B as $B = \mathbf{v} \cdot \hat{\alpha}(1 - A)$.

The component of the rotated and unrotated vectors perpendicular to the rotation axis are

$$\mathbf{v}_\perp = \mathbf{v} - (\mathbf{v} \cdot \hat{\alpha})\hat{\alpha} \quad (1.33)$$

$$\mathbf{v}_{r\perp} = \mathbf{v}_\perp A + \mathbf{v}_\perp \times \hat{\alpha}C \quad (1.34)$$

Item 3 says that the angle between these vectors is the rotation angle α . This means that the inner product between them is $|\mathbf{v}_{r\perp}||\mathbf{v}_\perp| \cos \alpha$. Taking the inner product:

$$\mathbf{v}_{r\perp} \cdot \mathbf{v}_\perp = |\mathbf{v}_\perp|^2 A = |\mathbf{v}_{r\perp}||\mathbf{v}_\perp| \cos \alpha \quad (1.35)$$

which leads to

$$A = \frac{|\mathbf{v}_{r\perp}|}{|\mathbf{v}_\perp|} \cos \alpha = \sqrt{A^2 + C^2} \cos \alpha \quad (1.36)$$

Following the first criterion, and using this result for B , the length squared of the rotated vector is

$$|\mathbf{v}_r|^2 = |\mathbf{v}|^2 A^2 + (\mathbf{v} \cdot \hat{\alpha})^2 (1 - A^2) + (\mathbf{v} \times \hat{\alpha})^2 C^2 \quad (1.37)$$

But from the requirement that $|\mathbf{v}_r|^2 = |\mathbf{v}|^2$, these terms rearrange to

$$|\mathbf{v}_\perp|^2 = |\mathbf{v}_\perp|^2 (A^2 + C^2) \quad (1.38)$$

which means that $A^2 + C^2 = 1$. Finally, the final result is

$$A = \cos \alpha \tag{1.39}$$

$$B = (\mathbf{v} \cdot \hat{\alpha})(1 - \cos \alpha) \tag{1.40}$$

$$C = \sin \alpha \tag{1.41}$$

We want now to make equation 1.31 look like a rotation matrix multiplied by the vector, i.e. we want to cast it in the form

$$\mathbf{v}_r = \mathbf{R} \cdot \mathbf{v} \tag{1.42}$$

In doing this, we can see that the first term looks like

$$I \cos \alpha \cdot \mathbf{v} \tag{1.43}$$

where I is the identity matrix. Similarly the second term can be cast as

$$\hat{\alpha} \hat{\alpha} (1 - \cos \alpha) \cdot \mathbf{v} \tag{1.44}$$

The third term can also be cast as a matrix multiply by noting that the cross product can be expressed using the Levi-Civita symbol¹ ϵ_{ijk} as

$$(\mathbf{v} \times \hat{\alpha})_i = \sum_{j,k=1}^3 \epsilon_{ijk} (\mathbf{v})_j (\hat{\alpha})_k \tag{1.45}$$

This can be rearranged by creating three matrices $\tau_k, k = 1, 2, 3$, defined by $(\tau_k)_{ij} = \epsilon_{ijk}$ to be

$$\mathbf{v} \times \hat{\alpha} = \left(\sum_{k=1}^3 \tau_k (\hat{\alpha})_k \right) \cdot \mathbf{v} \tag{1.46}$$

Assembling all three terms, we end up with

$$\mathbf{v}_r = \left\{ I \cos \alpha + \hat{\alpha} \hat{\alpha} (1 - \cos \alpha) + \sum_{k=1}^3 \tau_k (\hat{\alpha})_k \sin \alpha \right\} \cdot \mathbf{v} \tag{1.47}$$

which lets us identify the rotation matrix as

$$\mathbf{R} = I \cos \alpha + \hat{\alpha} \hat{\alpha} (1 - \cos \alpha) + \sum_{k=1}^3 \tau_k (\hat{\alpha})_k \sin \alpha \tag{1.48}$$

This rotation matrix can also be written in the form of an exponentiated matrix:

$$\mathbf{R} = \exp \left\{ \sum_{k=1}^3 \tau_k (\hat{\alpha})_k \alpha \right\} \tag{1.49}$$

¹http://en.wikipedia.org/wiki/Levi-Civita_symbol

Field	Translations $T(\Delta\mathbf{x})$	Scaling $S(t)$	Rotations $R(\mathbf{R})$
Scalar f	$T(\Delta\mathbf{x})f(\mathbf{x}) = f(\mathbf{x} - \Delta\mathbf{x})$	$S(t)f(\mathbf{x}) = f(\mathbf{x}/t)$	$R(\mathbf{R})f(\mathbf{x}) = f(\mathbf{R}^{-1} \cdot \mathbf{x})$
Vector \mathbf{v}	$T(\Delta\mathbf{x})\mathbf{v}(\mathbf{x}) = \mathbf{v}(\mathbf{x} - \Delta\mathbf{x})$	$S(t)\mathbf{v}(\mathbf{x}) = t \mathbf{v}(\mathbf{x}/t)$	$R(\mathbf{R})\mathbf{v}(\mathbf{x}) = \mathbf{R} \cdot \mathbf{v}(\mathbf{R}^{-1} \cdot \mathbf{x})$
Matrix \mathbf{m}	$T(\Delta\mathbf{x})\mathbf{m}(\mathbf{x}) = \mathbf{m}(\mathbf{x} - \Delta\mathbf{x})$	$S(t)\mathbf{m}(\mathbf{x}) = t^2 \mathbf{m}(\mathbf{x}/t)$	$R(\mathbf{R})\mathbf{m}(\mathbf{x}) = \mathbf{R} \cdot \mathbf{m}(\mathbf{R}^{-1} \cdot \mathbf{x}) \cdot \mathbf{R}^T$
Color A	$T(\Delta\mathbf{x})A(\mathbf{x}) = A(\mathbf{x} - \Delta\mathbf{x})$	$S(t)A(\mathbf{x}) = A(\mathbf{x}/t)$	$R(\mathbf{R})A(\mathbf{x}) = A(\mathbf{R}^{-1} \cdot \mathbf{x})$
SDF d	$T(\Delta\mathbf{x})d(\mathbf{x}) = d(\mathbf{x} - \Delta\mathbf{x})$	$S(t)d(\mathbf{x}) = t d(\mathbf{x}/t)$	$R(\mathbf{R})d(\mathbf{x}) = d(\mathbf{R}^{-1} \cdot \mathbf{x})$

Table 1.1: Summary of transformation properties for each field type.

Combined Transformations

Table 1.1 summarizes all of the pure transformation operators and their effect on the various field types. A full transformation is a combination of a translation followed by a pure transform followed by the inverse translation. If multiple pure transforms will be performed with the same translation point, then the pure transforms can be combined together inside a single pair of transform–inverse transform operations. Mathematically, a scaling and rotation done about a single point \mathbf{x}_o can be done together:

$$T(-\mathbf{x}_o)R(\mathbf{R})T(\mathbf{x}_o) \quad T(-\mathbf{x}_o)S(t)T(\mathbf{x}_o) \quad = \quad T(-\mathbf{x}_o)R(\mathbf{R})S(t)T(\mathbf{x}_o) \quad (1.50)$$

IMPLICIT FUNCTIONS

2.1 Starters

Implicit functions and implicit surfaces are a rich and extensively studied part of computer graphics, with an enormous literature (some of them are [15, 1, 7, 3, 10, 2, 12, 4, 13, 5, 14, 6, 19]). There are many applications in movies, biology, medicine, chemistry, physics, economics, and other fields. Here we introduce basic concepts, a few interesting and useful implicit functions, and use them throughout the rest of these notes as tools for precisely crafting many types of volumetric fields. They also act as simple volumes for helping us set up volume rendering later.

Implicit functions are scalar fields in 3D space that take on both positive and negative values. If $f(\mathbf{x})$ is a scalar field, it implicitly defines a surface by the equation $f(\mathbf{x}) = 0$. Of course, the value 0 is a choice we make by convention. We could have set the function to any fixed value. But this corresponds to simply redefining the function. Our convention is also that the function is positive on the inside of this surface, and negative outside of the surface.

Implicit functions are useful for more than just defining surfaces. They also can be used as control functions for building volumetric structures.

The simplest implicit function is an equation for a sphere of radius r centered at the origin:

$$f(\mathbf{x}) = r - |\mathbf{x}| \tag{2.1}$$

The implicit equation $f(\mathbf{x}) = 0$ is satisfied by the collection of points on the surface of a sphere of radius r . This function is not a unique implicit function for a sphere. These functions also implicitly define the sphere surface:

$$r^2 - |\mathbf{x}|^2 \tag{2.2}$$

$$1 - \frac{|\mathbf{x}|^n}{r^n}, \quad n > 0 \tag{2.3}$$

All of them share the property that outside of the surface of the sphere the function is less than zero, and inside the surface the function value is positive. The first version is also an example of a signed distance function, because its absolute value is the actual distance of \mathbf{x} from the surface of the sphere, while being positive inside and negative outside.

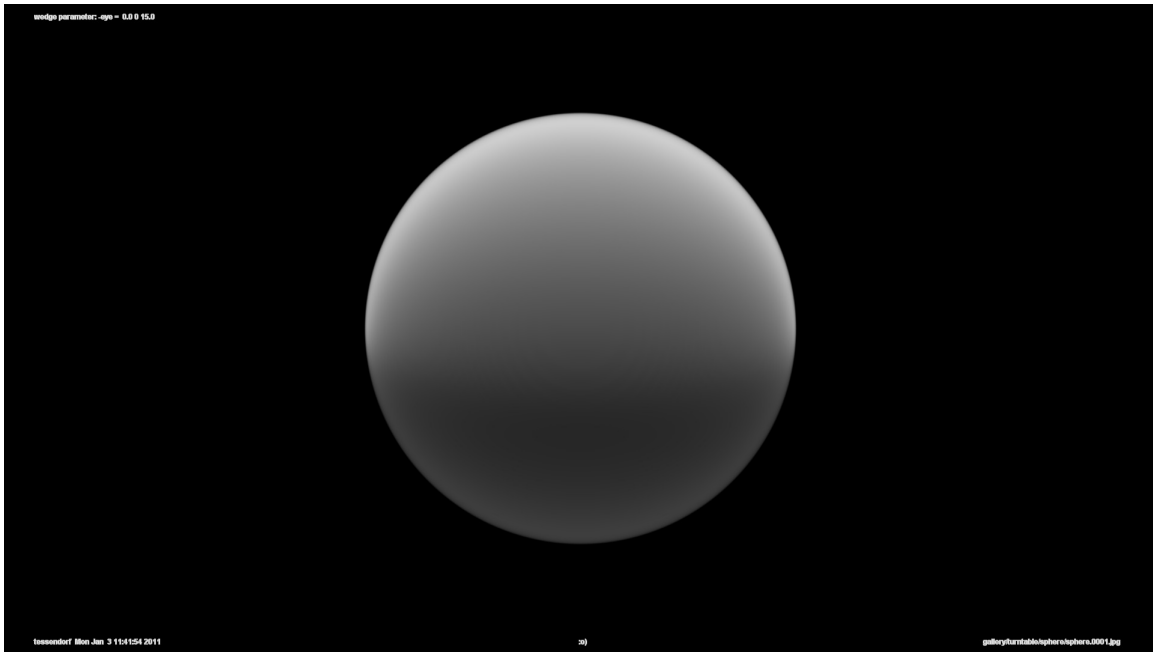
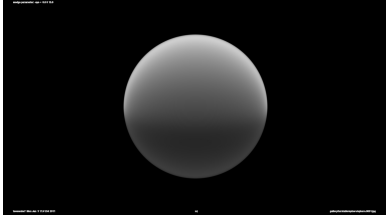
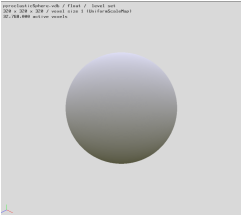


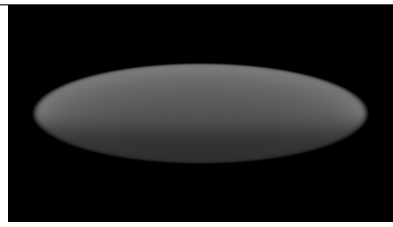
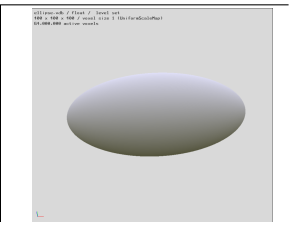

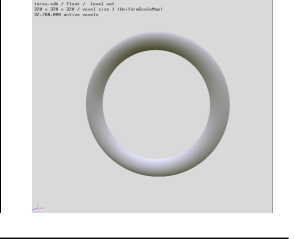
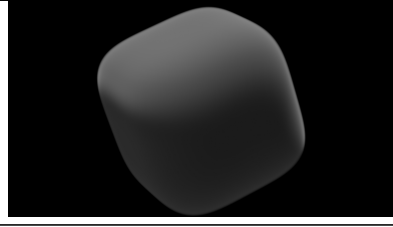
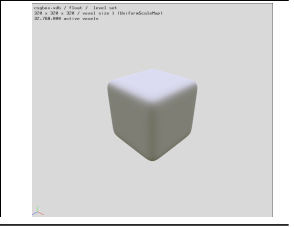
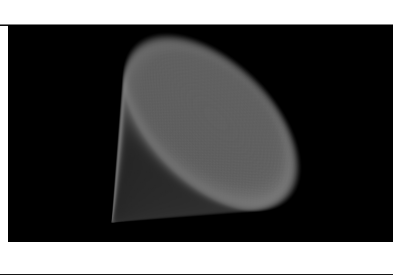
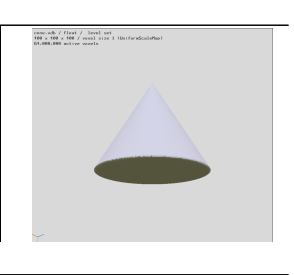
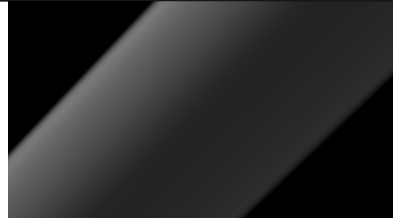
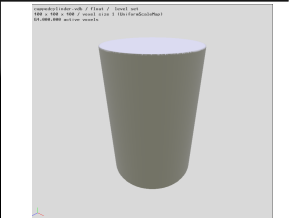
Figure 2.1: Volume render of the implicit function of a sphere.

For visualization purposes, we can use implicit functions to create a density field (see section 2.4) and render it volumetrically (chapter 3). For the implicit function of a sphere, equation 2.1, the result is in figure 2.1. The visualization comes from generating polygonal representations of the implicit surface using the methods discussed in chapter 5.

2.2 A library of implicit functions

A few handy implicit functions are listed in table 2.1. The accompanying images were generated according to the procedures in sections 2.4 and 3.

Function	Equation	Rendering	Visualization
Sphere	$r - \mathbf{x} $		

<p>Ellipse</p> $1 - \frac{Z^2}{r_{major}^2} - \frac{ \mathbf{x}_\perp ^2}{r_{minor}^2}$ $Z = \mathbf{x} \cdot \mathbf{n}$ $\mathbf{x}_\perp = \mathbf{x} - Z\mathbf{n}$		
<p>Torus</p> $4R_{major}^2 \mathbf{x}_\perp ^2 - \{ \mathbf{x} ^2 + R_{major}^2 - R_{minor}^2\}^2$ $\mathbf{x}_\perp = \mathbf{x} - (\mathbf{x} \cdot \mathbf{n})\mathbf{n}$		
<p>Box</p> $R^{2q} - x^{2q} - y^{2q} - z^{2q}$		
<p>Plane</p> $-(\mathbf{x} - \mathbf{x}_0) \cdot \mathbf{n}$		
<p>Cone</p> $\begin{aligned} \mathbf{x} \cdot \mathbf{n} &< 0 & \mathbf{x} \cdot \mathbf{n} < 0 \\ h - \mathbf{x} \cdot \mathbf{n} &> 0 & \mathbf{x} \cdot \mathbf{n} > h \\ \mathbf{x} \cdot \mathbf{n} - \mathbf{x} \cos(\theta_{max}) &> 0 & 0 < \mathbf{x} \cdot \mathbf{n} < h \end{aligned}$		
<p>Cylinder</p> $R - \mathbf{x} - (\mathbf{x} \cdot \mathbf{n})\mathbf{n} $		

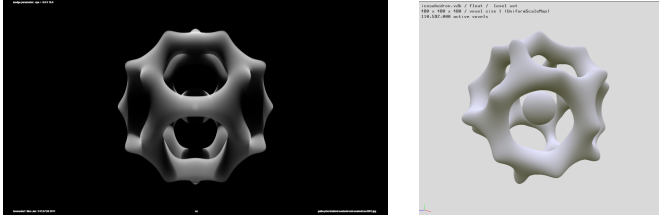
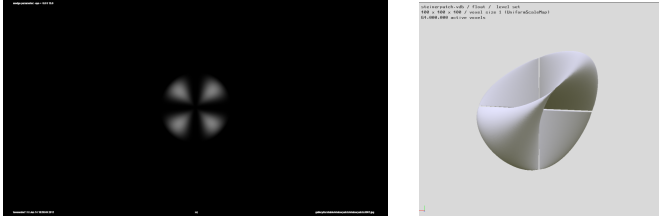
Icosahedron	$\begin{aligned} &\cos(x + Ty) + \cos(x - Ty) + \\ &\cos(y + Tz) + \cos(y - Tz) + \\ &\cos(z - Tx) + \cos(z + Tx) - 2 \\ &- 1.8\pi \end{aligned}$	$\begin{aligned} & \mathbf{x} \leq 1.8\pi \\ & \mathbf{x} > 1.8\pi \end{aligned}$	
Steiner Patch	$-(x^2y^2 + x^2z^2 + y^2z^2 - xyz)$		

Table 2.1: Equations for a few useful implicit functions

2.3 Constructive solid geometry (CSG)

Constructive solid geometry is a set of operations that allow us to combine multiple implicit surfaces into a new one. These operations are useful for volume modeling because the shaped implicit functions give more precise control of volumetric calculations.

CSG operations

There are three primary operations in CSG: union, intersection, and cutout. Unions are literally just the union of the two functions, and is built from looking at the maximum value. The union of two implicit functions f and g is denoted $f \cup g$, defined as the maximum value of the two functions:

$$(f \cup g)(\mathbf{x}) = \max\{f(\mathbf{x}), g(\mathbf{x})\} \quad (2.4)$$

Figure 2.2 shows the union of two implicit spheres.

The intersection of two implicit surfaces is the minimum of the values of their values:

$$(f \cap g)(\mathbf{x}) = \min\{f(\mathbf{x}), g(\mathbf{x})\} \quad (2.5)$$

and is illustrated in figure 2.3 for the intersection of two spheres.

The cutout operation

$$(f \wedge g)(\mathbf{x}) = \min\{f(\mathbf{x}), -g(\mathbf{x})\} \quad (2.6)$$

removes the second implicit surface from the first one, effectively cutting the shape of the second implicit surface from the first, as illustrated in figure 2.4.

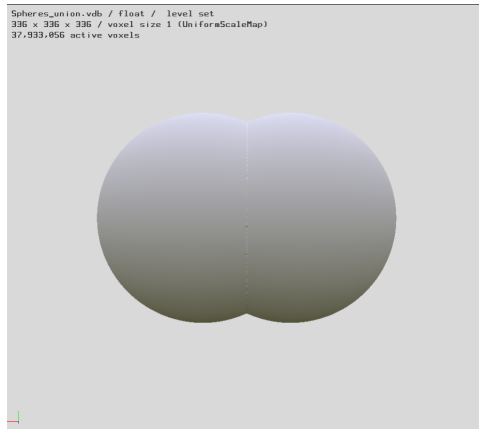


Figure 2.2: CSG union of two spheres.

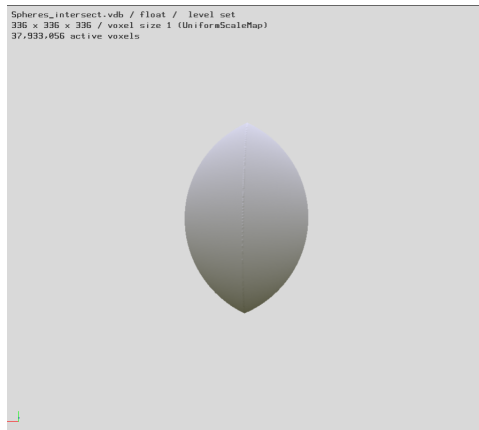


Figure 2.3: CSG intersection of two spheres.

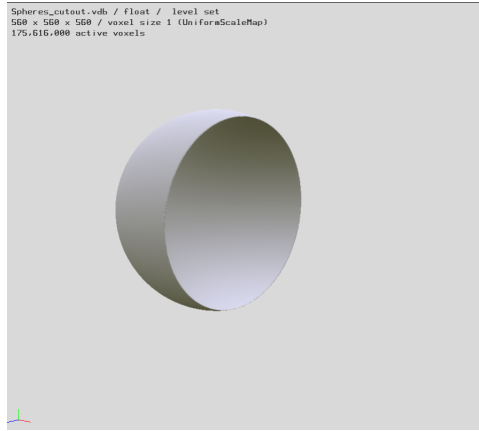


Figure 2.4: CSG cutout of one sphere from another.

Blends and other operations

Other operations of interest: masks, clamps, math, and blending

The *mask* is a form of volumetric switch. It takes an implicit function and returns 1 if the function is positive, and zero if it is negative or zero:

$$\text{mask}(f) = \begin{cases} 1 & f(\mathbf{x}) > 0 \\ 0 & f(\mathbf{x}) \leq 0 \end{cases} \quad (2.7)$$

This is useful as a switch in the following scalar field:

$$r = h \text{ mask}(f) + g (1 - \text{mask}(f)) \quad (2.8)$$

This scalar field has the value of the scalar field h inside the implicit surface f , and the value of the scalar field g outside of f .

A *clamp* limits the range of values of a scalar field:

$$\text{clamp}(f, f_0, f_1) = \begin{cases} f(\mathbf{x}) & f_0 \leq f(\mathbf{x}) \leq f_1 \\ f_0 & f(\mathbf{x}) < f_0 \\ f_1 & f(\mathbf{x}) > f_1 \end{cases} \quad (2.9)$$

Any traditional *math* function can be computed as a scalar field. For example, exponentiation of a scalar field:

$$\exp(f)(\mathbf{x}) = \exp(f(\mathbf{x})) \quad (2.10)$$

or trigonometric functions

$$\cos(f)(\mathbf{x}) = \cos(f(\mathbf{x})) \quad (2.11)$$

There are a variety of methods to *blend* implicit functions. Blends are similar to CSG operations, but are smoother. One of the common ones is the *Blinn Blend* of two implicit functions f and g into a blended third implicit function h as

$$h = \exp(f) + \exp(g) - \beta \quad (2.12)$$

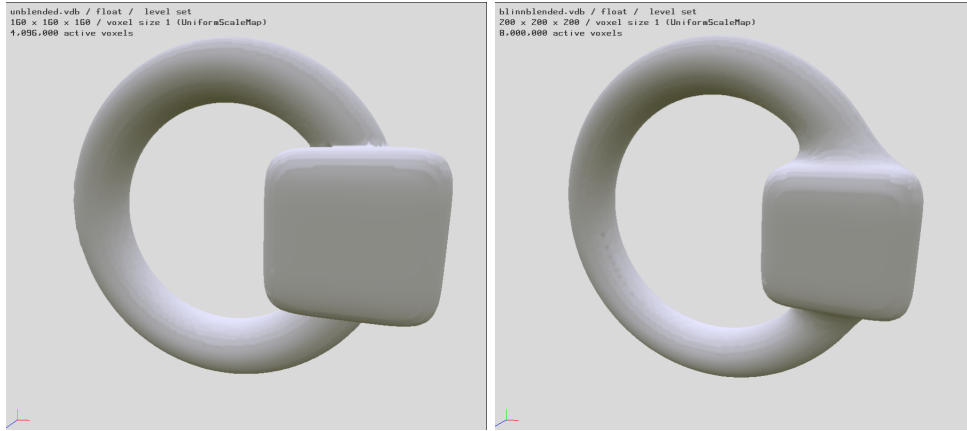


Figure 2.5: An illustration of blending of two implicit functions. Left: A torus and box, unioned together. Right: The Blinn blend of the torus and box.

The constant β defines the zero point of the implicit function. Figure 2.5 show two implicit shapes, a box and a torus, along with a blend of the two. Two other controls are possible by scaling the two implicit functions by factors s_f and s_g ,

$$h = \exp(f/s_f) + \exp(g/s_g) - \beta \quad (2.13)$$

The three constants s_f , s_g , and β can significantly change the shape of the implicit function. Small values of s_f and/or s_g cause the corresponding exponentials to rapidly switch on and off. The factor β controls how broad the implicit shape is.

The Blinn Blend of two implicit functions can be generalized to blending N implicit functions f_i , $i = 1, \dots, N$ as

$$h = \sum_{i=1}^N \exp(f_i/s_i) - \beta \quad (2.14)$$

There are other types of blending operations as well, each with advantages and disadvantages. The literature is very large.

Another operation that can be useful is adding or subtracting a constant value to an implicit function. Adding a positive amount to an implicit function increases the volume in the positive region. Subtracting value decreases the volume of the positive region. So a simple way of making an implicit shape expand/shrink is to add/subtract a value to the function. A simple application of this is to generate a shell, i.e. a hollowed out implicit function. This is done by cutting out a smaller version of an implicit function from a larger one. So if we want a shell of thickness h around an implicit shape f , then the shell is

$$\text{shell}(f) = \left(f + \frac{h}{2}\right) \wedge \left(f - \frac{h}{2}\right) \quad (2.15)$$

Figure 2.6 shows a cut away of a shelled sphere.

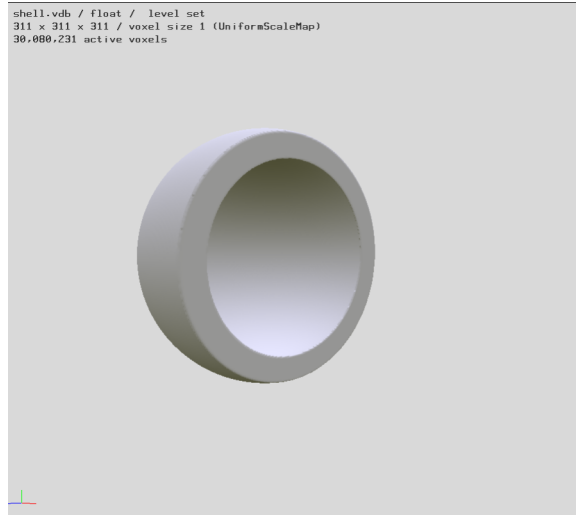


Figure 2.6: A shell sphere, cut away to show the thickness of the shell.

2.4 Using implicit functions as density in volume rendering

Implicit functions are volume rendered for two reasons. (1) Volume rendering is a way of visualizing the function by treating the negative region as invisible and the positive region as a smoke cloud. (2) Implicit functions serve in many roles as controls for trimming or extending other volumes that are rendered. So it is worthwhile to have a simple method of volume rendering any implicit function. Volume rendering is discussed in chapter 3. Here we just discuss a couple of simple procedures for making implicit functions suitable for volume rendering.

Volume rendering requires a density field, essentially a scalar field that is zero or positive everywhere. Negative values are a no-no. The negative region of an implicit function should be interpreted as transparent, which effectively means we want the density to be zero in the negative region. So one very simple definition of density is the mask function of an implicit function. Density is typically denoted by the symbol ρ , so the density field would be

$$\rho(\mathbf{x}) = \text{mask}(f)(\mathbf{x}) \tag{2.16}$$

and the density is zero everywhere outside of the implicit surface, and 1 inside the implicit surface.

The mask function is a hard step - density is either 0 or 1. This can be very difficult to render without artifacts because the ray march in volume rendering can have aliasing artifacts in such a rapid change. An alternative approach is to use the clamp function, along with scaling the implicit function:

$$\rho(\mathbf{x}) = \text{ramp}(f/s_f, 0, 1)(\mathbf{x}) \tag{2.17}$$

As with the mask, the density is zero outside of the implicit surface, and has a peak value of 1. But from the surface inward the value of the density ramps from 0 at the surface to 1 in the interior. The ramp makes the edge of the density softer and easier to volume render without artifacts.

VOLUME RENDERING

3.1 Rendering Equation

The algorithm for ray marching in volume rendering is essentially just the numerical approximation of the rendering equation for the amount of light $L(\mathbf{x}_C, \mathbf{n}_P)$ received by a camera located at position \mathbf{x}_C , at the pixel that is looking outward in the direction \mathbf{n}_P . The rendering equation accumulates light emitted by the volume along the line of sight of the pixel. The accumulation is weighted by the volumetric attenuation of the light between the volume point and the camera, and by the scattering phase function which scatters light from the light source into all directions. The rendering equation in this context is a single-scatter approximation of the fuller theory of radiative transfer:

$$L(\mathbf{x}_C, \mathbf{n}_P) = \int_0^\infty ds c^T(\mathbf{x}(s)) \kappa \rho(\mathbf{x}(s)) \exp \left\{ - \int_0^s ds' \kappa \rho(\mathbf{x}(s')) \right\} \quad (3.1)$$

The density $\rho(\mathbf{x})$ is a material property of the volume, representing the mass per unit volume present at any point in space. Note that anywhere that the density is zero has no contribution to the light seen by the camera. The ray path $\mathbf{x}(s)$ is a straight line path originating at the camera and moving outward along the pixel direction to points in space a distance s from the camera.

$$\mathbf{x}(s) = \mathbf{x}_C + s \mathbf{n}_P \quad (3.2)$$

The total color is a combination of the color emission directly from the volumetric material, and the color from scattering of external light sources by the material.

$$c^T(\mathbf{x}(s)) = c^E(\mathbf{x}(s)) + c^S(\mathbf{x}(s)) c^I(\mathbf{x}(s)) \quad (3.3)$$

Both c^E and c^S are material color properties of the volume, and are inputs to the rendering task. The illumination factor c^I is the amount of light from any light sources that arrives at the point $\mathbf{x}(s)$ and multiplies against the color of the material. For a single point-light at position \mathbf{x}^L , the illumination is the color of the light times the attenuation of the light through the volume, multiplied by the phase function for the relative distribution of light into the camera direction

$$c^I(\mathbf{x}) = c^L T^L(\mathbf{x}) P(\mathbf{n} \cdot \mathbf{n}^L) \quad (3.4)$$

with the light transmissivity being

$$T^L(\mathbf{x}) = \exp \left\{ - \int_0^D ds' \kappa \rho(\mathbf{x} + s' \mathbf{n}^L) \right\} \quad (3.5)$$

where D is the distance from the volume position \mathbf{x} and the position of the light: $D = |\mathbf{x} - \mathbf{x}^L|$, and \mathbf{n}^L is the unit vector from the volume position to the light position:

$$\mathbf{n}^L = \frac{\mathbf{x}^L - \mathbf{x}}{|\mathbf{x}^L - \mathbf{x}|} \quad (3.6)$$

For N light sources, this expression generalizes to a sum over all of the lights:

$$C^I(\mathbf{x}) = \sum_{i=1}^N c_i^L T_i^L(\mathbf{x}) P(\mathbf{n} \cdot \mathbf{n}_i^L) \quad (3.7)$$

The phase function can be any of a variety of shapes, depending on the material properties of the volume. One common choice is to ignore it as an additional degree of freedom, and simply use $P(\mathbf{n} \cdot \mathbf{n}^L) = 1$. Another choice that introduces only a single control parameter g is the Henyey-Greenstein phase function

$$P_{HG}(\mathbf{n} \cdot \mathbf{n}^L) = \frac{1}{4\pi} \frac{1 - g^2}{(1 + g^2 - 2g\mathbf{n} \cdot \mathbf{n}^L)^{3/2}} \quad (3.8)$$

This function is plotted in figure 3.1 for several values of g . As $g \rightarrow 1$, the phase function becomes sharply peaked in the forward direction, i.e. $\mathbf{n} \cdot \mathbf{n}^L \sim 1$. As $g \rightarrow -1$, the strong peak is in the backward direction, $\mathbf{n} \cdot \mathbf{n}^L \sim -1$. Phase functions have been measured and calculated for many natural materials, such as clouds, water, and tissues [9]. A model phase function called the Fournier-Forand phase function fits many natural materials well:

$$P_{FF}(\Theta) = \frac{1}{4\pi(1-\delta)^2\delta^\nu} \left[\nu(1-\delta) - (1-\delta^\nu) + (\delta(1-\delta^\nu) - \nu(1-\delta)) / \sin^2\left(\frac{\Theta}{2}\right) \right] + \frac{1 - \delta_{180}^\nu}{16\pi(\delta_{180} - 1)\delta_{180}^\nu} \{3 \cos^2 \Theta - 1\} \quad (3.9)$$

$$\delta = \frac{4}{3(n-1)^2} \sin^2\left(\frac{\Theta}{2}\right) \quad (3.10)$$

$$\delta_{180} = \frac{4}{3(n-1)^2} \quad (3.11)$$

$$\nu = \frac{3 - \mu}{2} \quad (3.12)$$

and ν , μ , and n are physical parameters. Figure 3.2 illustrates this phase function for several values of μ , along with plots of Petzold's phase function data for 3 ocean water conditions [8].

Finally, recognizing that the volumetric material occupies a finite volume of space, it is not necessary to integrate along a path from the camera to infinity. There is a point $s_0 \geq 0$ where the density starts, and a maximum distance s_{max} past which the density is zero. So the render equation can be reduced to evaluating the integral just within those bounds:

$$L(\mathbf{x}_C, \mathbf{n}_P) = \int_{s_0}^{s_{max}} ds C^T(\mathbf{x}(s)) \kappa \rho(\mathbf{x}(s)) \exp \left\{ - \int_0^s ds' \kappa \rho(\mathbf{x}(s')) \right\} \quad (3.13)$$

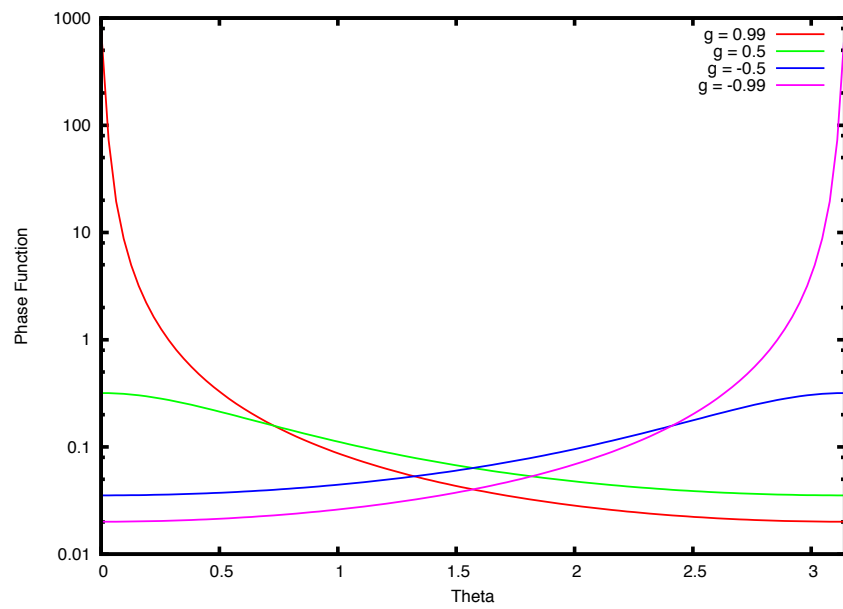


Figure 3.1: The Henyey Greenstein phase function for $g = 0.99, 0.5, -0.5, -0.99$.

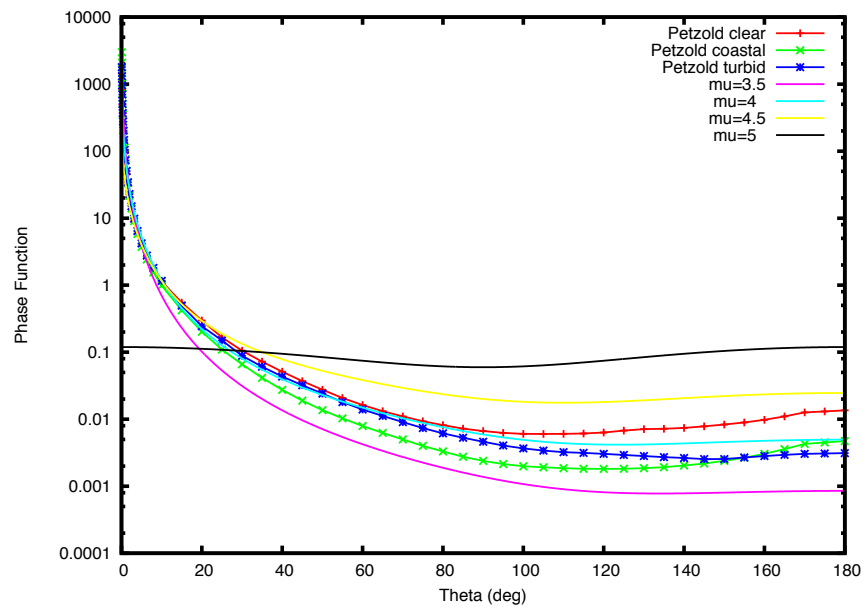


Figure 3.2: The Fournier-Forand phase function for $\mu = 0.35, 0.4, 0.45, 0.5$. The parameter n has the value 1.05. Petzold's measured phase functions for clear, coastal, and turbid ocean waters are shown also.

3.2 Ray Marching

Discretizing the rendering equation 3.13 leads to the ray march algorithm used in production volume rendering. The rendering equation 3.13 is decomposed into a set M of small steps of length Δs , with $M\Delta s = s_{max} - s_0$. Without approximation, the rendering equation becomes

$$L(\mathbf{x}_C, \mathbf{n}_P) = \sum_{j=0}^{M-1} T_j \int_0^{\Delta s} ds \, c^T(\mathbf{x}_j + s\mathbf{n}_P) \kappa \rho(\mathbf{x}_j + s\mathbf{n}_P) \\ \times \exp \left\{ - \int_0^s ds' \kappa \rho(\mathbf{x}_j + s'\mathbf{n}_P) \right\}$$

where

$$\mathbf{x}_j = \mathbf{x}_C + j\Delta s\mathbf{n}_P \quad (3.14)$$

and the transmissivity factor T_j is

$$T_j = \prod_{k=0}^{j-1} \Delta T_k \quad (3.15)$$

and

$$\Delta T_k = \exp \left\{ - \int_0^{\Delta s} ds \, \kappa \rho(\mathbf{x}_k + s\mathbf{n}_P) \right\} \quad (3.16)$$

Note that we can construct these quantities iteratively through the relationships

$$\mathbf{x}_j = \mathbf{x}_{j-1} + \Delta s\mathbf{n}_P \quad (3.17)$$

$$T_j = T_{j-1} \Delta T_{j-1} \quad (3.18)$$

with the initial conditions

$$\mathbf{x}_0 = \mathbf{x}_C \quad (3.19)$$

$$T_0 = 1 \quad (3.20)$$

which define the ray march process.

One of the first graphics papers on this problem is by Kajiya [18]. In that paper an approximation for optically thin density is applied, i.e. it is assumed that the density across a short path segment is relatively small. In these notes we do not make that assumption. In fact, only one significant assumption is made here, namely that the color field is constant across the length of a short path segment. We do not assume the optically thin approximation that Kajiya chose. This leads to a simple but significant improvement to the algorithm that solves difficulties in how the edges of clouds/smoke/whatever are handled in compositing.

The discretization step takes the form of choosing a march step size Δs that is sufficiently small that we can assume that the color c^T is constant within the length of the step Δs . With that single choice, the rendering equation reduces to

$$L(\mathbf{x}_C, \mathbf{n}_P) = \sum_{j=0}^{M-1} c^T(\mathbf{x}_j) T_j (1 - \Delta T_j) \quad (3.21)$$

This sum also can be handled via an iterative update of L . Combined with the iterations for \mathbf{x}_j and T_j the complete iteration is

$$\mathbf{x}_j = \mathbf{x}_{j-1} + \Delta s \mathbf{n}_P \quad (3.22)$$

$$L += c^T(\mathbf{x}_j) T_j (1 - \Delta T_j) \quad (3.23)$$

$$T_{j+1} = T_j \Delta T_j \quad (3.24)$$

Comparing to the optically-thin approach chosen by Kajiya, this algorithm is identical to that one *except* for the factor $(1 - \Delta T_j)$, which does not appear in Kajiya's treatment. However, if we apply an optically thin approximation, namely that $\Delta s \kappa \rho \ll 1$, then our factor reduces in the limit to just $\Delta s \rho(\mathbf{x}_j) \kappa$ which is the factor that appears in Kajiya's approach. So this ray march algorithm is an extension of Kajiya's which removes the optically-thin assumption. In practical use in production, it also has the benefit that it is easier to composite clouds rendered with this approach, because the edges of the clouds fade in opacity more correctly than the optically-thin approximation does.

The one item left to work out is the values of ΔT_j . This depends on how the density varies along the short path segment. The simplest approximation is to assume that the density is constant along the path. In that case

$$\Delta T_j = \exp(-\kappa \rho(\mathbf{x}_j) \Delta s) \quad (3.25)$$

Another possibility is that the density varies linearly along the short path segment. Suppose the density varies linearly from $\rho^0(\mathbf{x}_j)$ at the beginning of the path and $\rho^1(\mathbf{x}_j)$ at the end of the segment, then the result is similar to the constant case, but with the constant density replaced by the average density along the path.

$$\Delta T_j = \exp(-\kappa (\rho^0(\mathbf{x}_j) + \rho^1(\mathbf{x}_j)) \Delta s / 2) \quad (3.26)$$

In more general situations with the density having a complex behavior along the short path segment, we can take inspiration from the linear variation case. We can evaluate an average density $\langle \rho \rangle(\mathbf{x}_j)$ along the path segment, and arrive at

$$\Delta T_j = \exp(-\kappa \langle \rho \rangle(\mathbf{x}_j) \Delta s) \quad (3.27)$$

The average density can be evaluated, for example, by sampling the density at random positions along the path, i.e.

$$\langle \rho \rangle(\mathbf{x}_j) = \frac{1}{N_s} \sum_{i=1}^{N_s} \rho(\mathbf{x}_j + r_i \Delta s \mathbf{n}_P) \quad (3.28)$$

where the N_s numbers r_j are random numbers between 0 and 1.

If the color cannot be assumed to be constant in the interval Δs , then one approach to this is to subdivide the interval further. Here again the random sampling idea can be brought to bear. Suppose we decide to subdivide into N_s subsegments, within each we can assume that the color and density are constant. The procedure can be as follows

- generate $N_s - 1$ random numbers r_j and order them so that $r_1 < r_2 < r_3 < \dots < r_{N_s-1}$. For this notation, we can define $r_0 = 0$.

- Accumulate through the subintervals $j = 1, \dots, N_s - 1$ exactly as for the primary intervals:

$$\begin{aligned} \mathbf{x} & += r_j \Delta s \mathbf{n}_P \\ \Delta T & = \exp\{-(r_j - r_{j-1}) \Delta s \rho(\mathbf{x}) \kappa\} \\ L & += c(\mathbf{x}) T (1 - \Delta T) \\ T & *= \Delta T \end{aligned}$$

3.3 Deep Shadow Maps

When lights are used in the volume render, an additional ray march is required for each light to compute the transmissivity T^L (equation 3.5) between each light and the points \mathbf{x} on the primary ray march path. Computing these “secondary” ray marches at each point of the primary drastically slows the rendering process. However, a 3D volumetric map called a Deep Shadow Map (DSM) serves to reduce this additional computational effort substantially. DSMs have additional benefits: they are generated prior to the primary ray march, and as long as the lights and volume do not change in an animation, they do not have to be recalculated; changing the optical properties of the volume do not require changes to the DSMs.

Storing a DSM in a grid requires that values of the DSM off grid points must be obtained through an interpolation algorithm. The quantity T_L is bounded between 0 and 1, and so is not an ideal candidate for storage in a grid because many interpolation schemes can produce values outside of that range, leading to visually odd behavior. A better choice for the DSM grid is the integrated density, i.e.

$$DSM(\mathbf{x}) = \int_0^D ds' \rho(\mathbf{x} + s' \mathbf{n}^L) \quad (3.29)$$

This quantity has a minimum of 0 but is unbounded from above, reducing the visual oddity of interpolation errors. The transmissivity is then

$$T_L(\mathbf{x}) = \exp(-\kappa DSM(\mathbf{x})) \quad (3.30)$$

3.4 Camera Model

These equations for ray marching drive the specification for a pin-hole camera model in a volume rendering system. A camera C contains the following data:

1. A position in space, \mathbf{x}_C .
2. A unit vector for the view direction along the center of the image, \mathbf{n}_C .
3. A horizontal field of view, f_C , in degrees
4. An aspect ratio, a_C , consisting of the horizontal length divided by the vertical length of the image. Square images has aspect ratio of 1. An HD image has aspect ratio 16/9.
5. Near and far distances d_{near} and d_{far} , specifying the closest and farthest points of ray marching volume sampling.
6. A 3D unit vector \mathbf{v}_C defining the “up” direction of the image plane. This vector is perpendicular to the view direction \mathbf{n}_C .

7. A 3D unit vector \mathbf{u}_C defining the “horizontal” direction of the image plane. This vector is defined by $\mathbf{u}_C = \mathbf{v}_C \times \mathbf{n}_C$.

The basic model that maps points \mathbf{x} in 3D space to image plane locations \mathbf{q} is

$$\mathbf{q} = \frac{\mathbf{x} - \mathbf{x}_C}{\mathbf{n}_C \cdot (\mathbf{x} - \mathbf{x}_C)} - \mathbf{n}_C \quad (3.31)$$

This model assumes that the point in space is in front of the camera, i.e. $\mathbf{n}_C \cdot (\mathbf{x} - \mathbf{x}_C) > 0$. Note that by construction the vector \mathbf{q} is perpendicular to the view direction \mathbf{n}_C and is dimensionless. We can create horizontal and vertical components of the image plane location by using the horizontal and up directions defined for the camera:

$$\begin{aligned} u &= \mathbf{q} \cdot \mathbf{u}_C \\ v &= \mathbf{q} \cdot \mathbf{v}_C \end{aligned} \quad (3.32)$$

The horizontal field of view, f_C , and the aspect ratio, a_C of the image plane restrict the values of u, v that lie on the image plane. In particular, for the horizontal coordinate u to be in the field of view, we must have

$$-\tan(f_C/2) \leq u \leq \tan(f_C/2) \quad (3.33)$$

and for the vertical coordinate a similar expression after the aspect ratio is folded in:

$$-\frac{\tan(f_C/2)}{a_C} \leq v \leq \frac{\tan(f_C/2)}{a_C} \quad (3.34)$$

For ray marching, the inverse mapping is needed, i.e. for a given point on the image plane, a direction vector \mathbf{n}_P is needed. Suppose points on the image plane are given in a “normalized” coordinate fashion, i.e. by pairs (x, y) , with $0 \leq x \leq 1$ being the horizontal location and $0 \leq y \leq 1$ being the vertical direction. The u, v values are

$$\begin{aligned} u &= (2x - 1) \tan(f_C/2) \\ v &= (2y - 1) \frac{\tan(f_C/2)}{a_C} \end{aligned} \quad (3.35)$$

From these values the vector \mathbf{q} is built as

$$\mathbf{q} = u \mathbf{u} + v \mathbf{v} \quad (3.36)$$

Note that from equation 3.31 the vector $\mathbf{q} + \mathbf{n}_C$ is parallel the vector $\mathbf{x} - \mathbf{x}_C$ for any 3D point that maps to the location u, v . So the direction vector we seek is just the normalization of this one:

$$\mathbf{n}_P = \frac{\mathbf{q} + \mathbf{n}_C}{|\mathbf{q} + \mathbf{n}_C|} \quad (3.37)$$

3.5 Accelerating Ray Marching with Axis-Aligned Bounding Boxes

Many times, the spatial distribution of density in the volume has regions of zero density which do not contribute color or transmissivity loss. Ray marching through these regions uses computational resources without contributing interesting information to the render. Like ray tracing, acceleration data-structures and methods can be applied to (1) identify regions of zero density, and (2) skip over those regions quickly during

the march. Like ray tracing, there are many instances where this acceleration can speed up a volume render by many orders of magnitude. In this section we introduce the Axis-Aligned Bounding Box (AABB) that is familiar from ray tracing, and use the AABB, sometimes along with Kd-trees, to build the acceleration procedure.

In two dimensions, an AABB is a rectangle with edges along the primary axes. The bounds of the rectangle can be specified in terms of a lower left corner (LLC) and an upper right corner (URC). In 3D, the AABB is a rectangular box with faces along the primary axes, and again is described by a LLC and a URC. If the LLC and URC are

$$\begin{aligned} \mathbf{x}_{LLC} &= (x_{LLC}, y_{LLC}, z_{LLC}) \\ \mathbf{x}_{URC} &= (x_{URC}, y_{URC}, z_{URC}) \end{aligned} \tag{3.38}$$

the eight vertices of the AABB are

$$\begin{aligned} &(x_{LLC}, y_{LLC}, z_{LLC}) \\ &(x_{URC}, y_{LLC}, z_{LLC}) \\ &(x_{LLC}, y_{URC}, z_{LLC}) \\ &(x_{LLC}, y_{LLC}, z_{URC}) \\ &(x_{URC}, y_{URC}, z_{LLC}) \\ &(x_{URC}, y_{LLC}, z_{URC}) \\ &(x_{LLC}, y_{URC}, z_{URC}) \\ &(x_{URC}, y_{URC}, z_{URC}) \end{aligned}$$

The key aspect of AABBs is that the algorithm for ray intersection tests with them is extremely fast and efficient [24]. The intersection process has a computational signature pseudo-code like

```
bool intersect( AABB, ray, tmin, tmax )
```

If there is no intersection of the ray with the AABB, this method returns `false` and no ray march is needed. If there is an intersection, it returns `true` and the positive values of `tmin` and `tmax` are the distances from the ray to the closest and furthest intersections. If the ray begins inside the AABB, then `tmin` is zero. The ray march can skip over the intervening empty space, begin marching at `tmin` and stop at `tmax`. This makes it effective to use them to designate the most important regions to ray march, just as in ray tracing AABBs identify the regions of space that have objects to trace.

Suppose the volume is composed of two volumetric objects, with intersection distances (`tmin0`, `tmax0`) and (`tmin1`, `tmax1`). We can designate object 0 as the closer one, i.e. `tmin0 < tmin1`. There are two possible situations: the AABBs are separated, or they overlap (just touching on a face is overlap in this discussion). We are trying to establish a (`tmin`, `tmax`) pair that will ray march through a contiguous portion of volume that does not have gaps between AABBs. If the two AABBs do not overlap, then `tmin1 > tmax0`, and we should set our pair to (`tmin`, `tmax`) = (`tmin0`, `tmax0`). This ray march will terminate on the far side of object 0, so we will want to continue ray marching by advancing the ray to the position of `tmin1` and continue from there through `tmax1`.

If `tmin1 ≤ tmax0` the two volumes overlap, and the ray march pair should include both AABBs, i.e. the pair should be (`tmin`, `tmax`) = (`tmin0`, `tmax1`).

Now suppose we have a generalization of the situation. The volume is composed of N volume elements combined in some way, and for each one we have a AABB, $AABB_i$, $i = 0, \dots, N - 1$. We do not have any information about relative positioning other than the list of AABBs. The ray march along a single ray will look like a sequence of segments, each segment built from a subset of volume elements that overlap. Building this sequence requires only building the closest segment for the ray, following by iterating this for rays that begin where the previous segment left off. This algorithm is illustrated in Algorithm 1. For two volume

Algorithm 1 Raymarch through many segments.

```

procedure ITERATERAYMARCH( $\{AABB_i\}$ , Ray)
   $status \leftarrow true$ 
  while  $status$  do
    ( $status, tmin, tmax$ )  $\leftarrow$  FindClosestSegment( $\{AABB_i\}$ , Ray)
    if  $status$  then
      RayMarchAlongSegment(Ray,  $tmin, tmax$ )
      Ray  $\leftarrow$  AdvanceRayTo( $tmax$ )
    end if
  end while
end procedure

```

elements, our previous discussion gave us the FindClosestSegment() algorithm. For many unordered volume elements, this can be generalized into an iterative comparison shown in Algorithm 2.

Algorithm 2 Finding the closest segment.

```

procedure FINDCLOSESTSEGMENT( $\{AABB_i\}$ , Ray)
  ( $status, tmin, tmax$ )  $\leftarrow$  intersect( $AABB_0$ , Ray)
  for all  $aabb$  in  $\{AABB_i\}$  do
    ( $stat, t0, t1$ )  $\leftarrow$  intersect( $aabb$ , Ray)
    if  $stat$  then
      if not  $status$  then
        ( $status, tmin, tmax$ )  $\leftarrow$  ( $stat, t0, t1$ )
      else if  $t0 < tmin$  then
         $temp \leftarrow tmin$ 
         $tmin \leftarrow t0$ 
        if  $temp > t1$  then
           $tmax \leftarrow t1$ 
        end if
      else if  $t0 \leq tmax$  then
         $tmax \leftarrow t1$ 
      end if
    end if
  end for
  return  $\{status, tmin, tmax\}$ 
end procedure

```

AABBs used via Algorithms 1 and 2 can speed up volume rendering substantially. When the number of

AABBs grows to a large number (hundreds or thousands), the number of intersection evaluations can grow to the point where their cost becomes larger than the benefit from reduced ray march stepping. In that case, we can restore efficiency by adding a traditional acceleration structure such as a Kd-tree or BVH to quickly reduce the number of intersection calculations to a reasonable number.

3.6 Common Rendering Artifacts

3.7 Multiple Scattering

Monte Carlo Path Tracing and Tracking

Faux Lights

Feynman Path Integral

Feynman Path Integral Formulation for Radiative Transfer

Perturbation Expansion

Monte Carlo Path Generation in the Frenet-Serret Framework

Steepest Descents Approximation of the Feynman Path Integral

Solution for Faux Lights

3.8 Blackbody Emission

3.9 Curved Ray Marching

3D GRIDS

Gridded values of floats, vectors, and colors come into play for some volumetric algorithms. There are two reasons for this:

1. Some algorithms for constructing volumetric data depend on a grid structure. Examples are noise stamping and wisps.
2. For speed and efficiency, sampling procedural volumes to grids can be a good approach. In some situations, gridded values are problematic, but there are also some where the grid is not a problem and the speed benefits are a net win despite the higher memory usage.

For volume modeling and rendering, there are several variations of 3D grids that are useful, characterized by spatial structure and data structure. For spatial structure, rectangular grids are the simplest, and widely used because they are simple to write code for, and data read/write access is fast and simple. Frustum shaped grids are also very useful because they lay out grid elements in a pattern that is convenient for camera sampling. There are advantages and disadvantages to both, some of which are listed in table 4.1.

For the data structure, two arrangements are widely used. Full grids allocate memory to the grid in a contiguous block for all of the grid points during initialization, so even “empty” points take up memory. In contrast, Sparse grids allocate memory in one of several strategies that minimize the memory taken up by the “empty” grid points. In practice, block-partitioned schemes have proven to have a good trade-off between memory minimization, access speed, and coding complexity, and are simple enough that they can be implemented on GPUs[16].

Rectangular Grid		FrustumGrid	
Pros	Cons	Pros	Cons
Simple data structure	fixed resolution	Simple data structure	Tied closely to camera
Fast data access	unnatural shape	Fast data access	variable resolution
Multipurpose		holds only visible density	poor for animations
Flexible location			misses off-camera shadow

Table 4.1: Lists of pros and cons for rectangular and frustum grids.

In these notes we focus on 3D rectangular grids only. Frustum grids can be built from rectangular grids using a mapping functionality layered into the access logic.

4.1 Full grids

Full rectangular grids are very simple. The grid points are laid out in a regular rectangle with a fixed distance between each point. Each grid point is a corner of a “cell” of dimensions $(\Delta x, \Delta y, \Delta z)$. You can also choose to lay out the grid so that the grid points are centered inside each cell, or even at some random point inside a cell. The key point is that there is a collection of cells, and the grid point is some convenient reference location for each cell. The grid points correspond to an indexing scheme i, j, k , where $0 \leq i < N_x$, $0 \leq j < N_y$, $0 \leq k < N_z$, where N_x, N_y, N_z are the number of grid points in each direction, and the location of each grid point is

$$\mathbf{x}_{ijk} = \mathbf{x}_{llc} + (i\Delta x, j\Delta y, k\Delta z) \quad (4.1)$$

and \mathbf{x}_{llc} is the “lower-left corner” of the grid, i.e. the grid point with the lowest numeric values of all three coordinates.

Data values are stored at the grid points. We denote the gridded value at grid point ijk by g_{ijk} . These values may be scalar, vector, color, matrix, or possibly something else. A common usage will be to sample the value of a field at the grid points. The grid of values can be treated as a field also, but means of interpolation. For a point \mathbf{x} within the bounds of the grid, a linearly interpolated value is

$$\begin{aligned} g(\mathbf{x}) &= g_{ijk} (1 - w_i)(1 - w_j)(1 - w_k) \\ &+ g_{i+1jk} w_i(1 - w_j)(1 - w_k) \\ &+ g_{ij+1k} (1 - w_i)w_j(1 - w_k) \\ &+ g_{ijk+1} (1 - w_i)(1 - w_j)w_k \\ &+ g_{i+1j+1k} w_iw_j(1 - w_k) \\ &+ g_{i+1jk+1} w_i(1 - w_j)w_k \\ &+ g_{ij+1k+1} (1 - w_i)w_jw_k \\ &+ g_{i+1j+1k+1} w_iw_jw_k \end{aligned}$$

where the particular ijk values are the “nearest neighbor” grid point

$$(i, j, k) = \text{floor} \left(\frac{x - x_{llc}}{\Delta x}, \frac{y - y_{llc}}{\Delta y}, \frac{z - z_{llc}}{\Delta z} \right) \quad (4.2)$$

Outside of the volume of the grid, it is necessary to specify a value that should be returned. This is determined by the meaning of the particular data and the intended application. When used as a density field, it is usually appropriate to set the value outside of the grid to 0.

There is a significant downside to full grids. Holding a full grid in cpu memory limits the possible size of the grid. In practice, full grids much larger than approximately 2000^3 are impractical even on big machines. If multiple grids of data are needed for a task, the size of each is correspondingly smaller. But applications at film resolution frequently require grids that are on the order of $50,000^3$ size. This is because the grid must contain more structure than is directly viewed by a camera, either because the camera animates or because the out-of-view material contributes to lighting of the material in the camera view.

Fortunately, there is a way out. Practical applications of very large grids also generally have content in the grid that has a lot of “empty” space, i.e. regions where the density is 0 or some default value. This opens up the feasibility of using lossless compression schemes to hold the data in cpu memory. The method of *Sparse Grids* using block-partitioning has proven very useful for the applications encountered so far in VFX.

4.2 Sparse Grids

The major problem with full grids is that they consume a great deal of memory. Modest-resolution grids can have $2K \times 2K \times 2K$ voxels, and even on machines with sufficient memory for that, the growth curve memory for grids is steep. Two important observations about gridded volumetric data have driven grid technology to make them flexible and useable:

1. Gridded data needs to be very fine resolution to capture many kinds of interesting detail, finer in fact than can be achieved in core memory presently.
2. For most applications, there is a substantial amount of “wasted space”, i.e. regions of the volume where there is no useful information. This waste can have two forms (a) Density fields have a lot of empty space where the density is zero; (b) signed distance functions have useful information in a narrow-band around the surface, but distance information far from the surface has little value.

Because of these two observations, it is clear that there is a lot of benefit from creating sparse grids that only allocate core memory to voxels that have useful information. Multiple storage and access schemes are possible. The honors thesis [16], reproduced in appendix B, examined several of them and their relative performance properties. Of the ones they examined, the block-partition method was the most useful for storing density information, both because it can be implemented in a hierarchical fashion to allow extremely large grids, and because it can be implemented both in C/C++ and in OpenCL/CUDA with compatible data structures - allowing the option of moving gridded data between CPU and GPU systems.

A complementary option is to use out-of-core storage, moving data between RAM and disk as needed. Here again sparse storage schemes are essential so that resources are not wasted moving around useless data.

An open source sparse storage system is contained in the OpenVDB project[11]. This project includes many efficient and useful algorithms for gridded data.

A secondary value of sparse grid schemes is that they provide a convenient and efficient method of assembling axis-aligned bounding boxes for fast ray traversal in volume rendering, allowing the ray march to skip empty regions of the volume and reduce render time sometimes by several orders of magnitude.

4.3 Frustum-Shaped Grids

A very useful variation on rectangular grids are grids shaped like view frustums. The shape is achieved by distorting the shapes of voxel cells from rectangular boxes. Imaging a box that is distorted into the shape of a frustum. The cells are generated inside the distorted box by slicing along two planes that line up with image plane pixels, and one slice plane perpendicular to the view direction of the frustum. Each voxel cell in a frustum has a different shape and/or volume from any other voxel cell. If such a volume is used and the frustum is identical to the camera frustum, two advantages are realized: (1) volume data is stored in the grid only in regions that are visible to the camera, so no space is used storing out-of-view volumes; (2) voxel cells near the camera have smaller volume and higher spatial detail precisely in the region where the camera can most see that detail. Figure 4.1 illustrates these benefits. Using a camera-locked frustum grid, this

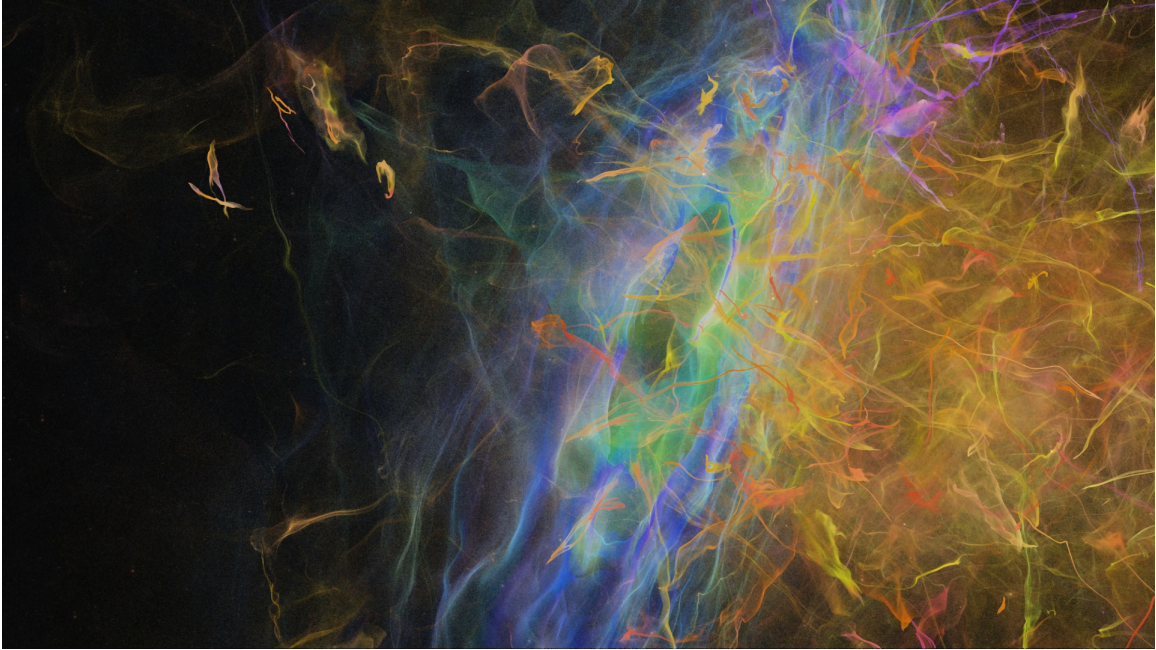


Figure 4.1: A frame from the DPA MFA Thesis project *3D Fractal Flame Wisps* by Yujie Shu [22]. Note the extremely fine distribution of particles from the wisp-like algorithms employed. The wisp data was stored in a frustum shaped volume matching the camera frustum.

scene is rendered with a 2000x2000x2000 grid. Using rectangular grids would have required sizes in excess of 5000x5000x5000.

However, there are potential downsides to using a frustum grid locked to the camera: (1) volume content that is time consuming to generate has to be regenerated each time the camera and frustum moves; (2) volume material outside the camera fov is not represented, so shadows from off-camera volumes will not be properly built.

A frustum grid is actually just a rectangular grid paired with a camera that provides the frustum data, and a frustum mapping function that maps points in space to locations in the grid. OpenVDB provides functionality for this. Here we describe a simple way to accomplish it.

A camera C as defined in section 3.4 is a convenient of specifying the parameters needed for a frustum grid mapping. What is needed is a 1-1 mapping between points in 3D space and locations in the frustum space. The frustum grid is oriented so that its x and y directions line up the the image plane of the camera, and the grid z direction corresponds to distance out from the near distance d_{near} . Assume that the number of cells in each direction is $N_x \times N_y \times N_z$. A point \mathbf{x} maps to camera-plane positions via the two step process of constructing \mathbf{q} and u, v from equations 3.31 and 3.32, then

$$x = \frac{1}{2} \left(\frac{u}{\tan(f_C/2)} + 1 \right) \quad (4.3)$$

$$y = \frac{1}{2} \left(\frac{v a_C}{\tan(f_C/2)} + 1 \right)$$

The depth value comes from the distance, formalized by the near and far values:

$$z = \frac{|\mathbf{x} - \mathbf{x}_C| - d_{near}}{d_{far} - d_{near}} \quad (4.4)$$

Within the frustum volume, x, y, z all have value ranges of 0 to 1, and convert to grid index values by

$$\begin{aligned} i &= x N_x \\ j &= y N_y \\ k &= z N_z \end{aligned} \quad (4.5)$$

Going the other way, starting with index values i, j, k , the corresponding position in 3D space comes from the steps:

$$\begin{aligned} u &= (2(i/N_x) - 1) \tan(f_C/2) \\ v &= (2(j/N_y) - 1) \tan(f_C/2) / a_C \\ Z &= d_{near} + (k/N_z)(d_{far} - d_{near}) \\ \mathbf{q} &= u\mathbf{u} + v\mathbf{v} \\ \mathbf{x} &= \mathbf{x}_C + \frac{(\mathbf{q} + \mathbf{n}_C)}{|\mathbf{q} + \mathbf{n}_C|} Z \end{aligned} \quad (4.6)$$

4.4 Spherical Solid Angle Grids

4.5 Wrapping Grids in Fields

When volume data is stored in a grid, we still want to be able to combine it with procedural volumes and even other gridded volumes. So it is important to be able to sample values of the volume data at any point in space, not just at the stored grid points. This means we need to combine the discrete volume values with an interpolation scheme to generate those values. A variety of interpolation methods could be used, e.g. trilinear interpolation, splines, hermite polynomials, etc. The interpolation scheme may also be supplemented by a boundary treatment to determine how the discrete data is extended past the bounds of the grid.

The combination of gridded volumetric data, an interpolation scheme, and boundary treatment is a wrapping procedure to make the discrete data look like a continuous field with values everywhere. We will denote such a wrapper a “gridded” field. For example, a scalar field defined from grid of scalar values g is

$$f = \text{gridded}(g) \quad (4.7)$$

Once a grid is wrapped in a field, further processing can proceed treating the data as a continuous field. Several advantages accrue from this approach:

- Processing algorithms can be implemented without the burden of grid-handling logic.
- Multiple wrapped-grid fields can be used, each with different choices for grid cell dimensions, point sizes, and locations. Choices for these parameters can be made by the user based on the best representation of the content of individual grids, without the need to fit all data into a common grid parameterization.

Figure 4.2: Multiple spheres stamped into a volume with various selections of blending operation.

4.6 Stamping Fields into Grids

Occasionally it is useful to have a representation of a field in a gridded form. That is, the values at the grid points are the value of some field at the location of those grid points. The gridded form of a field is not necessarily identical to the original field because the field can have detailed structure on a scale smaller than the cell dimensions, and it may have significant values at locations outside the scope of the grid. We label the process of sampling values from a field and storing the values on a grid as *stamping*.

A common use of stamping is to speed up the evaluation of a field. Suppose a field f is the product of an expensive collection of calculations. The combination in algorithm 3 replaces the expensive field with

Algorithm 3 Replacing a field with a stamped version

$g \leftarrow \text{stamp}(0)$	▷ Initialize grid with empty value.
$\text{stamp}(f, g)$	▷ Stamp field into grid g .
$f \leftarrow \text{gridded}(g)$	▷ Replace the field with a gridded version.

fast-evaluating interpolations of values on a grid. This approach limits the number of expensive evaluations to the number of grid points. For the gridded version to be an accurate representation of the original field, the grid parameters must be chosen based on the spatial structure of the field and the intended application.

Stamping fields is also a method of creating a complex volume by stamping many copies of fields in many locations. In this process, fields overlap and there is an additional opportunity to offer different methods of combining, or blending, fields in the overlap regions. In addition to CSG operations like union and intersection, many other blending operations are feasible. For example, addition, blinn blending, averaging, could be chosen. The appearance of the stamped volume will vary widely depending on the blend choice. Figure 4.2 illustrates the visual impact of many of these choices. For a set of fields $\{f_i\}$, algorithm 4 depicts the process. Common blend choices include

Algorithm 4 Algorithm for Stamping Multiple Fields into a Grid.

$g \leftarrow \text{stamp}(0)$	▷ Initialize grid with empty value.
$b \leftarrow \text{blend}$	▷ Choose a blend method.
for $f \in \{f_i\}$ do	
$\text{stamp}(f, g, b)$	▷ Stamp field into grid g using blend method b .
end for	

- **Add** Simple add of the current field value to the accumulated value at the grid point.
- **Replace** Replace the grid point value with the current field value.
- **Max** Replace the grid point value with the maximum of the grid point value and the current field value.
- **Min** Replace the grid point value with the minimum of the grid point value and the current field value.
- **Alpha** Set the value at the grid point with $g\alpha + f(1 - \alpha)$, for a predefined value $0 \leq \alpha \leq 1$.

For maximum control and variability, particles are an efficient method of controlling the properties of stamping. Each particle in a particle system can represent a field to be stamped into a grid. The attributes of the particles can drive the location of the field, scale, transformation, and even blending selection.

4.7 Higher Order Interpolation of Gridded Data

In section 4.1, linear interpolation was set out as the basic method for treating gridded data as a continuous field. Other types of interpolation could also be brought to bear. Depending on the specifics of the interpolation scheme, various qualities can be preserved or introduced. In this section, we construct a family of interpolation schemes that seek to treat the gridded data as if it is a sampling of a continuous function, and the interpolation seeks to restore a more accurate reconstruction of the function.

We begin with a derivation of the scheme in one dimension. The extension to three dimensions that follows is straight forward.

One Dimensional Derivation

Suppose $g(x)$ is a continuous function in one dimension. Assuming a grid of points in one dimension $x_m = m\Delta x, m = -\infty, \dots, -2, -1, 0, 1, 2, \dots, \infty$, with the value of g known only at those grid points $g_m \equiv g(x_m)$, the interpolation problem is to reconstruct the value of the function at any point x not on a grid point. We begin by decomposing the position x as $x = x_m + y$, where x_m is the nearest grid point “to the left” of x :

$$x_m = \begin{cases} \text{floor}(x/\Delta x) \Delta x & x > 0 \\ (\text{floor}(x/\Delta x) - 1) \Delta x & x < 0 \end{cases} \quad (4.8)$$

and y is the residual “distance” of x from that grid point. By construction, $y \geq 0$. The interpolation is generated directly from the collection of values in the neighborhood surrounding x_i :

$$g(x) \approx \sum_{j=-N+1}^N g(x_{m+j}) \omega_j \quad (4.9)$$

where $2N$ is the size of the neighborhood, and ω_j is the collection of weights for the interpolation. Basic linear interpolation can be written this way, with $N = 1, \omega_0 = 1 - y/\Delta x, \omega_1 = y/\Delta x$.

The goal here is to allow N to be arbitrary sized, and construct the collection of weights ω_j for any choice of N , independent of the particular function under interpolation.

The derivation of a systematic approach begins by observing the form of equation 4.9 when the function has a Fourier transform representation, i.e. the general function can be written

$$g(x) = \int_{-\infty}^{\infty} dk \tilde{g}(k) e^{ikx} \quad (4.10)$$

Inserting this into equation 4.9 and rearranging, we get

$$0 \approx \int_{-\infty}^{\infty} dk \tilde{g}(k) e^{ikx_m} \left(e^{iky} - \sum_{j=-N+1}^N e^{ik\Delta x j} \omega_j \right) \quad (4.11)$$

An algorithm that is independent of the function can follow by attempting to force the quantity in parentheses to be zero. For finite N , this cannot be forced always to be zero, and so that deviation is a measure of the error of the algorithm.

Because the interpolation uses only a finite number of points, the best we can hope for is to Taylor expand the quantity in parentheses and zero out the lowest order terms. This generates a set of $2N$ equations

$$\left(\frac{y}{\Delta x}\right)^m = \sum_{j=-N+1}^N j^m \omega_j, \quad m = 0, 1, \dots, 2N-1 \quad (4.12)$$

This is $2N$ equations for $2N$ unknowns ω_j , and so is a solveable linear system. Note that the first equation, for $m = 0$ is that the weights are normalized:

$$1 = \sum_{j=-N+1}^N \omega_j \quad (4.13)$$

From which follows the solution for ω_0 :

$$\omega_0 = 1 - \sum_{j=-N+1; j \neq 0}^N \omega_j \quad (4.14)$$

The solution of this linear system is that the weights are polynomials in $y/\Delta x$:

$$\omega_j = \sum_{r=1}^{2N-1} q_r^j \left(\frac{y}{\Delta x}\right)^r, \quad j = -N+1, -N+1, \dots, -1, 1, 2, \dots, N \quad (4.15)$$

and the coefficients q_r^j can be arranged as a $2N-1 \times 2N-1$ matrix that satisfies

$$\delta_{mr} = \sum_{j=-N+1; j \neq 0}^N j^m q_r^j, \quad m, r = 1, 2, \dots, 2N-1 \quad (4.16)$$

and δ_{mr} is the Kronecker delta. This set of equations can be solved linearly as well, and the results are shown in tables 4.2.

An additional property satisfied by the coefficients comes from the special case when $y = \Delta x$. In this situation, $\omega_1 = 1$ and all weights must be zero. This provides the ‘‘sanity check’’ criterion:

$$\delta_{j1} = \sum_{r=1}^{2N-1} q_r^j \quad (4.17)$$

Three Dimensions

The three dimensional version comes from applying the one dimensional interpolation three times.

$$g(\mathbf{x}) \approx \sum_{l=-N+1}^N \sum_{m=-N+1}^N \sum_{n=-N+1}^N g(\mathbf{x}_{i+l \ j+m \ k+n}) \omega_l \omega_m \omega_n \quad (4.18)$$

Table 4.2: Coefficients q_r^j for higher order interpolation.

$j \backslash r$	1
1	1.0

$$N = 1$$

$j \backslash r$	1	2	3
-1	-1/3	1/2	-1/6
1	1	1/2	-1/2
2	-1/6	0.0	1/6

$$N = 2$$

$j \backslash r$	1	2	3	4	5
-2	1/20	-1/24	-1/24	1/24	-1/120
-1	-1/2	2/3	-1/24	-1/6	1/24
1	1	2/3	-7/12	-1/6	1/12
2	-1/4	-1/24	7/24	1/24	-1/24
3	1/30	0	-1/24	0	1/120

$$N = 3$$

4.8 Finite Difference Gradient

When using fields that are based on gridded data, the issue arises of how to construct the gradient of the field. A similar issue arises for some fields that do not have a well-defined analytic expression for a gradient, for example the CSG operations. A natural approach is to use a finite difference expression to approximate the gradient. In the simplest case, the gradient could be expressed as

$$\nabla f(\mathbf{x}) \approx \left(\frac{f(\mathbf{x} + \hat{\mathbf{x}}\Delta x) - f(\mathbf{x} - \hat{\mathbf{x}}\Delta x)}{2\Delta x}, \frac{f(\mathbf{x} + \hat{\mathbf{y}}\Delta y) - f(\mathbf{x} - \hat{\mathbf{y}}\Delta y)}{2\Delta y}, \frac{f(\mathbf{x} + \hat{\mathbf{z}}\Delta z) - f(\mathbf{x} - \hat{\mathbf{z}}\Delta z)}{2\Delta z} \right) \quad (4.19)$$

The accuracy of this approximation can be seen by Taylor expanding the field f for a few terms, to find that there is an error proportional to powers of Δx , Δy , Δz .

The finite difference expression can be extended to higher order as desired to give higher quality. Illustrating with a 1-D function $f(x)$, the Taylor expansion

$$f(x + y) = \sum_{k=0}^{\infty} \frac{y^k}{k!} f^{(k)}(x) \quad (4.20)$$

can be used to assemble any order finite-difference approximation. Here $f^{(k)}$ is the k -th derivative of the function. A derivative can be represented as a finite sum of $2N$ terms:

$$\frac{d}{dx}f(x) \approx \sum_{n=-N}^N a_n f(x + n\Delta x) \quad (4.21)$$

with suitably chosen coefficients a_n . The approach to fixing their values comes from applying the Taylor expansion to this approximation to get

$$\sum_{k=0}^{\infty} \frac{\Delta x^k}{k!} f^{(k)}(x) \sum_{n=-N}^N n^k a_n \quad (4.22)$$

If this is intended to represent the first derivative, the coefficients must be fixed by the series of equations

$$\sum_{n=-N}^N a_n = 0 \quad (4.23)$$

$$\sum_{n=-N}^N n a_n = \frac{1}{\Delta x} \quad (4.24)$$

$$\sum_{n=-N}^N n^k a_n = 0 \quad k > 1 \quad (4.25)$$

Equation 4.23 and all equations 4.25 for even k are satisfied if the coefficients are antisymmetric, i.e. $a_{-n} = -a_n$. This fixed $a_0 = 0$. Further, it makes sense to scale the coefficients by $a_n = \alpha_n/\Delta x$. Once these choices are made, the remaining equations are

$$\sum_{n=1}^N n \alpha_n = \frac{1}{2} \quad (4.26)$$

$$\sum_{n=1}^N n^{2k+1} \alpha_n = 0 \quad k > 1 \quad (4.27)$$

Table 4.3: Coefficients for finite difference derivatives up to $N = 8$.

	$N = 1$	$N = 2$	$N = 3$	$N = 4$
α_1	0.5	0.57142857142857151	0.60810810810810767	0.63039399624765435
α_2		-0.071428571428571438	-0.1188063063063063	-0.15064623723160311
α_3			0.010698198198198195	0.02101573900354391
α_4				-0.00076349801959558095

	$N = 5$	$N = 6$	$N = 7$	$N = 8$
α_1	0.64535955746772966	0.65609985088633171	0.66418180822321082	0.67048320765246205
α_2	-0.17321459058935865	-0.18996590374149147	-0.20286437997462345	-0.213090576806653
α_3	0.029559879293859895	0.036481469793162628	0.042119156095069443	0.046767457952830598
α_4	-0.0017378184115276803	-0.0026999821278442119	-0.0035797086604024858	-0.004362453117772256
α_5	3.2972239295226059e-05	8.5528288191602389e-05	0.00014591197349313843	0.00020752664200811052
α_6		-9.6309840124388086e-07	-2.8080248121598451e-06	-5.2279046772863686e-06
α_7			2.0367364328490342e-08	6.598736368669243e-08
α_8				-3.26811256996574e-10

The coefficients for choices of N up to 8 are shown in table 4.3. Extending these results to three dimensions, the gradient looks like

$$\nabla f(\mathbf{x}) = \sum_{n=-N}^N \alpha_n \left(\frac{f(\mathbf{x} + n\Delta x \hat{\mathbf{x}})}{\Delta x}, \frac{f(\mathbf{x} + n\Delta y \hat{\mathbf{y}})}{\Delta y}, \frac{f(\mathbf{x} + n\Delta z \hat{\mathbf{z}})}{\Delta z} \right) \quad (4.28)$$

SIGNED DISTANCE FUNCTIONS AND LEVEL SETS

5.1 Signed Distance Function

A particular class of implicit function that has very useful properties is the *Signed Distance Function* (SDF). The SDF is an implicit function with two important properties:

1. The magnitude of the value of the SDF at any point in space is equal to the distance of that point from the closest point on the implicit surface.
2. If a point in space is outside of the implicit surface, the value of the SDF is negative. If a point is inside the implicit surface, the value of the SDF is positive.

The second property concerning the sign is a convention. In some applications the opposite sense of sign is used, i.e. positive outside and negative inside.

The fact that the magnitude of the SDF is a distance separates SDFs from scalar fields. Scalar fields and SDFs have the same transformation properties when it comes to rotations and translations, but their scale transformations differ as shown in table 1.1.

Signed distance functions have a very useful and special property: the gradient of a SDF is a unit vector. Like any other implicit function, the gradient is normal to the surface, but only distance functions are a unit vector. In the case of the sphere the property is easy to see. The SDF for a sphere is

$$\mathbf{f}(\mathbf{x}) = R - |\mathbf{x} - \mathbf{x}_0| \tag{5.1}$$

so the gradient is

$$\nabla \mathbf{f}(\mathbf{x}) = -\frac{\mathbf{x} - \mathbf{x}_0}{|\mathbf{x} - \mathbf{x}_0|} \tag{5.2}$$

Manifestly, $|\nabla \mathbf{f}| = 1$. For a simple, but nonrigorous, proof that this is generally true, we can construct any SDF as

$$\mathbf{f}(\mathbf{x}) = \beta |\mathbf{x} - \mathbf{x}^*(\mathbf{x})| \tag{5.3}$$

where $\mathbf{x}^*(\mathbf{x})$ is the location on the surface closest to the point \mathbf{x} , and β is +1 or -1 depending on whether \mathbf{x} is in or outside of the surface respectively. Taking the gradient of this

$$\nabla f(\mathbf{x}) = \beta \frac{\mathbf{x} - \mathbf{x}^*(\mathbf{x})}{|\mathbf{x} - \mathbf{x}^*(\mathbf{x})|} - \beta \nabla \mathbf{x}^*(\mathbf{x}) \cdot \frac{\mathbf{x} - \mathbf{x}^*(\mathbf{x})}{|\mathbf{x} - \mathbf{x}^*(\mathbf{x})|} \quad (5.4)$$

Two important properties are important to know here. First, because $\mathbf{x}^*(\mathbf{x})$ is the closest point on the surface to the point \mathbf{x} , the difference $\mathbf{x} - \mathbf{x}^*(\mathbf{x})$ is perpendicular to the surface. Second, the gradient $\nabla \mathbf{x}^*(\mathbf{x})$ is always tangent to the surface. Consequently, the second term, containing the inner product of $\nabla \mathbf{x}^*$ and $\mathbf{x} - \mathbf{x}^*$ is zero. It then follows that the gradient of any SDF is a unit vector.

The fact that the SDF gradient is a unit vector gives rise to a very useful quantity. The vector field $\mathbf{x}^*(\mathbf{x})$ defined as

$$\mathbf{x}^*(\mathbf{x}) = \mathbf{x} - f(\mathbf{x}) \nabla f(\mathbf{x}) \quad (5.5)$$

is called the *Closest Point Transform*, or CPT, and is the collection of points located on the surface of the SDF. This non-invertible map allows many algorithms to be sped up because it quickly locates the surface point for any spatial point. In some applications in which the implicit surface is not a SDF, it can be computationally efficient to convert the implicit function to a SDF in order to take advantage of the CPT. More on this in chapter 10.

5.2 Level Set

The special case of a signed distance function that has values on a 3D grid is called a *level set*. One way of creating a level set is to stamp a SDF into a grid. However, there are many scientific and graphics applications of level sets, and the most common way of creating them is via processing algorithms in the particular applications. For example, a closed model consisting of polygons can be converted into a level set via the Fast Marching Method. In reverse, a level set can be converted into a polygon model of surface via the Marching Cubes algorithm. Effectively, level sets and polygonal models are conceptually equivalent representations of geometry.

The amount of spatial detail that can be encoded in a level set is directly related to the cell resolution of the grid. Because the surface is at locations where the grid data interpolates to zero, at least two cells are needed to locate the surface position. This means that no surface features smaller than the size of one cell can be resolved by a level set.

Nominally, the level set representation of a large surface with many fine features would require a grid with many grid points. However many applications of level sets use the data in a way that reduces the data load and makes the memory demands quite manageable. The key observation is this: when the primary purpose of the level set is to identify the surface and surface properties immediately in the neighborhood of the surface, useful signed distance data is needed only in a “narrow” band of grid points about the surface. Grid points far away from the surface have little or no useful purpose, and so need not contain data. This allows the construction of data structures that store data only in bands of grid points, the size of the band being adjustable for the application at hand; at grid points outside of these bands, the points need only be identified as being inside or outside of the surface. The memory savings obtain from these data structures can be very large, and speed enhancements can also be quite large because processing algorithms can be directed to focus only in regions of useful information. The first such data structure to be defined was DT Grids [21], and the current favorite is the open source project OpenVDB [11].

Figure 5.1: Demonstration of the conversion steps polygon \rightarrow level set \rightarrow polygon, for well-matched surface and grid. (a) Original polygonal surface; (b) Level set in OpenVDB format, showing the active voxels; (c) Polygon surface generated from level set. Note that (a) and (c) are identical surfaces.

Figure 5.2: Demonstration of the conversion steps polygon \rightarrow level set \rightarrow polygon, for a “low-resolution” grid. (a) Original polygonal surface; (b) Level set in OpenVDB format, showing the active voxels; (c) Polygon surface generated from level set. The surface in (c) has lost details and features contained in the original surface.

5.3 Geometry vs Level Set

A polygonal closed surface is equivalent to a level set, as long as the size of the polygons and cell size of the level set are comparable. For example, a “well-constructed” surface can be converted into a level set with a cell resolution somewhat smaller than the smallest span of any of the polygons, using the Fast Marching Method. The level set can be converted back into polygons using Marching Cubes. If the cell resolution and the polygon sizes have been paired well, the output surface can many times be identical to the original, vertex for vertex, edge for edge, polygon for polygon. Figure 5.1 demonstrates this behavior.

But what happens when the cell resolution and polygon sizes are not well matched? For example, suppose the resolution of the level set grid is insufficient to resolve at least some of the polygons? Figure 5.2 demonstrates the impact of this situation. Details of the surface have been lost in the conversion to a “low-resolution” level set, so that the conversion back into a surface shows less structure and detail. Such a loss of detail can be of value for some problems. For example, it is part of a process for smoothing surfaces, as discussed in section 7.1. Level set oriented methods of fracturing use the grid resolution as a way of controlling the number and size of fragments in the fractured geometry, as shown in section 7.4. Other techniques that filter the level set can algorithm the surface geometry as well (section 7.3).

The word “noise”, used in the context of volumetric modeling, has several technical meanings. One of them is as the output of pseudo-random number generators that obey various probability density functions. For example, uniformly-distributed noise from the Mersenne Twistor, or gaussian, log-normal, or other distributions.

Another meaning for “noise” is the spatial pattern of values from any of many functions that could be used, sometimes coupled with itself or other functions using scale, translation, rotation, and multiplier transformations. Example of these spatial noise functions are Perlin noise, Worley noise, and others [17].

Here we explicitly present a few noises from pseudo-random number generators, and spatial noise functions. Fractal sums of spatial noise functions serve as a means of creating ever more complex and subtle noise patterns that can have many natural-looking patterns and properties.

6.1 Pseudo-Random Number Generators

Some standard pseudo-random number generators that have been commonly used for many years are `rand()` and `drand48()`. The algorithms behind these generators have significant flaws for volumetric purposes, because both have too-short period lengths and some patterns can be recognized in the sequence of numbers they produce.

A very powerful and widely available pseudo-random number generator is the Mersenne Twister¹[20]. This algorithm has an extremely long period, no obvious patterns in the sequence, and fits very well into the uses we have here.

Pseudo-random numbers are used to drive parameters for volumetric structures. For example, hundreds of spheres could be generated with randomized radius and location. They could also be used to randomize any other parameter that occurs in a volumetric model. In chapter 8 and appendix A, many particle attributes can be usefully randomized, and interesting, controllable spatial patterns created using random walks.

¹Mersenne Twister: http://en.wikipedia.org/wiki/Mersenne_Twister

6.2 Spatial Noise

Perlin Noise

FFT Noise

6.3 Fractal Sums of Spatial Noise

For spatial noise $SN(\mathbf{x})$, more complex patterns coming from linear combinations of multiple octaves of the noise. We can express fractal sums of the noise as

$$FSN(\mathbf{x}) = \frac{1-r}{1-r^N} \sum_{n=0}^N r^n SN((\mathbf{x} - \mathbf{x}_t) f f_j^n) \quad (6.1)$$

where N is the number of “octaves”, r is the roughness, f is the based frequency, f_j is the “fjump” for the successive change of frequency for each octave, and \mathbf{x}_t is translation of the noise. The factor $(1-r)/(1-r^N)$ serves to keep the range of the fractal summed noise the same as the range of the spatial noise.

6.4 Terrain Noise

$$TN(\mathbf{x}) = \begin{cases} A_+ FSN(\mathbf{x})^{\gamma_+} & \text{if } FSN(\mathbf{x}) \geq 0 \\ -A_- |FSN(\mathbf{x})|^{\gamma_-} & \text{if } FSN(\mathbf{x}) < 0 \end{cases} \quad (6.2)$$

6.5 Implicit Function + Noise

6.6 Pyroclastic Sphere

MORE IMPLICIT FUNCTION MANIPULATIONS

- 7.1 Smoothing
- 7.2 Roughening
- 7.3 Distortion
- 7.4 Fracturing Geometry

- 8.1 Particle Attributes
- 8.2 Guide and Child Particles
- 8.3 Random Walks

STAMPING NOISE INTO GRIDS

- 9.1 Noise Clouds
- 9.2 Spline and Surface Noise
- 9.3 Wisps
- 9.4 Spline and Surface Wisps
- 9.5 3D Fractal Flame Wisps

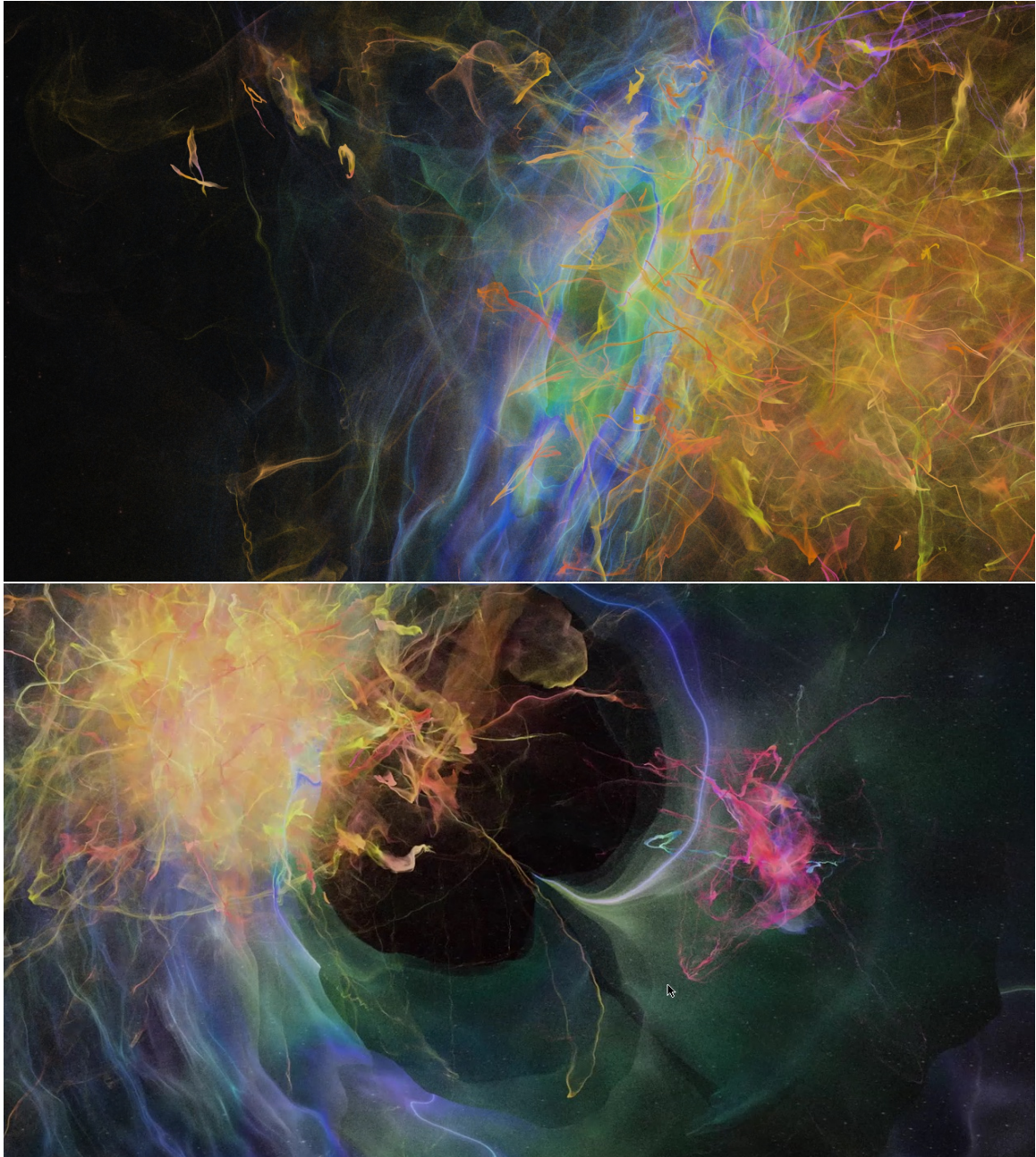


Figure 9.1: Several frames from a short film of evolving 3D fractal flame wisps. Artist: Yujie Shu.

VECTOR FIELDS

Fields fundamentally are prescriptions for doing work. They describe and execute specific algorithms and computations when invoked. Vector-valued fields also provide a collection of methods to alter the work done by other fields, for example by redirecting the location in space where the field is evaluated. In this chapter we explore several mechanisms for vector fields to encode, or warp, information about the structure of the space. We also explore some common ways to generate vector fields from noise as random velocity fields for advection and other applications. In chapters 11 and 12 both are brought to bear to displace and advect fields in a variety of ways.

10.1 Closest Point Transform

The Closest Point Transform (CPT) was introduced in chapter 5. Here we take a deeper look at its properties. Given a signed distance function f , the CPT $\mathbf{X}_{CPT}(\mathbf{x})$ is constructed as

$$\mathbf{X}_{CPT}(\mathbf{x}) = \mathbf{x} - f(\mathbf{x}) \nabla f(\mathbf{x}) \tag{10.1}$$

The CPT is a many-to-one mapping of points in space to the set of points on the surface of the signed distance function.

It can be interesting to think of vector fields that act as CPTs, independent of their definition in equation 10.1. A mathematician might want to think of and categorize the set of all possible CPTs. Of interest here is the question: given one or more CPTs, what operations or transformations can be applied to create other vector fields that are also CPTs? Such a set of operations and/or transformations could be thought of as a set of invariance operations on the set of CPTs, i.e. they map CPTs back to other CPTs.

There are some obvious CPT invariances that we can list here. When an SDF is scaled by α

$$f(\mathbf{x}) \rightarrow \alpha f(\mathbf{x}/\alpha) \tag{10.2}$$

the CPT becomes

$$\mathbf{X}_{CPT}^\alpha(\mathbf{x}) = \mathbf{x} - \alpha f(\mathbf{x}/\alpha) \nabla f(\mathbf{x}/\alpha) \tag{10.3}$$

or

$$\mathbf{X}_{CPT}^\alpha(\mathbf{x}) = \alpha \mathbf{X}_{CPT}(\mathbf{x}/\alpha) \tag{10.4}$$

showing that under scaling operations, the CPT transforms in the same way as the SDF.

An SDF that has been dilated uniformly is still an SDF, although with a different surface. The dilation of the SDF has the form

$$\mathbf{f} \rightarrow \mathbf{f} + c \tag{10.5}$$

for some constant value c . The CPT of the SDF is transformed by

$$\begin{aligned} \mathbf{X}_{CPT}^c(\mathbf{x}) &= \mathbf{X}_{CPT}(\mathbf{x}) - c \nabla \mathbf{f} \\ &= \mathbf{X}_{CPT}(\mathbf{x}) \pm c \frac{\mathbf{X}_{CPT}(\mathbf{x}) - \mathbf{x}}{|\mathbf{X}_{CPT}(\mathbf{x}) - \mathbf{x}|} \end{aligned} \tag{10.6}$$

and the choice of plus or minus sign depends on the sign of the underlying SDF.

Finally, a translation of the SDF by the amount \mathbf{x}_0 changes the CPT to

$$\mathbf{X}_{CPT}^{\mathbf{x}_0}(\mathbf{x}) = \mathbf{X}_{CPT}(\mathbf{x} - \mathbf{x}_0) + \mathbf{x}_0 \tag{10.7}$$

Note that, in these transformations, we are mostly able to express the transformed CPT without explicitly invoking the SDF, i.e. if we know completely the CPT, we can directly generate other CPTs using these example transformations without first creating or evaluating SDFs. It would be an interesting and beneficial exercise to find the set, or space, of all possible transformations of CPTs into other CPTs.

The CPT is a type of vector field called a *conservative vector field*, which is a vector field of the form

$$\mathbf{X}_{CPT}(\mathbf{x}) = \nabla U(\mathbf{x}) \tag{10.8}$$

for some scalar field $U(\mathbf{x})$. This scalar field is called a potential field. In particular, we can construct U for a given SDF \mathbf{f} as

$$U(\mathbf{x}) = \frac{1}{2} (|\mathbf{x}|^2 - \mathbf{f}^2(\mathbf{x})) \tag{10.9}$$

As a conservative vector field, it has no axial component and so no curl:

$$\nabla \times \mathbf{X}_{CPT}(\mathbf{x}) = \nabla \times \nabla U(\mathbf{x}) = 0 \tag{10.10}$$

From its definition in equation 10.1, the CPT also satisfies the differential condition

$$(\mathbf{X}_{CPT}(\mathbf{x}) - \mathbf{x}) \cdot \nabla \mathbf{X}_{CPT}(\mathbf{x}) = 0 \tag{10.11}$$

10.2 Near Point Transform

The CPT is a well defined entity computable from a signed distance function, representing the collection of points on the surface defined by the SDF. Any implicit function has a collection of points on its surface, so it should be possible to create an vector field with similar properties for an implicit function. The CPT is easily constructed from an SDF because the SDF has a value representing distance, and its gradient is a unit vector. For general implicit functions, neither of these properties is available. But we can at least approximate the form of the CPT by using a Taylor expansion approximation. Given an implicit function f , at a position \mathbf{x} not on the surface, we can define the vector \mathbf{y} as the displacement of \mathbf{x} to the surface:

$$f(\mathbf{x} + \mathbf{y}) = 0 \tag{10.12}$$

An approximate value of \mathbf{y} can be found from the first order Taylor expansion of this expression:

$$0 = f(\mathbf{x} + \mathbf{y}) \approx f(\mathbf{x}) + \mathbf{y} \cdot \nabla f(\mathbf{x}) \quad (10.13)$$

If we assume that \mathbf{y} has a form inspired by the CPT formula of $\mathbf{y} = d\nabla f(\mathbf{x})$, then we can solve this linear equation for the quantity d as

$$d = -\frac{f(\mathbf{x})}{|\nabla f(\mathbf{x})|^2} \quad (10.14)$$

The Near Point Transform (NPT) gives the approximate position of the surface point $\mathbf{x} + \mathbf{y}$ as

$$\mathbf{X}_{NPT}^f(\mathbf{x}) = \mathbf{x} - f(\mathbf{x}) \frac{\nabla f(\mathbf{x})}{|\nabla f(\mathbf{x})|^2} \quad (10.15)$$

and in particular, if the implicit function coincidentally happens to actually be a SDF, the formula for the NPT reduces to that of the CPT.

Although the NPT gives only an approximation of a point on the surface, that point should be closer than the point at which the NPT is evaluated. This means we can set up an iterative procedure, expressed in Algorithm 5, to get arbitrarily close to the surface point for the CPT using a suitable threshold value. Additionally, the iteration could be stopped by limiting the number of loops through the while. This iterative algorithm we refer to as the Iterative Near Point Transform (INPT), and denote the mathematical operation as $\mathbf{X}_{INPT}^f(\mathbf{x})$ for the implicit function f .

Algorithm 5 Iterative Computation of the CPT from the NPT.

```

 $\mathbf{x} \leftarrow \mathbf{X}_{NPT}^f(\mathbf{x})$ 
while  $|f(\mathbf{x})| > \text{threshold}$  do
     $\mathbf{x} \leftarrow \mathbf{X}_{NPT}^f(\mathbf{x})$ 
end while

```

Note that with a sufficient number of iterations, the INPT produces a point on the surface of the implicit function, i.e. the INPT converges to a CPT with $\mathbf{X}_{CPT}(\mathbf{x}) = \mathbf{X}_{INPT}^f(\mathbf{x})$. Recall that equation 5.3 expresses signed distance functions in terms of their CPT. This means that for the implicit function f we can construct a corresponding signed distance function g as

$$g(\mathbf{x}) = \text{sign}(f(\mathbf{x})) \left| \mathbf{X}_{INPT}^f(\mathbf{x}) - \mathbf{x} \right| \quad (10.16)$$

This algorithm can be used to “redistance” an implicit function into a SDF.

10.3 Composing Maps

10.4 Texture Coordinate Transfer

10.5 Velocity Fields

Velocity from Spatial Noise

Velocity from FFT Random Numbers

Velocity from Other Fields

10.6 Incompressibility

DISPLACEMENTS

The obvious application of SDFs, both procedural and level sets, is the representation of surface geometry by a volumetric field. Such a representation also presents an opportunity to create new fields and surface geometry using volumetric manipulations. This chapter is devoted to exploring displacements of the SDF. A classic form of displacement is the Pyroclastic displacement, which gets its name from the lumpy structure it generates, reminiscent of volcanic pyroclastic clouds. This is most easily illustrated by the pyroclastic sphere, which has been used extensively in VFX for snow avalanches and clouds. The pyroclastic sphere displacement generalizes to pyroclastic displacement of arbitrary SDFs. An extension of the general pyroclastic displacement is a method called *cumulo* because it accumulates multiple displacements to give a more realistic appearance of a cumulus cloud.

11.1 Pyroclastic Sphere

The pyroclastic sphere originated in Alan Kaplar's VFX work on the film XXX, to create the appearance of a dangerous growing avalanche chasing behind a skier. Its visual structure consists of a spherical volume of density with the surface displaced using a particular choice of noise function, so that the volume resembles a pyroclastic, or cauliflower, puff, as illustrated in figure 11.1. There are two ingredients in this process: (1) a spatial noise field, and (2) a displacement procedure that is driven by the spatial noise.

The mathematics of the sphere is sufficiently simple that we can build it two different ways. The first method constructs the density directly without referencing the concept of a SDF. The second method begins with the SDF and constructs the pyroclastic sphere as an implicit function, from which density can be constructed via the procedure described in section 2.4.

The algorithm is illustrated here for a pyroclastic sphere that has density of ρ_{ps} inside. The algorithm for calculating the density of a pyroclastic sphere at any point \mathbf{x} in space is as follows (see also algorithm 6):

1. Calculate the distance from the point of interest \mathbf{x} to the center of the sphere $\mathbf{x}_{\text{sphere}}$:

$$d = |\mathbf{x} - \mathbf{x}_{\text{sphere}}| \tag{11.1}$$

2. Compare d to the displacement bound d_{bound} of the Perlin noise and the radius R of the sphere. If $d < R$, \mathbf{x} is definitely inside the pyroclastic sphere, and the density is ρ_{ps} . If $d > R + d_{\text{bound}}$, then the point \mathbf{x} is definitely outside of the pyroclastic sphere, density is 0.

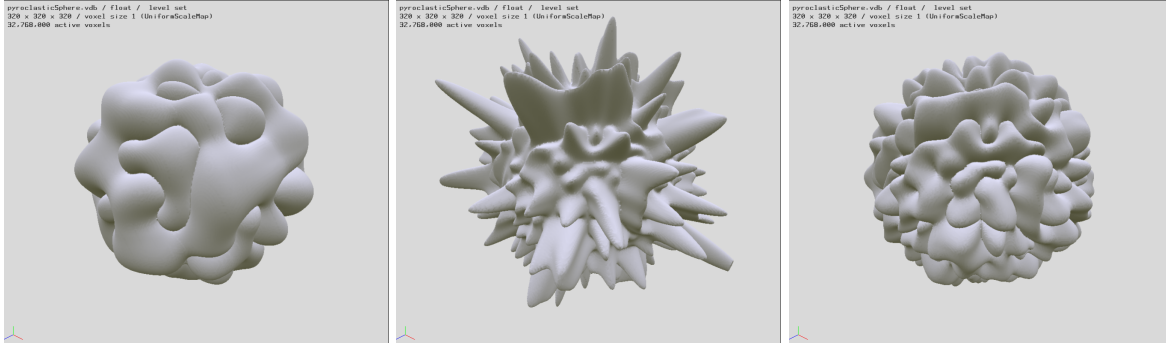


Figure 11.1: Examples of classic pyroclastically displaced implicit spheres.

3. If $0 < d - R < d_{\text{bound}}$, then compute the displacement: Create a point on the unit sphere surface $\mathbf{n} = (\mathbf{x} - \mathbf{x}_{\text{sphere}})/d$. The displacement by the noise is $r = |\mathbf{N}(\mathbf{n})|$. If $d - R < r$, the point \mathbf{x} is inside the pyroclastic sphere and the density is ρ_{ps} . Otherwise, the density is 0.

Algorithm 6 Computing the density inside a pyroclastic sphere.

```

 $d \leftarrow |\mathbf{x} - \mathbf{x}_{\text{sphere}}|$ 
if  $d < R$  then
     $\rho \leftarrow \rho_{pa}$ 
end if
if  $d > R + d_{\text{bound}}$  then
     $\rho \leftarrow 0$ 
end if
if  $0 < d - R < d_{\text{bound}}$  then
     $\mathbf{n} \leftarrow (\mathbf{x} - \mathbf{x}_{\text{sphere}})/d$ 
     $r \leftarrow |\mathbf{N}(\mathbf{n})|$ 
    if  $d - R < r$  then
         $\rho \leftarrow \rho_{ps}$ 
    else
         $\rho \leftarrow 0$ 
    end if
end if

```

The absolute value of the noise is used because it produces sharply cut “canyons” and smoothly rounded “peaks”. Rounded canyons would not have the visual impact as a pyroclastic structure.

The SDF version of this algorithm begins by noting that the SDF for a sphere is

$$\mathbf{f}_{\text{sphere}}(\mathbf{x}) = R - |\mathbf{x} - \mathbf{x}_{\text{sphere}}| \quad (11.2)$$

The comparisons used in each step of the above algorithm were based on examining the value of this quantity.

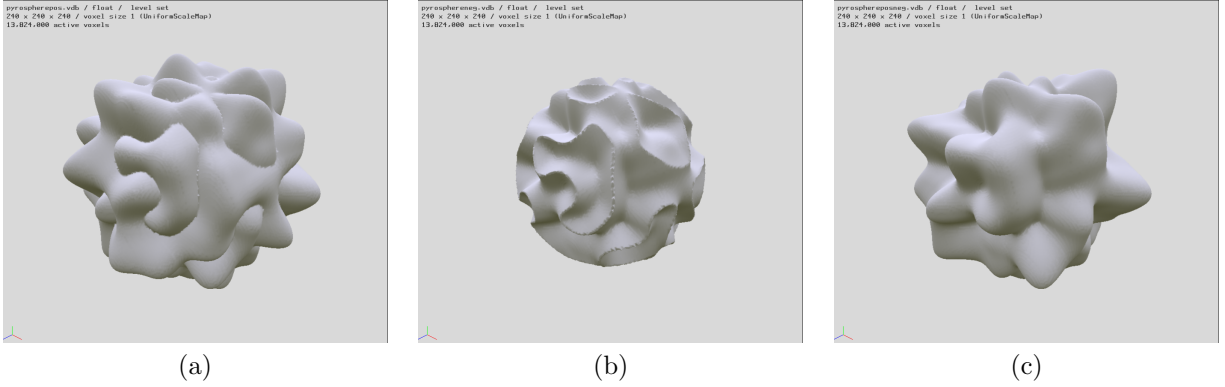


Figure 11.2: Pyroclastic displacement of a sphere using (a) positive displacement; (b) negative displacement; (c) both positive and negative displacement

Adding the pyroclastic noise component amounts to altering the SDF:

$$\mathbf{f}_{ps}(\mathbf{x}) = R - |\mathbf{x} - \mathbf{x}_{\text{sphere}}| + \left| \mathcal{N} \left(R \frac{\mathbf{x} - \mathbf{x}_{\text{sphere}}}{|\mathbf{x} - \mathbf{x}_{\text{sphere}}|} \right) \right| \quad (11.3)$$

and so the density field is

$$\rho(\mathbf{x}) = \rho_{ps} \text{mask}(\mathbf{f}_{ps})(\mathbf{x}) \quad (11.4)$$

There are many variations of this approach that lead to other structures and appearances. For example, switching from the absolute value of the noise to negative of the absolute value would cause the extremes of displacement to be sharp instead of rounded, and the valleys rounded instead of cusped, as illustrated in figure 11.2(b). Alternatively, removing the absolute value would give the bumps and equally rounded appearance on the extremes of displacement, as in 11.2(c). Using a power on the noise

$$\mathbf{f}_{ps}(\mathbf{x}) = R - |\mathbf{x} - \mathbf{x}_{\text{sphere}}| + \left| \mathcal{N} \left(R \frac{\mathbf{x} - \mathbf{x}_{\text{sphere}}}{|\mathbf{x} - \mathbf{x}_{\text{sphere}}|} \right) \right|^\gamma \quad (11.5)$$

gives control on the sharpness or flatness of the displacement extremes (figure 11.3).

For a volume consisting of many pyroclastic spheres, pyroclasts can be augmented to handle overlap. Simply unioning multiple SDFs works well. For algorithm 6, the equivalent merge is to set the density equal to the maximum of the density result for all of the overlapping pyroclasts.

For pyroclastic spheres, there is little compelling preference the SDF versus algorithm 6. However, in more general geometries, the SDF approach continues to be straightforward and efficient regardless of the geometric complexity involved. Generalizing algorithm 6 beyond spheres is problematic.

11.2 Pyroclastic Displacement of Arbitrary Geometry

The generalization of pyroclastic displacement to arbitrary geometry becomes very straightforward by using the level set for the geometry. Suppose a particular closed surface has the level set representation \mathbf{f}_{geom} . We

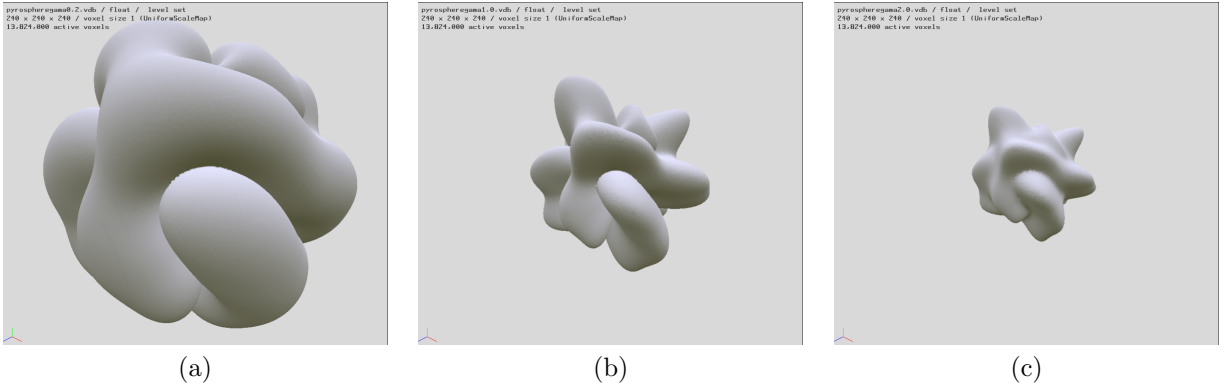


Figure 11.3: Pyroclastic displacement as a function of γ . (a) $\gamma = 0.2$; (b) $\gamma = 1$; (c) $\gamma = 2$.

can generate an implicit surface with pyroclastic displacement this way:

$$\mathbf{f}_{pyro}(\mathbf{x}) = \mathbf{f}_{geom}(\mathbf{x}) + |\mathbf{N}(\mathbf{x}_{geom}^*(\mathbf{x}))|^\gamma \quad (11.6)$$

where \mathbf{x}_{geom}^* is the CPT of the level set of the input geometry.

$$\mathbf{x}_{geom}^*(\mathbf{x}) = \mathbf{x} - \mathbf{f}_{geom}(\mathbf{x}) \nabla \mathbf{f}_{geom}(\mathbf{x}) \quad (11.7)$$

An unfortunate consequence of pyroclastic displacement is that the implicit function produced, i.e. equation 11.6, is not a SDF. The easiest way to see that is by taking a gradient and finding that it is not a unit vector. In general, most of the algorithms that manipulate a SDF produce an implicit function that is not a SDF. Consequently it is very worthwhile to work out how to accommodate non-SDF implicit functions from the very beginning.

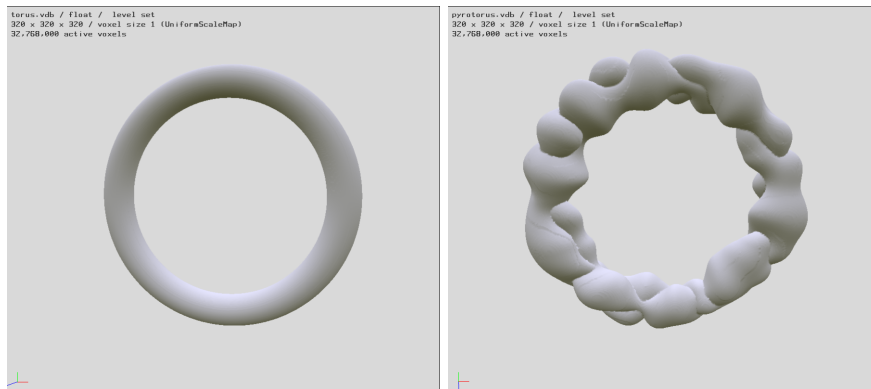
The key issue when pyroclastically displacing a general implicit function is how to locate the surface, given that the CPT is not available for general implicit functions. In this situation, we have the NPT and INPT introduced in section 10.2. Then the generalization of pyroclastic displacements to implicit functions g is

$$g_{pyro}(\mathbf{x}) = g(\mathbf{x}) + |\mathbf{N}(\mathbf{X}_{INPT}^g(\mathbf{x}))|^\gamma \quad (11.8)$$

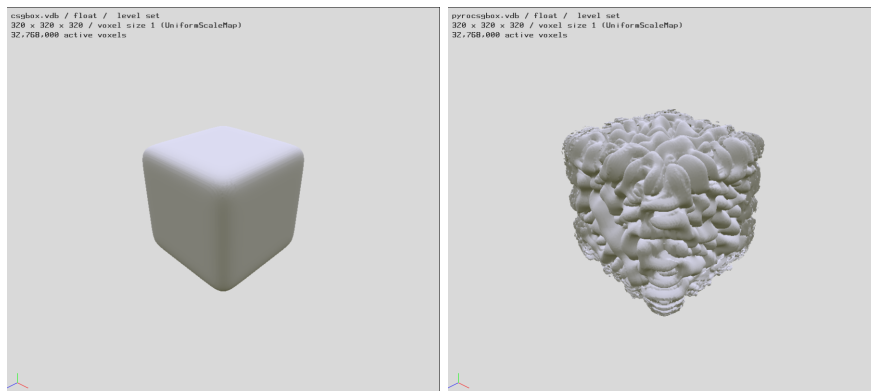
Figure 11.4 shows several examples of implicit surfaces pyroclastically displaced.

11.3 Cumulo

Pyroclastic displacement is very useful in volume modeling because it produces natural-looking surface detail. In some modeling problems, for example modeling clouds such as those depicted in figure 11.5, there are multiple layers of “bumps” that look like pyroclastic displacements layered on top of pyroclastically displaced surfaces. A procedure first described in [23] called *Cumulo* sets up the ability to apply multiple layers of pyroclastic displacement, with control over the characteristics of each layer.



(a)



(b)

Figure 11.4: Implicit surfaces and their pyroclastic displacements. (a) Torus; (b) Box.



Figure 11.5: Cumulus clouds with multiple layers of bumps.

11.4 Displacement via Local Transformations

An alternate approach to construction implicit functions that appear to be pyroclastically displaced to is actually displace the input implicit function, rather than altering the implicit function formula. Given an implicit function $f(\mathbf{x})$, the quantity $f(\mathbf{X}(\mathbf{x}))$ is also an implicit function, where $\mathbf{X}(\mathbf{x})$ is a vector field that remaps or displaces points in space. The art of choosing the pyroclastic structure comes down to the choice of the mapping function. We will refer to the mapping function $\mathbf{X}(\mathbf{x})$ as the *Pyroclastic Point Transform* (PPT) because of its origins and motivation, even though in many cases a pyroclastic appearance is not the goal.

An convenient advantage of using a PPT for generating pyroclastic displacement is that it does not alter the numerical range of values of the implicit function. Some algorithms for manipulating implicit functions anticipate that it will have values within a known range, particularly if the algorithm expects the implicit function to be an SDF or level set. Since the pyroclastic displacement approaches in the previous sections of this chapter alter the values of the implicit function, subsequent use of these other algorithms would be problematic.

Noisy Local Transformations

A first demonstration of using a PPT uses a sphere and Perlin noise, similar to section 11.1. As in that situation, we want displacements that are perpendicular to the surface, so the PPT has the form

$$\mathbf{X}(\mathbf{x}) = \mathbf{x} + \left| N \left(\frac{\mathbf{x} - \mathbf{x}_{\text{sphere}}}{|\mathbf{x} - \mathbf{x}_{\text{sphere}}|} \right) \right|^\gamma \left(\frac{\mathbf{x} - \mathbf{x}_{\text{sphere}}}{|\mathbf{x} - \mathbf{x}_{\text{sphere}}|} \right) \quad (11.9)$$

This PPT produces a pyroclastic sphere like that shown in figure 11.6. In more general situations with an arbitrary SDF f , a better choice of pyroclastic PPT would use the CPT and the fact that the gradient of the SDF is a unit vector:

$$\mathbf{X}(\mathbf{x}) = \mathbf{x} + |N(\mathbf{x}^*(\mathbf{x}))|^\gamma \nabla f(\mathbf{x}) \quad (11.10)$$

When the implicit function is not an SDF, this recipe has to be generalized to use the INPT instead of the CPT, and the gradient has to be explicitly normalized:

$$\mathbf{X}(\mathbf{x}) = \mathbf{x} + \left| N \left(\mathbf{X}_{INPT}^f(\mathbf{x}) \right) \right|^\gamma \frac{\nabla f(\mathbf{x})}{|\nabla f(\mathbf{x})|} \quad (11.11)$$

For the torus, this PPT produces the result in figure 11.7.

Distance-Preserving Transforms: Meepzoids

There is a particular situation that is of interest with the implicit function is a signed-distance function. Given the SDF $f(\mathbf{x})$, the displaced field $f(\mathbf{X}(\mathbf{x}))$ is *not* an SDF in general, because the gradient of the displaced field is no longer a unit vector. The displaced field is still a valid implicit function, but it has lost the attribute of having values that correspond to the distance from the point in space to the surface.

We can ask however, whether there are conditions under which the displaced SDF remains an SDF. If it is possible to build and control PPTs that preserve, it would eliminate the need to convert the pyroclastic field back into an SDF. The fundamental definition of the SDF is that its gradient is a unit vector. Taking the gradient of the displaced SDF gives

$$\nabla f(\mathbf{X}(\mathbf{x})) = (\nabla \mathbf{X}(\mathbf{x}))^T \cdot \nabla f(\mathbf{y})|_{\mathbf{y}=\mathbf{X}(\mathbf{x})} \quad (11.12)$$

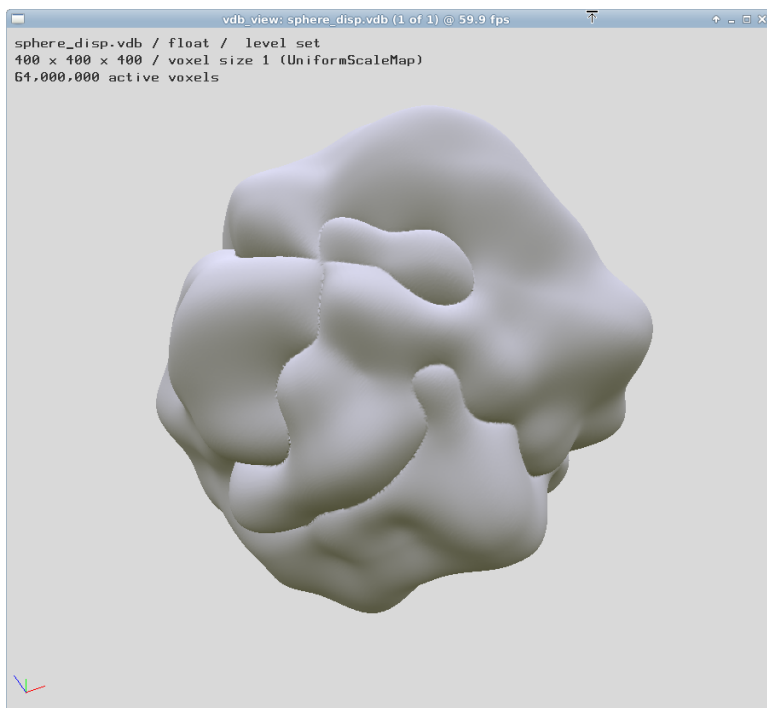


Figure 11.6: Pyroclastic sphere generated from a PPT with Perlin noise normal displacement.

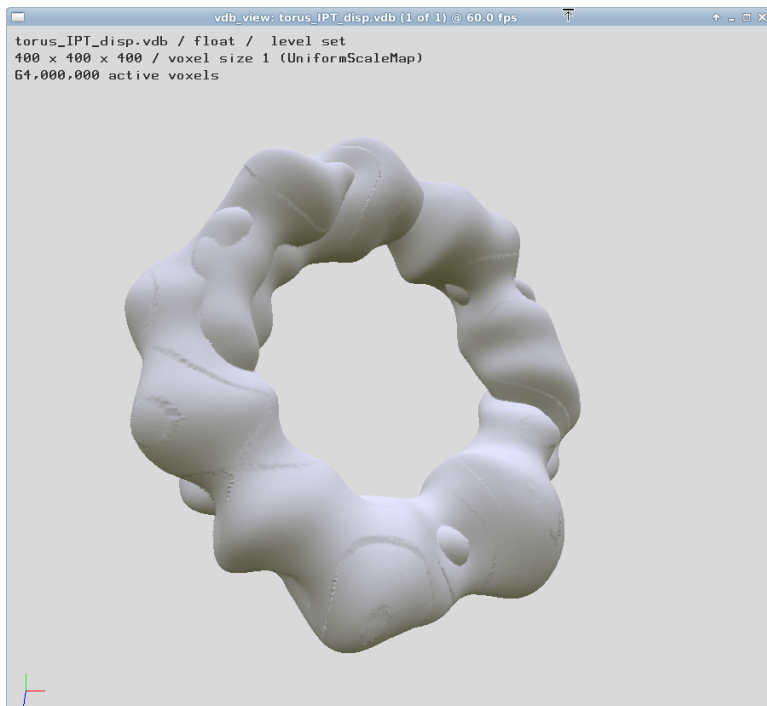


Figure 11.7: Pyroclastic torus generated from a PPT with Perlin noise normal displacement.

Requiring that the gradient of the displaced SDF be a unit vector imposes the requirement

$$1 = \nabla f(\mathbf{X}(\mathbf{x})) \cdot \nabla f(\mathbf{X}(\mathbf{x})) = \nabla f(\mathbf{y}) \cdot (\nabla \mathbf{X}(\mathbf{x})) \cdot (\nabla \mathbf{X}(\mathbf{x}))^T \cdot \nabla f(\mathbf{y})|_{\mathbf{y}=\mathbf{X}(\mathbf{x})} \quad (11.13)$$

This requirement can be satisfied whenever the PPT has the property

$$(\nabla \mathbf{X}(\mathbf{x})) \cdot (\nabla \mathbf{X}(\mathbf{x}))^T = 1 \quad (11.14)$$

which means that the gradient of the PPT is a matrix field that is orthogonal at every point in space. The continuous set of all such matrices is a Lie Group called $SO(3)$. In this situation, the gradient of the PPT is a local rotation, and has the form

$$\nabla \mathbf{X}(\mathbf{x}) = \exp\left(\sum_{k=1}^3 \tau_k \omega_k(\mathbf{x})\right) \quad (11.15)$$

using the matrices τ_k introduction in section 1.4, and any three scalar fields $\omega_k(\mathbf{x})$.

A viable strategy for creating distance-preserving PPTs is this:

1. Construct $\omega_1(\mathbf{x})$, $\omega_2(\mathbf{x})$, $\omega_3(\mathbf{x})$ from some artistic choices.
2. Integrate equation 11.15 to obtain the PPT.
3. Apply the PPT as needed.

Step 2 is the subject of the next subsection. A PPT that preserves distances will be designated a *Meepzoid*.

Solving for a Meepzoid

This subsection presents an integration algorithm for solving equation 11.15 for the meepzoid $\mathbf{X}(\mathbf{x})$. The approach solves for the vector field on a rectangular grid, and is similar to the Gauss-Seidel solution for Poisson's equation. It is based on an expression for the gradient in terms of a finite-difference representation, combined with an iterative refinement of a solution of the finite-difference. The Gauss-Seidel solution for Poisson's equation is based on the simplest finite-difference expression for a Laplacian. For this problem, we can solve for the meepzoid in the context of the more general expression for a finite difference gradient presented in section 4.8. For an N term finite difference, the gradient is

$$\mathbf{M}(\mathbf{x}) \equiv \nabla \mathbf{X}(\mathbf{x}) = \sum_{n=-N}^N \alpha_n \left(\frac{\mathbf{X}(\mathbf{x} + n\Delta\mathbf{x})}{\Delta x}, \frac{\mathbf{X}(\mathbf{x} + n\Delta\mathbf{y})}{\Delta y}, \frac{\mathbf{X}(\mathbf{x} + n\Delta\mathbf{z})}{\Delta z} \right) \quad (11.16)$$

where the coefficients are anti-symmetric, i.e. $\alpha_{-n} = -\alpha_n$, and so $\alpha_0 = 0$. Throughout this discussion we assume we have chosen the point \mathbf{x} to be one of the grid points for our rectangular grid, so that all values of \mathbf{X} are needed only at grid points.

Taking the inner product of this expression with $\Delta\mathbf{x}$, and isolating just one term $n = j$ (we will generalize shortly), we get

$$\alpha_j \mathbf{X}(\mathbf{x} + j\Delta\mathbf{x}) = \Delta\mathbf{x} \cdot \mathbf{M}(\mathbf{x}) - \sum_{\substack{n=-N \\ n \neq j}}^N \alpha_n \mathbf{X}(\mathbf{x} + n\Delta\mathbf{x}) \quad (11.17)$$

Since \mathbf{x} is an arbitrary point on the grid, we can shift the point by $-j\Delta\mathbf{x}$ to get

$$\mathbf{X}(\mathbf{x}) = \frac{1}{\alpha_j} \Delta\mathbf{x} \cdot \mathbf{M}(\mathbf{x} - j\Delta\mathbf{x}) - \frac{1}{\alpha_j} \sum_{\substack{n=-N \\ n \neq j}}^N \alpha_n \mathbf{X}(\mathbf{x} + (n-j)\Delta\mathbf{x}) \quad (11.18)$$

Two more equations like this one follow from taking the inner products with $\Delta\mathbf{y}$ and $\Delta\mathbf{z}$. We can also construct $2J$ equations for each of these by varying j between $-J$ and J for some integer $J \leq N$. Averaging all of these,

$$\begin{aligned} \mathbf{X}(\mathbf{x}) &= \mathbf{X}_t \\ &+ \frac{1}{6J} \sum_{\substack{j=-J \\ j \neq 0}}^J \frac{1}{\alpha_j} (\Delta\mathbf{x} \cdot \mathbf{M}(\mathbf{x} - j\Delta\mathbf{x}) + \Delta\mathbf{y} \cdot \mathbf{M}(\mathbf{x} - j\Delta\mathbf{y}) + \Delta\mathbf{z} \cdot \mathbf{M}(\mathbf{x} - j\Delta\mathbf{z})) \\ &- \frac{1}{6J} \sum_{\substack{j=-J \\ j \neq 0}}^J \frac{1}{\alpha_j} \sum_{\substack{n=-N \\ n \neq j}}^N \alpha_n (\mathbf{X}(\mathbf{x} + (n-j)\Delta\mathbf{x}) + \mathbf{X}(\mathbf{x} + (n-j)\Delta\mathbf{y}) + \mathbf{X}(\mathbf{x} + (n-j)\Delta\mathbf{z})) \end{aligned} \quad (11.19)$$

where \mathbf{X}_t is an overall translate amount that occurs as an integration constant.

This form sets up an iterative solution process, again like Gauss-Seidel. The initial solution can be $\mathbf{X}^0(\mathbf{x}) = \mathbf{x}$, and the expression for \mathbf{X}^{k+1} in terms of \mathbf{X}^k is

$$\begin{aligned} \mathbf{X}^{k+1}(\mathbf{x}) &= \mathbf{X}_t \\ &+ \frac{1}{6J} \sum_{\substack{j=-J \\ j \neq 0}}^J \frac{1}{\alpha_j} (\Delta\mathbf{x} \cdot \mathbf{M}(\mathbf{x} - j\Delta\mathbf{x}) + \Delta\mathbf{y} \cdot \mathbf{M}(\mathbf{x} - j\Delta\mathbf{y}) + \Delta\mathbf{z} \cdot \mathbf{M}(\mathbf{x} - j\Delta\mathbf{z})) \\ &- \frac{1}{6J} \sum_{\substack{j=-J \\ j \neq 0}}^J \frac{1}{\alpha_j} \sum_{\substack{n=-N \\ n \neq j}}^N \alpha_n (\mathbf{X}^k(\mathbf{x} + (n-j)\Delta\mathbf{x}) + \mathbf{X}^k(\mathbf{x} + (n-j)\Delta\mathbf{y}) + \mathbf{X}^k(\mathbf{x} + (n-j)\Delta\mathbf{z})) \end{aligned} \quad (11.20)$$

This algorithm has unknown convergence speed and properties.

Noisy Meepzoids

ADVECTION

12.1 Semi-Lagrangian

12.2 BFECC

12.3 Modified-MacCormack

12.4 Characteristic Maps

Gradient Stretch

Log Advection

12.5 Vaporous Emission Using Advection

CHAPTER
THIRTEEN

PUFFY CLOUDS

14.1 Navier-Stokes Equations and Solver Splitting

14.2 Advection

14.3 Incompressibility

Poisson Equation

Solution Methods

14.4 Boundary Conditions and Objects in Flows

14.5 Reflection Solver

14.6 Examples

Mushroom Cloud

Cold Flow Onto Floor

Hot and Cold Mixing

14.7 Gridless Advection

14.8 Semi-Lagrangian Mapping

14.9 Rendering Extreme Detail

14.10 Compressible Gases

14.11 Bernoulli Waves on Arbitrary Surfaces

VOLUMETRIC COMPOSITING

15.1 Value Compositing

15.2 Map Compositing

PARTICLE EDITING

A.1 Notes on Editing Particle Databases to Create Visual Detail

These notes were created around 1999. They describe how to generate and manipulate large amounts of particles procedurally, then render them one-by-one in a custom statistically-based particle renderer.

TRICKS FOR PDB PARTICLE SHADING AND EMISSION

Contents

- Intro to Pdb Editing
- The `emit()` function
- Creating child particles
- Child particle algorithms
 - Uniform space filling
 - Cauliflowers
 - Curves in space
 - Basic Random Walk
 - Correlated Random Walk
 - Spherical vs Cartesian walk
 - Levy Flight
 - Directed walk
 - Filling inside implicit surfaces
 - Time-dependence (pulses, shocks, and waves)

Intro to Pdb Editing

The PdbEditor executes a pdb shader once for each particle contained in the pdb file(s) specified. Before executing the shader, the particle attributes have been updated to the values for the current pdb particle. Those values are accessed via the `getAtt` function, and can be modified via the `setAtt` function. For example

```
float radius;  
getAtt("radiusPP", &radius);
```

retrieves the value of the particle radius, and places it into the variable `radius`. The names of the attributes, such as "radiusPP" in this example, are the ones contained in the Pdb file, and are generally based on the same names given by Maya or any other pdb generating package. Other attribute names commonly used are in the table below:

Attribute	Data Format	Description
radiusPP	float	radius
rgbPP	vector	color
position	vector	3D position of particle center
velocity	vector	3D velocity of particle
age	float	age of particle
id	int	particle id number

Probably the two most important attributes are the id and position. Because of that, the example editor shading routines below generally begin with the code segment

```
int id;
getAtt("id", &id);
vector position;
getAtt("position", &position);
```

The shading technology behind the pdb editor allows for a wide flexibility to manipulate particles, perform mathematical operations, compute vector mathematics, etc. The details can be found in its documentation. To some extent, many of the details can be learned just from following the examples below.

The pdb shading file contains all of the examples below.

The emit () function

The shader function emit() performs application specific work within the editor. In the partman application, the emit() call sends the particle attributes to the renderer, which renders a sphere created according to those attributes. In other applications, the emit() call performs other functions.

A simple example editor shader using the emit() call is:

```
int main()
{
int id;
getAtt("id", &id);
vector position;
getAtt("position", &position);

/* Change the position just for fun */
vector displacement;
displacement[0] = 2.3;
displacement[1] = 0.6777;
displacement[2] = -45;

position = position + displacement;
setAtt("position", position);

/* Now emit the particle */
emit();

return 7;
}
```

This editor shader will simply shift every particle in the pdb file over by the vector amount (2.3, 0.6777, -45). As each one is shifted, emit() is called to dispose of it. The reason why emit() is so important is because it may be called any number of times for each pdb "guide" particle. This allows us to turn one pdb particle into many "child" particles. As each child particle is created, the attributes are changed to suit the need, and emit() is called to dispose of it.

Creating child particles

Here is a simple example of how to create child particles and emit them:

```
int main()
{
  /* Retrieve guide particle info */
  int id;
  getAtt("id", &id);
  srand48(id);

  vector position;
  getAtt("position", &position);
  float radius;
  getAtt("radiusPP", &radius);

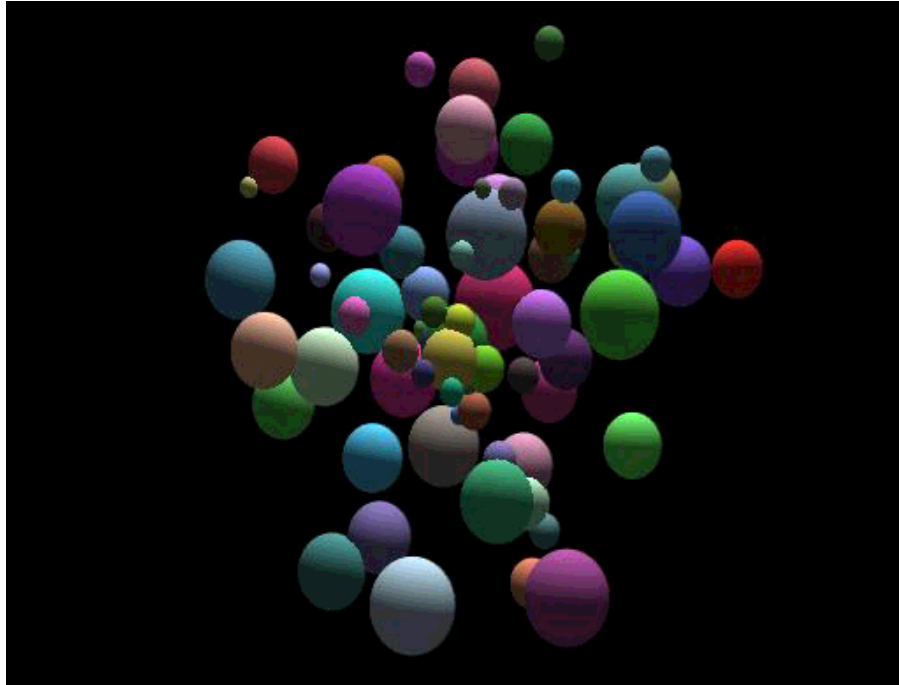
  /* Determine how many child particles are desired */
  float childmax;
  childmax = 100;

  /* Reduce size of child particles
   so that they fit inside guide particle */
  float childradius;
  childradius = 0.3 * radius / pow(childmax, 0.3333);
  setAtt("radiusPP", childradius);

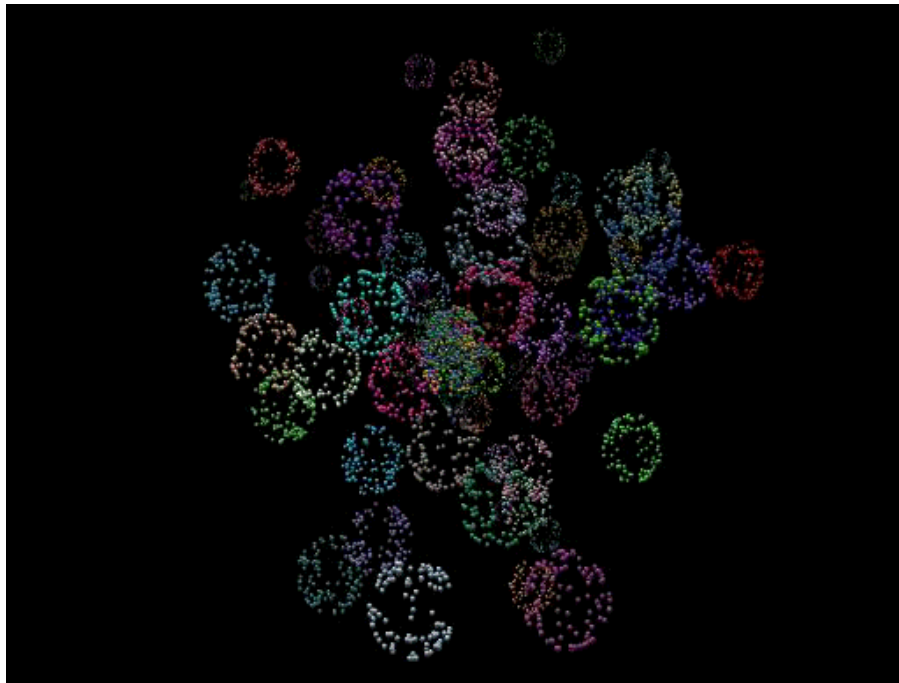
  float child;
  child = 0;

  /* Loop over all children, creating new positions */
  vector d;
  while( child < childmax )
  {
    d[0] = drand48() - 0.5;
    d[1] = drand48() - 0.5;
    d[2] = drand48() - 0.5;
    d = radius * d / sqrt( d . d );
    d = d + position;
    setAtt("position", d);
    emit();
    child = child + 1;
  }
  return 7;
}
```

The figures below show the effect of the child creation process using this code sample.



Guide particles



Child particles

In fact, the generic layout of the child creation editor can be outlined as follows:

```

/* Define global data structures for guide and child particles */

/* guide particle data */
int      guideid;
float    guideradius;
vector   guideposition;
vector   guidecolor;
vector   guidevelocity;
float    guideage;

/* child particle data */
float    childcounter;
float    maxchildcount;
float    childradius;
vector   childposition;
vector   childcolor;

/* others... */
/* .... */

/* Define functions to do the work */

void getGuideParticleInfo()
{
getAtt("id",      &guideid);
getAtt("radiusPP", &guideradius);
getAtt("position", &guideposition);
getAtt("rgbPP",   &guidecolor);
getAtt("velocity", &guidevelocity);
getAtt("age",     &guideage);
return;
}

void setChildParticleInfo()
{
/* dependent on application */
/* set child attributes that are fixed */
return;
}

void initChildParticle  ()
{
childcounter = 0;

/* if using random numbers, initialize seed
to track with the guide particle */
srand48(guideid);

/* give initial values of changing attributes */
return;
}

int moreChildren      ()
{
int flag;
flag = 0;
childcounter = childcounter + 1;
if(childcounter <= maxchildcount) {flag = 1;}
return flag;
}

```



```

}

int setChildParticle    ()
{
/* Do something to distinguish this child
from the many others being created.    */
return 1; /* return 1 to emit, 0 to not emit */
}

/* Operate main() in a fairly generic way */
int main()
{
/* Retrieve guide particle info of use */
getGuideParticleInfo();

/* Setup and Initialize child parameters */
setChildParticleInfo();
initChildParticle();

/* Loop over all children */
int emitdecision;
while( moreChildren() )
{
emitdecision = setChildParticle();
if(emitdecision==1){emit();}
}
return 7;
}

```

With this set-up, a variety of algorithms can be brought to bear to spread child particles around the volume of space based on pseudo-dynamical forms, guide particle data, statistics, and any other paradigm. The next section is devoted to providing several methods of filling the setChildParticle() function to achieve various ends. Many of these methods can be used simultaneously or in sequence.

Child particle algorithms

Most of the algorithms in this section for placing child particles are based on some randomizing scheme. This is done in order to achieve complexity in the gross structure of accumulated particle collection. One of the primary issues in taking this approach is the control of the shape and size of the collection after all of the randomization has taken place.

Uniform space filling

In this approach, child particles are placed randomly within a specified volume, without history. By history, I mean that the placement of a child has no correlation with the placement of any other child. For example, uniformly filling a rectangular box has this form:

```

float boxheight, boxwidth, boxlength; /* values set by some criterion */
vector displacement;

int setChildParticle_UniformRectangleFill()
{
displacement[0] = boxheight * (drand48() - 0.5);
displacement[1] = boxwidth * (drand48() - 0.5);
displacement[2] = boxlength * (drand48() - 0.5);
}

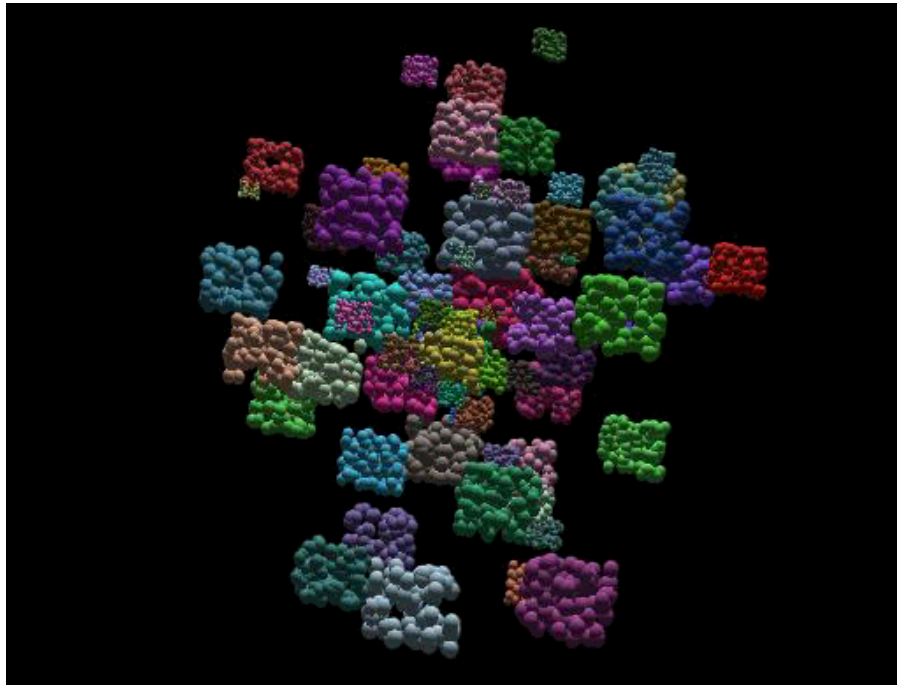
```

```

childposition = guideposition + displacement;
setAtt("position", childposition);
return 1;
}

```

The figure below is an example of uniform rectangle filling.



Rectangular uniform filling

For filling a sphere, the process is only a little different:

```

float sphereradius; /* value set by some criterion */
vector displacement;

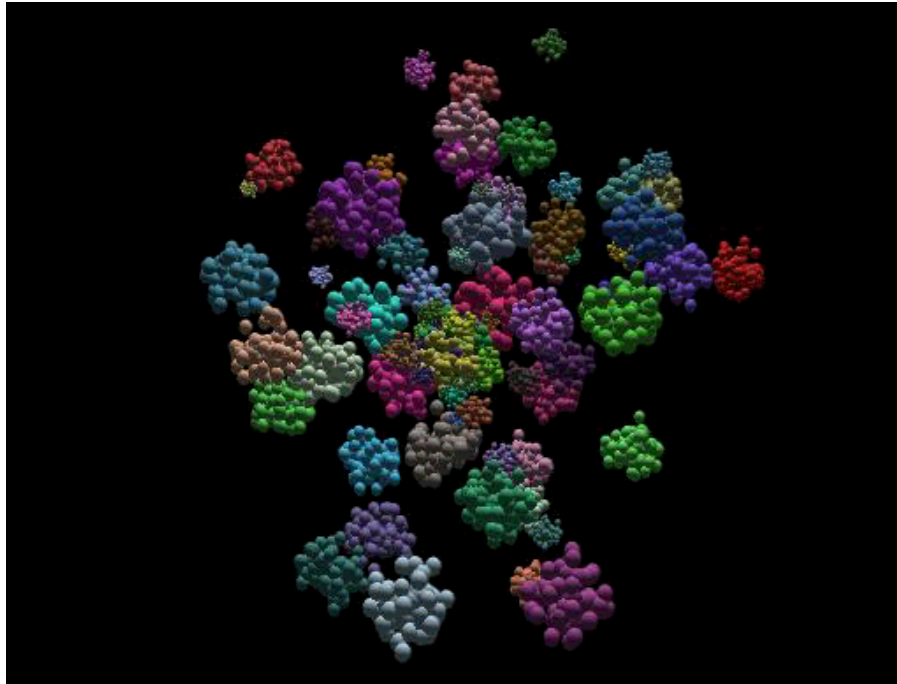
vector RandomUnitVector()
{
vector ruv;
float theta, phi;
theta = 3.14159265 * drand48();
phi = 2.0 * 3.14159265 * drand48();
ruv[0] = sin(theta) * cos(phi);
ruv[1] = sin(theta) * sin(phi);
ruv[2] = cos(theta);
return ruv;
}

int setChildParticle_UniformSphereFill()
{
displacement = RandomUnitVector();
displacement = sphereradius * displacement * drand48();
childposition = guideposition + displacement;
setAtt("position", childposition);
}

```

```
return 1;
}
```

The result is shown in the figure below.



Spherical uniform filling

To generate the images in this section, the `setChildParticleInfo()` routine has the form

```
void setChildParticleInfo()
{
/* dependent on application */
/* set child attributes that are fixed */

childradius = 0.2*guideradius;
setAtt("radiusPP", childradius);

maxchildcount = 100;

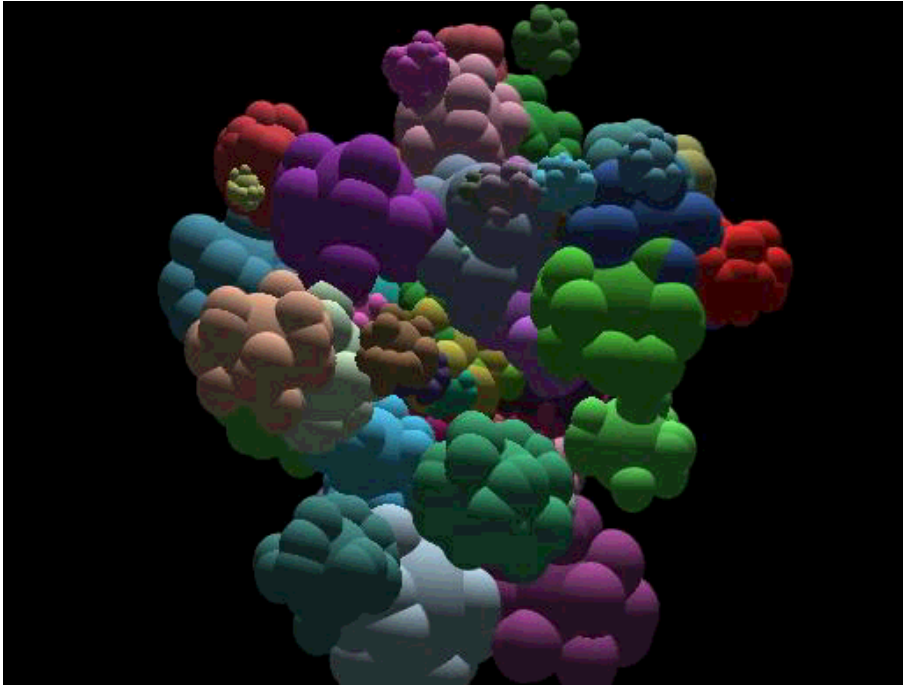
sphereradius = guideradius;
boxheight = guideradius;
boxwidth = 1.5 * guideradius;
boxlength = 2 * guideradius;

return;
}
```

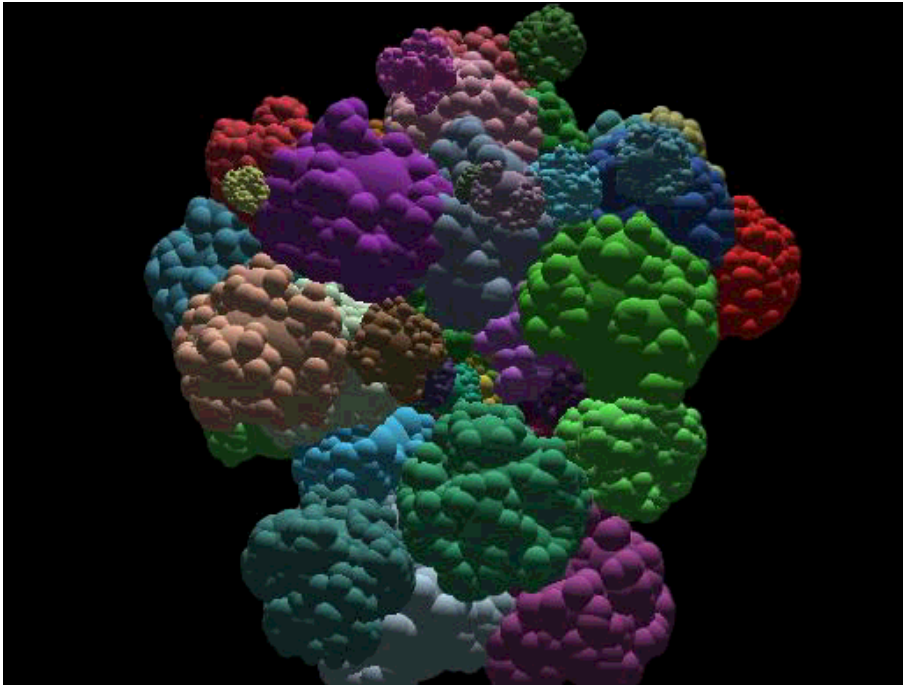
Cauliflowers

A cauliflower is a hierarchy of spheres. Beginning from a guide sphere, a collection of smaller spheres

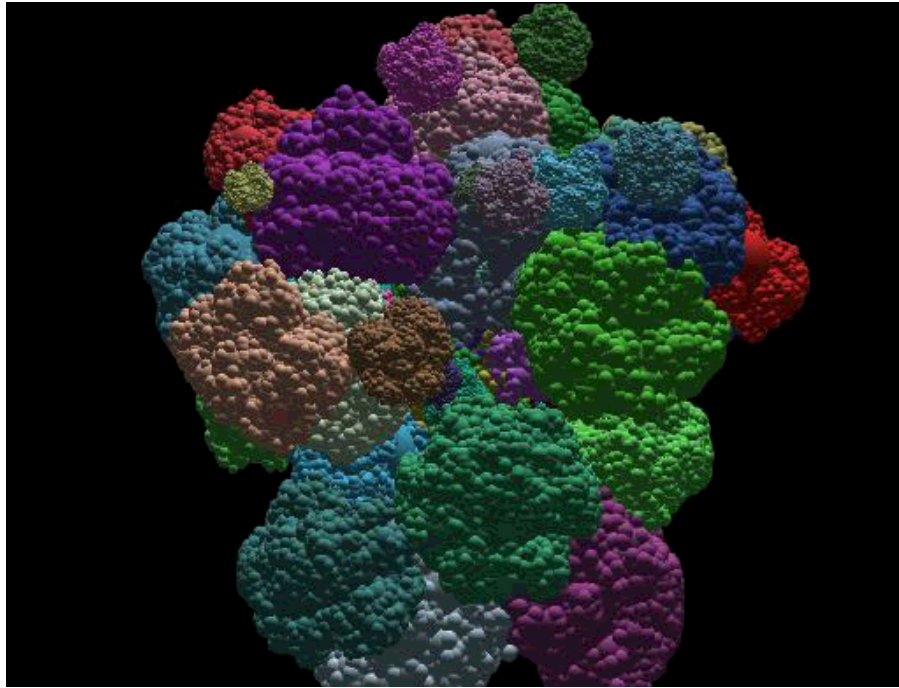
are placed on its surface. If you desire to go to one more level, each of the spheres on the surface of the guide sphere can have a set of spheres placed on their surfaces. This process can continue as far as you like in a recursive way. The figure below illustrates one, two and three levels of sphere placement.



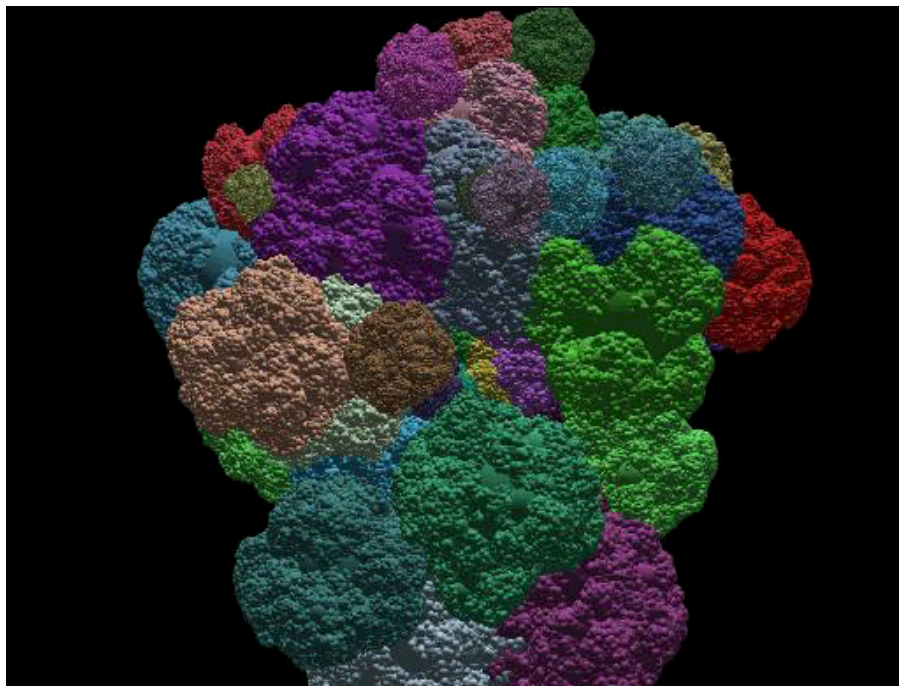
One recursion of cauliflower growth.



Two recursions of cauliflower growth.



Three recursions of cauliflower growth.



Four recursions of cauliflower growth.

Carrying this much farther would lead to a special object called *Fractal Growth Pattern*, which is also related to the random walk discussed in sections below.

For our algorithm, the new important parameters are:

```

/* cauliflower distribution */
float cauliflowerscale;
int nbcauliflowerclumps, nbcauliflowerrecursions;

```

The parameter `cauliflowerscale` controls the relative size of spheres in each recursion level. At each level, `nbcauliflowerclumps` spheres are placed on the surface of each sphere from the previous level. Finally, the total number of recursion levels is `nbcauliflowerrecursions`. These three parameters are used to recursively create the cauliflower. The approach is:

```

void putbumps(vector gpos, float gradius, int nb, float scale, int rec)
{
    int p, trackrec;
    float theta, phi, radius;
    vector pos;

    /* track the number of recursions */
    trackrec = rec-1;

    /* loop over particles to clump onto surface of parent */
    p = 0;
    while(p < nb)
    {
        /* place the particle on the surface of its parent */
        pos = RandomUnitVector();
        pos = gpos + gradius * pos;
        setAtt("position", pos);

        /* give the particle a reduced size */
        radius = gradius * scale;
        setAtt("radiusPP", radius);

        emit();

        /* now recurse to put particles on the surface of this particle */
        if(trackrec > 0){putbumps(pos, radius, nb, scale, trackrec);}
        p = p + 1;
    }
    return;
}

int setChildParticle_Cauliflower()
{
    emit(); /* emit guide particle */
    /* recursively places and emits particles */
    putbumps(guideposition, childradius, nbcauliflowerclumps,
        cauliflowerscale, nbcauliflowerrecursions);
    return 0; /* dont emit because emission occurred during recursion */
}

```

Several new things are going on in this algorithm. First, because of the recursion, all of the `emit()` calls take place inside `setChildParticle_Cauliflower()`, and its return value is zero so that no additional `emit()` calls are made. The second new technique is recursion. With each recursive call to `putbumps()`, the next level of spheres are placed, with a sphere radius that is larger or smaller than the previous level by a factor of `cauliflowerscale`. By setting this parameter less than one, the bumps get smaller and smaller. In the example images above, the values used were:

```

/* cauliflower placement */
cauliflowerscale = 0.5;
nbcauliflowerclumps = 20;
nbcauliflowerrecursions = 3;

```

Curves in space

Space curves are simply a parametric representation of the continuous sequence of points. We can draw the curve simply by placing particles in sequence along the length of the curve path. Positions on the curve are generated by knowing the position of an anchor point, and a parameter that traverses the length of the curve.

In the algorithm below, we generate a simple space curve which has constant helicity and curvature. For this, the important data to track is

```

/* Constant helicity and curvature space curve */
float pathlength, dpath, helicity, curvature;
vector Tangent, Normal, Binormal, T, N, B, G;

/* Space Curve Routines */

void ComputeSpaceCurveGamma()
{
float hh;
hh = 1.0/sqrt(1.0 + helicity*helicity);
G = (helicity * B - T ) * hh;
return;
}

vector ComputeSpaceCurvePosition()
{
vector Position;
float cl, sl, theta;
theta = pathlength * curvature;
sl = sin(theta);
cl = cos(theta);
float hh;
hh = 1.0/sqrt(1.0 + helicity*helicity);
Position = T * theta;
Position = Position - N * cl * hh;
Position = Position + G * (theta - sl) * hh;
Position = Position / curvature;
return Position;
}

void ComputeSpaceCurveTangent()
{
float cl, sl, theta;
theta = pathlength * curvature;
sl = sin(theta);
cl = cos(theta);
float hh;
hh = 1.0/sqrt(1.0 + helicity*helicity);
Tangent = T;
Tangent = Tangent + N * sl * hh;
Tangent = Tangent + G * (1.0-cl) * hh;
return;
}

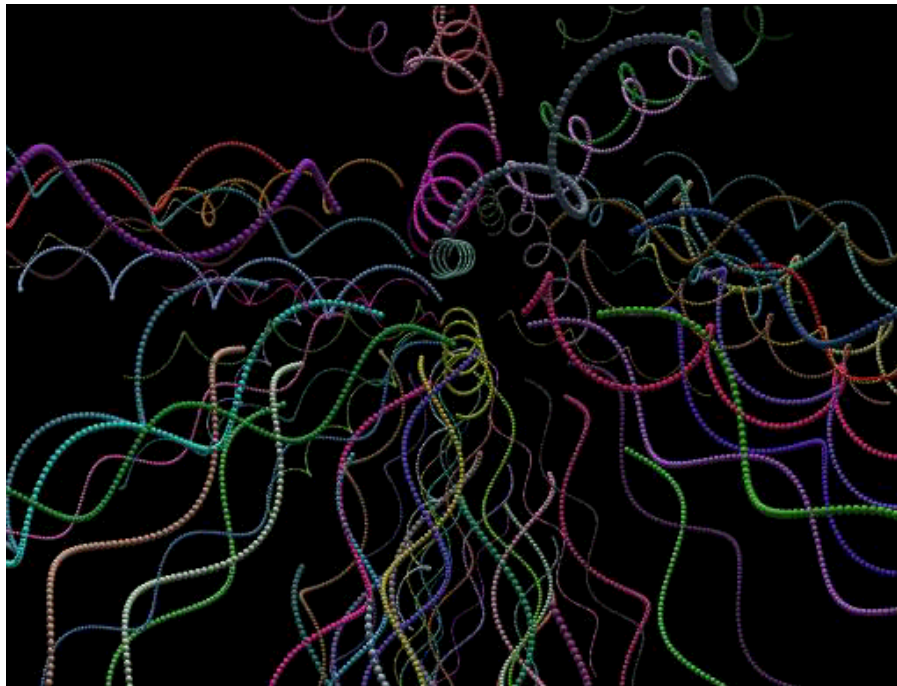
```

```

void ComputeSpaceCurveNormal()
{
float cl, sl, theta;
theta = pathlength * curvature;
sl = sin(theta);
cl = cos(theta);
float hh;
hh = 1.0/sqrt(1.0 + helicity*helicity);
Normal = N * cl + G * sl;
return;
}

void ComputeSpaceCurveBinormal()
{
float cl, sl, theta;
theta = pathlength * curvature;
sl = sin(theta);
cl = cos(theta);
float hh;
hh = 1.0/sqrt(1.0 + helicity*helicity);
Binormal = B - ( N * sl + G * (1.0-cl) ) * helicity * hh;
return;
}

```



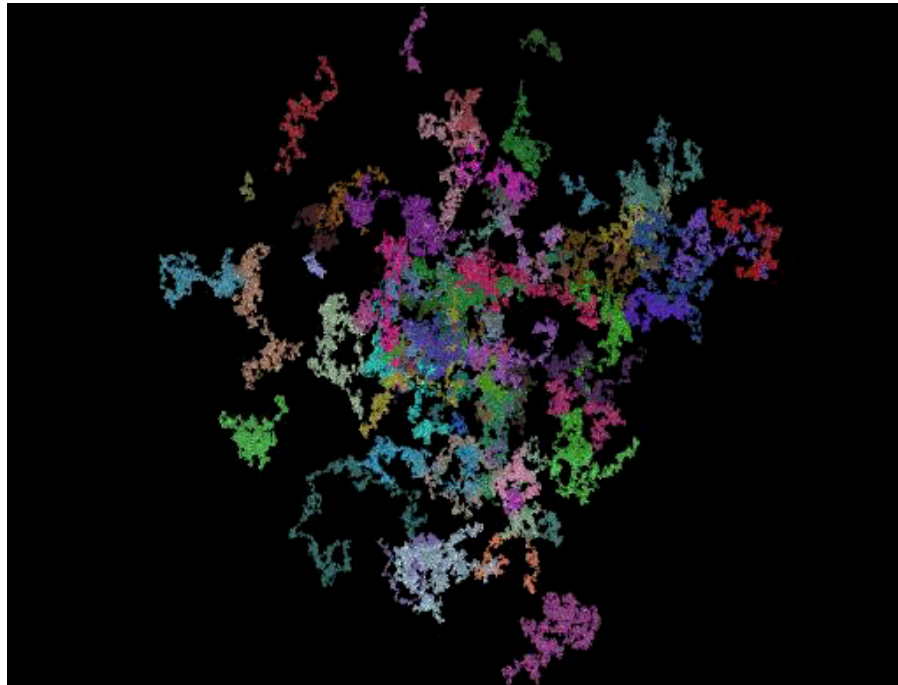
Space curves drawn from the guide particles.

Basic Random Walk

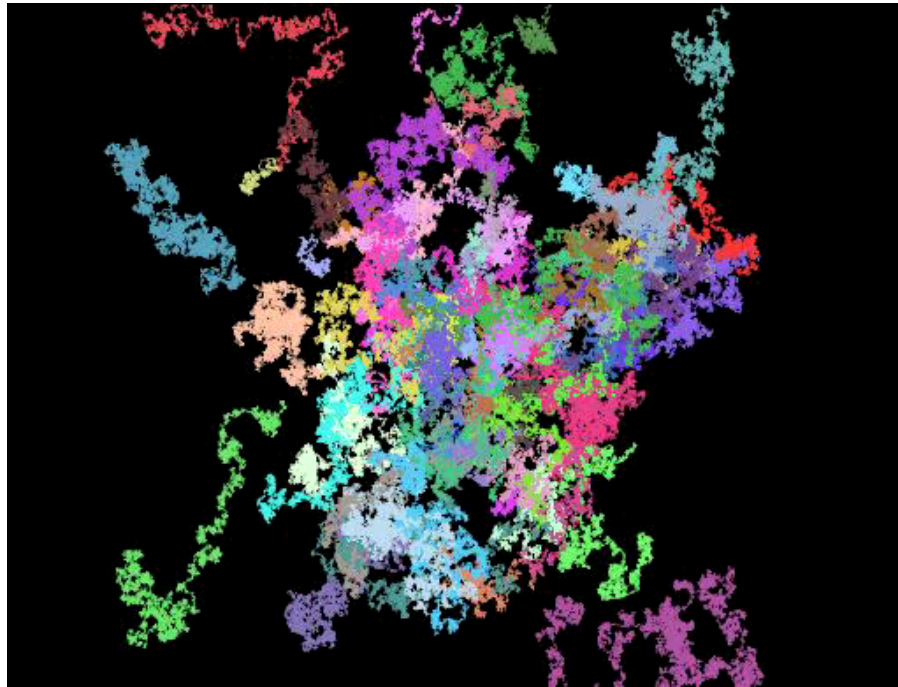
A random walk is basically just that. The basic of each successive child particle is a random step away from the position of the previous child particle. Over the space of many steps, the connected path can form a complex, fractally shape. Of course, there are many types of randomness, producing random

walks with a variety of characteristics. We explore several of these in this and the next four algorithm sections.

The figure below was generated from the most basic random walk method. To accomplish this, the particles are very small (0.005 times the guide particle radius), and lots of them are used (20000 per guide particle). When even more particles are used in the next image (200,000 per guide particle, over 15 million total), you can clearly see that the random walk marches all over space.



The most basic random walk. There are 20000 child particles for each guide particle, over 1.5 million total.



Longer version of the basic random walk. There are 200000 child particles for each guide particle, over 15 million total.

The random walk process uses the following code:

```
vector RandomStep()
{
vector ruv;
ruv[0] = drand48() - 0.5;
ruv[1] = drand48() - 0.5;
ruv[2] = drand48() - 0.5;
return ruv;
}

int setChildParticle_BasicRandomWalk()
{
childposition = childposition + walkstep * RandomStep();
setAtt("position", childposition);
return 1;
}
```

The routine `RandomStep()` generates a vector with components lying in the range $(-0.5, 0.5)$. The current child particle is placed at a position that is randomly displaced from the previous particle, with the distance of the displacement in the range $(0, \sqrt{3}/2 \text{ walkstep})$.

There are no bounds on the size or length of random walk achievable. Because of the randomness in the direction and size of the steps however, there is a theoretical estimate of the approximate range the random walk will occupy. Based on the statistics of the random numbers being used, the root mean square step sizes is $S = \text{walkstep}/\sqrt{24}$. After `maxchildcount` steps, the range of the random walk should be roughly $(S \sqrt{\text{maxchildcount}})$.

Correlated Random Walk

As might be expected, the correlated random walk is very similar to the basic random walk. The difference comes in one spot only: the random walk set is no longer independent from particle to particle. A statistical correlation is introduced in the following way:

```
int setChildParticle_CorrelatedRandomWalk()
{
walk = walk * mixin + walkstep * RandomStep() * mixout;
childposition = childposition + walk;
setAtt("position", childposition);
return 1;
}
```

The parameters `mixin` and `mixout` are mix the previous value of the random step with the new one, and so $0 < \text{mixin} < 1$, and $0 < \text{mixout} < 1$, with $\text{mixin} + \text{mixout} = 1$. The special case `mixin = 0` is the uncorrelated basic random walk, while at the other extreme `mixin = 1` produces curly straight lines.

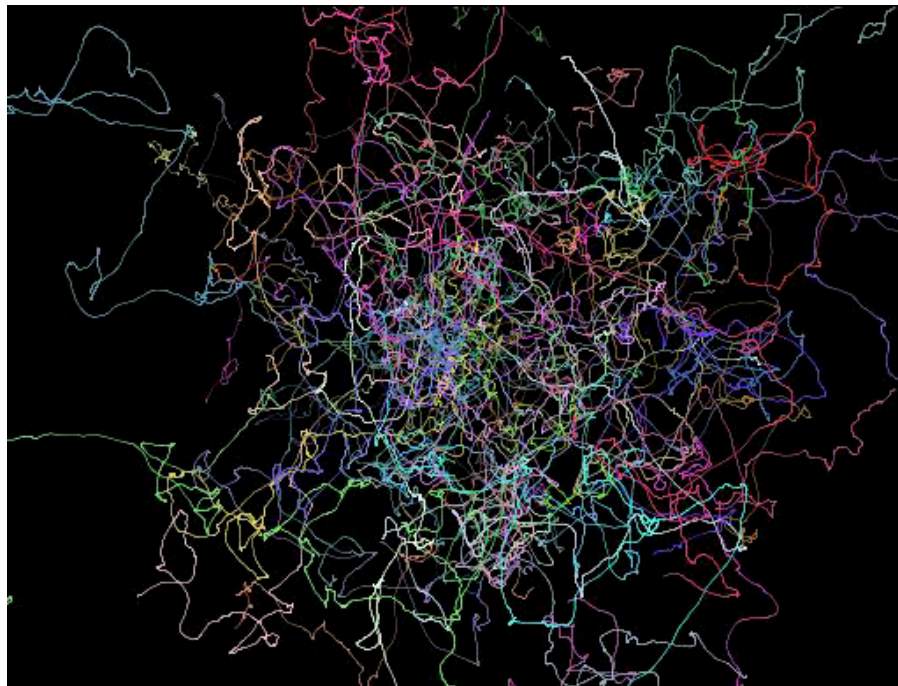
The range in between uncorrelated random walk and straight line possesses a wide range of behaviors that we will try to organize in this section.

The first thing to sort out is what the impact of correlation is. The figure below shows that clearly: correlation turns the path that the particles are laid out on into a "smooth" curve with unpredictable twists and turns. As the `mixin` parameter approaches 1, the number of kinks and turns become fewer. The example below for example, using 50000 particles per guide particle, with short spacing between them. The `mixin` is 0.9999. That's right, there are four 9's to the right of the decimal point.



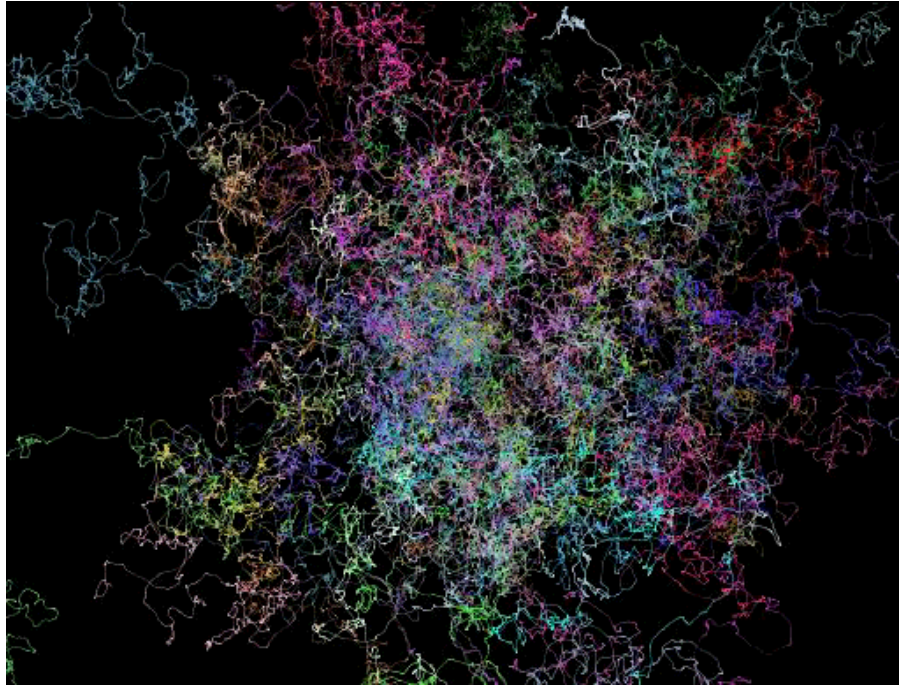
A correlated random walk, with correlation `mixin = 0.9999`. Despite the very high correlation, there is considerable structure in each path. There are 50000 child particles for each guide particle.

To give you some idea of the importance of 0.9999 versus 0.999 for example, the image below was produced with $\text{mixin} = 0.999$. There is a substantial difference between the two.



A correlated random walk, with correlation $\text{mixin} = 0.999$. There are 50000 child particles for each guide particle.

Finally, below is the same set of particles as the other two, but with $\text{mixin} = 0.99$.



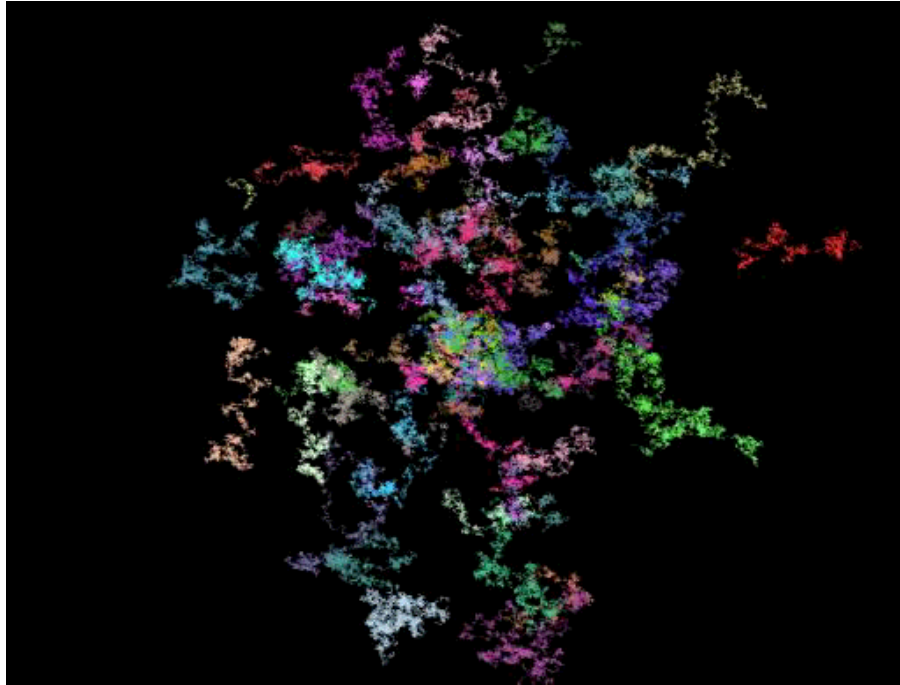
A correlated random walk, with correlation `mixin = 0.99`.

Spherical vs Cartesian walk

In the previous random walk examples, the vector step was of a "Cartesian" form, which means that the direction and length of the step were subject to randomization each step. This was accomplished by randomizing the three cartesian components of the step independently. An alternative useful method of conducting a random walk is to randomize the direction of each step, but leave the magnitude of the step fixed. This is "spherical" walk. In practice, it requires only a small change in the code to be accomplished. Using the random unit vector generator from the uniform spherical filling, the `setChildParticle` code is simply:

```
int setChildParticle_SphericalRandomWalk()
{
  childposition = childposition + walkstep * RandomUnitVector();
  setAtt("position", childposition);
  return 1;
}
```

So, in the spherical random walk, successive steps are a distance exactly `walkstep` apart, whereas in the basic cartesian random walk, the distance between steps can be as little as 0 and as much as `walkstep`. This has the effect of leaving the spherical random walk volume less dense in the center, as shown in the example below.

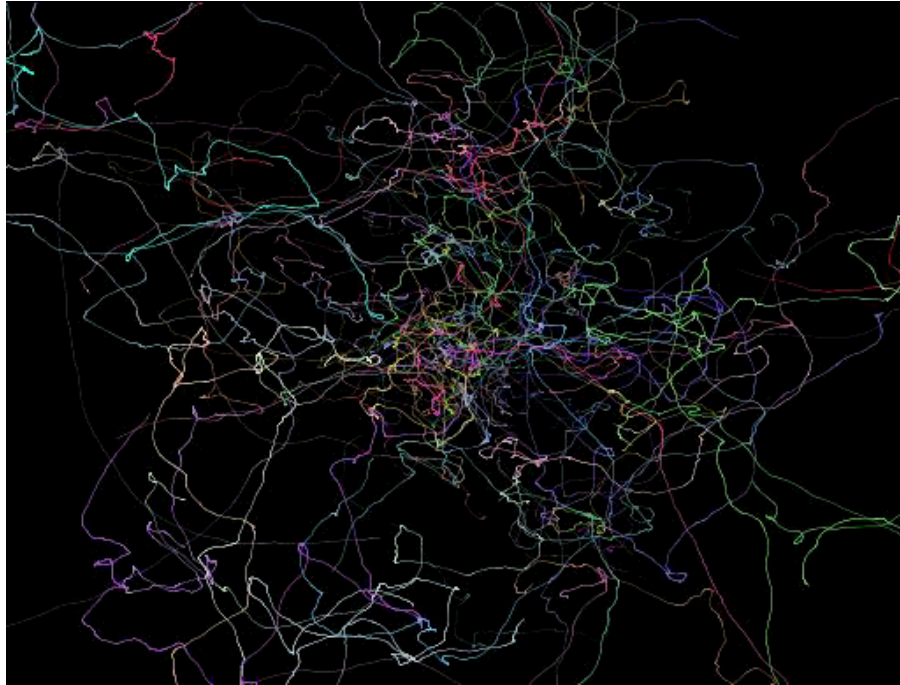


A spherical (and uncorrelated) random walk.

Just as with the cartesian random walk, we can introduce correlation in the walk. This is accomplished this way:

```
int setChildParticle_CorrelatedSphericalRandomWalk()
{
walk = walk * mixin + walkstep * RandomUnitVector() * mixout;
childposition = childposition + walk;
setAtt("position", childposition);
return 1;
}
```

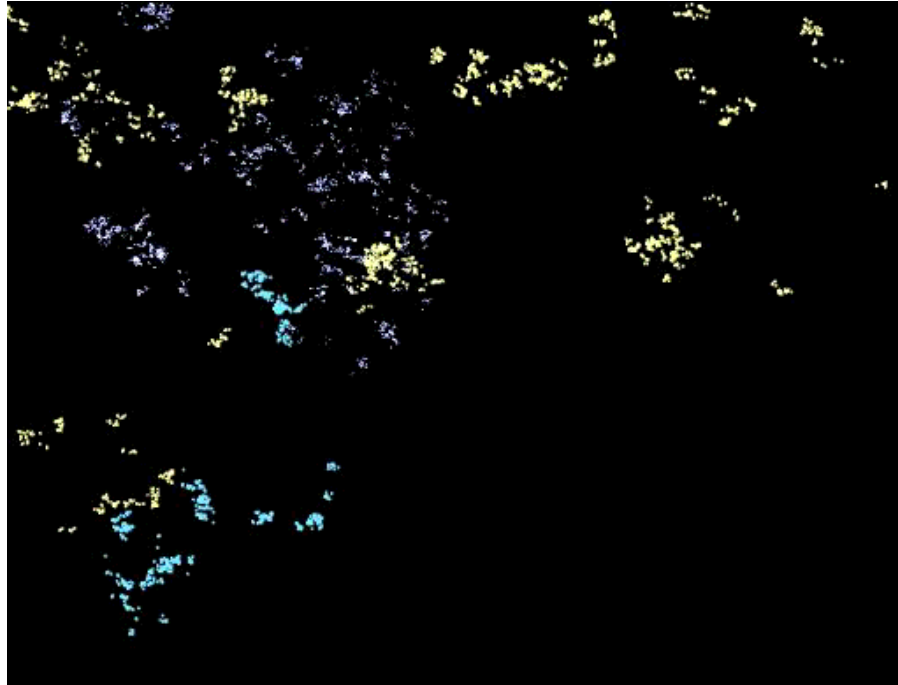
We have introduced the `mixin` and `mixout` parameters that same as in the basic cartesian case. An example of correlated spherical walk is below:



A spherical random walk with correlation $\text{mixin} = 0.999$.

Levy Flight

A Levy Flight is a kind of random walk. The fundamental difference between Levy Flights and other random walks is that the length of the steps of a Levy Flight are chosen at random from a power law distribution. The impact of that choice is that the average size of the walk after a number of steps is very different from other random walks. Levy Flights tend to have regions that look similar to ordinary random walks, but the regions are connected by occasional long single steps. This produces a nice clumping effect, as illustrated in the figure below.



A Levy Flight random walk. In this case, the parameter settings were $LevyMu = 2.2$ and $LevyMin = 0.03 * guideradius$. There was no correlation in the steps, and there were 2000 child particles (i.e. Levy steps) per guide particle.

The code for performing a Levy Flight is

```
vector RandomLevyStep()
{
    vector ruv;
    ruv = RandomUnitVector();
    float step;
    step = LevyMin * pow(drand48(), 1.0/(1.0-LevyMu));
    ruv = step * ruv;
    return ruv;
}

int setChildParticle_RandomLevyWalk()
{
    childposition = childposition + RandomLevyStep();
    setAtt("position", childposition);
    return 1;
}
```

Directed walk

Walking the Surface of Implicit Surfaces

As discussed so far, random walks traverse space without any real restrictions on where they travel. For some purposes, it is useful to perform a random walk in order to get the complex structure, but limit the region the walk can occur in. Limiting the walk is the focus of this section.

Before proceeding, the convention chosen here for implicit surfaces is that the implicit function has a value of zero on the surface, has a positive value inside the surface, and a negative value outside the surface.

First, we need a method of computing the relevant information about the implicit surface. For this problem, we need routines that can retrieve two pieces of information:

1. For any point in space, what is the value of the implicit function.
2. For any point in space, what is the "normal" vector, defined as the unit vector that lines up with the gradient of the implicit function at that point, pointing outwards from the surface.

These two type of information will be contained in two function: `getISFunction()` and `getISNormal()`. For the simplest case of a spherical implicit surface, these can take the form

```
float getISFunction(vector r)
{
    /* simple circle case */

    vector test;
    test = r - positionIS;
    float returnvalue;
    returnvalue = 1.0 - (test.test)/(radiusIS*radiusIS);
    return returnvalue;
}

int getISSign(vector r)
{
    /* > 0 => inside surface; < 0 => outside surface */
    float ISFunctionvalue;
    ISFunctionvalue = getISFunction(r);
    int returnvalue;
    returnvalue = 0;
    if(ISFunctionvalue > 0){ returnvalue = 1; }
    if(ISFunctionvalue < 0){ returnvalue = -1; }
    return returnvalue;
}

vector getISNormal(vector r)
{
    /* simple circle case */
    vector test;
    test = r - positionIS;
    test = test / sqrt(test . test);
    return test;
}
```

We have also added the routine `getISSign()` which uses `getISFunction()` to decide if a point in space is inside or outside the implicit surface. While we have built functions for spherical implicit surfaces, the process described here is applicable for any implicit surface.

The random walk illustrated in `setChildParticle_CorrelatedTraverseIS()` below is altered at each step to that is preferentially moves along the normal toward the surface of the Implicit Surface. At any step, the walk determines whether the current child position is inside or outside the surface, then re-oriens the random step so that it moves toward the surface. There is still a random element in each step perpendicular to the normal.

```

int setChildParticle_CorrelatedTraverseIS()
{
    float ISsign;
    vector ISnormal;
    ISnormal = getISNormal(childposition);

    /* ISsign > 0 => inside; ISsign < 0 => outside */
    ISsign = getISSign(childposition);

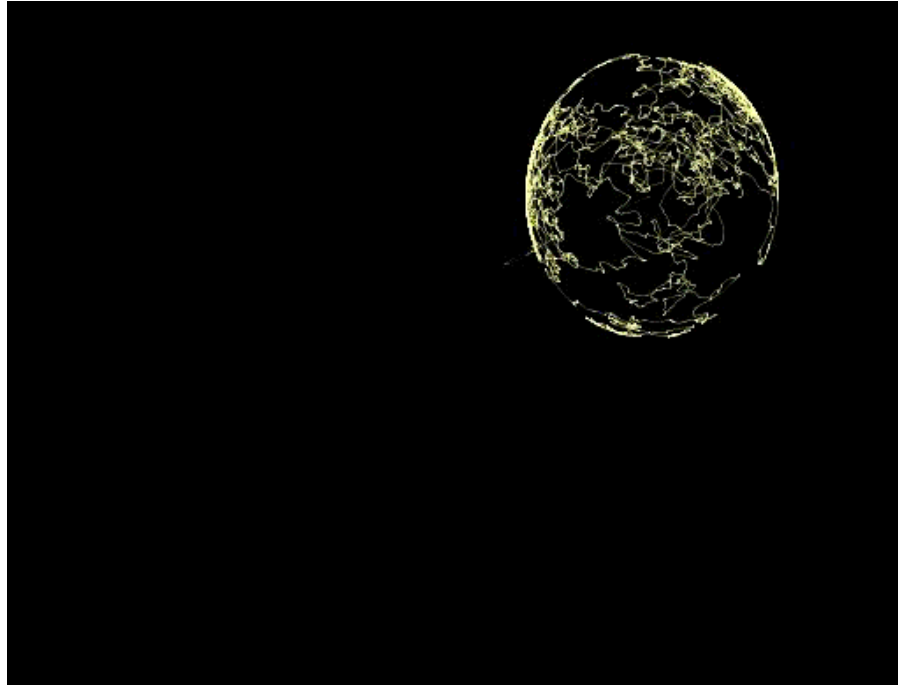
    /* step in a random direction perpendicular to IS */
    walk = RandomUnitVector();
    walk = walk - (walk . ISnormal) * ISnormal;
    walk = walk / sqrt(walk . walk);
    walkIS = walkIS * mixin + stepIS * (ISsign * ISnormal + walk) * mixout;
    childposition = childposition + walkIS;
    setAtt("position", childposition);
    return 1;
}

```

The two examples results below demonstrate that the random walk is bound to the implicit surface.



A random walk along the surface of an Implicit Surface (sphere), with `mixin = 0.0`.



The same random walk along the surface of an Implicit Surface as above, with `mixin = 0.99`.

Filling the inside (and outside) of Implicit Surfaces

It takes only a simple modification of the previous `setChildParticle` technique to have the random walk fill the inside of the implicit surface. The difference is that, when the particle is inside the implicit surface, the random walk is unaffected (whereas previously it was biased toward the surface). So now the routine `setChildParticle_CorrelatedFillIS()` looks like:

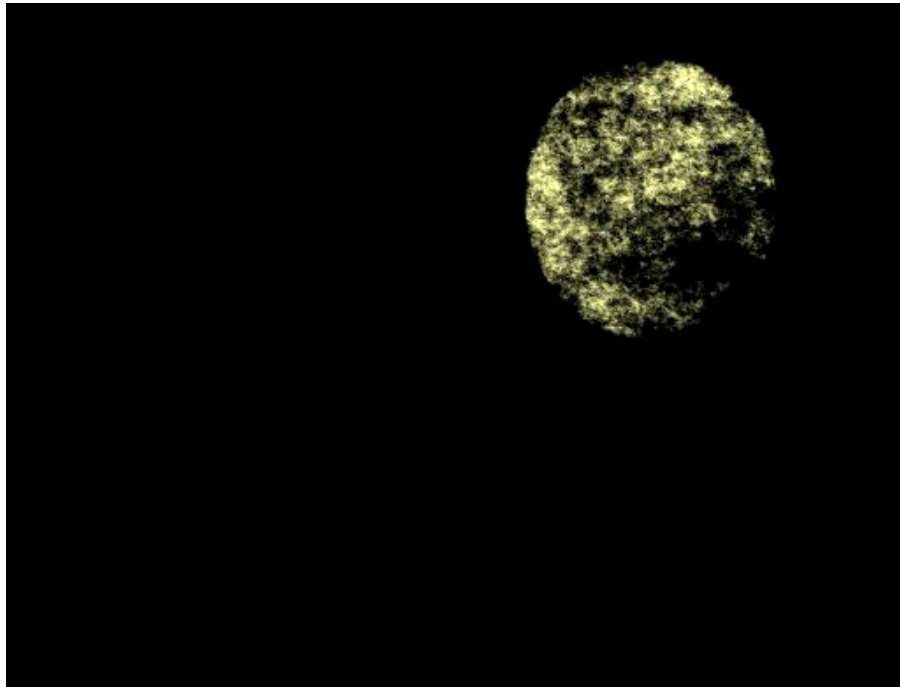
```
int setChildParticle_CorrelatedFillIS()
{
    float ISsign;
    /* ISsign > 0 => inside; ISsign < 0 => outside */
    ISsign = getISSign(childposition);

    /* step in a random direction perpendicular to IS */
    walk = RandomUnitVector();
    if(ISsign < 0) /* Outside: redirect inside */
    {
        vector ISnormal;
        ISnormal = getISNormal(childposition);
        walk = walk - (walk . ISnormal) * ISnormal;
        walk = walk / sqrt(walk . walk);
        walkIS = walkIS * mixin + stepIS * (ISsign * ISnormal + walk) * mixout;
    }
    else /* Inside: let it go */
    {
        walkIS = walkIS * mixin + stepIS * walk;
    }
    childposition = childposition + walkIS;
    setAtt("position", childposition);
    return 1;
}
```

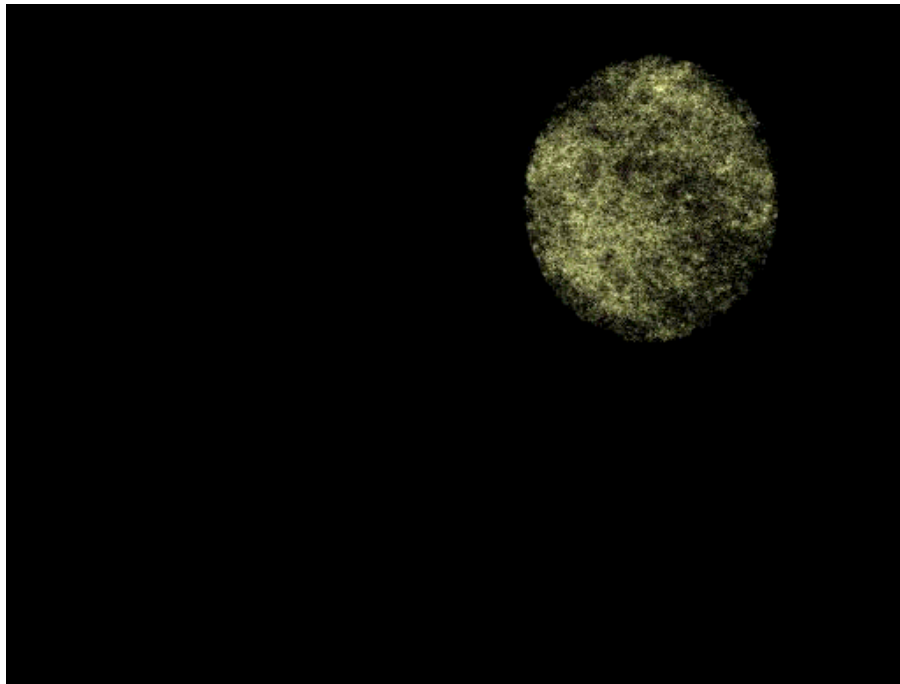
}

As you can see, the difference is that now if the particle is inside, the random walk is unaltered.

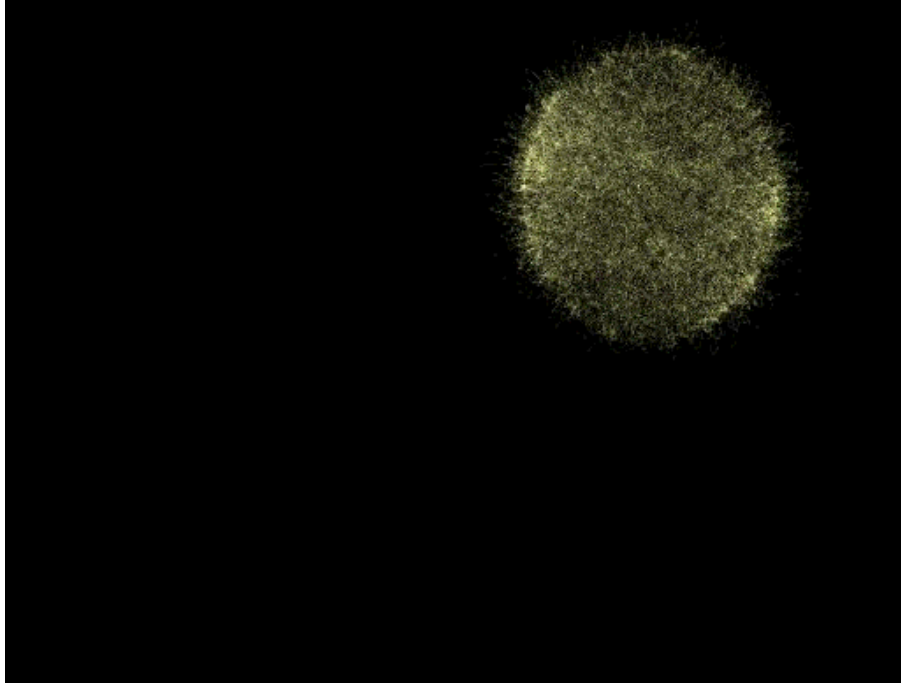
The examples below demonstrate filling for various random walk correlations.



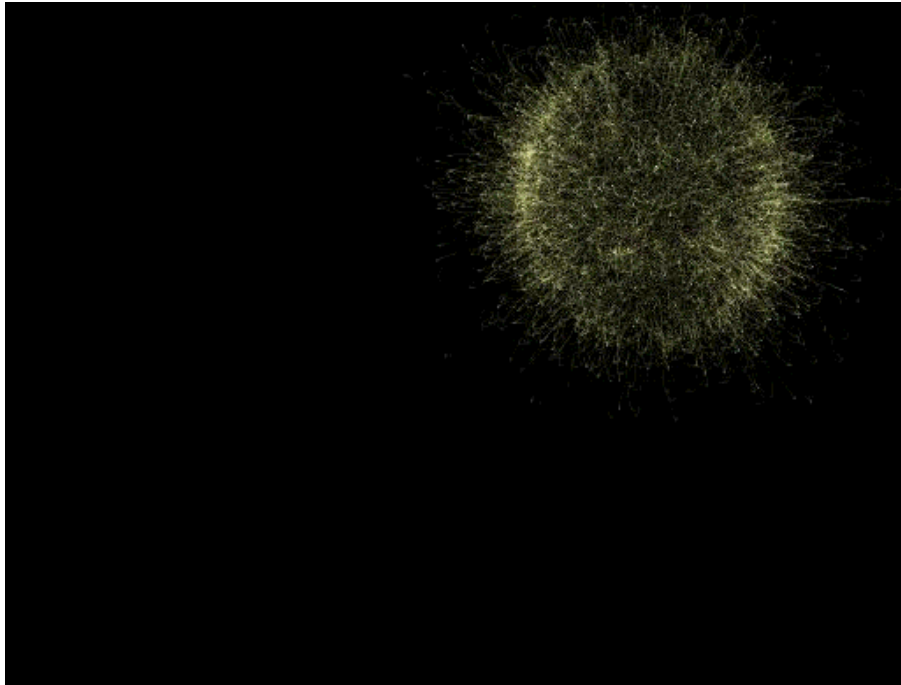
A random walk inside an Implicit Surface (sphere), with `mixin = 0.0`.



A random walk inside an Implicit Surface, with `mixIn = 0.5`.



A random walk inside an Implicit Surface, with `mixIn = 0.9`.



A random walk inside an Implicit Surface, with `mixIn = 0.95`.



A random walk inside an Implicit Surface, with `mixin = 0.99`.

Time-dependence (pulses, shocks, waves, and cheesy dynamics)

The idea behind pulse is that a procedural method exists to tag positions along a random walk, and control any behavior with that tag. Because we have access to the frame number via the parameter `$frame` provided by the `pdbEditor`, the tag can be positions along the walk in a time dependent way.

An example of this is the following code:

```
float cosh(float arg)
{
    float b;
    b = exp(arg);
    return 0.5*(b + 1.0/b);
}

float ComputeSpread()
{
    float pl;

    /* compute relative centroid of pulse based on speed and time */
    pl = pathlength - pulse_spreadspeed * $frame;

    /* use cosh function for a on-off behavior */
    return pulse_minspread + (pulse_maxspread-pulse_minspread) / cosh(pl/pulse_spread)
}

int setChildParticle_Pulse()
{
    /* Start out similar to a space curve */
}
```

```

childposition = guideposition + ComputeSpaceCurvePosition();
ComputeSpaceCurveNormal();
ComputeSpaceCurveBinormal();
ComputeSpaceCurveTangent();

pathlength = pathlength + dpath;

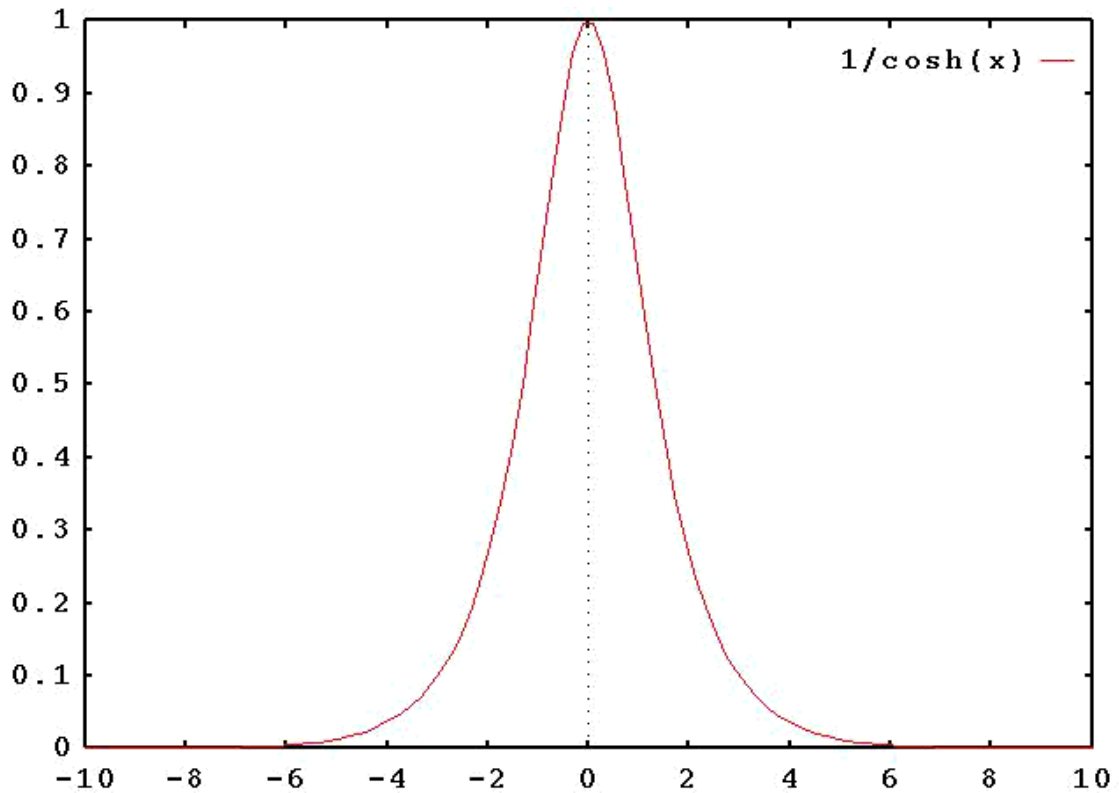
/* Now build a cloud around that position with a thickness */
float spread, angle;
spread = ComputeSpread();
int cloudsize, cloudcount;
cloudsize = pulse_cloudsize * (spread/pulse_minspread);
cloudcount = 0;
while(cloudcount < cloudsize)
{
    displacement = RandomStep();
    /* extract and scale component along tangent */
    pulse_position = childposition + dpath * (displacement.Tangent)*Tangent;

    /* extract and scale component perp to tangent */
    displacement = displacement - (displacement.Tangent) * Tangent;
    pulse_position = pulse_position + spread * displacement;
    setAtt("position", pulse_position);
    emit();
    cloudcount = cloudcount + 1;
}
return 0;
}

```

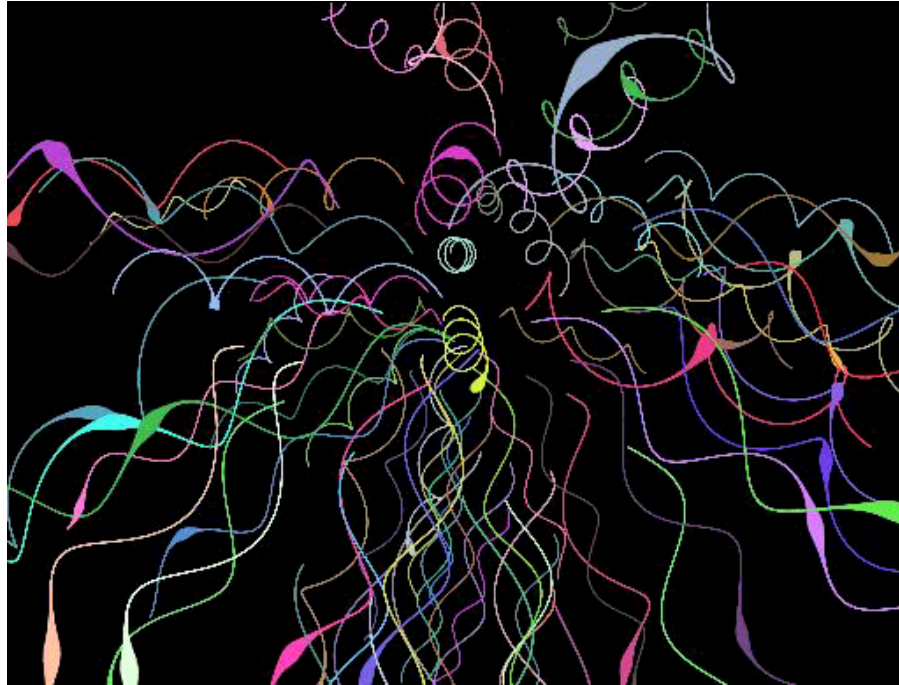
What does this code do? The first few lines of `setChildParticle_Pulse()` just set up the walk as a `SpaceCurve`, exact as in the previous *Curves in space* section. Then a routine called `ComputeSpread()` is invoked. The number that comes from this is used in two ways. First, a set of "grandchild" particles are created, and the number of them depends on the `spread` value - the higher the `spread`, the more particles. Each grandchild is created as part of an ordinary random placement, but the placement is confined to the disk perpendicular to the space curve at that point. The distance that the placement extends perpendicular to the curve is controlled by the value of `spread` also. Effectively, `spread` controls the local thickness of the curve in space.

Now examine how `spread` is computed. The computation involves the inverse of the `cosh()` function, which is plotted below:



A plot of the $1/\cosh()$ function.

So the `ComputeSpread()` routine returns the value `pulse_minspread` for all points on a walk except when `p1 = pathlength - pulse_spreadspeed * $frame` nears a value of zero, where it smoothly changes to the value `pulse_maxspread`. But that near-zero point is a function of frame number, so over a sequence of frames, the segment of a walk with `ComputeSpread()` different from `pulse_minspread` moves farther out on the walk, giving the illusion of pulse propagation. This image below illustrates a frame of pulses in a set of space curves.



A set of particle strings with a pulse. As time progresses, the pulse propagates along each string.

This effect can be modified so that instead of producing a limited region of a pulse, a shock-wave style effect can be produced. In this effect, once `ComputeSpread()` has smoothly changed from the value `pulse_minspread` to the value `pulse_maxspread`, it remain there. This is accomplished by the line

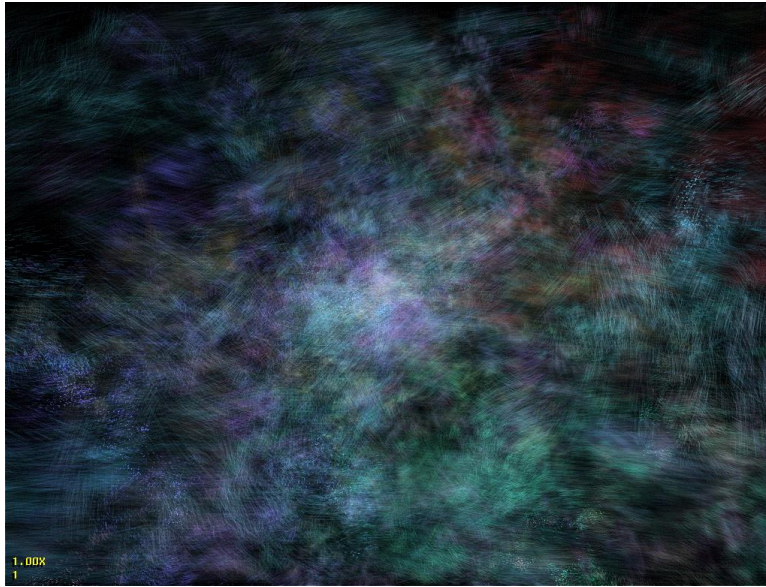
```
return pulse_minspread + (pulse_maxspread-pulse_minspread) /  
cosh(pl/pulse_spreadwavethickness);
```

in `ComputeSpread()` with the line

```
return minsread + (maxspread - minsread) * Heaviside(pl/spreadwavethickness);
```

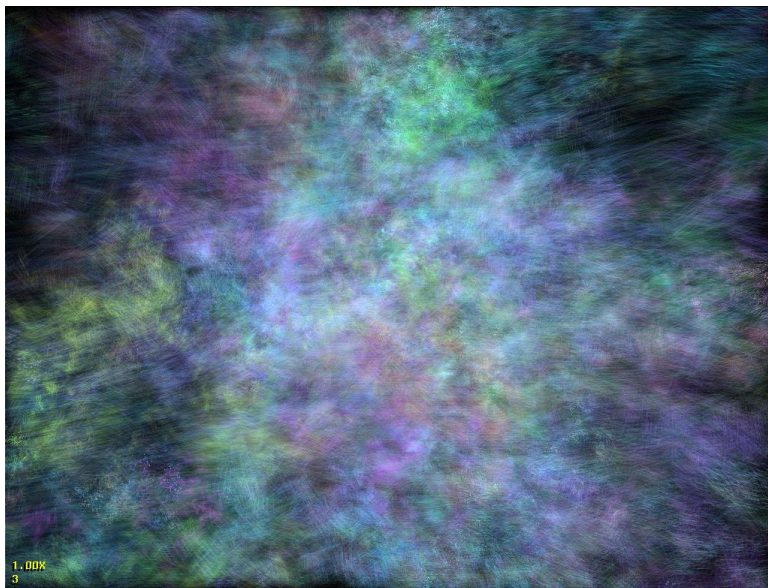
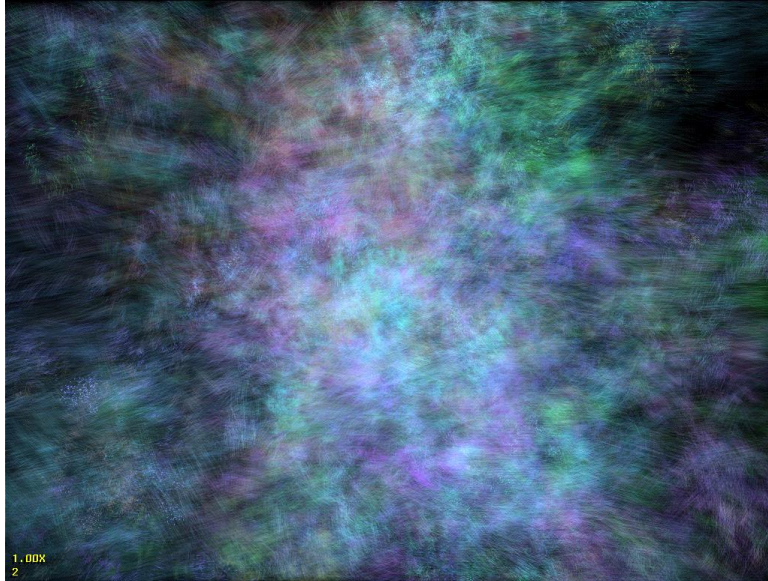
where the `Heaviside()` function is

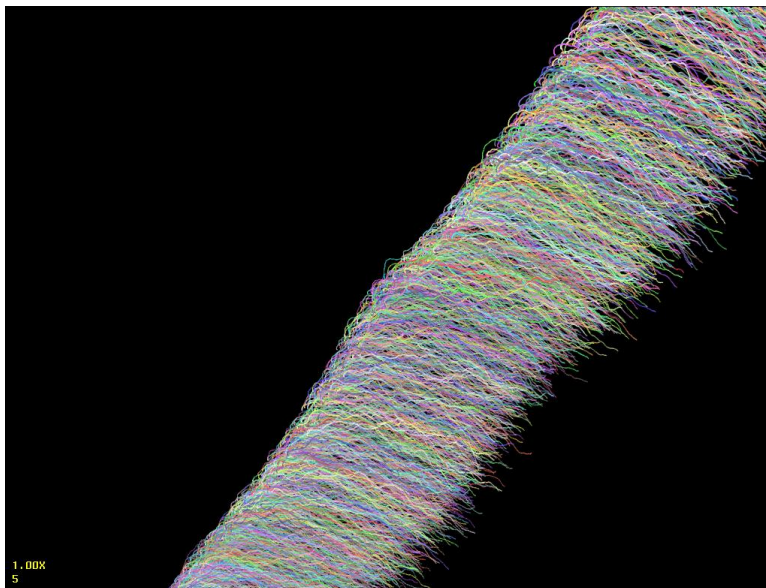
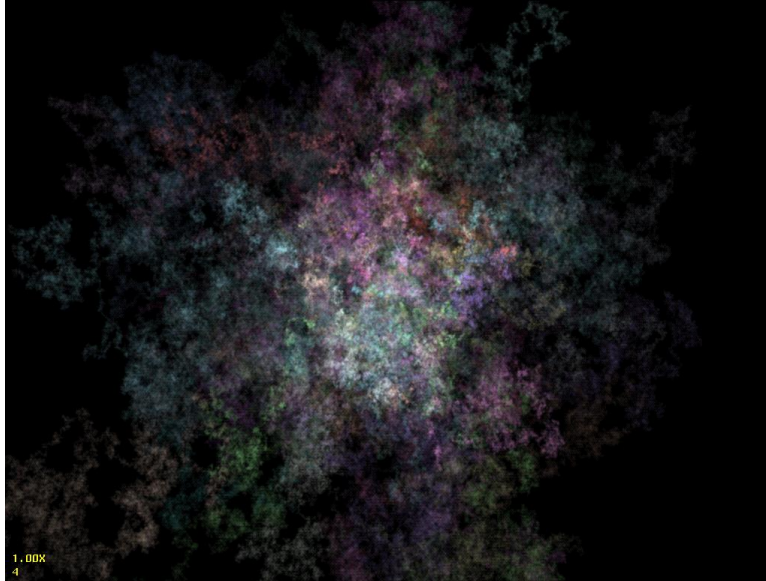
```
float Heaviside(float arg)  
{  
    float b;  
    b = exp(-arg);  
    return b/(1.0 + b);  
}
```

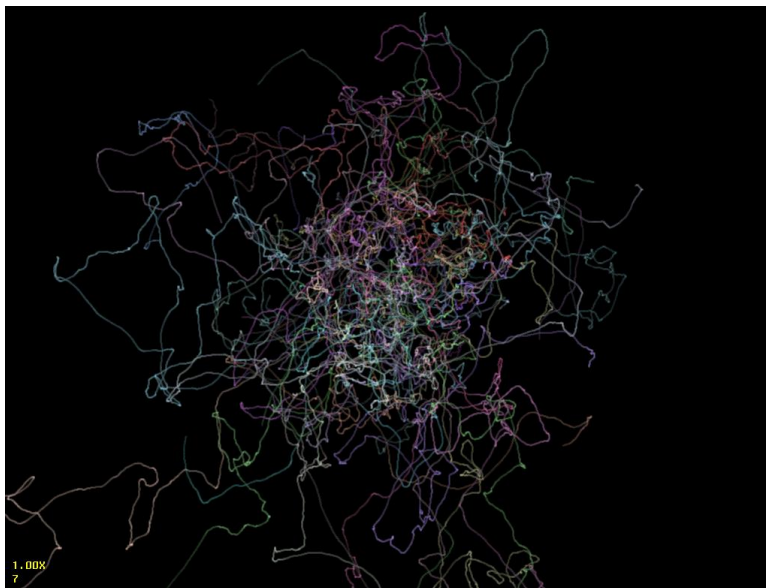
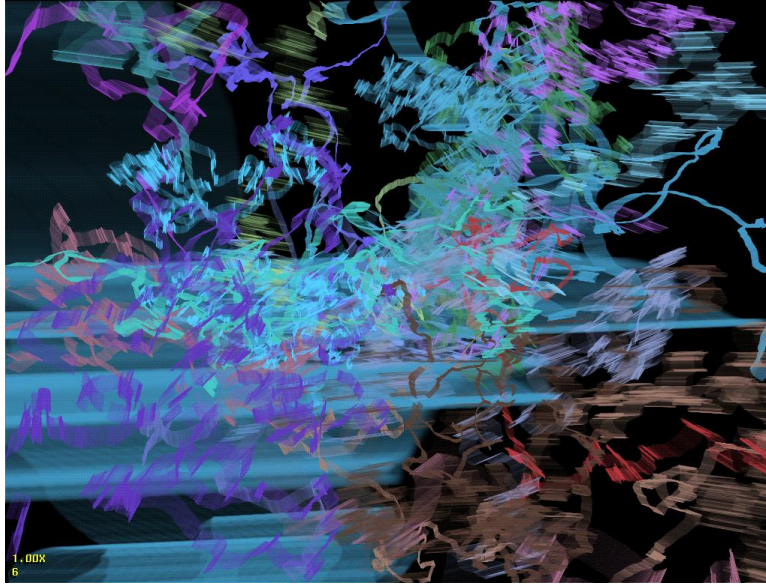


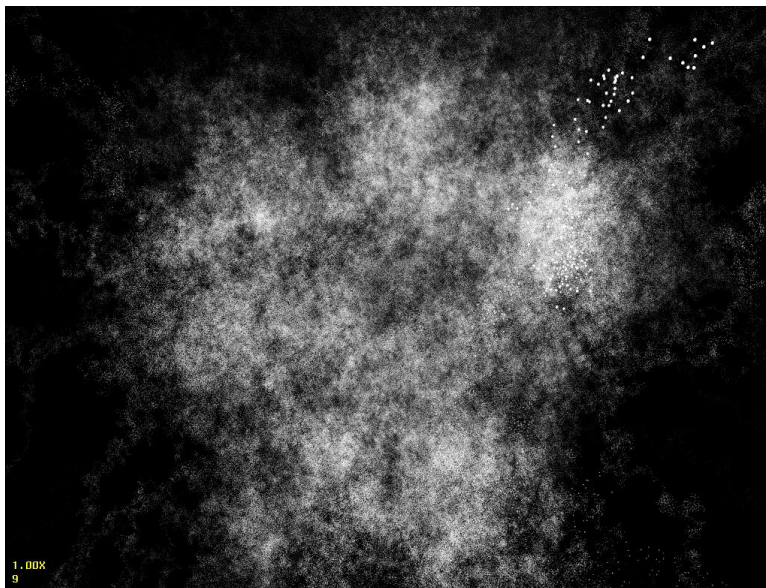
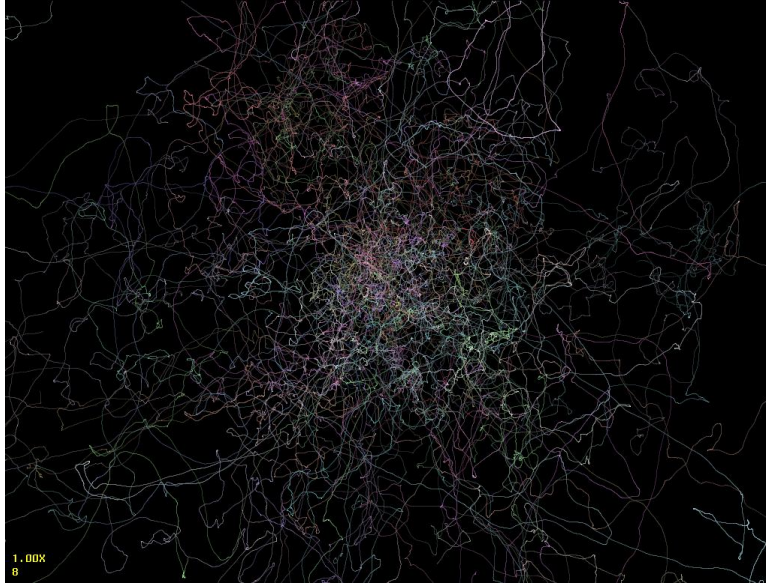
A.2 A Small Gallery of Particles

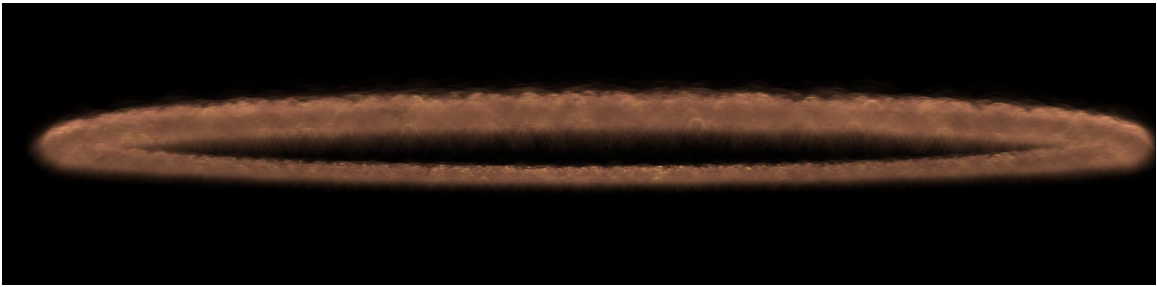
The following images were generated using the methods described above and the particle renderer *jahasa*.

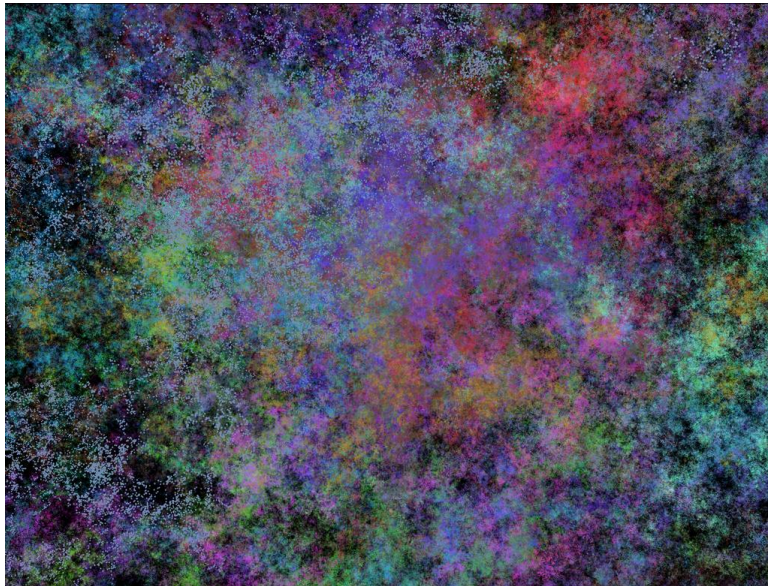












SPARSE GRID VOLUME RENDERING ON A GPU

This appendix is a report written for an honors undergraduate class

Authors: Sam Bryce and Zach Welch, Clemson University, 2012

B.1 Introduction

Volume rendering is a method for creating images from a three dimensional set of particles. Volume rendering has a wide variety of applications. Medical technology uses volume rendering in Computer Tomography (CT) scans and Magnetic Resonance Imaging (MRI). Scientific visualization applications often use volume rendering techniques. Volume rendering is often used in the digital special effects to render amorphous and/or gaseous elements like clouds, dust, smoke, or flames 5. Something like a cloud can be more realistically modeled by a volume of particles than the traditional graphics paradigm of representing objects as surfaces. This more realistic modeling in turn leads to a more realistic image. Volume renderers can take into account factors such as how light travels through various mediums and the effects of light scattering. The elements to be rendered are modeled as sets of volumetric data within a 3D grid. This grid is conceptually made up of $1 \times 1 \times 1$ cubes called voxels 2. Each voxel stores a density and a color of the particle at that specific voxel.

One of the key limitations of traditional volume rendering is memory size. If at each voxel we store a single 4 byte floating point number and three 4 byte floats representing the RGB color at this voxel, we end up storing 16 bytes for every voxel. A single voxel is inconsequential in terms of memory usage. However, volume rendering utilizes large grids of voxels to work. The larger the volume to be rendered and the more detailed the volume needs to be, the larger the array of voxels. Rendering a 1000^3 grid with 16 bytes allocated for each grid point would require a minimum of 15 GB of main memory, well beyond most machines. Also, it is very likely that a substantial majority of voxels are empty, meaning that much of the memory allocated is unused and therefore redundant. An ideal situation would involve storing only non-empty voxels in memory, with the understanding that if a voxel is not in memory, then it contains specified default values. This is the basic idea upon which sparse grids are built. Sparse grids, as opposed to dense grids that allocate memory for each grid point, regardless of its value, only store a portion of the data 4. When the volume renderer attempts to access data at a grid point, the sparse grid must determine whether this grid point is actually allocated and return the appropriate values.

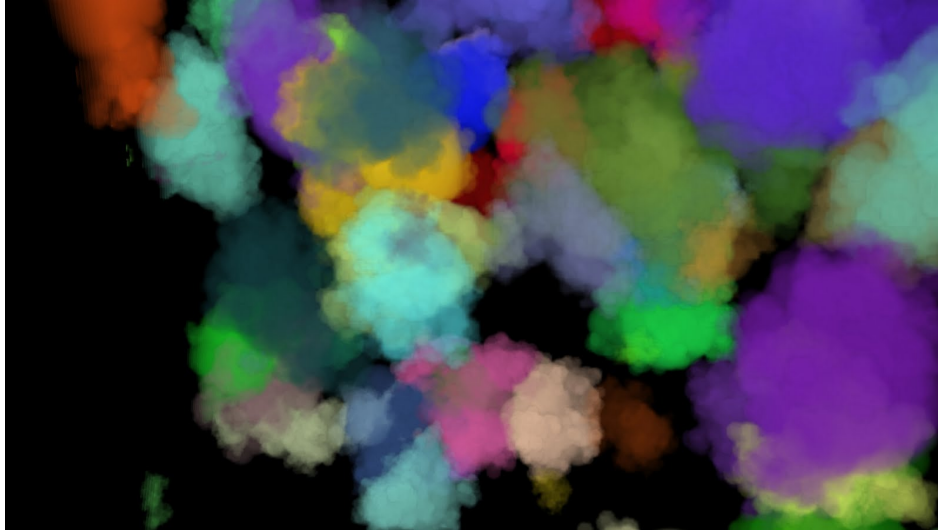


Figure B.1: A set of volume rendered clouds

B.2 Background

Sparse Grids

Sparse grids are not appropriate for all situations. If there are no memory management concerns, or if the importance of shorter render times outweighs decreasing the size of the data, then sparse grids will not likely be of much use. Even in situations where using less memory is an important goal, sparse grids may not be a useful solution. There is a threshold of data a sparse grid can hold beyond which the sparse grid is less memory efficient. This threshold varies depending on the implementation. Since sparse grids store memory in a noncontinuous fashion, extra information must be stored to retrieve the correct data. This information will end up being more of a burden on memory if too much of the grid is allocated. Sparse grids are useful for a specific class of problems. If the application contains a grid that is mostly empty and for which better memory usage justifies slower rendering speeds, sparse grids are appropriate. The added computation in looking up values in a sparse grid is what causes the slowdown in render time.

OpenCL

Specifically, we are interested in building and analyzing the performance of sparse grid implementations in OpenCL. OpenCL is a programming framework designed to take advantage of the heterogeneous nature of modern computer systems ¹. One of the major benefits of OpenCL is the ability to execute code on graphical processing units (GPU). GPUs are highly parallel, with many threads of execution. This inherent parallelism combined with the fast floating point processing cores used in GPUs makes these devices very appealing for solving parallelizable problems (like volume rendering). OpenCL seeks to take advantage of the attributes of the GPU. Programs written for OpenCL must have a host and at least one kernel written in order to function. The host program, written in a language like C or C++, is tasked with setting up the kernel and handling interactions between the kernel and the rest of the system.

Common tasks performed by the kernel include discovering what OpenCL devices are in the system, loading a kernel, choosing the appropriate OpenCL device for the kernel to run on, and loading data onto the OpenCL device. OpenCL devices are usually CPUs and GPUs, and the host program can specify what type of OpenCL device a kernel is to run on. Kernels are written in OpenCL C, a C like language with some alterations. The kernels written for the volume renderers used in this paper utilize the data parallel programming model. In other words, a set of input data is split up and a sequence of operations is performed on each element of the input data in separate threads of execution. In this case, kernels contain the sequence of operations. This same kernel is run concurrently on many threads.

Each data element is given a unique global id, and using this id, each thread spawned is given a different element's id. The host program specifies the number of threads to spawn, which is often the number of data elements to be processed. One of the important factors in OpenCL is device memory. Since the CPU the host is executing on and the device executing the kernel are almost always different, data the kernel needs must be loaded into the device's global memory. For most systems, this device memory is smaller than main memory, especially if the device in question is a GPU. Thus, efficient memory management is even more important when using OpenCL. OpenCL is an attractive language for volume renderers because of the speedup provided by GPU concurrency, but memory limitations are a serious concern. Sparse grids are a valuable extension that can potentially greatly expand the size of renderable grids.

Related Work

Several popular libraries already exist that utilize sparse grids. Field3D is an open source C++ library originally developed by Sony Pictures Imageworks 2. While Field3D uses dense grids by default (which are generally much faster but bigger), the library does have sparse grid options available. Field3D uses a scheme similar to the Block-partition sparse grid implementation described below. In a nutshell, Field3D divides the grid into multi-voxel blocks. These blocks are only allocated when a voxel within a given block is set. Field3D has some additional functionality not found in our implementations, such as deallocating blocks and iterating over blocks. This functionality was not necessary for our purposes and thus was not added.

Gigavoxel is a system developed by Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, and Elmar Eisemann 3. It uses N^3 -trees to store and access voxel data. Each node of an N^3 -tree can be subdivided into N^3 children. N^3 -trees are a more generalized form of octrees, which are N^3 trees with $N = 2$. Different values of N can be used to achieve different performance goals. If N is small, the data structure will be more memory efficient. If N is large, the data structure will allow for quicker traversal. Each node in Gigavoxel's N^3 -tree stores either a block of pointers or a single value. The single value represents a value that is shared by all elements in the particular N^3 sized block. Usually this will be zero, representing an empty block. All node data and blocks of voxels are stored in texture memory. Gigavoxel is capable of rendering 2048^3 sized RGBA data in real time on GPUs.

B.3 C++/C implementations

Before attempting to build an OpenCL volume renderer, a number of implementations of sparse grids were created. These implementations were analyzed and compared based on memory usage, lookup time, and load time. The grids described below were implemented as C++ classes with a common API. The one C struct implemented was written to match this API as closely as possible. Several variations are made where appropriate, but the important methods are :

`get(int,int,int)` - given an index for each dimension, return the value at grid point
`set(float,int,int,int)` - given an index for each dimension, set the grid point to the given float value
`init(int,int,int)` - given dimensions for x, y, and z components, initialize the grid
 In addition, all sparse implementations have an additional method
`setDefVal(float)` - sets the default value for the grid

Non Sparse Implementation

As a baseline for comparison, a dense grid class was implemented. Given X,Y, and Z dimensions for the grid, a 1D float array of size $X * Y * Z$ is dynamically allocated. Assuming an indexing scheme from 0 to N-1, grid point (i,j,k) would be accessed by indexing into the array

$$index = i + j * X + k * X * Y \tag{B.1}$$

This value is used both when getting and setting this grid point. Since there is very little computation and the memory is all allocated beforehand, both operations are performed very quickly.

C++ STL Maps

Two implementation of sparse grids were created using the Map class from C++'s Standard Template Library (STL). Abstractly, maps are a set of (key,value) pairs. A map can store a value with a unique key. If given a potential key, a map will either indicate that the key does not exist or return the corresponding value. The C++ STL map class is templated; a type must be specified for both the key and the value. In both implementations described below, the value type is float, while the key type varies based on implementation. Note that the map class is not currently portable to OpenCL; these two implementations served as a baseline for completely sparse grids.

The first implementation has a key,value type of $\langle int, float \rangle$; given an int key, there would potentially be a float value. This associative storage of data is very similar to the general idea of sparse grids and a sparse grid can easily be implemented using an $\langle int, float \rangle$ map. First, a default float value must be defined. Though commonly 0.0 (grid points with no data would have no density), the value is application specific. The grid must then be initialized with the X, Y, and Z dimensions of the grid. Though no memory is allocated, the grid dimensions must still be known for correct getting and setting. When a value F is to be set at grid point (i,j,k), the key value pair to be inserted into the map will be $(i + j * X + k * X * Y, F)$. This insertion is only performed if $F \neq defVal$. In this way, only relevant data is stored in the grid. When getting a value from position (i,j,k) on the grid, if the map does not contain key $i + j * X + k * X * Y$, then the default value is returned. If $i + j * X + k * X * Y$ does have an associated value, it is returned. Whereas the dense grid had O(k) set and get methods, the complexity of the set and get methods for the $\text{map} \langle int, float \rangle$ sparse grid is O(log n), where n is the number of elements in the map.

The second implementation of sparse grids using STL maps has a key,value pair type of $\langle int, \text{map} \langle int, \text{map} \langle int, float \rangle \rangle \rangle$. Each of the three ints in the key,value pair type correspond to a dimension of the grid. The get method works by checking for (i,j,k) if any value has been stored with an X index of i, if so whether any value has also been stored with Y value of j, and so on. Setting works in a similar though reverse way. The benefits of this implementation may not be initially obvious behind the layers of abstraction. The main benefit of this implementation is that there is no need for predefined grid dimensions. This allows for practically infinite sized grids, as long as the indices are valid ints and there is available memory. Grid data can be positioned anywhere without having to specify an offset as in other grid implementations. The benefits

described above also come at a large cost. The set and get methods of the `map< int, map< int, map< int, float >>>` sparse grid are $O((\log n)^3)$.

Red-Black Tree

Since the C++ STL map is not something that can be used in an OpenCL kernel, one of the potential avenues pursued by the authors was creating a sparse grid with an underlying data structure that would be compatible with OpenCL and also improve upon either the render time or the memory overhead. It was decided that the underlying data structure for storing non-empty grid points would be a binary search tree. The key for organizing the nodes of the tree is the $i + j * X + k * X * Y$, which is unique for each grid point. Binary search trees insert smaller keys into the left sub-tree and larger values into the right sub-tree. It is very possible that a user of the grid class would start at grid point (0,0,0) and loop over the acceptable values of i,j, and k to initialize their grid. This potentially leads to the worst case scenario for the tree, in which it essentially becomes a linked list of right sub-trees. Since this possibility could easily occur in use, it was decided to use a self balancing tree, specifically a red-black tree. While this may incur a hit in inserting new nodes, this should decrease the look up time, especially in the worst case. The red-black tree implementation of a sparse grid was written in C. Instead of a C++, a struct was used, and all methods had an additional `SparseGrid *` parameter. Notice that this implementation (as well as all sparse implementations) does introduce memory overhead for each non-empty value added to the grid. Pointers to the left and right subtrees and a key value must be stored for each float to be stored in the tree. This overhead means that for grids where greater than 25% of the grid points are non-empty, the sparse grid implementation is actually less memory efficient.

Block-Partition

The method for sparsely storing voxel data described here is similar to the scheme used by Field3D 2. This implementation partitions the grid in into cubes of one or more voxels. Often the grid is partitioned so that each block (the partition size) is a power of two in each dimension 4. This is implemented as an array of `float*` all originally set to null. When an element is to be set at a certain grid point, the correct block is found. If this block has not been allocated, the entire block containing the grid point will be allocated as an array of size $(partSize)^3$. The value is then set at the appropriate index in the appropriate block array. If the block has already been created, then the only step is having the correct element of the array is set. Looking up a value involves indexing to the correct block. If this block is null, the default value should be returned. If the block is allocated, the correct value within the block should be returned.

Unlike previous sparse grids implemented, the block-partition method does not attempt to only store filled voxels. Block-partition instead attempts to use the inherent spatial locality of particles within most grids. In the entities we are often trying to render, like clouds, if a specific voxel is filled, it is very likely that most if not all of its adjacent voxels will also be filled. This property does not hold along the outer edges of the entity, and it is possible there are non filled areas within the entity, but for the majority of particles spatial locality should apply. This coupled with the expensive cost of dynamically allocating space makes building grids using this algorithm generally faster than other implementations. Implementations like those using the STL maps and red-black trees carry a high cost in terms of the amount of data necessary to get to organize the grid (ex: the pointers for the left and sub-trees). In block-partition method, only one extra value is created for every $partSize^3$ voxels. While not all of these voxels may be used, the majority of them will likely be used for most volumes. It is not unlikely that for many sets of data the block-partition method may be more memory efficient than other sparse implementations discussed. Another very tangible benefit of this scheme is its handling of dynamically allocating memory. The first grid point to be set in a block

should be about as slow as other implementations since the block must be allocated before the value can be set. However, later values placed into that block will be inserted almost as quickly as in the dense grid (there is some extra computation involved in finding the correct block), since the space has already been allocated. This is one of the ways that this implementation takes advantage of the spatial locality of most data. The computational complexity of setting and getting values in partition-block grids is $O(k)$. This is the same order of complexity as dense grids. The multiplicative constant for dense grids is lower, so sets and gets on a dense grid will still run faster than the block-partition grid.

Comparison

With five different implementations of grids written, we were interested in seeing how these grids would work in use. The two important metrics of sparse grids are time and memory usage. Multiple grid sizes were used in testing; the tests were repeated with both a 256^3 grid and a 512^3 resolution grid of a wisp. The grids contained only floats, no color element was taken into account. The rendered 512^3 grid used for testing.

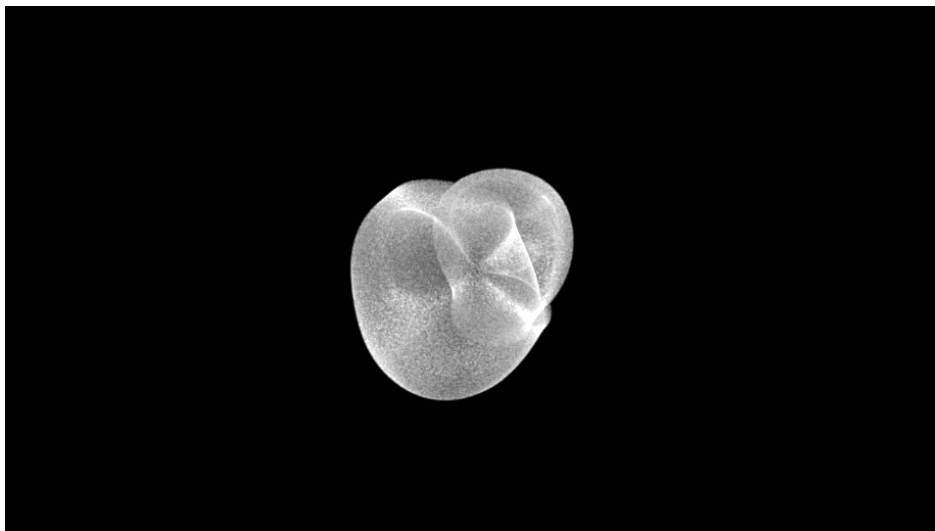


Figure B.2: The rendered 512^3 grid used for testing

To get a sense of how long each grid takes to set and access data, the time it took for each implementation to build a grid and iteratively get every point in the grid was measured. Loading a grid is a test of the speed of each grid's set method. Iterating over a grid in turn gives an idea of the relative time it takes each grid to look a value up. This test was repeated fifteen times, one immediately after the other. The results of the first five runs were discarded, since they almost always tended to be much less consistent than the latter runs. The load time and iteration time of the next ten runs were then averaged. The results of this testing were displayed below.

Several trends are worth discussing from this chart. Immediately apparent is the time inefficiency of the 3 nested STL maps implementation. This is not unexpected given its high computational complexity compared to the other grids, but its load and iteration times almost double that of the next slowest implementation. Unless the added benefit of not having to predefine maximum grid dimensions far outweigh the time costs displayed here, this implementation is not of much practical worth. The single STL map and the red-black

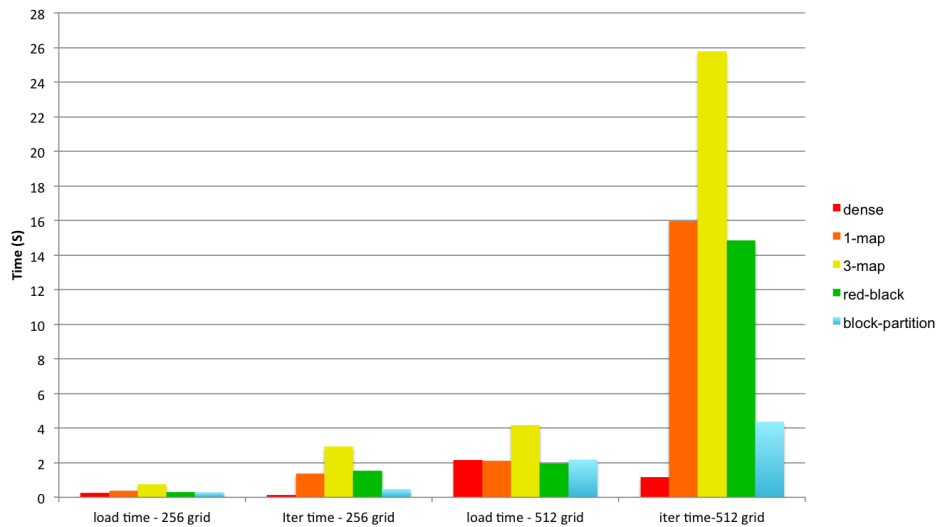


Figure B.3: CPU Grid Implementations - Time

implementation achieved similar results, which is not surprising. Maps like the ones found in the STL can be implemented using tree structures, and the two implementations were both $O(\log n)$ for setting and getting voxel data. The red-black tree performed slightly better, but this could potentially be attributed to a number of things like differences between the gcc and g++ compilers, extra overhead by the templates used in STL maps, or that the red-black implementation was written specifically for grids. This chart also makes it clear that sparse grids take the bigger time hit in accessing data. The sparse grids (nested map excluded) all performed about as well as the dense grid when it came to loading data. The dense grid iterates over both lists faster than it loads them. The opposite is true for the sparse grid implementations. The block-partition grid, which is clearly the most efficient sparse grid from a time standpoint, is twice as slow in iterating over its data as loading it. This means that getting a value is about four times slower for block-partition grids than for a dense grid.

Time is only one metric to analyze sparse grids. The other important metric is memory usage. To track memory usage, a 256^3 and a 512^3 grid were again loaded into memory. Valgrind, a tool that tracks memory allocation of programs, was used to see how much memory was being allocated in the process of loading each grid. The results are displayed below.

It is obvious that all of the sparse grids are much more memory efficient than the dense grid at storing data. Interestingly, the most efficient sparse grid (for these two images, which are representative of the majority of images rendered) is the block-partition grid, which does not attempt to contain only non-empty values. This is likely the case because the overhead incurred by the STL map and red-black tree implementations place a far greater burden on memory than the array of pointers to blocks and the unused but still allocated voxels.

Based on the tests we ran, the partition-block scheme of sparse grids is the clear winner. It is significantly faster and more memory efficient than any of its sparse counterparts. It is also clearly the most scalable, which is an important attribute for these grids considering that their purpose is in rendering larger grids of data. It is for these reasons that the sparse grids used in our volume renderers use block-partition grids.

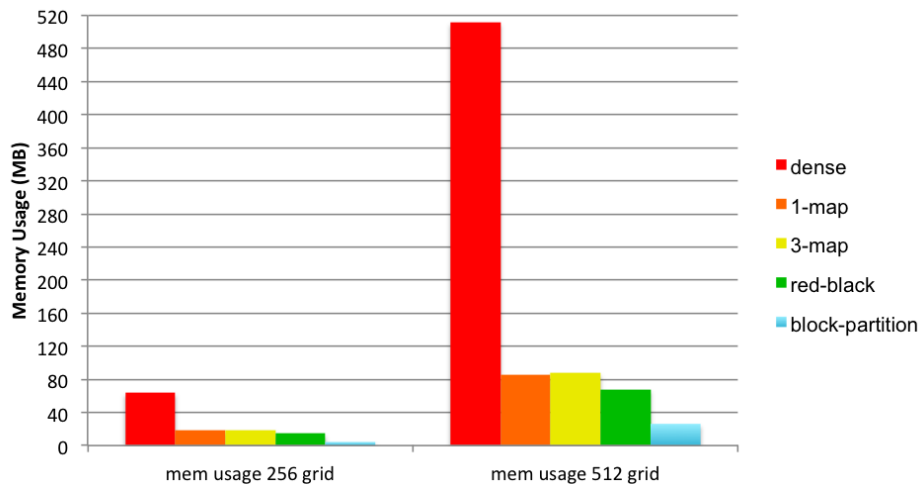


Figure B.4: CPU Grid Implementations - Memory

B.4 OpenCL implementations

Non-sparse

To serve as a foundation for building volume renderers utilizing sparse grids, we started with volume renderer with a C++ host that set up the volume to be rendered and the appropriate OpenCL setup. The renderer used two OpenCL kernels. The first kernel calculates and builds a deep shadow map for the volume. The second kernel performs the ray marching [5] and creates the image data, which is then returned to the host and written to an image file in the requested format. The shadow map kernel was removed during development to allow for more space in GPU memory.

Single Level of Abstraction

The original sparse grid implementation of a volume renderer in OpenCL features a system based on the partition-block implementation described above. The grid is built in the C++ code and loaded onto the GPU to be used by the kernel. Due to the nature of memory on a GPU, one major deviation is made from the C++ implementation. The C++ implementation kept an array of pointers to blocks of grid points, initially set to null. When a grid point was being set within a null block, space for the block would be dynamically allocated and the array of pointers would then point to their respective blocks. The problem with this implementation is that when the array is loaded into the graphics card's memory, the values of those array indexes are pointing to memory locations in a totally different address space, in main memory. To fix this problem, the initial array of pointers is replaced with an int array of the same size. Initially all values in the array are set to -1. Using the same indexing scheme described to index into the array of pointers with a given (i,j,k) , whenever a location is accessed for the first time, its value is changed from -1 to one less than the total number of locations indexed to so far. The first location accessed will be given a value of 0, the second location accessed will be given a value of 1, etc. These new values will be used to index into a second array of floats containing the actual values. Each time a new block is indexed, the float array will have to

be reallocated with `partSize3` more floats to accommodate a new partition. All of the newly allocated floats must be initialized to the default value.

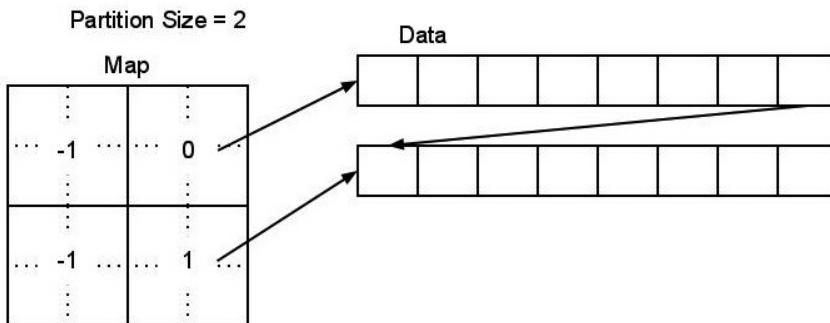


Figure B.5: A $4 \times 4 \times 2$ grid with a partition size of 2

Once both arrays are built in this fashion, getting a value from the grid is a two step process. The first step involves looking up the indexed int array value. If this is -1, then the default value is immediately returned. Otherwise, the array value is used to index into the correct partition in the float array, and then the correct value within the partition is returned. This difference allows the sparse grid to be loaded and correctly work on the GPU. The sparse grid was updated so that OpenCL's float4 type could also be stored at each data point. A separate array of float4s was kept and values were added and gotten identically to the float array. In the volume renderer the first three values of the float4 stored RGB color information. The original volume renderer using dense grids was heavily modified to accommodate the sparse grid implementation. Aside from trivial API differences between the grid classes on the C++ side, the majority of the changes to the C++ code involved getting all of the extra data needed by the sparse grid loaded onto the GPU. The sparse grid needs the mapping int array, the partition size, and the default values for the float and float4 arrays in addition to what is required of the dense grid. The kernels were altered to handle the additional arguments and the sparse data. The original dense grids relied on direct access to the arrays and were rewritten to correctly access the appropriate data.

Multiple Levels of Abstraction

Most of the overhead associated with the block-partition scheme hinges on the size of each partition. Larger partitions mean a smaller int array for mapping, but also means that more potentially unused grid points are being allocated. Conversely, depending on the size of the grid, it is possible that making the partition size will cause the total memory usage of the system to go up because of the increased size of the mapping array. In addition, it is possible that many of the integer blocks will be unused, containing a default value of -1. One way to alleviate this issue is to extend the idea of sparse grids so that the mapping array is itself sparsely stored into partitions. Adding an additional array of ints adds a layer of complexity both conceptually and computationally, but given a sparse enough grid the improvements in in memory usage are substantial. When setting a value at a grid point, the class must first check if the correct mapping partition has been allocated, allocate if necessary, and then do the same with the data partition before setting the value. Value look up also has the extra step of checking if the mapping partition has been allocated. The

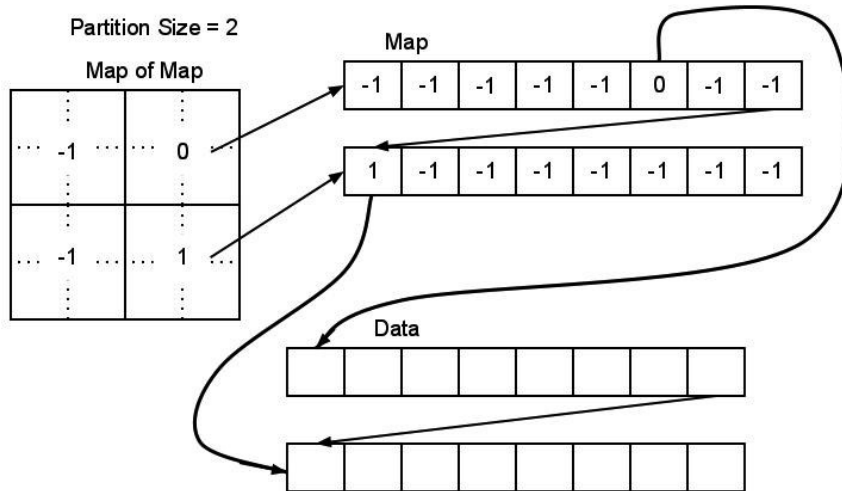


Figure B.6: An $8 \times 8 \times 4$ double sparse grid

doubly sparse grid implemented here has the same partition size for both levels, though a natural extension might be to have separate partition sizes for each level.

B.5 Results

Optimization

Several improvements were made to the two sparse renderers after achieving basic functionality. In the original version of the sparse renderer, only grids in which each dimension was a multiple of the partition size would correctly render. The double sparse renderer, in turn, would only correctly render with dimensions the multiple of the square of the partition size. Grids not fitting these requirements would produce images with noise around the edges. This noise was the result of the extra space created by dividing the grid into blocks; since the grid does not evenly divide into the blocks, a subset of the blocks will contain space for data that should not exist. This extra space would cause incorrect some indirect indexing into the sparse grid and thus distort the image. This was resolved by forcing non-multiple grid dimensions to the next highest multiple (a 101^3 grid will be forced to a 104^3 grid for single sparseness), ensuring the image produced will never have this issue.

Other changes were made to the renderers to improve performance in terms of both speed and memory usage. The deep shadow map kernel and its associated grid were removed from later versions of all three renderers. This change decreased both the render time of the image and the memory usage of the GPU since the shadow map data no longer had to be loaded on the card and there were fewer computations to perform in the ray march kernel. During the development and refinement of these renderers, we found that a partition size of four seems to produce optimal results for the test volume. The optimal partition size varies between volumes and the value that seemed the best compromise between render time and memory size.

Testing

Memory

In addition to creating an image, each renderer returns pertinent information like the partition size used, amount of time spent on the GPU, and the number of blocks allocated (for the sparse grids). From this data it is easy to calculate the amount of memory necessary to successfully create and store a grid. To compare the three renderers, the same image was rendered on each renderer. If a render was successful, the grid size was increased in each dimension by 250 voxels. When a grid gets too large for the GPU to store, an allocation failure flag is thrown and the render terminates. The results of testing are shown below. The results discussed below were generated by an Nvidia GeForce 9600M GT with 256 MB of memory. Additional tests were run on WHATEVER CHORTLES CARD IS and comparisons between the gathered data can be found later in the text.

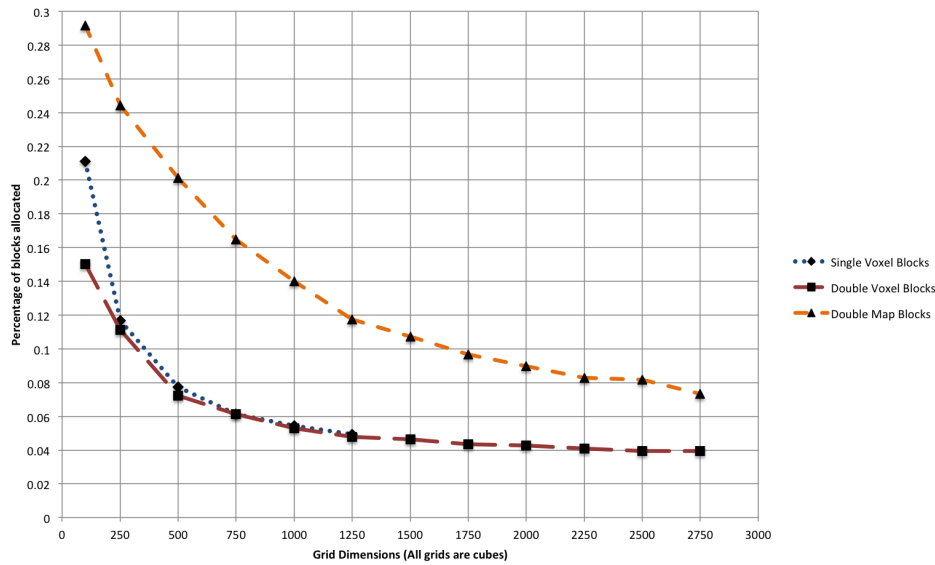


Figure B.7: Voxel Allocation in Sparse Renderers

The non-sparse implementation was only able to successfully render a 100^3 grid before exhausting the memory. The renderer using single sparse grid was able to successfully render grids up to 1250^3 , over three thousand times larger than the non-sparse renderer. The double sparse grid implementation achieved 2750^3 , six times larger than the single sparse renderer and over twenty thousand times larger than the non-sparse renderer. The sparse grid renderers are able to hold much greater volumes because they only allocate a very minute percentage of the voxels in the grid ($< 0.1\%$ in most cases). As a result, grids can be orders of magnitude larger. There is also a substantial gain in the additional layer of sparseness added for the double sparse renderer. Notice that the percentage of voxels allocated is much closer between the two sparse renderers than between either sparse renderer and the non-sparse one. Often, they differ by only a few thousandths of a percent. This would seem to imply that the majority of the savings are coming from making the mapping sparse as well. As the number of allocated map blocks reported from the double sparse implementation indicates, the vast majority of blocks are empty in the single sparse implementation. Significant memory

savings are made by making the map sparse as well. These savings only increase with the size of the actual grid, since the number of allocated blocks in the double sparse implementation stays roughly the same across grid sizes.

Render Time

The significant gains made by the sparse grid implementations in terms of memory footprint do come at the cost of render time. The non-sparse renderer may store all of its values, but because it does so, looking up a density value is as simple as directly indexing into an array. The single sparse renderer must perform a set of operations to see if the desired voxel has been allocated and if so what its values are. Having to perform these additional operations means a slower render. The double sparse renderer must essentially perform this set of operations twice to look up a value, and as a result is noticeably slower than even the single sparse render.

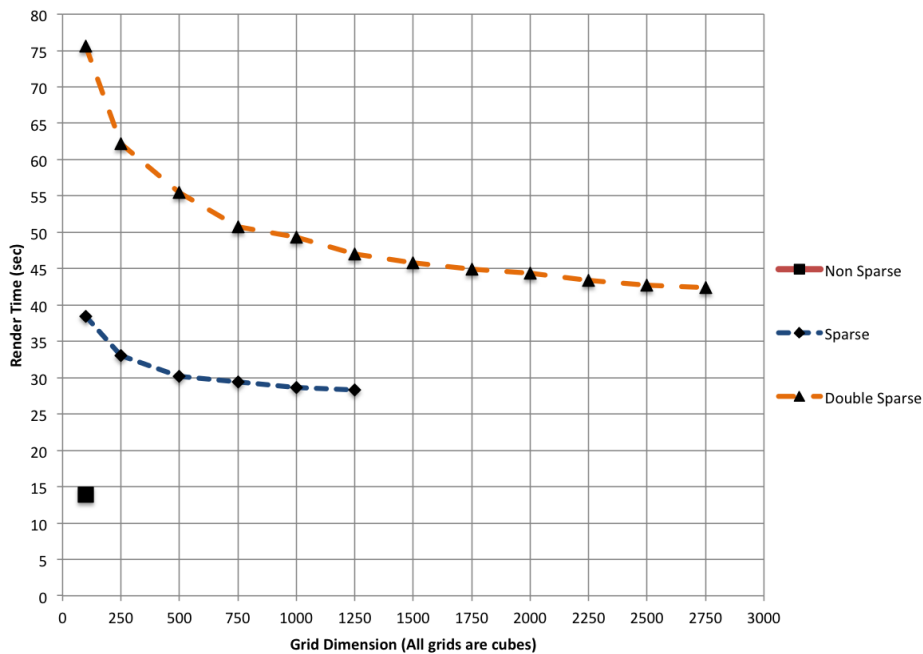


Figure B.8: Render Times Using Different Grid Sizes

Several trends, however, do appear in the data. Immediately clear is the fact that there is an inverse relation between grid size and render time for sparse grids. This relation is especially pronounced in the double sparse implementation results, which decrease by approximately 40% over the range of grid sizes. In smaller grids, render times are larger and there is a large change in render time between grid sizes. As grid sizes get larger, render time starts to decrease and there is less of a difference between the render time of different sized grids. These results may initially seem counterintuitive. Conventional wisdom would seem to imply that as grid size increased, so too should the time it takes to render the grid. Notice that render time trends are similar to the trend displayed percentage of map blocks allocated as grid sizes increase. Both start large with a fast rate of change and end with smaller values and a slow rate of change. A smaller percentage

of allocated blocks with increasing grid size implies that the rate of filled blocks in the grid is increasing at a slower rate the number of unfilled blocks. In other words, the majority of blocks added with an increase in grid size will go unallocated. While this is not necessarily desirable from a memory management perspective, it is actually a source of speed up when rendering a volume. If in the course of the look up function, the look up function finds that the desired voxel is not allocated, the look up function immediately returns the default value. This means that the majority of the computations needed to actually find the correct value are skipped, and the lookup function is much faster as a result. This speedup, combined with the parallel nature of graphics cards, leads to the shorter render times displayed in the double sparse implementation results. This same basic principle also applies to the single sparse renderer.

Card Scalability

CHORTLE RESULTS WOULD GO HERE

Several tests were run also run on WHATEVER CHORTLES CARD IS with MEM MB of memory. While the results clearly show that this is a more powerful card that can store much larger sparse grids, the trends remain largely the same. The benefits of these grids translate with more powerful cards.

Other

These results show that for many volume rendering applications in which memory is the limiting factor, sparse grids are valuable tools that can greatly increase the functionality of said application. If the volume needs to be rendered as fast as possible, or if the volume is mostly filled, sparse grids are probably not appropriate. Our results show that a single layer of sparseness allows grids with orders of magnitude more voxels for a relatively modest increase in render time. These memory savings come occur because only a minute percentage of voxels in the grid are actually allocated. More time is needed to render because additional computations are needed to correctly index to the allocated voxels. Eventually memory in a single level of sparseness the limiting factor becomes the size of the map used to index to allocated voxels rather than the voxels themselves. This limitation is solved by making the map itself sparse, creating a double level of sparseness. This double sparse renderer is slower than its less sparse counterpart, but also can render much larger volumes. With increasing grid size, these sparse renderers actually take less time to render. This is because the vast majority of voxels added by the increased grid size will be unallocated. Unallocated voxels can be looked up much faster than allocated voxels, which helps speed up the render. We recommend that a volume be rendered with the lowest degree of sparseness possible. Rendering an image using a sparse grid which can rendered with a non sparse renderer only increases the render time without real benefits. The value in sparse grids is in rendering grids so large they could not normally be rendered.

B.6 Conclusion

There are many possible methods of reducing the memory size of large volume grids. We implemented sparse grids using C++ STL maps, red-black trees, and block-partitions. Overall, block-partition sparse grids seem to be the most efficient method of storing volumetric data. Implementing block-partition grids in OpenCL allows large volume grids to be rendered on GPUs, which typically do not have as much RAM as CPUs. Using block-partition grids reduces memory usage but increases render time. Potential future work involves the use of texture memory on the GPU. Texture memory provides faster access time than global memory. To use texture memory for sparse grids, we envision writing the one-dimensional, sparse RGBA data to the

two-dimensional texture memory. We would choose an texture width that is some power of two in order to speed up indexing calculations. The logical extension of what has been presented in this paper would be to design a triply sparse grid. This could be useful for volumes that are extremely large but mostly empty. In general, however, there will be diminishing returns on adding extra levels of sparse mapping. Every extra layer of sparse mapping increases data access time significantly. Another potential improvement would be to find the optimal partition size or sizes computationally on the host before sending the data to the kernel. Generally, a partition size of around 4 is most memory efficient, but in some cases a larger partition size might save memory. We could develop an algorithm that analyzes that volume grid on the host and determines which partition size would result in the least possible memory usage. A drawback is that partition sizes that are powers of 2 make indexing calculations much faster. Using a partition size of 5 instead of 4 might save GPU memory, but it could double render time. Our current implementation coalesces a block into a single value only if every element in the block is exactly equal to one particular value. In other words, if the default value is zero, then a block of values will be allocated in the sparse grid if any of the values in the block are not exactly equal to zero. We could add a range of values that would be clamped to the default value. This would decrease memory usage but also decrease render quality.

1. Aaftab Munshi et al. "OpenCL Programming Guide", 1st ed. Upper Saddle River, NJ: Addison Wesley,2012
2. Magnus Wrenninge. (2011,11,15). Field3D: An Open Source File Format For Voxel Data[Online]. Available sites.google.com/site/field3d/
3. Cyril Crassin et al. "GigaVoxels: Ray-Guided Streaming for Efficient and Detailed Voxel Rendering" in ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D), Boston, MA, 2009 <http://maverick.inria.fr/Publications/2009/CNLE09>
4. Magnus Wrenninge et al. "Volumetric Methods in Visual Effects". SIGGRAPH 2010 Course Notes, pp. 6-70
5. Jerry Tessendorf and Michael Kowalski. "Resolution Independent Volumes". SIGGRAPH 2011 Course Notes

BIBLIOGRAPHY

- [1] Active implicit surface for animation.
- [2] Gaussian process implicit surfaces.
- [3] An implicit surface polygonizer.
- [4] Implicit surfaces.
- [5] Implicit surfaces slides.
- [6] Modeling with implicit surfaces that interpolate.
- [7] Morse theory for implicit surface modeling.
- [8] Ocean optics webbook: Scattering. petzold's measurements. http://www.oceanopticsbook.info/view/scattering/petzolds_measurements.
- [9] Ocean optics webbook: Scattering. the fournier-forand phase function. http://www.oceanopticsbook.info/view/scattering/the_fournierforand_phase_function.
- [10] On the velocity of an implicit surface.
- [11] Openvdb. <http://openvdb.org>.
- [12] Polygonalization of implicit surfaces.
- [13] Ridges and ravines on implicit surfaces.
- [14] Robust creation of implicit surfaces from polygonal meshes.
- [15] BLOOMENTHAL, J., Ed. *Introduction to Implicit Surfaces*. Morgan Kaufmann, 1997.
- [16] BRYCE, S., AND WELCH, Z. Sparse grid volume rendering on a gpu. Undergraduate honors thesis.
- [17] DAVID S. EBERT, F. KENTON MUSGRAVE, D. P. K. P. S. W. *Texturing & Modeling, A Procedural Approach*, 3rd ed. Morgan-Kaufmann, 2002.
- [18] KAJIYA, J. Kajiya paper on rendering equation.

- [19] LI, Q. Smooth piecewise polynomial blending operations for implicit shapes. *Computer Graphics Forum* 26, 157–171.
- [20] MATSUMOTO, M., AND NISHIMURA, T. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.* 8, 1 (Jan. 1998), 3–30.
- [21] NIELSON, M. B., AND MUSETH, K. Dynamic tubular grid: An efficient data structure and algorithms for high resolution level sets. *Journal of Scientific Computing* (2006).
- [22] SHU, Y. 3d fractal flame wisps. Master’s thesis, Clemson University, 2013.
- [23] TESSENDORF, J., AND KOWALSKI, M. Resolution independent volumes. Rhythm and Hues contribution to Siggraph Course “Production Volume Rendering”.
- [24] WILLIAMS, A., BARRUS, S., MORLEY, R. K., AND SHIRLEY, P. An efficient and robust ray-box intersection algorithm. In *ACM SIGGRAPH 2005 Courses* (New York, NY, USA, 2005), SIGGRAPH ’05, ACM.

INDEX

Closest Point Transform, 46, 61
composition, 9
constructive solid geometry, 8
CSG, 8
cumulo, 58

density, 58, 59
Dirac delta function, 8

Fast Marching Method, 46

level set, 46
Levi-Civita, 13

Marching Cubes, 46

pyroclastic, 58

Rendering-Equation, 23

Signed Distance Function, 45, 46, 58–61
signed distance function, 15

XXX, 58