**[Conway’s Game of Life for Curved Surfaces (Part 2)](https://0fps.net/2012/11/28/conways-game-of-life-for-curved-surfaces-part-2/)**

Posted on [November 28, 2012](https://0fps.net/2012/11/28/conways-game-of-life-for-curved-surfaces-part-2/)by [mikolalysenko](https://0fps.net/author/mikolalysenko/" \o "View all posts by mikolalysenko)

(This is the sequel to the following post on [SmoothLife](https://0fps.wordpress.com/2012/11/19/conways-game-of-life-for-curved-surfaces-part-1/).  For background information go there, or read [Stephan Rafler’s paper on SmoothLife here](http://arxiv.org/abs/1111.1567).)

Last time, we talked about an interesting generalization of Conway’s Game of Life and walked through the details of how it was derived, and investigated some strategies for discretizing it.  Today, let’s go even further and finally come to the subject discussed in the title: Conway’s Game of Life for curved surfaces!

Quick Review: SmoothLife

To understand how this works, let’s first review Rafler’s equations for SmoothLife.  Recall that f : \mathbb{R}^n \to \mathbb{R} is the state field and that we have two effective fields which are computed from *f*:

M(x,t) = \frac{1}{2 \pi h^2} \iint \limits_{|r| \leq h} f(x-r,t) dr

N(x,t) = \frac{1}{8 \pi h^2} \iint \limits_{h \leq |r| \leq 3h} f(x-r,t) dr

And that the next state of f is computed via the update rule:

f(x,t+1) = S(N(x,t), M(x,t))

Where:

\sigma(x, a, \alpha) = \frac{1}{1 + \exp(-\frac{4}{\alpha}(x - a))}

\sigma_n(n, a, b) = \sigma(n, a, \alpha_n) (1 - \sigma(n,b, \alpha_n)

\sigma_m(m, a, b) = (1 - \sigma(m, 0.5, \alpha_m) )a + \sigma(m, 0.5, \alpha_m) b

S(n,m)=\sigma_n(n, \sigma_m(m, b_1, d_1), \sigma_m(m, b_2, d_2))

And we have 6 (or maybe 7, depending on how you count) parameters that determine the behavior of the automata:

* [b_1, b_2]: The fraction of living neighbors required for a cell to stay alive (typically [\frac{2}{8}, \frac{3}{8}]).
* [d_1, d_2]: The fraction of living neighbors required for a cell to be born (typically \frac{3}{8}).
* \alpha_m:  The transition smoothness from live to dead (arbitrary, but Rafler uses \alpha_m \approx 0.148).
* \alpha_n: Transition smoothness from interval boundary (again, arbitrary but usually about \alpha_n \approx 0.028).
* h: The size of the effective neighborhood (this is a simulation dependent scale parameter, and should not effect the asymptotic behavior of the system).

Non-Euclidean Geometry

Looking at the above equations, the only place where geometry comes into the picture is in the computation of the *M* and *N* fields.  So it seems reasonable that if we could define the neighborhoods in some more generic way, then we could obtain a generalization of the above equations.  Indeed, a similar idea was proposed in Rafler’s original work to extend SmoothLife to spherical domains; but why should we stop there?

Geodesic Distance

The basic idea behind all of this is that we want to generalize the concept of a sphere to include curved surfaces.  Recall the usual definition of a sphere is that it is the set of all points within some distance of a given point.  To extend this concept to surfaces, we merely have to change our definition of distance somewhat.  This proceeds in two steps.

First, think about how the distance between two points is defined.  In ordinary Euclidean geometry, it is defined using the Pythagorean theorem.  That is if we fix a Cartesian coordinate system the distance between a pair of points p=(x_0, y_0), q=(x_1, y_1) is just:

d(p,q) = |p-q| =\sqrt{ (x_0-x_1)^2 + (y_0-y_1)^2 }

But this isn’t the only way we could define this distance.  While the above formula is pretty easy to calculate, it by far not the only way such a distance could be defined.  Another method is that we can formulate the path length [variationally](http://en.wikipedia.org/wiki/Calculus_of_variations).  That is we describe the distance between points *p* and *q* as a type of optimization problem; that is it is the [arc length](http://en.wikipedia.org/wiki/Arc_length) of the shortest path connecting the two points:

d(p,q) = \min \limits_{f : [0,1] \to \mathbb{R}^2\\f(0)=p, f(1)=q} \int \limits_0^1 |\partial_t f(t)| dt

At a glance this second statement may seem a bit silly.  After all, solving an infinite dimensional optimization problem is much harder than just subtracting and squaring a few numbers.  There is also something about the defining Euclidean distance in terms of arclength that seems viciously circular — since the definition of arclength is basically an infinitesimal version of the above equation — but please try not to worry about that too much:

Computing arclength by subdivisions. Image (c) Wikipedia 2007. Author: Mike Peel

Instead, the main advantage of working in the variational formulation is that it becomes possible to define distances in spaces where the shortest path between two points is no longer a straight line, as is the case on the surface of the Earth for example:

[](https://0fps.files.wordpress.com/2012/11/great-circle.png)

The shortest distance between any two cities on Earth is a circle — not a straight line!

Of course to define the arclength for something like a curve along a sphere, we need a little extra information.  Looking at the definition of arclength, the missing ingredient is that we need some way to measure the length of a tangent vector.  The most common way to do this is to introduce what is known as a [Riemannian metric](http://en.wikipedia.org/wiki/Riemannian_manifold#Riemannian_metrics), and the data it stores is precisely that!  Given a smooth manifold \Omega, a Riemannian metric continuously assigns to every point p \in \Omega a [symmetric](http://en.wikipedia.org/wiki/Symmetric_bilinear_form) [bilinear form](http://en.wikipedia.org/wiki/Bilinear_form) g_p : T_p \Omega \times T_p \Omega \to \mathbb{R} on the [tangent space](http://en.wikipedia.org/wiki/Tangent_space) of \Omega at *p*.  To see how this works let us, suppose we were given some curve f : [0,1] \to M, then we can just define the arclength of *f* to be:

\mathrm{arclength}(f) = \int_0^1 \sqrt{g_{f(t)} \left( \frac{d f}{dt}, \frac{d f}{dt} \right)} dt

Armed with this new generalization arclength, it is now pretty clear how we should define the distance between points on a Riemannian manifold (that is any differentiable manifold equipped with a Riemannian metric).  Namely, it is:

This is called the [geodesic distance](http://en.wikipedia.org/wiki/Geodesic) between *p* and *q*, after the concept of a geodesic which is a fancy name for `the shortest path between two points’.  The fact that concatenation of paths is associative implies that geodesic distance satisfies the [triangle inequality](http://en.wikipedia.org/wiki/Triangle_inequality), and so it is technically a [metric](http://en.wikipedia.org/wiki/Metric_(mathematics)).  The fact that our metric is allowed to vary from point to point allows us to handle much more flexible topologies and and surfaces.  For example, [parallel or offset curves](http://en.wikipedia.org/wiki/Parallel_curve) on a Riemannian manifold may not be straight lines.  This leads to violations of Euclid’s parallel postulate and so the study of Riemannian manifolds is sometimes called [non-Euclidean geometry](http://en.wikipedia.org/wiki/Non-Euclidean_geometry).  (It also has a few [famous applications in physics](http://en.wikipedia.org/wiki/General_relativity)!)

Geodesic Balls

Now that we have a definition of distance that works for any surface (with a Riemannian metric), let’s apply it to Rafler’s equations.  I claim that all we need to do is replace the spherical neighborhoods in SmoothLife with some suitably constructed geodesic neighborhoods.  In particular, we can define these neighborhoods to be balls of some appropriate radius.  Given a point *p*, we define the geodesic ball of radius *h* centered at *p* to be the set of all points within distance at most *h* of *p*:

B_{p,h} = \{ x \in \Omega : d(p,x) \leq h \}

Which naturally leads to defining the *M* and *N* fields as follows:

M(x,t) = \frac{1}{\mu(B_{x,h})} \int_{y \in B_{x,h}} f(y,t) dy

N(x,t) = \frac{1}{\mu(B_{x,3h}) - \mu(B_{x,h})}\int_{y \in B_{y, 3h} \setminus B_{y,h}} f(y,t) dy

That these equations constitute a generalization of SmoothLife to curved surfaces should be pretty clear (since we barely changed anything at all!).  One can recover the original SmoothLife equations by setting \Omega = \mathbb{R}^2 and taking a flat metric.  However, there is a major departure in that the very same equations can now be used to describe the dynamics of life on curved surfaces as well!

Discrete Riemannian Manifolds

Generalizing SmoothLife to surfaces didn’t change the equations much, but conceptually there is now a whole lot more to unpack.  To actually implement this stuff, we need to solve the following three problems:

1. First, we need to figure out what metric to use.
2. Then, we need some way to compute geodesic balls on a surface.
3. Finally, we have to discretize the fields on the surface in some way.

The last part follows along more or less the same sort of reasoning as we applied in the periodic/flat case, (though the details are a bit different).  However,  items 1 and 2 are new to SmoothLife on curved surfaces and requires some special explanation.

Metric Embeddings

Up to this point, we’ve been deliberately abstract in our dealings with metrics, and haven’t really said much about where they come from.  In the most general setting, we can think of a metric as something which is intrinsic to a surface — ie it is part of the very data which describes it.  However, in practical applications — especially those arising in computer graphics and engineering — metrics usually come from an [embedding](http://en.wikipedia.org/wiki/Embedding#Riemannian_geometry).  Here’s how it works:

Given a pair of differential manifolds M, N and a smooth map f : M \to N with a metric g on N, then f induces a metric on M by a [pullback](http://en.wikipedia.org/wiki/Pullback_(differential_geometry)).  That is, df be the differential of f, then there ia metric g^* on M for every point p \in M:

g^*_p( u, v ) = g_{f(p)} ( df_p(u), df_p(v) )

To make this very concrete, let us suppose that our surface is sitting inside 3D Euclidean space.  That is we have some smooth map h : \Omega \to \mathbb{R}^3 that takes our surface and sticks it into 3-space with the ordinary flat Euclidean metric.  Then the geodesic distance is just the length of the shortest path in the surface:

d(p,q) = \min_{f : [0,1] \to \Omega \\ f(0)=p, f(1)=q} \int_0^1 | \frac{d h(f(t))}{dt} | dt

In the case of a triangulated mesh, we can let the embedding map just be the ordinary piecewise linear embedding of each triangle into 3-space.

Geodesics on Triangulated Surfaces

Ok, so now that we have a Riemannian metric, the next thing we need to is figure out how to calculate the geodesic distance.  As far as I am aware, the first comprehensive solution to this problem was given by Mitchell, Mount and Papadimitrou back in 1987:

J. Mitchell, D. Mount and C. Papadimitrou. (1987) “[The Discrete Geodesic Problem](http://www.cs.umd.edu/~mount/Papers/mmp-sicomp-87.pdf)” SIAM Journal of Computing

The basic idea behind their algorithm is pretty straightforward, though the details get a little hairy.  It basically amounts to a modified version of Dijkstra’s algorithm that computes the single source shortest path to every point along each edge in the mesh.  The basic reason this works is that within the interior of each face the actual distance field (for any point source) is always piecewise quadratic.  However, there is a catch: the distance along each edge in the mesh might not be quadratic.  Instead you may have to subdivide each edge in order to get a correct description of the distance field.  Even worse, in that paper Mitchell et al. show that the exact distance field for a mesh with O(n) vertices could have O(n) distinct monotone components per edge, which gives their algorithm a worst case running time of O(n^2 \log(n)), (the extra log factor is due to the cost associated with maintaining an ordered list for visiting the edges.)

Quadratic time complexity is pretty bad — especially for large meshes — but it is really about the best you can hope for if you are trying to solve the problem properly.  In fact, a distance field on a piecewise linear triangular surface can have up to O(n^2) distinct piecewise polynomial components, and so you need O(n^2) bits just to encode the field anyway.  However, it has been observed that in practice the situation is usually not so bad and that it is often possible to solve the problem faster if you are willing to make do with an approximation. Over the years, many alternatives and approximations to geodesic distance have been proposed, each making various trade offs.  Here is a quick sampling of a few I found interesting:

K. Polthier and M. Schmies. (1998) “[Straightest Edge Geodesic](http://page.mi.fu-berlin.de/polthier/articles/straightest/straightest_preprint.pdf)” Mathematical Visualization

R. Kimmel and J.A. Sethian. (1998) “[Computing Geodesic Paths on Manifolds](http://math.berkeley.edu/~sethian/2006/Papers/sethian.kimmel.geodesics.pdf)” PNAS

V. Surazhky, T. Surazhky, D. Kirsanov, S. Gortler, H. Hoppe. (2005) “[Fast Exact and Approximate Geodesics on Meshes](http://research.microsoft.com/en-us/um/people/hoppe/geodesics.pdf)” SIGGRAPH 2005

The first two papers are important in that they describe different ways to look at approximating geodesic distance.  Polthier and Schmies give an alternative definition of geodesic distance which is easier to calculate, while Kimmel and Sethian describe a method that approximates the surface itself.  Both methods run in O(n \log(n)) time instead of O(n^2) — though one needs to be a bit careful here, since the n in Kimmel and Sethian (aka number of voxels) could actually be exponentially larger than the n (number of vertices) which is used in the other papers (in practice this is usually not much of a problem).  The last paper is also important as it is one of the most cited in this area, and discusses a simple a modification of Mitchell et al.’s method which gives some ability to make a tradeoff between performance and accuracy.  I also like this paper for its clarity, since the figures are much easier to follow than the original 1987 work.

One conclusion that can be drawn from all this is that computing geodesic distance is a hairy problem.  Digging around it is easy to get lost in a black hole of different possibilities.  Since this is not the main point of this post (and because I wrote most of this code while I was on an airplane without access to the internet), I decided to try just hacking something together and ended up going with the first thing that worked.  My approach was to use a modification of the [Bellman-Ford algorithm](http://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm) and compute the geodesic distance in two phases.  First, I did an iteration of Dijkstra’s algorithm on the edges of the mesh to get an upper bound on the distance, using the Euclidean distance lower bound to trim out far away vertices.  Then I iterated unfolding the faces to try to refine the distance field until it converged.

This approach may be novel, but only because it is s obviously terrible compared to other techniques that no one would ever bother to publish it.  Nonetheless, in my experiments I found that it gave a passable approximation to the distance field, provided that the mesh had enough subdivisions.  The downside though is that the Bellman-Ford algorithm and its variants are quite slow. If I ever have the time, I’d like to revisit this problem some day and try out some better techniques and see how they compare, but that will have to be the subject of another entry.  For now, this approach will have to do.

The Finite Element Method

Supposing that we now have some reasonable approximation to geodesic distance (and thus the shape of a geodesic ball) on our mesh, we now need to come up with a way to represent an approximation of the state field on the mesh.  Just like in the flat version of SmoothLife, we must use the [Galerkin method](http://en.wikipedia.org/wiki/Galerkin_method) to discretize the system, but this time there is a major difference.  When we were on a flat/periodic domain, we had the possibility of representing the field by a sum of Dirichlet or sinc elements, which turned out to make it really easy to calculate the effective fields *M* and *N*using the Fourier transform.  However on an arbitrary curved surface (or mesh) we usually don’t have any sort of translational symmetry and so such a representation is not always possible.

Instead, we will have to make do with a somewhat more general, but less efficient, set of basis functions.  In engineering applications, one popular choice is to write the field as a sum of [splines](http://en.wikipedia.org/wiki/Spline_(mathematics)), (aka smooth compactly supported piecewise polynomial functions).  In fact, this idea of using the Galerkin method with splines is so widely used that it even has its own special name: [the Finite Element Method](http://en.wikipedia.org/wiki/Finite_element_method) (FEM).  FEM is basically the default choice for discretizing something when you don’t really know what else to do, since it works for pretty much any sort of topology or boundary condition.  The tradeoff is that it can be a lot less efficient than other more specialized bases, and that implementing it is usually much more complicated.

Piecewise Linear Elements

As a first attempt at creating a finite element discretization for SmoothLife, I opted to go with C^0 piecewise linear or `tent’ functions for my basis functions:

Illustration of a piecewise linear approximation by C^0 tent functions. Image (c) Wikipedia 2012. Author: Krishnavedala

For a 2D triangulated mesh, the simplest piecewise linear basis functions are the [Barycentric coordinates](http://en.wikipedia.org/wiki/Barycentric_coordinate_system_(mathematics)) for each face.  To explain how this works, let us first describe how we can use these functions to parameterize a single triangle.  In a Barycentric coordinate system, a triangle is described as a subset of \Delta \subset \mathbb{R}^3:

The three vertices of the triangle correspond to the coordinates (\alpha, \beta, \gamma) = (1, 0, 0), (0, 1, 0), (0, 0, 1) respectively.  Using this coordinate system it is really easy to define a linear scalar function on \Delta.  Pick any 3 weights w_0, w_1, w_2 \in \mathbb{R} and define a map f : \Delta \to \mathbb{R} according to the rule:

f(\alpha, \beta, \gamma) = w_0 \alpha + w_1 \beta + w_2 \gamma

This formula should be pretty familiar if you know a bit about computer graphics.  This is the same way a triangle gets mapped into 3D!  To get a map from the triangle to any vector space, you can just repeat this type of process for each component separately.

Ok, now that we have some idea of how an embedding works for a single triangle let’s figure out how it goes for a complete mesh.  Again we start by describing the parameter space for the mesh.  Let us suppose that our mesh is abstractly represented by a collection of n vertices, V and m faces, F where each face, (i_0, i_1, i_2) \in F is a 3-tuple of indices into V (in other words, it is an index array).   To each vertex we associate a scalar  \alpha_0, \alpha_1, ... \alpha_{n-1} \in [0,1] and for every face we add a constraint:

\alpha_{i_0} + \alpha_{i_1} + \alpha_{i_2} = 1

The solution set of all of these constraints cuts out a subset of K \subset \mathbb{R}^n which is the parameter space for our mesh.  To define a piecewise linear scalar field on K, we just need to pick n different weights, one per each vertex.  Then we write any piecewise linear scalar field, f : K \to \mathbb{R}, analgous to the triangular case:

f(\alpha_0, \alpha_1, ..., \alpha_{n-1}) = \sum_{j=0}^{n-1} w_{j} \alpha_j

In other words, the basis or `shape’ functions in our discretization are just the Barycentric coordinates \alpha_i. To see what this looks like visually, here is another helpful picture from wikipedia:

The 2D linear finite element shape functions visualized for a planar triangulation. The field is plotted above while the topology of the mesh is visualized on the bottom. Image (c) Wikipedia 2007. Author Oleg Alexandrov

Integration on Meshes

Now that we know our basis functions, we need to say a bit about how we integrate them.  To do this, let us suppose that we have a piecewise linear embedding of our mesh which is specified (again in the linear shape functions) by giving a collection of vectors v_0, v_1, ... v_{n-1} \in \mathbb{R}^3.  These vertex weights determine an embedding of our mesh in 3D space according to the rule \phi : K \to \mathbb{R}^3

\phi( \alpha_0, \alpha_1, ... \alpha_{n-1} ) = \sum_{i=0}^{n-1} \alpha_i v_i

To integrate a scalar function f : K \to \mathbb{R} over the embedded surface \phi(K), we want to compute:

\int_{\phi(K)} f( \phi^{-1}(x) ) d A

This seems a little scary, but we can simplify it a lot using the fact that \phi is piecewise linear over the mesh and instead performing the integration face-by-face.  [Integrating by substitution](http://en.wikipedia.org/wiki/Surface_integral), we get:

The way you should read that f(...) is that every term is 0 except for the i_0, i_1, i_2 components, which are set to \alpha_0, \alpha_1, 1-\alpha_0-\alpha_1 respectively.

Computing the Effective Fields

Now let’s suppose that we have a state field f and that we want to compute the effective fields $late M$ and N as defined above.  Then by definition:

M(x,t) = \frac{1}{\mu(B_{x,h})}\int \limits_{B_{x,h}} f(y) dy = \int_K 1_{B_{x,h}}(y) f(y) dy

The function 1_{B_{x,h}} is just the indicator function for the geodesic ball centered at x with radius h, and is computed using the geodesic distance field we described above.  To actually evaluate the integral, let us suppose that f(...) is a weighted sum of linear shape functions:

f(\alpha_0, ..., \alpha_{n-1}) = \sum_{i=0}^{n-1} w_i \alpha_i

Then we apply the expansion we just described in the previous section:

Where:

\Delta w_{ij} = w_i - w_j

\Delta v_{ij} = v_i-v_j

This integral is pretty horrible to compute analytically, but fortunately we don’t have to do that.  Instead, we can approximate the true result by using a [numerical quadrature rule](http://en.wikipedia.org/wiki/Gaussian_quadrature).  The way quadrature rules work is you just sample the integrand and take linear combinations of the results.  For cleverly selected weights and sample points, this can even turn out to be **exact** — though working out exactly the right values is something of a black art.  Fortunately, a detailed understanding of this process turns out to be quite unnecessary, since you can just look up the sample points and weights in a book and then plug in the numbers:

J. Burkardt. “[Quadrature Rules for Triangles](http://people.sc.fsu.edu/~jburkardt/datasets/quadrature_rules_tri/quadrature_rules_tri.html)“

It should be obvious that the same trick works for computing N, just by replacing 1_{B_{x,h}} with 1_{B_{x,3h}} - 1_{B_{x,h}}.

Matrix Formulation and Implementation

To compute the next state of f, we can just sample the field at each vertex.  To do this, let us define:

m_i = M( v_i, t)

n_i = N(v_i, t)

Then we can update f according to the rule:

w_i \leftarrow S(m_i, n_i)

While this would work, it would also be very slow, since computing m_i, n_i at each vertex requires doing quadratic work.  Instead, it is better if we precalculate a bit to avoid doing so many redundant computations.  To understand how this works, observe that our rule for computing the quantities m_i is linear in the vector w_i.  This means that we can write it as a matrix:

m = K_M w

Where m, w are the vectors of coefficients and K_M is the matrix which we precalculate whose entries using the formula:

That is, it is a sum over all the faces incident to vertex k (which I wrote as F_k to save space), with quadrature computed as described above.  Since the outer radius for the cells are finite, K_M ends up being pretty sparse and so it is relatively easy to process and store.  Again, the same idea applies to the effective neighborhoods which we store in the matrix K_N.  Putting this together, we get the following algorithm:

* Use geodesic distance to precompute matrices K_M, K_N
* Initialize state vector w
* For each time step:
  + Set m = K_M w
  + Set n = K_N w
  + Set w_i = S(m_i, n_i)

Demo

If you want to try playing around with this stuff yourself, I made a WebGL demo that should work in Chrome (though I haven’t tested it in Firefox).  You can check it out here:

<http://mikolalysenko.github.com/MeshLife>

Be warned though!  It is very slow due to the inefficient calculation of geodesic distances.  Of course once it has finished assembling the matrices K_M, K_N it runs pretty well in the browser.  If you want to look at the code for the demo, you can try it out on github:

<https://github.com/mikolalysenko/RiemannLife>

The project is written using node.js and has dependencies on two other libraries I made, namely [trimesh.js](https://github.com/mikolalysenko/TrimeshJS) and [meshdata](https://github.com/mikolalysenko/MeshData) — both of which you can install via npm.

Conclusions

In conclusion, we’ve demonstrated the extended the domain of Rafler’s SmoothLife equations to curved surfaces.  While our current implementation is somewhat crude, it does suffice to illustrate that the basic theoretical concepts are sound.  In the future, it would be good to investigate alternative methods for computing geodesic distance.  At the moment, this is the main bottleneck in the pipeline, and solving it would make further investigation of SmoothLife and related phenomena much easier.

It would also be interesting to attempt a more detailed analysis of the dynamics of SmoothLife.  One interesting puzzle would be to try working out the stability of various solitons/gliders in the presence of curvature.  It seems that they tend to destabilize and bifurcate near regions of high curvature.  This is probably due to the fact that the soliton interacts with itself in these regions.  For small curvatures, the trajectory of the solitons are deflected, causing gravitational like effects on their motion.