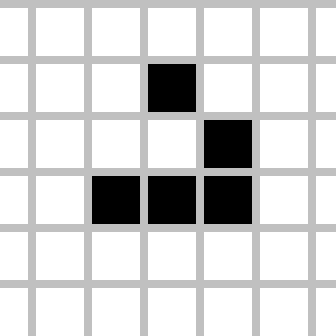
[**Conway’s Game of Life for Curved Surfaces (Part 1)**](https://0fps.net/2012/11/19/conways-game-of-life-for-curved-surfaces-part-1/)

Posted on [November 19, 2012](https://0fps.net/2012/11/19/conways-game-of-life-for-curved-surfaces-part-1/)by [mikolalysenko](https://0fps.net/author/mikolalysenko/)

Conway’s [Game of Life](http://en.wikipedia.org/wiki/Conway%27s_Game_of_Life) is one of the most popular and iconic [cellular automata](http://en.wikipedia.org/wiki/Cellular_automaton).  It is so famous that [googling it](https://www.google.com/search?q=conway%27s+game+of+life) loads up a working simulation right in your browser!  The rules for the Game of Life (GoL) can be stated in a few lines:

1. The world is an infinite rectangular array of cells, each of which can be in one of two states: alive or dead; and the neighborhood of each cell is a 3×3 grid centered on the cell.
2. The state of the world advances in discrete synchronous time steps according to the following two rules:
3. Cells which are alive remain alive if and only if they have between 2 and 3 neighbors; otherwise they die.
4. Cells which are dead become alive if they have exactly 3 neighbors; or else they stay dead.

Despite the superficial simplicity of these rules, the Game of Life can produce many unexpected and fascinating recurring patterns; like the glider soliton:

[](http://en.wikipedia.org/wiki/File:Game_of_life_animated_glider_2.gif)

A 3×3 glider in Conway’s Game of Life. Black represents a live cell; white represents dead. (c) 2008, Wikipedia. Author: Dhatfield

The fact that the rules for the Game of Life are so short and clear makes it a great project for initiating novice programmers.  Since Life was invented by [John Conway](http://en.wikipedia.org/wiki/John_Horton_Conway) in 1970, it has distracted many amateur and recreational mathematicians, leading to the discovery of many interesting patterns.  For example, [Paul Chapman](http://www.igblan.free-online.co.uk/igblan/ca/index.html) showed that Life is [Turing complete](http://en.wikipedia.org/wiki/Turing_completeness), and so it is in principle possible to translate any computer into a set of initial conditions.     As an amusing example of this concept, [here is an implementation of Conway’s Game of Life in the Game of Life](http://www.youtube.com/watch?v=xP5-iIeKXE8) created by [Brice Due](http://www.conwaylife.com/wiki/Brice_Due).  Another cool pattern is the famous [Gemini spaceship](http://www.conwaylife.com/wiki/index.php?title=Gemini) which is a complex self replicating system, and is perhaps the first true `organism’ to be discovered in life!  If any of this sounds interesting to you, you can play around with the Game of Life (and some other cellular automata) and look at a large and interesting library of patterns collected in the community developed [Golly software package](http://golly.sourceforge.net/).

SmoothLife

Now, if you’ve been programming for any reasonable length of time you’ve probably come across the Game of Life at least a couple of times before and so all the stuff above is nothing new.  However, a few weeks ago I saw an incredibly awesome video on reddit which inspired me to write this post:

This video is of a certain generalization of Conway’s Game of Life to smooth spaces proposed by Stephan Rafler:

S. Rafler, “[Generalization of Conway’s “Game of Life” to a continuous domain – SmoothLife](http://arxiv.org/abs/1111.1567)” (2011) Arxiv: 1111.1567

You can read this paper for yourself, but I’ll also summarize the key concepts here.  Basically, there are two main ideas in Rafler’s generalization:  First, the infinite grid of cells is replaced with an `effective grid’ that is obtained by averaging over a disk.  Second, the transition functions are replaced by a series of differential equations derived from a smooth interpolation of the rules for the discrete Game of Life.

Effective Grids

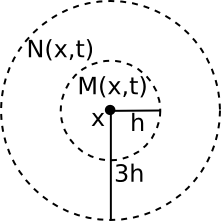
To explain how an `effective grid’ works, first consider what would happen if we replaced the infinite discrete grid in the game of life with a time-dependent continuous real scalar field, f : \mathbb{R}^2 \times \mathbb{R} \to \mathbb{R} on the plane.  Now here is the trick:  Instead of thinking of this is an infinite grid of infinitesimal cells, we give the cells a small — but finite! — length.  To do this, pick any small positive real number *h* which will act as the radius of a single cell (ie “the Planck length” for the simulation).  Then we define the state of the `*effective cell*‘ at a point *x* as the average of the field over a radius *h* neighborhood around x, (which we call M*(x,t)* following the conventions in Rafler’s paper):

M(x, t) = \frac{1}{\pi h^2} \int \limits_{0 \leq |y| \leq h} f(x-y,t) dy

Now for each cell, we also want to compute the effective “number of cells” in its neighborhood.  To do this, we use the same averaging process, only over a larger annulus centered at *x*. By analogy to the rules in the original GoL, a reasonable value for the outer radius is about 3*h.*Again, following Rafler’s conventions we call this quantity N*(x,t)*:

N(x, t) = \frac{1}{8 \pi h^2} \int \limits_{h \leq |y| \leq 3 h} f(x-y,t) dy

To illustrate how these neighborhoods fit together, I made a small figure:

[](https://0fps.files.wordpress.com/2012/11/neighborhoods1.png)

Now there are two things to notice about *M* and *N*:

1. They are invariant under continuous shifts.
2. They are invariant under rotations.

This means that if we define the state of our system in terms of these quantities, the resulting dynamics should also be invariant under rotations and shifts!  (Note:  These quantities are NOT invariant under scaling, since they are dependent on the choice of *h*.)

Smooth Transition Functions

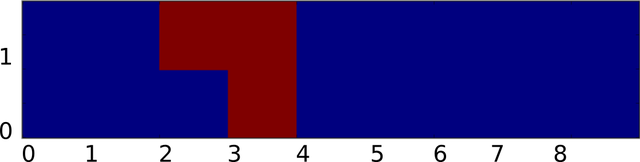
Of course getting rid of the grid is only the first step towards a smooth version of Life.  The second (and more difficult) part is that we also need to come up with a generalization of the rules for the game of life that works for continuous values.  There are of course many ways to do this, and if you’ve taken a course on real analysis you may [already have some idea on how to do this.](http://en.wikipedia.org/wiki/Mollifier)  However, for the sake of exposition, let’s try to follow along Rafler’s original derivation.

As a starting point, let’s first write the rules for the regular discrete Game of Life in a different way.  By some abuse of notation, suppose that our field *f* was a discrete grid (ie f: \mathbb{Z}^3 \to \{ 0, 1 \}) and that *N, M,*give the state of each cell and the number of live cells in the 1-radius [Moore neighborhood](http://en.wikipedia.org/wiki/Moore_neighborhood).  Then we could describe the rules of Conway’s Game of Life using the equation:

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

Where *S*represents the [transition function](http://en.wikipedia.org/wiki/Cellular_automaton) of the Game of Life, and is defined to be:

Since *S*is a function of two variables, we can graph it by letting the x-axis be the number cells in the neighborhood and the y-axis be the state of the cell:

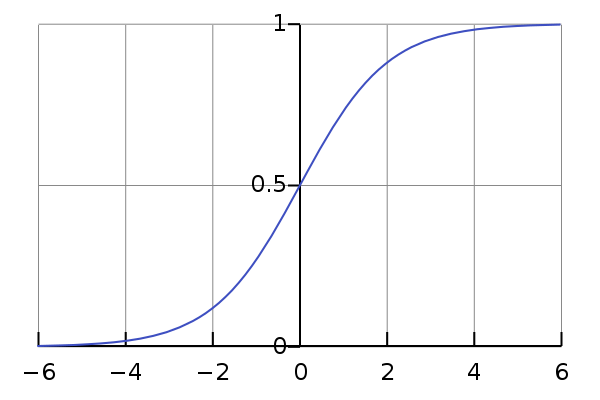
[](https://0fps.files.wordpress.com/2012/11/discrete_transition.png)

Plot of state transition for GoL. Y-axis is N, X-axis is M, red=alive, blue=dead.

Now given such a representation, our task is to `smooth out’ *S* somehow.  Again to make things consistent with Rafler, we first linearly rescale *N, M* so that they are in the range [0,1] (instead of from [0,9), [0,2) respectively).  Our goal is to build up a smooth approximation to *S*using [sigmoid functions](http://en.wikipedia.org/wiki/Sigmoid_function) to represent state transitions and thresholding (this is kind of like how [artificial neural networks](http://en.wikipedia.org/wiki/Artificial_neuron#Types_of_transfer_functions) approximate discrete logical functions).  Of course the term `sigmoid’ is not very precise since a sigmoid can be any function which is the integral of a [bump](http://en.wikipedia.org/wiki/Bump_function).   Therefore, to make things concrete (and again stay consistent with Rafler’s paper) we will use the logistic sigmoid:

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

It is a bit easier to understand what \sigma(x,a) does if we plot it, taking \alpha=4, a=1:

[](http://en.wikipedia.org/wiki/File:Logistic-curve.svg)

Plot of the logistic sigmoid \sigma(x,0) from x = -6 to 6. Image (c) Wikipedia 2008. Author Geoff Richards.

Intuitively, \sigma(x,a) smoothly goes to 0 if *x*is less than *a*, and goes to 1 if *x*is greater than *a*.  The parameter \alpha determines how quickly the sigmoid `steps’ from 0 to 1, while the parameter *a* shifts the sigmoid. Using a sigmoid, we can represent 0/1, alive/dead state of a cell in the effective field by thresholding:

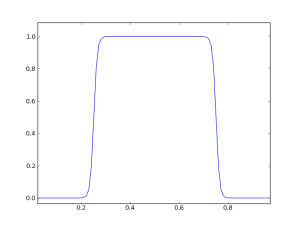
\mathrm{alive}(x,t) \approx \sigma(M(x,t), 0.5)

The idea is that we think of effective field values greater than 0.5 as a live, and those less than 0.5 as dead.  We can also adapt this idea to smoothly switch between any two different values depending on whether a cell is alive or dead:

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

The idea is that \sigma_m selects between *a* and *b* depending on whether the cell is dead or alive respectively.  The other thing we will need is a way to smoothly threshold an interval.  Fortunately, we can also do this using \sigma.  For example, we could define:

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

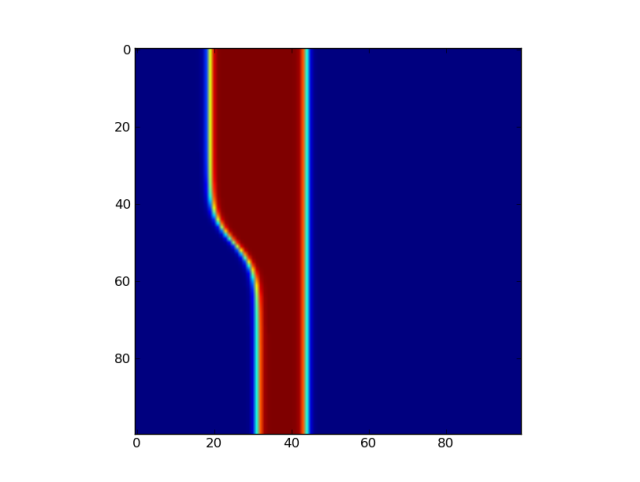
[](https://0fps.files.wordpress.com/2012/11/bisigmoid.png)

Plot of \sigma_n(0.25, 0.75) for *n*=0 to 1

Putting it all together, let’s now make a state transition function which selects the corresponding interval based on whether the state of the cell is currently alive or dead:

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

Where [b_1, d_1], [b_2, d_2] are a pair of intervals which are selected based on user specified parameters.  Based on the game of life, a reasonable set of choices should be b_1 \approx \frac{2}{8}, b_2 \approx d_1 \approx d_2 \approx \frac{3}{8}, which if plotted gives a chart that looks something like this:

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

This is the transition function for SmoothLife as defined by Rafler.  The squishing of this plot relative to the original plot is just a side effect of rescaling the axes to the unit interval, but even so the similarity is unmistakable.  The only thing left is to pick the value of \alpha.  In the original paper on SmoothLife, Rafler allows for two different values of \alpha; \alpha_n, \alpha_m for their appearance in \sigma_n, \sigma_m respectively.  Along with the bounds for the the life/death intervals b_1, b_2, d_1, d_2, there are 6 total parameters to choose from in building a cellular automaton for SmoothLife.

Timestepping

Now, there is still one nagging detail.  I have not yet told you how the time works in this new version of life.  In Rafler’s paper he gives a couple of possibilities.  One simple option is to just do discrete time stepping, for example:

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

However, this has the disadvantage that the time stepping is now discretized, and so it violates the spirit of SmoothLife somewhat.  Another possibility (advocated in Rafler’s paper) is to rewrite this as a differential equation, giving the following smooth update rule:

\frac{d f(x,t)}{dt} = 2 S(N(x,t), M(x,t))-1

I found this scheme to give poor results since it tends to diverge whenever the state of a cell is relatively stable (leading to field values outside the range [0,1]). (**UPDATE:**I just got an email from Stephan Rafler, apparently the equation I wrote earlier was wrong.  Also, he recommends that the field values be clamped to [0,1] to keep them from going out of bounds.)  A slightly better method which I extracted from the source code of SmoothLife is the following technique that pushes the state of the effective field towards S exponentially:

\frac{d f(x,t)}{dt}= S(N(x,t), M(x,t))-f(x,t)

It is also possible to replace *f(x,t)* with*M(x,t)* on the right hand side of the above equations and get similar results.  It is claimed that each of these rules can produce interesting patterns, though personally I’ve only observed the first and last choices actually working…

Discretizing SmoothLife

Now that we have the rules for SmoothLife, we need to figure out how to actually implement it, and to do this we need to discretize the field *f* somehow.  One common way to do this is to apply the [Galerkin method](http://en.wikipedia.org/wiki/Galerkin_method).  The general idea is that for a fixed *t,* we write *f* as a weighted sum of simpler basis functions \phi_i (which I will deliberately leave unspecified for now):

f(x,t) = \sum \limits_i w_i \phi_i(x)

The undetermined weights w_i are the degrees of freedom we use to represent f(x,t).  For example, we could represent f as a polynomial series expansion, in which case \phi_i(x) = x^i and the weights would just be the coefficients, d^i f(x,t) / d x^i.  But there is no reason to stop at polynomials, we could approximate f by whatever functions we find convenient (and later on we’ll discuss a few reasonable choices.)  What does this buy us?  Well if we use a finite number of basis elements, \phi_i, then we can represent our field *f* as a finite collection of numbers w_i!  Cool!  So, how do we use this in our simulation?

Well given that f(x,t) = \sum w_i \phi_i(x) at time *t*, and supposing that at time *t*+1 we have f(x,t+1) = \sum v_i \phi_i(x), all we need to do is solve for the coefficients v_i.  This could be done (hypothetically) by just plugging in f(x,t), f(x,t+1) into both sides of Rafler’s SmoothLife update equation:

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

To compute *M* we just plug in *f(x,t)*:

M(x,t) = \int \limits_{|x-y| \leq h} \sum \limits_i w_i \phi_i(x-y) dy

= \sum \limits_i w_i \int \limits_{|x-y| \leq h} \phi_i(x-y) dy

And a similar formula holds for *N* as well.  Therefore, we have the equation:

Unfortunately, it is usually not the case that there is an exact solution to this equation.  The underlying reason the above idea may fail is that it might not be possible to represent our solution in the basis functions \phi_i exactly.  So we have to modify the equation somewhat.  Instead of trying to get a perfect solution (which is generally impossible for arbitrary nonlinear PDEs or crazy boundary conditions), we’ll settle for approximating the solution as best as possible.  The way we do this that we seek to find a choice of w_i which minimizes an L^2 error in some suitable metric.  For example, we could try to solve the following the following optimization problem instead:

Minimize \int (f(x,t+1) - S(N(x,t), M(x,t)))^2 dx

This type of system is sometimes called the [weak formulation](http://en.wikipedia.org/wiki/Weak_formulation) of the boundary value problem, and by standard linear algebra arguments finding the solution amounts to doing a [subspace projection](http://en.wikipedia.org/wiki/Projection_%28linear_algebra%29).  In general this could be pretty annoying, but if we suppose that each of our \phi_i were [orthonormal](http://en.wikipedia.org/wiki/Orthonormal_basis) — that is,

\int \phi_i(x) \phi_j(x) dx = \left \{ \begin{array}{cc} 1 & \mathrm{if} \: i=j \\ 0 & \mathrm{otherwise} \end{array} \right.

Then we know that,

v_i = \int \phi_i(x) S(N(x,t), M(x,t)) dx

and so all we have to do to compute the next state is integrate *S(N,M)* against each \phi_i.  A similar derivation applies for the smooth time stepping rules as well (modulo a [small technical condition](http://en.wikipedia.org/wiki/Sobolev_space) of \phi_i that they must be differentiable).

Boundary Conditions

Ok, so we know (at least theoretically) what must be done to discretize the system.  But how can we possibly hope to represent an infinite space using a finite number of basis functions?  One simple solution is that we can make the space finite again by adding a periodicity condition.  That is, let us require that for all *x,y*:

f(x,y) = f( 2\pi + x, 2 \pi +y)

This means that we can parameterize the state of the infinite grid by just describing what happens within the region [0,2 \pi)^2.  This square is compact and so we have at least some hope of being able to approximate a periodic *f* by some finite number of functions covering this square.

Sinc Interpolation

Finally, we come to the point where we must pick a basis in order to go any further.  There is some level at which the particular choice of representative functions \phi_i is arbitrary and so one could reasonably justify almost any decision.  For example, I could use B-splines, polynomials, sine waves, fractals or whatever I felt like (at least as long as I can integrate it to compute *N* and *M*).  What I am going to describe here is by no means the only way to proceed.  For example, in his paper Rafler uses a very simple bilinear expansion in terms of quadrilateral elements to get a decent discretization.  (And if you want the details scroll up and read his write up).

However, because our system is periodic and translationally invariant (although not linear) there is one basis which has at least a somewhat more special status, and that is the Dirichlet or [aliased sinc](https://ccrma.stanford.edu/~jos/sasp/Rectangular_Window.html) basis.  Let *R* be the resolution of a uniform grid, and define:

\mathrm{asinc}_{R}( x ) = \frac{ \sin( R x / 2) }{ R \sin(x / w) }

Then we index our basis functions by a pair of indices 0 \leq i,j < R with:

\phi_{i,j}(x,y)=\mathrm{asinc}_{R} (x-\frac{2\pi i}{R})\mathrm{asinc}_{R}(y-\frac{2\pi j}{R}).

It may seem surprising at first, but these basis functions are actually orthogonal.  If you’ve never seen something like this before, please don’t be alarmed!  That definition (which seemed to come out of left field) is really a manifestation of the famous (and often misunderstood) [Nyquist sampling theorem](http://en.wikipedia.org/wiki/Nyquist%E2%80%93Shannon_sampling_theorem).  It basically says that if we write any periodic, band limited function as a sum of sincs:

f(x,y) = \sum_{i=0}^R \sum_{j=0}^R w_{i,j} \phi_{i,j}(x,y)

Then the weights w_{i,j} are just the values of the function at the grid points:

f(\frac{2 \pi }{R} i, \frac{2 \pi}{R} j) = w_{i,j}

This means that for any suitably smooth f, we can project it to our basis by just sampling it:

\int f(x,y) \phi_{i,j}(x,y) dx dy \approx f(i,j)

This makes our code way simpler, since we can avoid doing some messy numerical integration to get the weights.  Assuming that *S(N,M)* is smooth enough (which in practice it is), then all we have to do to perform a discrete update is figure out how to compute *M(x,y), N(x,y)*for any given sinc expansion of *f.*In other words, we want to compute:

M(x,y) = \int_{|p^2 + q^2| \leq h^2} f(x-p, y-q) dp dq

But this is just a convolution [convolution](http://en.wikipedia.org/wiki/Convolution) with the [indicator function](http://en.wikipedia.org/wiki/Indicator_function) of the [unit disk](http://en.wikipedia.org/wiki/Unit_sphere#Unit_balls_in_normed_vector_spaces):

M(x,y,t) = \int_{0}^{2 \pi} \int_{0}^{2 \pi} 1_{B_h}(p,q) f(x-p, y-q, t) dp dq

If you know a bit of Fourier analysis, you should know what comes next.  Just take a [Fourier transform](http://en.wikipedia.org/wiki/Fourier_transform) and apply the [convolution theorem](http://en.wikipedia.org/wiki/Convolution_theorem) to both sides, giving:

\hat{M}(\omega_x,\omega_y,t)=\hat{1_{B_h}}(\omega_x,\omega_y)\hat{f}(\omega_x,\omega_y)

Which we can actually solve exactly, since the Nyquist theorem implies that \hat{f} is a finite sum. Therefore:

M(x,y)=\sum \limits_{i=0}^{R}\sum \limits_{j=0}^{R}\hat{1_{B_h}}(i,j)\hat{f}(i,j)e^{2\pi\sqrt{-1}(ix+jy)}

And [by a bit of calculus](http://isi.ssl.berkeley.edu/~tatebe/whitepapers/FT%20of%20Uniform%20Disk.pdf), the Fourier transform of 1_{B_h} can be computed in closed form using Bessel functions.  This means that:

A similar formula for *N(x,y,t)* can be obtained by writing:

N(x,y) = \int (1_{B_{3h}} - 1_{B_{h}})(p,q) f(x-p,y-q,t) dp dq

Putting it all together, let w'_{i,j} be the new weights for *f* at time *t*+1.  Then to do a discrete time step we just set:

S(N(2 \pi i /R, 2 \pi j/R), M(2 \pi i/R, 2 \pi j / R)) \mapsto w'_{i,j}

I’ll leave it as an exercise to work out the update rules for the continuous time steps.

Implementation Details

Great, so we have some formulas for computing the next time step, but if we do it in a dumb way it is going to be pretty slow.  The reason is that each of the above steps requires summing up all the coefficients of *f* to calculate *M* and *N* at each point, which is going to take linear work with respect to the number of terms in our summation.  Spread out over all the points in the grid, this makes the total update O(n^2).  However, because the above sums are all Fourier series we can speed this up a lot using a fast Fourier transform.  Basically we precalculate the weights by evaluating the Bessel function at each frequency, then at each time step we Fourier transform f, multiply by the weights, inverse transform and repeat.  Here is what it looks like:

* Initialize f
* Precalculate \widehat{1_{B_h}}, \widehat{1_{B_{3h}}}
* For t=1 to n
  + Use FFT to compute \widehat{f}
  + Set M = \frac{1}{2 \pi h^2} \mathcal{F}^{-1}( \widehat{1_{B_h}} \widehat{f} )
  + Set N = \frac{1}{8 \pi h^2} \mathcal{F}^{-1}((\widehat{1_{B_{3h}}}-\widehat{1_{B_h}})\widehat{f})
  + Set f(i,j) = S(N(i,j), M(i,j))

And that’s it!

Demo

If you want to try out SmoothLife yourself in your browser, I made a few jsFiddles which illustrate the basic principle.  Here is a Fourier based implementation that follows the discussion in this article pretty closely:

<http://jsfiddle.net/mikola/aj2vq/>

I also made a second WebGL/GPU based implementation that uses a discretization similar to that proposed in Rafler’s original paper:

<http://jsfiddle.net/mikola/2jenR/>

You can run either of them in  your browser and try changing the parameters.

Next Time…

The stuff in this article is basically background material, and most of it can be found in Rafler’s paper or is already more-or-less documented in the source code for the [SmoothLife project on SourceForge](http://sourceforge.net/projects/smoothlife/).  However, in the sequel I plan to go a bit farther than the basic rules for SmoothLife and tell you how to make a version of SmoothLife for curved surfaces!