Generating textures on Surfaces with Reaction-Diffusion systems in the GPU

Leonardo Carvalho Maria Andrade* Luiz Velho

IMPA, Brazil *UFAL, Brazil

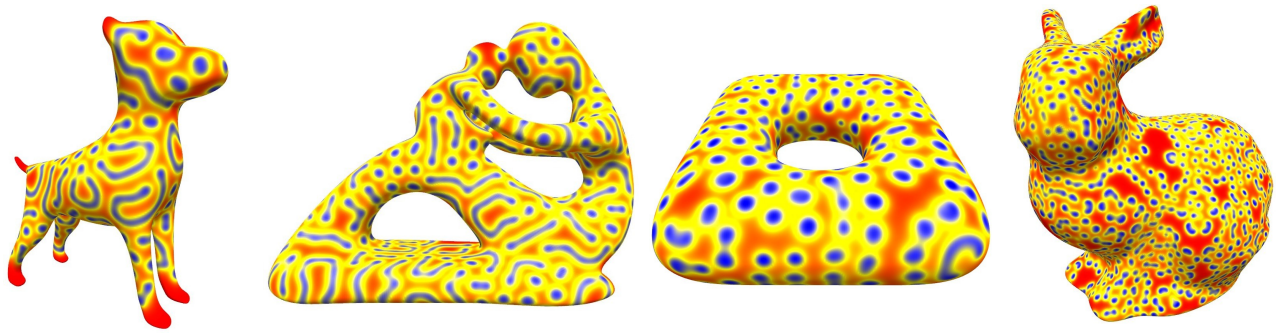

Figure 1: Some results of the method. Complex patterns are formed in a few seconds using CUDA.

Abstract

In this paper we present a method to create textures on smooth surfaces of arbitrary topology using Reaction-Diffusion systems in a graphics processing unit (GPU). To do this we use a parametrization of Catmull-Clark subdivision surfaces and obtain the metric information of the distortion caused by this parametrization, so we can calculate differential operators of functions defined on this surface. Then we solve the systems in the domain of parametrization of each patch. This process can be done in parallel for each point in the discretization of the surface, so the GPU implementation can heavily increase the velocity of computation.

Keywords: surfaces; differential operators; Reaction-Diffusion; CUDA; efficiency.

Authors' contact:
{leo1984,mcosta,lvelho}@impa.br

1. Introduction and Related Work

Turing [1952] described for the first time a chemical mechanism for pattern formation which is called *Reaction-Diffusion*, where two or more chemicals react one with the other, and they diffuse at different rates, resulting in stable patterns like spots and stripes. This mechanism has been replicated and expanded over the years by researchers in several areas [Epstein and Pojman, 1998]. Turk [1991] used it to generate textures that match the geometry of polyhedral surfaces. Moreover, Sanderson et al. [2006] used many Reaction-Diffusion models for textures synthesis. Bajaj et al. [2008] gave an approach to solve Reaction-Diffusion systems on surfaces using a Galerkin based finite element methods.

The mechanism of Reaction-Diffusion involves a numeric solution of a non-linear partial differential equations system. This nonlinearity makes it difficult to select appropriate parameters in order to ensure the formation of stable patterns, which may take to the user many attempts to obtain a reasonable result. Another problem is that the solution can be computational expensive, such that it can be time consuming. To minimize this problem, algorithms can take advantage of multiple processors computers, solving the PDEs in parallel. A good choice is using the Graphics Processing Units (GPUs), that were originally developed to accelerate graphics operations, like the rendering of a virtual scenario, but recently they have been used to solve more general problems that requires compute-intensive parallel computation, due to the design of those units. In this case we can say that they are General Purpose GPUs, or simply GP-GPU. In [Sanderson et al. 2009] it is described a GP-GPU-based approach for solving partial differential equations (PDEs) for Reaction-Diffusion models using Cg (C for graphics) [Mark et al. 2003], but this language requires that the programmers understand the graphics processing pipeline, and know how to solve their problems in this context. NVIDIA developed CUDA, a general purpose parallel computing architecture that allows the programmers to develop programs that run in the GPU using C as a high-level programming language [NVIDIA 2011]. With this architecture it is not necessary for the developer to know the graphics pipeline, so it is possible to make programs in the GPU without having to adapt the solution of a problem to the graphics pipeline.

Stam [2003] developed a method to simulate fluids on surfaces of arbitrary topology solving the Navier-Stokes equations in the domain of the surface parametrization. His method handles the distortion

caused by the parametrization and cross-patch boundary conditions. The author used a parametrization of a Catmull-Clark surface [Catmull and Clark 1978], using his evaluation method described in [Stam 1998]. It was briefly shown that this technique can be used in the solution of other types of equations, like the Reaction-Diffusion systems.

In our work we made a GPU implementation in CUDA of the scheme introduced by Stam to solve Reaction-Diffusion models on parametric surfaces.

2. Basic concepts

Let S be a surface formed by parametric patches $X_p : \Omega_p \to \mathbb{R}^3$, where $\Omega_p = [0,1] \times [0,1]$, and $X_p(x^1, x^2) = (y_p^1(x^1, x^2), y_p^2(x^1, x^2), y_p^3(x^1, x^2))$. We use the tangent vectors

$$\mathrm{X}_{x^k} = \left(\frac{\partial y^1}{\partial x^k}, \frac{\partial y^2}{\partial x^k}, \frac{\partial y^3}{\partial x^k} \right), \quad k = 1, 2,$$

where p is omitted to simplify notation, to define the local metric matrix $(g_{i,j})$:

$$g_{i,j} = \langle X_{x^i}, X_{x^j} \rangle, \quad i, j = 1, 2,$$

from which we get $G = \det(g_{i,j})$. The elements of the inverse matrix are denoted by $(g^{i,j}) = (g_{i,j})^{-1}$.

With this metric, we calculate differential operators of functions defined on S [Aris 1989]. In particular, the Laplacian is given by

$$\nabla^2 \varphi = \frac{1}{\sqrt{G}} \frac{\partial}{\partial x^i} \left(\sqrt{G} g^{i,j} \frac{\partial \varphi}{\partial x^j} \right).$$

where we are using Einstein's notation (where $a_i b^{i,j} c_j = \sum_{i,j} a_i b^{i,j} c_j$), with indices from 1 to 2.

To handle the intersection of adjacent patches, we use the transition functions defined in [Stam 2003]. Each edge of a domain Ω_p receives a label from 0 to 3, and the transition function from patch p_i to an adjacent patch p_j is calculated from a transition number t_k that depends only on the labels of the common edge of these patches.

In this work we use the Catmull-Clark subdivision scheme, with the evaluation technique developed by Stam [1998], that gives us a parametrization, where each quadrilateral in the polygonal base mesh generates a parametric patch .

We discretize a surface like in [Stam 2003], where for each patch there is a grid with $N \times N$ cells containing values from the patch, and there is an extra layer of cells with values from neighbour patches. The metric values from the parametrization are stored in a denser grid with $(2N+1) \times (2N+1)$ points. The Laplacian operator is discretized by a finite differences scheme for the derivatives, using metric values.

3. Reaction-Diffusion Systems

Reaction-Diffusion systems are defined by the non-linear partial differential equations:

$$\begin{cases} \frac{\partial a}{\partial t} &= F(a,b) + r_a \nabla^2 a, \\ \frac{\partial b}{\partial t} &= G(a,b) + r_b \nabla^2 b, \end{cases}$$

where a and b are substances distributed in space, F and G are functions that control the production rate of a and b, and the coefficients r_a and r_b are the diffusion rates.

These substances are affected by two processes: local chemical reactions, which means that the substances are transformed into each other, and diffusion which causes the substances to spread out over a surface in space.

We consider the Reaction-Diffusion model developed by Gray and Scott [1985], which is defined by

$$F(a,b) = -ab^2 + f(1-a),$$
$$G(a,b) = ab^2 - (f+k)b,$$

where f and k are real parameters.

Depending on the initial conditions the solution of this system forms different patterns. It is necessary to choose appropriate values for the parameters f and k, otherwise the result converge in time to a trivial solution like $a = 1$ and $b = 0$ for all points.

4. Implementation in the GPU

We see that the problem described here can be easily parallelized, so it is suitable to be solved using many core processors, which can considerably improve the performance of the method.

4.1 Data Structures

To implement the method in CUDA, firstly the problem data must be transfered to the GPU memory. In CPU the grid data are stored in arrays of size w × h × n_patches, where n_patches is the number of patches of the surface, such that the value `v(i,j,p)` at position (i,j) and patch $p \in [0, \cdots, n\_patches - 1]$ is accessed via `v[i+j*w+p*w*h]`. For scalar fields we have `w=h= N+2`, so the value $\varphi_{i,j}^p$ of a scalar field φ centered at cell i,j and patch p is stored at `phi(i,j,p)`. For the metric we use `w=h=2*N+1` to store the values $\sqrt{g}, g^{11}, g^{12}$ and g^{22}. The value $(\sqrt{g})_{i,j}^p$ is accessed via `g(2*i-1, 2*j-1, p)`, and similarly for the other values. The arrays could be just copied to the GPU global memory using arrays in the same format and be used the same way as in the CPU, but this way would not take advantage of the GPU capabilities. A better option is put data in the texture or

surface memory, which are cached in the texture cache, optimized for 2D spatial locality. In our case we can use a layered surface reference putting the data of each patch in a layer. The data of the patches are stored in CUDA arrays created with `cudaMalloc3DArray()` and copied from and to CPU using `cudaMemcpy3D()`. The value `g(i,j,p)` in surface memory is accessed via `surf2DLayeredread(&a,surf_ref,i*4,j,p)` where `surf_ref` is the surface reference bound to the corresponding CUDA array, and we have to multiply the x-coordinate by the byte size of the element because surface memory uses byte addressing. We can also write in the grid using `surf2DLayeredwrite(a,surf_ref,i*4,j,p)`.

4.2 Precomputing

The surface evaluation needs to be calculated only once, we calculate for each point of the discretization its position on the surface, the derivatives for each direction x_1 and x_2, and from that we calculate the metric information.

The surface is evaluated with an implementation in CUDA of the method described in [Stam 1998]. Each point on the surface can be evaluated by $X_p(x^1, x^2) = \sum_{i=1}^{K} \varphi_i(x^1, x^2)\mathbf{p}_i^p$, where K is the number of control points used by patch p, $\mathbf{p}_i^p$ is the projection of the i-th control point into the eigenspace of the Catmull-Clark subdivision matrix, φ_i depends on the eigen-data of this matrix and on cubic B-spline basis functions. To minimize the number of calculations, we firstly evaluate the basis functions since they only depend on the local coordinates of each point in the discretization, so they can be used for every patch. Then we evaluate the surface using a CUDA implementation of the function `EvalSurf` described in [Stam 1998], also calculating the first derivatives at each direction.

With position and derivatives it is straightforward to get the metric. The positions and derivatives data are kept in OpenGL vertex buffer objects to be used in the drawing of the surface.

4.3. Solving Reaction-Diffusion

In CUDA we create special functions called *kernels*, that are executed in parallel, each one in one CUDA thread. The threads are distributed hierarchically in *blocks* and *grids*, such that threads form a one, two or three-dimensional block, and blocks form a one, two or three-dimensional grid. Each thread block is managed by one GPU core, that executes a group of 32 threads called *warp*. If all the threads in a warp execute the same instructions then they are all executed in parallel, otherwise each execution path is executed serially. So to prevent loosing performance it is important to distribute the threads such that in the same block most of the kernels have the same execution path.

Another important issue refers to the memory management. Using appropriate structures we can improve the performance of the reading/writing operations. In our case, using the texture and surface memories we get the best performance in the execution of threads in the same warp that read texture addresses that are close together in 2D.

To solve the Reaction-Diffusion equations, we distribute the threads such that each block processes points in the same patch of the surface. This way we prevent that threads in the same warp execute data that are not close in 2D. Each block is two-dimensional, containing a total number of threads that is a multiple of 32, such that none of the warps contains less than the maximum warp size. The blocks are organized in three-dimensional grids (this requires a GPU with CUDA capabilities 2.0 or above), where the first two dimensions correspond to the block distribution in a patch, and the third dimension indicates the patch index. Each kernel will process the point at coordinates (i, j) of patch p, where $i, j \in [1, \cdots, N]$, $p \in [0, \cdots, n\_patches - 1]$. To identify (i, j) and p at each kernel, we calculate:

```
int i = blockIdx.x*blockDim.x +
        threadIdx.x + 1;
int j = blockIdx.y*blockDim.y +
        threadIdx.y + 1;
int p = blockIdx.z;
```

If N is not a multiple of the block dimensions, then in some blocks we will have $i > N$ or $j > N$, we can ignore these cases, but this reduces the performance of the program, because there will be some threads in the same warp with different execution paths. Then it is better to avoid these cases always when it is possible, choosing properly the block dimensions.

After identifying the position and patch index, each kernel calculates the values of the rates of change da and db of each concentration, and keep these values in the per-thread local memory. Then we call the function `__syncthreads()` to make sure that all threads have already computed the rates of change before we change the values of a and b. This is done in two steps, first we solve the non-linear part of the equation using a forward Euler method. For the linear part we use a simple iterative method to solve the implicit equation $a^{n+1} - \Delta t r_a \nabla^2 a^{n+1} = a^n$, where a^n is the solution at time $n\Delta t$, and similarly for concentration b.

To update the boundaries, we use a kernel that gets for each boundary of each patch the corresponding neighbour patch index and the transition number t_k, and uses the transition function to calculate the position of the cell at the neighbour patch. The

informations about the neighbour patches and the transition numbers t_k are stored each in an array of size $4n\_patches$, created when the surface was constructed. Then `neigh_indices[p*4+e]` corresponds to the neighbour patch index of patch p at edge $e \in [0, 1, 2, 3]$. Another kernel is responsible for the corners cells, been called only four times per patch, calculating the average of the cells next to each cell.

5. Results

For our tests we used an NVIDIA GeForce GTX 470, which has 448 CUDA cores. We initialize the concentration values, assigning for most of the points 100% of chemical a and 0% of b, and in some regions we assign 50% for a and 25% for b, with a $\pm 1\%$ random noise. In our examples we calculated circular regions in the domain of the patches, randomly changing the center and the radius of each circle. This randomness in the initial conditions avoids too much symmetrical results, so we can get a larger diversity of patterns generated by the method.

Figure 1 shows some results of the implementation of this method. For the first two models we used $f = 0.03$ and $k = 0.06$, for the last two we used $f = 0.02$ and $k = 0.055$. The parameters r_a and r_b define the size of the spots and stripe, we chose their values according to the scale of each mesh. The red points correspond to $a = 1$, blue to $a = 0.2$, and yellow to $a = 0.6$, this is processed in GLSL shaders.

Table 1 shows the time taken to calculate one iteration of the method for some meshes, changing only the size of the grids. We used quadrilateral meshes modeled using some tool or converted from well known triangular meshes. A limitation of the structures we used is that there is a limit size for texture dimensions, so we were not able to run the program with N = 64 with denser meshes, like Stanford bunny model. However for those cases it is usually sufficient to use a small grid size.

Surface	$n\_patches$	**N=8**	**N=16**	**N=32**	**N=64**
Toroidal	128	2ms	6ms	16ms	60ms
Fertility	166	3ms	7ms	22ms	75ms
Dog	238	4ms	10ms	30ms	107ms
Bunny	1292	29ms	56ms	160ms	---

Table 1: Time taken to calculate one iteration, varying the model and grid size.

6. Conclusion and Future Works

In this work we showed how to solve the system of Reaction-Diffusion on surfaces using a GPU implementation using a parametrization of Catmull-Clark surfaces. We used suitable structures to take advantage of the GPU resources, increasing the performance of the numeric solution. For future works we may study different schemes, to generate more complex results, simulating for example natural patterns formed on the skin of animals.

References

ARIS, R., 1989. *Vectors, Tensors and the Basic Equations of Fluid Mechanics*. Dover Publications.

BAJAJ, C., ZHANG, Y. and XU, G., 2008. Physically-based surface texture synthesis using a coupled finite element system. In *Proceedings of the 5th international conference on Advances in geometric modeling and processing*, GMP'08, Berlin, Heidelberg. Springer-Verlag.

CATMULL, E. and CLARK, J., 1978. Recursively generated b-spline surfaces on arbitrary topological meshes. *Computer-aided Design*, 10:350–355. doi: 10.1016/0010-4485(78)90110-0.

GRAY, P. and SCOTT, S.K., 1985. Sustained oscillations and other exotic patterns of behavior in isothermal reactions. *J. phys. Chem*., 89, pp. 22–32

EPSTEIN, I.R. and POJMAN, J.A., 1998. *An Introduction to Nonlinear Chemical Dynamics* . Topics in Physical Chemistry. Oxford University Press, New York.

MARK, W.R., GLANVILLE, R.S., AKELEY, K., and KILGARD, M.J., 2003. Cg: a system for programming graphics hardware in a c-like language. *ACM Trans. Graph.*. ISSN 0730-0301. doi: 10.1145/882262.882362.

NVIDIA, 2011. NVIDIA CUDA Programming Guide 4.1.

SANDERSON, A.R., KIRBY, R.M., JOHNSON, C.R., and YANG, L., 2006. Advanced Reaction-Diffusion Models for Texture Synthesis. *Journal of Graphics Tools*, 11(3):47–71.

SANDERSON, A.R., Meyer, M.D., Kirby, R.M. and Johnson, C.R., 2009. A framework for exploring numerical solutions of advection-reaction-diffusion equations using a gpu-based approach. *Comput. Vis. Sci.* ISSN 1432-9360. doi: 10.1007/s00791-008-0086-0. 12(4):155–170.

STAM, J., 1998. Exact evaluation of catmull-clark subdivision surfaces at arbitrary parameter values. In *Proceedings of SIGGRAPH-1998*, 395–404.

STAM, J., 2003. Flows on surfaces of arbitrary topology. *ACM Trans. Graph.*, 22(3): 724–731.

TURING, A.M., 1952. The chemical basis of morphogenesis. *Philosophical Transactions of the Royal Society of London*. Series B, Biological Sciences, 237(641):37–72.