Fluid Simulation on Surfaces in the GPU

Leonardo Carvalho
IMPA
Rio de Janeiro, Brazil
leo1984@impa.br

Maria Andrade
UFAL
Maceió, Brazil
mcosta@impa.br

Luiz Velho
IMPA
Rio de Janeiro, Brazil
lvelho@impa.br

***Abstract*—In this paper we present a method to simulate fluids on smooth surfaces of arbitrary topology using a graphics processing unit (GPU). To do this we use the 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 the Navier-Stokes equations on surfaces are solved in the domain of parametrization of each surface 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; Catmull-Clark subdivision; Differential operators; GPU; cuda; efficiency; Fluid simulation.**

I. Introduction

In recent years, many researches were developed to use the Navier-Stokes equations for fluid simulation. In particular, these investigations have contributed in many areas, like special effects industry. Bridson [1] presents a practical introduction to fluid simulation for computer graphics, with an overview of algorithms used to simulate two and three-dimensional incompressible flows.

Stam [2] proposed an algorithm called *stable-fluids* that solves the Navier-Stokes equations for three-dimensional fluids, which is fast, stable and it is the basis to simulate smoke, water and fire, but this process is dissipative. Fedkiw et al. [3] made a change in the discretization in order to reduce the problem of dissipation.

Stam [4] also 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 [5], using his evaluation method described in [6].

Fluid simulation is generally a computing intensive task, so it can take a lot of time to be performed. To minimize this problem, algorithms can exploit multiple processors computers, solving the partial differential equations in parallel. A good choice is to use the Graphics Processing Units (GPUs), which 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 require compute-intensive parallel computation, due to the design of these units. In this case, they are called General Purpose GPUs, or simply GP-GPU. Initially, developers could take advantage of the GPU power through the graphics pipeline using shading languages such as Cg (C for graphics) [7], but this requires that the programmers understand the graphics processing pipeline, and know how to solve their problems in this context. More recently, NVIDIA developed CUDA, a general purpose parallel computing architecture that allows the development of programs that run in the GPU using C as a high-level programming language [8]. With this architecture it is not necessary for the developer to know the graphical pipeline, so it is possible to make programs in the GPU without having to adapt the solution of a problem to this pipeline. NVIDIA [9] has developed numerous practical techniques for creating realistic effects in the GPU, including the stable fluids method. Some researchers implemented fluid simulation in the GPU, like [10] that implemented real-time simulations of large scale three dimensional liquids. In [11] a method was developed for rendering the surface of fluids in real-time using SPH [12]. In [13] it is described a technique to solve the incompressible Navier-Stokes fluid equations using SMAC. All these methods were developed for fluid-simulation in non-curved spaces with two or three dimensions. Hegeman et al. [14] simulate flow for an arbitrary surface of genus zero using GPU and conformal map.

In our work we made a GPU implementation in CUDA of the scheme developed by Stam [4] for fluid-simulation on smooth surfaces of arbitrary topology. The next sections present: the basic concepts of a surface formed by parametric patches, with some differential operators defined on this surface, and the transition functions that make coordinate changes between patches; the parametrization of subdivision surfaces; a discretization of a surface and its operators; the solution of Navier-Stokes equations in this scheme; and finally the implementation in the GPU of the method.

II. 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 want to calculate properties defined on S using values in the domain space $(x^1,x^2) \in \Omega_p$, but to avoid distortions it is necessary to include geometric information about the surface in the calculations. We use the tangent vectors

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

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 $(g^{i,j}) = (g_{i,j})^{-1}$ can be obtained by

$$g^{1,1} = \frac{g_{2,2}}{G}, \quad g^{2,2} = \frac{g_{1,1}}{G}, \quad g^{1,2} = g^{2,1} = -\frac{g_{1,2}}{G}.$$

With this metric, we can calculate differential operators of functions defined on S. These operators were taken from [15]. The gradient of a scalar function φ on S is given by

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

where we are using Einstein's notation [1], with indices from 1 to 2. The divergence of a vector function $\mathbf{u}$ is

$$\nabla \cdot \mathbf{u} = \frac{1}{\sqrt{G}}\frac{\partial}{\partial x^i}\left(\sqrt{G}u^i\right).$$

The Laplacian is then

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

To correctly calculate these operators, we must handle the intersection of adjacent patches, using transition functions from one domain to another. In [4], each edge of a domain Ω_p receives a label from 0 to 3, defined in a counterclockwise order, then the transition function from patch p_i to an adjacent patch p_j is given by $\phi_{\langle e_i, e_j \rangle}$, where e_i and e_j are the labels of the common edge of these patches, the operator $\langle \cdot, \cdot \rangle$ is defined by[2] $\langle e_i, e_j \rangle = (4 + e_i - (e_j + 2)\%4)\%4$, and

$$\begin{aligned}
\phi_0(x^1, x^2) &= (x^1, x^2),\\
\phi_1(x^1, x^2) &= (x^2, 1 - x^1),\\
\phi_2(x^1, x^2) &= (1 - x^1, 1 - x^2),\\
\phi_3(x^1, x^2) &= (1 - x^2, x^1).
\end{aligned}$$

Figure 1 shows an example of the transition function from a patch p_i (on the left) to patch p_j (on the right). In this case, $e_i = 1, e_j = 0$, so $\langle e_i, e_j \rangle = 3$, and the transition function is then ϕ_3. So, for a function φ defined on the surface, a value on the edge 1 of patch p_i is equal to a value in edge 0 of patch p_j.

In general the relation is given by $\varphi_{p_i}(x^1, x^2) = \varphi_{p_j}(\phi_{\langle e_i, e_j \rangle} T_{e_i}(x^1, x^2))$, where

$$\begin{aligned}
T_0(x^1, x^2) &= (x^1, x^2 + 1),\\
T_1(x^1, x^2) &= (x^1 - 1, x^2),\\
T_2(x^1, x^2) &= (x^1, x^2 - 1),\\
T_3(x^1, x^2) &= (x^1 + 1, x^2).
\end{aligned}$$

To transform vectors from one patch to another, we also need the Jacobian matrix M_i of each transition function ϕ_i,

[1]The Einstein's notation means $a_i b^{i,j} c_j = \sum_{i,j} a_i b^{i,j} c_j$.

[2]Where $a\%b$ means a modulus b.

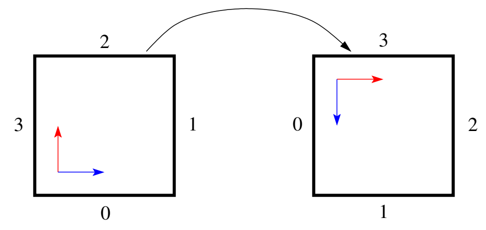

Fig. 1. Example of transition function.

which we can easily see that is a rotation matrix of angle $i(\pi/2)$ counterclockwise, i.e.

$$M_i = R_{i\pi/2}, \quad i \in [0, 3].$$

For a vector function $\mathbf{v}$ defined on the surface, the relation between a vector in the edge between patches p_i and p_j is $\mathbf{v}_{p_i}(x^1, x^2) = M_{\langle e_i, e_j \rangle}\mathbf{v}_{p_j}(\phi_{\langle e_i, e_j \rangle} T_{e_i}(x^1, x^2))$.

III. Subdivision Surfaces

In computer graphics it is usual to use a coarse polygon mesh that approximates the shape of a desired surface. To obtain the smooth surfaces, each polygonal face is subdivided into smaller faces that better approximate the smooth surface and in the limit of subdivision we get the smooth surface.

In this work we used the Catmull-Clark subdivision surface, which is a generalization of bi-cubic uniform B-spline for arbitrary meshes. This process generates limit surfaces that are C^2 continuous everywhere except at extraordinary vertices where they are C^1 continuous. In particular, at each point on a surface the tangent plane can be defined.

Stam [6] developed a technique to evaluate the limit surface of a Catmull-Clark subdivision surface, the result is a parametrization, where each quadrilateral in the polygonal base mesh generates a parametric patch $X_p : \Omega_p \to \mathbb{R}^3$, where $\Omega_p = [0, 1] \times [0, 1]$, so it fits the scheme described in the last section, then it is a good candidate to be used in the method that we will present. The author made his implementation publicly available thanks to Alias–wavefront at http://www.dgp.toronto.edu/~stam/reality/Research/SubdivEval.

IV. Discretization

We want to make calculations using differential operators in a discrete set of points of S. At each point we must be able to obtain the metric from the parametrization, and the partial derivatives necessary to the operators. Usually we can not get the continuous derivatives, so we approximate them using a finite differences scheme.

A. Domain discretization

If each Ω_p is quadrilateral, a simple and natural way of discretizing the points is by using an $N \times N$ regular grid. To get accurate and unbiased derivatives we use the so-called MAC grid [1], [16], which is a staggered grid, where values from scalar functions are calculated at the center of cells, the first coordinates of a vector function are located at vertical edges, and the second coordinates are located at horizontal edges, see Figure 2. This kind of grid was also used by Stam [4], and we mostly follow the model developed there.

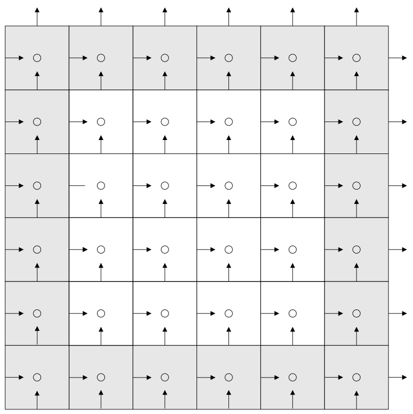

Fig. 2. Discretization grid.

The metric values are stored in a denser $(2N+1)\times(2N+1)$ grid, such that at every position of the grid (center, edges and corners), there is the metric information. This has to be calculated only once, in a precomputation step.

To handle boundary conditions, we add cells that are outside of the patch domain (the gray cells in Figure 2). So the grid resolution for each patch is in fact $(N+2)\times(N+2)$, the first coordinates of vector fields are stored in a $(N+3)\times(N+2)$ grid, the second coordinates in a $(N+2)\times(N+3)$ grid, and the metric values in $(2N+3)\times(2N+3)$ grids. The values at the extern cells are obtained from neighbour patches, or, when there isn't a neighbour patch at some side, they receive values according to boundary conditions. For scalar fields, a value at a boundary cell can be obtained using grid versions of the transition functions:

$$
\begin{aligned}
[0,i,j] &= (i,j),\\
[1,i,j] &= (j, N+1-i),\\
[2,i,j] &= (N+1-i, N+1-j),\\
[3,i,j] &= (N+1-j, i).
\end{aligned}
$$

Then for a scalar field φ we make

$$
\begin{aligned}
\varphi_{0,i} &= \varphi^3_{[t_3,N,i]}, \quad \varphi_{N+1,i} = \varphi^1_{[t_1,1,i]},\\
\varphi_{i,0} &= \varphi^0_{[t_0,i,N]}, \quad \varphi_{i,N+1} = \varphi^2_{[t_2,i,1]},
\end{aligned}
$$

where φ^k is the scalar field of the adjacent patch at edge k, $i=1,\cdots,N$, and $t_k = \langle k, e_k\rangle$, see Figure 3.

For vector fields, it is necessary to multiply the values from a neighbour patch adjacent to edge k by the transition matrix M_{t_k}. Define T such that $T\mathbf{u}_{(i+0.5,j)} = \left(u^1_{(i+0.5,j)}, 0\right)$ and $T\mathbf{u}_{(i,j+0.5)} = \left(0, u^2_{(i,j+0.5)}\right)$ for integer values i and j. Then we can get boundary values for vector fields using:

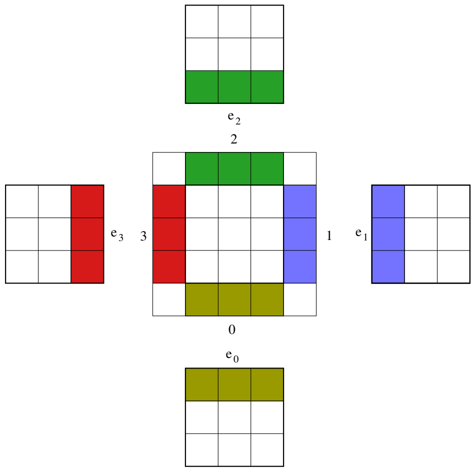

Fig. 3. Boundary cells.

$$
\begin{aligned}
\left(u^1_{i-\frac{1}{2},0}, u^2_{i,-\frac{1}{2}}\right) &= M_{t_0}\left(T\mathbf{u}^0_{[t_0,i-\frac{1}{2},N]} + T\mathbf{u}^0_{[t_0,i,N-\frac{1}{2}]}\right),\\
\left(u^1_{N+\frac{3}{2},j}, u^2_{N+1,j-\frac{1}{2}}\right) &= M_{t_1}\left(T\mathbf{u}^1_{[t_1,\frac{3}{2},j]} + T\mathbf{u}^1_{[t_1,1,j-\frac{1}{2}]}\right),\\
\left(u^1_{i-\frac{1}{2},N}, u^2_{i,N+\frac{3}{2}}\right) &= M_{t_2}\left(T\mathbf{u}^2_{[t_2,i-\frac{1}{2},1]} + T\mathbf{u}^2_{[t_2,i,\frac{1}{2}]}\right),\\
\left(u^1_{-\frac{1}{2},j}, u^2_{0,j-\frac{1}{2}}\right) &= M_{t_3}\left(T\mathbf{u}^3_{[t_3,N-\frac{1}{2},j]} + T\mathbf{u}^3_{[t_3,N,j-\frac{1}{2}]}\right),
\end{aligned}
$$

where $\mathbf{u}^k$ is a vector field of the adjacent patch at edge k.

To compute the metric information $\sqrt{G}$ at a boundary cell, we just copy this value from an adjacent patch like any scalar field, because it does not depend on the orientation of the patches. When t_k is even then it is easy to see that the metric data $g^{1,1}, g^{1,2}$ and $g^{2,2}$ do not change, so they can be simply copied from the neighbour patch. But for an odd t_k, we must swap values $g^{1,1}$ and $g^{2,2}$, and change the sign of $g^{1,2}$, due to the changing in orientation of the derivatives X_{x^1} and X_{x^2}.

To set the value at corner cells we may calculate some average of the neighbours cells, our results were satisfactory using just the average of the two boundary cells adjacent to each corner cell.

B. Discretization of operators

The differential operators must be discretized so we can work in the domain described in last section. Let φ be a scalar field defined in the center of each cell. The gradient of φ is a vector field, so we store its coordinates in cell edges. The first coordinates are calculated in vertical edges $(i-0.5, j)$, the required derivatives at these positions can be discretized as:

$$
\begin{aligned}
\left(\frac{\partial\varphi}{\partial x^1}\right)_{i-\frac{1}{2},j} &\approx \frac{\varphi_{i,j}-\varphi_{i-1,j}}{h},\\
\left(\frac{\partial\varphi}{\partial x^2}\right)_{i-\frac{1}{2},j} &\approx \frac{\varphi_{i-1,j+1}-\varphi_{i-1,j-1}+\varphi_{i+1,j+1}-\varphi_{i+1,j-1}}{4h},
\end{aligned}
$$

where h is the grid spacing.

Similarly, for values in horizontal edges we have

$$\left(\frac{\partial\varphi}{\partial x^1}\right)_{i,j-\frac{1}{2}} \approx \frac{\varphi_{i+1,j}-\varphi_{i-1,j}+\varphi_{i+1,j-1}-\varphi_{i-1,j-1}}{4h},$$
$$\left(\frac{\partial\varphi}{\partial x^2}\right)_{i,j-\frac{1}{2}} \approx \frac{\varphi_{i,j}-\varphi_{i,j-1}}{h}.$$

Then the gradient coordinates can be calculated:

$$\begin{aligned}(\nabla\varphi)^1_{i-\frac{1}{2},j} &= \left(g^{1,1}\right)_{i-\frac{1}{2},j}\left(\frac{\partial\varphi}{\partial x^1}\right)_{i-\frac{1}{2},j} \\ &\quad + \left(g^{1,2}\right)_{i-\frac{1}{2},j}\left(\frac{\partial\varphi}{\partial x^2}\right)_{i-\frac{1}{2},j}\end{aligned}$$

$$\begin{aligned}(\nabla\varphi)^2_{i,j-\frac{1}{2}} &= \left(g^{2,1}\right)_{i-\frac{1}{2},j}\left(\frac{\partial\varphi}{\partial x^1}\right)_{i,j-\frac{1}{2}} \\ &\quad + \left(g^{2,2}\right)_{i-\frac{1}{2},j}\left(\frac{\partial\varphi}{\partial x^2}\right)_{i,j-\frac{1}{2}}\end{aligned}$$

To calculate the divergence of a vector field $\mathbf{u}$ we need the derivatives:

$$\left(\frac{\partial}{\partial x^1}\left(\sqrt{G}u^1\right)\right)_{i,j} \approx \frac{\left(\sqrt{G}u^1\right)_{i+\frac{1}{2},j}-\left(\sqrt{G}u^1\right)_{i-\frac{1}{2},j}}{h},$$
$$\left(\frac{\partial}{\partial x^2}\left(\sqrt{G}u^2\right)\right)_{i,j} \approx \frac{\left(\sqrt{G}u^2\right)_{i,j+\frac{1}{2}}-\left(\sqrt{G}u^2\right)_{i,j-\frac{1}{2}}}{h},$$

where $\left(\sqrt{G}u^1\right)_{i,j} = \left(\sqrt{G}\right)_{i,j}u^1_{i,j}$. Then we can get

$$\begin{aligned}(\nabla\cdot\mathbf{u})_{i,j} &= \frac{1}{\left(\sqrt{G}\right)_{i,j}}\left(\frac{\partial}{\partial x^1}\left(\sqrt{G}u^1\right)\right)_{i,j} \\ &\quad + \frac{1}{\left(\sqrt{G}\right)_{i,j}}\left(\frac{\partial}{\partial x^1}\left(\sqrt{G}u^1\right)\right)_{i,j}.\end{aligned}$$

The Laplacian can be calculated doing $\nabla^2\varphi = \nabla\cdot\nabla\varphi$.

V. Fluid simulation

An incompressible fluid is a velocity field $\mathbf{u}$ satisfying the Navier-Stokes equations:

$$\begin{cases}\dfrac{\partial\mathbf{u}}{\partial t} &= -\frac{1}{\rho}\nabla p - (\mathbf{u}\cdot\nabla)\mathbf{u} + \frac{1}{\rho}\nabla\cdot\left(\eta\left(\nabla\mathbf{u}+\nabla\mathbf{u}^T\right)\right)+\mathbf{f}, \\ \nabla\cdot\mathbf{u} &= 0\end{cases}$$

where p is the pressure, ρ is the fluid density, η is the viscosity coefficient and $\mathbf{f}$ is an external force. The first equation is called the momentum equation, and the second one is the incompressibility equation, which is the same to say that the fluid's volume is constant (consequence of Reynold's Transport Theorem.)

Here we will treat the fluids in a Eulerian viewpoint, where we look at quantities in the fluid at fixed points in space. Another option would be a Lagrangian viewpoint, where the fluid is viewed as a particle system, where each point is a separate particle with a position $\mathbf{x}$ and velocity $\mathbf{u}$ [1]. The Eulerian viewpoint was chosen because it is more suitable to the discretization scheme described in the last section.

The Navier-Stokes equations can be solved numerically by splitting, where it is divided into four equations:

$$\begin{aligned}\frac{\partial\mathbf{u}}{\partial t} &= -(\mathbf{u}\cdot\nabla)\mathbf{u} && \text{(advection)},\\ \frac{\partial\mathbf{u}}{\partial t} &= \frac{1}{\rho}\nabla\cdot\left(\eta\left(\nabla\mathbf{u}+\nabla\mathbf{u}^T\right)\right) && \text{(viscosity)},\\ \frac{\partial\mathbf{u}}{\partial t} &= \mathbf{f} && \text{(external forces)},\\ \frac{\partial\mathbf{u}}{\partial t} &= -\frac{1}{\rho}\nabla p, && \\ \text{such that } \nabla\cdot\mathbf{u} &= 0 && \text{(incompressibility)}.\end{aligned}$$

Temam [17] was the first to prove that this splitting scheme works. Let $\mathbf{u}^n$ be the solution of the Navier-Stokes equations at time $n\Delta t$. We start with a divergence-free velocity field $\mathbf{u}^0$, which is the initial condition for the equation. We calculate $\mathbf{u}^{n+1}$ using the values from $\mathbf{u}^n$. Each equation can be solved using a suitable algorithm, the result from one equation is given as input to another equation. This sequence of solutions must be done in a sequence, such that the output of one equation must satisfy the necessary conditions to the input of the next equation. For example to guarantee volume conservation the advection step must be done with a divergence-free velocity field, so this step must be calculated just after the calculation of incompressibility [1]. This was not followed in [4], where the advection was calculated before the incompressibility conditions, so the results there may be less accurate.

Given a divergence-free $\mathbf{u}^n$, we can start calculating the result $\mathbf{u}^A$ of advection. Observe that, from the advection equation, we get

$$\frac{\partial\mathbf{u}^A}{\partial t} = -(\mathbf{u}^n\cdot\nabla)\mathbf{u}^A = -(\bar{\mathbf{u}}^n\cdot\nabla_{\mathbb{R}^2})\mathbf{u}^A,$$

where $\bar{\mathbf{u}}^n = (u^1_n g^{11} + u^2_n g^{12}, u^1_n g^{12} + u^2_n g^{22})$, and $\nabla_{\mathbb{R}^2}$ is the gradient in $\mathbb{R}^2$. This is equivalent to an advection in $\mathbb{R}^2$ with velocity field $\bar{\mathbf{u}}^n$, which can be solved using a semi-Lagrangian technique, where we calculate the trajectory of each point using $\bar{\mathbf{u}}^n$ to find its position at the time $t-\Delta t$. This position can fall at any point of the domain, or even at a point in the domain of another patch. To get the velocity inside a domain Ω at an arbitrary position (i,j) we interpolate the values around this position for each component. For points outside the domain Ω, we look for a domain that contains this point, searching this point in the domain of one neighbour patch, always applying the transition function to get the coordinates of the point in the current patch. This process is repeated until we find a patch domain that contains the point. The velocity at position (i,j) is multiplied by the transition matrix from the original domain to the domain of the patch that contains this point, this matrix can be calculated by the sum s of every t_k from each visited patch domain, this sum results in the total number of rotations necessary to go from the original domain to the final domain, then the matrix is $M_{s\%4}$.

We can then use $\mathbf{u}^A$ as input for the next step, the addition of external forces. The equation $\frac{\partial \mathbf{u}}{\partial t} = \mathbf{f}$ can be discretized using a simple forward Euler: $\mathbf{u}^F = \mathbf{u}^A + \Delta t \mathbf{f}$. So, we just sum the values of the external forces to the current velocity.

When the fluid is viscous ($\eta > 0$) we need to solve the viscosity equation $\frac{\partial \mathbf{u}}{\partial t} = \frac{1}{\rho}\nabla \cdot \left(\eta \left(\nabla \mathbf{u} + \nabla \mathbf{u}^T\right)\right)$. In the planar (or volumetric) case, when η is constant this equation simplifies to $\frac{\partial \mathbf{u}}{\partial t} = \frac{\eta}{\rho}\nabla \cdot \nabla \mathbf{u}$, because we have $\nabla \cdot \nabla \mathbf{u}^T = \nabla(\nabla \cdot \mathbf{u}) = \nabla(0) = 0$. But in surfaces generally we have $\nabla \cdot \nabla \mathbf{u}^T \neq \nabla(\nabla \cdot \mathbf{u})$, so we can not make this simplification. In [4], this was not noticed, so the author used the simplified equation, which can be viewed as an approximation of the fluid viscosity.

The viscosity equation is discretized as

$$\left(I - \frac{\eta}{\rho}\Delta t A\right)\mathbf{u}^V = \mathbf{u}^F,$$

where I is the identity, and A is a discretization of the operator $\nabla \cdot \left(\nabla \mathbf{u} + \nabla \mathbf{u}^T\right)$, calculated using the discretization of the gradient and divergent operators. This is a linear system that can be solved using some simple iterative method. We only used constant values for η, but this scheme can also be applied for variable viscosity fluids.

According to Helmholtz-Hodge Decomposition Theorem we can decompose the velocity field into a curl-free component and a divergence-free component. To solve the incompressibility conditions, we calculate the divergence-free component of the velocity discretizing the equation $\frac{\partial \mathbf{u}}{\partial t} = -\frac{1}{\rho}\nabla p$, as

$$\mathbf{u}^P = \mathbf{u}^V - \frac{\Delta t}{\rho}\nabla p.$$

This is a projection of the current velocity into a divergence-free space. The pressure p can be obtained by solving the Poisson equation $\frac{\Delta t}{\rho}\nabla^2 p = \nabla \cdot \mathbf{u}^F$, which is a linear system that can be solved with some iterative method, improved with a multigrid technique. Defining $\varphi = \frac{\Delta t}{\rho}p$, this becomes simply $\nabla^2 \varphi = \nabla \cdot \mathbf{u}^F$, and the solution of projection is $\mathbf{u}^P = \mathbf{u}^F - \nabla \varphi$. Since this is the last step, we have $\mathbf{u}^{n+1} = \mathbf{u}^P$.

We can add a scalar field representing the concentration of particles moving through the velocity field, satisfying:

$$\frac{\partial s}{\partial t} = -(\mathbf{u} \cdot \nabla)s + \kappa \nabla^2 s + S$$

where s is the concentration, κ is a diffusion rate and S is source of concentration. This field can be used to visualize the fluid. To find this field we split its equation into three parts:

$$\frac{\partial s}{\partial t} = -(\mathbf{u} \cdot \nabla)s \qquad \text{(advection)},$$
$$\frac{\partial s}{\partial t} = \kappa \nabla^2 s \qquad \text{(diffusion)},$$
$$\frac{\partial s}{\partial t} = S \qquad \text{(sources)}.$$

We can start with the sources equation, which is similar to the external forces addition for velocity field. The equation is discretized by $s_1 = s_0 + S\Delta t$.

The next step is diffusion, which can be discretized by $(I - \Delta t \kappa \nabla^2)s_2 = s_1$, forming a linear system of equations with a sparse matrix, whose solution can be found (or approximated) using an iterative method.

The last step is the advection, observe that

$$\frac{\partial s}{\partial t} = -(\mathbf{u} \cdot \nabla)s = -(\bar{\mathbf{u}} \cdot \nabla_{\mathbb{R}^2})s,$$

where $\bar{\mathbf{u}} = (u^1 g^{11} + u^2 g^{12}, u^1 g^{12} + u^2 g^{22})$. So we advect s using the velocity field given by $\bar{\mathbf{u}}$.

VI. 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. One possibility is to use the processors of a graphics processing unit (GPU). We implemented the method in the GPU using CUDA.

A. 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 $w \times h \times n\_patches$, where $n\_patches$ is the number of patches of the surface, such that the value `g(i,j,p)` at position (i, j) and patch $p \in [0, \cdots, n\_patches - 1]$ is accessed via `g[i + j*w + p*w*h]`. For scalar fields $w = h = N + 2$, and the value $\varphi^p_{i,j}$ is stored at `phi(i,j,p)` For vector fields, the first coordinate uses $w = N+3, h = N+2$ and the second coordinate uses $w = N+2, h = N+3$. So `u1(i,j,p)` stores the value $(u^1)^p_{i-0.5,j}$ of the first coordinate of a vector field $\mathbf{u}$, and `u2(i,j,p)` stores the value $(u^2)^p_{i,j-0.5}$ of the second coordinate of $\mathbf{u}$. For the metric we use $w = h = 2N + 3$ to store the values $\sqrt{g}, g^{11}, g^{12}$ and g^{22}. The value $(\sqrt{g})^p_{i,j}$ is accessed via `g(2*i, 2*j, 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 to 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 texture/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)` of a grid in texture memory is accessed via

```
tex2DLayered(tex_ref, i+0.5, j+0.5, p)
```

where `tex_ref` is a texture reference bound to some CUDA array. The sum with 0.5 is necessary to align the grid positions with texture coordinates. We use non-normalized texture coordinates, with linear filtering. So if the value i is any floating-point number between 0 and $w-1$, and j between 0 and $h-1$, then the result is a bilinear interpolation of the four neighbour grid points around this position.

For a grid in surface memory, `g(i,j,p)` is accessed via

```
surf2DLayeredread(&a, surf_ref, i*4, j, p)
```

where 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).
```

B. Precomputing

The surface evaluation needs to be calculated only once, we compute 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 [6]. Each point on the surface is given 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 eigen-space of the Catmull-Clark subdivision matrix, φ 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 depend only 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 [6], 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.

C. Solving equations

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 Navier-Stokes 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. When we are processing scalar fields, each kernel will process the point (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, choosing properly the block dimensions.

For the velocity field each kernel of coordinates (i, j, p) processes the values $(u^1)_{i-0.5,j}^p$ and $(u^2)_{j,i-0.5}^p$, where $i \in [1, \cdots, N+1]$ and $j \in [1, \cdots, N]$.

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 values t_k are stored each in an array of size $4 * n\_patches$, created when the surface was constructed. Then the neighbour patch index of patch p at edge $e \in \{0, 1, 2, 3\}$ is accessed by doing `neigh_indices[p*4 + e]`. 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.

To calculate the advection step for the concentration and velocity fields, we calculate the field $\bar{\mathbf{u}}$ of the velocity modified by the metric as we saw before, and put it in the current velocity field in the texture memory, we also store a copy of the current velocity and the current concentration s in the texture memory. For each position of the velocity or concentration we calculate the trajectory of a particle traveling according to the velocity $\bar{\mathbf{u}}$. In the calculation of this trajectory, the point can fall in an arbitrary position, where we the value of the velocity or concentration is calculated efficiently by the GPU using its texture fetching units. Generally most of the points fall in a nearby location, so the texture access is optimized using the texture cache. When a point falls in a different patch we may loose a bit of the performance, since threads in the same warp may have different execution paths.

The addition of external forces is a simple operation, where we get the forces defined by some function, and just sum them to the current velocity multiplied by the time variation. Similarly we add concentrations from sources, but in this case we limit the values to avoid concentrations bigger then 100%.

For the viscosity step, we use an iterative method to solve the linear system. In the GPU, the velocity values are updated in parallel, then to avoid conflicts with reading/writing operations, in each thread we calculate the new velocity value using the current velocity field, and we call `__syncthreads()` to make sure that all other threads had already calculated their corresponding new values so we can safely update the field.

For the projection step we find φ (a scalar field) that satisfies $\nabla^2\varphi = \nabla \cdot \mathbf{u}$, using a multigrid v-cycle scheme with Jacobi iterations [18]. We run some iterations to calculate an approximation φ_A of φ in the highest level, improve this result summing it with the error e that satisfies $\nabla^2 e = \nabla \cdot \mathbf{u} - \nabla^2\varphi_A$. The error e is calculated in a lower level, where the grid size is smaller then the grid size of the highest level. Again we run some iterations to find an approximation of e and improve it with the error of this approximation, calculated in a even smaller level. This process of improving the error calculation is repeated until we reach the lowest level, when $N = 1$, and then the result of one level is summed to the next level and improved with more iterations until we come back to the highest level, where we finally get φ. The most computing intensive step is the calculation of the Jacobi iterations, that must be done for each point of the grid of all the patches. But it can be easily parallelized since the operation for each point is exactly the same. So we created a kernel to calculate the Jacobi iteration to improve the approximation for the current level. After each iteration we must update the neighbour cells to keep the result consistent. After finding φ we run a kernel that updates the velocity subtracting $\nabla\varphi$ from the current $\mathbf{u}$.

For the diffusion of concentration, generally it is sufficient to do some Jacobi iterations, but if a more precise result is desired it is possible to use a multigrid scheme similar to the one used in the projection step.

VII. Results

In our tests we used an NVIDIA GeForce GTX 470, which has 448 CUDA cores. We implemented some forces, like the gravity force, also used in [4], which is proportional to the concentration s and the projection of the downward direction into the tangent plane at each surface point, and a force similar to something "walking" on the surface, following a curve and pushing the fluid with a force tangent to this curve.

For visualization of the fluid, we mapped the concentration values to colors, assigning one color for the 0% concentration, another one for the 100% concentration, and interpolating these colors for intermediate concentration values.

In Figure 4 we can see the result at two different steps with a toroidal surface, where we put sources of concentration at the center of each patch. In the left we see one of the first steps of the simulation, and in the right we see the result after several steps, also changing the position of the surface (in the gravity force calculation, the downward direction is relative to the viewer, so it changes in relation to the surface as we move it).

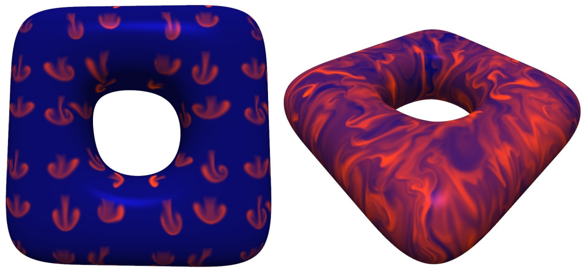
Fig. 4. Toroidal surface.

For Figure 5 we used a quadrangulation of the Stanford bunny, the initial concentration is shown in the left, and in the right picture we can see the result after some steps of the algorithm, using the gravity force.

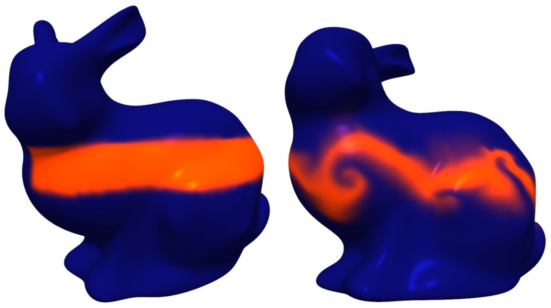
Fig. 5. Two steps of the simulation using the bunny.

In Figure 6 we see for two surfaces the result after some steps using forces "walking" in circular paths at each patch.

In Table I we can compare the time taken for one full step (including all substeps) in the gpu and in the cpu (with a 2.80GHz processor) for each surface we tested. We executed the simulation with the same parameters for all surfaces, only changing the resolution of the grids. We can see how the gpu implementation is much faster than the cpu implementation. 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 the denser meshes (like bunny). However for dense meshes it is usually sufficient to use a small grid size. In most of the cases it is not even necessary to use more than a few hundreds of patches. The fertility surface we used was created using a 3D modeling tool, it only uses 166 patches, but it is a good approximation of the well known triangular mesh.

In Table II we see how long each sub-step takes in the simulation. It is based on the simulation using the fertility model (166 patches), with the circular forces, and $N = 32$. We show all steps in the order we run them, including the update of the texture, which is used only for visualization. We can see that the most expensive steps are the diffusion, viscosity and projection, taking more than 80% of the total simulation. In most of the cases, we work with inviscid fluids (without viscosity), so this step is not a big problem. We can reduce the number of iterations used in the projection and

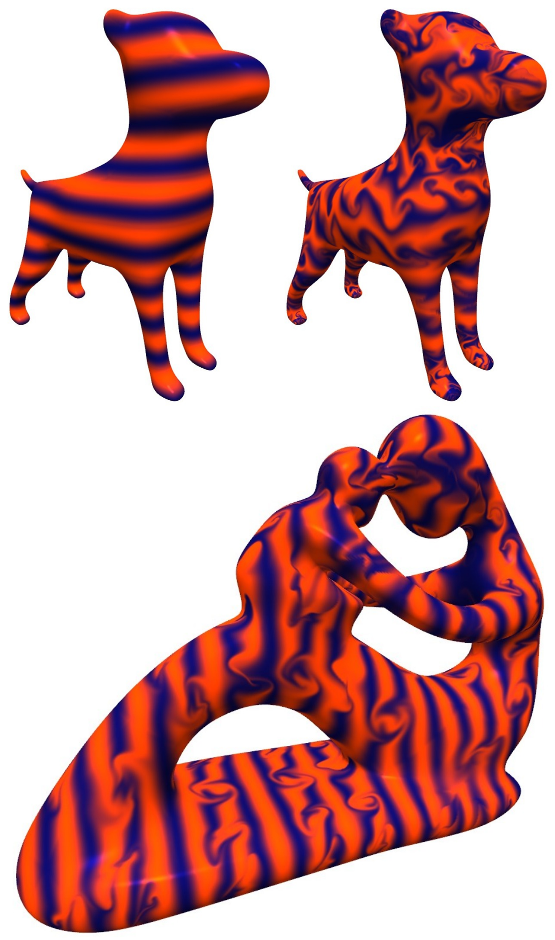

Fig. 6. Forces following circular paths on the dog and fertility models.

surface ($n\_patches$)	$N = 8$	$N = 16$	$N = 32$	$N = 64$
toroidal (128)	14/ 106	24/ 378	54/ 1511	152/ 6209
fertility (166)	16/ 144	28/ 517	70/ 2078	193/ 8574
dog (238)	20/ 228	40/ 828	94/ 3413	278/ 16169
bunny (1292)	93/ 3247	181/ 15227	469/ 63440	—

TABLE I
TIME TAKEN IN MILISECONDS FOR ONE FULL STEP FOR EACH SURFACE TESTED IN GPU/CPU.

diffusion steps, so that it takes a smaller time to be computed, but this also reduces the precision of the method. Changing this parameter we can balance quality and performance as desired.

VIII. CONCLUSION AND FUTURE WORKS

In this work we implemented fluid simulation on surfaces using the GPU. We used suitable structures to take advantage of the GPU resources, increasing the performance of the simulation. For future works we may study different solvers to improve even more the method, specially for the projection method, which is the most computationally expensive step. We can also study how this scheme can be applied to solve other types of equations, like reaction-diffusion systems for natural patterns generation, or any other problems that are usually solved in two dimensions, which we can extend to work on surfaces.

Step	**Average time**	**Percent**
Add forces	0.28ms	0.39%
Viscosity	19ms	26.58%
Projection	32.48ms	45.45%
Add sources (s)	0.08ms	0.11%
Advection (s)	2.01ms	2.81%
Diffusion (s)	10.8ms	15.11%
Update texture	1.81ms	2.53%
Advection	5.01ms	7.01%
Total	71.47ms	100.00%

TABLE II
TIME TAKEN FOR EACH SUB-STEP.

REFERENCES

[1] R. Bridson, *Fluid Simulation for Computer Graphics*. A K Peters/CRC Press, Sep. 2008.
[2] J. Stam, “Stable fluids,” in *Proceedings of SIGGRAPH 99*, ser. Computer Graphics Proceedings, Annual Conference Series, Aug. 1999, pp. 121–128.
[3] R. Fedkiw, J. Stam, and H. W. Jensen, “Visual simulation of smoke,” in *Proceedings of ACM SIGGRAPH 2001*, ser. Computer Graphics Proceedings, Annual Conference Series, Aug. 2001, pp. 15–22.
[4] J. Stam, “Flows on surfaces of arbitrary topology,” *ACM Trans. Graph.*, vol. 22, no. 3, pp. 724–731, 2003.
[5] E. Catmull and J. Clark, “Recursively generated b-spline surfaces on arbitrary topological meshes,” *Computer-aided Design*, vol. 10, pp. 350–355, 1978.
[6] J. Stam, “Exact evaluation of catmull-clark subdivision surfaces at arbitrary parameter values,” in *Proceedings of SIGGRAPH*, 1998, pp. 395–404.
[7] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard, “Cg: a system for programming graphics hardware in a c-like language,” *ACM Trans. Graph.*, vol. 22, no. 3, pp. 896–907, Jul. 2003.
[8] NVIDIA, *NVIDIA CUDA Programming Guide 4.1*, 2011.
[9] F. Randima, *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Addison-Wesley Professional (April 1, 2004), 2004.
[10] N. Chentanez and M. Müller, “Real-time eulerian water simulation using a restricted tall cell grid,” *ACM Trans. Graph.*, vol. 30, no. 4, pp. 82:1–82:10, Aug. 2011.
[11] W. J. van der Laan, S. Green, and M. Sainz, “Screen space fluid rendering with curvature flow,” in *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, ser. I3D ’09. New York, NY, USA: ACM, 2009, pp. 91–98.
[12] J. J. Monaghan, “Smoothed particle hydrodynamics,” *Annual Review of Astronomy and Astrophysics*, vol. 30, pp. 543–574, 1992.
[13] C. E. Scheidegger, J. Comba, and R. D. da Cunha, “Practical cfd simulations on programmable graphics hardware using smac,” *Computer Graphics Forum*, vol. 24, no. 4, pp. 715–728, 2005.
[14] K. Hegeman, M. Ashikhmin, H. Wang, H. Qin, and X. Gu”, “Gpu-based conformal flow on surfaces,” *Comunications in information and systems*, vol. 9, pp. 197–212, 2009.
[15] R. Aris, *Vectors, Tensors and the Basic Equations of Fluid Mechanics*. Dover Publications, 1989.
[16] F. H. Harlow and J. E. Welch, “Numerical Calculation of Time-Dependent Viscous Incompressible Flow of Fluid with Free Surface,” *Physics of Fluids*, vol. 8, no. 12, pp. 2182–2189, 1965.
[17] R. Temam, “Sur l’approximation de la solution des équations de Navier-Stokes par la méthode des pas fractionnaires II,” *Arch. Rat. Mech. Anal.*, vol. 33, pp. 377–385, 1969.
[18] D. Kincaid and E. Cheney, *Numerical Analysis: Mathematics of Scientific Computing*, ser. Pure and applied undergraduate texts. American Mathematical Society, 2002.