# A GPU-based method for approximate real-time fluid flow simulation

Tomasz Rożen, Krzysztof Boryczko, Witold Alda

*University of Science and Technology AGH*
*Institute of Computer Science*
*al. Mickiewicza 30, 30-059 Kraków, Poland*

**Abstract**

Fluid flow can be realistically simulated by physical models. We present a method for simplifying the Navier-Stokes equations by relaxing the incompressibility constraint. Our method allows for low-cost real-time simulation of two-dimensional fluid flow with enough accuracy for computer graphics. The implementation takes advantage of recent programmable floating-point graphics hardware that performs all the necessary computations.

*Key words:*
fluid dynamics, graphics hardware, physically-based simulation,
pseudo-compressibility method

## 1 Introduction

Fluid behaviour is an intriguing phenomenon which has captured the attention of researchers for years. Due to its universal presence in various forms, e.g. smoke, fire, wind, cloud formation, weather, water movement, ocean waves, any graphics system designed to produce realistic images and/or animation requires a good fluid solver. It plays also important role in other fields, like realistic texture synthesis, paint programs emulating traditional painting methods (such as watercolour) as well as scientific simulations and engineering. However simple and ordinary these phenomena may seem their realistic simulation presents a complex and challenging task. The reason behind this is the coexistence of many physical processes such as advection, diffusion, gravitation,

*Email address:* rozen@agh.edu.pl (Tomasz Rożen).

turbulence and surface tension. Computational Fluid Dynamics (CFD), despite its long history, is still an active field of development.
Usually fluid simulation is conducted off-line and then visualised as a post-process. However, real-time graphics application developers would also like to take advantage of these phenomena. This is why numerous approaches have been taken to create simplified and less accurate models yet still producing realistic and visually compelling outputs. Most techniques are based on direct simulation of the incompressible Navier-Stokes equation or, recently, by means of a semi-Lagrangian treatment introduced by Stam [5]. Although steps have been taken to animate full 3D fluid behaviour these methods are restricted to small-scale environments and usually require full processor time. Further animation cost reduction can be accomplished by restricting the class of problems to 2D fluid flow only, while still producing full three dimensional graphics. The idea behind this is to utilize pressure field to generate fluid surface [1]. An alternative approach has been proposed by Klein et al. [7] where water flow is coupled with a shallow-water equation [3] to produce fluid surface. In this paper we present a low-cost approximate method for simulating 2D fluid flow, with emphasis on real-time computation, which then may be used to produce fluid surface. We utilize the power of graphics hardware to perform the necessary calculations.
The Graphics Processing Unit (GPU) is a low-cost off-the-shelf 3D graphics card designed to be extremely fast at processing polygons and pixels for rendering. In the past decade we have witnessed constant improvement in performance (on average it doubled every twelve months), much faster than that of traditional CPUs. With the recent introduction of floating-point calculations and high-level programming languages it has become a standalone processing unit able to perform arbitrary computational task, not limited to computer graphics, and has drawn much attention in various fields, e.g. simulations [10], global illumination [11], database operations [9] and linear algebra [18]. Because of its highly parallel architecture and stream processing model GPUs are well suited for fluid simulation where data (velocity and pressure) is stored in textures located in video memory and computations are performed by fragment shaders. Our method takes full advantage of graphics hardware to achieve real-time 2D simulation of incompressible fluid flow.

## 2 Approximate Navier-Stokes

The fundamental equations in fluid mechanics are the Navier-Stokes equations, which describe the dynamic behaviour of a viscous fluid. In this section we assume that fluid density and temperature are constant in space and time thus it can be completely described by its velocity $\mathbf{u}$ and pressure field $p$. By imposing that the fluid conserves both mass and momentum we obtain the

incompressible form of the Navier-Stokes equations:

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\,\mathbf{u} - \frac{1}{\rho}\nabla p + \nu \nabla^2 \mathbf{u} + \mathbf{f} \tag{1}$$

$$\nabla \cdot \mathbf{u} = 0 \tag{2}$$

Where $\nu$ is the kinematic viscosity of the fluid, $\rho$ is its density and $\mathbf{f}$ is the external force. $\nabla$ is the vector of spatial derivatives; $\nabla = (\partial/\partial x, \partial/\partial y)$ in two dimensions. For a detailed derivation of this equations we refer the reader to any of the standard text books on fluid mechanics e.g. [13]. In general there is no analytic solution so numerical methods have to be employed.
The solution to the Navier-Stokes equations is computed in two alternating steps. We begin with equation (1) solved by means of central differences on a staggered marker and cell grid. Although there are more advanced methods of solving partial differential equations (PDEs) this approach has proved adequate for the needs of our simulation; its simplicity allows for straightforward and efficient implementation on a resource-limited architecture, like the GPU. Secondly, the Poisson equation is derived for the pressure field. By computing gradient of equation (1) and applying mass conservation condition (2) we obtain:

$$\nabla^2 p = -\nabla\,(\mathbf{u} \cdot \nabla \mathbf{u}) \tag{3}$$

This equation, when spatially discretized, becomes a sparse linear system. Solving this system of equations can become a bottleneck for real-time applications; most efficient methods, like the multigrid solver, offer linear complexity. Furthermore these are iterative methods where the number of steps may vary, being faster when the field is close to divergent free. Chen et al. [1] proposed an alternative approach to enforcing mass conservation. Instead of solving the sparse linear system resulting from Poisson equation, they approximated it with a penalty method (for a survey of pseudo-compressibility methods refer to [16]), replacing equation (2) with:

$$\epsilon p + \nabla \mathbf{u} = 0 \tag{4}$$

Temam [14] proved, that equations (1) and (4) with $\epsilon \to 0$, $\epsilon > 0$ tend towards the solution of Navier-Stokes equations for incompressible fluids. In this method the incompressibility constraint is lifted and the computed values of $\mathbf{u}$ and $p$ are approximations of the original values. Moreover $p$ can be eliminated completely from equation (1).
Chen used the above equation to compute fluid pressure and then generate a height field representing fluid surface. The downside of this approach is that it slowly converges to the incompressible form when not in stationary state,

resulting in unnatural fluid behaviour. Producing good results requires small time step $dt$ and penalty parameter $\epsilon$, Chen's values were 0.001s and 0.005 respectively; larger values may lead to numerical instability.
We propose a different method of approximating the Navier-Stokes equation, relaxing slightly the incompressibility condition with:

$$p = (1 - \gamma)\,\tilde{p} - \frac{1}{\epsilon}\,(\nabla \cdot \mathbf{u}) \tag{5}$$

Where $\tilde{p}$ is the pressure field from previous simulation step, gamma is the damping factor, $\gamma > 0$, $\gamma \to 0$ and eps is penalty parameter, $\epsilon > 0$, $\epsilon \to 0$. The damping factor gamma is required because of precision of numerical calculation - for turbulent flow where $\gamma = 0$ the computations may 'explode'; for $\gamma \to 1$ the simulation converges slowly. Tuning $\gamma$ and $\epsilon$ may be an challenging task, which depends on the flow parameters and environment condition.
Equation (5) is similar to what Temam proposed in [15]:

$$\nabla \mathbf{u} + \epsilon \frac{\partial p}{\partial t} = 0 \tag{6}$$

Which, when finite difference method is applied, becomes:

$$p = \tilde{p} - \frac{1}{\epsilon}\,(\nabla \mathbf{u})\,dt \tag{7}$$

However our equation, when solved numerically, possesses greater stability and faster convergence, even for larger time steps. It can be shown, that equations (1) and (5) tend towards the solution of Navier-Stokes equation for $\epsilon \to 0$. When gamma equals 0 or 1 equation (5) becomes equation (7) or (4) respectively. The former and the latter equations have been proven by Temam.
The system (1) and (5) should be supplemented with initial conditions for velocity $u_0$ and pressure $p_0$ and appropriate boundary conditions. In this paper, for the sake of simplicity, we consider only fixed boundary conditions $\partial\Omega$ for a fluid laying in some bounded domain $\Omega$. The boundary conditions should be such, that there should be no flow through the walls i.e. the normal component of flow velocity should be zero at boundaries. The stability of our method has been empirically tested on various initial and boundary conditions, simulating a spectrum of different flows, including flow in a pipe, flow through a dam, flow around a block. In each case the simulation was run for minutes observing numerical stability and measuring the largest possible time step. A simulations is considered unstable when small variations on the initial conditions cause the numerical solution to expose high variations after some time $t$. Our simulation proved to be stable, provided that the parameters are chosen appropriately. In general for smaller time step, flow speed, Reynolds number and larger penalty parameters the solution is stable. The accuracy of

the computed flow is sufficient for real-time applications, it appears natural and emulates many natural phenomena e.g. vortices.
To solve the Navier-Stokes equations we apply explicit time integration methods. The major disadvantage of such method is the severe restriction imposed on the time step size. However the simplicity of these methods, as well as the low computation cost of the pseudo-compressibility approximation, allow for efficient implementation. This results in multiple simulation steps per second making a good trade-off for the small time step.
In the next section we give a through account of the GPU implementation.

## 3 GPU implementation

We have successfully implemented our method on an ATI Radeon 9700 and nVidia GeForce 6 graphics cards. All numerical calculation of our fluid solver are performed on the programmable, floating-point graphics hardware utilizing the rendering pipeline. Simulation data is stored in high speed video memory. Developing general purpose programs for GPU requires knowledge of the specific field as well as the details of computer graphics which are explained briefly hereunder.
Rendering a 3D scene involves a fixed sequence of steps, some of which may be executed in parallel. These are known as the graphics pipeline. In the first stage user supplied data representing a polygonal mesh is transformed from abstract 3D world-space coordinates to 2D screen coordinates and a depth value, indicating the distance between a virtual camera and a graphical primitive. This is known as the vertex processing stage. Beside position, a vertex may have other properties, like texture coordinates and colour or even arbitrary data, which are also transformed by the vertex shading unit. Groups of vertices form rendering primitives, e.g. triangles.
Screen space converted primitives are then rasterized during the second step producing a set of fragments. Data from vertices are interpolated to provide fragments with the necessary information to update pixels. Each fragment is processed according to a set of rules either fixed or programmable. This step is know as fragment processing. In this stage textures are sampled and the resulting texels are fetched into the fragment processor. The input data is transformed, including various operations, to produce the resulting colour. At the end of the pipeline each fragment is tested against some values, e.g. the current depth of fragments in the frame buffer, to decide whether or not a fragment is updated. Programmability can be introduced at two stages, vertex and fragment processing, by uploading user defined programs, either in assembly language or high-level graphics programming language. The latter is represented by OpenGL Shading Language. Most non-graphic applications of GPU take advantage of fragment pipeline ignoring the vertex processor. The

reason behind this is the vertex processing unit possesses knowledge only of the one vertex being processed. The fragment shader has no such limitations, on the contrary, it can access arbitrary texels or even performing dependent texture reads. In order to process large number of primitives the graphics hardware exposes a highly parallel architecture. Vertices and fragments are stream processed by one of multiple units in SIMD model. The graphics hardware enforces independent processing of each primitive resulting in stall-less processing.
We employed a stream programming model in our application making a distinction between simulation data and computational kernels. The former are stored in video memory as floating point textures. As our simulation has been performed only in two dimensions there is a direct mapping between texels and grid cells. The values of $u$, $v$ and $p$ were stored in different channels of a single texture, representing a staggered grid where pressure is defined at the centre of cells while velocity is defined at cells faces. Fragment programs serve the purpose of computation kernels, performing computations by means of rendering to texture. A single quadrilateral covering the whole viewport is drawn to update the state texture, which in turn may be used to stream data to the next fragment program. In our method the equations (1) and (5) described in previous section are implemented as fragment programs.
The overall algorithm can be divided into the following steps:

(1) Set initial conditions.
(2) Apply boundary conditions.
(3) Compute $u$ and $v$ by eqn. (1) (store in red and green channel)
(4) Copy computed $u$ and $v$ to previous texture
(5) Compute $p$ by eqn. (5) (store in alpha channel)
(6) Repeat steps 2 to 5

Two textures are used to represent the current and previous step of simulation. Step 4 is necessary because on current hardware reading and writing to the same texture in a single fragment shader may result in undefined behaviour. This restriction may be lifted in the future.
In our implementation we used OpenGL for graphics rendering with various extensions: ARB_shading_language_100 for fragment program compilation and linking, ARB_render_texture and ARB_pbuffer for rendering to off-screen buffers. To access floating point textures we employed the ATI_texture_float extension which is more versatile than NV_float_buffer and is available on both ATI and nVidia hardware (staring with GeForce series 6).
Our research focused on computing fluid flow with application in real-time which could be used to generate a fluid surface. However, due to the relaxation of incompressibility constrain, a way of enforcing mass conservation had to be found. Instead one could adopt Klein et al. [7] proposition of handling fluid flow and surface independently. They employed the so-called shallow water equation to generate the surface, where the results of the computed flow

are affecting wave generation. The latest hardware supports texture fetching in the vertex program which could be used when generating polygonal mesh for fluid surface. This would result in GPU-only implementation of fluid surface, avoiding the huge costs of copying data between system and video memory.
For testing purposes a traditional CPU based application has also been developed. The next section describes a direct GPU-CPU comparison as well as the results from various graphic cards.

## 4 Results and analysis

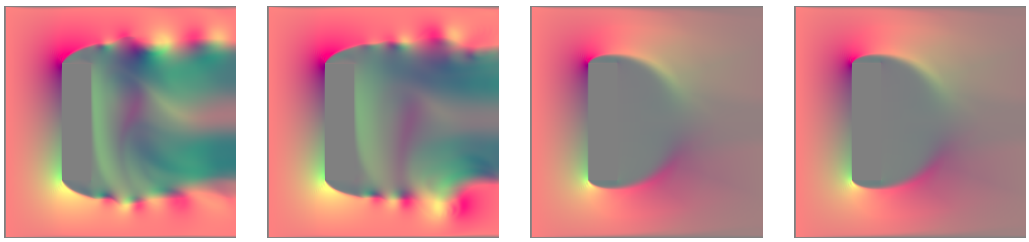
Here we present the results of our technique and its effectiveness. For the CPU implementation we used a Dell computer system with 2.8 GHz Intel Pentium processor and 1GB of memory. The graphics chips were the ATI FireGL V3100 with core frequency of 400 MHz and 128 MB of video memory. For comparison we also made the experiment with an nVidia GeForce 6600GT with 500 MHz core frequency and also 128 MB of video memory. The time values account only for simulation time and do no include rendering time nor the time required for copying data between the system and graphics processor in CPU version.
Table 1 gives the speed comparison between the CPU-Pentium and the GPU-FireGL implementation for various grid sizes.

Table 1
Time of a single simulation frame in ms

| Grid scale | FireGL | CPU | FireGL/CPU Speedup | GeForce |
|---|---|---|---|---|
| 64 | 1.11 | 1.92 | 1.74 | 0.65 |
| 128 | 1.13 | 7.08 | 6.26 | 1.17 |
| 256 | 3.34 | 32.29 | 9.66 | 4.41 |
| 512 | 12.7 | 225.89 | 17.76 | 17.06 |
| 1024 | 31.94 | 1015.57 | 31.79 | 67.80 |

As one can see the performance of the GPU is superior over that of the CPU by a factor of up to 32. Due to its architecture performance gain on graphics hardware is even bigger for larger grids. In our implementations all computations were done on the graphics chip and the results are stored in video memory and then rendered right to the frame buffer. One could even use a fragment program to generate vertex data for 3D fluid surface. With OpenGL it is possible to copy data between texture memory and vertex buffers. This way we can avoid copying data between graphics and system memory which is know as the biggest bottleneck for many graphics applications.
It has been observed that the impact of various boundary conditions as well

as simulation parameters on simulation time can be neglected. For simulation cases where only a part of the grid has to be updated one could employ early-z culling [8] or similar method. However this improvement for our setup would be of minor benefit.
The rightmost column of table 1 shows the results from nVidia GeForce 6600GT graphics card. Strangely the results are much worse than those from FireGL board. The former has higher theoretical memory bandwidth as well as a 25% higher processor clock rate. Unfortunately we have not been able to point out the exact cause of such low performance. Both graphics cards run the same version of our application including the fragment programs. We believe that it may be connected with the use of ATI floating point render target OpenGL extension on nVidia hardware.



a) b) c) d)

Fig. 1. Comparision of results of simulation for a) 32-bit nVidia b) 32-bit ATI c) 16-bit nVidia d) 16-bit ATI

It is important to note that the fragment programs are compiled and processed by the graphics hardware and display drivers. For faster graphics processing a 24bit floating-point precision may replace the full IEEE 32-bit precision. Moreover fragment program compiler may decide to introduce optimizations changing the behaviour of the given program. This statement can be extended on other elements of the graphics pipeline as well.

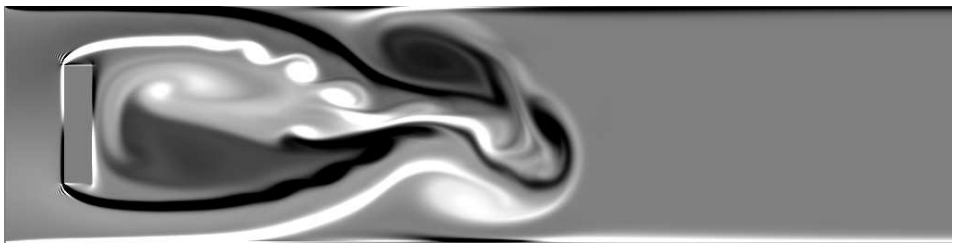
Table 2
Simulation frame rate

| FP precision | nVidia | ATI |
|---|---|---|
| 32bit | 814 | 1002 |
| 16bit | 1698 | 1058 |

Figure 1 shows the comparison of results from simulation on nVidia and ATI hardware. The images show the actual simulation data stored in texture, velocity is packed into red and green channels for horizontal and vertical component respectively (the values are biased by 0.5). Results after 50000 time steps differ significantly (a and b). We conducted another experiment this time using 16bit floating-point simulation data (figure 1 c and d). Interestingly the outputs differ only slightly. We believe that this is because of the optimizations mentioned before. Table 2 shows frame rates of those experiments which seem

to prove this statement.
Figure 2 shows a few frames from our real-time simulation, showing fluid flowing in a pipe with an obstacle. These images show the vorticity of fluid flow, $\omega = \nabla \times \mathbf{u}$ (biased so, that the grey colour represents zero vorticity).

a)



b)

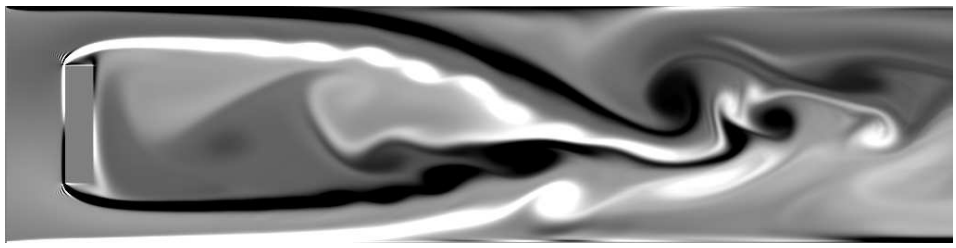


Fig. 2. Two subsequent snapshots from the simulation on a 256x1024 grid

## 5 Conclusions and future work

In this paper we have shown a way to simulate fluid flow totally on the graphics hardware, which in recent years have become a fully functional processing unit. The whole computations and data is stored on the video memory so no CPU-to-GPU computation is required. Additionally we have presented a approximate method which may be used in real-time graphical applications.
For now we have only considered 2D fluid dynamics problems which we would like to extend to the 3D domain. Another problem was the small time step (1/30s) which had to be employed. We would like to implement the semi-Lagrangian method, as in [5], to make this method more versatile and stable.

## References


[1] J. X. Chen, N. da Vittoria Lobo, C. E. Hughes and J. M. Moshell. Real-time fluid simulation in a dynamic virtual environment. *IEEE Computer Graphics and Applications*, pp. 52-61, May-June 1997.

[2] Anita T. Layton, Michiel van de Pane. A numerically efficient and stable algorithm for animating water waves. *The Visual Computer*. January 2002.

[3] Michael Kass, Gavin Miller. Rapid, stable fluid dynamics for computer graphics. *Computer Graphics*, Volume 24, Number 4, August 1990.

[4] James F. O'Brien, Jessica H. Hodgins. Dynamic simulation of splashing fluids. *Proceedings of Computer Animation '95*, pp. 198-205, April 1995.

[5] Jos Stam. Stable Fluids. *Proceedings of SIGGRAPH 1999*, pp 121-128

[6] Matthias Muller, David Charypar, Markus Gross. Particle based fluid simulation for interactive application. *Eurographics/SIGGRAPH Symposium on Computer Animation* (2003).

[7] Thomas Klein, Mike Eissele, Daniel Weiskopf, Thomas Ertl. Simulation, Modelling and Rendering of Incompressible Fluids in Real Time. *Workshop on Vision, Modelling, and Visualization VMV '03* , pages 365-373.

[8] Pedro V. Sander, Natalya Tatarczuk, Jason L. Mitchell. Explicit Early-Z Culling for Efficient Fluid Flow Simulation and Rendering. *ATI Research Technical Report*, August 2, 2004.

[9] Naga K. Govindaraju, Brandon Lloyd, Wei Wang, Ming Lin, Dinesh Manocha. Fast Computation of Database Operations using Graphics Processors. *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*.

[10] Zhe Fan, Feng Qiu, Arie Kaufman, Suzanne Yoakum-Stover. GPU Cluster for High Performance Computing. *ACM / IEEE Supercomputing Conference 2004*, November 06-12, Pittsburgh, PA.

[11] Nathan A. Carr, Jesse D. Hall and John C. Hart. GPU Algorithms for Radiosity and Subsurface Scattering. *Graphics Hardware* (2003).

[12] Kenneth Moreland and Edward Angel. The FFT on a GPU. *Graphics Hardware* (2003).

[13] Frank M. White. *Fluid Mechanics*. McGraw-Hill 2001.

[14] Roger Temam. *Bulletin de la Societe Mathematique de France*, Gauthier-Villars, Paris, 1968, pp. 115-152.

[15] Roger Temam. Sur l'approximation de la solution des equations de Navier-Stokes par la methode des pas fractionnaires I. *Arch. Rat. Mech. Anal. 32 (1969)*, pp. 135-153.

[16] Jie Shen. Pseudo-compressibility methods for the unsteady incompressible Navier-Stokes equations. *Proceedings of the 1994 Beijing Symposium on Nonlinear Evolution Equations and Infinite Dynamical Systems*, 68–78, Ed. Boling Guo, ZhongShan University Press, 1997.

[17] Enhua Wu, Youquan Liu, Xuehui Liu. An improved study of real-time fluid simulation on GPU. *The Journal of Computer Animation and Virtual World*, july 2004.

[18] Jeff Bolz Ian Farmer Eitan Grinspun Peter Schroder. Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. *In Proceedings of ACM SIGGRAPH 2003*.