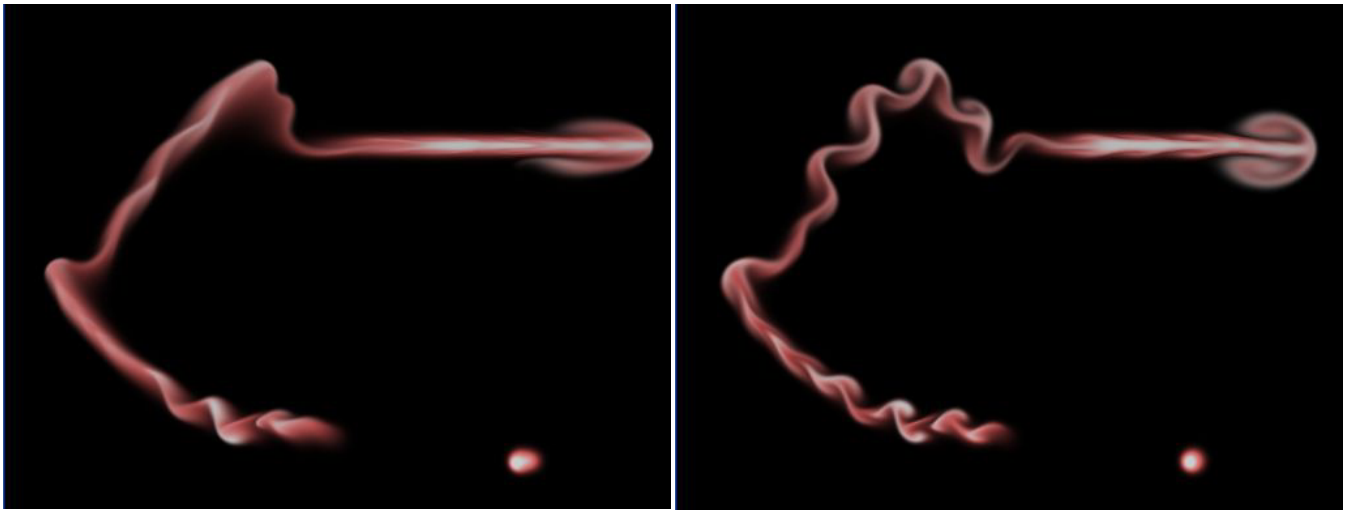
Explicit Early-Z Culling for Efficient Fluid Flow Simulation and Rendering

ATI Research Technical Report
August 2, 2004

Pedro V. Sander
ATI Research, Inc.
62 Forest Street
Marlborough, MA 01752, USA

Natalya Tatarchuk
ATI Research, Inc.
62 Forest Street
Marlborough, MA 01752, USA

Jason L. Mitchell
ATI Research, Inc.
62 Forest Street
Marlborough, MA 01752, USA



(a) Brute force **(b) Early-z**

Figure 1: Comparison of fluid flow simulations with and without our early-z acceleration techniques. Both simulations use a 512×512 grid (cropped for the figure) and render at 53fps.

Abstract

We present an efficient algorithm for simulation and rendering of fluid flow on graphics hardware. Our algorithm takes advantage of explicit early-z culling to reduce the amount of computation during both the simulation and rendering steps. Our approach is straightforward to implement and speeds up our simulations in some cases by a factor of three.

1. Introduction

In recent years, graphics processors have been applied to broader areas of computation, such as simulation of natural phenomena. Due to their highly parallel nature and increasingly general computational models, GPUs are well matched with the demands of fluid flow simulation.

In this paper we present acceleration techniques for simulation and rendering of fluid flow. We have implemented a fluid simulation on the GPU based on the solution of Navier-Stokes equations which uses explicit early-z culling as a means of avoiding certain unnecessary computations. More specifically, we present a culling technique for the projection step of the fluid flow simulation and a culling technique for efficiently rendering fluid flow to the screen.

Fluid flow simulation naturally maps to current graphics hardware by performing the different steps of the simulation using full screen quadrilaterals and doing all the work in pixel shaders (see Harris [2003b] for a thorough description). Prior to execution of a pixel shader, the graphics hardware performs a check of the interpolated z value against the z value in the z buffer. This occurs for any pixels which are actually going to use the primitive's interpolated z (rather than compute z in the pixel shader itself). This additional check provides not only an added efficiency win when using long, costly pixel shaders, but also provides a form of pixel-level flow control in specific situations. The z buffer can be thought of as containing condition codes governing the execution of expensive pixel shaders. Inserting inexpensive rendering passes whose only job is to appropriately set the "condition codes" for subsequent expensive rendering passes can increase performance significantly. This approach is known as "early-z culling" in real-time rendering.

Our techniques are based on the observation that fluid flow is often concentrated in sub-regions of the simulation grid. The early-z optimizations that we employ significantly reduce the amount of computation on regions that have little to no fluid density or pressure, saving computational resources for regions with higher flow concentration, or for rendering other objects in the scene.

This paper is structured as follows. Section 2 describes previous work on fluid simulation and how it relates to our approach. Section 3 outlines our algorithm. Sections 4 and 5 describe the two acceleration techniques that we employ. In Section 6, we present results of our algorithm. In Section 7, we conclude and present directions for future work.

2. Previous work

Early methods for fluid simulation were based on explicit integration schemes. These methods do not produce a stable simulation unless the simulation timestep is very small. Stam [1999] introduced an unconditionally stable model for fluid simulation. Stam's approach uses a semi-Lagrangian integration method and a projection step to ensure incompressibility [Chorin 1967]. This solver allows for much higher timesteps, resulting in faster, real-time stable simulations. Many recent papers on fluid simulation for different natural phenomena, such as smoke [Fedkiw et al. 2001], fire [Nguyen et al. 2002], and clouds [Harris et al. 2003], are partly based on this solver. We also use this solver as a basis for our acceleration methods for both 2D and 3D simulation. For details on Stam's stable Navier-Stokes solver, we refer the reader to the paper [Stam 1999]. For a thorough description of fluid simulation on graphics hardware, we recommend Harris [2003b].

Recently presented optimization techniques simulate fluid flow more efficiently by using the graphics hardware. Harris et al. [2003] use a red-black Gauss-Seidel relaxation method on their fluid simulation as a vectorized optimization technique. They also achieve faster rendering rates by amortizing their simulation over several frames. In their 3D solver, they propose using a "flat 3D texture" that stores all slices of a 3D volume in order to improve the efficiency of their simulation. Rasmussen et al. [2003] describe an interpolation method to create high-resolution 3D fields from a small number of 2D fields, significantly reducing computation and memory requirements. Several methods have been proposed to approximate light scattering and other visual phenomena in order to achieve realistic results in less time (e.g., Fedkiw et al. [2001] and Harris et al. [2003]).

To our knowledge, the optimization methods to date do not dynamically adapt based on the fact that fluid is often localized, or at least, more highly concentrated in certain areas of the uniform simulation grid. This paper presents an acceleration method that takes advantage of this observation to dynamically reduce computation on regions that have little to no fluid density or pressure. Our method is simple to implement and can be used in conjunction with any of the methods outlined above.

3. Algorithm overview

Next, we outline the steps of our algorithm, which simulates incompressible fluid flow. We perform our simulations with no viscosity, so the diffusion step is omitted. All of the steps of our fluid simulation algorithm are implemented on the GPU using HLSL pixel shaders and are executed by rendering full-screen quadrilaterals to renderable textures. For additional details on how such a flow algorithm is implemented on the GPU, please refer to Harris [2003b]. First we will describe how the early z approach optimizes the 2D fluid flow simulation, and later describe how to extend this approach to 3D fluid flow.

The first two passes insert flow into the density and velocity buffers based on mouse input:

```
InsertVelocity()
InsertDensity()
```

Next, both the density and velocity buffers are advected based on the content of the velocity buffer:

```
AdvectVelocity()
AdvectDensity()
```

Finally the projection step is computed and the velocity buffer is updated in order to remain mass conserving. The brute-force algorithm for computing pressure is as follows:

```
ComputeDivergence()
for ( int i = 0; i < n; i++ )
    UpdatePressure()
SubtractGradient()
```

The `UpdatePressure()` pass is the bottleneck of the algorithm, as it has to be executed approximately 30 times in order to yield visually pleasing results. Our algorithm adds a new pass to prime the z buffer, and employs early-z culling during the pressure computation step:

```
ComputeDivergence()
SetZBufferUsingPressureFromPreviousIteration()
for(int i = 0; i < n; i++)
    UpdatePressureWithEarlyZCulling()
SubtractGradient()
```

The details of the algorithm can be found in Section 4.

After each step of the simulation, we render the density buffer to the screen in a single pass:

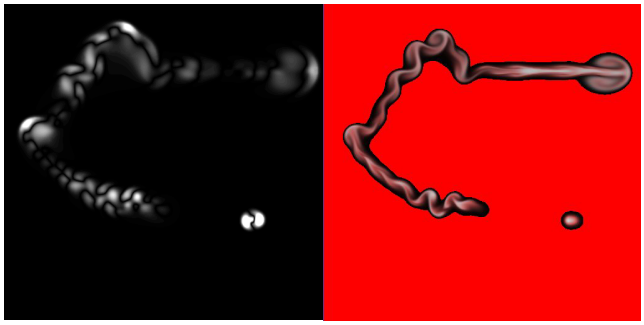
```
RenderDensityToScreen()
```

Due to current hardware limitations, bilinear interpolation of 32-bit floating point buffers must be performed in the pixel shader. This can be very costly if the screen resolution is significantly higher than the simulation grid resolution. In order to reduce the rendering cost, we again

employ early-Z culling to skip this computation on cells that have little or no density:

```
SetZBufferUsingDensity()
RenderDensityToScreenWithEarlyZCulling()
```

Additional details of this particular optimization can be found in Section 5.



(a) Pressure buffer for simulation culling (b) Culled pixels during rendering tagged in red

Figure 2: Visualization of early-z culling.

4. Projection optimization

In this section, we describe an optimization that is performed during the projection step, the most expensive step of the simulation. This step is performed as a series of rendering passes to solve a linear system using a relaxation method. This optimization could also be considered for the diffusion step when simulating highly viscous fluid.

When approximating the solution to this linear system, the higher the number of iterations (rendering passes), the more accurate the result. Instead of performing the same number of passes on all cells, we perform more passes on regions where the pressure is higher and fewer passes on regions with little or no pressure. This is accomplished by performing an additional inexpensive rendering pass that sets the z value of each cell in the simulation based on the maximum value of that cell and a four of its nearby cells from the pressure buffer of the previous iteration of the simulation. We simply set the z value for a particular cell x to be

$$depth = \textbf{saturate}(\alpha P + \beta)$$

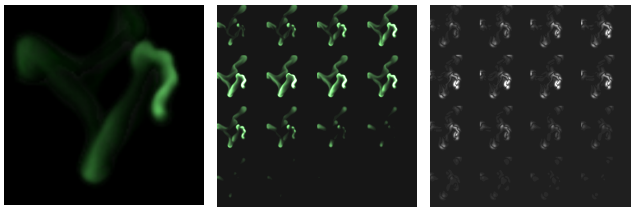
where P is the maximum pressure among the neighbors of x, and α and β are constants. We achieved best results with $\alpha = 2.0$ and $\beta = 0.1$. The β value ensures that, even where the pressure is very small, at least some pressure computation passes will be performed.

We take into account the pressure of the neighbors of x, because each relaxation step computes the new pressure for a given cell as a function of their neighbors. Thus, cells with high pressure may significantly increase the pressure of cells around it. In our experiments we looked at neighbors that were 2 cells away in each of the four directions.

After priming the z buffer, we perform the pressure computation passes. In order to reduce this computation, we set the depth compare state to "less than or equal" and linearly increase the z value on each of the projection passes. On the first pass, the z value is set to 1/N, where N is the total number of pressure passes. On the second pass it is 2/N, and so on. Therefore, on the first pass, all cells are processed (because of our β value), and on subsequent passes, the number of cells that are processed gradually decreases. Figure 2a shows the pressure buffer which is used to set the z buffer that culls the projection computation. Darker values indicate regions of lower pressure, where fewer iterations need to be performed.

The passes that enforce boundary conditions on the pressure computation are not culled. However, since they only affect the pixels on the grid boundaries, it does not hinder the performance of the heavily fill-bound simulation.

Note that this optimization is an approximation and does not necessarily yield physically correct results. However, the visual improvement of using this method is evident, and performing 50 pressure computation passes with this culling technique yields more realistic results than performing 10 pressure computation passes with the brute force algorithm in the same amount of time.



(a) 3D view (b) Density buffer (c) Pressure buffer

Figure 3: Visualization of 3D flow and the density and pressure buffers.

4.1. Extension to 3D fluid flow

We also extended the above optimization to 3D fluid flow simulation. Harris [2003] introduces the idea of simulating 3D flow using a tiled 2D texture (Figure 3ab). This allows each step of the simulation to be performed with one single pass for all the slices, without having to switch render targets. The downside is that the texture coordinate computation is a bit more expensive and proper care must be taken at the boundaries of the slices to correctly account for boundary conditions. However, using this technique is more efficient than having to constantly switch render targets.

Our optimization naturally extends to 3D flow. As in 2D flow, pressure computation is performed by doing multiple passes to update the pressure buffer (Figure 3c). Similarly, we perform one pass to prime the z buffer based on the pressure of the previous simulation iteration, and then, when performing the pressure computation passes, we perform early-z culling the same way we did with 2D flow.

Passes to enforce boundary conditions are rendered as several thin quadrilaterals that tile the texture square. As in the 2D simulation, during these passes, none of the cells are culled.

5. Rendering optimization

The next optimization is performed at rendering time. When rendering the 32-bit floating point density buffer to the screen, bilinear interpolation must be performed in the pixel shader. This computation can be expensive, especially if the dimensions of render target are significantly higher than those of the fluid simulation. In order to avoid applying the bilinear interpolation shader to regions of the screen that have very small density, prior to rendering the contents of the density buffer, we perform an inexpensive rendering pass that simply sets the z buffer value to the density of that particular pixel using "nearest" as the texture lookup filter. When rendering, we set the depth compare state to "less than or equal", and set the z value to a small constant ε in the vertex shader (e.g., 0.01). Thus, all pixels whose densities are smaller than ε are culled. In Figure 2b, all pixels that were too dark to be visible were culled and are tagged in red for visualization purposes. Figure 1b shows the final rendering without the tagged pixels.

Note that this optimization does not directly affect the simulation, since it is only used for rendering. Since less computation time is used for rendering, this optimization does indirectly affect the simulation, because more computation time can be used by the simulation to improve its quality. As shown in the results section, this optimization is particularly useful for applications that zoom in to regions of the flow, or render the result to high resolution buffers.

While the projection optimization works well with 3D flow, unfortunately our rendering optimization does not yield an improvement for 3D flow. The cost of an additional pass for *each* slice, and of switching shaders multiple times is very high.

6. Results

In this section, we present some results of applying the optimizations described in the earlier sections. Figure 1 compares a brute-force simulation and a simulation with early-z culling. Both examples simulate and render the 512×512 simulation grid at 53 frames per second. Since the brute-force approach performs the same number of pressure computations passes on all cells, it only manages 10 pressure computation passes. Our early-z method performs somewhere between 5 and 50 pressure computation passes, depending on the value in Figure 2a. Since our optimization allows for a high number of passes on areas of high pressure, it yields a more realistic result. Our method also renders the flow to a 512×512 window using the optimization outlined in Section 5. However, for this example the rendering optimization does not yield a significant improvement, as will be made clear below.

Figure 4 graphs the performance of our 2D fluid simulation with and without each of our optimizations. The frame-rate is on the y-axis, while the resolution is on the x-axis (we have data points for 128×128, 256×256, 512×512, and 1024×1024 simulation grid resolutions). In each case, the screen resolution for the rendered output is the same as the simulation grid resolution. Five curves are plotted. One using the brute-force method, two using the simulation optimization, and two using both the simulation and rendering optimizations. When the optimizations are used, the frame rate is variable, so we use two curves for measuring performance, one without any flow, and one with the entire grid filled with flow. The actual frame rate will be somewhere between these two curves, depending on how much density and pressure is present. As evidenced by the three lowest curves on the graph, the penalty incurred by having the extra pass to set the z buffer is extremely small. On the other hand, if significant portions of the screen have no flow, the savings can be significant with no visual loss in quality. At 128×128, the savings due to the simulation optimization can be up to a factor of two with no visual loss in simulation quality. At 1024×1024, the savings can be up to a factor of three. The improvement due to the rendering optimization was not significant in this experiment due to the fact that the resolution of the output window is the same as the grid resolution. Since the bulk of the computation happens during the projection step of the simulation, the resolution of the output window would have to be larger than that of the simulation grid for the rendering optimization to yield significant improvement.

Figure 5 graphs the same experiment, but with the output window resolution fixed at 1024×1024. In this case, if the simulation grid has significantly lower resolution, the savings due to the rendering optimization are significant—nearly a factor of two for the 128×128 simulation grid.

Figures 7 and 8 show different examples in which the pressure buffer is not cleared from one pass to the next (causing extremely swirling-like flow). In Figure 7, the small number of passes in the pressure buffer coupled with a slow frame rate results in a velocity field that is not mass conserving. In contrast, when using our culling techniques, the result is significantly more stable.

3D Flow. Figure 9 shows results applied to a 128×128×16 3D fluid flow simulation. Note that, although the improvement is not as significant as in the 2D simulations, the result using our techniques is more realistic.

Blockers. These culling techniques are also very suitable for fluid flow simulations with blockers. Since no computation needs to be performed on most cells that are blocked (approximately half of the cells in Figure 6), these methods can further reduce computational costs. Note that blocked cells that have neighbors that are not blocked cannot be culled and need to be processed in order to yield the proper effect when fluid collides with the blocker.

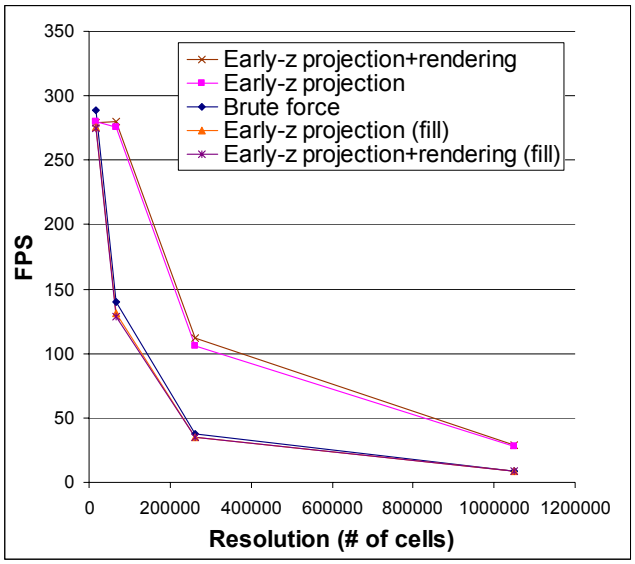


Figure 4: Timings of the different culling methods (screen resolution set to simulation resolution)

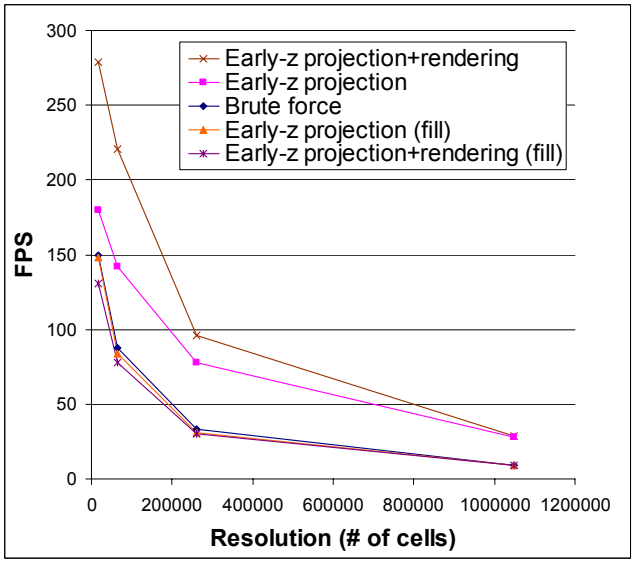


Figure 5: Timings of the different culling methods (1024×1024 screen resolution)

7. Summary and future work

We have presented optimization techniques that take advantage of early-z culling to efficiently simulate and render fluid flow.

The methods presented in this paper are straightforward to implement and yield a significant improvement in rendering speed for the same quality, or conversely, an improvement in quality for a given frame rate. Our results demonstrate that we obtain simulations that look more physically accurate than brute-force simulations at a given rendering speed.

Our method excels in scenes where flow is concentrated on specific regions of the grid, such as scenes with blockers. The main limitation of this approach is that it does not yield a significant improvement to simulations that have a large amount of fluid over the entire simulation grid. But even in the worst case, our simulations will not significantly impair the quality or rendering speed in such settings, as evidenced by Figures 4 and 5.

For future work, it would be interesting to further investigate methods to take advantage of the locality of fluid in the simulation. We are currently investigating a method with adaptive grid sample locations. A hierarchical culling approach could also yield significant savings.

Figure 6: Pink flow colliding with blocker in Van Gogh's Starry Night.

References

Chorin, A. 1967. A Numerical Method for Solving Incompressible Viscous Flow Problems. Journal of Computational Physics 2, pages 12–26.

Fedkiw, R., Stam, J., and Jensen, H. 2001. Visual Simulation of Smoke. In Proceedings of SIGGRAPH 2001, pages 15–22.

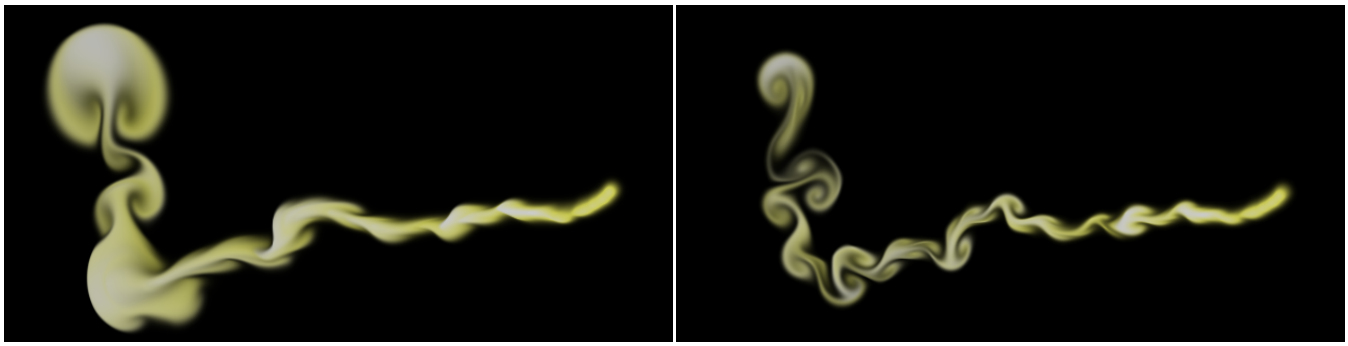
Harris, M. J., Baxter, W. V., Scheuermann, T., and Lastra, A. 2003. Simulation of cloud dynamics on graphics hardware. In Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, pages 92–101.

Harris, M. J. 2003b. Real-time cloud simulation and rendering. Ph.D. dissertation. University of North Carolina at Chapel Hill,, 2003.

Nguyen, D., Fedkiw, R., and Jensen, H. 2002. Physically Based Modeling and Animation of Fire. In Proceedings of SIGGRAPH 2000, pages 736–744.

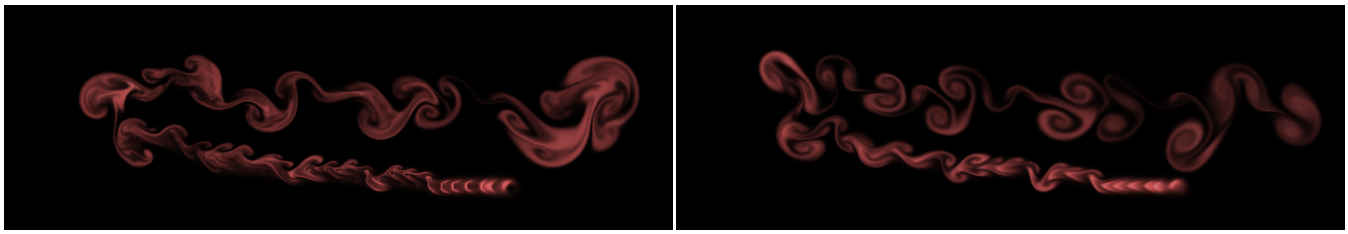
Rasmussen N., Nguyen, D. Q., Geiger, W., and Fedkiw, R. 2003. Smoke simulation for large scale phenomena. In Proceedings of SIGGRAPH 2003, pages 703-715.

Stam, J. 1999. Stable fluids. In Proceedings of SIGGRAPH 1999, pages 121–128.



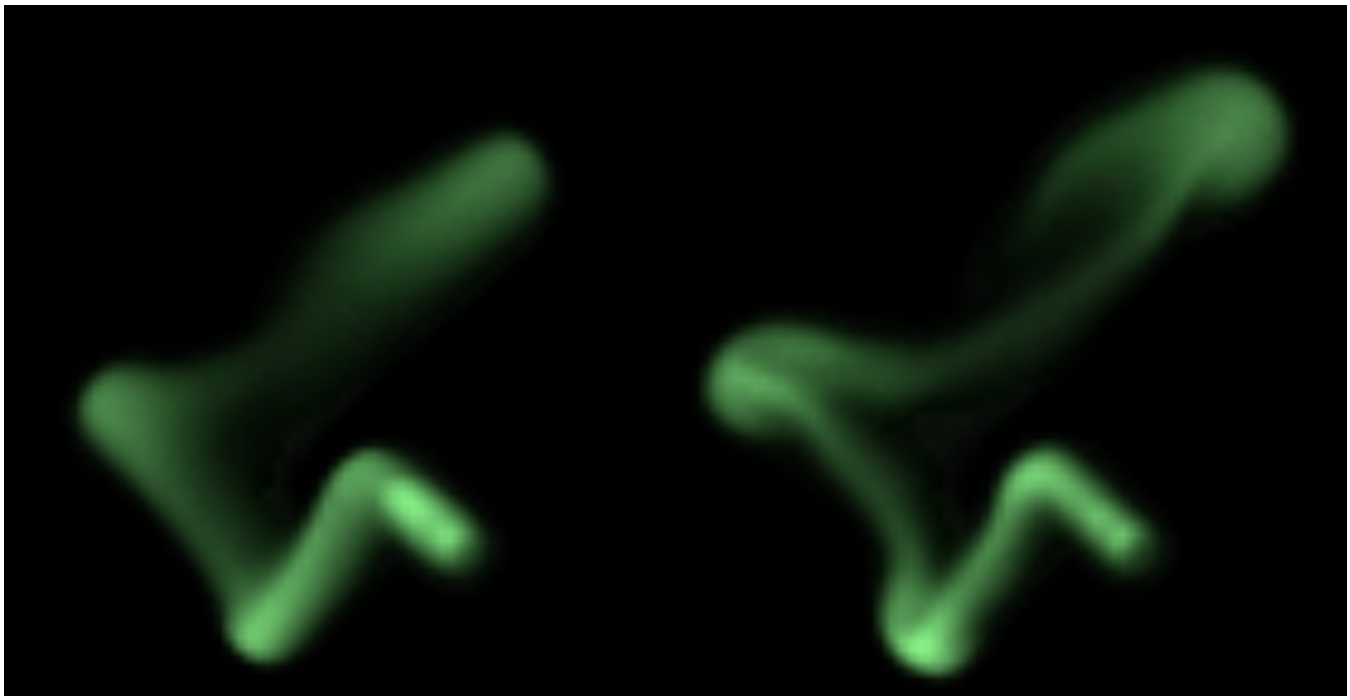
(a) Brute force (3 projection passes) **(b) Early-z (up to 15 projection passes)**

Figure 7: Side by side of a 1024×1024 2D simulation. Both simulations render at 25fps. The small number of passes on the brute-force example causes the velocity field not to be mass conserving.



(a) Brute force (5fps) **(b) Early-z (16fps)**

Figure 8: Side by side of a 1024×1024 2D simulation. Both simulations have 40 projection passes. The lower frame rate on the brute-force example causes artifacts when flow is inserted at a constant, faster rate (e.g., interactive mouse input).



(a) Brute force (3 projection passes) **(b) Early-z (up to 30 projection passes)**

Figure 9: 128×128×16 3D flow simulation. Both examples render at 25fps.