REAL TIME FLUID DYNAMICS

Computer Graphics 2 Final Project

By: Dima Mukhin #7773184

Based on an article and research paper by Jos Stam

FEBRUARY 2, 2019

Table of Contents

Real-Time Fluid Dynamics

Motivation

Fluid flows can be found everywhere in reality, from leaves flowing through a river to smoke flowing through objects. As video games these days try to achieve more realistic results, it is important to develop a way to simulate fluid dynamics that look and behave realistically, yet can be calculated and rendered in real time. For example, such real-time simulation algorithm could light a fire through a campfire in a video game. Therefore, in this project I decided to focus on researching, implementing, and describing a real-time fluid dynamics solver.

Introduction

In this project, I will describe a simple real-time implementation of fluid dynamics based on an article titled "Real-Time Fluid Dynamics for Games" by Jos Stam, which is based on his research paper on Stable Fluids. This fluid dynamics solver can be used to implement fluid-like effects such as dye moving and diffusing through water, smoke flowing in the air, or even fire coming out of a torch. The algorithms presented in this paper are based on the Navier-Stokes equations of fluid flow, which aim for physical accuracy, and thus are extremely hard to solve. On the other hand, Jos Stam in his paper suggests a simplified solver for fluid dynamics that can be computed in real time, and achieves high visual quality without deviating much from reality.

The implementation part of this project will demonstrate an example of real-time fluid simulation of smoke or dye flowing through water/air in 2 dimensions. While the application is implemented in 2 dimensions for simplicity, it is important to note that it can be easily extended to 3 dimensions by adding another dimension to some of the variables involved in the simulation.

The Navier-Stokes equations

Before we talk about the approximation algorithms for fluid simulation, it is important that we understand the basics of the algorithms our fluid solver will be based on, the Navier-Stokes equations. In physics, the Navier-Stokes equations are a set of equations describing the motion of viscous fluid substances. The Navier-Stokes equations are very useful because they describe the physics of many phenomena such as ocean currents, and air flow around objects. However, these equations are very hard to solve, especially in three dimensions, and it is not even proven whether a solution in 3 dimensions always exists. In fact, this problem is one of the seven most important open problems in mathematics, and has a prize offer of 1 million US dollars for a solution or a counterexample.

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

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

Figure 1: The Navier-Stokes equations for velocity (top) and density (bottom)

Fluid simulations in computer graphics

There are several different ways to simulate fluid dynamics in computer graphics. In this project, we will simulate the state of a fluid at a given point in time by using a velocity vector field. Each point on this 2D or 3D vector field represents the velocity of the fluid at any given point. To understand this concept, imagine a container filled with completely still water. In this state, our vector field will contain zero vectors across the 3D grid. However, if at some point we push the container, the water will start to move. This will be represented as non-zero velocity vectors in the direction of movement across the 3D grid.

A velocity field by itself, however, will be quite pointless, since there is nothing to display but the velocity at any given point. We can throw particles into our scene and calculate their velocity based on their location in the velocity grid, and then draw these particles to see how the fluid behaves. Using particles is a simple and effective solution for displaying things like leaves flowing on water. However, using particles for displaying the motion of smoke or fire in the air will be extremely inefficient when trying to achieve accurate effects. To solve this issue, we can instead use a "particle density" grid. The particle density grid will look a lot like our velocity field, but instead of holding velocity vectors, it will contain density values for our smoke or fire in space. Each cell in the particle density grid will have a matching cell in the velocity field so we could move the smoke, and each value in each cell will represent the density that we could use to display the smoke.

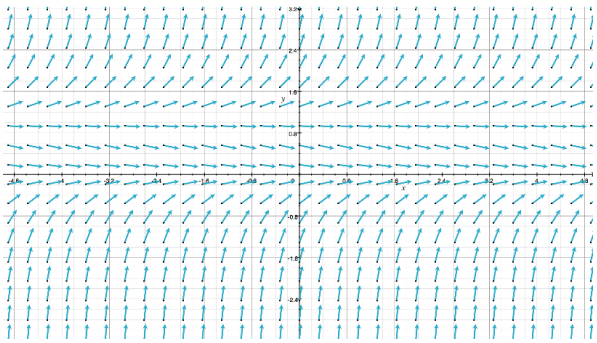

Figure 2: Velocity grid

The Fluid Simulator: Introduction

To simulate the evolution of velocity and density through the velocity and density fields through time we could use the Navier-Stokes equations. Given the current state of our velocity and density fields, the Navier-Stokes equations will tell us exactly how both grids will change over a single time step. The evolution of the velocity field can be described by the top Navier-Stokes equation in figure 1, while the evolution of the density field can be explained by the bottom equation.

By looking at both equations in figure 1, it should be clear that both look very similar. However, the top equation for the velocity field is much harder than the equation for the density field. The reason behind that is that the top equation is non-linear, while the bottom one is linear. Thankfully, since we are fine with giving up some physical accuracy for performance, and because both equations are very similar to each other, we can build an algorithm for the density moving through a fixed velocity field, and then use some of the components of the density algorithm to develop an algorithm for solving the velocity field through time. To explain how this works, we will first show how to develop the density algorithm based on its Navier-Stokes equation.

Moving Densities

According to the Navier-Stokes equation, moving the densities in the field in a single time step consists of 3 steps. The first step is very simple, and it states that we should add density to our field from external sources. These external sources could be a user clicking on the screen to add more smoke, or maybe a burning log in the scene generating fire and smoke. The second step states that density should diffuse through the grid at a certain rate. To understand this step, imagine dropping a drop of red color dye onto a container of water. The dye will naturally diffuse in all directions and the red color will fade. The third and last step of the density moving algorithm states that density should move according to forces applied to it by the velocity field. Therefore, if we have a velocity field where all cells point to the left, our dye from the previous example should also move to the left in a given time step. Next, let's look into each step in greater detail and develop an algorithm for them.

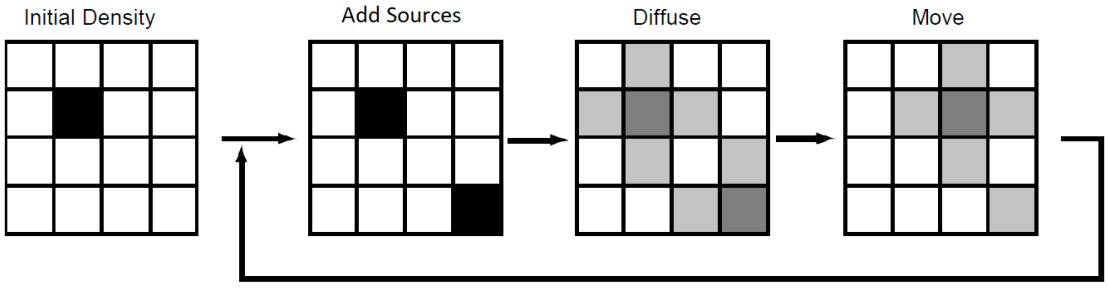

Figure 4: Moving densities in a single time step

Adding Sources to Densities

The first step, adding sources to the density field, states that we should add density to our field from external sources. This step is very easy and simple to implement, and it just consists of adding values to our grid of density.

```
void addSource(int x, int y, float amount) {
    density[x][y] += amount;
}
```

Figure 5: add sources step pseudo code

Diffusing Densities

The second step, diffusing densities in the density field, states that density should diffuse through the grid at a certain rate *diffuseRate*. To understand how the diffusion step will work, lets first look at what happens at a single density cell. At any given cell in our grid, the cell will exchange densities with 4 of its neighbors. Therefore, the cell density will decrease by losing density to its neighbors, but also increase by gaining densities flowing from its neighbors. This logic results in the following formula in figure 6.

```
density[x][y] = prevDensity[x - 1][y] +
                prevDensity[x + 1][y] +
                prevDensity[x][y - 1] +
                prevDensity[x][y + 1] -
                4 * prevDensity[x][y];
```

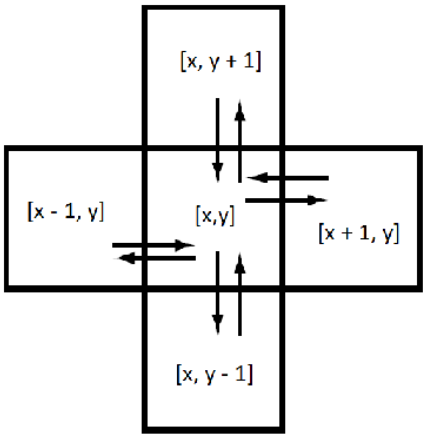

Figure 6: On the left: pseudo code for simple diffusion at a given cell [x,y]. On the right: illustration of diffusion of a cell [x,y] with its neighbors.

Therefore, to apply diffusion we can simply implement the algorithm from above to every single cell in our density grid. However, if we do so, we get a problem. Unfortunately, for large time steps, or large diffusion rates, the above algorithm becomes unstable, density values overflow, become negative and finally diverge. The reason behind this is that for large time steps or diffusion values, our values will overflow because we can no longer look at just the neighboring cells. To fix this problem, we can use a more stable method for the diffusion step. The new diffusion algorithm will find the densities which when diffused backward in time yield the densities we started with, resulting in the following equation in figure 7.

```
prevDensity[x][y] = density[x][y] - a * (
                    density[x - 1][y] +
                    density[x + 1][y] +
                    density[x][y - 1] +
                    density[x][y + 1] -
                    4 * density[x][y]);
```

Figure 7: The new stable density equation

Since the new algorithm "goes back in time" to figure out the new density values, all of the densities on the right-hand side of the equation above are unknowns. Fortunately, our new equation is linear and easy to solve in real time. To solve it, we can use the Gauss-Seidel relaxation algorithm, which is an iterative method used to solve a linear system of equations. The variable a in figure 7 is simply the calculated diffusion rate which is based on the diffusion property constant of the fluid, the time step, and the size of the grid. Figure 9 below shows how the new algorithm goes back in time to calculate the new diffused density. Note how a single time step results in new density values further away from the neighbors of the previous time step.

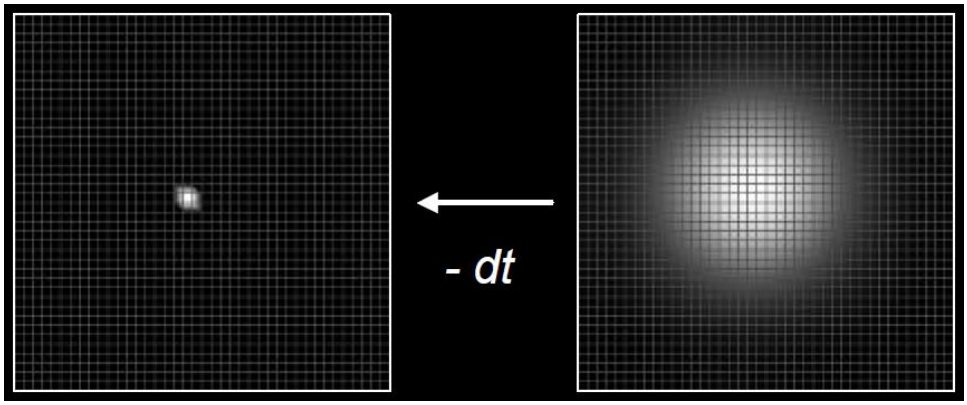

Figure 9: Diffusing densities back in time [7]

Adding Velocity Forces to Densities

The final step, adding velocity forces to the density field, states that density should move according to forces applied to it by the velocity field. One way to achieve this step is to simply move every density to one of its neighbors based on its current velocity vector. However, this approach is very naïve and just like the first approach we looked at for diffusion, it is not stable and doesn't work for large time steps. Another way we could move densities is to consider every density cell as a particle, and apply velocity vectors to it. This method sounds very simple at first. However, converting these particles back to cells in our density grids will be much harder. A better way to move the density is by using a technique very similar to the technique we used for the diffusion step.

To implement the final step of our density solver, for a given cell in the density field, we will consider the center of the cell as a particle. Then, much like in the diffusion step, we will trace this particle back in time using the velocity field and our time step. After that, we will take the 4 density cells closest to the “back in time” location of the density particle, and we will interpolate the density values of these 4 cells based on the distance from the particle. The interpolated density value we get is our moved density. This concept might be easier to understand by looking at Figure 10 below.

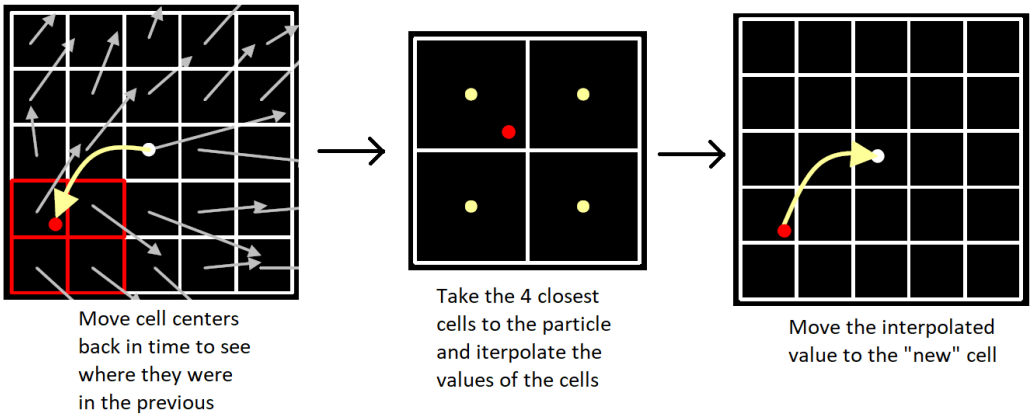

Figure 10: Algorithm for moving density using forces from the velocity grid

Adding all the density steps together

Now that we know how to perform all 3 necessary steps to move the densities all that is left is combining all of them together. To do this, we will need to keep track of the size of our grid, the velocity field as well as the previous velocity field, the time step size, the diffusion rate, and finally the density grid as well as the previous density grid. We will perform every step one after the other, starting from adding new sources to the densities, then diffusing the densities, and finally moving them according to the vector field. In between each step we will swap the old velocity grid with the new velocity grid, and the old density grid with the new density grid. This ends our density solver, next we will see how we can reuse the density solver algorithm to solve the harder velocity solver algorithm.

Evolving Velocities

To develop the velocity solver, similarly to the density solver, we will use the top Navier-Stokes equation from figure 1.

Adding Sources to Velocities

According to the last term on the right-hand side of the equation, velocity changes due to external sources. This means that we should add new velocity forces to our velocity grid from external sources. For example, we could have a fan blowing air into our scene, thus adding new velocity vectors to our grid. In the demo application of the project, the user can click the right mouse button and drag it on the screen to add new velocity forces to the velocity grid. This step is very similar to the first step of the density solver, which means we can reuse or duplicate the same algorithm with some minor changes to implement this step for the velocity algorithm.

Diffusing Velocities

According to the middle term on the right-hand side of the equation, velocity also diffuses just like our density, which accounts for the effects of viscosity. This means that different kinds of fluids diffuse their velocity forces differently according to their viscosity constant. For example, viscous fluids will have a higher viscosity constant which will make the velocity field more “smooth”, resulting in a more viscous like fluids. The algorithm for diffusing velocity will be almost identical to the diffusion algorithm of densities, therefore, we can either make it reusable or duplicate it and make it work with our velocity field. The only difference will be that our velocity field holds vectors rather than single values, so we would have to diffuse the values once for each dimension, 2 if we use only 2 dimensions, or 3 if we use 3 dimensions.

Adding Velocity Forces to Velocities

Finally, let's look at the first term on the right-hand side of the equation. This is the term that makes the equation non-linear, and makes it hard to solve in real time. However, it also looks a lot like the first term on the right-hand side of the density equation that says that the density should move according to forces of the velocity field. Therefore, we can try to naively interpret this term as saying that the velocity should move along itself. In fact, we can even apply the same algorithm from the density step to our velocity to achieve this. This was one of Jos Stam’s fundamental discoveries for developing the real-time fluid dynamics solver. Thanks to this discovery, the fluid solver time complexity reduces significantly, and makes it possible to simulate fluids in real time.

Conservation of Mass

So we now know how to perform every step to evolve the velocity according to the Navier-Stokes equation. However, there is still one more step missing to make the fluid look more realistic. This step states that fluid should conserve its mass, which is a very important property of real fluids. What this means is that the flow into a cell should be equal to the flow out of the cell. The problem is that after each of the velocity steps, our fluid is never mass conserving, and we need to develop another step to “correct” the fluid to be mass conserving. The visual effect of this correction will result in many velocity “vortices” to appear in our velocity grid, which will curve the flow of our density and make it look much more realistic.

To achieve this correction effect, we will use a mathematical result known as the Hodge decomposition of a vector field, which states that every vector field is the sum of a mass conserving field and a gradient field. Figure 11 below shows this mathematical result. On the left-hand side, we have our current velocity field. The first field on the right-hand side is the mass conserving field that we want to get, notice how it has many velocity “swirls” to conserve the mass. The last field on the right is the gradient field which is considered as the worst case for us. Therefore, to get a mass conserving field, we will have to subtract the gradient field from our current vector field. Now we just need to figure out how to find the gradient field of a given vector field.

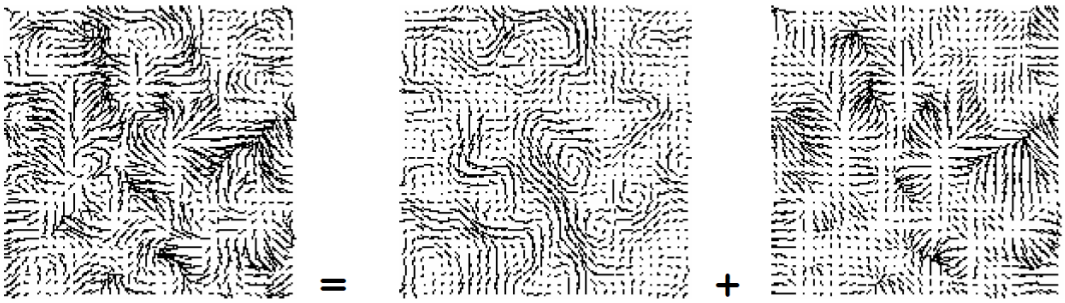

Figure 11: Hodge decomposition of vector fields. Every vector field is the sum of a mass conserving field and a gradient field

If we look closely at the gradient field above, we can see that the flow at some points either all direct outwards or inwards. In fact, the gradient field can be visualized as the steepest descent of a height map. For example, imagine a terrain with mountains, hills, valleys. The gradient field for the height map of this terrain will look like many vectors pointing down-hill towards the steepest descent. Therefore, computing the gradient field for our vector field will involve computing a vector representation of the height map for our grid.

To compute the gradient field we can use Poisson's equation, which will help us compute a gradient field for a given vector field. Poisson's algorithm consists of a linear system very similar to the one we had for the diffuse step Therefore, we can also combine it with Gauss-Seidel relaxation to find the gradient field and subtract it from our current velocity field to finally get the mass conserving field.

Adding all the velocity steps together

Now that we know how to perform all 4 necessary steps to evolve the velocities all that is left is combining all of them together. To do this, we will need to keep track of the size of our grid, the velocity field as well as the previous velocity field, the time step size, and the viscosity rate. We will perform every step one after the other, starting from adding new sources to the velocity, then diffusing the velocities, and finally moving the velocity based on its own values. In between each step we will swap the old velocity grid with the new velocity grid, and correct the grid based on the conservation of mass algorithm.

Setting Boundaries

The last thing that is left for our solver to be complete is to set boundaries to our fluid container. We assume that our fluid is contained in a container with solid walls, and therefore, nothing can escape our container. To make sure that nothing escapes our container, we need to negate the velocity at the boundary cells based on their neighbors. For example, a right boundary cell will have to contain the negative velocity value of the cell to its left to make sure that no density escapes the container. Unfortunately, this doesn't work properly. Simply negating vectors at the boundaries will cause a bouncing effect of the density instead of a more natural effect of the densities flowing along the walls of our container. To fix this, we only need to negate the x-direction for the vertical walls, and similarly only negate the y directions for our horizontal walls. The boundary vectors, therefore, will cause the density to naturally flow out of the wall without escaping (for more details see figure below). To calculate the velocities at the corners, we simply take the average of the 2 (or 3 in 3D) edge cells adjacent to it.

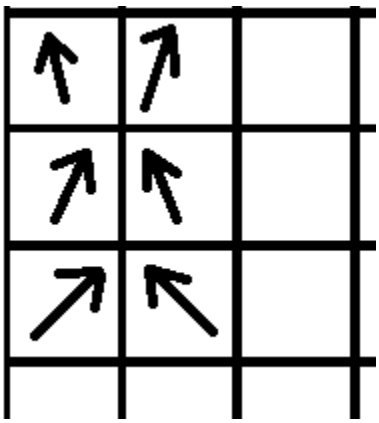

Figure 12: Left vertical edge vectors negate vectors to their right only in the x-direction while keeping the same y-direction.

Conclusion

With the boundary algorithm, we finish all the algorithms necessary for the fluid dynamics solver. Now all that is left is to call all the algorithms we talked about in the right order. First, we add external forces and densities to our grids. Next, we diffuse and apply forces to the densities. After that, we diffuse and apply forces to the velocity grid. In between all the steps we need to constantly set the boundaries and not forget to keep track of the previous state of our velocity and density fields. Finally, we would also need to use conservation of mass algorithm on the velocity grid to conserve the mass of the fluid and make it look more realistic.

Screen-shots

The following are some screen-shots from the demo app, as well as other implementations by Jos Stam.

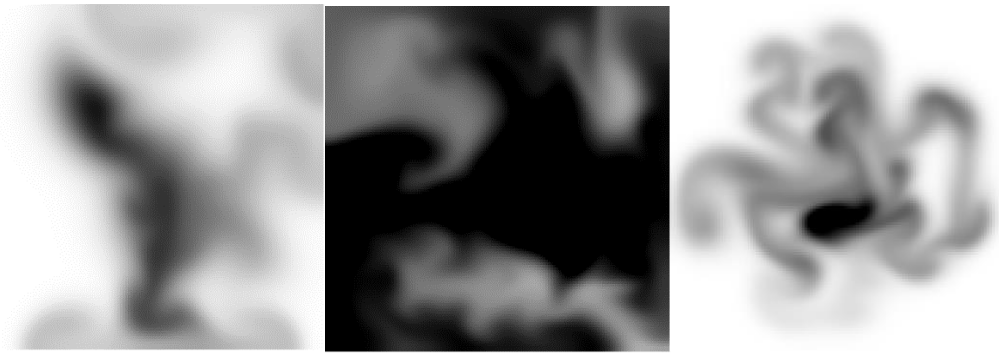

Figure 13: Screen-shots from the demo app included with the project

Figure 14: Fire effect created with the MAYA Fluid Effects technology

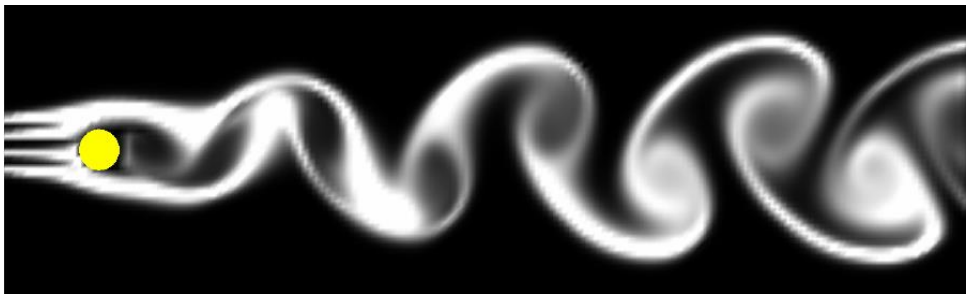

Figure 15: Von Karmann vortex street behind a sphere

References

n.d. "Gauss-Seidel method." *Wikipedia.* Accessed March 2019. https://en.wikipedia.org/wiki/Gauss%E2%80%93Seidel_method.

Khan Academy. n.d. "Vector Fields." Accessed March 2019. https://www.khanacademy.org/math/multivariable-calculus/thinking-about-multivariable-function/ways-to-represent-multivariable-functions/a/vector-fields.

n.d. "Millennium Prize Problems." *Wikipedia.* Accessed March 2019. https://en.wikipedia.org/wiki/Millennium_Prize_Problems#Navier%E2%80%93Stokes_existence_and_smoothness.

n.d. "Navier-Stokes equations." *Wikipedia.* Accessed March 2019. https://en.wikipedia.org/wiki/Navier%E2%80%93Stokes_equations.

Stam, Jos. 2003. "Real-Time Fluid Dynamics for Games." *Autodesk Research.* https://www.autodeskresearch.com/publications/games.

Stam, Jos. 1999. "Stable Fluids." *SIGGRAPH.* https://dl.acm.org/citation.cfm?id=311548.

Stam, Jos. 2002. "Stable Fluids Presentation." *SIGGRAPH course.* http://www.dgp.utoronto.ca/~stam/reality/Talks/FluidsTalk/FluidsTalkNotes.pdf.