BLAZE: A Parallel Fluid Solver and Renderer on the GPU

ALEXANDRE SIROIS-VIGNEUX, McGill University

Fig. 1. Render of three distinct scenes simulated and rendered on the GPU using Blaze, the program presented in this paper. This shows from left to right: fire debris, an explosion and some ground fires.

Blaze is a fluid solver and renderer running entirely on the GPU. Our framework avoids expensive memory transfers between host and device by only allocating grid information on the device which leads to remarkable performance gain in practice. Blaze can read scene descriptions generated using a minimalist user interface for a simple and yet powerful fluid workflow.

Code: https://github.com/asiroisvigneux/Blaze
Video: https://youtu.be/lxIIJf6EVqo

Additional Key Words and Phrases: Computational Fluid Dynamics, Gauss-Seidel red-black, GPU, Raymarching

1 INTRODUCTION

Fluid simulation and rendering are often approached in the visual effects industry as two separate sequential processes where the content of the simulation needs to be stored on disk to later be rendered by another piece of software. Those large memory transfers impose serious restrictions on how those applications can be accelerated using graphics hardware. Storing the volumetric data on disk also imposes noticeable stress on the network in terms of bandwidth and requires massive storage solutions as some of those simulations can easily reach tens of terabytes per version. It is also important to note that those simulations are usually visualized using the graphic pipeline which does not accurately represent the final ray marched image. This discrepancy is often a source of problems as some artifacts in the fluid might only become visible using the high-quality render engine which usually happens after the whole simulation. Our method skips these intermediate steps and goes directly to the final high-quality render in the least amount of steps. The output of our program can then be trusted at all times without any loss of fidelity. Only the particle files and the rendered frames need to live on disk which requires modest storage hardware compared to the requirement of also storing the volumetric grids.

2 RELATED WORK

In 1999, Stam presented his paper on stable fluids that, for the first time, proposed an unconditionally stable model to simulate complex fluid-like flows. This publication was a major contribution to computational fluid dynamics (CFD) for computer graphics and has triggered multiple follow-up publications in the years after [Stam 1999]. In 2008, Bridson published the first edition of his book "Fluid Simulation for Computer Graphics" which was revised with a second edition in 2015 covering all aspects of fluid simulation, from the mathematics and algorithms to implementation. This will become a reference to many interested in entering the field of CFD [Bridson 2015]. In 2009 Gomes et al. investigated the acceleration that could result from porting the solve of the sparse linear systems for the non-divergent projection to the GPU [Gomes 2009]. Their comparison between the CPU and GPU implementation has shed some light on the benefits and challenges associated with the use of programmable GPUs for fluid simulations.

Author's address: Alexandre Sirois-Vigneux, McGill University, alexandre.sirois-vigneux@mail.mcgill.ca.

3 METHODS

Our implementation of the solver and renderer is done completely on the GPU using the CUDA Toolkit 10.2. The idea is to move the entire fluid simulation and rendering pipeline to the graphic card to avoid the expensive cost of transferring grid information between host and device memory. Those memory transfers are known to be the primary source of bottlenecks in most GPU accelerated applications. In our framework, only the source particles will be pushed to the device and, similarly, only the framebuffer containing the rendered pixels will be retrieved from GPU memory. This design choice allows us to maximize occupancy on the GPU and keep it fully saturated at virtually any point of execution. The structure of our program was inspired by two great repositories [Bitterli 2021] and [Allen 2021]. For a more detailed overview of the program's execution, see Fig. 2.

3.1 Simulation

Instead of solving for the Navier-Stokes equations, we solve for the Euler equations of inviscid fluid by effectively dropping the viscosity term from the equations.

$$\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} + \frac{1}{\rho} \nabla p = \vec{g} \tag{1}$$

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

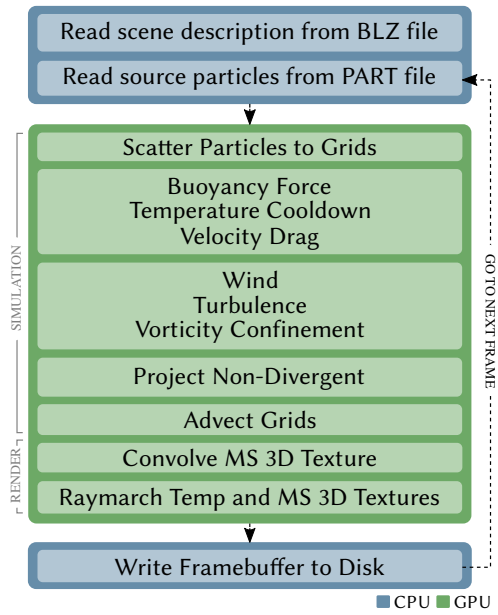

Fig. 2. Execution overview of Blaze

We use splitting to decompose the complex equations into multiple smaller components that are then numerically solve one after the other [Bridson 2015]. The three main steps of our solve are in order: Body Forces

$$\frac{\partial \vec{u}}{\partial t} = \vec{g}, \tag{3}$$

Pressure / Incompressibility

$$\frac{\partial \vec{u}}{\partial t} + \frac{1}{\rho}\nabla p = 0 \quad \text{such that} \quad \nabla \cdot \vec{u} = 0 \tag{4}$$

and Advection

$$\frac{\partial \vec{q}}{\partial t} + \vec{q} \cdot \nabla \vec{u} = 0. \tag{5}$$

Efficient Memory Usage: To minimize our memory footprint in GPU memory, we chose to simulate density and temperature as a single field. The range of temperature values is remapped at render time to use the full range as smoke density and the high end of the spectrum to simulate light emission from fire. This does impose some limitations on the type of phenomena that can be model with our solver, but it also allows it to handle higher grid resolution. Fig. 3 gives a detailed view of the primary data structures required in GPU memory at runtime.

Staggered Grids: Following the recommendation in [Bridson 2015], we use Marker-and-Cell (MAC) or Staggered grids to store the velocity information, see Fig. 4. Although it makes the implementation more complex and can lead to more warp divergence due to misalignment between grid dimensions, it also provides second-order accurate central differentiation for free.

$$\frac{\partial q}{\partial x} \approx \frac{q_{i+1/2} - q_{i-1/2}}{\Delta x} \tag{6}$$

This is especially useful when trying to make our velocity divergence-free by avoiding the non-trivial null-space problem [Bridson 2015].

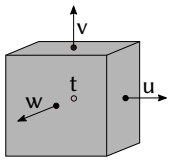

Fig. 4. A single MAC grid voxel where the velocity values lives at the boundaries of the grid cell while everything else (temperature, pressure and divergence) is stored at the center.

Project Non-Divergent: To make our velocity grids divergence-free we need to numerically update our velocities with

$$\vec{u}^{n+1} = \vec{u}^{n} - \Delta t \frac{1}{\rho}\nabla p \quad \text{such that} \quad \nabla \cdot \vec{u}^{n+1} = 0. \tag{7}$$

This leads to the following Poisson problem

$$-\frac{\Delta t}{\rho}\nabla \cdot \nabla p = -\nabla \cdot \vec{u} \tag{8}$$

that can be expressed as a linear system of equations with unknown pressure. There exist multiple ways of solving such a system like Jacobi, Gauss-Seidel or Conjugate Gradient to name a few. Our program implements a Gauss-Seidel red-black solver since previous work shows it outperforms both Jacobi and Conjugate Gradient in the context of a parallel GPU implementation [Gomes 2009].

One important advantage of this algorithm compared to Jacobi is that it can be performed in-place. This leads to lower memory consumption while taking advantage of the previously calculated values to speed up convergence. The idea behind this technique is to conceptually apply a red-black checker pattern across the grid so no neighboring voxels share the same color, see Fig. 5.

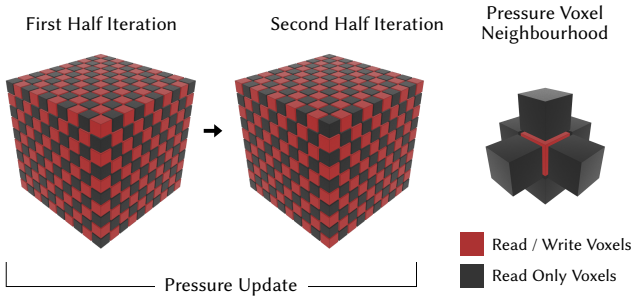

Fig. 5. Visualization of the Gauss-Seidel red-black partitioning of the pressure grid. Each color is solved consecutively to avoid write hazards.

The Gauss-Seidel update of a pressure voxel will involve itself and all its direct neighboring voxels tagged with a different color, meaning those are read-only for the course of the first half iteration. The second half iteration will update the other color while keeping the previously updated color unchanged. This two-step process eliminates the write hazards inherent to parallel implementation of the Gauss-Seidel algorithm thus ensuring deterministic results. In practice, the color change is sequentially performed on each 2D slice of the domain to exploit precalculated values as much as possible.

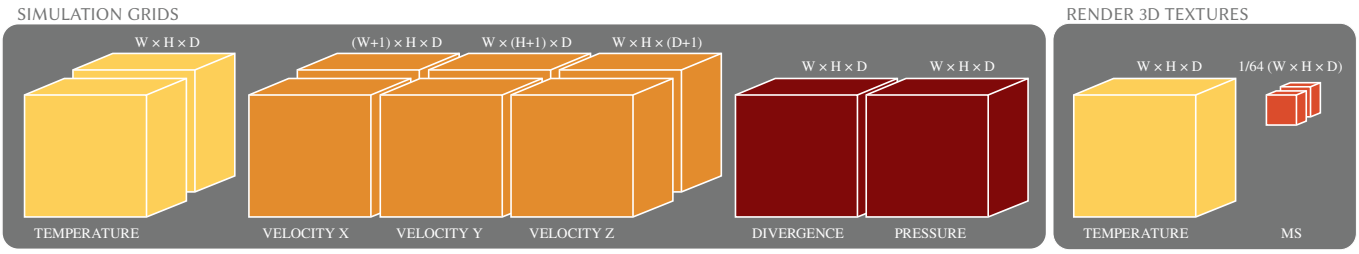

Fig. 3. Data Structures in GPU memory. The dimensions of the structures are indicated above where W, H and D represent the width, height and depth of the fluid domain respectively. Some structures are duplicated in depth to indicate the presence of a front and back buffer in memory. This data duplication is necessary for tasks such as advection and convolution where in-place operations are not possible.

Grid Advection and Interpolation: Velocity and temperature are transported through the grid using Semi-Lagrangian [Stam 2003] advection with third-order accurate Runge-Kutta method [Ralston 1962].

$$\begin{aligned} k1 &= f(q^n) \\ k2 &= f(q^n + \frac{1}{2}\Delta t k_1) \\ k3 &= f(q^n + \frac{3}{4}\Delta t k_2) \\ q^{n+1} &= q^n + \frac{2}{9}\Delta t k_1 + \frac{3}{9}\Delta t k_2 + \frac{4}{9}\Delta t k_3 \end{aligned} \tag{9}$$

This allows us to reach good accuracy even when using large time steps. We use trilinear interpolation for intermediate velocity samples and tricubic interpolation for the actual advection of the fluid quantities.

Tricubic interpolation is much more expensive, but also two orders of magnitude more accurate than trilinear interpolation [Bridson 2015]. Because of their heavy use in the program, both tricubic and trilinear kernels have been optimized to reduce their use of local variables. This inevitably leads to less readable code but has proven to be an efficient way to get more than a two-time speedup on those function calls. The original implementation of the tricubic kernel had so many local variable declarations that it would exhaust all available registers on the GPU, consequently limiting the number of threads that could be launched in parallel.

Numerical Errors as Viscosity: As mention before, we chose to drop the viscosity term from the Navier-Stokes equations. The idea behind this is twofold. First, we want to make sure we capture as many visual details as possible within the limits of our grid resolution. Secondly, most numerical methods for simulating fluids unavoidably introduce errors that can be physically reinterpreted as viscosity [Bridson 2015]. Not explicitly modeling viscosity also reduces the amount of calculation required per time step.

Particle-to-Grid by Replacement: At the beginning of each time step, particles are scattered to the grids (P2G) through a GPU kernel where each thread scatters a single particle. The write hazards are resolved using atomic functions that perform a read-modify-write atomic operation on the grids. This ensures that no thread will overwrite a value that was just set by another thread running concurrently. The use of those operations is known to lead to severe performance degradation [Gao et al. 2018], but it is possible to reduce the impact of atomic functions by minimizing particle overlap in the particle files directly. Such optimization can easily be done in Houdini as shown in Fig. 6.

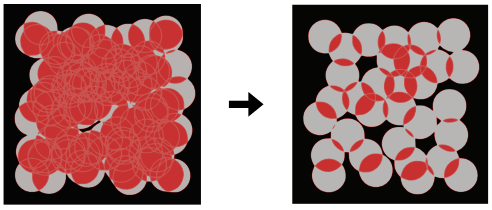

Fig. 6. Both images show the same random distribution of particles in 2D, but particles strongly overlapping others were removed from the image on the right.

Conducting this simple step has led to performance comparable to completely avoiding the use of atomic functions since threads rarely run into write-conflicts. As opposed to common implementation, our sourcing is done by replacement instead of by addition which has multiple advantages in terms of control. First, it is easier to model abrupt changes such as drastic velocity changes without having to inject excessive force to counter the momentum already present in the simulation. This kind of discontinuity in the grids also tends to create interesting details as the simulation evolves. Secondly, values inside the grids stay bounded by the source which allows the user to easily define shader ramps as well as other parameters depending on the range of values inside the grids.

Our P2G is done in three steps. We start by clearing the back buffer of all grids, we then scatter the particles to the back buffers using atomic functions. Finally, values exceeding a threshold of 1×10^{-5} on the back buffer will overwrite the values of the front buffer. This "set" operation is a little more computationally expensive than "add", but it is negligible compared to other steps of the solver. The particle scattering uses a cubic pulse function [Quilez 2021] that smoothly transition from zero at the particle surface to one at its center. One thing to note is that since each thread handles a single particle, it is possible to make this step very slow when scattering large particles. It is therefore more efficient to source a point cloud with thousands of small particles in the shape of a sphere instead of one big particle to fill the same space in order to take advantage of the parallel nature of the hardware.

Dense Wind: As opposed to the current trend in fluid simulations, our approach is to use a dense domain instead of a sparse

one. It is a common mistake to neglect the contribution of air motion in smoke and fire simulations for computer graphics. Although some phenomena can properly be modeled with sparse domains, most of them will be lacking important features that are too often replaced by excessive procedural turbulence. Our program allows the user to define each side of the domain as either a solid boundary or a free surface. Free surfaces can be coupled with a wind force to simulate the interaction of the environment on our finite simulated domain. The user can define a procedural vector Perlin noise using cudaNoise [Lehtinen 2021] along with a wind direction that will be blended on two opposite sides of the domain over one second.

$$\vec{u}^{n+1} = \vec{u}_{wind}\Delta t + \vec{u}^{n}(1 - \Delta t) \tag{10}$$

This wind velocity is being transported across the domain similar to a wind tunnel until it exits on the other side. This leads to more natural behavior than adding procedural turbulence directly on top of the rendered voxels as it lets the solver smoothly handles the integration of the procedural wind with what is already happening in the simulation domain.

Air Turbulence: It is also possible to inject turbulence into the simulation. Two masks can be used to control where turbulence is added. The first mask uses temperature to keep the turbulence outside the renderable volume as we commonly want to add details at the interface between air and density without perturbing the general motion of the simulation. The second mask is derived from the velocity magnitude to add more turbulence to fast-moving regions of the simulation where strong pressure fronts can appear. This also allows us to modulate the turbulence in ways that will not affect the very slow-moving part of the simulation. Like the wind force, this turbulence is also generated using [Lehtinen 2021], but instead of using the Perlin noise $\vec{\psi}$ directly, we transform it into curlnoise using the formula provided in [Bridson 2015].

$$\vec{u}_{curlnoise} = \nabla \times \vec{\psi} \tag{11}$$

The curlnoise is calculated at the voxel centers similar to how divergence is computed and is temporarily stored on the velocity back buffers before being scattered to the front buffers. The scattering step needs to rely on atomic functions as the quantities from the voxel centers need to be weighed and distributed to the corresponding faces using backward trilinear interpolation. Failing to use atomic functions would lead to non-deterministic behavior because of the write hazards. To control the magnitude of the vectors generated from the curl computation in floating-point precision, we clamp the magnitude of the curlnoise using the following expression.

$$\vec{u}_{curlnoise} \leftarrow \frac{\vec{u}_{curlnoise}}{||\vec{u}_{curlnoise}||_2} \min\{||\vec{u}_{curlnoise}||_2, ||\vec{\psi}||_2\}. \tag{12}$$

Vorticity Confinement: Our solver also supports vorticity confinement which can be used to boost the rotational component of the velocity field [Fedkiw et al. 2001]. The idea behind this technique is to restore some momentum lost during the non-divergent projection. It is also possible to provide a negative parameter to prevent the simulation from spinning as in candle simulations. Our implementation follows the one outlined in Bridson's book [Bridson 2015]. The results are visually appealing, but the velocity averaging required with MAC grids to properly compute the curl as a finite central difference over $2\Delta x$ makes this operation computationally expensive. Similarly to the curlnoise implementation, we clamp the magnitude of the vorticity vectors to prevent instability due to numerical errors.

3.2 Rendering

To use the highly optimized hardware trilinear interpolation, we copy the temperature grid to a CUDA 3D texture. This has the disadvantage of increasing the memory footprint, but at the time of this writing, atomic operations on textures are not allowed to be used under current GPU hardware which explains why we must maintain a temperature grid for simulation as well as a temperature texture for rendering [Gao et al. 2018].

Multiple Scattering as a Gaussian Convolution: We also create another CUDA 3D texture at $\frac{1}{4}$ the temperature grid resolution that stores the emission contribution as specified by the shader. This texture will then be iteratively convolved with a $3 \times 3 \times 3$ discreet Gaussian kernel to approximate multiple scattering (MS) of the emission [Premože et al. 2004]. This convolution was also implemented with $5 \times 5 \times 5$ and $7 \times 7 \times 7$ kernels to lower the number of iterations, but it was empirically demonstrated that more iterations of the $3 \times 3 \times 3$ kernel led to better performance.

Grid Raymarching: Our renderer starts by testing for intersections with the domain. This intersection testing procedure comes from [Prunier 2021] as it outperforms the implementation provided by Nvidia in the CUDA Samples. Our program does not use any kind of acceleration structure to speed up rendering. This brute force approach still works reasonably well in practice as rendering is rarely the bottleneck of execution. Each thread renders a single pixel and will only start raymarching if the ray intersects the domain. As raymarching is performed, both the temperature and the approximated multiple scattering are sampled and interpreted according to the shader definition. High-temperature values can be remapped by a vector-valued ramp to light emission colors. Single scattering is also a vector-valued parameter that allows the user to define the complement of the wavelength to be absorbed by the participating media [Wrenninge 2012]. As previously mentioned the temperature grid is used to drive both the density and emission components of the shader. The general structure of the ray marcher is inspired by the implementation provided by OpenVDB [Museth et al. 2021]. Only directional lights are supported by the renderer as those are highly efficient to render volume with and can be combined to create more elaborate lighting effects.

Writing the Framebuffer to Disk: Our program can render to multiple file formats such as PPM, PNG using LodePNG [Vandevenne 2013] and OpenEXR using Tiny OpenEXR [Fujita 2021]. None of those formats have external dependencies since everything is provided as header files in the third-party directory. OpenEXR is the default and recommended format as it is the standard in visual effects while also being the fastest option. The files are written to disk using RLE compression to keep the GPU waiting as little time as possible.

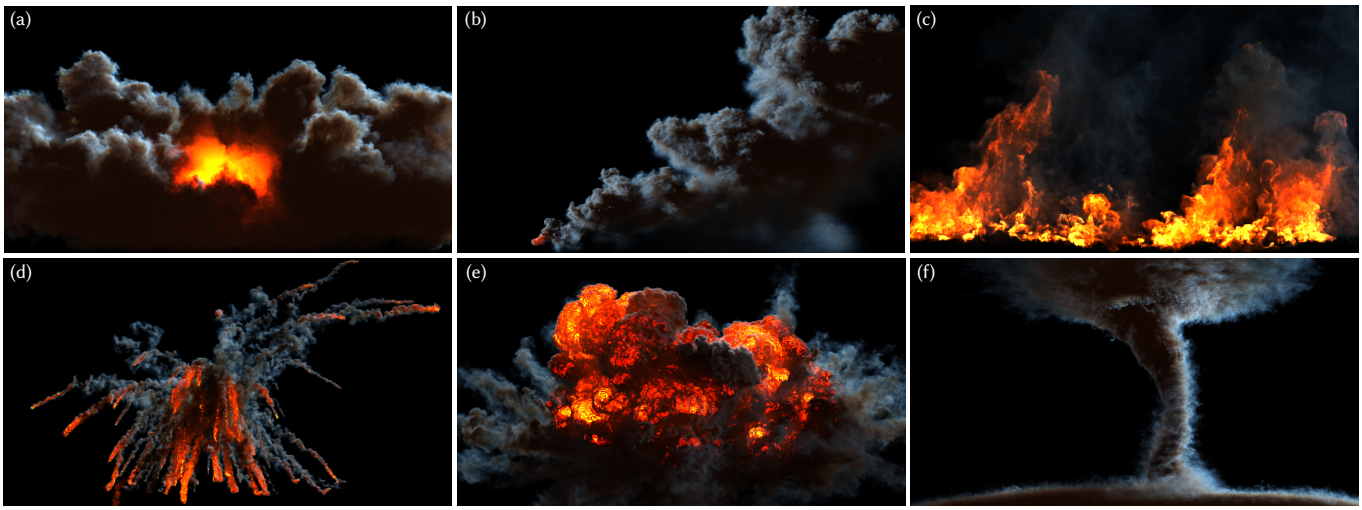

Fig. 7. Multiple simulations and renders generated with Blaze. (a) Rocket Launch, (b) Smoke Plume, (c) Ground Fire, (d) Fire Debris , (e) Explosion, (f) Tornado

3.3 Scene Description

The program expects a JSON file with a .blz extension that defines how the simulation and render will be performed. We use the RapidJSON C++ library [Yip 2021] to parse the scenes. Such scene files are provided in the scenes directory as part of the project on GitHub.

Custom Binary Particle Files: A particle file is a custom binary file format with a .part extension optimized to speed up the sourcing step of the solver. It consists of a series of 8 floats (32 Bytes) per particle following the memory layout shown in Fig. 8.

POS.X	POS.Y	POS.Z	VEL.X	VEL.Y	VEL.Z	PSCALE	TEMP

Fig. 8. A particle file stores each particle sequentially as 8 floats: 3 floats for the position, 3 floats for the velocity, 1 float for the pscale (2 times the radius) and 1 float for the temperature.

We initially tried to store the particles in ASCII JSON files, but our tests have shown this could be sped up by more than two orders of magnitude by using our particle format. This speed gain is especially critical as it is one of the two single-threaded CPU processes that runs while the GPU is waiting. The source particles can either be static (read once on the first time step) or animated (read on each time step).

Houdini Digital Asset: All scenes and particle files were generated using the provided Houdini Digital Asset (HDA). It is therefore possible to create novel scenes using it within Houdini Apprentice, the free version of the software.

4 RESULTS

Here are six scenes that were created using the provided HDA. The scenes as well as the Houdini file that was used to generate them are all provided along with the full source code on GitHub. All scenes were run using an Nvidia GeForce GTX 1080 Ti GPU with 11 GB of VRAM. See Table 1 for a detailed breakdown of the runtime performance achieved with our implementation.

4.1 Rocket Launch

In Fig. 7 (a), a small cluster of particles is injecting high temperature and velocity toward the ground. Velocity drag is used to contain the outward motion and give a sense of scale to the simulation. Some velocity particles are also scattered on the ground to simulate friction with the terrain hence contributing to the rolling motion of the smoke.

4.2 Smoke Plume

In Fig. 7 (b), a small source close to the ground is emitting temperature. The wind force is used to slowly move the plume across the grid. Our high-order advection scheme couple with the tricubic interpolation help at preserving the fine details. High-frequency curlnoise is added around the plume to create more fine details. A small amount of negative cooldown is also used to counter dissipation. This helps at keeping the smoke plume crisp and dense throughout the simulation.

4.3 Ground Fire

In Fig. 7 (c), patches of temperature particles are scattered on the ground emitting higher temperature in places where the point density is more important. Strong high-frequency turbulence is added to the simulation to generate more details around the fast-moving flames. A strong cooldown is also used to keep the fire close to the ground and make the smoke dissipate quickly. The wind plays a critical role in making those scenes convincing.

4.4 Fire Debris

In Fig. 7 (d), small sources are emitting temperature in multiple directions. Cooldown is used to make the fire trails disappear quickly. Particles with noisy velocity are sourced in the domain one frame ahead of the temperature particles to create more complex sourcing profiles through advection. The wind also plays an important role here by transporting the smoke trails as expected in an exterior environment.

Table 1. Average simulation and render time per frame in seconds at 1280×720

Scene	Domain Voxels #	Max Particles #	GPU Mem.	P2G	Turb.	Solver	Advect	MS	Render	Others	Total
Ground Fire	12 M	12,865	617 MB	0.00	0.02	0.24	0.08	0.00	0.11	0.06	0.51
Explosion	98 M	278,722	4,803 MB	0.00	0.20	3.27	0.59	0.01	0.38	0.14	4.59
Fire Debris	99 M	42,074	4,828 MB	0.03	0.17	3.56	0.49	0.01	0.07	0.13	4.47
Rocket Launch	80 M	228,325	3,924 MB	0.03	0.17	2.86	0.45	0.06	0.24	0.13	3.95
Tornado	87 M	302,402	4,236 MB	0.05	0.18	3.12	0.55	0.00	0.11	0.10	4.11
Smoke Plume	100 M	64,720	4,865 MB	0.03	0.20	3.38	0.58	0.01	0.17	0.13	4.49

4.5 Explosion

In Fig. 7 (e), the first frames of the simulation inject hundreds of thousands of particles into the simulation to precisely define the shape of the explosion. The amount of outward velocity sourced stays conservative to preserve the fine details scattered to the grid and prevent the creation of large pressure fronts.

4.6 Tornado

In Fig. 7 (f), choreographed particles are animated following the shape of a tornado while emitting temperature. The domain is also filled with additional velocity particles that represent the air pulled toward the tornado. Those air particles need to be very sparse in order to inject enough velocity so the tornado keeps its shape without preventing the fluid solver from properly moving the fluid according to the divergence-free velocity field.

5 CONCLUSIONS

Our framework tries to leverage the parallel nature of modern GPU hardware to accelerate the whole fluid simulation and rendering pipeline commonly present in the visual effects industry. Our implementation eliminates the bottlenecks related to memory transfers between host and device as large data structures required for fluid simulation only exist in GPU memory. The scene description is also designed to provide the smallest number of user-defined parameters while still preserving all the necessary flexibility to model many fluid phenomena. This approach leads to a very fast and portable system that minimizes the number of steps required to get from the idea to the final pixels.

REFERENCES

R. Allen. 2021. raytracinginoneweekendincuda. https://github.com/rogerallen/raytracinginoneweekendincuda. (2021).

B. Bitterli. 2021. incremental-fluids. https://github.com/tunabrain/incremental-fluids. (2021).

R. Bridson. 2015. *Fluid simulation for computer graphics.* CRC press.

R. Fedkiw, J. Stam, and H. W. Jensen. 2001. Visual simulation of smoke. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. 15–22.

S. Fujita. 2021. tinyexr. https://github.com/syoyo/tinyexr. (2021).

M. Gao, X. Wang, K. Wu, A. Pradhana, E. Sifakis, C. Yuksel, and C. Jiang. 2018. GPU optimization of material point methods. *ACM Transactions on Graphics (TOG)* 37, 6 (2018), 1–12.

G. A. A. Gomes. 2009. Linear solvers for stable fluids: GPU vs CPU. *17th EncontroPortugues de ComputacaoGrafica (EPCG09)* (2009), 145–153.

H. Lehtinen. 2021. cuda-noise. https://github.com/covexp/cuda-noise. (2021).

K. Museth, P. Cucka, M. Aldén, and D. Hill. 2021. OpenVDB. (2021). https://www.openvdb.org.

S. Premože, M. Ashikhmin, J. Tessendorf, R. Ramamoorthi, and S. Nayar. 2004. Practical rendering of multiple scattering effects in participating media. In *Proc. of Eurographics Symposium on Rendering*, Vol. 2. Citeseer, 363–374.

J.-C. Prunier. 2021. Scratch a Pixel 2.0. (2021). https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/ray-box-intersection.

I. Quilez. 2021. Inigo Quilez Personal Website. (2021). https://www.iquilezles.org/www/articles/functions/functions.htm.

A. Ralston. 1962. Runge-Kutta methods with minimum error bounds. *Mathematics of computation* 16, 80 (1962), 431–437.

J. Stam. 1999. Stable fluids. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*. 121–128.

J. Stam. 2003. Real-time fluid dynamics for games. In *Proceedings of the game developer conference*, Vol. 18. 25.

L. Vandevenne. 2013. lodepng. https://github.com/lvandeve/lodepng. (2013).

M. Wrenninge. 2012. *Production volume rendering: design and implementation.* CRC Press.

M. Yip. 2021. rapidjson. https://github.com/Tencent/rapidjson. (2021).